



Edição 138 :: R\$ 14,90

Conformação arquitetural
Sincronizando o código à arquitetura do projeto

Criando uma aplicação corporativa

Desenvolvendo a camada
de persistência com Hibernate

Construa ambientes de alta disponibilidade

Implementando aplicações tolerantes
a falhas com Apache Tomcat e Hazelcast

DEV MEDIA

REST X SOAP

Conheça as diferenças e vantagens



Twitter Finagle

Desenvolvendo sistemas
distribuídos de alta performance

Spring MVC e Bootstrap

Integrando tecnologias
por um design responsivo

ISSN 1676-836-1



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 138 • 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

06 – Criando uma aplicação corporativa em Java – Parte 2

[Bruno Rafael Sant'Ana]

20 – Web services REST versus SOAP

[Web services REST versus SOAP]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

30 – Twitter Finagle: Desenvolvendo sistemas distribuídos

[Leonardo Gonçalves da Silva]

42 – Spring MVC: Construa aplicações responsivas com Bootstrap – Parte 2

[Pablo Bruno de Moura Nóbrega e Marcos Vinícius Turisco Dória]

Conteúdo sobre Engenharia de Software

52 – Conformação arquitetural: sincronizando o código e a arquitetura do projeto

[Gabriel Novais Amorim]

Conteúdo sobre Novidades, Artigo no estilo Curso

61 – Construindo e otimizando um ambiente de alta disponibilidade – Parte 1

[Cleber Muramoto]



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 2

Como desenvolver a camada de persistência com o Hibernate

ESTE ARTIGO FAZ PARTE DE UM CURSO

Na primeira parte do artigo abordamos sobre a origem das especificações JSF, JPA e CDI e falamos um pouco sobre elas. Também deixamos todo o ambiente de desenvolvimento configurado, além de criarmos o projeto no Eclipse. A essa altura, nosso projeto já contém o arquivo *persistence.xml*, que é um dos principais requisitos para trabalharmos com JPA, porém isso é apenas o início quando se trata de persistência de dados. Para podermos armazenar os dados em alguma base de dados, temos que criar também as classes que representarão nossas entidades, e para isso usaremos todos os conceitos sobre mapeamento objeto/relacional apresentados no primeiro artigo da série.

Ainda falando sobre persistência, é importante planejarmos quais de nossas classes acessarão o banco de dados, pois não é uma boa prática ter código de acesso a dados espalhado por todo o sistema. Por isso, faremos uso do padrão Data Access Object, mais conhecido como DAO, que será composto por diversas classes e interfaces. Até poderíamos lidar com transações dentro dessas classes, o que é comum de se ver em muitas aplicações, mas optamos por uma solução mais elegante, que fará uso de um Interceptor para interceptar as chamadas aos métodos que precisem de transações.

Como o leitor pode perceber, construir a parte que lida com a persistência dos dados envolve muitas coisas e, portanto, temos bastante trabalho a ser feito.

Fique por dentro

Nesse artigo mostraremos como desenvolver toda a camada de persistência da aplicação criada para gerenciamento de bibliotecas. Isso inclui a implementação das entidades e do DAO, além de um esquema para gerenciamento de transações. Para tirar proveito dos recursos disponíveis na plataforma Java EE, faremos uso de um Interceptor – relacionado ao CDI – que irá gerir todas as transações. Também será mostrado como criar uma classe conversora (com a anotação @Converter) para converter o estado dos atributos LocalDate de uma entidade em algum tipo de dado adequado que possa ser gravado no banco de dados. Essa possibilidade de conversão que a JPA provê é muito útil na hora de persistirmos entidades com atributos tipados com as novas classes da API de data e hora do Java 8 como, por exemplo, a classe LocalDate. Dessa forma, este artigo é de grande importância para todos os leitores que buscam adquirir conhecimento sobre os recursos disponíveis dentro da plataforma Java EE.

Criando e mapeando as entidades

A partir de agora, vamos criar as entidades e seus mapeamentos. Vale ressaltar que essa é uma parte de grande importância, pois o Hibernate irá criar as tabelas no banco de dados de acordo com o código implementado em nossas entidades e como as mapeamos.

Em nosso código teremos onze entidades, a saber: **Autor**, **Categoria**, **Editora**, **Emprestimo**, **Endereco**, **FuncionarioBiblioteca**, **Leitor**, **Livro**, **Pessoa**, **Telefone** e **Usuario**. E como esperado, teremos uma tabela para cada entidade. No entanto, haverá uma tabela a mais que não terá uma entidade correspondente (de nome *livro_autor*), totalizando doze tabelas. Isso ocorre porque existe um relacionamento do tipo **ManyToMany** entre as entidades **Livro**

e **Autor**, o que leva o persistence provider a gerar uma tabela de ligação *livro_autor* para viabilizar esse relacionamento.

Para organizar o projeto, iremos colocar cada tipo de classe em um pacote diferente. Sendo assim, todas as entidades ficarão no pacote `br.com.javamagazine.entidades`.

Listagem 1. Código da entidade Autor.

```
package br.com.javamagazine.entidades;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Autor {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String nome;

    //getters e setters
}
```

Na **Listagem 1** temos o código da entidade **Autor**, que nada mais é do que uma classe Java comum com algumas anotações. A anotação `@Entity` que vai em cima da classe, por exemplo, indica que ela é uma entidade. Como não estamos usando a anotação `@Table` para especificar o nome da tabela, seu nome será derivado do nome da entidade, que por padrão é o nome não qualificado da classe. Portanto, será criada uma tabela chamada *autor* para essa entidade. Já a anotação `@Id` serve para demarcar a chave primária. Neste caso, teremos uma coluna chamada *id* para representar essa chave. A anotação `@GeneratedValue` indica que o valor da chave primária deve ser gerado automaticamente, e de acordo com a estratégia (`strategy`) passada dentro da anotação, o persistence provider irá se orientar para viabilizar isso criando, por exemplo, uma coluna do tipo `auto-increment/identity` ou uma `sequence` no banco de dados. O persistence provider irá utilizar a estratégia indicada se for suportada pelo banco de dados.

A JPA define quatro tipos de estratégia, a saber:

- `@GeneratedValue(strategy=GenerationType.AUTO)`: indica que o persistence provider deve escolher a melhor estratégia para geração dos valores da PK de acordo com o banco de dados. É a opção com maior portabilidade. Quando usamos a anotação sem definirmos explicitamente uma estratégia, essa é a opção `default`;
- `@GeneratedValue(strategy=GenerationType.IDENTITY)`: especifica o uso de uma coluna `auto-increment/identity` como estratégia para geração dos valores da chave primária;
- `@GeneratedValue(strategy=GenerationType.SEQUENCE)`: especifica o uso de uma sequence para geração dos valores da chave primária;
- `@GeneratedValue(strategy=GenerationType.TABLE)`: especifica que o persistence provider deve atribuir valores às chaves

primárias de nossas entidades com base em uma tabela auxiliar que irá garantir que os valores das chaves sejam únicos.

Para a tabela *autor* escolhemos a estratégia **IDENTITY** para geração automática dos valores da chave primária. Faremos uso dessa estratégia porque ela é uma das mais simples e também porque é muito comum ser usada uma coluna auto-increment/`identity` para representar a chave primária, mesmo nos casos onde o banco não é gerado através de um framework ORM. Além disso, o tipo de banco de dados que iremos utilizar, o MySQL, suporta essa estratégia, ao contrário de alguns outros.

As **Listagens 2, 3 e 4** mostram o código das entidades **Categoria**, **Endereco** e **Telefone**. Como pode ser verificado, esse conteúdo é similar ao código da classe **Autor**, já analisada; inclusive usam as mesmas anotações.

Listagem 2. Código da entidade Categoria.

```
package br.com.javamagazine.entidades;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Categoria {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String nome;

    //getters e setters
}
```

Listagem 3. Código da entidade Endereco.

```
package br.com.javamagazine.entidades;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Endereco {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String endereco;
    private String complemento;
    private String bairro;
    private String estado;
    private String cidade;
    private String cep;

    //getters e setters
}
```

Criando uma aplicação corporativa em Java – Parte 2

Listagem 4. Código da entidade Telefone.

```
package br.com.javamagazine.entidades;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Telefone {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String numero;
    private boolean fixo;

    //getters e setters
}
```

Já na **Listagem 5** temos o código da entidade **Editora**. Conforme analisado, a anotação **@OneToMany** especifica um relacionamento do tipo um para muitos entre uma editora e seus telefones. Assim, uma editora poderá ter muitos telefones. Por sua vez, a anotação **@JoinColumn**, colocada sobre a lista de telefones, define qual coluna da tabela *telefone* será a chave estrangeira (nesse caso a coluna *id_editora*). Logo após, a anotação **@OneToOne** determina um relacionamento do tipo um para um entre editora e endereço, ou seja, uma editora poderá ter apenas um endereço e um endereço poderá pertencer somente a uma editora.

A anotação **@JoinColumn**, colocada em cima do **endereco**, indica que a coluna *id_endereco* da tabela *editora* será chave estrangeira. Note que tanto na anotação **@OneToMany** como na anotação **@OneToOne** foi especificado um atributo **cascade** e é importante entender para que ele serve. Esse atributo foi usado para propagar o efeito das operações executadas através de um **entityManager** sobre instâncias de **Editora** nas entidades relacionadas, que seriam **Telefone** e **Endereco**. Por exemplo, se persistirmos uma editora no banco de dados através do método **persist()**, seus telefones e seu endereço também serão persistidos, conforme mostrado na **Listagem 6**.

Uma vez entendida a entidade **Editora**, vejamos como funciona a entidade **Emprestimo** (**Listagem 7**). Em nosso caso, um empréstimo de um livro é realizado por um funcionário da biblioteca para um leitor. Dessa forma, fica evidente que o ato de realizar o empréstimo tem relação direta com o livro que está sendo emprestado, e também com o funcionário da biblioteca e com o leitor, pois ambos estão participando desse ato. Logo, quando vamos transpor tudo isso para o mundo OOP, precisamos relacionar a entidade **Emprestimo** com as entidades **Livro**, **Leitor** e **FuncionarioBiblioteca**. Portanto, teremos três relacionamentos do tipo **ManyToOne** para expressar essas relações, já que um único livro pode ser emprestado várias vezes, um mesmo leitor pode tomar emprestado vários livros (para cada livro um empréstimo diferente, no caso do nosso sistema) e um mesmo funcionário da biblioteca pode realizar vários empréstimos.

Listagem 5. Código da entidade Editora.

```
package br.com.javamagazine.entidades;

import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;

@Entity
public class Editora {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String nome;
    @OneToMany(cascade={CascadeType.ALL})
    @JoinColumn(name="id_editora")
    private List<Telefone> telefones = new ArrayList<Telefone>();
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="id_endereco")
    private Endereco endereco;
    private String site;
    private String email;

    //getters e setters
}
```

Listagem 6. Código relacionado ao uso do cascade.

```
//persistindo editora e entidades relacionadas por causa do cascade
Editora editora = new Editora();
editora.getTelefones().add(new Telefone());
editora.setEndereco(new Endereco());
//persiste tudo com uma única invocação do método persist()
entityManager.persist(editora);
```

A classe **java.time.LocalDate** do Java 8 foi usada para definirmos o tipo da data do empréstimo, da data prevista para devolução e da data efetiva da devolução. Quando temos em nossa classe atributos do tipo **java.util.Date** ou **java.util.Calendar**, podemos usar a anotação **@Temporal** da JPA para realizarmos o mapeamento, mas como estamos usando o tipo **java.time.LocalDate**, teremos que criar uma classe que atuará como um conversor (mais detalhes serão explicados quando for mostrado o código dessa classe). Isso é necessário porque a anotação **@Temporal**, por enquanto, não suporta o tipo **java.time.LocalDate**.

No código exibido na **Listagem 8**, referente à entidade **Livro**, a anotação **@ManyToOne** da variável **editora** indica um relacionamento do tipo muitos para um, pois podemos ter muitos livros publicados por uma única editora. Logo após, a anotação **@ManyToMany** define um relacionamento do tipo muitos para muitos, pois muitos livros podem pertencer a um mesmo autor, e um livro pode ser escrito por mais de um autor. Em conjunto com a anotação **@ManyToMany** usamos a anotação **@JoinTable** para configurar o nome da tabela de ligação – nesse caso chamada *livro_autor* – que irá relacionar os livros e os autores. Através dos

atributos **joinColumns** e **inverseJoinColumns** configuramos as colunas da tabela de ligação – *id_livro* e *id_autor* – que serão as chaves estrangeiras. Para ficar mais claro, teremos a tabela *livro_autor* que possuirá as colunas *id_livro* e *id_autor* como chaves estrangeiras, enquanto as chaves primárias estão nas tabelas *livro* e *autor*, ou seja, *livro_autor* estabelece um vínculo entre as tabelas *livro* e *autor*.

Listagem 7. Código da entidade Emprestimo.

```
package br.com.javamagazine.entidades;

import java.time.LocalDate;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
public class Emprestimo {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private LocalDate dataEmprestimo;
    private LocalDate dataPrevista;
    private LocalDate dataDevolucao;
    @ManyToOne
    @JoinColumn(name = "id_livro")
    private Livro livro;
    @ManyToOne
    @JoinColumn(name = "id_leitor")
    private Leitor leitor;
    @ManyToOne
    @JoinColumn(name = "id_funcionarioBiblioteca")
    private FuncionarioBiblioteca funcionarioBiblioteca;

    //getters e setters
}
```

Com relação à anotação **@ManyToOne**, podemos encontrá-la em dois lugares dentro das **Listagens 7 e 8**. Ela foi usada em cima da variável **categoria**, presente na classe **Livro**, para indicar que muitos livros podem pertencer a uma mesma categoria. Essa anotação também foi usada para estabelecer um relacionamento bidirecional entre as entidades **Livro** e **Emprestimo**, e nesse caso o relacionamento é definido por meio da própria anotação **@ManyToOne** utilizada na classe **Emprestimo** e da anotação **@OneToMany** utilizada na classe **Livro**. Dentro da anotação **@OneToMany**, que aparece em cima da variável **emprestimos** na classe **Livro**, temos o atributo **mappedBy** para indicar que a entidade **Livro** é o lado inverso do relacionamento, enquanto a entidade **Emprestimo** é a dona do relacionamento.

A entidade **Usuario**, por sua vez, pode ser encontrada na **Listagem 9**. Com as explicações que foram dadas nas listagens anteriores é possível entender o papel de cada anotação utilizada nesse código, com exceção da anotação **@Column**, ainda não abordada. Utilizamos **@Column** com o atributo **unique** para

assegurar que não existam dois nomes de usuários iguais. Dessa forma será criada uma constraint no banco que irá garantir que esse tipo de duplicidade não aconteça.

Listagem 8. Código da entidade Livro.

```
package br.com.javamagazine.entidades;

import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;

@Entity
public class Livro {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String nome;
    @ManyToOne
    @JoinColumn(name = "id_editora")
    private Editora editora;
    @ManyToMany
    @JoinTable(name = "livro_autor",
        joinColumns = {@JoinColumn(name = "id_livro")},
        inverseJoinColumns = {@JoinColumn(name = "id_autor")})
    private List<Autor> autores;
    @ManyToOne
    @JoinColumn(name = "id_categoria")
    private Categoria categoria;
    private String isbn;
    @OneToMany(mappedBy = "livro")
    private List<Emprestimo> emprestimos;

    //getters e setters
}
```

Listagem 9. Código da entidade Usuario.

```
package br.com.javamagazine.entidades;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Usuario {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    @Column(unique = true)
    private String usuario;
    private String senha;
    private boolean admin = true;
    @OneToOne(mappedBy = "usuario")
    private FuncionarioBiblioteca funcionarioBiblioteca;

    //getters e setters
}
```

Criando uma aplicação corporativa em Java – Parte 2

Por último, analisemos as três entidades restantes: **Pessoa**, **FuncionarioBiblioteca** e **Leitor**, cujos códigos são apresentados nas **Listagens 10, 11 e 12**, respectivamente.

Ao analisar esse código, observe que as classes **Funcionario-Biblioteca** e **Leitor** são subclasses de **Pessoa**, o que sinaliza um relacionamento por meio de herança. Assim como ocorreu em nosso exemplo, o uso da herança é algo comum ao desenvolvimento de aplicações Java, e foi pensando nisso que a JPA incorporou mecanismos para mapearmos nossas hierarquias de classes – que nesse caso são entidades.

Listagem 10. Código da entidade Pessoa.

```
package br.com.javamagazine.entidades;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Pessoa {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String nome;
    private String email;
    private String sexo;
    private String cpf;

    //getters e setters
}
```

Listagem 11. Código da entidade FuncionarioBiblioteca.

```
package br.com.javamagazine.entidades;

import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;

@Entity
public class FuncionarioBiblioteca extends Pessoa{

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="id_usuario")
    Usuario usuario;
    @OneToMany(cascade={CascadeType.ALL})
    @JoinColumn(name="id_funcionarioBiblioteca")
    private List<Telefone> telefones = new ArrayList<Telefone>();
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="id_endereco")
    private Endereco endereco;

    //getters e setters
}
```

Listagem 12. Código da entidade Leitor.

```
package br.com.javamagazine.entidades;

import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;

@Entity
public class Leitor extends Pessoa{

    private String turma;
    private String matricula;
    @OneToMany(cascade={CascadeType.ALL})
    @JoinColumn(name="id_leitor")
    private List<Telefone> telefones = new ArrayList<Telefone>();
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="id_endereco")
    private Endereco endereco;

    //getters e setters
}
```

Como podemos verificar, todas elas serão entidades e cada uma será representada por sua respectiva tabela no banco de dados. Para atingirmos esse resultado usaremos a anotação **@Inheritance** com a estratégia **JOINED**. Repare que definimos uma chave primária (usando a anotação **@Id**) apenas na entidade **Pessoa**, mas na realidade o JPA provider criará as três tabelas (**pessoa**, **funcionariobiblioteca** e **leitor**) com suas respectivas chaves primárias (nesse caso representadas pela coluna *id* de cada tabela). Os valores dessas chaves primárias serão utilizados para relacionar os registros da tabela *pessoa* com os registros da tabela *funcionariobiblioteca* ou os registros da tabela *pessoa* com os registros da tabela *leitor*.

Como no *persistence.xml* configuramos a propriedade **hibernate.hbm2ddl.auto** com o valor **create**, se rodarmos o nosso projeto no JBoss WildFly, todas as tabelas serão criadas automaticamente, pois nesse momento já temos todas as entidades prontas com seus respectivos mapeamentos.

O atributo **mappedBy**

Quando existe um relacionamento bidirecional, temos que definir um lado como dono do relacionamento (*owning side*) e o outro lado como inverso (*inverse side*). O lado inverso do relacionamento é o lado que fica com o atributo **mappedBy**. No momento em que persistimos as entidades envolvidas em uma relação bidirecional, a entidade que é dona do relacionamento é levada em consideração para que seja criada uma associação dentro do banco de dados através da chave estrangeira.

Para ficar mais claro, podemos tomar como exemplo as entidades **Emprestimo** e **Livro**. Como a entidade **Livro** não é a dona da relação, não é suficiente setarmos a lista de empréstimos no livro – **livro.setEmprestimos(listaEmprestimos)**; – antes de persistirmos os objetos. Dessa forma os relacionamentos entre o livro e os

emprestimos não seriam criados dentro da base de dados. Devido à entidade **Emprestimo** ser a dona do relacionamento, devemos associar cada objeto do tipo emprestimo com o livro que está sendo emprestado – **emprestimo.setLivro(livro)**.

Após isso, certamente se persistissemos os objetos, a coluna da chave estrangeira *id_livro*, presente na tabela *emprestimo*, seria preenchida corretamente, criando as associações dentro do banco. Isso porque apesar de em um relacionamento bidirecional termos que “amarrar” as duas pontas, o lado que é o dono do relacionamento é quem define o preenchimento da chave estrangeira na base de dados.

Criando a classe de conversão para LocalDate

Observe que a entidade **Emprestimo** (*Listagem 7*) possui três atributos do tipo **java.time.LocalDate**. Para cada atributo desse tipo, o persistence provider criará uma coluna correspondente do tipo *TINYBLOB* – isso no momento em que a tabela for gerada a partir da entidade e seus mapeamentos. O *TINYBLOB* é um dos quatro tipos de *BLOB* existentes no MySQL, e um *BLOB* (*Binary Large Object*) nada mais é do que um campo que serve para armazenar qualquer tipo de dado em formato binário. Vale ressaltar que a aplicação funcionaria normalmente, porém armazenar datas em colunas do tipo *TINYBLOB* definitivamente não é uma boa prática, visto que a performance pode ser prejudicada.

Por enquanto, a JPA não tem uma anotação que mapeie adequadamente atributos desse tipo (**LocalDate**). Para contornarmos esse problema/restrição, usaremos um recurso da JPA relacionado a conversões. Este consiste em criar uma classe anotada com **@Converter**, que implemente **AttributeConverter** e que sobrecreva os dois métodos dessa interface. O código dessa classe é mostrado na *Listagem 13*.

Listagem 13. Classe de conversão para usar com LocalDate.

```
package br.com.javamagazine.conversores;

import java.sql.Date;
import java.time.LocalDate;
import javax.persistence.AttributeConverter;
import javax.persistence.Converter;

@Converter(autoApply = true)
public class LocalDateConversor implements AttributeConverter<LocalDate, Date> {

    @Override
    public Date convertToDatabaseColumn(LocalDate dataDaEntidade) {
        return (dataDaEntidade == null) ? null : Date.valueOf(dataDaEntidade);
    }

    @Override
    public LocalDate convertToEntityAttribute(Date dataDoBancoDeDados) {
        return (dataDoBancoDeDados == null) ? null : dataDoBancoDeDados
            .toLocalDate();
    }
}
```

A anotação **@Converter** indica que a classe será responsável por realizar conversões. No nosso caso, irá converter de **java.time.LocalDate** para **java.sql.Date** na hora de gravar as datas na base de dados e fazer o oposto na hora de recuperar essas datas do banco e atribuí-las aos atributos da entidade **Emprestimo**. Os métodos que sobrecrevemos, **convertToDatabaseColumn()** e **convertToEntityAttribute()**, realizarão todo esse trabalho. Por fim, é interessante notar que dentro da anotação **@Converter** setamos o atributo **autoApply** como **true**, o que possibilita que essas conversões sejam feitas de forma automática para qualquer atributo do tipo **java.time.LocalDate** que esteja presente em qualquer entidade.

Após a criação dessa classe conversora é recomendado que você execute novamente o projeto no JBoss para que a tabela *emprestimo* seja recriada. Dessa vez, as colunas que representam as datas serão criadas com sendo do tipo *DATE*, em vez do tipo *TINYBLOB*.

Populando o banco de dados

Como mencionado na primeira parte do artigo, no GitHub foram disponibilizadas duas versões da aplicação para gerenciamento de bibliotecas. A versão completa contém todas as classes que fazem parte do DAO, e também os managed beans e telas para cadastro das categorias, autores, editoras, livros, leitores, funcionários da biblioteca e usuários. A versão simplificada – que é igual a apresentada neste artigo – não contém todas as classes DAO, managed beans e telas de cadastro, visando evitar listagens muito parecidas devido à similaridade entre os códigos. Desse modo conseguimos focar no que é mais importante e explicar detalhadamente cada código apresentado.

Nessa versão reduzida teremos apenas uma tela de cadastro, onde poderemos inserir funcionários da biblioteca e usuários. Mas se não temos as demais telas, como poderemos cadastrar os outros itens? Resolvemos essa questão de forma simples: cadastrando tudo diretamente no banco de dados através de instruções SQL. Assim, o leitor poderá criar o projeto de acordo com o que é ensinado no artigo e executá-lo no servidor de aplicação.

Dito isso, após subir o servidor de aplicação e as tabelas serem criadas, os comandos SQL da *Listagem 14* devem ser executados via “MySQL Command-Line Tool”.

Como não queremos que as tabelas sejam recriadas na próxima execução do JBoss (pois perderíamos os dados que acabamos de inserir via SQL), devemos alterar a propriedade *hibernate.hbm2ddl.auto* no *persistence.xml* para o valor **validate**. Feito isso, clique com o botão direito no JBoss, localizado na aba *Servers* do Eclipse, e escolha a opção *Clean....* Assim, na próxima inicialização do servidor de aplicação o Hibernate irá realizar apenas as validações necessárias.

Criando o Interceptor e as classes DAO

Durante o desenvolvimento das entidades e das demais classes do projeto, utilizaremos a injeção de dependências através do Weld, que é a implementação de referência da especificação CDI. Para habilitarmos esse recurso com CDI e assim aproveitarmos

Criando uma aplicação corporativa em Java – Parte 2

todas as vantagens advindas do mesmo, basta criarmos um arquivo vazio chamado *beans.xml* na pasta *WEB-INF*.

Listagem 14. Instruções SQL para popular a base de dados.

```
INSERT INTO endereco(id, bairro, cep, cidade, complemento, endereco, estado)
VALUES(1, 'Pq Sao Rafael', '08311080', 'Sao Paulo', '', 'Avenida Baronesa de Muritiba,
num 51', 'SP');
INSERT INTO endereco(id, bairro, cep, cidade, complemento, endereco, estado)
VALUES(2, 'Santa Terezinha', '02460000', 'Sao Paulo', '', 'Rua Luis Antonio dos Santos,
num 110', 'SP');
INSERT INTO pessoa(id, cpf, email, nome, sexo) VALUES(1, '34087772486', 'iris@meudominio.com', 'Iris', 'F');
INSERT INTO leitor(matricula, turma, id, id_endereco) VALUES('35012', 'TADS01', 1, 1);
INSERT INTO autor(id, nome) VALUES(1, 'Ricardo R. Lecheta');
INSERT INTO categoria(id, nome) VALUES(1, 'Informatica');
INSERT INTO editora(id, email, nome, site, id_endereco) VALUES(1, 'novatec@novatec.com.br', 'Novatec', 'www.novatec.com.br', 2);
INSERT INTO livro(id, isbn, nome, id_categoria, id_editora) VALUES(1, '9788575222928', 'Google Android para Tablets', 1, 1);
INSERT INTO livro(id, isbn, nome, id_categoria, id_editora) VALUES(2, '9788575224014', 'Desenvolvendo para iPhone e iPad', 1, 1);
INSERT INTO livro_autor(id_livro, id_autor) VALUES(1, 1);
INSERT INTO telefone(id, fixo, numero, id_leitor) VALUES(1, b'099999-9999', 1);
INSERT INTO telefone(id, fixo, numero, id_leitor) VALUES(2, b'15555-5555', 1);
INSERT INTO telefone(id, fixo, numero, id_editora) VALUES(3, b'098888-8888', 1);
INSERT INTO telefone(id, fixo, numero, id_editora) VALUES(4, b'17777-7777', 1);
```

Uma coisa comum de se encontrar em classes DAO são códigos para gerenciamento de transações (ver **BOX 1**). No entanto, se mantivermos código que lida com transações em cada classe que faz parte do DAO, acabaremos com uma grande quantidade de código de infraestrutura duplicado. Para evitarmos essa situação, antes de começarmos a criar nossas classes DAO, vamos criar um mecanismo para gerenciar as transações de nosso sistema e com isso evitar que esse tipo de código fique espalhado em nosso DAO ou por toda a aplicação. A solução que iremos adotar para isso faz uso de um Interceptor e um Interceptor binding (assuntos abordados na especificação de Interceptors e na especificação de CDI).

BOX 1. Transações

Uma transação pode ser definida como uma série de ações subsequentes que devem ser concluídas com sucesso. Caso alguma dessas ações não seja concluída com sucesso, as anteriores devem ser desfeitas, voltando o estado do sistema ao que era antes da transação iniciar. Uma transação deve terminar ou com um commit ou com um rollback.

O Interceptor será de suma importância para chegarmos à solução do problema relacionado a código duplicado, então vejamos onde ele se encaixa. Cada método que precisar de uma transação será marcado com uma anotação específica (posteriormente veremos mais detalhes) e quando for feita uma chamada a um desses métodos, essa chamada será interceptada por um Interceptor que cuidará do gerenciamento da transação.

Assim como o gerenciamento de transações, é comum encontrarmos outras tarefas que devem ser realizadas em diversas partes do sistema, como geração de logs, controle de autorização/

segurança, entre outras. Dizemos que esse tipo de tarefa é uma preocupação transversal (*crosscutting concern*), pois atravessa o sistema todo, ou boa parte dele. Para lidarmos com isso, a programação orientada a aspectos (ou AOP) sugere que encapsulemos essa preocupação transversal, e uma forma de fazer isso é por meio de um Interceptor. Portanto, ao tratarem de Interceptors, as especificações Java trazem consigo um pouco da AOP para dentro da Java EE.

Listagem 15. Interceptor que irá gerenciar as transações.

```
package br.com.javamagazine.interceptadores;

import java.io.Serializable;
import javax.inject.Inject;
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;
import javax.persistence.EntityManager;
import br.com.javamagazine.util.MensagemUtil;

@Interceptor @Transacional
public class TransacionalInterceptor implements Serializable {

    private static final long serialVersionUID = 1L;
    @Inject
    private EntityManager entityManager;

    @AroundInvoke
    public Object intercept(InvocationContext context){
        Object resultado = null;

        try {
            entityManager.getTransaction().begin();
            resultado = context.proceed();
            entityManager.getTransaction().commit();
        } catch (Exception e) {
            entityManager.getTransaction().rollback();
            MensagemUtil.addMensagemDeErro("Erro - ",
                "Detalhes do erro:" + e.getClass().getName() + " - " + e.getMessage());
            e.printStackTrace();
        }

        return resultado;
    }
}
```

A **Listagem 15** mostra o código do Interceptor que irá gerenciar as transações da aplicação. Veja que é uma classe comum que é anotada com **@Interceptor**. Dentro do código temos um método chamado **intercept()** que recebe a anotação **@AroundInvoke** e por conta disso será responsável por interceptar as chamadas aos nossos métodos de negócio que precisarem de uma transação. Dentro desse método usamos o **entityManager** que foi injetado via CDI para iniciarmos uma transação. Logo em seguida invocamos o método **proceed()** do contexto para indicar que nesse momento queremos que seja executada a lógica contida no método que

estamos interceptando, e por último damos `commit()` na transação. Caso ocorra alguma exceção é executado o `rollback()` e adicionada uma mensagem que será mostrada ao usuário.

Listagem 16. Interceptor binding para associar o Interceptor com os beans.

```
package br.com.javamagazine.interceptadores;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.interceptor.InterceptorBinding;

@InterceptorBinding
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Transacional { }
```

Agora que criamos o Interceptor, temos que achar um meio de dizer para ele quais métodos precisam de uma transação e devem ser interceptados. Fazemos isso por meio de um **Interceptor binding**. Sendo assim, criaremos um conforme a **Listagem 16**. Para isso, basta definir uma anotação chamada **Transacional** e marcá-la com **@InterceptorBinding**. E para estabelecermos um vínculo entre nosso Interceptor e o Interceptor binding, usamos a anotação **@Transacional** em cima da classe **TransacionalInterceptor** (vide **Listagem 15**). A partir de agora, sempre que um método precisar ser executado dentro de uma transação, é necessário apenas anotá-lo com **@Transacional**.

Listagem 17. Conteúdo do arquivo beans.xml.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">

    <interceptors>
        <class>br.com.javamagazine.interceptadores.TransacionalInterceptor</class>
    </interceptors>
</beans>
```

Para que nosso interceptor funcione, precisamos registrá-lo no arquivo `beans.xml` criado anteriormente, dentro de `WEB-INF`. A **Listagem 17** mostra como deve ficar nosso `beans.xml`.

Ao observarmos novamente a **Listagem 15**, que apresenta o código do Interceptor **TransacionalInterceptor**, podemos notar que um Entity Manager é injetado através da anotação **@Inject**. Pelo fato do processo de criação de um Entity Manager ser um pouco mais complexo do que o normal (não basta usar o `new()` para criar a instância, é necessário instanciar a classe a partir de uma **EntityManagerFactory**), precisamos “ensinar” ao CDI provider como criar um Entity Manager. Assim, sempre que um

Entity Manager for requerido, o CDI será capaz de produzir um e injetá-lo no injection point apropriado.

A **Listagem 18** mostra a classe que ficará encarregada de produzir os Entity Managers. A anotação **@Produces** indica o produtor de Entity Manager e a anotação **@RequestScoped** indica que deve ser produzido um Entity Manager por requisição. No final de cada requisição, o Entity Manager que foi criado é fechado através do método `fecharEntityManager()`, que recebe o Entity Manager via parâmetro. No entanto, para que isso funcione é necessário usar a anotação **@Disposes** conforme mostrado na listagem.

Listagem 18. Fábrica de Entity Manager.

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

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.context.RequestScoped;
import javax.enterprise.inject.Disposes;
import javax.enterprise.inject.Produces;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

@ApplicationScoped
public class FabricaDeEntityManager {

    private static EntityManagerFactory factory = Persistence.createEntityManagerFactory("bibliotecaPersistence");
    @Produces @RequestScoped
    public EntityManager criarEntityManager() {
        return factory.createEntityManager();
    }

    public void fecharEntityManager(@Disposes EntityManager manager) {
        manager.close();
    }
}
```

A classe **TransacionalInterceptor** também faz uso do método **addMensagemDeErro()**, presente na classe **MensagemUtil**. O código dessa classe é apresentado na **Listagem 19**. Como pode ser verificado, **MensagemUtil** é bastante simples e não requer muitas explicações. A classe disponibiliza alguns métodos públicos que delegam todo o trabalho para o método privado **addMensagem()** que, por sua vez, usa um método de **FacesContext** para adicionar uma mensagem que será exibida ao usuário.

Concluída essa etapa, nosso próximo passo é criarmos as classes DAO e para cada uma delas criaremos também uma interface. Dessa forma, na hora de injetar qualquer DAO que precisarmos, poderemos usar uma variável do tipo da interface com a anotação **@Inject** em cima dela e assim diminuímos o acoplamento, pois não estamos declarando a implementação concreta que deve ser usada. Além disso, quando definimos uma interface podemos ter mais de uma implementação para ela. No caso de um DAO, podemos ter, por exemplo, uma implementação que use JPA e outra que use JDBC, e através de algum recurso do CDI podemos escolher qual implementação deve ser injetada (usando Qualifiers, Alternatives ou a anotação **@Vetoed** conseguimos configurar isso).

Criando uma aplicação corporativa em Java – Parte 2

Listagem 19. Código da classe MensagemUtil, que irá adicionar mensagens para o usuário visualizar.

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

import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;

public class MensagemUtil {

    public static void addMensagemDeErro(String mensagemDeErro, String detalhes){
        addMensagem(FacesMessage.SEVERITY_ERROR, mensagemDeErro, detalhes);
    }

    public static void addMensagemDeAviso(String mensagemDeAviso,
                                         String detalhes){
        addMensagem(FacesMessage.SEVERITY_WARN, mensagemDeAviso, detalhes);
    }

    public static void addMensagemInformativa(String mensagemInformativa,
                                              String detalhes){
        addMensagem(FacesMessage.SEVERITY_INFO, mensagemInformativa, detalhes);
    }

    private static void addMensagem(FacesMessage.Severity severidade,
                                    String mensagem, String detalhes){
        FacesMessage msg = new FacesMessage(severidade, mensagem, detalhes);
        FacesContext.getCurrentInstance().addMessage(null, msg);
    }
}
```

Para simplificar o artigo, os códigos das interfaces serão omitidos, mas de qualquer forma é fácil saber quais são os métodos declarados em cada interface: basta observar na implementação concreta quais métodos foram sobrescritos (os que estão marcados com a anotação `@Override`).

Para que possamos reutilizar código, usaremos um DAO genérico, conforme demonstra a **Listagem 20**. É comum que desenvolvedores criem primeiro o DAO genérico e posteriormente façam as demais classes DAO herdarem dele. Contudo, essa prática traz alguns problemas e por isso iremos adotar uma estratégia diferente, dando lugar à composição no lugar da herança, ou seja, nossas classes terão um DAO genérico em vez de serem um. Como nossas classes terão um DAO genérico, elas irão delegar as tarefas para ele. Com essa estratégia diminuímos o acoplamento. Além disso, não herdaremos os comportamentos do DAO genérico e com isso poderemos escolher quais métodos implementar em cada classe. Assim, em cada implementação de método, na maioria das vezes iremos apenas invocar o método correspondente da classe `GenericoDaoImpl` (por exemplo: o método `inserir()` da classe `EmprestimoDaoImpl` contém apenas uma chamada ao método `inserir()` da classe `GenericoDaoImpl`).

Vamos imaginar uma situação hipotética apenas para ficar mais claro o problema de herdar um comportamento nesse caso. Imagine que precisamos gravar no banco de dados alguns logs de atividades realizadas dentro do sistema e para isso poderíamos ter uma classe `LogDao` que herdaria de `GenericoDao` e ganharia de bandeja todos os comportamentos do DAO genérico, inclusive o comportamento do método `remover()`. Agora, imagine também

que um dos requisitos do sistema é que esses logs não sejam deletados da base de dados. Nessa situação poderíamos até sobrecrever o método `remover()` e lançar uma `Exception` caso alguém tentasse usá-lo, mas se os logs não devem ser apagados, não faz sentido termos um método `remover()` na classe `LogDao`. Usando composição ao invés de herança não teremos esse problema.

Listagem 20. Implementação do DAO genérico.

```
package br.com.javamagazine.daoimpl;

import java.util.List;
import javax.persistence.EntityManager;
import br.com.javamagazine.dao.GenericoDao;

public class GenericoDaoImpl<T, K> implements GenericoDao<T, K> {

    private Class<T> classeDaEntidade;
    private EntityManager entityManager;

    public GenericoDaoImpl(Class classeDaEntidade, EntityManager entityManager) {
        this.classeDaEntidade = classeDaEntidade;
        this.entityManager = entityManager;
    }

    @Override
    public void inserir(T entidade) {
        entityManager.persist(entidade);
    }

    @Override
    public void remover(T entidade) {
        entityManager.remove(entityManager.merge(entidade));
    }

    @Override
    public void atualizar(T entidade) {
        entityManager.merge(entidade);
    }

    @Override
    public T pesquisarPorID(K id) {
        return entityManager.find(classeDaEntidade, id);
    }

    @Override
    public List<T> listarTodos(){
        return entityManager.createQuery("select t from " + classeDaEntidade.getSimpleName() + " t").getResultList();
    }
}
```

A **Listagem 21** exibe o código da classe `EmprestimoDaoImpl`, que implementa a interface `EmprestimoDao`, ou seja, é a implementação concreta que irá cuidar da persistência dos objetos que representam os empréstimos. Nessa classe, usamos a injecção de dependência via construtor. Como colocamos a anotação `@Inject` no construtor, o CDI provider irá instanciar a classe `EmprestimoDaoImpl` quando necessário fornecendo um Entity Manager como argumento, e esse Entity Manager será passado ao DAO genérico para que possa utilizá-lo. Lembre-se que os Entity Managers injetados serão produzidos pela fábrica que criamos

anteriormente que usa `@Produces`. Outro ponto que deve ser notado é que os métodos do DAO de empréstimo apenas delegam o trabalho para os métodos do DAO genérico, promovendo a reutilização de código.

Listagem 21. Código da classe `EmprestimoDaoImpl`, que compõe o DAO.

```
package br.com.javamagazine.daoimpl;

import java.io.Serializable;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import br.com.javamagazine.dao.EmprestimoDao;
import br.com.javamagazine.dao.GenericoDao;
import br.com.javamagazine.entidades.Emprestimo;

public class EmprestimoDaoImpl implements EmprestimoDao, Serializable{

    private static final long serialVersionUID = 1L;
    private GenericoDao<Emprestimo, Integer> genericoDao;

    @Inject
    EmprestimoDaoImpl(EntityManager entityManager){
        genericoDao = new GenericoDaoImpl<Emprestimo, Integer>(
            Emprestimo.class, entityManager);
    }

    @Override
    public void inserir(Emprestimo emprestimo) {
        genericoDao.inserir(emprestimo);
    }

    @Override
    public void atualizar(Emprestimo emprestimo) {
        genericoDao.atualizar(emprestimo);
    }
}
```

O código mostrado na **Listagem 22** se refere à implementação da classe `LeitorDaoImpl`. Como essa implementação é muito semelhante à da classe `EmprestimoDaoImpl` vista anteriormente, não detalharemos o seu conteúdo.

A **Listagem 23** apresenta o código da classe `FuncionarioBibliotecaDaoImpl`, que também é muito semelhante ao das classes DAO vistas nas listagens anteriores. A maioria dos métodos dessa classe apenas invocam algum outro método no DAO genérico, com exceção do método `existeFuncionarioCadastrado()`, que contém uma lógica que verifica se existe algum funcionário cadastrado no banco de dados; em caso afirmativo retorna `true`, caso contrário retorna `false`. Mais adiante veremos o código da classe `FuncionarioEUsuarioVerificadorBean`, onde esse método será usado.

Já na **Listagem 24** temos o código da classe `UsuarioDaoImpl`. Nessa classe temos três métodos: `listarUsuarios()`, `existeUsuarioCadastrado()` e `pesquisarUsuario()`. O método `listarUsuarios()` é responsável por retornar uma lista de objetos do tipo `Usuario`; lista que é preenchida com o conteúdo presente na tabela `usuario` do banco de dados. O método `existeUsuarioCadastrado()` verifica se existe algum usuário cadastrado na base de dados, retornando `true` se já existir algum ou `false` se não existir ao menos um. Esse método será usado pela classe `FuncionarioEUsuarioVerifica-`

Listagem 22. Código da classe `LeitorDaoImpl`, que compõe o DAO.

```
package br.com.javamagazine.daoimpl;

import java.io.Serializable;
import java.util.List;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import br.com.javamagazine.dao.GenericoDao;
import br.com.javamagazine.dao.LeitorDao;
import br.com.javamagazine.entidades.Leitor;

public class LeitorDaoImpl implements LeitorDao, Serializable {

    private static final long serialVersionUID = 1L;
    private GenericoDao<Leitor, Integer> genericoDao;

    @Inject
    LeitorDaoImpl(EntityManager entityManager){
        genericoDao = new GenericoDaoImpl<Leitor, Integer>(
            Leitor.class, entityManager);
    }

    @Override
    public Leitor pesquisarPorId(Integer id) {
        return genericoDao.pesquisarPorId(id);
    }

    @Override
    public List<Leitor> listarLeitores() {
        return genericoDao.listarTodos();
    }

    //restante dos métodos omitidos
}
```

`dorBean`, como veremos adiante. Por último temos o método `pesquisarUsuario()`, que será usado para validar se o usuário e senha digitados na tela de login são válidos. A lógica empregada é bastante simples: é passada uma `String` JPQL para o método `createQuery()` do Entity Manager, e este método retorna um objeto do tipo `Query`. Com esse objeto em mãos, o utilizaremos para que seja feita uma consulta ao banco de dados em busca de um registro que seja compatível com o usuário e senha digitados na tela de login. Se algum registro for encontrado será retornado um objeto do tipo `Usuario` preenchido com os dados desse registro, caso contrário será retornado `null`.

Para encerrarmos a implementação do DAO, falta apenas a classe `LivroDaoImpl`, apresentada na **Listagem 25**. Nessa listagem temos quatro métodos: `pesquisarPorId()`, `listarLivros()`, `listarLivrosDisponiveisParaEmprestimo()` e `listarLivrosEmprestados()`. O método `pesquisarPorId()` irá delegar para o DAO genérico a tarefa de pesquisar um livro pelo id e o método `listarLivros()` irá delegar para o DAO genérico a tarefa de recuperar uma lista com todos os livros. Os outros dois métodos estão relacionados ao empréstimo e devolução de livros. Neste momento, lembre-se que na tela de empréstimos só deverão ser exibidos os livros disponíveis para empréstimo e na tela de devolução, deverão aparecer apenas os livros que estão emprestados. E estas são, respectivamente, as funções destes dois métodos.

Criando uma aplicação corporativa em Java – Parte 2

Listagem 23. Código da classe FuncionarioBibliotecaDaoImpl, que compõe o DAO.

```
package br.com.javamagazine.daoimpl;

import java.io.Serializable;
import java.util.List;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import br.com.javamagazine.dao.GenericoDao;
import br.com.javamagazine.dao.FuncionarioBibliotecaDao;
import br.com.javamagazine.entidades.FuncionarioBiblioteca;

public class FuncionarioBibliotecaDaoImpl implements FuncionarioBibliotecaDao, Serializable {

    private static final long serialVersionUID = 1L;
    private GenericoDao<FuncionarioBiblioteca, Integer> genericoDao;

    @Inject
    FuncionarioBibliotecaDaoImpl(EntityManager entityManager){
        genericoDao = new GenericoDaoImpl<FuncionarioBiblioteca,
        Integer>(FuncionarioBiblioteca.class, entityManager);
    }

    @Override
    public void inserir(FuncionarioBiblioteca funcionarioBiblioteca) {
        genericoDao.inserir(funcionarioBiblioteca);
    }

    @Override
    public void remover(FuncionarioBiblioteca funcionarioBiblioteca) {
        genericoDao.remover(funcionarioBiblioteca);
    }

    @Override
    public void atualizar(FuncionarioBiblioteca funcionarioBiblioteca) {
        genericoDao.atualizar(funcionarioBiblioteca);
    }

    @Override
    public List<FuncionarioBiblioteca> listarFuncionariosBiblioteca() {
        return genericoDao.listarTodos();
    }

    @Override
    public boolean existeFuncionarioCadastrado() {
        List<FuncionarioBiblioteca> listaDeFuncionarios = listarFuncionariosBiblioteca();

        if(listaDeFuncionarios != null && listaDeFuncionarios.size() > 0){
            return true;
        }

        return false;
    }
}
```

Listagem 24. Código da classe UsuarioDaoImpl, que compõe o DAO.

```
package br.com.javamagazine.daoimpl;

import java.io.Serializable;
import java.util.List;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.Query;
import br.com.javamagazine.dao.GenericoDao;
import br.com.javamagazine.dao.UsuarioDao;
import br.com.javamagazine.entidades.Usuario;

public class UsuarioDaoImpl implements UsuarioDao, Serializable {

    private static final long serialVersionUID = 1L;
    private GenericoDao<Usuario, Integer> genericoDao;
    private EntityManager entityManager;

    @Inject
    UsuarioDaoImpl(EntityManager entityManager){
        genericoDao = new GenericoDaoImpl<Usuario, Integer>(Usuario.class, entityManager);
        this.entityManager = entityManager;
    }

    @Override
    public List<Usuario> listarUsuarios() {
        return genericoDao.listarTodos();
    }

    @Override
    public boolean existeUsuarioCadastrado() {
        List<Usuario> listaDeUsuarios = listarUsuarios();

        if(listaDeUsuarios != null && listaDeUsuarios.size() > 0){
            return true;
        }

        return false;
    }

    @Override
    public Usuario pesquisarUsuario(Usuario usuario) {
        Usuario usuarioEncontrado = null;

        Query query = entityManager.createQuery("select u from Usuario u
        where u.usuario = :usuarioParam and u.senha = :senhaParam")
        .setParameter("usuarioParam", usuario.getUsuario())
        .setParameter("senhaParam", usuario.getSenha());

        try{
            usuarioEncontrado = (Usuario)query.getSingleResult();
        }catch(NoResultException e){
            // não faz nada, se não achar o usuário deixa retornar null
        }

        return usuarioEncontrado;
    }
}
```

Listagem 25. Código da classe LivroDaoImpl, que compõe o DAO.

```
package br.com.javamagazine.daoimpl;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import br.com.javamagazine.dao.GenericoDao;
import br.com.javamagazine.dao.LivroDao;
import br.com.javamagazine.entidades.Emprestimo;
import br.com.javamagazine.entidades.Livro;

public class LivroDaoImpl implements LivroDao, Serializable {

    private static final long serialVersionUID = 1L;
    private GenericoDao<Livro, Integer> genericoDao;

    @Inject
    LivroDaoImpl(EntityManager entityManager){
        genericoDao = new GenericoDaoImpl<Livro, Integer>(Livro.class, entityManager);
    }

    @Override
    public Livro pesquisarPorId(Integer id) {
        return genericoDao.pesquisarPorId(id);
    }

    @Override
    public List<Livro> listarLivros() {
        return genericoDao.listarTodos();
    }

    @Override
    public List<Livro> listarLivrosDisponiveisParaEmprestimo() {
        List<Livro> todosLivros = listarLivros();
        List<Livro> livrosDisponiveis = new ArrayList<Livro>();

        for(Livro livro : todosLivros){
            List<Emprestimo> emprestimos = livro.getEmprestimos();

            boolean todosEmprestimosTemDataDevolucao = true;

            for(Emprestimo emprestimo : emprestimos){
                if(emprestimo.getDataDevolucao() == null)
                    todosEmprestimosTemDataDevolucao = false;
            }

            if(todosEmprestimosTemDataDevolucao)
                livrosDisponiveis.add(livro);
        }

        return livrosDisponiveis;
    }

    @Override
    public List<Livro> listarLivrosEmprestados() {
        List<Livro> todosLivros = listarLivros();
        List<Livro> livrosEmprestados = new ArrayList<Livro>();

        for(Livro livro : todosLivros){
            List<Emprestimo> emprestimos = livro.getEmprestimos();

            for(Emprestimo emprestimo : emprestimos){
                if(emprestimo.getDataDevolucao() == null)
                    livrosEmprestados.add(livro);
            }
        }

        return livrosEmprestados;
    }

    //restante dos métodos omitidos
}
```

Para entender a lógica contida nesses métodos, precisamos estar cientes de que um livro pode estar relacionado a muitos empréstimos e que cada um desses empréstimos tem uma data de devolução, que representa a data efetiva em que a devolução do livro foi realizada. Enquanto um livro não é devolvido, a data de devolução permanece nula. Dito isso, os dois métodos em questão recuperam uma lista com todos os livros e iteram sobre ela. A cada passada dentro do loop **for** são recuperados os empréstimos de um livro específico e verificado se os empréstimos possuem data de devolução. Se todos os empréstimos vinculados àquele livro específico possuírem data de devolução, significa que o livro está disponível e pode ser emprestado, mas se por ventura algum dos empréstimos tiver a data de devolução nula, significa que ele já se encontra emprestado.

Note que apenas fizemos uso de uma pequena parte de tudo que está disponível dentro das especificações da JPA e do CDI. Isso porque cada sistema tem requisitos diferentes, e escolhemos justamente as partes de ambas as especificações que pudessem nos ajudar a atender os requisitos da aplicação de gerenciamento de bibliotecas.

Podemos falar de algumas possibilidades citando alguns exemplos para que tudo se torne um pouco mais tangível para o leitor. Um exemplo relacionado à JPA seria o uso da anotação **@Inheritance** com a estratégia **JOINED**. Existem outras estratégias que poderiam ser adotadas, como **SINGLE_TABLE** e **TABLE_PER_CLASS**. E por que utilizar a opção **JOINED**? O banco de dados fica normalizado (o que não acontece com a opção **SINGLE_TABLE**), podemos aplicar constraints **NOT NULL** nas colunas mapeadas para os atributos das subclasses (isso não seria possível com a opção **SINGLE_TABLE**), tem performance razoável (não tão boa quanto se usássemos **SINGLE_TABLE**, mas melhor do que se usássemos **TABLE_PER_CLASS** em uma situação onde SQL UNIONs não fossem suportados).

Outro exemplo relacionado ao CDI, seria o uso de Qualifiers, Alternatives e **@Vetoed** – assuntos que apenas mencionamos vagamente. No caso da nossa aplicação, não foi necessário o uso de nenhum desses recursos, porém vale a pena conhecê-los, pois são de grande utilidade. O uso de um Qualifier pode ser aplicado quando temos mais de uma implementação para um determinado tipo e precisamos informar de alguma forma ao CDI provider qual

Criando uma aplicação corporativa em Java – Parte 2

das implementações deve ser injetada. A anotação `@Alternative`, quando usada em algum bean, o torna indisponível para injeção, no entanto, ao adicionar também uma declaração `alternatives` no `beans.xml`, tornamos o bean novamente disponível para injeção. Para finalizar, vamos falar da anotação `@Vetoed`, que ao ser utilizada em alguma classe, faz com que essa classe seja ignorada pelo container CDI.

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 Vesteer. Entusiasta de linguagens de programação e tecnologia.



Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



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/>

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

CURSOS ONLINE

.net
magazine

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



CONHEÇA OS CURSOS MAIS RECENTES:

- Curso Padrões de Projeto com C#
- Curso básico de ASP .NET
- Curso de Introdução ao .NET Framework
- Curso Básico de C#
- C# 5 e suas novidades
- ASP.NET MVC – Sistema de Vestibular



Para mais informações :

www.devmedia.com.br/curso/netmagazine

(21) 3382-5038

Web Services REST versus SOAP

Diferenças e vantagens ao adotar essas tecnologias em seu projeto Java

A comunicação entre sistemas e a capacidade de expor serviços através da Internet se tornaram uma necessidade comum para a grande maioria dos sistemas corporativos. Seja apenas para prover suporte a outros módulos na mesma rede privada ou para permitir o acesso público a um determinado serviço, os web services são uma das tecnologias mais utilizadas tanto no Java como em outras linguagens de grande porte e fazem parte do dia a dia dos desenvolvedores.

Entre as abordagens existentes para a implementação de web services, o protocolo SOAP e o REST são as opções de maior destaque nos dias de hoje, estando presentes em grande parte das discussões relacionadas a arquiteturas orientadas a serviços na web. Ganhando destaque no mercado do início da década de 2000, o protocolo SOAP teve grande importância em 2003, quando passou a ser uma recomendação da W3C para desenvolvimento de serviços web, sendo o padrão mais implementado na época e deixando um legado de sistemas e integrações que perdura até hoje.

O REST, por sua vez, foi desenvolvido juntamente com o protocolo HTTP 1.1 e, ao contrário do SOAP, que tem como objetivo estabelecer um protocolo para comunicação de objetos e serviços, propôs algumas ideias de como utilizar corretamente os verbos HTTP (GET, POST, PUT, HEAD, OPTIONS e DELETE) para criar serviços que poderiam ser acessados por qualquer tipo de sistema. Sua concepção, portanto, não era de um protocolo, mas sim de um Design Pattern arquitetural para serviços expostos numa rede, como a internet, através do protocolo HTTP.

Para introduzir essas duas tecnologias, nesse artigo apresentaremos o funcionamento de ambos os métodos, explicando as principais características, vantagens e desvantagens de cada um deles e também demonstraremos como implementá-los em um projeto Java.

Além disso, apresentaremos também as boas práticas de desenvolvimento de cada uma dessas tecnologias

Fique por dentro

Este artigo abordará as principais tecnologias para criação de web services: REST e SOAP, explicando o conceito por trás de cada uma delas e como aplicá-las em um projeto Java. O assunto em análise será de bastante interesse para desenvolvedores que não possuem domínio sobre web services e que desejam conhecer estas opções, assim como para desenvolvedores mais experientes, que buscam compreender as diferenças e vantagens de cada uma dessas abordagens e quando adotá-las.

e introduziremos situações reais em que essas soluções podem ser aplicadas, através de um exemplo a ser desenvolvido no final do artigo.

Com isso, seremos capazes de diferenciar claramente ambas as abordagens e teremos o conhecimento necessário para entender em quais cenários cada uma delas se aplica, permitindo, assim, que o leitor tenha um arsenal maior para o desenvolvimento e consumo de serviços dos mais variados formatos.

Uma breve introdução sobre web services

Web services são, por definição, serviços expostos em uma rede que permitem a comunicação entre um ou mais dispositivos eletrônicos, sendo capazes de enviar e processar dados de acordo com sua funcionalidade.

Essa comunicação, por sua vez, segue alguns protocolos, principalmente relacionados ao formato da transmissão de dados, o que permite que, uma vez que um sistema implemente essas regras, qualquer outro sistema que siga o mesmo protocolo seja capaz de se comunicar com ele.

Devido a isso, os web services se tornaram extremamente populares, pois acabaram por permitir a comunicação entre plataformas completamente diferentes (Java, C#, C++, Ruby) sem grandes esforços.

Entre essas padronizações estipuladas, as duas de maior destaque são o protocolo SOAP e o modelo de design REST, as quais iremos discutir nos próximos tópicos.

Protocolo SOAP

O protocolo SOAP, abreviação para *Simple Object Access Protocol*, é uma especificação para a troca de informação entre sistemas, ou seja, uma especificação de formato de dados para envio de estruturas de dados entre serviços, com um padrão para permitir a interoperabilidade entre eles. Seu design parte do princípio da utilização de XMLs para a transferência de objetos entre aplicações, e a utilização, como transporte, do protocolo de rede HTTP.

Os XMLs especificados pelo SOAP seguem um padrão definido dentro do protocolo. Esse padrão serve para que, a partir de um objeto, seja possível serializar o mesmo para XML e, também, deserializá-lo de volta para o formato original. Além do formato dos objetos, nesse protocolo também são definidos os padrões que os serviços SOAP devem seguir, ou seja, a especificação dos endpoints que as implementações de SOAP devem ter.

A serialização de objetos, que acabamos de descrever, tem como objetivo formar uma mensagem SOAP, composta pelo objeto denominado envelope SOAP, ou **SOAP-ENV**. Dentro desse envelope existem mais dois componentes: o **header SOAP**, que possui informações de atributos de metadados da requisição como, por exemplo, IP de origem e autenticação; e o **SOAP body**, que possui as informações referentes à requisição, como o nome dos métodos que deseja se invocar e o objeto serializado que será enviado como payload da requisição.

Na **Listagem 1** apresentamos um exemplo de um envelope SOAP simplificado representando uma requisição SOAP. Nele podemos identificar claramente os elementos **body** e **header** e também ter uma breve visão da sintaxe do SOAP. Dentro do **body** ainda se nota a descrição do método para o qual esse envelope deve ser direcionado (**GetRevista**) e o payload específico que foi enviado a essa função.

Listagem 1. Exemplo de um envelope SOAP que transporta um objeto com o nome "RevistaNome".

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetRevista xmlns:m="http://www.example.org/revista">
      <m:RevistaNome>Java Magazine</m:RevistaNome>
    </m:GetRevista>
  </soap:Body>
</soap:Envelope>
```

Como verificado, não existe muito segredo no transporte de objetos através do SOAP. No caso do nosso exemplo, a requisição ao método **GetRevista** é enviada para o servidor juntamente com o objeto **RevistaNome**, que servirá de parâmetro de entrada ao nosso método e tem, como conteúdo, a **String** "Java Magazine". Ao chegar ao servidor, essa informação é parseada e a aplicação realiza a lógica necessária para processá-la.

Como boa prática, esse serviço também tem seu funcionamento e formato de dados de entrada especificados pelo protocolo SOAP.

Essa definição de como os métodos que compõem um web service SOAP devem funcionar é feita através de um documento XML chamado WSDL, abreviação de *Web Service Description Language*, um XML que descreve o formato dos dados que o serviço aceita, seu funcionamento e comportamento (se é assíncrono ou não, por exemplo) e os dados de saída do método, incluindo desde o formato até o envelope SOAP que deve ser retornado.

Na **Listagem 2** apresentamos um exemplo de WSDL especificando nosso serviço **GetRevista**, que aceita uma **String** como entrada e devolve, como retorno, outra **String**.

Listagem 2. Exemplo de WSDL para o serviço getRevista.

```
<definitions name="HelloService"
  targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="GetRevistaRequest">
    <part name="RevistaNome" type="xsd:string"/>
  </message>
  <message name="GetRevistaResponse">
    <part name="RevistaNome" type="xsd:string"/>
  </message>

  <portType name="Revista_PortType">
    <operation name="GetRevista">
      <input message="tns:GetRevistaRequest"/>
      <output message="tns:GetRevistaResponse"/>
    </operation>
  </portType>

  <binding name="Revista_Binding" type="tns:Revista_PortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetRevista">
      <soap:operation soapAction="GetRevista"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          use="encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          use="encoded"/>
      </output>
    </operation>
  </binding>

  <service name="GetRevista">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Revista_Binding" name="Revista_Port">
      <soap:address
        location="http://www.example.com/Revista/"/>
    </port>
  </service>
</definitions>
```

Conforme podemos verificar, o WSDL apresenta diversos atributos referentes ao serviço, como a localização do endpoint, nomes de cada um dos serviços, nomes de cada parâmetro e seu respectivo tipo. Através dessa definição, qualquer sistema externo pode se

comunicar com o web service que criamos no WSDL, enviando a requisição através do protocolo SOAP.

Entre os elementos do WSDL que apresentamos na **Listagem 2**, podemos observar que a definição de um serviço é basicamente composta por quatro partes principais. A primeira delas, definida pela tag **<message>**, é onde definimos o formato de nossos dados, podendo ser esses de entrada e saída. Nesse ponto, podemos utilizar tanto atributos primitivos, como no exemplo o uso de **String**, como também utilizar schemas XMLs (chamados de XSDs) para definir nossos objetos.

Uma vez definido o formato de nossas informações, a segunda parte de um WSDL é especificar as operações que desejamos expor e suas respectivas entradas e saídas. Isso é feito na tag **<portType>**, onde definimos a operação **GetRevista** e determinamos, também, os dados de entrada e saída, baseados nas tags **<message>** que criamos anteriormente.

Em seguida, especificamos a tag do WSDL chamada de **<binding>** e, dentro desta, a nossa tag **portType** (criada no passo anterior) e suas respectivas operações a um serviço. Esse serviço, que representará a implementação de nosso web service, é determinado na tag **<service>**, onde explicitamos o endereço do endpoint através da tag **<soap:address location>**.

Além de tudo isso, o SOAP também possibilita funcionalidades bastante interessantes, como é o caso do WS-Addressing. Esse tipo de tecnologia permite que, uma vez enviada alguma informação dentro de um envelope SOAP, seja possível o envio de um parâmetro extra definindo um endpoint de callback para ser chamado logo que a requisição termine de ser processada.

Dessa forma, uma vez que termine a requisição, o web service é capaz de chamar essa URL de callback, passando em seu conteúdo o resultado do processamento e permitindo, dessa forma, o trabalho assíncrono entre sistemas.

Modelo arquitetural REST

Ao introduzir e adotar essas padronizações, muitos sistemas começaram a implementar esse modelo na criação de seus web services, fazendo com que o protocolo SOAP se tornasse uma das tecnologias essenciais no desenvolvimento de web services em sistemas corporativos.

No entanto, o protocolo SOAP trazia também diversas desvantagens. A primeira delas (e talvez uma das mais importantes) é o fato de que ao transportar todas as informações dentro de um envelope SOAP em XML, o conteúdo dos dados enviados de um sistema para outro se torna muitas vezes maior que o necessário, elevando tanto o consumo da banda de rede como o tempo de processamento dos dados.

Em segundo lugar, a utilização de web services SOAP não faz o uso correto dos verbos HTTP. Isso se deve ao fato de que todas as requisições SOAP são feitas através do POST de um XML, que contém o envelope SOAP. Porém, em grande parte das requisições, o mais adequado seria outro verbo, como um GET ou PUT, de acordo com a funcionalidade exposta pelo serviço. Graças a essa limitação do protocolo, o SOAP acaba por contrariar alguns dos

princípios de uma boa modelagem HTTP, ferindo as boas práticas dessa especificação.

Por esses e outros motivos, desenvolvedores e pesquisadores vêm adotando outra abordagem para criar seus web services, chamada de serviços REST. A modelagem por trás de um serviço REST parte do princípio de seguir as boas práticas da criação de serviços HTTP e utilizar esses padrões para desenvolver web services simples e performáticos.

Um dos pontos cruciais nesse modelo arquitetural é o uso correto dos métodos disponibilizados pelo HTTP. Ao contrário do SOAP, que só utiliza o método POST para transmitir os dados, uma arquitetura REST prevê que, dentro de um cenário ideal, o método HTTP a ser utilizado seja diretamente relacionado à funcionalidade do serviço a ser consumido.

Portanto, serviços de busca de informações são feitos através de métodos **GET**, serviços de atualização de informação através de métodos **PUT**, serviços de criação de dados através do método **POST**, serviços de deleção através do **DELETE** e assim por diante. A partir do correto uso dos verbos HTTP ganhamos também a vantagem de não termos diversas URLs para cada um dos nossos serviços, podendo expor somente uma URL e, conforme o método, executar uma ação diferente.

Como exemplo, vamos imaginar um simples serviço CRUD (Criação, atualização, leitura e deleção de dados) para clientes. Dentro de uma arquitetura REST, esse serviço pode ser definido somente com uma URL, por exemplo: <http://nosso.exemplo.com/client/12345>. Para realizar uma consulta ao cliente 12345, basta realizarmos uma requisição **GET** à URL mencionada e para deletá-lo, realizar uma requisição **DELETE**.

O mesmo pode se aplicar às requisições **PUT** e **POST**, onde utilizamos o corpo da requisição que nos foi enviado para atualizar ou criar uma nova entidade em nosso sistema, respectivamente. Deste modo, é fácil observar que, utilizando os métodos corretamente, os web services REST se tornam bastante simples de serem desenvolvidos e altamente intuitivos, sem a necessidade de descrições, como os WSDLs ou documentações extensas, para explicar sua funcionalidade.

Outra vantagem do REST em relação ao SOAP se refere ao formato do dado a ser trafegado. Enquanto o SOAP se restringe a utilizar XMLs, o REST não impõe restrições quanto a isso, ou seja, qualquer um pode ser enviado em suas requisições. Diretamente relacionado a isso, como boa prática, uma requisição REST deve sempre conter o header **Content-Type** para explicitar o formato do dado em sua requisição e, para o retorno, o header **Accepts**, determinando o formato de dado que a aplicação espera. Como exemplo, caso uma aplicação deseje enviar um dado em XML e receber, como retorno, um JSON, basta ela explicitar ambos os formatos nos respectivos headers e o web service REST deve ser capaz de resolver a requisição de acordo com essas necessidades.

Dentro do REST, outra diferença é o chamado HATEOAS, abreviação para *Hypermedia as the Engine of Application Status*. Considerado um dos principais benefícios dessa arquitetura, o HATEOAS define que, dentro de uma aplicação REST, o estado da aplicação

e da interação do usuário deve ser controlado através das URLs (Hypermedia) retornadas pelo serviço. Isso implica que, em um serviço REST, quando a primeira URL dessa aplicação é acessada, todos os possíveis “caminhos” que o cliente pode seguir a partir dali são retornados dentro da resposta da requisição, como as URLs dos serviços que podem ser invocados no passo seguinte.

Através dessa técnica evitamos diversos problemas em nossos clientes, uma vez que eles só precisam ter a primeira URL de nossa aplicação hardcoded em seu código, possibilitando que as URLs seguintes sejam capturadas durante a própria execução do web service.

Aplicações RESTful

As características apresentadas anteriormente são somente algumas das principais definições do modelo REST. Podemos dizer que, uma aplicação que segue todas as definições e funcionalidades desse modelo, é chamada de RESTful. Essas especificações arquiteturais, denominadas de **Architectural Constraints**, são divididas em cinco grupos principais, cada um referenciando as características mais importantes do modelo REST.

A primeira funcionalidade desse grupo exige que um sistema RESTful siga o modelo Cliente-Servidor. Nesse modelo, as responsabilidades de cada uma das partes ficam claramente divididas, deixando todo ou grande parte do código do lado do servidor, que expõe essas funcionalidades através de uma interface uniforme ao cliente (web services) e, consequentemente, minimiza problemas na aplicação provenientes de código no cliente.

Em segundo lugar, serviços RESTful devem ser stateless, ou seja, não devem guardar estado. Isso explica que, dada uma requisição, todas as informações necessárias para o processamento desta devem estar presentes no payload enviado ao serviço.

Também como uma necessidade importante, é essencial que os serviços REST sejam cacheados, ou seja, o cliente deve ser capaz de acessar um cache das respostas, evitando que sejam feitas requisições seguidas a recursos que não sofreram alteração e permitindo, portanto, um menor consumo de banda. Além disso, quando um desses recursos for atualizado, esse cache deve ser capaz de automaticamente notar isso e responder a requisição atualizada ao cliente.

Outro ponto importante para o cliente é que os serviços REST devem poder ser separados em camadas de forma transparente ao usuário. Isso implica que, se quisermos colocar um serviço intermediário ou um Load Balancer entre o serviço final e o consumidor desse sistema, não deve haver grandes alterações em ambos os lados, sendo essa transição o mais transparente possível.

Por fim, um sistema RESTful deve ser capaz de prover uma interface uniforme através de endpoints HTTP para cada um dos recursos disponíveis possibilitando, também, a implementação dos itens que mencionamos anteriormente, incluindo o HATEOAS, uso correto das URLs e dos métodos HTTP e a capacidade de descrever o tipo de dado através de headers (**Content-Type** e **Accept**).

Criando os nossos web services em Java

Agora que entendemos os principais conceitos por trás desses dois modelos de web service, iremos apresentar como criá-los dentro de um projeto Java. Para isso, vamos introduzir duas bibliotecas bastante importantes: a JAX-WS, para projetos SOAP; e a JAX-RS, para projetos REST.

Configurando nosso projeto

O primeiro passo é realizar o download de um servidor que implementa as bibliotecas da Java EE 6, permitindo assim o uso das funcionalidades do JAX-WS e do JAX-RS. Como escolha para esse artigo, decidimos adotar o servidor WildFly, considerado o sucessor do famoso JBoss AS (veja o endereço para download na seção [Links](#)).

Feito o download do WildFly, para realizar a instalação basta descompactá-lo em alguma pasta de seu computador e, em sua IDE, criar a configuração de um novo servidor e identificar, dentro das propriedades, o caminho do diretório em que os arquivos do mesmo foram descompactados. Neste exemplo, iremos utilizar a IDE Eclipse Kepler para desenvolver nossos serviços e o plugin do JBoss Tools para configurar nosso servidor.

Nota:

A instalação e configuração do plugin do JBoss, juntamente com a instalação do servidor na IDE, podem ser visualizadas com mais detalhes no endereço indicado na seção [Links](#).

Configurado o servidor, podemos agora criar o projeto na IDE. Para isso, basta clicarmos em *New > Dynamic Web Project*, nomearmos nosso projeto e escolher, como runtime de nossa aplicação, o servidor recém-configurado, o WildFly 8. Na [Figura 1](#) mostramos essa configuração.

Feito isso, precisamos configurar o projeto como sendo do tipo Maven. Essa etapa pode ser feita com a ajuda do plugin do Maven (que pode ser baixado pelo próprio eclipse), bastando adicionar o *pom.xml* apresentado na [Listagem 3](#) na raiz do projeto e clicar com o botão direito no projeto e escolher *Configure > Convert to Maven Project*.

Como pode ser visto no *pom.xml*, basicamente definimos duas dependências ao nosso projeto: a dependência relacionada ao **resteasy-jaxrs** e a dependência relacionada ao **resteasy-jackson**. Essas bibliotecas servem, respectivamente, para implementar o redirecionamento das requisições HTTP aos respectivos serviços REST, e, também, para permitir o processamento de objetos em formato JSON.

Criando nosso serviço SOAP

Com o projeto configurado, podemos demonstrar os passos para a criação de um web service SOAP. Para viabilizar esse exemplo, iremos utilizar a biblioteca JAX-WS e, também, as *Annotations* do Java, para declarar quais serão nossos métodos que irão representar os serviços expostos na web.

Assim, uma vez que a aplicação seja colocada no servidor, o JAX-WS é responsável por verificar quais das classes estão anotadas

Web Services REST versus SOAP

com `@WebService` e informar ao servidor quais são as URLs de conexão para cada uma dessas classes, criando também os respectivos WSDLs.

Na **Listagem 4** apresentamos a classe `JavaMagazineSoap`, onde criamos um serviço SOAP expondo o método `hello()`, responsável por receber uma `String` e imprimir a mensagem `Hello` no console.

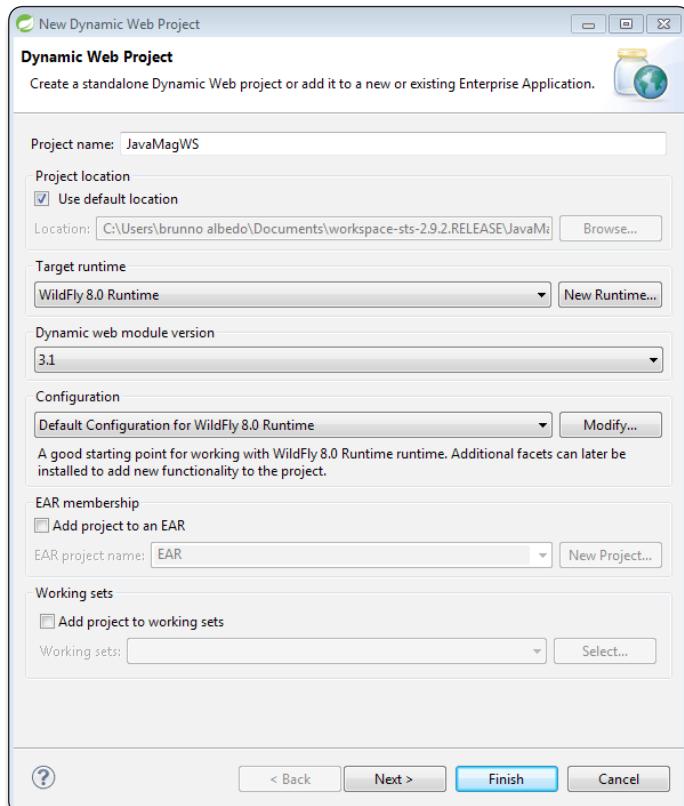


Figura 1. Configuração de nosso projeto

Listagem 3. Pom.xml a ser adicionado ao projeto.

```
<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"> <modelVersion>4.0.0
  </modelVersion>
  <groupId>com.java.mag</groupId>
  <artifactId>WS</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jaxrs</artifactId>
      <version>3.0.6.Final</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jackson-provider</artifactId>
      <version>3.0.6.Final</version>
    </dependency>
  </dependencies>
</project>
```

Listagem 4. Criando uma classe para expor um web service SOAP.

```
@WebService
public class JavaMagazineSoap {

  public String hello(String hi){
    System.out.println("Hello!!!");
    return hi;
  }
}
```

Como demonstrado nesse código, a criação de web services SOAP é bastante simples, sendo necessário apenas anotar a classe que queremos para que seus métodos sejam expostos como serviços.

Assim que finalizar o desenvolvimento dessa classe, você pode subir nosso projeto no WildFly e então observar o WSDL gerado pelo JAX-WS. Para verificar esse arquivo, basta acessar o diretório indicado no log (exibido no console de saída de sua IDE), na linha `WSDL published to:`. No caso do projeto desse artigo, a linha de log gerada foi: `WSDL published to: file: /Users/brunnoattorre1/Documents/servers/Wildfly8.2/standalone/data/wsdl/JavaMagazineWS.war/JavaMagazineSoapService.wsdl`.

Nota:

O caminho será gerado de acordo com o nome do seu projeto. No nosso caso, como o nome do projeto é `JavaMagazineWS`, o contexto do caminho ficou com esse nome. Caso seu projeto tenha outro nome, basta substituir o caminho de acordo com a respectiva nomenclatura que será apresentada.

Também é importante notar que, no desenvolvimento dos serviços SOAP, podemos utilizar objetos dentro dos parâmetros de entrada e saída. Na **Listagem 5** apresentamos um exemplo que utiliza objetos como parâmetros de entrada, representados pelos objetos `user1` e `user2`, e saída do serviço, representado pelo objeto retornado pelo método `hello()`. Além disso, também implementamos a classe `User`, que representará o objeto que servirá de parâmetro para o nosso web service.

Casos de uso do SOAP

Para apresentarmos um caso de uso real de web services SOAP e contextualizarmos o que mostramos até aqui, vamos introduzir nesse tópico uma situação em que devemos desenvolver um sistema capaz de receber notificações externas de diversas aplicações.

A escolha do SOAP, nesse caso, se dá pelo fato de que, em nosso cenário fictício, trabalharemos com sistemas das mais diversas linguagens e, muitos deles, não possuem suporte ao REST. Numa situação semelhante a essa, o SOAP se mostra como uma alternativa mais viável, uma vez que viabiliza um suporte mais extenso principalmente para plataformas mais antigas.

O exemplo dessa implementação se encontra na **Listagem 6**, onde apresentamos o código da classe `NotificationSoap`, responsável por receber uma notificação através do protocolo SOAP.

Listagem 5. Implementação do web service SOAP JavaMagazineSoapObject.

```
@WebService  
public class JavaMagazineSoapObject {  
  
    public User hello(User user1, User user2){  
        System.out.println(user1.getName());  
        return user2;  
    }  
  
    class User{  
        private String name;  
  
        public String getName() {  
            return name;  
        }  
  
        public void setName(String name) {  
            this.name = name;  
        }  
    }  
}
```

Listagem 6. Classe NotificationSoap implementando um web service de notificação.

```
@WebService  
public class NotificationSoap {  
  
    @Oneway  
    public void notificate(Message message){  
        System.out.println("Recebeu a mensagem "+ message.getPayload());  
    }  
  
    class Message implements Serializable{  
        private String payload;  
        private String error;  
        public String getPayload() {  
            return payload;  
        }  
        public void setPayload(String payload) {  
            this.payload = payload;  
        }  
        public String getError() {  
            return error;  
        }  
        public void setError(String error) {  
            this.error = error;  
        }  
    }  
}
```

Nessa listagem, além das anotações que já apresentamos anteriormente, introduzimos a anotação `@Oneway`. Essa anotação define que o serviço referente ao método que for anotado será apenas de notificação, ou seja, ele não terá retorno, servindo somente para enviar mensagens. Neste caso, mesmo que ocorra alguma exceção no processamento da mensagem, essa exceção não será retornada ao cliente. Deste modo, é válido ressaltar que, quando desejamos criar serviços de notificação SOAP, é sempre importante tratarmos corretamente a exceção no próprio serviço, evitando que erros não tratados aconteçam.

Uma vez implementada essa classe, basta subirmos o servidor e acessarmos o WSDL gerado (da mesma forma que acessamos no exemplo anterior) para observarmos as características do serviço

criado e, caso seja necessário testar, basta utilizarmos um cliente capaz de realizar requisições SOAP.

Nota:

O aplicativo SoapUI é uma alternativa viável para desenvolvedores que precisam testar seus web services. Ela permite, através de um WSDL previamente gerado, realizar requisições com um conteúdo definido pelo usuário. Na seção de **Links** adicionamos mais algumas informações de como realizar o download e utilizar essa ferramenta para testar web services.

Criando serviços REST

Agora que entendemos o básico por trás da criação de serviços SOAP, vamos introduzir a biblioteca JAX-RS, responsável por habilitar a criação de serviços utilizando o modelo REST. Para isso, partiremos do mesmo princípio da biblioteca JAX-WS, ou seja, implementaremos, dentro de uma classe Java comum, os métodos que desejamos expor via web service REST.

Uma vez implementada, utilizaremos annotations do Java para indicar a classe e os métodos que desejamos disponibilizar via REST, referenciando também outras informações como o tipo do verbo HTTP para uma determinada requisição, o conteúdo produzido por determinado serviço e a URI de cada um dos serviços.

Além do JAX-RS, iremos utilizar a biblioteca RESTEasy, que declaramos anteriormente como dependência de nosso projeto. Esta será responsável por tratar as requisições REST e direcioná-las ao serviço correto. A configuração dessa biblioteca é feita no arquivo `web.xml`, sendo o arquivo de nosso exemplo apresentado na [Listagem 7](#).

Nessa listagem podemos observar a configuração de alguns itens importantes. O primeiro deles, indicado pelo `context-param` de nome `resteasy.scan`, permite referenciar se queremos que o RESTEasy rastreie em nosso projeto todas as classes que possuem a anotações `@Path` do JAX-RS e, em seguida, disponibilize essas classes como serviços REST. No caso de nosso projeto, indicamos o valor do parâmetro `resteasy.scan` como `true` e possibilitamos ao RESTEasy realizar esse trabalho automaticamente.

Em segundo lugar, configuramos o listener `ResteasyBootstrap` e a servlet `HttpServletDispatcher`, responsáveis por, para cada requisição, fazer o redirecionamento à classe correta, que foi escaneada no passo anterior.

Por fim, definimos também qual serão as URLs que servirão de endpoints para nossas requisições REST. Essas são indicadas tanto no parâmetro `resteasy.servlet.mapping.prefix` como, também, no `servlet-mapping` do nosso servlet `HttpServletDispatcher`, ambas colocando, como endereço da URL, o endpoint com o caminho `/rest/*`.

Uma vez terminadas as configurações dentro do `web.xml`, basta anotarmos uma classe com as anotações do JAX-RS para podermos indicar, ao RESTEasy, que essa classe servirá como um serviço REST. Para demonstrar o uso dessas anotações, na [Listagem 8](#) apresentamos o código da classe `JavaMagazineRest`, que recebe uma requisição GET e devolve um HTML com a mensagem `"Hello, World!!"`.

Web Services REST versus SOAP

Listagem 7. Conteúdo do arquivo web.xml com a configuração do RESTEasy.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  id="WebApp_ID" version="3.1">
  <display-name>JavaMagazineWS</display-name>

  <context-param>
    <param-name>resteasy.scan</param-name>
    <param-value>true</param-value>
  </context-param>

  <context-param>
    <param-name>resteasy.servlet.mapping.prefix</param-name>
    <param-value>/rest</param-value>
  </context-param>

  <listener>
    <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>resteasy-servlet</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>resteasy-servlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Listagem 8. Código da classe JavaMagazineRest, que implementa um web service REST.

```
@Path("/rest")
public class JavaMagazineRest {

    @GET
    @Produces("text/html")
    public String example(){
        return "<html lang='en'><body><h1>Hello, World!!</h1></body></html>";
    }
}
```

Conforme podemos verificar, a criação da classe **JavaMagazineRest** e o uso das anotações do JAX-RS podem ser explicados em três etapas principais. A primeira delas, definida pela anotação **@Path** aplicada em um método, permite que indiquemos o caminho pelo qual o serviço REST irá responder. No caso desse exemplo, qualquer requisição à URI `/rest` será redirecionada ao nosso serviço.

Logo em seguida, explicitamos o método HTTP no qual o conteúdo da requisição deve ser enviado pelo cliente. Assim, uma vez que essa requisição chegue à aplicação, será respondida pelo método anotado de nossa classe **JavaMagazineRest**. No nosso exemplo, isso é feito através da anotação **@GET**, especificando que o recurso anotado só responderá às requisições do tipo GET.

Por fim, a anotação **@Produces** indica o tipo do dado de retorno (no nosso exemplo, HTML). Caso recebêssemos algum parâmetro de entrada, poderíamos fazer uso da anotação **@Consumes**, que indica o tipo de dado aceito como input para o serviço.

Com a classe desenvolvida, para observarmos o REST em funcionamento, basta subir nossa aplicação no WildFly, acessar a URL `http://localhost:8080/JavaMagazineWS/rest/rest` e então verificar a mensagem “Hello, World”.

Casos de uso do REST

Agora que aprendemos os conceitos básicos por trás do desenvolvimento de serviços REST, apresentaremos nesse tópico uma análise simplificada de um caso de uso real dessa opção de web service para um sistema comercial. Para isso, suponha um cenário onde vamos desenvolver um serviço de CRUD para um cadastro de usuários.

A construção desse serviço é feita a partir do código da **Listagem 9**, onde implementamos a classe **CrudRest**. Esta define os endpoints REST que servirão para expor os endereços de criação, atualização, recuperação e deleção de instâncias da classe **User**, que representa os usuários de nosso sistema.

Nesse código, podemos observar que seguimos alguns dos princípios primordiais do REST. O primeiro deles está relacionado ao caminho de nossas URLs: todas seguem o mesmo padrão e nomenclatura, somente mudando o tipo do método HTTP que será utilizado. Esse tipo de abordagem permite que os serviços se tornem mais intuitivos, ou seja, não é necessária uma extensa documentação para descrever o que cada um faz, pois o próprio método HTTP já nos indica sua funcionalidade.

Além disso, também relacionado às URLs expostas, implementamos o conceito de trabalhar com as entidades do sistema na forma de **recursos** nos links e operações REST. Esse conceito permite que, em uma aplicação REST, sempre trabalhemos com a referência dos recursos que desejamos utilizar nas próprias URLs (por exemplo, em nosso caso, será uma palavra que servirá de identificador único para cada uma de nossas entidades).

Contextualizando essa funcionalidade para o nosso exemplo, se quisermos fazer qualquer ação em cima do objeto **User** com **id** igual a “345abd”, a URL de destino seria sempre a representada pelo endereço `/user/345abd`. A partir daí, qualquer ação que desejamos tomar sobre esse recurso (**User**) deve ser definida pelos métodos HTTP correspondentes. Consequentemente, uma requisição do tipo **DELETE** a essa URL iria remover o usuário cujo **id** é igual “345abd”, enquanto uma requisição do tipo **GET** iria trazer, como retorno, os dados desse usuário.

Além das URLs, podemos observar que também definimos em nossa classe quais são os tipos de dados de entrada e saída para cada um dos serviços. Nas anotações **@Consumes** e **@Produces**, especificamos, respectivamente, qual o tipo de dado de entrada e saída. Em nosso caso serão as informações e objetos serializados no formato JSON. É importante notar que, uma vez que chegue uma requisição a esses serviços, os headers **Content-Type** (tipo de dado de envio) e **Accept** (tipo de dado que aceita como retorno)

devem ter, como conteúdo, o mesmo valor dos definidos em ambas as anotações (**@Consumes** e **@Produces**).

Por fim, para observarmos o funcionamento desses serviços REST recém-desenvolvidos, basta subir nossa aplicação e realizar as respectivas requisições (com os verbos HTTP corretos e os headers de **Content-Type** e **Accept** claramente descritos). Uma vez que essas requisições sejam feitas, podemos verificar o comportamento de cada um de nossos métodos REST e, também, observar os resultados que serão retornados.

Nota:

Como sugestão para testar esses web services, o plug-in do Google Chrome chamado Postman possibilita que sejam feitas as mais diversas requisições, permitindo explicitar os verbos HTTP a serem usados e também definir os headers "Content-type" e "Accept". Para mais informações, adicionamos uma referência na seção [Links](#).

Considerações finais a respeito do uso de REST x SOAP

Com os principais conceitos por trás de web services, tanto do protocolo SOAP como do modelo REST, analisados, é importante entendermos as diferenças e vantagens na utilização de cada um.

Dito isso, a primeira diferença que precisamos ter em mente é que web services SOAP seguem um protocolo específico, enquanto os serviços REST seguem um modelo arquitetural. Essa diferença é importante pois, enquanto o SOAP exige alguns padrões para funcionar, ou seja, se não seguir todos os padrões não conseguimos expor nem consumir um serviço desse tipo; o REST indica algumas boas práticas e aconselha um modelo arquitetural, fican-

do a critério do desenvolvedor utilizar todos ou somente os que atendem à necessidade do projeto.

Outro ponto que o REST se difere do SOAP está no fato de ser uma solução mais leve e de fácil implementação, acarretando em um ganho considerável de velocidade de processamento, sendo assim bastante prático e indicado para projetos onde a performance é essencial.

O SOAP, no entanto, tem funcionalidades muito úteis em situações específicas, trazendo um leque de artefatos para auxiliar o desenvolvedor. Soluções que envolvem autenticação, manter a sessão entre requisições, WS-Addressing (URLs de callback), entre outras, já vêm embutidas no SOAP e podem ser muito úteis em determinadas situações;

Vale a pena lembrarmos também que o desenvolvimento de aplicações REST bem arquitetadas depende muito do time por trás de sua implementação. Os padrões SOAP, por sua vez, já são engessados e limitam a liberdade de customização por parte do desenvolvedor. Devido a isso, o REST é mais aberto e permite que sejam desenvolvidas as mais diversas combinações de recursos dentro de seu modelo arquitetural.

No entanto, isso acarreta tanto vantagens como desvantagens. Vantagens pois, para uma equipe mais experiente, a customização e utilização das funcionalidades necessárias se tornam mais fáceis e práticas de serem implementadas. Como desvantagem, está o fato de que, caso não se tenha um cuidado especial na modelagem e na implementação dos serviços, uma arquitetura REST pode levar o projeto ao fracasso. Com o intuito de auxiliar a escolha entre as tecnologias REST e SOAP, na **Tabela 1** resumimos as principais diferenças, vantagens e desvantagens de cada uma.

Listagem 9. Web services REST para o CRUD do objeto User.

```
@Path("/user")
public class CrudRest {

    @Path("/{id}")
    @GET
    @Produces("application/json")
    public Response getUser(@PathParam("id") String id) {
        // Simulado a busca por um usuário
        System.out.println("buscando usuario "+ id);
        User user = new User();
        user.setName("user Test");
        return Response.status(200).entity(user).build();
    }

    @POST
    @Consumes("application/json")
    @Produces("application/json")
    public Response updateUser( User user) {
        // Simula uma inserção no banco
        System.out.println("Atualizando usuario ");
        return Response.status(200).entity(user).build();
    }

    @Path("/{id}")
    @PUT
    @Consumes("application/json")
    @Produces("application/json")
    public Response deleteUser(@PathParam("id") String id) {
        //Simula uma delecao
        System.out.println("deletando usuario "+ id);

        return Response.status(200).build();
    }

    public Response createUser(@PathParam("id")String id,User user) {
        // Simula uma atualizacao no banco
        System.out.println("Atualizando usuario "+ id);
        return Response.status(200).entity(user).build();
    }
}

class User{
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Web Services REST versus SOAP

	SOAP	REST
Transporte de dados	Mais pesado e custoso, pois necessita do uso do envelope SOAP.	Mais leve e dinâmico, pois não possui um padrão específico para serialização de dados.
Tipo de dados	Sempre trabalha com XML	Pode trabalhar com qualquer tipo de dado, desde que seja especificado na implementação do serviço.
Requisições HTTP	Suporta somente POST	Suporta (e aconselha) o uso de todos os verbos HTTP.
Modelo arquitetural	É um protocolo.	É um conjunto de boas práticas na modelagem de web services.
Supporte em outras linguagens	É suportado basicamente por todas as linguagens, principalmente linguagens mais antigas.	É melhor suportado por linguagens mais modernas e voltadas ao desenvolvimento para web (JavaScript, por exemplo).
Funcionalidades	Possui diversas funcionalidades mais avançadas em sua implementação padrão, como o WS Addressing.	Estipula somente o básico dos serviços, sendo necessária a implementação de funcionalidades mais avançadas.
Uso dentro do Java	Possui diversas funcionalidades, como a criação de classes a partir de WSDLs, que facilitam o desenvolvimento.	Exige a criação e implementação dos métodos que receberão as chamadas HTTP e permite uma maior liberdade na escolha das funcionalidades que deseja implementar.

Tabela 1. Principais diferenças entre SOAP e REST

Enfim, ambas as tecnologias apresentam vantagens e desvantagens e possuem situações específicas nas quais se encaixam melhor para solucionar um determinado problema. Portanto, para escolher a tecnologia ideal para o seu projeto, é aconselhável a opinião de profissionais mais experientes e uma análise de todo o contexto de execução em que os serviços irão rodar.

Autor



Bruno F. M. Attorre

brattorre@gmail.com

Trabalha com Java há quatro anos. Apaixonado por temas como Inteligência Artificial, ferramentas open source, BI, Data Analysis e Big Data, está sempre à procura de novidades tecnológicas na área. Possui as certificações OCJP, OCWCD e OCEEJBD.



Links:

Especificação W3C SOAP.

<http://www.w3.org/TR/soap/>

Definições do REST pela W3C.

<http://www.w3.org/2001/sw/wiki/REST>

Página com instruções de instalação do JBoss Tools e do WildFly 8.

<http://www.mastertheboss.com/wildfly-8/configuring-eclipse-to-use-wildfly-8>

Cliente do Postman.

<https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojopjoooidkmcomcm>

Cliente do SOAPUI.

<http://www.soapui.org>

Uso do SOAPUI para testes SOAP.

<http://www.soapui.org/Getting-Started/your-first-soapui-project.html>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Conhecimento faz diferença!

Processo: Medição de Software:

Edição 24 :: Ano 2

Gerenciamento de Projetos: Definição + Fase Inicial

Teste: Execute testes funcionais com Hudson e Selenium RC

Agilidade: Negociação de Requisitos

Edição 28 :: Ano 2

Evolução da Engenharia de Software: Definições, preocupações e custo

Aulas desta edição:
• Estratégia de Teste Funcional baseada em Casos de Uso – Partes 5 a 9

Processo: A importância da comunicação no processo de software

Agilidade: Acompanhamento de projetos ágeis distribuído através do Daily Meeting

engenharia de software magazine

Edição 29 :: Ano 3

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

ISSN 1983127-7



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!**



Twitter Finagle: Desenvolvendo sistemas distribuídos

Revolução na forma como várias empresas constroem suas soluções de alta performance

Vivemos um momento interessante no mercado de tecnologia: a cada dia surgem várias empresas de internet, ou mais precisamente, “startups”, procurando emplacar um novo produto e arrebatar a atenção (e o dinheiro) da maior quantidade de público possível. Mas algo elas têm em comum: pressa, pois a máxima “tempo é dinheiro” nunca foi tão verdadeira.

Essa urgência de construir soluções a um baixo custo, em tempo recorde e com qualidade faz com que essas empresas novas, ao contrário da maioria das corporações estabelecidas, busquem criar ou reaproveitar softwares inovadores, algumas vezes pouco conhecidos, que nem sempre estão baseados em algum padrão de mercado, que sejam preferencialmente open source e que deem conta do recado, tanto em agilidade de desenvolvimento quanto em performance e estabilidade.

Foi nesse contexto que anos atrás o Twitter criou toda uma linha de ferramentas e frameworks para construir uma solução mais robusta e compatível com suas necessidades de desempenho e escalabilidade. Estamos falando do Finagle, tecnologia que vem revolucionando a forma como as soluções são construídas em grandes empresas como o próprio Twitter, Foursquare, Pinterest, SoundCloud e Tumblr.

O Finagle permite a construção de sistemas fortemente orientados a serviços, fazendo uso extensivo de IO não bloqueante (NIO) e processamento concorrente, além de aceitar múltiplos protocolos de comunicação e não ser executado em servidores de aplicação como o JBoss AS ou o Oracle WebLogic.

Um detalhe interessante e fundamental: o Finagle não está totalmente comprometido em seguir alguma

Fique por dentro

Neste artigo vamos aprender o que é o framework Twitter Finagle, seus conceitos arquiteturais, que privilegiam a composição e o reaproveitamento de serviços, abstrações, APIs de desenvolvimento e as principais funcionalidades. Deste modo será possível entender porque ele é uma opção viável para o desenvolvimento de sistemas escaláveis e porque ele vem sendo adotado cada vez mais por empresas de internet e Startups para a criação de soluções de alta performance.

especificação ou padrão pré-estabelecido de desenvolvimento e justamente por esse motivo não está preso a tradicionalismos ou restrições técnicas da plataforma Java EE ou qualquer outra.

Além disso, ressalta-se que o Finagle foi criado para ser executado sobre a Java Virtual Machine e o desenvolvimento das aplicações pode ser feito tanto em Java quanto em Scala (que gera bytecodes compatíveis com a JVM). Como a equipe de engenheiros do Twitter desenvolveu boa parte do Finagle com a linguagem Scala, algumas APIs e classes requerem alguma adaptação para serem utilizadas quando desenvolvemos com Java. Outra característica marcante desse framework é ser nativamente integrado com ferramentas de monitoramento, rastreabilidade de execução e clustering.

O que é o Finagle?

O Finagle é um framework para construção de sistemas distribuídos extensível e configurável para a JVM, sendo empregado para construir servidores altamente concorrentes e de alta performance. Através de sua API é possível implementar tantos clientes quanto servidores e por ser extensível, permite customizações nos protocolos de comunicação e na configuração de funcionalidades

dos servidores e clientes, como adição de rastreabilidade, log e métricas, oferecendo também mecanismos para compor serviços a partir de serviços pré-existentes, levando o reaproveitamento ao máximo.

Os seguintes objetivos guiaram a idealização desse framework:

- **Não ser apenas um framework web:** o Finagle pode ser usado para criar Web Services, RESTful Services e outros tipos de serviços, com contratos complexos e suportando inclusive protocolos especializados, além do HTTP;
- **Isolar responsabilidades e preocupações:** de acordo com as melhores práticas de arquitetura SOA, sabemos que um serviço deve ser coeso, com responsabilidades bem definidas e focado apenas em fazer aquela função para o qual foi criado. No Finagle essa prática de divisão de responsabilidades é encorajada, pois ele oferece uma API especializada para composição de serviços;
- **Confiabilidade e isolamento de falhas:** Cada servidor Finagle é executado em sua própria JVM e deve ser responsável por apenas um serviço. Se esse servidor falhar por qualquer motivo, pode ser mais facilmente recuperado ou reiniciado do que se estivesse sendo executado em um servidor de aplicações, por exemplo. Por default, cada servidor possui também mecanismos de detecção de falhas, balanceamento e reconexão com outros servidores;
- **Produtividade de desenvolvimento:** o Finagle vem acompanhado de ferramentas para geração de código de clientes e servidores e essas ferramentas se encarregam de boa parte da complexidade acidental do desenvolvimento, que é aquela que não tem nada a ver com o negócio, por exemplo, tratamento de conexões de rede, protocolos, IO etc.

Os engenheiros criadores do Finagle também chegaram à conclusão de que apenas através do uso de abstrações corretas de processamento concorrente, assincronismo e IO não bloqueante seria possível construir servidores que fossem capazes de atender a características de performance tão grandes, a um custo compatível. Partindo deste princípio, eles definiram as seguintes abstrações arquiteturais: Server, Service, Filter e Future.

Abstrações do Finagle

A abstração central do Finagle é o Serviço (Service) e qualquer implementação dessa abstração deve estender a classe `com.twitter.finagle.Service`. Conceitualmente, um Service é uma “Interface funcional”, um conceito novo do Java 8, mas bem conhecido principalmente em linguagens de programação que seguem o paradigma funcional. Uma interface funcional, tal qual descrito na documentação do Java 8, tem exatamente um método abstrato que aceita exatamente um parâmetro e retorna uma resposta. No caso do Serviço, esse parâmetro é do tipo “Request” (requisição) e o segundo parâmetro é do tipo “Response” (resposta), por exemplo, se estamos criando um serviço que utiliza o protocolo HTTP, o “request” será do tipo `HttpRequest` e o resultado `HttpResponse`. Por sua vez, se o serviço utilizar o protocolo Thrift, o “request” e “response” serão do tipo `byte[]` (array de bytes).

Serviços que trabalhem com protocolos mais complexos como o Thrift (ver **BOX 1**) podem ser criados a partir de IDLs (*Interface Definition Language* – vide **BOX 2**). As IDLs simplificam o processo de desenvolvimento, já que as interfaces de serviço descritas nessa linguagem servem para a geração de código através das ferramentas especializadas do Finagle.

BOX 1.0 protocolo Thrift

O protocolo Thrift é um protocolo de comunicação que faz parte de um framework que leva esse mesmo nome, tendo sido criado pelo Facebook para permitir a troca de mensagens entre sistemas escritos nas mais diversas linguagens de programação. Sua arquitetura prevê níveis de customização no formato das mensagens, que podem ser textuais como HTTP ou JSON, ou binárias. Além disso, as mensagens suportam versionamento, o que na prática é bem útil, pois clientes e servidores podem evoluir de forma independente (respeitando-se algumas limitações descritas na documentação do protocolo). Hoje, o Thrift é um projeto open source mantido pela fundação Apache.

BOX 2. IDLs

IDLs são linguagens especializadas que servem para criar definições abstratas de serviços e tipos complexos como enumerados (assim como o enum do Java), estruturas (assim como o “struct” da linguagem C/C++), exceções, etc. A IDL utilizada pelo Finagle faz parte do framework Thrift, assim como o protocolo (veja a seção [Links](#)), e pode ser utilizada para gerar classes que facilitam a chamada a serviços Finagle para a maioria das linguagens de programação, como Java, C++, Python, C#, entre outras.

Em uma IDL encontramos os elementos necessários para definição de abstrações que serão mapeadas para as linguagens de programação, a saber:

- **namespace:** corresponde ao nome do pacote na linguagem Java;
- **struct:** utilizado normalmente para definir estruturas de dados que podem ser passadas por parâmetro ou como retorno de serviços;
- **enum:** corresponde ao enum do Java;
- **exception:** define o contrato de erros que podem ser lançados pelo serviço. É o equivalente às Exceptions do Java;
- **service:** é o elemento principal. É através dele que criamos o comportamento e as operações que serão expostas aos clientes e se parecem muito com as interfaces Java e seus métodos abstratos.

Outra peça importante da arquitetura é o Servidor e sua função é apenas garantir que uma instância de **Service** seja corretamente criada e configurada e esteja disponível para receber requisições. É através de servidores que configuramos aspectos como uso de monitoramento, logging, filtros, protocolos e outros aspectos técnicos.

Filtros são outra abstração importantíssima, pois servem para alterar o comportamento de clientes e serviços de maneira isolada. Os filtros devem implementar a interface `com.twitter.finagle.Filter` e apresentam também um comportamento funcional, tal como os serviços. Eles transformam e enriquecem requisições, podendo alterar ou adicionar parâmetros, converter valores, fazer validações e normalmente são utilizados para criar mecanismos de controle de erros, cache, segurança, entre outros.

Na **Figura 1** visualizamos como clientes, filtros e servidores interagem entre si para o atendimento de requisições. Uma característica importante de um filtro é que ele age de forma desacoplada de um servidor/serviço, o que traz grande flexibilidade para a composição e reaproveitamento de novos serviços.

Twitter Finagle: Desenvolvendo sistemas distribuídos



Figura 1. Integração entre elementos da arquitetura Finagle

De volta para o Future

Desde o Java 5 existe uma interface desconhecida e desprezada pela maioria dos desenvolvedores, chamada **Future** (`java.util.concurrent.Future`). Em termos simples, um **Future** representa o resultado de um processamento realizado em outra thread, ou seja, é através dele que podemos obter o valor resultante de um processamento assíncrono. Além disso, em caso de erro, também podemos recuperar as exceções. O Finagle baseia sua arquitetura nesse mesmo conceito de **Future**, mas utilizando outra classe, do tipo `com.twitter.util.Future`.

Esta classe **Future** oferece uma API bem mais completa ao desenvolvedor, pois permite que sejam aplicadas operações quando o resultado de um **Future** estiver pronto ou permite que vários resultados de processamento assíncronos sejam combinados para se tornarem um único resultado.

É possível, por exemplo, utilizar objetos **Future** para orquestrar a chamada a serviços remotos e usar seus métodos para combinar o resultado em um único retorno. Outra possibilidade é associar a um **Future** operações de transformação, tais como `map` ou `flatMap`, que são disparadas apenas quando o **Future** está completo, isto é, tem algum resultado. Mais à frente veremos na prática o uso de operações de transformação da classe `com.twitter.util.Future`, como `transformedBy`.

Finagle na prática

Nesse artigo demonstraremos como criar serviços, servidores, filtros e Futures a partir de uma pequena aplicação que consulta o serviço de fotos online Instagram e mostra as imagens obtidas no navegador de internet. Para isso será necessário construir dois serviços: o primeiro para encapsular o acesso ao Instagram, chamado de “ImageService”; e o segundo, chamado de “Foto Service”, para expor uma interface HTTP para consulta às imagens fornecidas por “ImageService”. A Figura 2 mostra a visão geral da arquitetura.

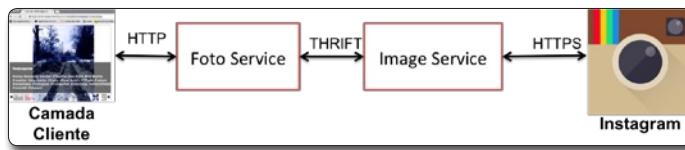


Figura 2. Visão geral da arquitetura da aplicação

A solução proposta é constituída de vários componentes que permitem ao usuário visualizar fotos do Instagram através de serviços do Finagle, a saber:

- **Aplicação Web (HTML + JavaScript)**: O browser interage com o serviço Foto Service através de uma interface HTML + JavaScript

que primeiro autentica o usuário da aplicação no Instagram e em seguida mostra as fotos obtidas;

- **Foto Service**: esse serviço Finagle expõe operações de autenticação no Instagram e consulta a fotos utilizando como transporte o protocolo HTTP. O Foto Service interage com o serviço Finagle “Image Service” através do protocolo *thrift*, que é mais eficiente para o envio de grandes quantidades de informação por ser binário e sua serialização ser mais eficiente;

- **Image Service**: esse serviço serve de fachada para acesso à API de consulta ao Instagram. Ele oferece um contrato definido em IDL Thrift e com operações simples de autenticação e consulta a imagens;

- **Instagram**: serviço de armazenamento de fotos mundialmente famoso. Utilizamos uma API chamada JInstagram (veja a seção **Links**) para fazer a consulta ao catálogo de fotos e usuários.

Obtendo o Finagle

A maneira mais fácil de utilizar o Finagle em qualquer projeto é através de alguma ferramenta de gerenciamento de dependências, tais como o Apache Maven e o Gradle. Para os exemplos desse artigo, faremos uso do Maven e do JDK 8; portanto, será necessário que o ambiente de desenvolvimento tenha instaladas essas ferramentas (veja a seção **Links**). Visto que adotaremos o Maven para gerenciar a build, criaremos o arquivo de projeto `pom.xml`, descrito na **Listagem 1**. Este será responsável, dentre outras coisas, por fazer o download de todos os pacotes básicos para construção e execução dos serviços e servidores.

Observando atentamente o `pom.xml`, podemos encontrar referências para o `finagle-thrift_2.10` e para o `finagle-http_2.10`, que fazem parte do “core” da API do Finagle (veja as linhas 40 e 46). O número 2.10 refere-se à versão da linguagem Scala compatível com a versão do Finagle a ser utilizada.

Dando continuidade à análise do `pom.xml`, observamos na linha 78 o uso de um plugin do Maven chamado “Scrooge”. Este plugin foi desenvolvido pelos engenheiros do Twitter e pela comunidade justamente para gerar o código de serviços e clientes baseado na API do Finagle, a partir de arquivos IDL do projeto, possibilitando aos programadores se concentrar apenas nas regras de negócio.

Na configuração do plugin Scrooge (linhas 79 à 107) determinamos que todo o código fonte deve ser gerado na pasta `target/generated-sources/scrooge`. Posteriormente, faremos referência às classes geradas para implementar os servidores e clientes da solução.

Com o arquivo `pom.xml` pronto, podemos utilizá-lo para gerar o projeto Eclipse através do comando: `mvn eclipse:eclipse`. A versão sugerida do IDE Eclipse para esse projeto é a chamada “Luna”, que tem ótimo suporte ao Java 8.

Nota:

Visando a simplicidade, todos os serviços, servidores, clientes e demais artefatos da solução serão criados em um projeto único no IDE, contudo, em implementações reais, cada serviço e servidor e seus respectivos clientes devem ter seu código fonte mantido em estruturas separadas, o que trará mais organização e independência para as equipes que darão manutenção ao sistema.

Listagem 1. Arquivo pom.xml para setup do projeto Maven.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <project
03   <modelVersion>4.0.0</modelVersion>
04   <groupId>br.com.javamagazine.finagle</groupId>
05   <artifactId>finagle_java_example</artifactId>
06   <version>1.0</version>
07   <repositories>
08     <repository>
09       <id>twitter</id>
10      <url>http://maven.twimg.com/</url>
11    </repository>
12  </repositories>
13  <dependencies>
14    <dependency>
15      <groupId>com.sachinhandiekar</groupId>
16      <artifactId>jInstagram</artifactId>
17      <version>1.0.9</version>
18    </dependency>
19    <dependency>
20      <groupId>com.twitter</groupId>
21      <artifactId>finagle-thrift_2.10</artifactId>
22      <version>6.24.0</version>
23    </dependency>
24    <dependency>
25      <groupId>com.twitter</groupId>
26      <artifactId>finagle-http_2.10</artifactId>
27      <version>6.24.0</version>
28    </dependency>
29  </dependencies>
30  <build>
31    <resources>
32      <resource>
33        <directory>target/generated-sources/scrooge</directory>
34    </resource>
35      <resource>
36        <directory>src/main/resource</directory>
37      </resource>
38    </resources>
39    <plugins>
40      <!-- Generate Finagle scala classes for thrift definitions -->
41      <plugin>
42        <groupId>com.twitter</groupId>
43        <artifactId>scrooge-maven-plugin</artifactId>
44        <version>3.14.1</version>
45        <configuration>
46          <language>java</language> <!-- default is scala -->
47          <thriftOpts>
48            <thriftOpt>--finagle</thriftOpt>
49          </thriftOpts>
50          <outputDirectory>target/generated-sources</outputDirectory>
51          <dependencyIncludes>
52            <include>event-logger-thrift</include>
53          </dependencyIncludes>
54        </configuration>
55        <executions>
56          <execution>
57            <id>thrift-sources</id>
58            <phase>generate-sources</phase>
59            <goals>
60              <goal>compile</goal>
61            </goals>
62          </execution>
63        </executions>
64      </plugin>
65    </plugins>
66  </build>
67 </project>
```

Primeiro serviço: Instagram Image Service

O primeiro serviço que criaremos, chamado **InstagramImageService** (vide **Listagem 2**), será o que permite a consulta de imagens no Instagram. Ele será o único da solução a ser definido com o uso de uma IDL, descrita no arquivo *imageservice.thrift*, para conseguirmos interagir com ele da forma mais rápida e otimizada possível através do protocolo Thrift.

O serviço **InstagramImageService** disponibiliza as seguintes operações que podem ser chamadas remotamente: **getAuthorizationUrl()** e **getListImages()**. Ao entrar na aplicação, o usuário que não está autenticado no Instagram será direcionado para a URL retornada por **getAuthorizationUrl()**. Caso forneça usuário e senhas corretos, uma chave única identificando a sessão atual será retornada pelo Instagram e posteriormente passada por parâmetro para **getListImages()**, que retornará uma lista de fotos a serem mostradas no navegador do usuário autenticado.

Todas as operações e classes que compõem o serviço **InstagramImageService** são definidos no arquivo *imageservice.thrift* (vide **Listagem 3**), que deve estar na pasta *src/main/thrift* para que o plugin Scrooge funcione corretamente. Uma vez executado o comando do Maven *mvn clean compile*, esse plugin será invocado e gerará várias classes (**ExceptionType**, **ImageInfo**, **ImageService** e **ImageServiceException**) que correspondem exatamente aos tipos definidos na IDL.

A classe **ImageService**, gerada pelo plugin Scrooge, é uma agregadora de classes e interfaces que facilita a vida do desenvolvedor ao abstrair o uso da API do Finagle, detalhes de protocolo e tratamentos de erro. Ela também contém uma classe interna chamada **Service**, que estende a classe **com.twitter.finagle.Service**. Esse serviço não possui nenhuma regra de negócio, mas tem a capacidade de transformar requisições representadas por arrays de bytes (**byte[]**) em chamadas a métodos da interface **ImageService.ServiceIface**.

A classe **InstagramImageService**, que contém as regras de negócio, deve implementar a interface **ImageService.ServiceIface**, permitindo assim a união entre o código gerado e o código criado pelo programador.

A partir do diagrama de classes apresentado na **Figura 3**, podemos ver o relacionamento entre as classes que foram geradas (**Service**, **ServiceIface** e **ServiceToClient**) e as classes que devem ser criadas pelo desenvolvedor (**ImageServer** e **InstagramImageService**). Estas cinco classes são analisadas a seguir:

- **InstagramImageService**: implementa o contrato definido por **ImageService.ServiceIface** e é mantida pelo desenvolvedor;
- **ImageService.ServiceIface**: interface gerada automaticamente pelo plugin Scrooge e contida na classe **ImageService**. Não deve ser alterada manualmente, pois em caso de uma nova execução do plugin, qualquer alteração será perdida;

Twitter Finagle: Desenvolvendo sistemas distribuídos

Listagem 2. Código da classe InstagramImageService.

```
01 package br.com.javamagazine.finagle.service;
02 import java.util.*;
03 import org.jinstagram.instagram;
04 import org.jinstagram.auth.InstagramAuthService;
05 import org.jinstagram.auth.model.*;
06 import org.jinstagram.auth.oauth.InstagramService;
07 import org.jinstagram.entity.common.*;
08 import org.jinstagram.entity.users.feed.MediaFeedData;
09 import br.com.javamagazine.finagle.*;
10 import com.twitter.util.*;
11
12 public class InstagramImageService implements ImageService.ServiceInterface {
13     private static final String CLIENT_ID = "SUBSTITUA PELO SEU CLIENT ID";
14     private static final String CLIENT_SECRET = "SUBSTITUA PELO SEU SECRET";
15     private static final String REDIRECT_URI = "http://127.0.0.1:9797";
16     private static final String INSTAGRAM_TAG = "nature";
17     private static final Token EMPTY_TOKEN = null;
18
19     public Future<String> getAuthorizationUrl() {
20         return Future.value(getInstagramService().getAuthorizationUrl(EMPTY_TOKEN));
21     }
22
23     public Future<List<ImageInfo>> getListImages(String key) {
24         Verifier verifier = new Verifier(key);
25         Future<Token> value = Future.value(getInstagramService().getAccessToken(
26             EMPTY_TOKEN, verifier));
27         return value.transformedBy(new FutureTransformer<Token, List<ImageInfo>>() {
28             public Future<List<ImageInfo>> flatMap(Token token) {
29                 final Instagram i = new Instagram(token);
30                 List<ImageInfo> resultList = new ArrayList<ImageInfo>();
31                 try {
32                     for(MediaFeedData media : i.getRecentMediaTags(INSTAGRAM_TAG).getData()) {
33                         Images images = media.getImages();
34                         Caption caption = media.getCaption();
35                         if (caption == null)
36                             continue;
37                         ImageData standardResolution = images.getStandardResolution();
38                         ImageInfo ii = new ImageInfo();
39                         ii.setImageText(caption.getText());
40                         ii.setNomeUsuario(caption.getFrom().getFullName());
41                         ii.setImageUrl(standardResolution.getImageUrl());
42                         resultList.add(ii);
43                     }
44                 } catch (Throwable e) {
45                     e.printStackTrace();
46                     return Future.exception(new ImageServiceException(e.getMessage(),
47                         ExceptionType.INTEGRATION));
48                 }
49                 return Future.value(resultList);
50             }
51
52             public Future<List<ImageInfo>> rescue(Throwable t) {
53                 return Future.exception(new ImageServiceException(t.getMessage(),
55                     ExceptionType.AUTH));
56             }
57         });
58     }
59
60     public static InstagramService getInstagramService() {
61         InstagramService service = new InstagramAuthService().apiKey(CLIENT_ID)
62             .apiSecret(CLIENT_SECRET).callback(REDIRECT_URI).build();
63         return service;
64     }
65 }
66 }
```

Listagem 3. Arquivo descriptor de IDL imageservice.thrift necessário ao InstagramImageService.

```
namespace java br.com.javamagazine.finagle

struct ImageInfo {
    1: required string imageUrl;
    2: required string imageText;
    3: required string nomeUsuario;
}

enum ExceptionType {
    AUTH,
    INTEGRATION
}

exception ImageServiceException {
    1: string message,
    2: ExceptionType type;
}

service ImageService {

    string getAuthorizationUrl(),
    list<ImageInfo> getListImages(string key) throws (1: ImageServiceException
        isException);
}
```

- **Service:** classe gerada automaticamente pelo plugin Scrooge que contém toda a complexidade referente à serialização de objetos, tratamento de erros, conversões para arrays de bytes, etc.;
- **ImageServer:** Implementação do servidor que responderá às requisições dos clientes, instanciando e invocando o serviço **InstagramImageService**;
- **ServiceToClient:** Cliente gerado automaticamente pelo plugin Scrooge para acesso remoto ao serviço **InstagramImageService**. Essa classe implementa a interface **ImageService.ServiceIface**, a mesma interface do serviço.

Na classe **InstagramImageService**, que concentra toda a lógica de integração ao site de imagens Instagram, dois métodos (**getAuthorizationUrl()** e **getListImages()**) serão implementados. Para abstrair possíveis complexidades de autorização e uso da API REST para consulta a fotos disponibilizada por esse site, utilizaremos o framework JInstagram.

Como as APIs remotas do Instagram não são de acesso livre, é necessário que o desenvolvedor obtenha os códigos de acesso, disponibilizados em forma de duas chaves: "client id" e "client secret". Para isso, o desenvolvedor deve ir ao site de desenvolvedores

do Instagram (veja a seção **Links**) e solicitar o registro da sua aplicação (conforme a **Figura 4**).

Um ponto de atenção durante esse registro é a informação *REDIRECT URI*, que deve apontar para o endereço `http://127.0.0.1:9797`. Esse endereço aponta para o servidor HTTP Finagle que ainda iremos construir e serve para o Instagram retornar o status de sucesso caso o usuário tenha entrado com as credenciais de login e senha corretos.

Caso a aplicação seja registrada com sucesso, será apresentada uma tela com as informações *CLIENT ID* e *CLIENT SECRET* (vide **Figura 5**). O valor desses campos deve ser informado no código fonte da classe **InstagramImageService**. Mais especificamente nas linhas 13 e 14 da **Listagem 2**.

O método `getAuthorizationUrl()`, implementado a partir da linha 33, tem a função de retornar o endereço de autenticação que o Instagram usará para autorizar os clientes da aplicação (por exemplo: `https://instagram.com/auth`). O retorno desse método deve ser encapsulado em um **Future** através do método `Future.value()`. Quando fazemos isso, estamos indicando ao Finagle que ele deve utilizar outra thread para processar a lógica de consulta à URL de autorização (veja a linha 20 da **Listagem 2**), liberando a thread principal na qual o serviço **InstagramImageService** está sendo executado para atender a mais requisições.

A implementação do método `getListImages()`, iniciado a partir da linha 38, apresenta uma série de novidades, demonstrando melhor a aplicabilidade da classe **Future**. Inicialmente criamos uma instância de `Future<Token>`, o que na prática significa que o objeto **Token** (que representa um identificador de sessão do Instagram) será criado assincronamente por outra thread.

Quando o objeto **Token** for criado de forma assíncrona da chamada ao site Instagram, ele será transformado em um objeto do tipo `List<ImageInfo>`, que representa uma lista de informações de fotos. Essa transformação é realizada pela operação de **Future** chamada `transformedBy()` (vide

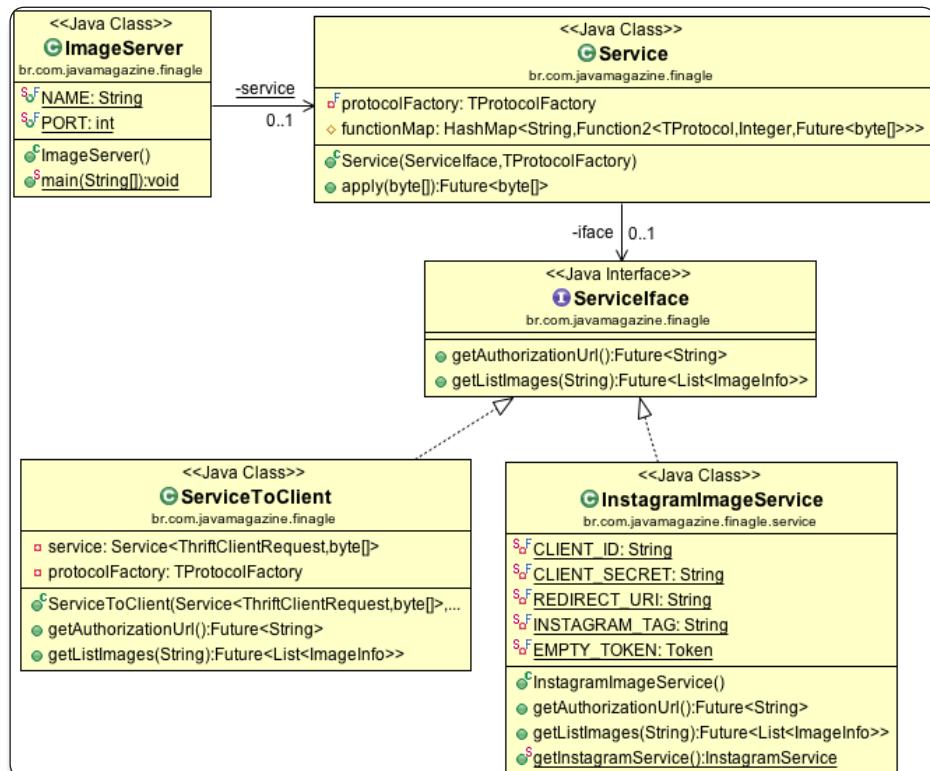


Figura 3. Diagrama de classes retratando as classes geradas e as classes implementadas.

The first version of the Instagram API is an exciting step forward towards making it easier for users to have open access to their data. We created it so that you can surface the amazing content Instagram users share every second, in fun and innovative ways.

Build something great.

[Register Your Application](#) then [dive into the documentation](#)

Figura 4. Registro de aplicações no site do Instagram para desenvolvedores

linha 42). Em caso de erro, uma exceção será retornada (linha 53).

O método `transformedBy()` recebe como parâmetro uma instância de `com.twitter.util.FutureTransformer`, que é responsável por tratar situações de sucesso e de erro na execução de Futures. De acordo com a documentação, o método `flatMap()`, da

classe `FutureTransformer`, sempre é acionado quando o processamento é finalizado com sucesso, enquanto `rescue()` é acionado quando ocorre algum erro.

Uma vez que o serviço **InstagramImageService** esteja completo, precisamos criar um servidor, chamado **ImageServer** (**Listagem 4**), para permitir que clientes

Twitter Finagle: Desenvolvendo sistemas distribuídos

remotos o acessem. Em termos gerais, ele associa a instância de **InstagramImageService** a um socket, utilizando para isso a API específica do Finagle, **ServerBuilder** (vide linha 27).

A classe **ServerBuilder**, que faz parte da API do Finagle, é um utilitário para configuração de Servidores. É através dela que parametrizamos diversas informações como endereço e porta onde o servidor será disponibilizado, nível de Log (DEBUG, INFO, etc.), informações de protocolos de transporte, detalhes de monitoramento, entre outros.

Listagem 4. Implementação de ImageServer.

```
01 package br.com.javamagazine.finagle;
02
03 import java.net.InetSocketAddress;
04 import java.util.concurrent.TimeUnit;
05
06 import org.apache.thrift.protocol.TBinaryProtocol;
07
08 import br.com.javamagazine.finagle.ImageService.Service;
09 import br.com.javamagazine.finagle.service.InstagramImageService;
10
11 import com.twitter.finagle.builder.Server;
12 import com.twitter.finagle.builder.ServerBuilder;
13 import com.twitter.finagle.thrift.ThriftServerFramedCodec;
14 import com.twitter.util.Duration;
15
16 public class ImageServer {
17     public static final String NAME = "ImageService";
18     public static final int PORT = 7810;
19     private static Service service;
20
21     public static void main(String[] args) {
22
23         ImageService.Serviciface processor = new InstagramImageService();
24         service = new ImageService.Service(processor,
25             new TBinaryProtocol.Factory());
26
27         final Server server = ServerBuilder.safeBuild(service,
28             ServerBuilder.get().name(NAME).codec(ThriftServerFramedCodec.get())
29             .bindTo(new InetSocketAddress(PORT)));
30
31         System.out.println(String.format("Servidor %s executando...", NAME));
32         Runtime.getRuntime().addShutdownHook(new Thread() {
33             public void run() {
34
35                 System.out.println("Terminando server " + NAME);
36
37                 if (server != null) {
38                     server.close(new Duration(TimeUnit.SECONDS.toNanos(10)));
39                 }
40             }
41         });
42     }
43 }
```

Segundo Serviço: Foto Service

O próximo serviço que compõe a solução é o responsável por expor ao usuário a interface HTML/HTTP. No entanto, ao contrário de **InstagramImageService**, ele não é criado a partir da geração de código de uma IDL, mas sim de forma manual. Sendo assim, a classe que criaremos será a **MyHttpService**.

Por ser um serviço Finagle, **MyHttpService** deve estender a classe **com.twitter.finagle.Service** e implementar o método **apply()**, cuja lógica nesse caso será de direcionar o processamento para uma função *handler* escolhida mediante o parâmetro **path** extraído da requisição do usuário.

A classe servidora que contém e inicializa o serviço **MyHttpService** é **FotoHttpServer2**. Esta associa cada URL exposta pelo servidor HTTP a uma implementação de **java.util.function.Function**, que no nosso exemplo chamaremos de *handler*. Os handlers que fazem parte de **FotoHttpServer2** são **loadResource**, **ackUsuario**, **createGaleria** e **autorizarUsuario** (apresentados nas Listagens 5 a 8).

Listagem 5. Função para processamento de arquivos HTML.

```
25 private static final Function<String, Function<HttpRequest, Future<HttpServletResponse>>>
26     loadResource = rname -> request -> {
27         HttpResponse htmlResult = null;
28         try {
29             htmlResult = response(request, HttpStatus.OK);
30             QueryStringDecoder queryDecoder = new QueryStringDecoder(
31                 request.getUri());
32             Map<String, List<String>> parameters = queryDecoder.getParameters();
33             String s = IOUtil.parseTemplate(rname, parameters);
34             htmlResult.setContent(ChannelBuffers.copiedBuffer(s.getBytes()));
35             htmlResult.headers().add("Content-Type", IOUtil.mimeType(rname));
36         } catch (Throwable e) {
37             htmlResult = new DefaultHttpResponse(request.getProtocolVersion(),
38                 HttpStatus.NOT_FOUND);
39         }
40     }
41
42     return Future.value(htmlResult);
43 }
```

Listagem 6. Função para processamento de requisição feita pelo Instagram.

```
42 private static final Function<HttpRequest, Future<HttpResponse>>
43     ackUsuario = (
44         request) -> {
45         DefaultHttpResponse response = null;
46         QueryStringDecoder queryDecoder = new QueryStringDecoder(
47             request.getUri());
48         List<String> list = queryDecoder.getParameters().get("code");
49         if (list != null && list.size() > 0) {
50             String code = list.get(0);
51             response = response(request, HttpStatus.TEMPORARY_REDIRECT);
52             response.headers().add("Location", "/index.html?id=" + code);
53         } else {
54             response = response(request, HttpStatus.FORBIDDEN);
55         }
56     }
57
58     return Future.value(response);
59 }
```



Figura 5. Aplicação registrada no Instagram

Listagem 7. Função para criação da galeria de fotos.

```
61 private static final Function<HttpRequest, Future<HttpResponse>>
62     createGaleria = (
63         request -> {
64             QueryStringDecoder queryDecoder = new QueryStringDecoder(
65                 request.getUri());
66             String code = queryDecoder.getParameters().get("id").get(0);
67
68             return imageClient().getListImages(code)
69                 .transformedBy(new FutureTransformer<List<ImageInfo>, HttpResponse>() {
70
71                 public Future<HttpResponse> flatMap(List<ImageInfo> list) {
72                     final StringBuilder sb = new StringBuilder();
73                     sb.append("<ul>");
74                     list.forEach(media -> {
75                         sb.append(String.format(
76                             "<li><img src=\"%s\" alt=\"%s\" title=\"%s\" /></li>",
77                             media.getImageUrl(), media.getImageText(),
78                             media.getNameUsuario())));
79                     });
80                     DefaultHttpResponse res = response(request, HttpStatus.OK);
81                     res.setEntity(ChannelBuffers.copiedBuffer(String.valueOf(sb).getBytes()));
82                     res.headers().add("Content-Type", "text/html");
83                     return Future.value(res);
84
85                 public Future<HttpResponse> rescue(Throwable err) {
86                     LOG.error("Erro ao consultar imageservice", err);
87                     HttpResponse response = response(request, HttpStatus.BAD_REQUEST);
88                     if (err instanceof ImageServiceException) {
89                         ImageServiceException imageEx = ImageServiceException.class
90                             .cast(err);
91                         switch (imageEx.getType()) {
92                             case AUTH:
93                                 LOG.error("Erro de autenticacao", imageEx);
94                                 response = response(request, HttpStatus.TEMPORARY_REDIRECT);
95                                 response.headers().add("Location", "/error");
96                             break;
97                         default:
98                             break;
99                         }
100                     }
101                     return Future.value(response);
102                 }
103             });
104 }
```

Listagem 8. Função para autorização do usuário.

```
105 private static final Function<HttpRequest, Future<HttpResponse>>
106     autorizarUsuario = 106 (request) -> {
107         return imageClient().getAuthorizationUrl().transformedBy(
108             new FutureTransformer<String, HttpResponse>() {
109
110             public Future<HttpResponse> flatMap(String s) {
111                 DefaultHttpResponse r = response(request, HttpStatus.FOUND);
112                 r.headers().add("Location", s);
113                 return Future.value(r);
114             }
115
116             public Future<HttpResponse> rescue(Throwable t) {
117                 HttpResponse r = response(request, HttpStatus.TEMPORARY_REDIRECT);
118                 r.headers().add("Location", "/error");
119                 return Future.value(r);
120             }
121
122         });
123     }
```

Para entendermos melhor o relacionamento entre URLs e Handlers, vejamos a URL `css/primeui-1.1-min.css` (na linha 128 da **Listagem 9**). Quando o browser invocar essa URL, o handler `loadResource`, que está relacionado a essa URL através do mapa `R` (veja a linha 125), será chamado no método `apply()` da classe `MyHttpService`, executando a lógica do handler selecionado.

Essa organização foi adotada por apresentar algumas vantagens, a saber:

- Simplicidade:** A implementação de `MyHttpService` ficou bastante simples, pois delega toda a lógica para os handlers. Com essa solução é possível adicionar facilmente mais URLs e mais handlers;
- Coesão:** Os handlers são independentes entre si. Cada um deve ser responsável por resolver apenas o problema para o qual foi criado;
- Reaproveitamento e composabilidade:** Por serem instâncias de `java.util.function.Function`, novidade do Java 8, novos handlers podem ser criados a partir dos existentes, utilizando-se para isto os métodos default `compose()` ou `andThen()`, que existem nessa interface.

O serviço `MyHttpService` deverá fazer várias chamadas às operações do serviço `InstagramImageService`. Para isso, utilizaremos uma classe que simplifica e facilita esse acesso chamada `ImageService.ServiceToClient`. Essa classe pode ser instanciada através do uso do método `create()` (linha 164). Esse método recebe como parâmetros o endereço e a porta do serviço `InstagramImageServer` ao qual o cliente vai se conectar, além da quantidade máxima de conexões que esse mesmo cliente pode estabelecer com o servidor.

O handler mais interessante sem dúvida é o `createGaleria` (linha 61), que invoca a operação `getListImages()` do cliente e transforma o resultado obtido em um `Future<HttpResponse>` utilizando o nosso já conhecido `transformedBy()`.

Quando o método `getListImages()` é invocado (linha 66), o Finagle se encarrega de criar ou reaproveitar outra thread para executar a chamada ao serviço remoto e uma instância de `Future<List<ImageInfo>>` é disponibilizada imediatamente ao programa principal. Note que não sabemos exatamente se a chamada ao serviço remoto retornará o resultado desejado ou algum tipo de erro, mas isso não importa, pois associado ao `Future` está uma instância de `FutureTransformer` que certamente fará o processamento do sucesso ou falha.

Por último, utilizamos o método `main()` (linha 153 em diante) para instanciar a classe `MyHttpService`, permitindo que o servidor HTTP possa finalmente receber requisições no endereço `http://localhost:9797`.

Visão geral do funcionamento da solução

O funcionamento geral da solução pode ser melhor entendido através do diagrama da **Figura 6**. A descrição das interações é apresentada a seguir:

Twitter Finagle: Desenvolvendo sistemas distribuídos

1. O usuário abre o browser e acessa o endereço `http://localhost:9797/instagram`. Esse é o endereço da página inicial da aplicação;
2. O handler `autorizarUsuario` é invocado e é feita uma chamada à operação remota `getAuthorizationUrl()`, do serviço `InstagramImageService` (que está respondendo a solicitações em `localhost:7810`). Esse handler fará o redirecionamento do browser para a URL retornada com a chamada desse serviço;
3. A operação `getAuthorizationUrl()` contata o Instagram utilizando a API JInstagram para obter a URL de autenticação, onde o usuário deverá fornecer suas credenciais;
4. O usuário é redirecionado para a URL do Instagram a fim de fornecer seu login e senha;
5. O usuário entra com seu usuário e senha;
6. O site do Instagram autentica o usuário e caso essa operação seja realizada com sucesso, o próprio Instagram faz uma chamada de volta ao nosso servidor HTTP, no endereço `http://localhost:9797/`;
7. O handler `ackUsuario`, associado à URL `http://localhost:9797/code?`, é invocado e extrai o parâmetro `code`, que representa a sessão atual válida e que dá acesso às fotos;
8. O usuário da aplicação é redirecionado para a página `index.html`;
9. O handler `loadResource` faz o acesso ao disco do servidor para carregar em memória o conteúdo do arquivo `index.html`;
10. Quando a página `index.html` é mostrada no browser, ela solicita que o endereço `/resource/content.html?id=` seja carregado pelo servidor;
11. O handler associado ao endereço do passo 10, `createGaleria`, faz a chamada à operação remota `getListImages()` no serviço `InstagramImageService` e finalmente as fotos são retornadas para compor o conteúdo de `index.html`.

O resultado final mostrado ao usuário pode ser visualizado na [Figura 7](#).

Aplicação HTML

Para não prolongar o artigo e manter o foco no Finagle, a interface HTML construída para visualização das fotos obtidas do Instagram é extremamente simples e tem seu conteúdo dinâmico

Listagem 9. Exemplo de implementação de Servidor HTTP Finagle.

```
01 package br.com.javamagazine.finagle;
02
03 import java.util.*;
04 import java.util.concurrent.ConcurrentHashMap;
05 import java.util.function.Function;
06
07 import org.jboss.netty.buffer.ChannelBuffers;
08 import org.jboss.netty.handler.codec.http.*;
09 import org.slf4j.*;
10
11 import br.com.javamagazine.finagle.client.ImageServiceClient;
12
13 import com.twitter.finagle.*;
14 import com.twitter.util.*;
15 import com.twitter.util.TimeoutException;
16
17 public class FotoHttpServer2 {
18
19 private static final Logger LOG = LoggerFactory.getLogger(FotoHttpServer2.class);
20
21 private static DefaultHttpResponse response(HttpRequest request,
22     HttpResponseStatus status) {
23     return new DefaultHttpResponse(request.getProtocolVersion(), status);
24 }
25
26
27 private static ImageService.ServiceToClient imageClient() {
28     return ImageServiceClient.create("localhost", ImageServer.PORT, 10);
29 }
30 /**
31 * Local onde ficam os handlers apresentados nas Listagens 6, 7, 8 e 9.
32 */
33
34 private static final Map<String, Function
35 <HttpRequest, Future<HttpResponse>>> R = 126
36     new ConcurrentHashMap<String, Function
37 <HttpRequest, Future<HttpResponse>>>();
38
39 static {
40     R.put("/css/primeui-1.1-min.css",
41         loadResource.apply("/primeui-1.1-min.css"));
42     R.put("/js/primeui-1.1-min.js",
43         loadResource.apply("/primeui-1.1-min.js"));
44     R.put("/?code=", ackUsuario);
45
46     R.put("/index.html?id=", loadResource.apply("/index.html"));
47     R.put("/instagram", autorizarUsuario);
48     R.put("/resource/content.html?id=", createGaleria);
49     R.put("/error", loadResource.apply("/err.html"));
50 }
51
52
53 static class MyHttpService extends Service<HttpRequest, HttpResponse> {
54     @Override
55     public Future<HttpResponse> apply(HttpRequest request) {
56         for (Map.Entry<String, Function<HttpRequest, Future<HttpResponse>>> entry : R
57             .entrySet()) {
58             if (request.getUri().startsWith(entry.getKey())) {
59                 return entry.getValue().apply(request);
60             }
61         }
62         return Future.value(response(request, HttpStatus.NOT_FOUND));
63     }
64
65     public static void main(String[] args) {
66         final ListeningServer server = Http.serve(":9797", new MyHttpService());
67         try {
68             Await.ready(server);
69         } catch (TimeoutException e) {
70             e.printStackTrace();
71         } catch (InterruptedException e) {
72             e.printStackTrace();
73         }
74     }
75
76     public static ServiceToClient create(String host, int port, int connectionLimit) {
77         Service<ThriftClientRequest, byte[]> service = ClientBuilder
78             .safeBuild(ClientBuilder.get())
79             .hosts(new InetSocketAddress(host, port)).codec(ThriftClientFramedCodec.get())
80             .hostConnectionLimit(connectionLimit);
81         ServiceToClient serviceToClient =
82             new ImageService.ServiceToClient(service, new TBinaryProtocol.Factory());
83         return serviceToClient;
84     }
85 }
```

servido pelo serviço **MyHttpService**, através de chamadas ao servidor **FotoHttpServer2**.

Uma única página HTML, apresentada na **Listagem 10**, é responsável por fazer a chamada à URL `/resource/content.html` (linha 17). Esta chamada efetua toda a consulta aos servidores, como descrito anteriormente. Para facilitar a construção da tela e diminuir a quantidade de código, foi utilizado o framework JavaScript chamado PrimeUI, que é derivado do framework PrimeFaces, solução bem conhecida pelos desenvolvedores Java EE (veja a seção **Links** para mais informações).

Dito isso, quando a página é carregada, a função JavaScript que faz o carregamento das imagens (linha 15) é automaticamente invocada. Assim que o HTML com a lista de fotos for retornado pelo servidor **FotoHttpServer2**, o conteúdo do layer chamado `#images` será preenchido (linha 18) e a função `puigallery` fará todo o trabalho de dispor as imagens em formato de *slideshow*.

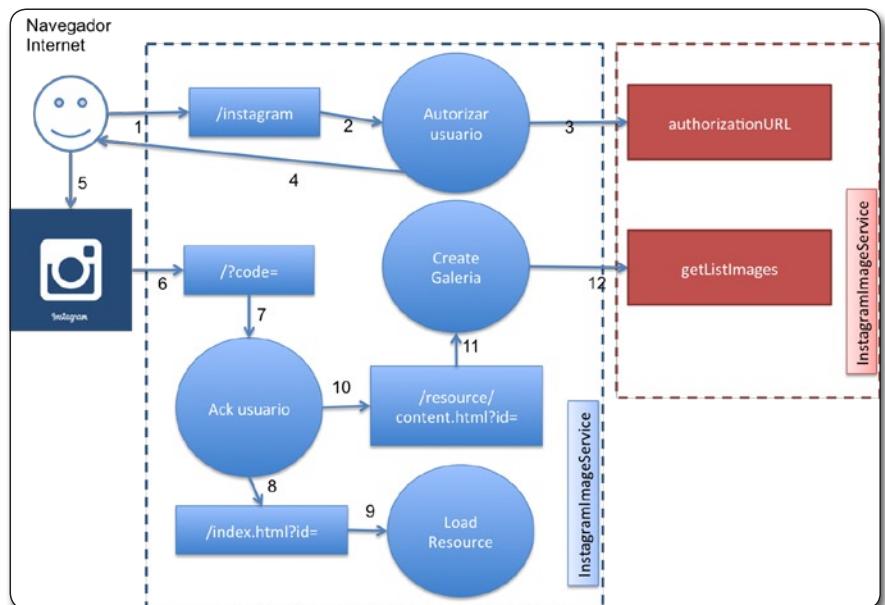


Figura 6. Diagrama de interação entre FotoHttpServer2 e InstagramImageService

Listagem 10. Página HTML para interface com o usuário.

```

01 <html>
02   <head>
03     <link rel="stylesheet"
04       href="https://code.jquery.com/ui/1.9.0/themes/smoothness/
05         jquery-ui.css"/>
06     <link rel="stylesheet" href="/css/primeui-1.1-min.css"/>
07     <script type="text/javascript"
08       src="https://code.jquery.com/jquery-1.8.2.min.js"></script>
09     <script type="text/javascript"
10       src="https://code.jquery.com/ui/1.9.0/jquery-ui.min.js"></script>
11     <script type="text/javascript" src="/js/primeui-1.1-min.js"></script>
12   </head>
13   <body>
14     <script type="text/javascript">
15
16       $(function() {
17         code = '<%=id%>';
18         $('#images').load('/resource/content.html?id=' + code, function() {
19           $('#images').puigallery();
20         });
21       });
22     </script>
23     <div id="images"></div>
24   </body>
25 </html>

```

É fácil perceber o potencial do Finagle como um grande framework para implementação de serviços e servidores, mas não é apenas isso que o torna um dos projetos que mais desperta a atenção da comunidade e das startups. Ao longo dos anos, sua arquitetura extensível permitiu que várias funcionalidades e o suporte a diversos protocolos fossem adicionados.

Como destaques, podemos citar sua compatibilidade com o `kestrel`, serviço de mensageria para JVM extremamente performático, `memcached`, sistema de cache de objetos distribuído normalmente

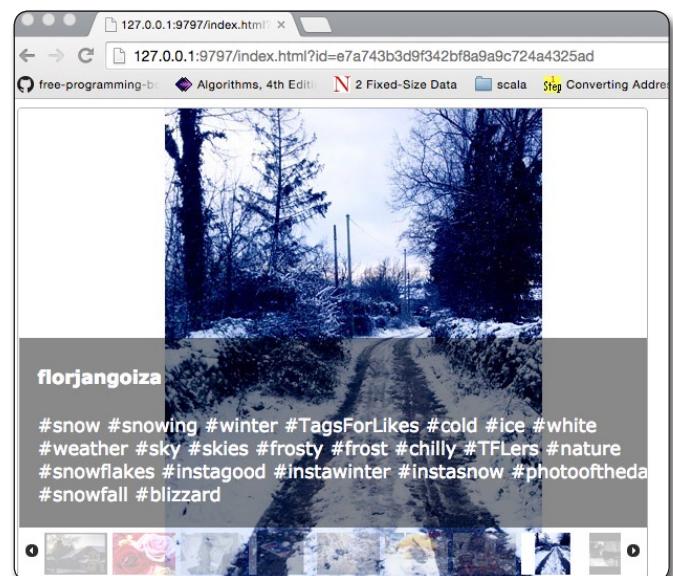


Figura 7. Interface de slide de fotos obtidas do Instagram

presente em soluções de alta performance, e `redis`, um banco de dados NoSQL normalmente usado como cache store. O Finagle possui APIs especializadas para interagir com cada um desses produtos.

Outro ponto importante é o controle sobre o ambiente onde nossos serviços estão sendo executados. Uma vez que o ambiente de produção começa a se tornar mais complexo, passa a ser ainda mais importante manter o controle de como os serviços e servidores estão interagindo entre si, além de acompanhar o desempenho de cada serviço individualmente, através de monitoramento e métricas. Nesse ponto, o Finagle vai além de muitos frameworks de mercado, pois já vem, de fábrica, com total suporte

Twitter Finagle: Desenvolvendo sistemas distribuídos

a rastreabilidade distribuída, através do uso da ferramenta Zipkin, construída pelos engenheiros do Twitter para identificar problemas como latência de rede e lentidão quando existe chamadas e dependência entre serviços.

Com relação ao monitoramento, o framework já define uma série de métricas que podem ser facilmente disponibilizadas para servidores como o Graphite, que é um servidor open source especializado em monitoramento de sistemas operacionais. Ademais, ainda é possível que o programador defina suas próprias métricas, que podem ser utilizadas pelo pessoal de infraestrutura ou pelos gestores e responsáveis pelo negócio para análise do desempenho ou qualidade com que os clientes estão sendo atendidos pelo sistema.

Links:

Integração com os serviços do Instagram.

<http://instagram.com/developer/register/>

JInstagram, API de integração ao Instagram.

<http://jinstagram.org/>

Página oficial do Finagle.

<https://twitter.github.io/finagle/>

Página do projeto Apache Maven.

<http://maven.apache.org>

Thrift: o Guia.

<http://diwakergupta.github.io/thrift-missing-guide/>

Autor



Leonardo Gonçalves da Silva

leogsilva@acm.org

É pós-graduado em Qualidade de Software e atua com arquitetura de sistemas e desenvolvimento na plataforma Java desde 1998. Atualmente trabalha com desenvolvimento Mobile e soluções para Cloud Computing.



Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



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

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

Spring MVC: Construa aplicações responsivas com Bootstrap – Parte 2

Integre o Spring MVC com o Bootstrap

ESTE ARTIGO FAZ PARTE DE UM CURSO

Na primeira parte desta série, analisamos os benefícios proporcionados por frameworks consolidados e líderes de mercado como o Spring MVC e o Bootstrap. Explicamos detalhadamente o funcionamento da biblioteca *web* do Spring, analisamos suas vantagens e vimos na prática como aplicá-lo em um projeto através da criação dos recursos iniciais necessários para o funcionamento do nosso sistema. No que diz respeito ao Bootstrap, também verificamos como ele funciona e comentamos rapidamente sobre alguns dos elementos fornecidos pelo framework, como *grids*, painéis e botões.

Nesta segunda parte vamos explorar o fenômeno da utilização em grande quantidade de dispositivos móveis para navegar em páginas da internet. Por este motivo, os projetistas de bibliotecas de *front-end* tiveram que adaptar seus produtos de maneira que suportassem múltiplos *browsers*. Uma das ferramentas que nasceu com este comportamento foi o Bootstrap, que será bastante explorado nas páginas seguintes, no que diz respeito a *layouts* e componentes. Além dele, também usaremos o framework Tiles, com o objetivo de gerar trechos de código reutilizáveis para nossas páginas JSP. Em nosso projeto, as duas bibliotecas serão integradas ao *back-end* desenvolvido na primeira parte do artigo, finalizando nossa aplicação de clínica médica.

Criando as páginas do sistema

Dando continuidade ao desenvolvimento do sistema, nosso último passo será criar as páginas JSP.

Fique por dentro

A criação de páginas web que sejam compatíveis com layouts responsivos tem sido adotada na maioria dos novos projetos como uma forma de as empresas se adequarem ao crescente número de dispositivos móveis em uso pelos clientes. Sendo assim, nesta segunda parte do artigo vamos conhecer o Bootstrap, um framework que simplifica a construção de interfaces responsivas, e combiná-lo com o back-end do sistema de clínica médica criado utilizando o Spring MVC na primeira parte deste estudo.

Para facilitar essa etapa, em toda a aplicação não teremos validações de campos que são obrigatórios e em cada uma das telas geradas utilizaremos alguns recursos do Bootstrap que serão explicados individualmente, no que diz respeito à utilidade e possíveis customizações. Antes, porém, vamos inserir no projeto a folha de estilo e os arquivos JavaScript necessários para nossa aplicação funcionar corretamente.

Em primeiro lugar, faça o download do Bootstrap a partir do endereço fornecido na seção **Links** e descompacte o arquivo baixado. Em seguida, crie uma pasta *css* dentro de *src/main/webapp* e coloque, no diretório adicionado, o arquivo *bootstrap.min.css* que está no conteúdo descompactado. Agora crie uma pasta *js* dentro de *src/main/webapp* e coloque nela o arquivo *bootstrap.min.js*, que também está no conteúdo descompactado.

Terminados estes dois passos, faça o download do jQuery a partir do endereço fornecido na seção **Links** e coloque o arquivo *jquery-2.1.3.min.js* na pasta *js*. Pronto! Nossos arquivos requeridos para o Bootstrap funcionar já estão nos lugares corretos.

Criando o template

A primeira estrutura que iremos criar é o *template* geral das páginas no sistema. É neste ponto que exploraremos alguns dos

recursos do framework Tiles. O *template* servirá para definirmos alguns trechos comuns a todas as páginas do sistema (como o menu, a *div* que irá mostrar as mensagens de erro e sucesso do sistema, e o painel que envolve o conteúdo de todas as telas). Como dissemos no início do artigo anterior, nossas páginas ficarão dentro de WEB-INF/jsp. Esta estratégia aumenta a segurança, porém apenas os *controllers* têm acesso a elas. Para evitar esta limitação, o *template* e a página inicial ficarão na raiz da pasta web do projeto. Esta abordagem tem duas vantagens: em primeiro lugar podemos abrir a página inicial do sistema diretamente, sem que precisemos fazer redirecionamentos; em segundo, nos sistemas que vão para produção (não é o caso do que estamos desenvolvendo aqui), o Google conseguiria indexar a página em seu buscador, uma vez que ela estaria acessível.

Desta forma, clique com o botão direito em cima de *src/main/webapp*, selecione a opção *New* e depois clique em *JSP File*. Na tela que surge, informe o nome “template.jsp” e clique em *Finish*. O Eclipse gerará um pequeno código que deve ser alterado para ficar igual ao da **Listagem 1**.

Listagem 1. Código da página template.jsp.

```

01 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
02   pageEncoding="ISO-8859-1"%>
03 <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>
04 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
05 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
06   "http://www.w3.org/TR/html4/loose.dtd">
07 <html>
08   <head>
09     <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
10    <title>Clínica JM</title>
11    <link rel="stylesheet" href="${pageContext.request.contextPath}/css/
12      bootstrap.min.css">
13    <script src="${pageContext.request.contextPath}/js/jquery-2.1.3.min.js">
14    </script>
15    <script src="${pageContext.request.contextPath}/js/bootstrap.min.js">
16    </script>
17   </head>
18   <body>
19     <nav class="navbar navbar-default" role="navigation">
20       <div class="container-fluid">
21         <div class="navbar-header">
22           <button type="button" class="navbar-toggle collapsed"
23             data-toggle="collapse" data-target="#navbar-1">
24             <span class="sr-only"></span>
25             <span class="icon-bar"></span>
26             <span class="icon-bar"></span>
27             <span class="icon-bar"></span>
28           </button>
29           <a class="navbar-brand" href="${pageContext.request.contextPath}
30             /index.jsp">Início</a>
31         </div>
32       <div class="collapse navbar-collapse" id="navbar-1">
33         <ul class="nav navbar-nav">
34           <li class="dropdown">
35             <a href="#" class="dropdown-toggle" data-toggle="dropdown"
36               role="button" aria-expanded="false">
37               Médicos<span class="caret"></span></a>
38             <ul class="dropdown-menu" role="menu">
39               <li><a href="${pageContext.request.contextPath}/
40                 preparaCadastroMedico.do">Cadastrar</a></li>
41             </ul>
42           <li><a href="${pageContext.request.contextPath}/medico/
43             listar.do">Listar</a></li>
44         </ul>
45       </div>
46     </nav>
47     <c:if test="${mensagem != null}">
48       <div class="${mensagem.tipoMensagem.classeCss}"
49         role="alert">${mensagem.texto}</div>
50     </c:if>
51   </body>
52 </html>
```

Os principais trechos desse código são:

- Linhas 2 e 3: importa as *taglibs* do Tiles e do core do JTSI;
- Linhas 9 e 11: declara a folha de estilo e o arquivo JavaScript do Bootstrap;
- Linha 10: declara o arquivo JavaScript do jQuery, requerido para que alguns componentes do Bootstrap funcionem, e também utilizado para realizar as chamadas em Ajax que criaremos mais à frente na aplicação. Essa linha deve vir antes da que declara o JavaScript do Bootstrap, caso contrário as páginas apresentarão erro em tempo de execução;
- Linhas 14 a 51: define o menu horizontal do sistema através do componente **nav** do Bootstrap;
- Linhas 17 a 22: insere um botão na div. Este botão é mostrado automaticamente pelo Bootstrap em dispositivos com telas menores;
- Linhas 53 a 55: testa se existe um atributo **mensagem** na requisição. Em caso positivo, exibe a mensagem de erro, sucesso ou aviso na tela utilizando o componente de mensagens do Bootstrap;

- Linhas 57 a 61: declara o componente de painel do Bootstrap e seu corpo;
- Linha 59: o componente **insertAttribute**, do Tiles, declara um possível ponto de substituição no sistema. Isto significa que páginas que usem esta página como *template* podem redefinir o código que está dentro da *tag* que, no nosso caso, mostra o corpo da *view*.

Criando a página inicial

Agora que criamos nosso *template*, vamos elaborar a página inicial do sistema. Assim, crie o arquivo *index.jsp* dentro da pasta *src/main/webapp*. Da mesma maneira que no passo anterior, um código será gerado e deve ser modificado para que fique semelhante ao da **Listagem 2**.

Listagem 2. Código da página index.jsp.

```
01 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
02 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
03 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
04 <html>
05     <tiles:insertDefinition name="template">
06         <tiles:putAttribute name="corpo">
07             <p align="center">Bem vindo. Escolha sua opção logo acima.</p>
08         </tiles:putAttribute>
09     </tiles:insertDefinition>
10 </html>
```

Observe que na linha 5 temos o componente **insertDefinition** do Tiles e nele informamos o atributo **name** com o valor **template**. Lembre-se que no arquivo *tiles.xml* declaramos a página *template.jsp* com o nome *template*. Portanto, é o código desta JSP que será colocado neste trecho em tempo de execução pelo Tiles. Já na linha 6 inserimos a instrução **<tiles:putAttribute />**, que indica que estamos redefinindo um trecho criado com **<tiles:insertAttribute />** (neste caso, o corpo) presente no *template*.

Após salvar o arquivo, vamos rodar a aplicação. Para isso, é necessário que o WildFly já esteja configurado no Eclipse. Caso não tenha o servidor criado na IDE, na seção **Links** há um *post* do blog do autor em que é possível ver como realizar esta etapa. Quando tudo estiver pronto, para executar a aplicação clique com o botão direito em cima da raiz do projeto, selecione *Run As*, depois a opção *Run On Server*, marque o item *WildFly* na tela que surge e pressione o botão *Finish* para confirmar. Após a publicação, a tela inicial do sistema será exibida conforme a **Figura 1**.

Criando as páginas de cadastro e listagem de médicos

Nossa próxima página será a de cadastro de médicos. Esta é nossa primeira página acessível apenas pelos *controllers*. Para gerá-la, primeiro insira uma nova pasta chamada *jsp* dentro de *WEB-INF*. Feito isso, crie o arquivo *cadastrarMedico.jsp* dentro do diretório gerado e deixe o código igual ao da **Listagem 3**.

Listagem 3. Código da página cadastrarMedico.jsp.

```
01 <%@ page language="java" contentType="text/html;
    charset=ISO-8859-1" pageEncoding="ISO-8859-1%">
02 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
03 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
04 <%@ taglib prefix="springform"
    uri="http://www.springframework.org/tags/form" %>
05 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
06 <html>
07     <tiles:insertDefinition name="template">
08         <tiles:putAttribute name="corpo">
09             <h1>Cadastrar Médico</h1>
10
11             <springform:form method="post" action="${pageContext.request
                .contextPath}/medico/cadastrar.do" modelAttribute="medico">
12                 <div style="width: 300px;">
13                     <div class="form-group">
14                         <label for="nome">Nome:</label>
15                         <springform:input id="nome" path="nome"
                            cssClass="form-control"/>
16                     </div>
17                     <div class="form-group">
18                         <label for="especialidade">Especialidade:</label>
19                         <springform:select path="especialidade" cssClass="form-control">
20                             <option value="">selecione</option>
21                             <c:forEach items="${especialidades}" var="e">
22                                 <option value="${e}">${e.descricao}</option>
23                             </c:forEach>
24                         </springform:select>
25                     </div>
26                 </div>
27
28                 <input type="submit" value="Gravar" class="btn btn-primary" />
29             </springform:form>
30         </tiles:putAttribute>
31     </tiles:insertDefinition>
32 </html>
```

Os principais trechos desse código são:

- Linha 4: importa a *taglib* Spring Form, nativa do Spring MVC. Ela traz uma série de componentes úteis para uma maior produtividade no desenvolvimento do sistema, conforme será demonstrado por alguns itens a seguir;
- Linha 11: usa o componente **form** da biblioteca Spring Form e nele declara **modelAttribute="medico"**, o que significa que um objeto mapeado com o nome **medico** será utilizado dentro do formulário. Neste código, **medico** é uma instância da classe **Medico** que foi adicionada ao **HashMap** do método **redirecionaCadastroMedico()** de **NavegacaoController**. Lembrando que o

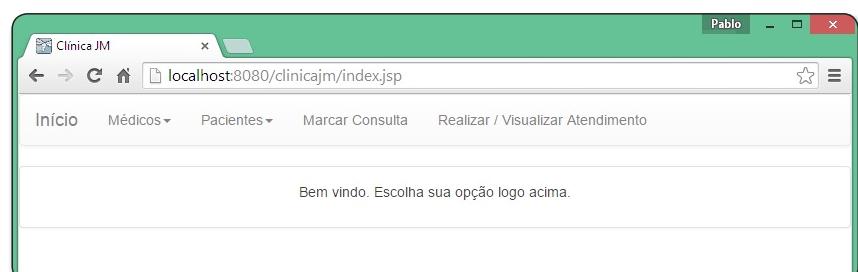


Figura 1. Tela inicial do sistema

HashMap mencionado serve para armazenar atributos a serem repassados para a tela;

- Linha 15: declara o componente **input** do Spring Form que aponta para a propriedade **nome** da variável **medico** através do atributo **path** da **tag**;
- Linha 19: o componente **select** do Spring Form aponta para a propriedade **especialidade** da variável **medico** através do atributo **path** da **tag**;
- Linhas 21 a 23: itera sobre a lista de especialidades de médicos mapeada pelo atributo **especialidades** e coloca cada uma das especialidades como uma opção no **select**. O objeto **especialidades** foi adicionado ao **HashMap** do método **redirecionaCadastroMedico()** de **NavegacaoController**;
- Linha 28: insere o botão de **submit** de formulário do Bootstrap.

Para visualizar a tela de cadastro de médicos, abra a página inicial do sistema, accese o menu **Médicos** e clique em *Cadastrar*. Aproveite e insira alguns médicos no banco de dados. A **Figura 2** exibe como ficou a *view*.

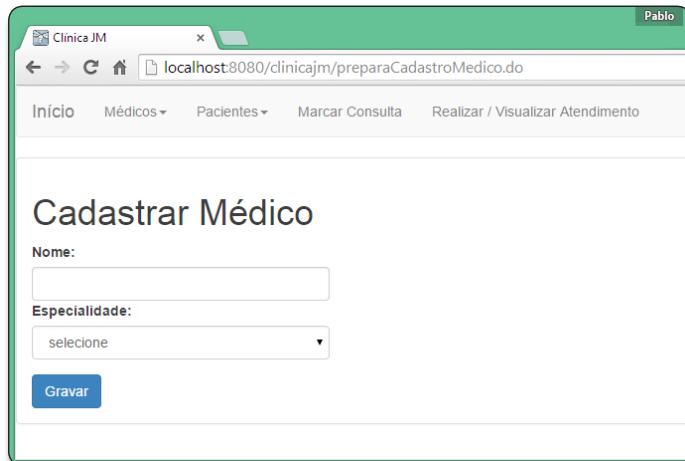


Figura 2. Tela de cadastro de médico

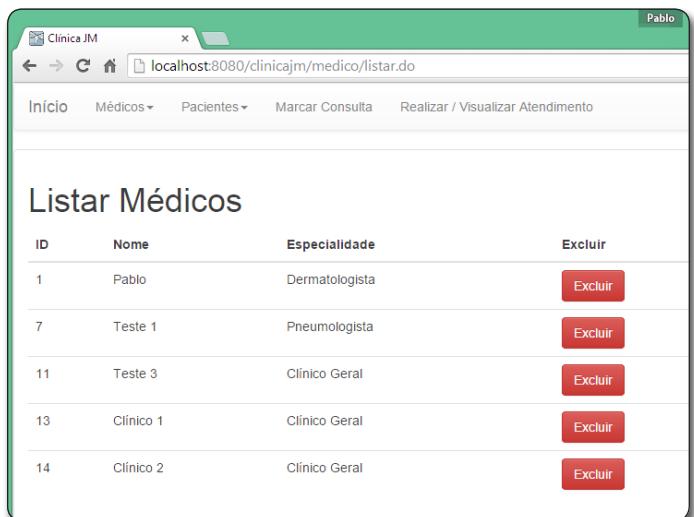
Agora que a página de cadastro de médicos está funcionando, vamos criar a de listagem. Assim, crie o arquivo *listaMedicos.jsp* dentro de *WEB-INF/jsp* e modifique o código para ficar igual ao da **Listagem 4**.

Esse código tem de diferencial o uso de uma tabela dentro de um painel do Bootstrap (lembre-se do que está no *template*). O framework monta, nestes casos, uma tabela sem bordas entre as colunas e com uma aparência bem moderna. Além dela, colocamos na linha 27 um botão do Bootstrap que aparecerá em cada linha da tabela para excluir o médico desejado.

Após modificar todo o código, accese a opção de listar médicos pelo menu para ver a página. A **Figura 3** mostra como ficou a tela.

Criando as páginas de cadastro e listagem de pacientes

Uma vez que o cadastro e a listagem de médicos estão funcionando, vamos agora trabalhar nas páginas que manipulam os dados



ID	Nome	Especialidade	Excluir
1	Pablo	Dermatologista	<button>Excluir</button>
7	Teste 1	Pneumologista	<button>Excluir</button>
11	Teste 3	Clinico Geral	<button>Excluir</button>
13	Clinico 1	Clinico Geral	<button>Excluir</button>
14	Clinico 2	Clinico Geral	<button>Excluir</button>

Figura 3. Tela de listagem de médicos

Listagem 4. Código da página listaMedicos.jsp.

```
01 <%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
02 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
03 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
04 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
05 <html>
06   <tiles:insertDefinition name="template">
07     <tiles:putAttribute name="corpo">
08       <h1>Listar Médicos</h1>
09
10      <table class="table">
11        <thead>
12          <tr>
13            <th>ID</th>
14            <th>Nome</th>
15            <th>Especialidade</th>
16            <th>Excluir</th>
17          </tr>
18        </thead>
19        <tbody>
20          <c:forEach var="m" items="${medicos}">
21            <tr>
22              <td>${m.id}</td>
23              <td>${m.nome}</td>
24              <td>${m.especialidade.descricao}</td>
25              <td>
26                <a href="${pageContext.request.contextPath}/medico/excluir.do?idMedico=${m.id}">
27                  <input type="button" value="Excluir" class="btn btn-danger"/>
28                </a>
29              </td>
30            </tr>
31          </c:forEach>
32        </tbody>
33      </table>
34    </tiles:putAttribute>
35  </tiles:insertDefinition>
36 </html>
```

dos pacientes da clínica. Para criar a primeira delas, o cadastro de paciente, gere o arquivo *cadastrarPaciente.jsp* dentro do diretório das telas anteriores e deixe o código igual ao da **Listagem 5**.

Listagem 5. Código da página *cadastrarPaciente.jsp*.

```

01 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
02 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
03 <%@ taglib prefix="springform" uri="http://www.springframework.org/
    tags/form"%>
04 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
05 <html>
06     <tiles:insertDefinition name="template">
07         <tiles:putAttribute name="corpo">
08             <h1>Cadastrar Paciente</h1>
09
10            <springform:form method="post" action="${pageContext.request
    .contextPath}/paciente/cadastrar.do" modelAttribute="paciente">
11                <div style="width: 300px;">
12                    <div class="form-group">
13                        <label for="nome">Nome:</label>
14                        <springform:input id="nome" path="nome"
                            cssClass="form-control"/>
15                    </div>
16                    <div class="form-group">
17                        <label for="dataNascimento">
18                            Data de Nascimento (DD/MM/AAAA):</label>
19                            <springform:input id="dataNascimento" path="dataNascimento"
                                maxlength="10" cssClass="form-control"/>
20                        </div>
21
22                <input type="submit" value="Gravar" class="btn btn-primary"/>
23            </springform:form>
24        </tiles:putAttribute>
25    </tiles:insertDefinition>
26 </html>
```

Observe que esse código é bem semelhante ao da página para cadastro de médicos. A diferença mais relevante é que no cadastro de paciente declaramos a instrução **modelAttribute="paciente"**, o que significa que dentro do formulário iremos trabalhar com a variável **paciente**, que é uma instância da classe **Paciente** e foi adicionada ao **HashMap** do método **redirecionaCadastroPaciente()** de **NavegacaoController**. Lembrando que o **HashMap** mencionado serve para armazenar atributos a serem repassados para a tela.

Após as alterações no código da página, acesse a tela inicial do sistema, entre no menu *Pacientes* e clique em *Cadastrar*. Aproveite e insira alguns pacientes no banco de dados tomando cuidado para fornecer a data de nascimento no padrão correto. A **Figura 4** mostra como ficou a aparência dessa funcionalidade na aplicação.

Vamos agora criar a página de listagem de pacientes. Deste modo, crie o arquivo *listaPacientes.jsp* dentro de *WEB-INF/jsp* e modifique o código para ficar igual ao da **Listagem 6**.

Mais uma vez, o código de listagem de pacientes é bem semelhante ao de listagem de médicos. A grande diferença neste caso é que estamos utilizando a tag **formatDate** da taglib de formatação de dados da JSTL na linha 26 para a data de nascimento do paciente ser exibida no formato brasileiro.

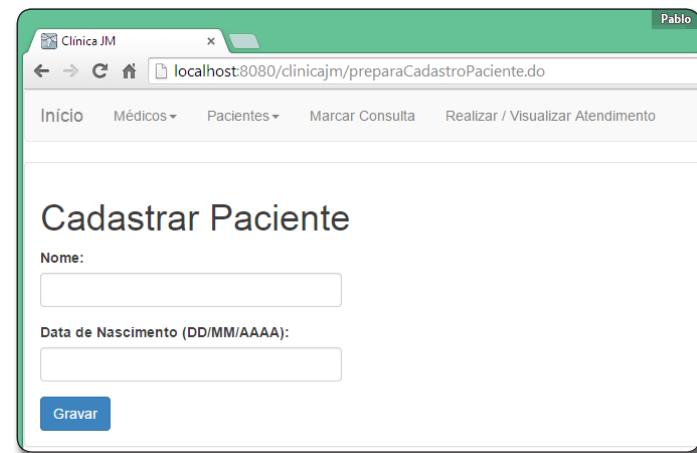


Figura 4. Tela de cadastro de paciente

Listagem 6. Código da página *listaPacientes.jsp*.

```

01 <%@ page language="java" contentType="text/html;
    charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
02 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
03 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
04 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
05 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
06 <html>
07     <tiles:insertDefinition name="template">
08         <tiles:putAttribute name="corpo">
09             <h1>Listar Pacientes</h1>
10
11            <table class="table">
12                <thead>
13                    <tr>
14                        <th>ID</th>
15                        <th>Nome</th>
16                        <th>Data de Nascimento</th>
17                        <th>Excluir</th>
18                    </tr>
19                </thead>
20                <tbody>
21                    <c:forEach var="p" items="${pacientes}">
22                        <tr>
23                            <td>${p.id}</td>
24                            <td>${p.nome}</td>
25                            <td>
26                                <fmt:formatDate value="${p.dataNascimento}"
                                    pattern="dd/MM/yyyy"/>
27                            </td>
28                            <td>
29                                <a href="${pageContext.request.contextPath}/paciente/
                                    excluir.do?idPaciente=${p.id}">
30                                    <input type="button" value="Excluir"
                                        class="btn btn-danger"/>
31                                </a>
32                            </td>
33                        </tr>
34                    </c:forEach>
35                </tbody>
36            </table>
37        </tiles:putAttribute>
38    </tiles:insertDefinition>
39 </html>
```

Para visualizar a página, acesse a opção de listagem de pacientes no menu do sistema. A **Figura 5** exibe a aparência final da tela.

Figura 5. Tela de listagem de pacientes

Criando as páginas de operações com as consultas dos pacientes

Agora que temos as funcionalidades envolvendo médicos e pacientes completas na aplicação, partiremos para a elaboração das operações relacionadas às consultas dos pacientes. Para criar a primeira delas, o cadastro de consulta, gere o arquivo *cadastrarConsulta.jsp* dentro do diretório das telas anteriores e deixe o código igual ao da **Listagem 7**.

Esta é a nossa primeira página com processamento Ajax do projeto. Dentre os principais pontos desse código estão os seguintes:

- Linhas 12 a 32: define a função **carregaMedicosDaEspecialidade()**, responsável por preencher o *select* com os médicos da especialidade selecionada. A função é chamada quando alguma opção no *select* de especialidades é marcada;
- Linha 13: remove todos as opções do *select* de médicos via jQuery;
- Linha 15: adiciona a opção padrão ao select, com o *label* “selecione” e o valor vazio;

Listagem 7. Código da página *cadastrarConsulta.jsp*.

```

01 <%@ page language="java" contentType=
02   "text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
03 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
04 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
05 <%@ taglib prefix="springform" uri="http://www.springframework.org/
06   tags/form"%>
07 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
08   "http://www.w3.org/TR/html4/loose.dtd">
09 <html>
10   <tiles:insertDefinition name="template">
11     <tiles:putAttribute name="corpo">
12       <h1>Cadastrar Consulta</h1>
13       <script type="text/javascript">
14         function carregaMedicosDaEspecialidade() {
15           $('#medico option').remove();
16           var selectMedicos = $('#medico');
17           selectMedicos.append('<option value="">selecione</option>');
18           if ($('#especialidade').val() != "") {
19             $.ajax({
20               type: "get",
21               url: "${pageContext.request.contextPath}/medico/
22                 listarPorEspecialidade.do",
23               data: 'especialidade=' + $('#especialidade').val(),
24               success: function(medicos) {
25                 $.each(medicos, function(i, medico) {
26                   selectMedicos.append('<option value="' + medico.id + '">' +
27                     medico.nome + '</option>');
28                 });
29               },
30             });
31           }
32         }
33       </script>
34       <springform:form method="post" action="${pageContext.request
35         .contextPath}/consulta/agendar.do" modelAttribute="consulta">
36         <div style="width: 300px;">
37           <div class="form-group">
38             <label for="especialidade">Especialidade:</label>
39             <select id="especialidade" onchange=
40               "carregaMedicosDaEspecialidade();"
41               class="form-control">
42               <option value="">selecione</option>
43               <c:forEach items="${especialidades}" var="e">
44                 <option value="${e}">${e.descricao}</option>
45               </c:forEach>
46             </select>
47             <div class="form-group">
48               <label for="medico">Médico:</label>
49               <springform:select id="medico" path="medico.id"
50                 cssClass="form-control">
51                 <option value="">selecione</option>
52               </springform:select>
53               <div class="form-group">
54                 <label for="paciente">Paciente:</label>
55                 <springform:select id="paciente" path="paciente.id"
56                   cssClass="form-control">
57                   <option value="">selecione</option>
58                 <c:forEach items="${pacientes}" var="p">
59                   <option value="${p.id}">${p.nome}</option>
60                 </c:forEach>
61               </springform:select>
62               <div class="form-group">
63                 <label for="data">Data (DD/MM/AAAA):</label>
64                 <input id="data" type="text" name="data" maxlength="10"
65                   class="form-control"/>
66               </div>
67               <div class="form-group">
68                 <label for="hora">Hora (HH:MM):</label>
69                 <input id="hora" type="text" name="hora" maxlength="5"
70                   class="form-control"/>
71               </div>
72             <input type="submit" value="Gravar" class="btn btn-primary" />
73           </springform:form>
74         </div:putAttribute>
75       </tiles:insertDefinition>
76     </html>

```

- Linhas 17 a 31: se foi selecionada alguma opção com o valor diferente de vazio no *select* de especialidades, ou seja, se não for a primeira opção, é feita uma requisição Ajax do tipo *get* para **MedicoController** no endereço http://endereco_servidor/clinicajm/medico/listaPorEspecialidade.do, passando a especialidade selecionada pelo usuário (linhas 20 e 21);
- Linhas 22 a 26: se a execução foi realizada com sucesso, percorre o JSON retornado e preenche o *select* de médicos com a lista de médicos devolvida. O procedimento inicia na linha 22, onde uma função é definida tendo como parâmetro a variável **medicos** que recebe os dados retornados pela chamada Ajax;
- Linhas 23 a 25: itera sobre o parâmetro **medicos** e define uma nova função que recebe cada objeto do array de médicos retornado no JSON através do parâmetro **medico**. Além dele, a função define um outro parâmetro chamado **i**, que funciona como um contador. Na linha 25 a rotina adiciona o médico que está sendo tratado no loop como opção do *select*;
- Linhas 27 a 29: se houve erro na chamada Ajax, mostra a mensagem de erro para o usuário no formato de alerta do JavaScript;
- Linhas 39 a 44: declara o *select* de especialidades. Veja que a função JavaScript **carregaMedicosDaEspecialidade()** é chamada no método **onchange()** e que não estamos usando o *select* do Spring Form porque não temos necessidade de passar a especialidade do médico via atributo *path* para o controller;
- Linhas 48 a 50: insere o *select* de médicos apenas com a opção **default** (não aponta para nenhum médico) na página.

Depois de finalizar as alterações no código da página, clique no item *Marcar Consulta* do menu do sistema. A **Figura 6** mostra como ficou a página.

Cadastrar Consulta

Especialidade:

Médico:

Paciente:

Data (DD/MM/AAAA):

Hora (HH:MM):

Gravar

Figura 6. Tela de cadastro de consulta

Para testar o Ajax, selecione uma especialidade e observe que o *select* de médicos é recarregado. Caso queira simular o erro na conversão de data feita no controller, informe uma data ou hora fora do padrão esperado pelo sistema e clique em *Gravar*.

Uma vez que temos o cadastro de consultas funcionando, vamos partir para a gravação do atendimento do paciente. Esta ação foi dividida em duas telas: a primeira traz a listagem de consultas do paciente, onde o usuário deve selecionar a consulta desejada; e a segunda é a gravação do atendimento em si. Assim, seguindo a sequência lógica, vamos começar com a página inicial do fluxo, gerando o arquivo *listaConsultas.jsp*. Em seguida, modifique o código para ficar igual ao da **Listagem 8**.

Nesta página também temos processamento Ajax e os principais pontos do seu código podem ser explicados da seguinte forma:

- Linhas 12 a 54: define a função **carregaConsultasPorPaciente()**, responsável por preencher a lista de consultas do paciente. A função é chamada quando alguma opção no *select* de pacientes é marcada;
- Linhas 13 e 14: oculta a div que possui a tabela com a lista de consultas e a div que informa que o paciente não tem consultas;
- Linhas 16 a 53: se foi selecionada alguma opção com o valor diferente de vazio no *select* de pacientes (a primeira opção tem o label "selecione" e o valor vazio), é feita uma requisição Ajax do tipo *get* para **ConsultaController** no endereço http://endereco_servidor/clinicajm/consulta/listaPorPaciente.do, passando o id do paciente selecionado pelo usuário (linhas 19 e 20);
- Linhas 21 a 48: se a execução foi realizada com sucesso, testa se o paciente não tem consultas marcadas e, neste caso, mostra a div que exibe a mensagem informando esta situação. Do contrário, percorre o JSON retornado e preenche a tabela de consultas com os dados devolvidos. O procedimento inicia na linha 21, onde uma função é definida tendo como parâmetro a variável **consultas** que recebe os dados retornados pela chamada Ajax;
- Linha 23: exibe a div informando que o paciente não tem consultas, se for o caso;
- Linha 25: exibe a div com a lista de consultas, se o paciente tiver pelo menos uma marcada;
- Linha 26: remove todas as linhas do corpo da tabela;
- Linhas 30 a 46: itera sobre o parâmetro **consultas** e define uma nova função que recebe cada objeto do array de consultas retornado no JSON através do parâmetro **consulta**. Além dele, a função define outro parâmetro chamado **i**, que funciona como um contador. Entre as linhas 34 e 45 a rotina adiciona a consulta atual como linha na tabela;
- Linha 31: armazena na variável **dataConsulta** a data da consulta do paciente através do construtor do tipo **Date** que recebe um *long* (formato recebido no JSON);
- Linha 32: caso a data de atendimento do paciente seja diferente de **null**, armazena na variável **dataAtendimento** a data de atendimento do paciente através do construtor do tipo **Date** que recebe um *long* (formato recebido no JSON);
- Linhas 37 e 38: chama a função **formataData()**, definida para mostrar a data e a hora da consulta e do atendimento no formato brasileiro;

- Linhas 49 a 51: se houve erro na execução, mostra a mensagem de erro para o usuário no formato de alerta do JavaScript;
- Linhas 56 a 66: define a função **formataData()**, que recebe um **Date** e o retorna no padrão brasileiro;
- Linhas 73 a 77: declara o *select* de pacientes. Veja que a função JavaScript **carregaConsultasPorPaciente()** é chamada no método **onchange()** e que não estamos usando o *select* do Spring Form

porque não temos necessidade de passar o id do paciente via atributo **path** para o *controller*;

- Linhas 83 a 86: declara a div que exibe a informação que o paciente não tem consultas. Como no carregamento da página ainda não temos um paciente selecionado, a div fica inicialmente oculta através da instrução **display: none**;
- Linhas 88 a 101: declara a div que exibe as consultas do paciente.

Listagem 8. Código da página listaConsultas.jsp.

```

01 <%@ page language="java" contentType="text/html;
02   charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
03 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
04 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
05 <%@ taglib prefix="springform" uri="http://www.springframework.org/
06   tags/form"%>
07 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
08   "http://www.w3.org/TR/html4/loose.dtd">
09 <html>
10   <tiles:insertDefinition name="template">
11     <tiles:putAttribute name="corpo">
12       <h1>Listar Consultas do Paciente</h1>
13       <script type="text/javascript">
14         function carregaConsultasPorPaciente(){
15           $("#lista_consultas").hide();
16           $("#sem_consultas").hide();
17
18           if ($("#paciente").val() != "") {
19             $.ajax({
20               type: "get",
21               url: "${pageContext.request.contextPath}/consulta/
22                 listarPorPaciente.do",
23               data: "idPaciente=" + $("#paciente").val(),
24               success: function(consultas) {
25                 if (consultas.length == 0) {
26                   $("#sem_consultas").show();
27                 } else {
28                   $("#lista_consultas").show();
29                   $("#tabela_consultas tbody tr").remove();
30                   var dataConsulta;
31                   var dataAtendimento;
32
33                   $.each(consultas, function(i, consulta) {
34                     dataConsulta = new Date(consulta.dataConsulta);
35                     dataAtendimento = consulta.dataAtendimento == null ?
36                       null : new Date(consulta.dataAtendimento);
37
38                     $("#tabela_consultas tbody").append(
39                       "<tr>" +
40                         "<td>" + consulta.medico.nome + "</td>" +
41                         "<td>" + formataData(dataConsulta) + "</td>" +
42                         "<td>" + formataData(dataAtendimento) + "</td>" +
43                         "<td>" +
44                           "<a href=${pageContext.request.contextPath}/consulta/
45                             detalharConsulta.do?idConsulta=" + consulta.id + ">" +
46                           "<input type='button' value='Selecionar'
47                             class='btn btn-success' />" +
48                           "</a>" +
49                         "</td>" +
50                         "</tr>" +
51                     );
52                 });
53               },
54             error: function() {
55               alert("Erro ao pesquisar consultas do paciente");
56             }
57           });
58         }
59       }
60       var dia = d.getDate();
61       var mes = d.getMonth() + 1;
62       var hora = d.getHours();
63       var minuto = d.getMinutes();
64
65       return (dia < 10 ? '0' + dia : dia) + '/' + (mes < 10 ? '0' + mes : mes)
66       + '/' + d.getFullYear() + '' + (hora < 10 ? '0' + hora : hora) + ':' + (minuto
67       < 10 ? '0' + minuto : minuto);
68     }
69   </script>
70   <div style="width: 220px;">
71     <div class="row">
72       <div class="col-md-6">Paciente:</div>
73       <div class="col-md-6">
74         <select id="paciente" name="paciente"
75           onchange="carregaConsultasPorPaciente();">
76           <option value="">selecione</option>
77           <c:forEach items="${pacientes}" var="p">
78             <option value="${p.id}">${p.nome}</option>
79           </c:forEach>
80         </select>
81       </div>
82     </div>
83     <div id="sem_consultas" style="display: none;">
84       <p>&nbsp;</p>
85       <span style="color: red">Paciente sem consultas</span>
86     </div>
87     <div id="lista_consultas" style="display: none;">
88       <table class="table" id="tabela_consultas">
89         <thead>
90           <tr>
91             <th>Médico</th>
92             <th>Data da Consulta</th>
93             <th>Data do Atendimento</th>
94             <th>Selecionar</th>
95           </tr>
96         </thead>
97         <tbody>
98           </tbody>
99         </table>
100      </div>
101      </div>
102      </tiles:putAttribute>
103    </tiles:insertDefinition>
104  </html>

```

Da mesma forma que o item anterior, a div fica inicialmente oculta porque no carregamento da página ainda não selecionamos um paciente;

- Linhas 89 a 100: declara a tabela que vai mostrar as consultas do paciente. Observe que ela tem apenas o cabeçalho e a declaração do corpo.

Após modificar o código da tela, clique no item *Realizar > Visualizar Atendimento* do menu do sistema. A **Figura 7** demonstra a aparência da página.

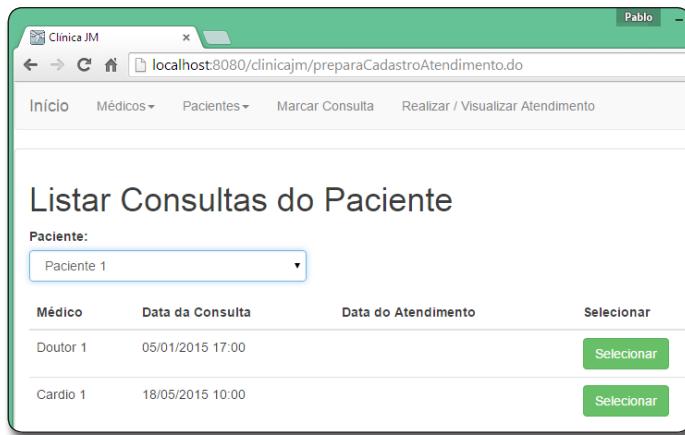


Figura 7. Tela de listagem de consultas

Para testar o Ajax, selecione um paciente e observe que a lista de consultas dele é mostrada em seguida. Caso ele não tenha

consultas, a div com a mensagem informando que o paciente não tem consultas é exibida.

Nossa última página no sistema servirá para realizarmos a atualização da consulta com as informações do atendimento do paciente no banco de dados. Desta forma, crie o arquivo *realizarAtendimento.jsp* dentro da mesma pasta da tela anterior e modifique o código para que fique igual ao da **Listagem 9**.

Os principais trechos desse código podem ser explicados da seguinte forma:

- Linha 10: como esta página foi criada para gravar ou visualizar um atendimento, aqui há um teste para exibir o título da página de acordo com a ação;
- Linhas 24 e 28: deixa os campos *textarea* como *readonly* se for apenas uma visualização do atendimento;
- Linhas 32 a 39: exibe o botão de gravar ou de voltar. Esta definição depende do usuário: se ele está gravando ou visualizando o atendimento;
- Linhas 41 a 44: coloca alguns atributos da consulta como campos *hidden* para que o objeto mapeado pela variável **consulta** chegue no *controller* com os dados que já estavam preenchidos antes, como a data da consulta e o id do paciente. Se não fizéssemos isso, os campos ficariam nulos, causando erro no *update*.

Depois que o código da página foi alterado, acesse o item *Realizar > Visualizar Atendimento* do menu do sistema e clique em uma das consultas desejadas para visualizar a tela criada.

A **Figura 8** mostra como ficou a *view* no caso de uma gravação de atendimento.

Listagem 9. Código da página realizarAtendimento.jsp.

```

01 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
02 pageEncoding="ISO-8859-1"%>
03 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
04 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
05 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
06 <%@ taglib prefix="springform" uri="http://www.springframework.org/
07 tags/form"%>
08 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
09 "http://www.w3.org/TR/html4/loose.dtd">
10 <html>
11   <tiles:insertDefinition name="template">
12     <tiles:putAttribute name="corpo">
13       <h1>${
14         consulta.dataAtendimento == null ? "Realizar Atendimento":
15           "Visualizar Atendimento"
16       }</h1>
17       <springform:form method="post" action="${pageContext.request
18 .contextPath}/consulta/atender.do" modelAttribute="consulta">
19         <div style="width: 500px;">
20           Especialidade: ${consulta.medico.nome}
21           <br />
22           Paciente: ${consulta.paciente.nome}
23           <br />
24           Data da Consulta: <fmt:formatDate value="${consulta.dataConsulta}"
25             pattern="dd/MM/yyyy HH:mm"/>
26         </div>
27         <br />
28         <div style="width: 500px;">
29           <label for="sintomas">Sintomas:</label>
30           <springform:textarea id="sintomas" path="sintomas" readonly=
31             "${consulta.dataAtendimento != null}" cssClass="form-control"/>
32         </div>
33         <div class="form-group">
34           <label for="receita">Receita:</label>
35           <springform:textarea id="receita" path="receita" readonly=
36             "${consulta.dataAtendimento != null}" cssClass="form-control"/>
37         </div>
38         <div>
39           <c:if test="${consulta.dataAtendimento == null}">
40             <input type="submit" value="Gravar" class="btn btn-primary"/>
41           </c:if>
42           <c:if test="${consulta.dataAtendimento != null}">
43             <a href="${pageContext.request.contextPath}/
44               cadastroAtendimentoPasso1.do">
45               <input type="button" value="Voltar" class="btn btn-warning"/>
46             </a>
47           </c:if>
48           <springform:hidden path="id"/>
49           <springform:hidden path="dataConsulta" />
50           <springform:hidden path="medico.id" />
51           <springform:hidden path="paciente.id" />
52         </div>
53       </springform:form>
54     </tiles:putAttribute>
55   </tiles:insertDefinition>
56 </html>

```

Clinica JM

localhost:8080/clinicajm/consulta/detalharConsulta.do?idConsulta=3

Inicio Médicos▼ Pacientes▼ Marcar Consulta Realizar / Visualizar Atendimento

Realizar Atendimento

Especialidade: Doutor 1
Paciente: Paciente 1
Data da Consulta: 05/01/2015 17:00
Sintomas:

Receita:

Gravar

Figura 8. Tela de realização / visualização de atendimento

Conhecer melhor os dois frameworks líderes de mercado em suas categorias – o Spring MVC entre as bibliotecas para desenvolvimento Java *web* e o Bootstrap entre as soluções para elaboração de *front-end* – fez com que compreendêssemos porque os dois estão nestas posições: ambos fornecem recursos diferenciados e são extremamente simples de se utilizar em projetos dos mais diversos tamanhos. No caso do Bootstrap, a sua escolha decorre, na maioria das vezes, pelo suporte ao *layout* responsivo, uma vez que existe uma exigência cada vez maior de compatibilidade de sistemas *web* com dispositivos móveis.

Entretanto, de nada adianta analisarmos e conhecermos os principais recursos dos dois produtos se não colocarmos em prática a teoria aprendida nas documentações fornecidas por eles. Por este motivo, o sistema que desenvolvemos certamente nos deu mais segurança de escolha e ajudou a compreender melhor de onde vem a grande aceitação dos dois frameworks entre os desenvolvedores.

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Autor



Pablo Bruno de Moura Nóbrega

<http://pablonobrega.wordpress.com>

É Líder de Projeto Java, certificado OCJP e OCWCD, Graduado em Ciências da Computação pela Universidade de Fortaleza – UNIFOR, Mestre em Computação pela Universidade Estadual do Ceará – UECE, cursa MBA em Gerenciamento de Projetos na UNIFOR, trabalha na Secretaria Municipal de Finanças de Fortaleza – SEFIN e desenvolve sistemas há cerca de nove anos.



Autor



Marcos Vinicios Turisco Dória

É Analista de Sistemas Java, certificado OCJA e OCJP, Graduado em Ciências da Computação pela Universidade de Fortaleza – UNIFOR, trabalha na Secretaria de Finanças de Fortaleza - SEFIN e desenvolve há cerca de cinco anos.



Links:

Página de download do Eclipse Luna.

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

Página de download do WildFly.

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

Página de download do PostgreSQL.

<http://www.enterprisedb.com/products-services-training/pgdownload>

Página de download do Bootstrap.

<http://getbootstrap.com/getting-started/#download>

Endereço para download do jQuery em versão compactada.

<http://code.jquery.com/jquery-2.1.3.min.js>

Spring Framework Reference Documentation.

<http://docs.spring.io/spring/docs/4.2.0.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/>

Tutorial Apache Tiles.

<http://tiles.apache.org/framework/tutorial/index.html>

Conformação arquitetural: sincronizando o código e a arquitetura do projeto

Ferramentas e técnicas para manter o código conforme o projeto arquitetural

Arquitetura de software é uma disciplina de desenvolvimento de software que engloba inúmeras preocupações técnicas. Conceituando arquitetura de software, pode-se dizer que é um nível de design além dos algoritmos e estrutura de dados, cuja preocupação gira em torno de como o sistema está organizado e estruturado. Em *An Introduction to Software Architecture*, David Garlan e Mary Shaw, incluem como questões de organização e estruturação de um sistema, elementos como protocolos de comunicação, sincronização e acesso a dados, atribuição de funcionalidades a elementos de design, distribuição física de módulos entre diferentes nós computacionais, composição de elementos de design, escalonamento e desempenho, e escolha entre diferentes alternativas de arquitetura. Todas as questões envolvidas na organização e estruturação de um sistema, comentadas anteriormente, fazem parte do escopo dos requisitos não funcionais. Por isso que uma das explicações mais singelas e, muitas vezes a frase mais aceita como resumo geral sobre arquitetura de software, citam que: as preocupações da disciplina de arquitetura de software englobam os requisitos não funcionais.

Traçando um paralelo desses conceitos abstratos de arquitetura de software com a plataforma Java, pode-se pensar a arquitetura de software como a disciplina que vai organizar e estruturar os componentes do sistema (pacotes, classes e subsistemas) e ditar as regras de como esses componentes podem ou devem se comunicar e relacionar, levando-se em consideração requisitos não funcionais de performance, manutenibilidade, confiabilidade, entre outros.

Durante a concepção e elaboração de um software, um dos principais documentos produzidos é o documento

Fique por dentro

Ao longo do tempo a arquitetura planejada para um software tende a se distanciar do que foi inicialmente projetado. A representação do projeto por meio de diagramas UML em documentos de arquitetura de software perdem total conexão com o código fonte e, quando se torna necessária alguma alteração mais complexa, que exija o entendimento arquitetural da solução e a consistência do código real com o código planejado, percebe-se os problemas causados pela disparidade arquitetural.

O cenário ideal seria um em que, durante todo o ciclo de vida do software, o código fonte estivesse sempre refletindo o que foi projetado para a sua arquitetura. Com base nisso, esse artigo apresenta a importância da arquitetura de software, técnicas e ferramentas que auxiliam a manter o código fonte conforme a arquitetura planejada e técnicas para melhorar a conformação arquitetural em um código já deteriorado.

de arquitetura de software. Esse documento serve para guiar o desenvolvimento do software e deve fornecer uma visão geral da arquitetura abrangente do sistema, usando diversas visões de arquitetura para descrever diferentes aspectos do software, tais como dependências entre pacotes, organização do sistema em camadas, descrição de casos de uso, etc.

Entretanto, ao longo do tempo as decisões de projeto propostas e determinadas no documento de arquitetura de software acabam sendo esquecidas e o projeto do sistema começa a deteriorar-se. O cenário de deterioração da arquitetura de software é caracterizado pela não conformidade do código atual com o projeto descrito no documento de arquitetura. Esse cenário é menos comum em projetos novos, onde as diretrizes de desenvolvimen-

to para o projeto ainda estão sendo seguidas à risca. Contudo, durante a manutenção do sistema, especialmente quando há uma mudança considerável na equipe e muitas requisições de alterações, o novo código tende a ser desenvolvido fora das diretrizes determinadas no início do projeto, tanto pelo esquecimento do documento de arquitetura de software quanto por desenvolvedores preferirem outras abordagens em relação às originalmente propostas no início do projeto. Dessa forma, o código acaba se transformando em um grande e confuso emaranhado de padrões ou anti-padrões, causando muitos inconvenientes, tais como dificuldades de manutenção e evolução do sistema, duplicação de código, entre outros.

A solução para esse problema é prever disparidades entre a arquitetura e o código a fim de evitar que elas se concretizem ou verificar constantemente o código em busca de problemas em relação à arquitetura e corrigi-los. As abordagens para a resolução desses problemas geralmente consistem na adição de um processo ou subprocesso de commit ou verificação de código juntamente com a adoção de ferramentas para este fim. O termo genérico para essas abordagens é conformação arquitetural, definida como uma técnica para verificar se a representação de baixo nível de um software, como o seu código fonte, está de acordo com o seu projeto arquitetural planejado.

Existem muitas formas de conformação arquitetural, ou seja, muitas formas para controlar a arquitetura do software. Entre elas estão o check customizado de regras arquiteturais em tempo de commit, a inspeção contínua do código em busca de problemas arquiteturais e a verificação de regras arquiteturais

em tempo de desenvolvimento. As técnicas em si são muito diferentes em sua abordagem, possibilitando com isso a aplicação da conformação arquitetural em diferentes contextos de diferentes formas.

Sendo assim, os próximos tópicos deste artigo apresentarão os conceitos mais importantes sobre arquitetura de software e as particularidades das técnicas mais difundidas para conformação arquitetural.

Documento de Arquitetura de Software

Em *The Unified Modeling Language User Guide* (Booch, Rumbaugh e Jacobson), temos uma das definições mais concisas e completas sobre arquitetura de software:

“Uma arquitetura é um conjunto de decisões significativas sobre a organização de um sistema de software, a seleção dos elementos estruturais e suas interfaces pelas quais o sistema é composto, juntamente com seu comportamento, como especificado nas suas colaborações entre esses elementos, a composição desses elementos, e o estilo arquitetural que dirige essa organização – esses elementos, suas interfaces, suas colaborações e sua composição.”

De forma geral, todas as definições de arquitetura de software referem-se a restrições, organização, padrões e responsabilidades de módulos e componentes e também de sistemas como um todo.

Durante a concepção de um sistema, quando sua arquitetura começa a ser modelada e toma forma, é imprescindível documentá-la de forma que todas as decisões e restrições acerca do projeto sejam registradas para guiar o desenvolvimento de todo



o sistema. Atualmente, o documento mais utilizado para isso é o documento de arquitetura de software (DAS), cuja disseminação se deu por meio do RUP (*Rational Unified Process*), já que o DAS é um dos artefatos mais importantes no RUP.

O documento de arquitetura de software descreve as grandes ideias da arquitetura de software de um projeto, incluindo a análise arquitetural, as restrições, os requisitos não funcionais, a relação entre módulos, pacotes e sistemas, etc. A seguir são apresentados os tópicos mais comuns do documento de arquitetura de software (geralmente esse é o seu índice):

- **Introdução:** fornece uma visão geral de todo o sistema e, principalmente, seu escopo;
- **Representação Arquitetural:** descrição das visões utilizadas para descrever a arquitetura do sistema em questão;
- **Restrições Arquiteturais:** lista de restrições que têm alguma relação significativa com a arquitetura. Exemplos de restrições incluem quais navegadores devem suportar a aplicação, particularidades de sistemas legados que influenciarão o sistema, entre outros;
- **Visões:** geralmente um capítulo para cada visão arquitetural (visão de casos de uso, visão lógica, visão de processos, visão de implantação e visão de implementação);
- Tópicos complementares descrevendo decisões específicas acerca do desempenho, qualidade, componentes de software de terceiros, ferramentas, entre outros.

A arquitetura do sistema é apresentada nesse documento principalmente por meio de diagramas UML, provendo dife-

rentes visões da arquitetura, desde visões de alto nível, como diagramas de casos de uso, até visões de mais baixo nível, como diagramas de sequência e diagramas de classe. Além disso, o documento de arquitetura de software geralmente contém informações sobre as tecnologias e ferramentas utilizadas no projeto e suas versões, e informações sobre a implantação do software e ambiente de execução. A seção **Links** contém o endereço para um exemplo de um documento de arquitetura de software. Esse documento faz parte dos artefatos de exemplo do RUP. Além disso há também um link de documentação descrevendo todo o conteúdo do documento de arquitetura de software.

Disparidade Arquitetural

A essência do documento de arquitetura de software é ser um resumo das decisões arquiteturais que deve, além de servir de guia para o desenvolvimento, ser um guia para novos desenvolvedores que venham a fazer parte do projeto, permitindo o entendimento global do sistema em pouco tempo.

Entretanto, quando o código fonte já não reflete a sua arquitetura planejada, ou seja, quando não reflete o que está definido no documento de arquitetura, temos um caso de disparidade entre o sistema real e o sistema planejado. Esse problema acontece com bastante frequência e geralmente é causado por prazos apertados onde é necessário adotar padrões não especificados na arquitetura para obter uma solução mais rápida ou negligência por parte dos desenvolvedores em seguir as restrições propostas, entre outros.



Infelizmente é comum que, com o passar do tempo, o documento de arquitetura de software já esteja muito defasado em relação ao que está implementado no código fonte, o que faz o documento perder o seu propósito, aumentando a curva de aprendizado de novos desenvolvedores e, o mais preocupante, tornando o sistema complexo e com diferentes soluções por todas as partes implementadas ao gosto dos desenvolvedores. Nesse ponto a arquitetura proposta e a arquitetura implementada já não têm nada em comum. É a disparidade arquitetural, chamada também de violação arquitetural.

Tipos de disparidade arquitetural

Dentre as violações mais comuns, destacam-se primeiramente problemas relacionados a dependências entre módulos ou classes. Durante o projeto do sistema geralmente se define como os módulos e classes devem se comunicar. Essas definições são apresentadas principalmente por meio de diagramas de classe, diagramas de sequência e principalmente pelo diagrama de pacotes e representam as dependências entre os artefatos de software do sistema.

A **Figura 1** mostra um exemplo de um diagrama de pacotes definindo as dependências entre módulos. De acordo com a arquitetura planejada nesse diagrama, as classes do pacote de apresentação (view) podem depender, ou seja, usar, apenas o pacote de modelo (model) que, por sua vez, pode depender apenas do pacote de persistência (persistence). Além dessas dependências comentadas anteriormente, por meio do diagrama de pacotes pode-se visualizar todas as dependências entre pacotes representadas por uma flecha tracejada. Quando o código não reflete esta definição, o que muitas vezes é comum, tem-se uma disparidade arquitetural. Esse é o pior tipo de violação, pois está relacionada a grandes conjuntos de artefatos de software e afeta o sistema como um todo, não apenas em pontos específicos.

Nota:

A UML não define formalmente um diagrama exclusivo para representação de pacotes. Os componentes gráficos utilizados para representar pacotes são todos da perspectiva do diagrama de classe. Entretanto, é comum criar diagramas de classe UML que mostram apenas os pacotes e suas dependências, chamando-os informalmente de diagrama de pacote.

Outro tipo de disparidade bastante comum é a presença de comportamentos em classes que não foram previstos na arquitetura. Isso acontece quando uma classe possui métodos além do especificado. Porém, como nem todo sistema é especificado com esse nível de detalhamento, esse tipo de violação acaba passando despercebida e faz parte do cotidiano de desenvolvedores de software tanto ler e entender quanto escrever comportamentos não previstos na arquitetura.

A disparidade arquitetural traz consigo muita complexidade que, por consequência, causa muitos prejuízos devido à maior quantidade de tempo requerido na resolução de problemas, maior tempo necessário para aprendizado do sistema, entre outros. Os próximos tópicos mostrarão as técnicas mais difundidas atu-

almente para manter a conformação arquitetural entre código e documentação, além de comentar diretrizes para sua utilização e implementação.

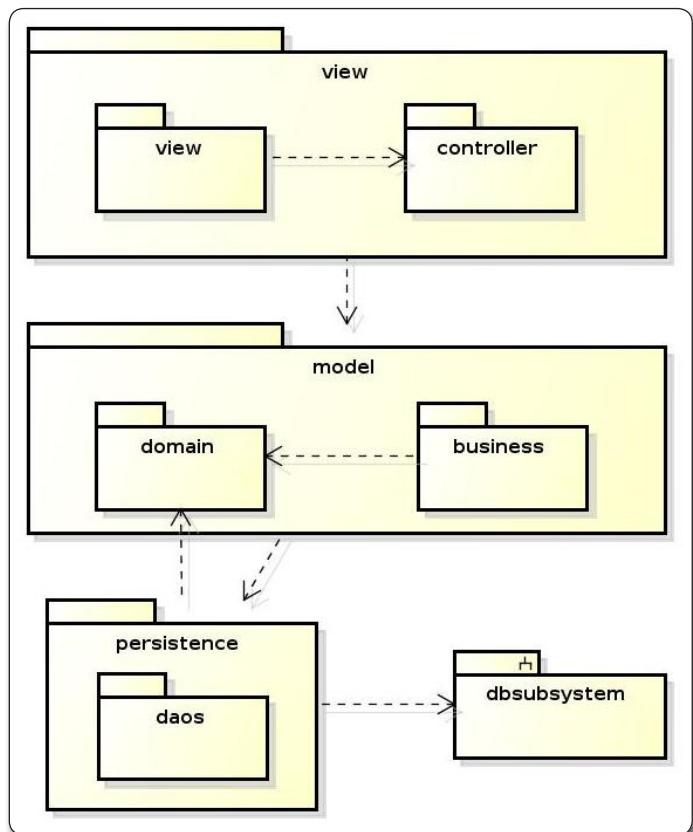


Figura 1. Diagrama de pacotes mostrando as dependências entre pacotes

Conformação Arquitetural

Diz-se que um sistema está conforme sua arquitetura quando a implementação adere aos princípios de projeto e restrições propostas em sua arquitetura. Essa é a melhor definição para conformação arquitetural. Exemplos de restrições arquiteturais incluem as definições de como os pacotes e classes podem ser utilizados entre si.

A princípio, os benefícios obtidos com a conformação arquitetural parecem abstratos demais. Argumentos como facilidade de manutenção ou facilidade de inserção de novos membros na equipe, por serem associados como resultados de forma genérica a tantas outras soluções, acabam não convencendo sobre o motivo de se preocupar com a conformação arquitetural, ainda mais em tempos em que surgem linhas de pensamento afirmando que a arquitetura deveria ser pensada com prazo de validade, ou seja, que a arquitetura deverá ser refeita após um período de tempo determinado (veja na seção **Links** o endereço para um artigo sobre arquitetura de software com prazo de expiração).

Mas tendo em vista que o documento de arquitetura de software descreve de forma implícita diversos atributos de qualidade, para que um software seja percebido de forma positiva por seus usuários,

ele deve estar conforme a sua arquitetura. Sendo assim, é inegável a importância da conformação arquitetural para a qualidade de um software, além dos outros benefícios já comentados.

As técnicas para conformação arquitetural geralmente envolvem a definição de restrições e validações estáticas sob o código fonte. Isso se deve ao fato das linguagens de programação não possuírem meios suficientes para restringir como os componentes do sistema podem ou devem se relacionar.

Na linguagem Java, o mecanismo de restrição são os modificadores de acesso (**public**, **private**, **protected**, **default**) que definem quais componentes podem ou não acessar ou visualizar outros componentes do software. Entretanto, apesar do uso dos modificadores de acesso serem a base para as restrições arquiteturais, apenas o uso deles não é suficiente para realizar a conformação arquitetural de forma eficaz, visto que a dependência dos modificadores de acesso para garantir a conformação significa depender do próprio código fonte que, como visto anteriormente, tende a perder a sincronia com a arquitetura projetada ao longo do tempo. O ideal é dispor de técnicas que sejam independentes do código fonte.

Com base nisso, nos próximos tópicos serão mostradas as práticas mais comuns de se manter a arquitetura projetada conforme a implementação. São técnicas que vão da análise constante de código até a validação do código fonte na IDE.

Análise de código

Uma das formas mais simples para manter a conformação arquitetural é por meio da análise de código com alguma ferramenta de verificação, como a ferramenta Sonar.

Nessa técnica, o código é monitorado constantemente por algum sistema de verificação que gera relatórios sobre diversos aspectos do código fonte, possibilitando a visualização de inconsistências arquiteturais no projeto. Quando alguma inconsistência é encontrada, deve-se alterar o código fonte para que o mesmo volte ao estado de conformidade arquitetural.

O Sonar é uma ferramenta bastante utilizada para medir e monitorar o código fonte e permite visualizar diversos aspectos arquiteturais, como dependências entre pacotes e classes, duplicações e complexidade de código. Sendo assim, é uma excelente opção para monitorar o código fonte em busca de problemas de conformação arquitetural. A **Figura 2** mostra um exemplo de como o Sonar identifica dependências entre pacotes e classes. Com base nessas informações é possível agir sobre as dependências identificadas e corrigir problemas de disparidade arquitetural para voltar ao nível de conformidade ideal. De acordo com a **Figura 2**, o projeto em questão, Apache Commons Lang, possui sete dependências entre pacotes e 21 dependências entre arquivos.

Quando se clica no número de dependências, no Sonar, ele exibe a matriz de dependências, uma forma fácil para se navegar entre as dependências existentes entre os componentes do projeto. A **Figura 3** apresenta a matriz de dependências do projeto Apache Commons Lang. Essa matriz mostra inicialmente o número de dependências que um pacote possui em relação a outro pacote. A primeira coluna da matriz é uma lista com os pacotes do projeto, os cabeçalhos das colunas subsequentes são os mesmos pacotes da primeira coluna, usadas para relacionar os pacotes entre si e mostrar aonde as dependências ocorrem.



As dependências mostradas pelo Sonar com destaque em vermelho são dependências cíclicas que deveriam ser removidas, as demais são dependências simples. Veja na seção **Links** o endereço para acessar o dashboard do Sonar para esse projeto.

Para visualizar detalhes de uma dependência cíclica, deve-se dar um duplo clique na dependência marcada em vermelho. Feito isso o Sonar irá exibir os arquivos envolvidos na dependência cíclica em questão.

Como exemplo, considere na **Figura 3** o pacote *src/main/java/org/apache/commons/lang3/text*. Esse pacote possui a indicação de duas dependências cíclicas em relação ao pacote *src/main/java/org/apache/commons/lang3*. Ao dar um duplo clique na dependência, o Sonar exibe os detalhes dos arquivos, nos pacotes referidos, que possuem as dependências, como mostra a **Figura 4**. Isso significa que a classe **ObjectUtils** usa a classe

StrBuilder e a classe **StrBuilder** possui uma referência à classe **ObjectUtils**.

O propósito da matriz de dependências é facilitar a visualização das dependências entre componentes de um projeto. O Sonar fornece uma matriz de dependências dinâmica que possibilita obter informações sobre o relacionamento entre todos os pacotes e classes de um projeto. Essa é a principal fonte de conhecimento sobre a estrutura arquitetural de um projeto.

A análise de código para manter a conformação arquitetural, apesar de ser simples para implementar, é a alternativa mais trabalhosa, tendo em vista que ela adota a correção ao invés da prevenção, ou seja, com o uso de uma ferramenta de análise de código permite-se que o código tenha muitos pontos de disparidade arquitetural que serão corrigidos conforme a identificação dos mesmos e a disponibilidade da equipe para realizar as alterações.



Figura 2. Identificação de dependências entre pacotes e classes no Sonar

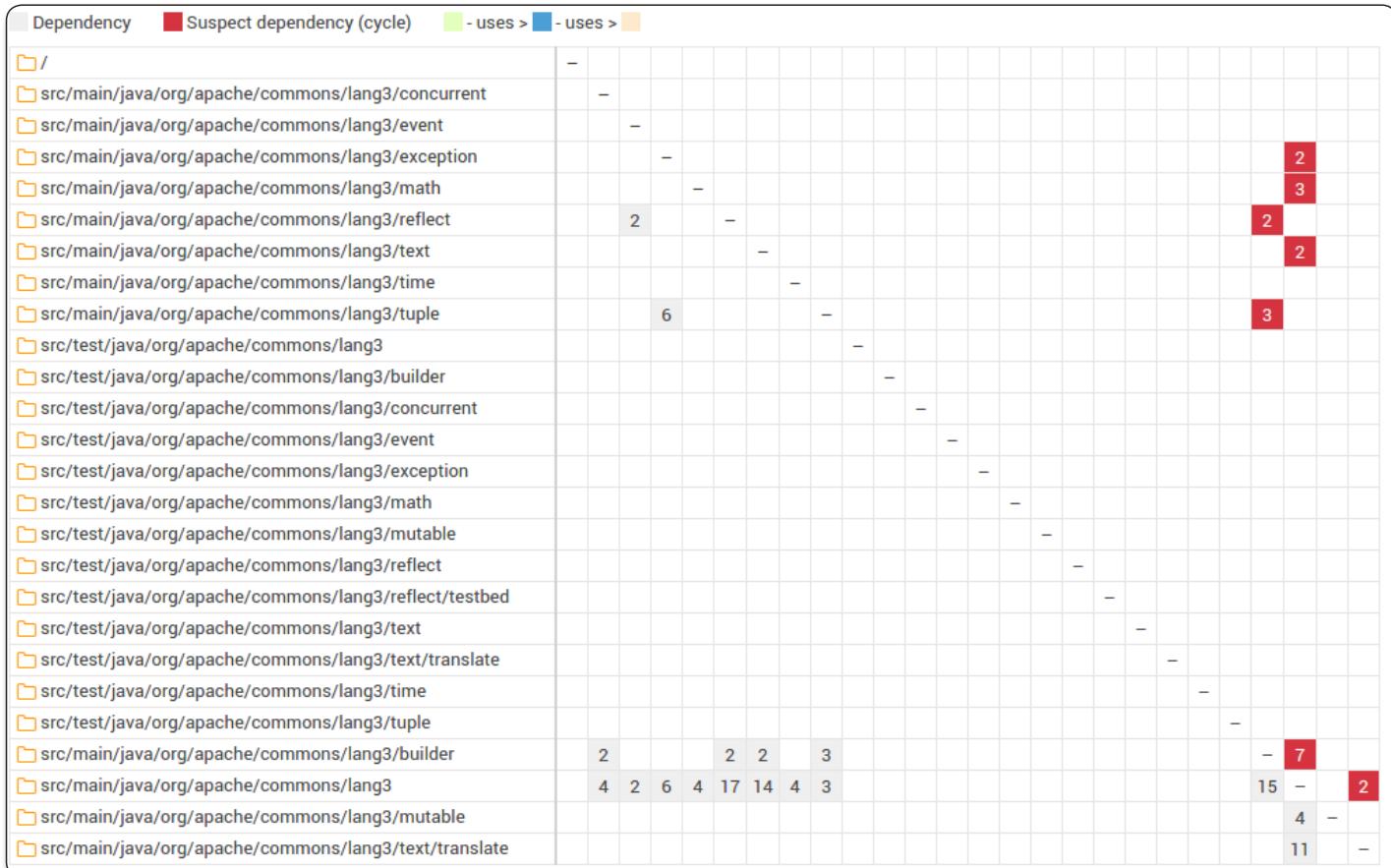


Figura 3. Matriz de dependências do Sonar

Conformação arquitetural: sincronizando o código e a arquitetura do projeto

[src/main/java/org/apache/commons/lang3](#)

[src/main/java/org/apache/commons/lang3/ObjectUtils.java](#) USES [src/main/java/org/apache/commons/lang3/text/StrBuilder.java](#)

[src/main/java/org/apache/commons/lang3/text](#)

Figura 4. Classes que possuem dependências cílicas

Essa técnica é mais conveniente quando o código fonte encontra-se em um estado avançado de disparidade arquitetural e a única forma de identificar disparidades arquiteturais é a verificação do mesmo em busca de discrepâncias para serem corrigidas. Existem também abordagens de prevenção que envolvem a validação de commits e são mais úteis quando o código está conforme a arquitetura e deseja-se evitar que ele entre em estado de não conformidade. Ainda assim, o uso de mais de uma abordagem para verificação e validação da conformidade arquitetural faz sentido e é recomendado, pois cada técnica atua de forma diferente, seja validando o que já foi feito ou prevenindo o que será feito. Além disso, o Sonar não é uma ferramenta exclusiva para conformação arquitetural, ela é uma ferramenta de qualidade de código que auxilia a manter o código limpo e livre de inconsistências, sendo de grande ajuda ao propósito da conformação.

No próximo tópico será apresentada mais uma abordagem para manter a conformação arquitetural: a validação de código em tempo de desenvolvimento ainda na IDE, possibilitando que, enquanto o desenvolvedor realiza seu trabalho, inconsistências arquiteturais sejam identificadas antes do commit por meio de verificação na própria IDE.

Validação de código na IDE

A validação de código ainda durante o desenvolvimento, na própria IDE, é uma abordagem de prevenção, pois identifica violações antes do código ser enviado para o controle de versão. Diferentemente da abordagem de análise de código com ferramentas de verificação, a validação na IDE permite que apenas o desenvolvedor veja se está cometendo violações e possibilite-o corrigir e, então, fazer o commit de um código que já esteja conforme a arquitetura.

Uma ferramenta muito eficaz para o controle de código na IDE é o DCLcheck, um plugin para a IDE Eclipse que reporta violações arquiteturais encontradas no código fonte na mesma aba de erros e alertas de compilação. As violações arquiteturais são exibidas como erro, forçando o desenvolvedor a manter a conformidade do código em tempo de codificação.

Esse mecanismo de validação do DCLcheck é possível graças à DCL (*Dependency Constraint Language*). A DCL é uma linguagem de domínio específica e declarativa que permite a definição de restrições entre módulos (conjunto de classes) de um projeto. As restrições suportadas pela DCL incluem regras de dependências que especificam quais módulos podem acessar, declarar, es-



tender, criar, lançar exceções, depender e implementar. A Figura 5 mostra a sintaxe da DCL, que é bastante simples e intuitiva. Por meio dessa linguagem é possível declarar que o pacote “model” não pode depender do pacote “view” com uma declaração parecida com o seguinte:

MODEL cannot-depend VIEW

Os elementos VIEW e MODEL são variáveis definidas pelo desenvolvedor, na DCL, para pacotes Java e devem ser declaradas antes do seu uso. A declaração dos pacotes VIEW e MODEL, por exemplo, poderia ser realizada da seguinte maneira:

```
module VIEW: com.gabrielamorim.jm.view.*
module MODEL: com.gabrielamorim.jm.model.*
```

Essa declaração quer dizer que o módulo VIEW é formado por todas as classes do pacote `com.gabrielamorim.jm.view` enquanto que o módulo MODEL é composto por todas as classes do pacote `com.gabrielamorim.jm.model`.

A forma de utilização do DCLcheck é por meio da ativação do mesmo sobre o projeto no Eclipse. Uma vez ativado, o plugin irá criar um arquivo chamado `architecture.dcl`, onde é possível definir todas as restrições arquiteturais do projeto. A partir do momento em que se tem restrições configuradas no arquivo `architecture.dcl`, o plugin passa a verificar o código fonte e alertar sobre erros devido a restrições arquiteturais.

A Figura 6 apresenta a IDE Eclipse com o DCLcheck sendo utilizado em um projeto de exemplo.

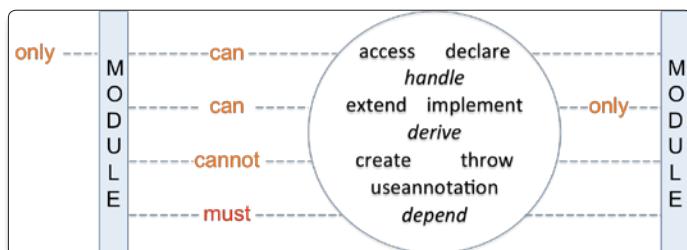


Figura 5. Sintaxe da linguagem DCL

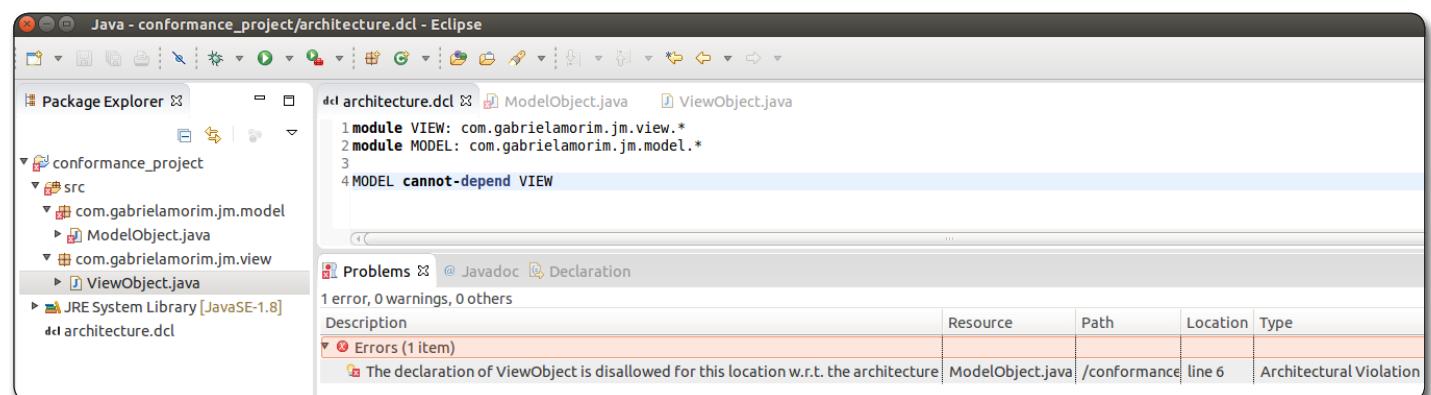


Figura 6. Violação arquitetural detectada no DCLcheck

A aba *Package Explorer* mostra como o projeto é configurado. Note a presença do arquivo `architecture.dcl` na hierarquia e como as restrições estão declaradas nesse arquivo na aba principal.

Esse projeto de exemplo contém duas classes: a `ModelObject` no pacote `com.gabrielamorim.jm.model` e a `ViewObject` no pacote `com.gabrielamorim.jm.view`. De acordo com a declaração no arquivo `architecture.dcl`, o módulo MODEL não pode depender do módulo VIEW. Sendo assim, quando a classe `ModelObject` declara um objeto do tipo `ViewObject`, acontece uma violação arquitetural. A violação é reportada na aba *Problems*, do Eclipse, como se fosse um erro de compilação, com a mensagem *“The declaration of ViewObject is disallowed for this location w.r.t. the architecture”* e tipo *“Architectural Violation”*. Essa abordagem inibe o commit de violações arquiteturais e é ilustrada na Figura 6.

Por meio das restrições declaradas com a sintaxe DCL é possível especificar diversos padrões arquiteturais e padrões de projeto para o sistema em desenvolvimento. Veja uma referência na seção **Links** para obter mais informações e recursos avançados sobre o DCLcheck.

A maior vantagem do DCLcheck é não permitir que o desenvolvedor realize seu trabalho fora das restrições arquiteturais, prevenindo a degradação da arquitetura do projeto. Ainda assim é recomendado utilizar a verificação do projeto com alguma outra ferramenta, como o Sonar, pois apesar do DCLcheck prevenir o desenvolvimento fora dos padrões estabelecidos, ele depende de restrições criadas para o projeto e alguma restrição que deveria estar presente, por algum motivo pode passar despercebida. Ao utilizar uma ferramenta de prevenção juntamente com uma ferramenta de verificação, torna-se muito mais difícil que disparidades passem despercebidas pelo projeto.

O trabalho da conformação arquitetural

A conformação arquitetural envolve muito trabalho em ambas as abordagens apresentadas, tanto em um projeto já existente, onde se deseja fazer com que o código fonte se adeque à arquitetura projetada, quanto em um projeto novo, onde deseja-se manter a conformação arquitetural desde o início.

Basicamente, em um projeto existente, o trabalho de conformação será realizado ao longo do tempo, conforme as disparidades forem encontradas e de acordo com a disponibilidade da equipe para corrigi-las. O lado positivo desta abordagem é que, geralmente, a arquitetura do sistema já foi planejada, restando apenas o trabalho de fazer com que o código reflita o projeto. Entretanto, em algumas situações, talvez a conformação do código fonte com o projeto arquitetural não seja mais possível devido a inúmeras restrições e modificações no código ao longo do tempo que não foram refletidas no documento de arquitetura de software. Neste caso, o ideal seria redefinir o documento de arquitetura de software para refletir o código atual.

Em um projeto novo, cujo objetivo seja utilizar a validação antes do commit, existe apenas a premissa de que se tenha um documento de arquitetura de software para que as regras do projeto arquitetural sejam mapeadas na ferramenta de validação antes do início do desenvolvimento. Isso nem sempre é possível devido à natureza dos produtos de software, onde as alterações são comuns, mesmo após a definição formal da arquitetura. É preciso estar preparado para esse cenário de mudanças e estar apto a absorvê-las com rapidez, o que pode ser feito com a utilização das duas técnicas de conformação arquitetural desde o início do projeto. Assim, quando houver uma alteração que modifique a estrutura arquitetural já validada até aquele momento, basta utilizar a verificação do código para encontrar as disparidades de acordo com as novas regras e realizar a correção imediatamente.

Além do desafio da conformação arquitetural em si, outro desafio é manter o que foi implementado e, a partir de então, sempre ter todos os artefatos em sincronia com a realidade.

Infelizmente é comum que essas técnicas sejam adotadas por um período de tempo, chegando a causar alguns efeitos positivos e tornando o ambiente mais maduro e produtivo, mas antes que se chegue ao ápice da produtividade e dos benefícios da conformação arquitetural, tudo o que foi implementado acaba caindo lentamente em desuso até que o cenário de caos volte a se instalar e, assim, o ciclo volte a se repetir. Por isso é importante fazer com que a conformação arquitetural passe do status de apenas técnicas e ferramentas para o nível cultural da organização, tornando-se com isso parte intrínseca do dia a dia do processo de desenvolvimento de software.

A conformação arquitetural é uma técnica muito importante para alinhar código fonte e projeto arquitetural. A implementação de técnicas para lidar com a disparidade entre código e arquitetura é fundamental para um ambiente de desenvolvimento maduro e sustentável, no qual a produtividade e a qualidade possam alcançar patamares elevados. Dentre as técnicas existentes para

lidar com a conformação arquitetural, foram apresentadas duas abordagens: a análise de código com o intuito de identificar pontos no código fonte que não estejam conforme o projeto arquitetural e a prevenção de inclusão de disparidades arquiteturais no código fonte por meio da verificação, em tempo de desenvolvimento, do código.

Além dessas abordagens, há outras alternativas para auxiliar no controle da arquitetura no código fonte. Outra abordagem também bastante utilizada é a verificação do código antes do commit, em uma verificação no chamado pré-commit, que pode ser implementada com um *svn hook*, por exemplo. Esse método é parecido com a verificação em tempo de desenvolvimento e mostra-se também uma opção interessante para validação de código.

Autor



Gabriel Novais Amorim

novais.amorim@gmail.com – blog.gabrielamorim.com

Tecnólogo em Análise e Desenvolvimento de Sistemas (Unifieo) e especialista (MBA) em Engenharia de Software (FIAP). Trabalha com desenvolvimento de software há seis anos e atualmente tem trabalhado no desenvolvimento de soluções SOA sob plataforma Java. Possui as certificações OCJP, OCEJWCD, OCEEJBD, IBM-OOAD, IBM-RUP, CompTIA Cloud Essentials e SOACP. Seus interesses incluem arquitetura de software e metodologias ágeis.



Links:

Arquitetura de software com tempo de expiração.

<http://blog.gabrielamorim.com/arquitetura-de-software-com-tempo-de-expiracao-e-modularidade/>

Documento de Arquitetura de Software.

http://www.wthreex.com/rup/portugues/process/artifact/ar_sadoc.htm

Exemplo de um Documento de Arquitetura de Software.

http://www.wthreex.com/rup/portugues/examples/csports/ex_sad.htm

Análise de código com o Sonar.

<http://nemo.sonarqube.org/dashboard/index/269309>

Site do DCLcheck.

<http://java.llp.dcc.ufmg.br/dclsuite/>

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!



Construindo e otimizando um ambiente de alta disponibilidade – Parte 1

Veja neste artigo como implementar aplicações tolerantes a falhas com Apache Tomcat e Hazelcast

ESTE ARTIGO FAZ PARTE DE UM CURSO

Eventualmente uma aplicação ou sistema pode se tornar crítico a ponto de “nunca” poder parar. As razões pelas quais esses sistemas devem estar sempre disponíveis são as mais diversas, mas em geral pode-se dizer que há questões financeiras e/ou de segurança que podem ser impactadas por eventos de indisponibilidade.

Ao pesquisarmos na internet encontramos inúmeros casos internacionais reportando queda de ações de determinadas empresas devido à indisponibilidade dos sistemas das mesmas. Mas não precisamos ir tão longe para pesquisar por falhas em serviços de grandes corporações. Há alguns anos, mais precisamente em 2009, o sistema de pagamentos de uma grande rede brasileira de cartões de débito e crédito ficou algumas horas fora do ar nas vésperas do Natal, o que fez com que muitos consumidores optassem por utilizar cartões que eram aceitos pela máquina da empresa concorrente, que, coincidentemente ou não, bateu o recorde do volume de transações.

Garantir a disponibilidade de um sistema não é uma tarefa trivial, pois deve-se levar em conta aspectos que vão muito além de componentes de software, como redundância de partes elétricas e outros componentes de infraestrutura como links, firewalls, roteadores e assim por diante.

Fique por dentro

Neste artigo vamos mostrar como construir um ambiente com tolerância a falhas e balanceamento de carga, ou seja, como disponibilizar uma infraestrutura capaz de prover seus serviços mesmo com parte de seus servidores inoperantes.

Naturalmente, à medida que introduzimos componentes para prover redundância a uma solução, aumentamos sua complexidade e potencialmente adicionamos uma sobrecarga para fazer estes componentes conversarem entre si, ou seja, ao melhorar a disponibilidade podemos estar comprometendo o tempo de resposta e o desempenho de uma aplicação.

Ao longo deste artigo vamos explorar como tirar o melhor proveito de soluções como o Apache Web Server e Hazelcast para construir um ambiente de alta disponibilidade que tenta extrair o melhor de cada uma das soluções com o objetivo de minimizar a sobrecarga ocasionada pelas mesmas.

As primeiras tentativas de criar sistemas redundantes em Java eram extremamente complexas e confusas e em geral eram “amarradas” a servidores de aplicação e ao uso das primeiras encarnações dos EJBs, que apresentavam um modelo de publicação complicado e um modelo de desenvolvimento improdutivo.

Hoje temos à nossa disposição dezenas de frameworks que oferecem, além de alta disponibilidade, outros benefícios como escalabilidade e desempenho. Independentemente de qual framework seja adotado, em geral há características comuns que são inseridas quando se quer prover um ambiente de alta disponibilidade através da adição de servidores em redundância: a complexidade

aumenta e há um *overhead* intrínseco que é introduzido por soluções que utilizam a rede como meio de comunicação.

Neste artigo vamos explorar alguns conceitos de infraestrutura envolvidos para construir um ambiente tolerante a falhas, porém focando principalmente no ponto de vista do Software, de forma a instruir o leitor em boas práticas na criação de aplicações que ofereçam alta disponibilidade, avaliando como podemos configurar alguns componentes em particular de maneira otimizada, minimizando o *overhead* ocasionado pela adição de mais servidores e frameworks de replicação de dados.

Nas próximas seções vamos mostrar como configurar o Apache para atuar como平衡ador de carga bem como o Hazelcast, que atuará como solução de replicação da Sessão HTTP, porém primeiramente vamos discutir alguns dos principais conceitos envolvidos na construção desse tipo de ambiente.

Pontos únicos de falha, tolerância a falhas e alta disponibilidade

A grosso modo, podemos dizer que um componente possui tolerância a falhas (*FT-Fault Tolerance*) se o mesmo puder continuar a oferecer seu serviço, mesmo de forma degradada, quando parte de seus subcomponentes encontrarem-se em falha. Para lidar com esse tipo de cenário, ou seja, prover FT a um componente, normalmente empregam-se técnicas de *redundância*, as quais, por sua vez, consistem em prover e configurar partes sobressalentes para os subcomponentes que fazem parte de um equipamento ou serviço.

Como sabemos, criar um ambiente com FT não é uma tarefa trivial e tende a se tornar cada vez mais complexa à medida que novos componentes ou serviços são introduzidos na arquitetura de um sistema. Com o objetivo de lidar com esse problema foi criada uma metodologia para avaliar subcomponentes suscetíveis a interrupções, conhecida como Análise de Ponto Único de Falha (SPOF, Analysis-Single Point of Failure Analysis). No geral, a sigla SPOF refere-se a um subcomponente ou serviço que, em caso de falha, poderá

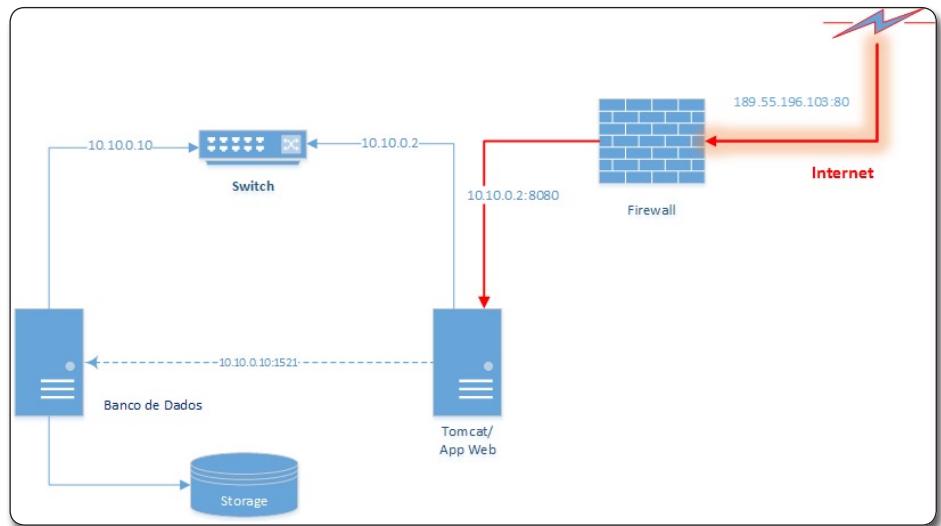


Figura 1. Infraestrutura sem tolerância a falhas

ocasionar a perda total de um componente ou sistema.

Para entender melhor esse conceito, consideremos o diagrama da Figura 1, que representa uma estrutura muito simplificada de um Data Center, sem qualquer redundância em seus serviços, o qual disponibiliza uma aplicação Web que pode ser acessada através da Internet.

Esse Data Center é formado apenas por seis componentes “macro”:

1. Firewall;
2. Servidor Tomcat;
3. Switch de rede LAN;
4. Banco de Dados;
5. Storage.
6. Link Externo (de Internet).

Uma requisição oriunda da Internet irá acessar aplicações web hospedadas no Data Center utilizando o endereço IP do Link Externo (189.55.196.103). Esta requisição chegará no Firewall, que redirecionará os acessos na porta 80 a este IP de Internet para o endereço do servidor Tomcat na rede interna (10.10.0.2:8080). Este, por sua vez, poderá se comunicar com um banco de dados pela mesma rede (10.10.0.10:1521).

Se olharmos de forma “macro” o diagrama da Figura 1, perceberemos que se qualquer um dos seis componentes falhar, o acesso às aplicações hospedadas nos servidores Tomcat não será mais possível. Por outro lado, ao olhar de forma “micro”,

avaliando apenas os servidores de banco e web, notamos que os próprios servidores podem apresentar diversos SPOFs como, por exemplo:

- Fontes de Energia;
- Discos Rígidos;
- Placas de Rede.

Para eliminar estes SPOFs e introduzir FTs aos servidores é necessária uma mudança considerável na infraestrutura. No caso de fontes de alimentação, geralmente basta equipar os servidores com múltiplas fontes de energia que devem ser alimentadas por “tomadas” independentes. Para resolver o problema de falha nos discos, usualmente são empregadas técnicas de virtualização de armazenamento como RAID.

No caso de placas de rede, o problema é um pouco mais complicado, pois além de disponibilizar uma placa adicional é necessário fazer com que as duas interfaces de rede comporte-se como se fossem uma só, utilizando técnicas de virtualização de IP conhecidas como Network Bonding (veja a seção **Links**), que geralmente são oferecidas “out of the box” em algumas distribuições do Linux para servidor. Porém o problema não para por aí. Se não houver também redundância nos switches, apenas 50% do problema estará resolvido. Assim, para oferecer FT na rede seria necessário adquirir switches com a

habilidade de funcionar em redundância e fazer a ligação de cada interface física de rede em um deles, como demonstra a **Figura 2**.

Com uma rede tolerante a falhas, resta eliminar os SPOF no Firewall e nos serviços lógicos. Para atacar o primeiro caso, além de modificar novamente a infraestrutura com a introdução de Firewalls

atuando em redundância, dependemos da aquisição de um Link redundante, o que por sua vez requer uma estrutura de roteamento do fornecedor do Link. Já para eliminar os SPOF atrelados aos servidores de banco e Web, dependemos de soluções particulares para cada um.

No caso dos servidores de Web, que são o foco deste artigo, precisamos levar em

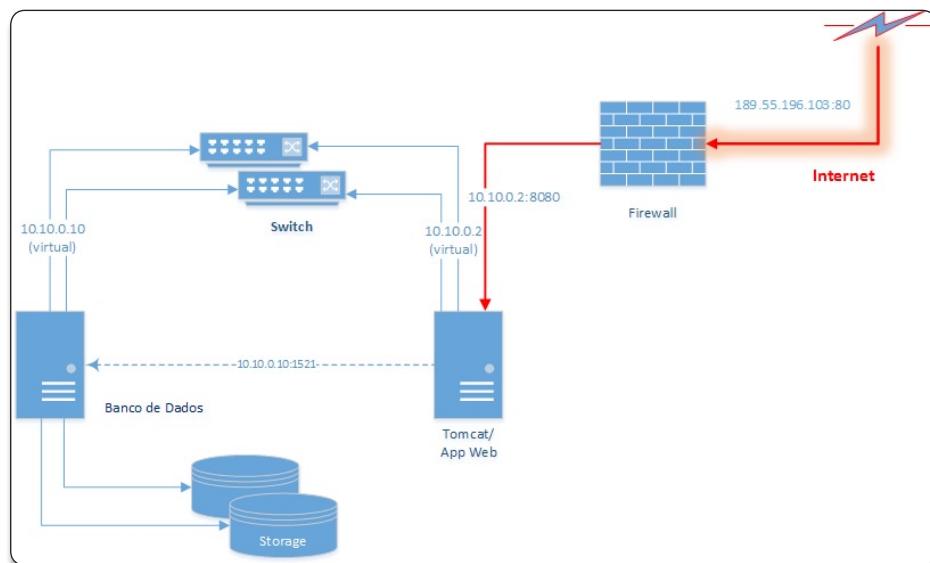


Figura 2. Infraestrutura com tolerância a falhas na rede e storage

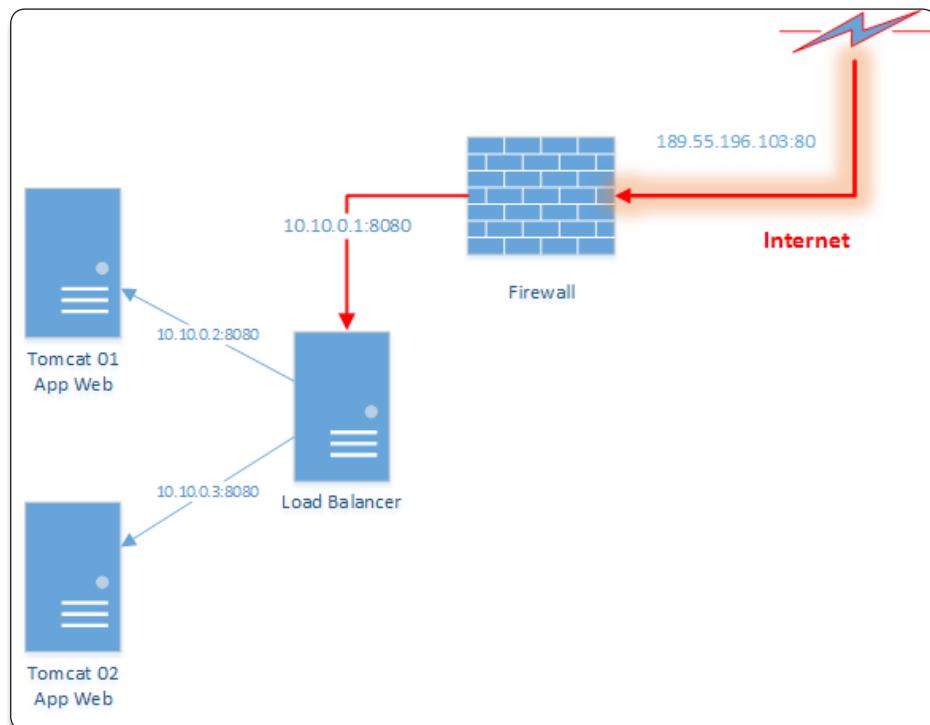


Figura 3. Infraestrutura com balanceamento de carga

conta duas questões para avaliar como vamos atacar o problema do SPOF:

- Se introduzirmos mais servidores, como vamos rotear uma requisição (vinda da Internet) para os mesmos?
- A aplicação publicada nos servidores Web faz uso da Sessão HTTP (stateful) ou não (stateless)?

A primeira pergunta é respondida com uma técnica conhecida como balanceamento de carga (*LB-Load Balancing*), que consiste em colocar um *appliance* ou servidor entre o firewall e os servidores de aplicação que atuará como um proxy, que sabe quais conjuntos de servidores podem efetivamente atender a uma requisição vinda da internet. Esquematicamente podemos representar o balanceamento de carga conforme a **Figura 3**.

A solução de balanceamento de carga resolve a questão de FT dos servidores Web e também a questão de escalabilidade horizontal de uma aplicação, que em teoria, pode passar a atender mais requisições simultâneas à medida que adicionarmos mais servidores Web. Entretanto, a solução introduz um novo SPOF, que é o próprio LB. Como nosso objetivo principal é focar na parte de alta disponibilidade dos servidores que contêm uma aplicação Web, vamos deixar de lado a questão de FT do LB, porém o leitor interessado pode consultar algumas referências na seção **Links**, que mostram como criar clusters de LBs baseadas em IPs virtuais.

Em seguida, passando à segunda questão, somos levados a pensar se devemos nos preocupar em preservar ou não o estado contido em uma sessão HTTP. Se a aplicação não faz uso da *HttpSession*, o que na prática só é verdade no caso de aplicações que não disponibilizam interfaces para interação humana, como Web Services e similares; não há com o que se preocupar. Por outro lado, em geral qualquer aplicação web baseada em frameworks como JSF ou GWT, precisará armazenar dados na sessão de um usuário em algum momento. Nesse caso temos que decidir se a sessão HTTP será replicada entre todos os nós ou não. O fato de replicar a sessão implica que vamos oferecer algo conhecido como

Transparent Failover, o que quer dizer que se um dos servidores Web falhar, o usuário não vai “perceber” que ocorreu um problema e continuará utilizando normalmente a aplicação, por exemplo, sem precisar se logar novamente. Caso optemos por não replicar a sessão, temos um problema a menos para resolver, porém em alguns casos isso pode ser inaceitável.

Neste artigo vamos mostrar como configurar e disponibilizar um ambiente com servidores Web em alta disponibilidade, com balanceamento de carga e replicação de sessão utilizando os seguintes componentes e frameworks:

- Apache Web Server – Balanceador de Carga (LB);
- Tomcat Web Server – Servidor para publicação das aplicações web Java;
- Hazelcast – DataGrid para replicação da Sessão HTTP.

Configurando o Ambiente

Nos próximos tópicos vamos fazer uma breve descrição de cada um desses componentes para entendermos como os mesmos poderão ser integrados para formar uma solução de alta disponibilidade e, em seguida, mostrar como devemos configurá-los de forma que os mesmos possam efetivamente se comunicar.

Apache Web Server

O servidor Apache nasceu praticamente ao mesmo tempo que a linguagem Java (~1994), porém levou um certo tempo para que ambos pudessem ser integrados. O Apache inicialmente era utilizado para disponibilizar as primeiras aplicações web com conteúdo dinâmico baseadas em *CGI* (*Common Gateway Interface*), a qual serviu como alicerce para criação da primeira versão da linguagem PHP.

Atualmente o Apache é disponibilizado em uma estrutura modular, que permite a adição ou remoção de funcionalidades via configuração. Por exemplo, se desejarmos que o tráfego de requisições ocorra de maneira ‘confidencial’, podemos instalar o módulo SSL (*mod_ssl*) para configurar quais caminhos, e.g. */secure*, deverão ser solicitados utilizando HTTPS em vez de HTTP.

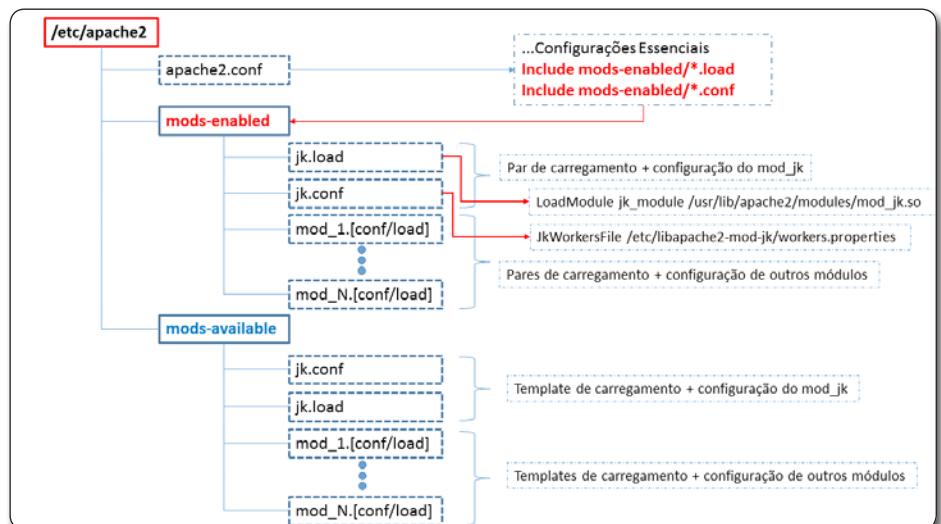


Figura 4. Estrutura dos arquivos de configuração do Apache

Nesta seção vamos focar em configurar apenas um dos módulos do Apache no Linux Mint, o módulo *mod_jk*, que é responsável por fazer o balanceamento de carga entre o Apache e o Tomcat. Porém, vamos fazer algumas observações com respeito aos módulos *prefork* e *worker*, que podem ter impacto na quantidade de requisições e tempo de resposta.

- A seguir estão enumerados os passos para fazer a instalação básica do Apache:
1. Abrir um terminal do Linux;
 2. Autenticar-se como usuário root (executar o comando *su*);
 3. Executar *apt-get install apache2*;
 4. Testar se o Apache foi instalado com o comando *service apache2 start*;
 5. Confirmar que o serviço está executando: *service apache2 status* (a mensagem de saída será *apache2 is running*);
 6. Acessar a URL *http://localhost* em um navegador. Você verá uma “welcome page” do Apache;
 7. Parar o Apache com o comando *service apache2 stop*.

O próximo passo é customizar a instalação do Apache para fazer a configuração inicial do balanceamento de carga. Para isso, primeiramente vamos entender como se dispõe a estrutura de configuração do servidor, que pode ser representada como mostrado na Figura 4

Se acessarmos o diretório no qual o Apache foi instalado (*/etc/apache2*) e listarmos o

Nota:

O módulo *mod_jk* não é o único com a capacidade de realizar o balanceamento de carga e muitos preferem alternativas oferecidas por outros módulos como *mod_proxy*, *mod_proxy_balancer* e *mod_cluster*. A vantagem do *mod_proxy_balancer* é que o mesmo oferece protocolos para realizar a comunicação entre o Apache e servidores por trás do mesmo, ao passo que o *mod_jk* só utiliza o protocolo *ajp*, que é um protocolo de comunicação binário, muito mais eficiente que o HTTP. Além disso, a configuração do *mod_jk* é um pouco mais simples e o mesmo é considerado um LB mais completo e maduro. Ainda assim, atualmente o JBoss recomenda o uso do *mod_cluster* em vez do *mod_jk*, pois o *mod_cluster* possui mecanismos para notificar o Apache com estatísticas reais de utilização de CPU, permitindo regras de平衡amento mais dinâmicas e inteligentes.

seu conteúdo veremos que há um arquivo chamado *apache2.conf*. Neste arquivo são apresentadas as configurações básicas do servidor e também são definidos quais módulos listados no diretório *mods-enabled* serão efetivamente carregados de acordo com a diretiva “Include” da API do Apache.

O diretório *mods-enabled* contém diversos arquivos terminados com os sufixos *.load* ou *.conf*, os quais são responsáveis, respectivamente, por carregar um módulo compilado do Apache e por configurar o comportamento deste módulo. Os módulos compilados do Apache ficam todos em uma pasta chamada *modules*, porém podem ser colocados em qualquer

diretório, desde que a diretiva *LoadModule* contida nos arquivos **.load* aponte para o caminho correto.

Nesta estrutura vemos também que há um diretório chamado *mods-available*, que é muito similar ao *mods-enabled*, porém o seu conteúdo é simplesmente ignorado na inicialização do servidor. Este diretório contém apenas arquivos que podem ser pensados como “templates” de configuração para apresentar módulos do Apache, que devem ser copiados para o diretório *mods-enabled* para serem efetivados. Assim, quando quisermos habilitar um novo módulo, podemos partir de uma configuração localizada em *mods-available*, copiar para *mods-enabled* e customizá-la.

Finalmente, vamos observar de maneira mais detalhada o arquivo de configuração *jk.conf* e modificá-lo para fazer a configuração do balanceamento de carga (*mod_jk*). Inicialmente faremos apenas a configuração do serviço para o monitoramento do balanceamento de carga, pois ainda não temos os nós do Tomcat disponíveis para serem balanceados. Posteriormente vamos revisitar esta configuração para “plugar” as instâncias do Tomcat.

Omitindo os comentários do arquivo *jk.conf*, a configuração do mesmo deverá ficar semelhante à mostrada na **Listagem 1**.

Listagem 1. Configuração do mod_jk.

```
<IfModule jk_module>
    JkWorkersFile /etc/libapache2-mod-jk/workers.properties

    JkLogFile /var/log/apache2/mod_jk.log
    JkLogLevel info

    JkShmFile /var/log/apache2/jk-runtime-status

    JkWatchdogInterval 60

    <VirtualHost localhost:80>
        JkMount /jkstatus* jkstatus
    </VirtualHost>

</IfModule>
```

As configurações do Apache por vezes podem parecer um pouco confusas, misturando XML com texto livre, porém, em geral, quando há uma linha que contém texto simples nos arquivos *.conf*, a mesma se refere à configuração de algum atributo de um módulo, ou seja, cada linha pode ser pensada como um par [chave, valor]. No caso do arquivo *jk.conf*, o significado de cada propriedade “encapsulada” no elemento **<IfModule jk_module>** é apresentado a seguir:

1. JkWorkersFile: arquivo de configuração do balanceamento de carga, como veremos adiante;

2. JkLogFile: Arquivo de log do mod_jk;

3. JkLogLevel: Nível do log;

4. JkShmFile: arquivo que será usado como memória compartilhada do Apache. O Apache, além de threads, coordena diversos subprocessos que, por padrão, têm áreas de memória isoladas

uns dos outros. Quando há necessidade de troca de informação entre os subprocessos, este arquivo é utilizado para ler/escrever dados;

5. JkWatchdogInterval: Monitor do mod_jk. O “WatchDog” é uma thread que executa diversas tarefas de forma periódica, como checar o status das conexões entre o Apache e os servidores em balanceamento e liberar recursos. Este atributo especifica com que periodicidade esta thread deve rodar e por padrão vem configurado com 60 segundos.

O arquivo *jk.conf* ainda apresenta outro conjunto de diretivas, encapsuladas no elemento **VirtualHost**, o qual provê um mecanismo que permite adicionar diversas regras para determinar quem é o responsável por atender a uma requisição em determinado caminho. A diretiva **JkMount** é análoga a uma configuração de um servlet-mapping, na qual definimos um caminho (url-pattern) e uma classe responsável por atender às requisições no mesmo.

No Apache, entretanto, não existe o conceito de “classes” e sim de serviços, e no caso da **Listagem 1**, estamos dizendo que as requisições no caminho */jkstatus** deverão ser atendidas por um serviço *jkstatus*. Este serviço está definido no arquivo especificado pela propriedade **JkWorkersFile**, o qual, por padrão, pode ser encontrado em */etc/libapache2-mod-jk/workers.properties*.

Listagem 2. Configuração do monitoramento em workers.properties.

```
worker.list=jkstatus
worker.jkstatus.type=status
```

A primeira linha da configuração, *worker.list*, declara quais serviços estarão ativos. No caso, temos apenas um serviço, chamado *jkstatus*. Se tivéssemos dois ou mais, bastaria declará-los separados por “,”, e.g., *worker.list=srv01,srv02*.

E o que exatamente é um serviço? No contexto do mod_jk, um serviço é “qualquer coisa” declarada de acordo com a sintaxe *[worker].[serviço].[propriedade]*. O serviço é apenas um “nome” que poderá posteriormente ser referenciado em *worker.list* e o que este serviço faz depende de como configuramos suas propriedades. Na **Listagem 2** criamos um serviço chamado *jkstatus* (poderia ser qualquer outro nome, como *svc*, etc.) e atribuímos ao mesmo, através da propriedade *type*, o tipo **status**. As outras opções para o atributo *type* são **lb** (loadbalancer) e **ajp13**, que referencia um servidor web (Tomcat), como veremos mais à frente. Se tentarmos utilizar qualquer outro valor para a propriedade *type*, que não seja **status**, **lb** ou **ajp3**, o mod_jk não será inicializado.

Para checar se a configuração está correta, basta reiniciar o Apache com o comando *service httpd restart* e acessar a URL *http://localhost/jkstatus*. O resultado deverá ser similar ao da **Figura 5**.

Por enquanto, o serviço de monitoramento não apresenta nenhuma informação relevante, pois ainda não estabelecemos um平衡ador de carga e nós capazes de atenderem as requisições. Para realizar esta tarefa vamos precisar configurar o Tomcat para

que o mesmo possa funcionar como um servidor “por trás” de um Load Balancer, como veremos a seguir.

Tomcat Web Server

Neste artigo vamos utilizar o Tomcat como exemplo de servidor web Java para integrarmos com o Apache, entretanto os mesmos conceitos valem para outros servidores como o Jetty e JBoss/WildFly.

Nosso objetivo é montar um cluster de servidores Tomcat, que normalmente teriam endereços IP distintos e seriam física ou logicamente separados uns dos outros e do Apache, porém para simplificar e para que o leitor que não tem a disponibilidade de múltiplas máquinas possa testar um cluster, vamos trabalhar apenas com o endereço localhost.

Para configurarmos o Tomcat:

1. Criar um diretório *tomcat-cluster*;
2. Baixar a última versão do servidor web do site do Tomcat (8.0.15) e copiá-la para o diretório criado no passo 1;
3. Extrair o arquivo baixado no passo anterior e renomear o diretório para *node01*;
4. Entrar no diretório *node01* e editar o arquivo *conf/catalina.properties*;

4.1 Adicionar a linha *tomcat.route =node01*.

5. Em seguida, editar o arquivo *conf/server.xml*:

5.1 Remover ou comentar os conectores que utilizam a porta 8080;

5.2 Adicionar o atributo *jvmRoute* ao elemento *Engine*. Esse elemento já existe no arquivo e deverá ficar da seguinte forma:

```
<Engine name="Catalina" defaultHost="localhost"
jvmRoute="\${tomcat.route}">
```

5.3 Desabilitar a *SSLEngine* no elemento *Listener*, que também existe no arquivo. Para fazer isto basta modificar o valor de *on* para *off*, como demonstra a configuração a seguir:

```
<Listener className="org.apache.catalina.core.
AprLifecycleListener" SSLEngine="off"/>
```

Feito isso devemos repetir os passos 2 a 4 para o segundo nó do cluster, porém desta

JK Status Manager for localhost:80

The screenshot shows the JK Status Manager interface for the Apache server at port 80. It displays the following information:

- Server Version: Apache/2.4.7 (Ubuntu) mod_jk/1.2.37
- JK Version: mod_jk/1.2.37
- Server Time: Sat, 03 Jan 2015 21:57:04 BRST
- Unix Seconds: 1420329424
- Start auto refresh (every 10 seconds) | Change format XML | [Read Only] [Dump] [S>Show only this worker, E>Edit worker, R=Reset worker state, T=Try worker recovery]
- Legend [Hide]**
 - Name Worker name
 - Type Worker type
 - Route Worker route
 - Act Worker activation configuration ACT=Active, DIS=Disabled, STOP=Stopped
 - State Worker error status OK=OK, ERR=Error with substates IDLE=No requests handled, BUSY=All connections busy, REC=Recovering, PRB=Probing, FRC=Forced Recovery
 - D Worker distance
 - F Load Balance factor
 - M Load Balance multiplicity
 - V Load Balance value
 - Acc Number of requests

Figura 5. Tela de Monitoramento do mod_jk

vez vamos criar um diretório chamado *node02*, nome que também vamos utilizar para o atributo da *jvmRoute*. Como iremos utilizar dois servidores compartilhando o mesmo endereço IP, precisamos que não haja conflitos de portas para os conectores do Tomcat e por isto devemos modificar as mesmas na configuração de *node02*. Sendo assim:

1. Em *server.xml* modifique de 8005 para 8006 o valor da porta de shutdown:

```
<Server port="8006" shutdown="SHUTDOWN">
```

2. No mesmo arquivo, modifique de 8009 para 8010 a porta do conector AJP:

```
<Connector port="8010" protocol="AJP/1.3"
redirectPort="8443"/>
```

Para checar se a configuração está correta basta executar o comando *startup.sh* localizado no diretório *bin* de um dos nós. Se tentarmos acessar a página de administração (ou qualquer outra) do Tomcat pelos endereços *http://localhost:8080* ou *http://localhost:8009*, veremos que não é possível fazê-lo.

No primeiro caso, o acesso falha simplesmente porque nós removemos o conector na porta 8080, portanto não há nenhum socket escutando nesta porta. No segundo caso, o acesso falha porque o socket que está escutando na porta 8009 não entende o protocolo HTTP e sim o protocolo AJP, que

Nota:

Caso o leitor prefira, pode-se utilizar endereços IP virtuais ao invés de modificar as portas. Para criar um IP virtual no Linux basta digitar o comando *ifconfig [interface]:[num] [ip] netmask [máscara] up*, e.g., *ifconfig eth0:1 10.10.0.2 netmask 255.255.255.0 up*. O conector AJP também deverá ser instruído a fazer o bind neste endereço, bastando para isso adicionar o atributo *address=10.10.0.2* ao elemento *Connector*.

é um protocolo binário exclusivo para a comunicação entre o Apache e o Tomcat.

Para podermos ter acesso às aplicações publicadas no Tomcat, as requisições deverão passar pelo Apache primeiro e para isto precisamos configurá-lo como um Load Balancer, conforme veremos a seguir.

Configurando o Apache como um Load Balancer

Com a pré-configuração que realizamos no Apache, modificá-lo para funcionar como um Load Balancer é uma tarefa muito simples. O primeiro passo para isso consiste em editar o arquivo *workers.properties* para criar templates de web servers que passarão por um processo de平衡amento de carga, como mostrado na **Listagem 3**.

O *template_web* definido pode ser pensado como se fosse uma classe abstrata, que será especializada para os nós *node01* e *node02* do Tomcat. O principal atributo deste template é o tipo do worker, no caso *ajp13*,

o que significa que estamos declarando um serviço que se conectará a um servidor e será balanceado.

O próximo passo da configuração de balanceamento consiste em especializar este template, atribuindo valores particulares a propriedades específicas de cada nó, como mostrado na **Listagem 4**.

Duas coisas importantes devem ser destacadas com relação à esta listagem. A primeira delas está ligada aos nomes utilizados para os workers, que devem ser exatamente iguais aos declarados nos atributos *jvmRoute* que foram configurados nos servidores Tomcat.

Listagem 3. Adicionando templates de servidores web em workers.properties.

```
worker.template_web.type=ajp13
worker.template_web.lbfactor=1
worker.template_web.retries=2
worker.template_web.socket_timeout=30
worker.template_web.socket_keepalive=True
worker.template_web.connection_pool_size=16
worker.template_web.connection_pool_timeout=600
worker.template_web.reply_timeout=300000
worker.template_web.max_reply_timeouts=5
worker.template_web.ping_timeout=2000
worker.template_web.ping_mode=A
```

Listagem 4. Especializando um template em workers.properties.

```
worker.node01.reference=worker.template_web
worker.node02.reference=worker.template_web
worker.node01.host=localhost
worker.node01.port=8009
worker.node02.host=localhost
worker.node02.port=8010
```

A segunda é que para especializar um worker precisamos apenas modificar dois atributos: *host* e *port*, graças à configuração com templates que fizemos na **Listagem 3**. Vale ressaltar, no entanto, que a utilização de templates não é obrigatória, ou seja, poderíamos ter declarado individualmente cada atributo a ser utilizado por um worker, e.g., *worker.node01.ping_mode=A*, porém à medida que adicionamos mais e mais nós a utilização de templates facilita muito a manutenção do arquivo.

Por fim, basta criar um Load Balancer para delegar as requisições para estes workers, como mostra a **Listagem 5**.

Como podemos notar, um serviço de Load Balancing, no mod_jk, é declarado como um worker do tipo *lb* e aceita uma lista de nós, separados por vírgula, a serem balanceados. O mod_jk permite que sejam declarados diversos Load Balancers, que além de configurados devem ser declarados como ativos em *worker.list*. Mais para frente vamos discutir a opção *sticky_session*, utilizada para “prender” um cliente em uma instância do Tomcat.

O último passo consiste em configurar em qual caminho de requisições o Apache deverá delegar para o loadbalancer. Para isto precisamos adicionar apenas uma linha, que contém a diretiva de mapeamento de requisição, no arquivo *jk.conf*. Esta diretiva deve ser adicionada dentro do elemento *VirtualHost*, no qual anteriormente declaramos o caminho para o *jkstatus*, como pode ser visto na **Listagem 6**.

Listagem 5. Configurando o LB para node01 e node02.

```
worker.list=jkstatus,loadbalancer
worker.loadbalancer.type=lb
worker.loadbalancer.balance_workers=node01,node02
worker.loadbalancer.sticky_session=true
```

Listagem 6. Configurando o caminho de requisições para o LB.

```
<VirtualHost localhost:80>
    JkMount /jkstatus* jkstatus
    JkMount /* loadbalancer
</VirtualHost>
```

Ao declarar o caminho */** para ser atendido pelo LB estamos dizendo que, por padrão, todas as requisições serão atendidas pelo mesmo, a menos que o caminho da requisição coincida com um caminho mais específico que tenha sido declarado em *jk.conf*.

Ao reiniciar o Apache e acessar novamente a tela de status, veremos uma imagem similar à da **Figura 6**.

Vemos que são exibidas diversas informações para cada um dos nós do Tomcat. A descrição destas informações pode ser encontrada no final da página do serviço *jkstatus*, porém vamos incrementar o significado de algumas a seguir:

- **State:** sinaliza o estado da conexão entre o Apache e o Tomcat. Este atributo é “Lazy”, ou seja, mesmo que o Tomcat esteja fora, se não for feita nenhuma solicitação ao mesmo, através do LB, o estado ficará OK;
- **V:** Conta quantas requisições foram delegadas para um nó em uma janela de tempo recente;
- **Acc:** Informa o número total de requisições delegadas para um nó específico que retornaram com sucesso;
- **Err:** Informa o número de requisições delegadas para um nó específico que retornaram com erro (HTTP status ≥ 400);
- **F:** Indica o “peso” do nó. Este número é um valor que é fixado através da propriedade *lbfactor* (vide **Listagem 3**). Quando o hardware dos servidores que hospedam o Tomcat é diferente, e.g., um nó tem oito processadores e outro apenas quatro, convém modificar este valor para que o servidor com hardware superior receba mais requisições.

Para verificar como estes números se modificam na prática, vamos testar um cenário em que apenas um dos nós do Tomcat está disponível, como descrito a seguir:

1. Certifique-se que todos os nós do Tomcat estão parados;
2. Inicialize apenas o nó node01;
3. Abra uma shell e execute o comando *curl http://localhost/ > /dev/null* (a / após localhost é importante, caso contrário o Apache usará sua welcome page ao invés do平衡ador);
4. Veja que o número de acessos a ambos os nós cresce (coluna ACC) e que o node02 está com status ERR. Verifique também que a taxa de acesso ao node02 cresce em um ritmo inferior em relação ao node01. Isto ocorre porque o Apache não tenta fazer requisições em um servidor detectado como falho com a mesma frequência que o faz em um servidor operacional;

JK Status Manager for localhost:80

Server Version: Apache/2.4.7 (Ubuntu) mod_jk/1.2.37 Server Time: Sat, 03 Jan 2015 22:34:49 BRST
JK Version: mod_jk/1.2.37 Unix Seconds: 1420331689

[Start auto refresh] (every 10 seconds) | [Change format] XML ▾
[Read Only] [Dump] [S=Show only this worker, E>Edit worker, R=Reset worker state, T=Try worker recovery]

Listing Load Balancing Worker (1 Worker) [Hide]

[S|E|R] Worker Status for loadbalancer

Type	Sticky Sessions	Force Sticky Sessions	Retries	LB Method	Locking	Recover	Wait	Time Error	Escalation Time	Max Reply	Timeouts	[Hide]
lb	True	False	2	Request	Optimistic	60	30			0		

Good	Degraded	Bad	Stopped	Busy Max	Busy Next	Maintenance	Last Reset	[Hide]
2	0	0	0	0	53/115		5	

Balancer Members [Hide]

Name	Type	Hostname	Address:Port	Connection Pool	Timeout Connect	Timeout Prepost	Timeout Reply	Timeout Recovery	Retries	Options	Max Packet Size	[Hide]
node01	ajp13	localhost	127.0.0.1:8009	600	2000	2000	300000	2	0		8192	
node02	ajp13	localhost	127.0.0.1:8010	600	2000	2000	300000	2	0		8192	

Name	Act	State	DFMV	Acc	Sess	Err	CER	E	Wr	Rd	Busy	Max Con	Route	RR	Cd	Rs	LRL	
[S E R]	node01	ACT	OK/IDLE	0	1	1	0	0	(0/sec)	0	0	0	(0/sec)	0	0	0	node01	0/05
[S E R]	node02	ACT	OK/IDLE	0	1	1	0	0	(0/sec)	0	0	0	(0/sec)	0	0	0	node02	0/05

Figura 6. Monitorando os nós do LoadBalancer

5. Inicie o node02;
6. Faça novamente algumas requisições à <http://localhost/>. Agora verifique que o *state* do nó que está parado muda para *ERR* e o número de requisições com sucesso (*Acc*) começa a aumentar no nó que está ativo.

Ao efetuar um pequeno número de requisições a <http://localhost/> pode-se verificar que o *state* do nó que está parado muda para *ERR* e o número de requisições com sucesso (*Acc*) começa a aumentar no nó que está ativo.

Nota:

O comando curl é muito útil quando queremos fazer scripts para simular a execução de múltiplas requisições e também para evitar que requisições tenham afinidade com um nó, caso a opção sticky_session esteja habilitada. Atualmente, algumas ferramentas e plugins de desenvolvimento disponíveis para navegadores (e.g. Firebug) oferecem a opção de ‘copiar’ uma requisição no formato curl.

Hazelcast

Como visto no artigo “Hazelcast: Simplificando a computação distribuída”, publicado na Java Magazine 134, o framework Hazelcast é uma solução de *Data Grid* capaz de oferecer escalabilidade horizontal para uma aplicação. Ao passo que o artigo supracitado apresentou uma visão geral sobre a API e estruturas de dados do Hazelcast, neste artigo o objetivo é explorar como podemos aplicar suas features, utilizando-o como uma ferramenta de monitoramento, capaz de oferecer

informações consolidadas de todos os membros de um cluster. Na segunda parte vamos avaliar estratégias para melhorar o desempenho de caches distribuídos e como integrar o data grid a uma aplicação web de modo a prover replicação da sessão HTTP. Ao passo que alguns mecanismos de replicação de sessão requerem configurações específicas para viabilizar a replicação de estado, no Hazelcast isto ocorre de maneira transparente devido à funcionalidade de *backups* “nativa” das estruturas de dados do framework.

Para demonstrar como podemos integrar o Hazelcast com uma aplicação web, vamos construir um protótipo muito simples, capaz de apresentar algumas funcionalidades de monitoramento úteis para administradores de sistemas. Inicialmente vamos utilizar apenas as APIs “core” do Hazelcast e posteriormente vamos mostrar como configurar o framework para prover alta disponibilidade de sessões.

Normalmente, servidores de aplicação possuem interfaces que permitem checar algumas estatísticas da JVM como utilização de memória, tempo gasto em garbage collection, etc. Entretanto, estas estatísticas são apenas locais, ou seja, para verificar o estado de todos os nós, um administrador teria que se logar em cada instância e navegar até determinada tela em cada uma delas. O que queremos fazer é disponibilizar uma visão consolidada de todos os nós que fazem parte de um cluster, de modo que o usuário possa checar a “saúde” de todos os servidores acessando uma única instância, como mostrado na Figura 7

Esse gráfico exibe a utilização da memória em MB (eixo y) ao longo do tempo (eixo x). Para chegarmos nesse ponto vamos precisar agregar alguns componentes à nossa aplicação, representados esquematicamente na **Figura 8**, que detalha o fluxo de uma requisição.

Assim, o ciclo de vida de uma requisição pode ser enumerado como:

1. Cliente solicita estatísticas de determinada natureza, e.g. quantidade de memória em uso ou quantidade de registros em um cache, utilizando Ajax para se comunicar com um serviço REST.

- Vamos utilizar o framework RESTEasy do JBoss para expor os serviços.

2. Ao receber a solicitação, o Apache aplicará sua política de balanceamento de carga para escolher um dos nós;

3. O nó que receber a solicitação coletará suas estatísticas e pedirá aos outros que façam o mesmo, através do Hazelcast.

- Vamos utilizar o Framework de *Executors* do Hazelcast, o qual permite que tarefas sejam submetidas a qualquer nó do cluster, simplificando o conceito de processamento distribuído;

4. A resposta será enviada em formato JSON para o cliente de modo que o mesmo possa trabalhar de maneira simples com ela e popular o gráfico com determinados dados.

- Vamos utilizar a biblioteca JavaScript Flot, que é uma extensão do jQuery, para plotar os gráficos. Para implementar um efeito de gráfico dinâmico cujos valores dos eixos x e y se “movem” ao longo do tempo, vamos utilizar simplesmente a função `setTimeout` do JavaScript para fazer requisições periódicas.

Nota:

Normalmente não é aconselhável utilizar a função `setTimeout` para executar requisições em períodos muito curtos, e.g. um segundo, pois pode-se comprometer a escalabilidade da aplicação. O ideal seria utilizar notificações push ou server side events com WebSockets ou o Atmosphere Framework, porém isso adicionaria uma complexidade extra que vai além do escopo deste artigo.

Cluster Memory Usage

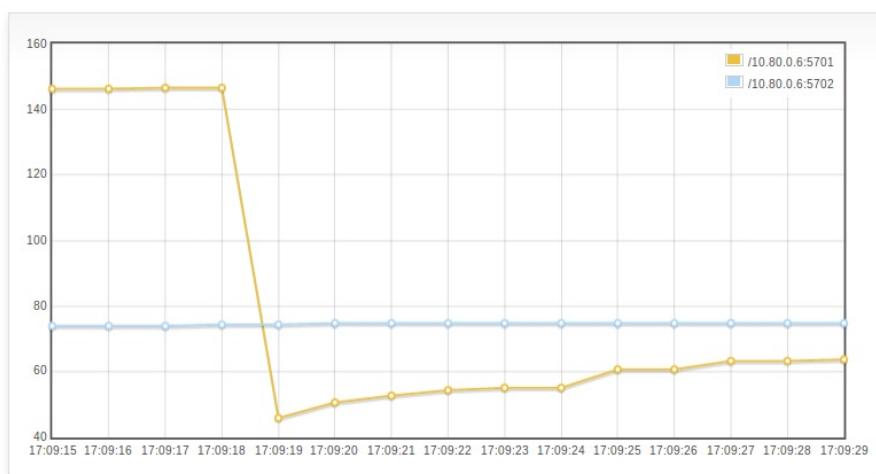


Figura 7. Monitorando servidores de forma centralizada

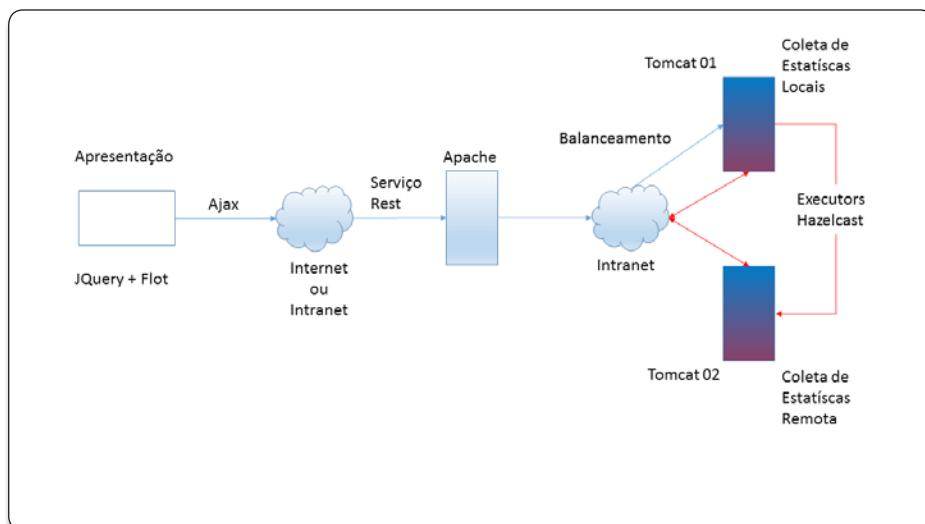


Figura 8. Fluxo de uma requisição

Configurando o Hazelcast

O primeiro passo para construir nossa aplicação consiste em configurar o Hazelcast. Este pode ser inicializado de maneira programática ou através de um arquivo XML, que em geral é mais interessante do ponto de vista de manutenção.

Na **Listagem 7** mostramos a configuração que será utilizada neste artigo e que será estendida, na parte 2, à medida que precisarmos. Esta configuração é feita de forma que possa ser utilizada para compor clusters em uma única máquina com o endereço de *loopback*, ou seja, não é necessário que a máquina esteja conectada a uma rede. Caso o leitor deseje testar a configuração em uma rede e com múltiplas máquinas, basta modificar o valor do IP dentro do elemento *interface* (aninhado em *interfaces*) para o IP da respectiva instância. Por fim, tem-se o elemento *join*, que especifica o uso do protocolo TCP para que membros possam descobrir e se conectar a outros nós durante a inicialização do Hazelcast. A tag *tcp-ip* permite que sejam declarados múltiplos endereços IP, aninhando-se elementos também rotulados de *interface*, para que novas instâncias possam “perguntar” quais nós já estão conectados ao cluster e se unirem aos mesmos. No caso, como trabalharemos com um cluster “local”, basta declararmos um único endereço.

Construindo e otimizando um ambiente de alta disponibilidade – Parte 1

Listagem 7. Configuração do Hazelcast.

```
<hazelcast xmlns="...">
  <group>
    <name>jm-group</name>
    <password>jm-pass</password>
  </group>

  <network>
    <port auto-increment="true">9510</port>
    <interfaces enabled="true">
      <interface>127.0.0.1</interface>
    </interfaces>
    <join>
      <tcp-ip>
        connection-timeout-seconds="10"
        enabled="true"
        <interface>127.0.0.1</interface>
      </tcp-ip>
      <multicast enabled="false" />
      <aws enabled="false" />
    </join>
  </network>
</hazelcast>
```

Para inicializar o Hazelcast a partir da configuração desta listagem, vamos criar uma nova classe com esta responsabilidade. O tipo auxiliar, **HazelcastUtil**, apresentado na **Listagem 8**, além de inicializar o Hazelcast, manterá “em cache” o objeto **HazelcastInstance**, que é a fábrica do framework que disponibiliza os objetos e serviços do data grid. Também vamos manter em cache o endereço do nó que inicializou o Hazelcast, pois esta informação será utilizada mais para frente. Note que o endereço do nó é um **InetSocketAddress**, ou seja, um par [IP:porta] que é distinto para cada processo da JVM que “bootar” o Hazelcast, mesmo executando na mesma máquina.

Criando o modelo de dados e o Serviço

A próxima etapa de construção da nossa aplicação consiste em definir o modelo de dados. Como nosso intuito é prover estatísticas de memória, vamos criar um POJO para encapsular esta informação, como mostrado na **Listagem 9**, que declara apenas dois atributos:

- A quantidade de memória alocada pela JVM;
- A quantidade de memória em uso pela JVM.

Em seguida vamos somar estas informações a dados adicionais que refletem a “origem” e o estado destas estatísticas, encapsulando a classe **MemoryStats** da **Listagem 9** na classe **NodeInfo** da **Listagem 10**. Como veremos a seguir, a classe **NodeInfo** será utilizada como objeto de comunicação entre o servidor e o cliente, e ao encapsular as informações nela podemos futuramente estendê-la com novos atributos como, por exemplo, tempo gasto em Garbage Collection, número de threads em execução, etc., sem alterar a assinatura do serviço que expõe este objeto.

Com o modelo definido podemos partir para a camada de “negócio”. Basicamente precisamos apenas implementar um serviço que retorne as estatísticas de todos os nós em uma única

Listagem 8. Classe para inicialização do Hazelcast.

```
public class HazelcastUtil {

  public static String getLocalAddress() {
    return ADDRESS;
  }

  public static HazelcastInstance hazelcast() {
    return HC;
  }

  static final HazelcastInstance HC;

  static final String ADDRESS;

  static {
    try {
      Config cfg = new UrlXmlConfig(
        HazelcastUtil.class.getClassLoader().getResource("hazelcast-config.xml")
      ).setInstanceName("jm-cluster");

      HC = Hazelcast.getOrCreateHazelcastInstance(cfg);
      ADDRESS = HC.getCluster().getLocalMember().getSocketAddress()
        .toString();
    } catch (IOException e) {
      throw new ExceptionInInitializerError(e.getMessage());
    }
  }
}
```

Listagem 9. POJO para representar o consumo de memória da JVM de um servidor.

```
public class MemoryStats implements Serializable {

  long used;
  long max;

  // Construtor recebe um objeto da API de JMX
  public MemoryStats(MemoryUsage usage) {
    this.used = usage.getUsed();
    this.max = usage.getMax();
  }

  //gets e sets
}
```

Listagem 10. Classe para identificar a qual nó pertence determinada informação.

```
public class NodeInfo implements Serializable {

  //estado: a informação é válida?
  boolean valid;
  //origem: de qual nó pertence as informações encapsuladas nesta classe
  String address;

  MemoryUsage mem;

  //gets e sets
}
```

chamada. Coletar as informações de memória localmente pode ser feito através da API de JMX e estender este processamento para o cluster pode ser feito de maneira muito simples com o framework de processamento distribuído do Hazelcast, como mostra o código da **Listagem 11**.

Listagem 11. Serviço para coletar estatísticas de memória de todos os nós.

```
public class JMXMonitor {  
  
    public static List<NodeInfo> gatherMemoryView() {  
        HazelcastInstance hc = HazelcastUtil.hazelcast();  
  
        IExecutorService service = hc.getExecutorService("executor");  
  
        Callable<NodeInfo> gatherMemTask = (Serializable & Callable<NodeInfo>) () -> {  
            // Este bloco será executado na memória local de cada servidor.  
            MemoryMXBean bean = ManagementFactory.getMemoryMXBean();  
            NodeInfo info = new NodeInfo();  
  
            info.setAddress(HazelcastUtil.getLocalAddress());  
            info.setMem(bean.getHeapMemoryUsage());  
            info.setValid(true);  
  
            return info;  
        };  
  
        Collection<Future<NodeInfo>> values = service.submitToAllMembers(  
            gatherMemTask).values();  
  
        return values.stream().map(f -> {  
            try {  
                return f.get(1, TimeUnit.SECONDS);  
            } catch (Exception e) {  
                // Em caso de timeout ou outra Exception, retorne um objeto com  
                // flag valid=false.  
                return new NodeInfo();  
            }  
        }).collect(Collectors.toList());  
    }  
}
```

Ao realizar esse processamento distribuído no Hazelcast, parece que estamos trabalhando apenas no escopo de um processo local. Isto pode ser verificado nesta listagem, na qual criamos um objeto do tipo **Callable<NodeInfo>** para representar a “tarefa” de captura de estatísticas, que será executada em cada nó que estiver conectado ao cluster. Note que temos que fazer um *cast* para informar que a instância de *Callable* a ser produzida a partir da expressão lambda pela JVM também deve ser serializável, para que possa ser “transferida” para os outros nós.

Com a classe de negócio pronta, basta expô-la como um serviço REST. O primeiro passo para isso consiste em criar uma classe para representar um serviço, que por sua vez poderá ser acessado através de uma URL como *http://<ip>/<contexto>/rest/memory/stats*. Isto poderia ser realizado diretamente na classe **JMXMonitor**, porém para manter o negócio desacoplado da API REST, vamos criar uma nova classe para expor este serviço, como mostra o código da **Listagem 12**.

O último passo consiste em integrar o Framework RESTEasy à nossa aplicação. Isso exige uma configuração no arquivo *web.xml* (ver **Listagem 13**) para especificar o servlet que será responsável por delegar requisições a serviços REST, os quais, por sua vez, deverão ser expostos através da classe **RestApp** (ver **Listagem 14**). Internamente, o RESTEasy consumirá as informações contidas nas

anotações dos beans retornados por **getSingletons()** para criar regras de associação entre os caminhos de URLs e os métodos declarados nesses objetos.

Listagem 12. Expondo a coleta de estatísticas como um serviço REST.

```
//Anotações da API JAX-RS  
import javax.ws.rs.*;  
  
@Path("/memory")  
public class JMXResource {  
  
    @GET  
    @Path("/stats")  
    @Produces("application/json")  
    public List<NodeInfo> stats() {  
        return JMXMonitor.gatherMemoryView();  
    }  
}
```

Listagem 13. Configuração do RESTEasy no web.xml.

```
<web-app xmlns="">  
    <context-param>  
        <param-name>resteasy.scan</param-name>  
        <param-value>false</param-value>  
    </context-param>  
  
    <context-param>  
        <param-name>resteasy.servlet.mapping.prefix</param-name>  
        <param-value>/rest</param-value>  
    </context-param>  
  
    <servlet>  
        <servlet-name>resteasy-servlet</servlet-name>  
        <servlet-class>  
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher  
        </servlet-class>  
        <init-param>  
            <param-name>javax.ws.rs.Application</param-name>  
            <param-value>br.jm.resource.RestApp</param-value>  
        </init-param>  
    </servlet>  
  
    <servlet-mapping>  
        <servlet-name>resteasy-servlet</servlet-name>  
        <url-pattern>/rest/*</url-pattern>  
    </servlet-mapping>  
</web-app>
```

Listagem 14. Registro de serviços REST.

```
public class RestApp extends Application {  
  
    @Override  
    public Set<Object> getSingletons() {  
        HashSet<Object> set = new HashSet<Object>();  
        set.add(new JMXResource());  
  
        return set;  
    }  
}
```

Construindo e otimizando um ambiente de alta disponibilidade – Parte 1

Criando a camada de apresentação

Com a camada de serviços pronta, podemos começar a elaborar a interface para exibir estes dados. O conteúdo HTML da página que vamos criar é muito simples, como pode ser verificado na **Listagem 15**.

Listagem 15. Página HTML que exibirá os gráficos.

```
<html xmlns=...">
<body>
  <div id="header">
    <h2>Cluster Memory Usage</h2>
  </div>

  <div id="content">
    <div class="container">
      <div id="heap" class="placeholder"></div>
    </div>
  </div>
</body>
</html>
```

Em seguida temos que codificar a parte um pouco mais complicada, em JavaScript. Basicamente, o que queremos é criar um efeito de gráfico que se movimente, mostrando os valores da

quantidade de memória utilizada dos 15 últimos segundos. Para fazer isto, temos que executar requisições periodicamente no servidor e traduzir os dados para o formato da API de plotagem que vamos utilizar, o *Flot*.

O Flot exige que os dados de um gráfico sejam expressos como coleções de objetos no formato *label: "nome", options:{...}, data: [[x₀,y₀], [x₁,y₁], ..., [x_N,y_N]]*. No nosso caso, as coordenadas x estarão associadas a timestamps e as coordenadas y à quantidade de memória utilizada naquele timestamp. Para obter este formato, temos que “tratar” as informações vindas do servidor, que serão instâncias da classe **NodeInfo** (**Listagem 10**). A **Listagem 16** exibe o código JavaScript responsável por realizar esta manipulação de dados e atualizar o gráfico representado na **Figura 7**. Como o código é muito extenso, vamos enumerar e comentar brevemente o que o mesmo faz:

- Ao carregar a página, a atualização do gráfico será “agendada” para ocorrer em intervalos de 1s, através da função **setInterval()**, que por sua vez irá executar a função **doChart()**;
- A função **doChart()** irá realizar requisições AJAX no serviço REST codificado na **Listagem 12**;
- Ao receber os dados do servidor, estes serão manipulados para serem apresentados corretamente e para criar o efeito de linha do tempo;

CURSOS ONLINE

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

Feita para Desenvolvedores de Software e DBAs



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>

(21) 3382-5038



Listagem 16. Código JavaScript para obter os dados do servidor e atualizar o gráfico.

```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<link href=".css/style.css" rel="stylesheet" type="text/css">
<script language="javascript" type="text/javascript"
src=".js/jquery.js"></script>
<script language="javascript" type="text/javascript"
src=".js/jquery.flot.js"></script>
<script type="text/javascript">
$(&function() {

    //onde será feita a requisição
    var dataurl = "/server-monitor/rest/memory/stats";

    //número máximo de pontos
    var MAX = 15;

    //marcador de quais nós estão válidos
    var nodes = {};

    //histórico do consumo de memória. Estrutura do tipo Map<String,List<long[]>>,
    //onde a chave é o nome do nó e cada elemento da lista representa um par
    //de coordenadas (x,y), onde x será associado a um número sequencial
    //e y ao valor do consumo de memória.
    var data = {};

    //marcadores do eixo x. Traduzem um índice [0,1,...]
    //para um timestamp em formato hh:mm:ss.
    var ticks = [];

    function doChart() {
        //Obtém os dados do servidor via Ajax.
        $.ajax({
            url : dataurl,
            type : "GET",
            dataType : "json",
            success : onDataReceived
        });

        function checkAvailableNodes(series) {
            nodes = {};
            //consome os dados do servidor e marca quais nós estão OK.
            $(series).each(function(k, v) {
                if (v.valid) {
                    if (!data[v.address]) {
                        data[v.address] = [];
                    }
                    nodes[v.address] = true;
                } else {
                    //se os dados de um nó estiverem inválidos (e.g. ocorreu um
                    //timeout no processamento distribuído), remova-o do //histórico.
                    if (data[v.address]) {
                        delete data[v.address];
                    }
                }
            });
            //consumir dados anteriormente armazenados no histórico.
            //Se algum ficou inválido nesta requisição, removê-lo.
            for (var k in c.data) {
                var cnt = nodes[k];
                if (data[k] && !nodes[k]) {
                    delete data[k];
                }
            }
        }

        //Traduz a resposta do servidor para o formato da API do Flot.
        function doMergedCharts(series) {
            for ( var ix in series) {
                var v = series[ix];
                var n = data[v.address];

                //inicializar o histórico de estatísticas de um nó, caso ainda não //tenha sido feito.
                if (!n) {
                    data[v.address] = n = [];
                }

                //Se atingirmos o limite de pontos, descartamos o mais antigo e
                //subtraímos 1 do índice de cada posição, para dar o efeito
                //que o gráfico está 'andando'
                if (n.length >= MAX) {
                    n.shift();
                    $(n).each(function(a, b) {
                        b[0] = a
                    });
                }

                n.push([ n.length, v.mem.used / 1024 / 1024 ]);
            }
        }

        //formata um número com zeros à esquerda
        function pad(num, size) {
            var s = num + "";
            while (s.length < size)
                s = "0" + s;
            return s;
        }

        //Atualiza as labels do eixo x
        function doTicks() {
            //Estamos aproximando o valor pelo horário do cliente...
            //O correto seria utilizar dados vindos do Servidor.
            var now = new Date();

            ticks = [];

            for (var i = 0; i < MAX; i++) {
                var label = now.getHours() + ":" + pad(now.getMinutes(), 2)
                    + ":" + pad(now.getSeconds(), 2);
                ticks.push([ i, label ]);
                now.setMilliseconds(now.getMilliseconds() + 1000);
            }

            //Replota o gráfico.
            function doPlot() {
                var options = {
                    lines :{
                        show :true
                    },

```

Construindo e otimizando um ambiente de alta disponibilidade – Parte 1

Continuação: Listagem 16. Código JavaScript para obter os dados do servidor e atualizar o gráfico.

```
points : {
    show : true
},
xaxis : {
    ticks : ticks
}
};

var flotData = [];

for (var k in data) {
    var d = {
        bars : {
            show : false
        },
        lines : {
            show : true
        },
        points : {
            show : true
        }
    };
    d.label = k;
    d.data = data[k];
    flotData.push(d);
}

$.plot("#heap", flotData, options);

function onDataReceived(series) {
    checkAvailableNodes(series);
    doMergedCharts(series);
    doTicks();
    doPlot();
}

//Scheduler para atualizar os dados.
setInterval(doChart, 1000);
};

</script>
```

- Uma alternativa para simplificar o código no cliente seria enviar diretamente o JSON no formato esperado, porém isso exigiria que o histórico fosse armazenado no servidor.
- A variável MAX controla o número de pontos que serão mostrados, ou seja, será exibido o valor atual (resultado da última requisição) até MAX-1 pontos anteriores.

Testando a alta disponibilidade e o balanceamento de carga da aplicação

Para testar se a interface funciona ininterruptamente, mesmo quando um dos nós do Tomcat se torna indisponível, basta subir duas (ou mais) instâncias do servidor Web e acessar a interface de monitoramento através do Apache. Logo que um nó se tornar indisponível você verá que os dados dele deixarão de aparecer no gráfico, que continuará sendo continuamente atualizado.

Se observarmos os contadores de requisição do Apache durante o ciclo de requisições, perceberemos que cada nó recebeu aproximadamente o mesmo número de requisições.

Essencialmente, este teste demonstra que esta configuração é suficiente para prover alta disponibilidade para uma aplicação *stateless*, ou seja, para o usuário é indiferente o nó do Tomcat que é acessado, porém o que aconteceria se algum dado estivesse armazenado na Sessão HTTP? Basicamente, este dado se “perderia”, pois estaria atrelado à memória local apenas de um servidor. Para lidar com este tipo de problema precisamos configurar a sessão em alta disponibilidade, o que é uma tarefa simples de se fazer com o Hazelcast, porém há alguns detalhes que devem ser observados para se obter um bom desempenho. Este e outros assuntos serão explorados na segunda parte deste artigo.

Atualmente existem plataformas que oferecem recursos na nuvem que permitem a criação de serviços com escalabilidade e alta disponibilidade de maneira transparente, porém muitas organizações preferem configurar e manter seus próprios datacenters por questões de confidencialidade de dados estratégicos, custo no longo prazo ou até mesmo por obrigações legais.

Independentemente de utilizarmos uma solução pronta ou configurarmos um ambiente para alta disponibilidade, os conceitos envolvidos e o cuidado que deve ser tomado nesses ambientes são similares e sem um plano bem definido, garantir a qualidade e disponibilidade dos serviços pode se tornar uma tarefa cada vez mais complexa à medida que novos componentes são introduzidos em um sistema.

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:

Network Bonding

<http://tinyurl.com/network-bonding>

Indisponibilidade Redecard

<http://tinyurl.com/redecard-unavailable>

mod_jk vs mod_proxy_ajp

<http://tinyurl.com/mod-jk-vs-mod-proxy-ajp>

Cluster Ativo-Ativo de Apache

<http://wp.me/p51BIG-3>

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.

Supporte 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.**

➤ A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer

