



Edição 154 :: R\$ 14,90

 DEV MEDIA

Gráficos no PrimeFaces

Aprenda a utilizar
gráficos para exibir dados na aplicação

Apache Camel: Um guia completo

Conheça o framework e
simplifique a integração entre sistemas

BDD em Java com o framework Spock

Dominando conceitos na teoria e na prática

SPRING + ANGULARJS

Projetando aplicações
diferenciadas para a nuvem

ISSN 1678-836-1



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**

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultora Técnica Anny Caroline (annycarolinegnr@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araújo

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

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

Sumário

06 – Aprenda a trabalhar com gráficos no PrimeFaces

[Eduardo Felipe Zambom Santana e Luiz Henrique Zambom Santana]

[Destaque – Vanguarda](#)

[Artigo sobre Novidades](#)

16 – Criando aplicações com Spring e AngularJS para a nuvem

[Jarbas Lima]

[Artigo no estilo Curso](#)

26 – Apache Camel: Um guia completo – Parte 1

[Rodrigo Cunha Santana]

[Conteúdo sobre Boas Práticas](#)

42 – Especificando e testando em Java com BDD através do Spock

[Willian Oki]

Programador Java: Por onde começar?

Descubra nesse vídeo como entrar na carreira Java com o pé direito!



DEVMEDIA

<http://www.devmedia.com.br/programador-java-por-onde-comecar/33638>

Aprenda a trabalhar com gráficos no PrimeFaces

Desenvolva uma aplicação para cadastro de pesquisas eleitorais com JPA e PostgreSQL e apresente os resultados na forma de gráficos com o PrimeFaces

Um dos principais diferenciais do PrimeFaces em relação aos seus concorrentes é a grande quantidade de componentes disponibilizados, que podem ser utilizados para os mais diversos fins, como para a criação de telas de cadastro, listagens, galerias de imagens, entre outras opções. Adicionalmente, são oferecidos componentes para a criação de gráficos, os quais são bastante simples de serem utilizados.

Para a geração de um gráfico, é necessário apenas informar os dados e alguns parâmetros, como o tipo do gráfico desejado (se ele será um gráfico de barras, linhas, pizza ou outro tipo) e o seu título. Os dados utilizados podem ser recuperados de diversas fontes, como de arquivos texto, criados diretamente na aplicação, coletados de um banco de dados, entre outras opções.

Com o intuito de explorar o uso de gráficos em um projeto web com o PrimeFaces, este artigo demonstrará o desenvolvimento de uma aplicação para cadastro e visualização de pesquisas eleitorais. As informações necessárias para a geração desses gráficos serão recuperadas de um banco de dados PostgreSQL, que será manipulado com o Hibernate. Para possibilitar o cadastro das informações, serão criados dois formulários: um para a inserção dos dados dos candidatos que participam de uma eleição e outro para o cadastro dos resultados das pesquisas de opinião. Por fim, será implementada uma tela para a visualização das pesquisas, na qual serão exibidos gráficos de linhas, barras e pizza.

Fique por dentro

Este artigo é útil por apresentar como adicionar gráficos em sistemas desenvolvidos com o PrimeFaces, biblioteca para a construção de interfaces ricas (e em JSF) com o usuário e que oferece, além de componentes para a criação de telas de cadastro e listagem, opções para a criação de diferentes tipos de gráficos, que podem ser utilizados para a análise de dados ou para a geração de relatórios.

Configuração do JSF e do PrimeFaces

Antes de começar a implementação do projeto, é necessário fazer sua configuração. Em nosso exemplo, para a gestão das dependências será empregado o Apache Maven, como mostra a **Listagem 1**, onde é apresentado o arquivo *pom.xml* com as dependências do PrimeFaces 5.1, do JSF 2.2, do Hibernate e do driver para a conexão com o banco de dados PostgreSQL.

Além das dependências, é necessário criar o arquivo *web.xml*, como mostrado na **Listagem 2**, no diretório *WEB-INF*. Nele devem ser realizadas a configuração do JavaServer Faces, o que é feito com a inclusão do servlet JSF na tag **<servlet>**, e o mapeamento das requisições que serão atendidas por esse servlet, o que é feito na tag **<url-pattern>**.

A última configuração necessária é a definição do banco de dados da aplicação e seus atributos. Nesse projeto utilizaremos o PostgreSQL, SGBD completo, robusto, gratuito e de código fonte aberto. Essa configuração será feita no arquivo *persistence.xml*, que possui todos os parâmetros para a conexão da aplicação com o banco (veja a **Listagem 3**).

Listagem 1. Configuração das dependências do projeto Maven.

```
<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.devmedia.primefaces</groupId>
  <artifactId>graficos</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

  <dependencies>
    <dependency>
      <groupId>org.primefaces</groupId>
      <artifactId>primefaces</artifactId>
      <version>5.1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
```

```
      <groupId>org.glassfish</groupId>
      <artifactId>javax.faces</artifactId>
      <version>2.2.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.1.0.Final</version>
    </dependency>
    <dependency>
      <groupId>postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>9.1-901-1.jdbc4</version>
    </dependency>
  </dependencies>
</project>
```

Listagem 2. Código do arquivo web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <display-name>com.devmedia.primefaces</display-name>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>/index.xhtml</welcome-file>
  </welcome-file-list>
  <error-page>
    <exception-type>javax.faces.application.ViewExpiredException
    </exception-type>
    <location>/index.xhtml</location>
  </error-page>
</web-app>
```

Nesse arquivo são definidas as entidades da aplicação utilizando a tag `<class>` e diversas propriedades para o acesso ao banco utilizando a tag `<property>`, por exemplo a URL, o driver JDBC, o usuário e a senha, se as tabelas do banco de dados devem ser criadas automaticamente e se as consultas SQL devem ser mostradas no console. A Tabela 1 descreve a função de cada parâmetro utilizado em nosso projeto.

Desenvolvendo a aplicação

Iniciaremos o desenvolvimento da aplicação criando o banco de dados e as classes que serão necessárias para manipular as informações. Em seguida, mostraremos como implementar as telas de cadastro e, finalmente, como criar os gráficos.

```
<groupId>org.glassfish</groupId>
<artifactId>javax.faces</artifactId>
<version>2.2.0</version>
<scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.1.0.Final</version>
</dependency>
<dependency>
  <groupId>postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>9.1-901-1.jdbc4</version>
</dependency>
</dependencies>
</project>
```

Listagem 3. Configuração do banco de dados no arquivo persistence.xml.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="devmedia">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <!-- entidades mapeadas -->
    <class>com.devmedia.primefaces.model.Candidato</class>
    <class>com.devmedia.primefaces.model.Pesquisa</class>
    <class>com.devmedia.primefaces.model.CandidatoPesquisa</class>
    <!-- dados da conexão -->
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:postgresql://localhost:5432/eleicao" />
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
      <property name="javax.persistence.jdbc.user" value="postgres" />
      <property name="javax.persistence.jdbc.password" value="eduardo" />
      <!-- propriedades do hibernate -->
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.PostgreSQLDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

Assim, a aplicação possibilitará o cadastro de candidatos e pesquisas eleitorais, e com esses dados poderá gerar gráficos que mostram a porcentagem de intenções de voto de cada candidato e sua evolução nas várias pesquisas realizadas.

Banco de dados e entidades

Para o funcionamento da aplicação serão necessárias três tabelas: *Candidato*, *Pesquisa* e *CandidatoPesquisa*. A primeira será útil para armazenar os dados do político que será candidato na eleição e terá

Aprenda a trabalhar com gráficos no PrimeFaces

| Parâmetro | Descrição |
|---|---|
| javax.persistence.jdbc.url | URL de conexão ao banco de dados. Deve conter o IP do servidor de banco (localhost se for a máquina local) e a porta. |
| javax.persistence.jdbc.driver | Driver JDBC que será utilizado, sempre condizente com o banco de dados da aplicação. |
| javax.persistence.jdbc.user e javax.persistence.jdbc.password | Usuário e senha para acessar o banco de dados. |
| hibernate.dialect | Dialeto da linguagem SQL que será utilizado. Deve ser definido de acordo com o banco de dados da aplicação. |
| hibernate.show_sql | Mostra os comandos SQL no console da aplicação. Esse parâmetro é útil para testes, mas deve ser evitado em produção, pois deixa o sistema mais lento. |
| hibernate.hbm2ddl.auto | Gera automaticamente as tabelas do banco de dados sempre que houver alguma alteração nas entidades. |

Tabela 1. Atributos da JPA no arquivo persistence.xml

as seguintes colunas: *nome*, *partido* e *número do partido*. A segunda servirá para guardar os dados da pesquisa e terá as colunas *data da pesquisa* e *número de pessoas que responderam à pesquisa*. Por fim, a tabela *CandidatoPesquisa* relacionará as duas tabelas anteriores e armazenará a *quantidade de intenções de voto que cada candidato teve em cada pesquisa*. Apesar de utilizarmos um banco de dados relacional, não é necessário definir os scripts SQL para a criação das tabelas, pois a aplicação criará automaticamente as tabelas do banco de dados na primeira vez que for executada.

É necessário implementar as classes que representarão esses dados na aplicação. Como já mencionado, para o acesso aos dados será utilizado o Hibernate. Por isso, essas classes devem receber anotações desse framework, como a `@Entity`, que define que a classe é uma entidade; `@Column`, que transforma um atributo em uma coluna da tabela; `@id`, que define que um atributo da classe é o identificador único da entidade; `@GeneratedValue`, que indica que o valor do campo é gerado automaticamente; e `@SequenceGenerator`, que indica como os valores serão gerados. A Listagem 4 mostra o código da classe **Candidato**, que possui os atributos *id*, *nomeCandidato*, *nomePartido* e *numeroPartido*.

Listagem 4. Código da classe Candidato.

```
package com.devmedia.primefaces.model;  
  
//imports omitidos...  
  
@Entity  
public class Candidato {  
  
    @Id  
    @SequenceGenerator(name="candidato_seq",  
    sequenceName="candidato_seq", allocationSize=1)  
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator="candidato_seq")  
    private int id;  
  
    @Column(name="nome_candidato", nullable=false)  
    private String nomeCandidato;  
  
    @Column(name="nome_partido", nullable=false)  
    private String nomePartido;  
  
    @Column(name="numero_partido", nullable=false)  
    private int numeroPartido;  
  
    //gets e sets...  
}
```

Já a Listagem 5 mostra o código da classe **Pesquisa**, que possui apenas três atributos: *id*, *dataPesquisa* e *numeroEleitores*, atributo que indica a quantidade total de pessoas que responderam à pesquisa e que será útil para calcular a porcentagem de votos de cada candidato. Além das anotações já empregadas na classe **Candidato**, essa entidade utiliza também a `@Temporal`, que é necessária quando o campo a ser persistido for do tipo `Date`.

Listagem 5. Código da classe Pesquisa.

```
package com.devmedia.primefaces.model;  
  
//imports omitidos...  
  
@Entity  
public class Pesquisa {  
  
    @Id  
    @SequenceGenerator(name="pesquisa_seq",  
    sequenceName="pesquisa_seq", allocationSize=1)  
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator="pesquisa_seq")  
    private int id;  
  
    @Column(name="numero_eleitores", nullable=false)  
    private int numeroEleitores;  
  
    @Temporal(TemporalType.DATE)  
    private Date dataPesquisa;  
  
    //gets e sets  
}
```

A Listagem 6, por sua vez, mostra o código da classe **CandidatoPesquisa**, que representa o relacionamento entre as entidades **Pesquisa** e **Candidato**. Além disso, ela ainda possui os atributos *id* e *intencaoVoto*, que recebe a quantidade de votos que um candidato recebeu em uma pesquisa. Nessa entidade ainda são utilizadas as anotações `@ManyToOne`, que mapeia o relacionamento nas tabelas do banco dados, e a `@JoinColumn`, que define o nome da coluna que será a chave estrangeira na tabela.

Camada de acesso a dados

Com as entidades definidas, devemos criar as classes que possuem os métodos que serão responsáveis pelo acesso ao banco de dados. Assim, será criada uma classe para cada entidade da

aplicação e cada uma dessas classes terá um método para salvar e um ou mais métodos para recuperar os dados. A **Listagem 7** mostra a classe de acesso referente à entidade **Candidato**.

Listagem 6. Código da classe **CandidatoPesquisa**.

```
package com.devmedia.primefaces.model;

//imports omitidos...

@Entity
public class CandidatoPesquisa {

    @Id
    @SequenceGenerator(name="cand_pesq_seq",
        sequenceName="cand_pesq_seq", allocationSize=1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator="cand_pesq_seq")
    private int id;

    @ManyToOne(optional=false)
    @JoinColumn(name = "id_candidato")
    private Candidato candidato;

    @ManyToOne(optional=false)
    @JoinColumn(name = "id_pesquisa")
    private Pesquisa pesquisa;

    @Column(name="intencao_voto", nullable=false)
    private int intencaoVoto;

    //gets e sets
}
```

Listagem 7. Código da classe de acesso a dados da entidade **Candidato**.

```
package com.santana.primefaces.dao;

//imports omitidos...

public class CandidatoDAO {

    private EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("devmedia");
    private EntityManager em = factory.createEntityManager();

    public boolean salvar(Candidato candidato) {
        try {
            em.getTransaction().begin();
            em.persist(candidato);
            em.getTransaction().commit();
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    public List<Candidato> getAll() {
        return em.createQuery("Select candidato from Candidato candidato")
            .getResultList();
    }
}
```

Note que é criado o **EntityManager** logo no início da classe utilizando o nome da unidade de persistência definida no arquivo *persistence.xml*. Com esse objeto é possível executar operações como salvar uma entidade ou recuperar uma lista de objetos do banco. Assim, foram criadas as funções **salvar()**, que persiste um objeto do tipo **Candidato**, e **getAll()**, que retorna uma lista com todos os candidatos cadastrados.

A **Listagem 8** mostra o código da classe de acesso referente à entidade **Pesquisa**. Do mesmo modo, essa classe também possui os métodos **salvar()** e **getAll()**.

Listagem 8. Código da classe de acesso a dados da entidade **Pesquisa**.

```
package com.santana.primefaces.dao;

//imports omitidos...

public class PesquisaDAO {

    private EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("devmedia");
    private EntityManager em = factory.createEntityManager();

    public boolean salvar(Pesquisa pesquisa) {
        try {
            em.getTransaction().begin();
            em.persist(pesquisa);
            em.getTransaction().commit();
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    public List<Pesquisa> getAll() {
        return em.createQuery("Select pesquisa from Pesquisa pesquisa")
            .getResultList();
    }
}
```

Por fim, também devemos implementar a classe de acesso a dados para **CandidatoPesquisa**. Como apresentado na **Listagem 9**, ela possui os métodos **salvar()**, para persistir um candidato relacionado a uma pesquisa; **getAll()**, que retorna todos os registros relacionados à classe **CandidatoPesquisa** salvos no banco de dados; **getByPesquisa()**, que retorna os registros relacionados a uma pesquisa; e **getByPesquisaAndCandidato()**, que retorna um objeto **CandidatoPesquisa** referente a um candidato e uma pesquisa.

Telas de cadastro e de listagem

Para conseguir gerar os gráficos, é necessário primeiro cadastrar candidatos e pesquisas eleitorais. Sendo assim, foram criadas duas telas: uma para o cadastro de candidatos e outra para o cadastro de pesquisas, na qual também serão cadastrados os relacionamentos entre candidatos e pesquisas.

Iniciaremos a implementação do cadastro com a criação do managed bean, que nada mais é do que uma classe que permite a comunicação do código Java com as telas que serão apresentadas ao usuário.

Aprenda a trabalhar com gráficos no PrimeFaces

Listagem 9. Código da classe de acesso a dados da entidade CandidatoPesquisa.

```
package com.santana.primefaces.dao;  
  
//imports omitidos...  
  
public class CandidatoPesquisaDAO {  
  
    private EntityManagerFactory factory = Persistence.createEntityManagerFactory  
        ("devmedia");  
    private EntityManager em = factory.createEntityManager();  
  
    public boolean salvar(CandidatoPesquisa candidatoPesquisa) {  
        try {  
            em.getTransaction().begin();  
            em.persist(candidatoPesquisa);  
            em.getTransaction().commit();  
            return true;  
        } catch (Exception e) {  
            e.printStackTrace();  
            return false;  
        }  
    }  
  
    public List<CandidatoPesquisa> getAll() {  
        return em.createQuery("Select cp from CandidatoPesquisa cp").getResultList();  
    }  
  
    public List<CandidatoPesquisa> getByPesquisa(Pesquisa pesquisa) {  
        return em.createQuery("Select cp from CandidatoPesquisa cp where cp.pesquisa  
.id = :idPesquisa")  
            .setParameter("idPesquisa", pesquisa.getId())  
            .getResultList();  
    }  
  
    public CandidatoPesquisa getByPesquisaAndCandidato(Pesquisa pesquisa,  
Candidato candidato) {  
        return (CandidatoPesquisa) em.createQuery("Select cp from CandidatoPesquisa  
cp where cp.pesquisa.id = :idPesquisa and cp.candidato.id = :idCandidato")  
            .setParameter("idPesquisa", pesquisa.getId())  
            .setParameter("idCandidato", candidato.getId())  
            .getSingleResult();  
    }  
}
```

A **Listagem 10** mostra o código do managed bean que será utilizado na tela de cadastro e listagem de candidatos.

Em **CandidatoManagedBean** podemos verificar dois métodos: **cadastraCandidato()**, que chama o método para cadastro do candidato no banco de dados — **candidatoDAO.salvar()** — passando as informações fornecidas pelo usuário e retorna uma mensagem indicando que a operação foi realizada com sucesso; e **getCandidatos()**, que retorna uma lista com todos os candidatos cadastrados.

Listagem 10. Managed bean para cadastro e listagem de Candidatos.

```
package com.devmedia.primefaces.mb;  
  
//imports omitidos...  
  
@ManagedBean(name = "CandidatoMB")  
@ViewScoped  
public class CandidatoManagedBean {  
  
    private CandidatoDAO candidatoDAO = new CandidatoDAO();  
  
    private Candidato candidato = new Candidato();  
  
    public void cadastraCandidato() {  
        candidatoDAO.salvar(candidato);  
        FacesContext.getCurrentInstance().addMessage(null,  
            new FacesMessage(FacesMessage.SEVERITY_INFO, "Sucesso!",  
                "Candidato Cadastrado com Sucesso!"));  
    }  
  
    public List<Candidato> getCandidatos() {  
        return candidatoDAO.getAll();  
    }  
  
    // gets e sets
```

A partir disso, vamos criar a página com o formulário de cadastro de candidatos (vide **Listagem 11**). Note que existe um campo no formulário para cada atributo definido em **Candidato**. Ainda nessa tela, são utilizadas as seguintes tags do PrimeFaces:

- **<p:messages>**: exibe mensagens na tela enviadas pelo managed bean;
- **<p:panelGrid>**: cria uma tabela com os componentes do formulário;
- **<p:inputText>**: cria os campos de texto para o formulário;
- **<p:outputLabel>**: apresenta um rótulo para cada campo do formulário;
- **<p:commandButton>**: renderiza um botão para efetuar a ação de cadastrar o candidato.

A **Figura 1** mostra a tela de cadastro de candidatos renderizada. Note que ela é bastante simples e apenas exibe um formulário para o preenchimento dos dados do candidato. Como o id será gerado automaticamente pelo PostgreSQL, não precisa ser informado.

| | |
|--------------------|----------------------|
| Nome: | <input type="text"/> |
| Partido: | <input type="text"/> |
| Número Partido: | <input type="text"/> |
| ★ Cadastrar | |

Figura 1. Tela de cadastro de candidatos

Além disso, criamos a tela para listagem dos candidatos. A **Listagem 12** mostra seu código, no qual foi adicionado um **<p:dataTable>**. Nesse DataTable, para cada atributo da classe foi criada uma coluna com a tag **<p:column>**, e dentro dessa coluna foi inserido o dado que desejamos exibir. A **Figura 2** mostra a tela de listagem de candidatos renderizada.

Para o cadastro e listagem das pesquisas foram desenvolvidas classes similares às implementadas para a entidade **Candidato**. A **Listagem 13** apresenta o código do managed bean **PesquisaManagedBean**, que possui um construtor que inicializa as entidades **CandidatoPesquisa** para todos os candidatos cadastrados. Isso é necessário para o cadastro da pesquisa pois nessa tela o usuário digitará a quantidade de votos que cada candidato recebeu.

Além disso, foi implementado o método **cadastraPesquisa()**, responsável por salvar os objetos das entidades **Pesquisa** e **CandidatoPesquisa** no banco de dados.

| Nome | Partido | Número Partido |
|---------|---------|----------------|
| José | PAX | 15 |
| João | PAB | 25 |
| Eduardo | XYZ | 55 |

Figura 2. Tela de listagem dos candidatos cadastrados

Listagem 11. Tela de cadastro de candidatos (`cadastrarCandidato.xhtml`).

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">
</head>
</head>
<h:body>
    <h:form>
        <p:messages id="messages" />
        <p:panelGrid columns="2">
            <p:outputLabel for="nomeCandidato" value="Nome:" />
            <p:inputText id="nomeCandidato"
                         value="#{CandidatoMB.candidato.nomeCandidato}" />
            <p:outputLabel for="numeroPartido" value="Partido:" />
            <p:inputText id="numeroPartido"
                         value="#{CandidatoMB.candidato.numeroPartido}" />
            <p:outputLabel for="numeroPartido" value="Número Partido:" />
            <p:inputText id="numeroPartido"
                         value="#{CandidatoMB.candidato.numeroPartido}" />
            <p:commandButton value="Cadastrar" icon="ui-icon-star"
                             action="#{CandidatoMB.cadastraCandidato}" update="messages">
                </p:commandButton>
        </p:panelGrid>
    </h:form>
</h:body>
</html>
```

Figura 12. Tela de listagem de Candidatos (`listaCandidato.xhtml`).

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">
</head>
</head>
<h:body>
    <p:dataTable var="candidato" value="#{CandidatoMB.candidatos}"
                  style="width:800px;">
        <p:column headerText="Nome">
            <h:outputText value="#{candidato.nomeCandidato}" />
        </p:column>
        <p:column headerText="Partido">
            <h:outputText value="#{candidato.nomePartido}" />
        </p:column>
        <p:column headerText="Número Partido">
            <h:outputText value="#{candidato.numeroPartido}" />
        </p:column>
    </p:dataTable>
</h:body>
</html>
```

Como os gráficos serão mostrados na tela de listagem de pesquisas, na classe **PesquisaManagedBean** ainda foram implementados os métodos para a geração dos gráficos. Porém, para facilitar a explicação, eles serão apresentados separadamente na próxima seção.

Com o managed bean implementado, é possível desenvolver a tela de cadastro de pesquisa, a qual é dividida em duas partes. Na primeira, temos o formulário para cadastro dos dados da pesquisa (o número total de eleitores e a data de realização), e, na segunda, temos uma tabela onde podemos informar o total de votos que cada candidato cadastrado recebeu.

A **Listagem 14** mostra o código dessa tela. Observe que foi declarado um **<p:panelGrid>** para a primeira parte do cadastro e um **<p:dataTable>** para a segunda parte. Além dessas tags e das outras já mencionadas, utilizamos o componente **<p:calendar>** para exibir um calendário quando o usuário for preencher a data de realização da pesquisa. A **Figura 3** mostra essa tela.

| Número de Eleitores | <input type="text" value="0"/> | |
|---------------------|--------------------------------|--------------|
| Data: | <input type="text"/> | |
| Nome Candidato | Número Partido | Número Votos |
| José | 15 | 0 |
| João | 25 | 0 |
| Eduardo | 55 | 0 |

Figura 3. Tela para cadastro de pesquisas

Listagem 13. Managed bean para cadastro e listagem de pesquisas.

```
package com.devmedia.primefaces.mb;
//imports omitidos...
@ManagedBean(name = "PesquisaMB")
@ViewScoped
public class PesquisaManagedBean {
    private PesquisaDAO pesquisaDAO = new PesquisaDAO();
    private CandidatoDAO candidatoDAO = new CandidatoDAO();
    private CandidatoPesquisaDAO candidatoPesquisaDAO =
        new CandidatoPesquisaDAO();
    private Pesquisa pesquisa = new Pesquisa();
    private List<CandidatoPesquisa> candidatosPesquisa =
        new ArrayList<CandidatoPesquisa>();
    private Pesquisa selecionada;
    public PesquisaManagedBean() {
        List<Candidato> candidatos = candidatoDAO.getAll();
        for (Candidato candidato : candidatos) {
            CandidatoPesquisa cp = new CandidatoPesquisa();
            cp.setCandidato(candidato);
            cp.setPesquisa(pesquisa);
            candidatosPesquisa.add(cp);
        }
    }
    public void cadastraPesquisa() {
        pesquisaDAO.salvar(pesquisa);
        for (CandidatoPesquisa cp : candidatosPesquisa) {
            candidatoPesquisaDAO.salvar(cp);
        }
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_INFO,
                "Sucesso!", "Informação sobre candidato cadastrada com sucesso!"));
    }
    // gets e sets
    // métodos para a geração dos gráficos omitidos
}
```

Aprenda a trabalhar com gráficos no PrimeFaces

Desenvolvendo os gráficos

Com os candidatos, pesquisas e seus relacionamentos cadastrados, é possível gerar os gráficos a partir dos resultados das pesquisas. Nesse projeto serão criados três tipos de gráficos: um de linha, que mostra a evolução de um candidato em cada pesquisa realizada; um de pizza, que mostra a porcentagem de cada candidato em uma pesquisa; e um gráfico de barras, que mostra o número de votos de um candidato em uma pesquisa.

Listagem 14. Tela para cadastro de pesquisas (cadastrarPesquisa.xhtml).

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">
<h:head>
</h:head>
<h:body>
<h:form>
    <p:messages id="messages" />
    <p:panelGrid columns="2">
        <p:outputLabel for="numeroEleitores" value="Número de Eleitores" />
        <p:inputText id="numeroEleitores"
                     value="#{PesquisaMB.pesquisa.numeroEleitores}" />
        <p:outputLabel for="dataPesquisa" value="Data:" />
        <p:calendar id="dataPesquisa"
                     value="#{PesquisaMB.pesquisa.dataPesquisa}" />
    </p:panelGrid>
    <br><br>
    <p: dataTable var="cp" value="#{PesquisaMB.candidatosPesquisa}"
                   style="width:800px;">
        <p:column headerText="Nome Candidato">
            <h:outputText value="#{cp.candidato.nomeCandidato}" />
        </p:column>
        <p:column headerText="Número Partido">
            <h:outputText value="#{cp.candidato.numeroPartido}" />
        </p:column>
        <p:column headerText="Número Votos">
            <h:inputText value="#{cp.intencaoVoto}" />
        </p:column>
    </p: dataTable>
    <br><br>
    <p:commandButton value="Cadastrar" icon="ui-icon-star"
                      action="#{PesquisaMB.cadastraPesquisa}" update="messages">
    </p:commandButton>
</h:form>
</h:body>
</html>
```

Listagem 15. Método para a geração do gráfico de pizza.

```
public PieChartModel getGraficoPizzaPesquisa() {
    if (pesquisaSelecionada != null) {
        PieChartModel graficoPizzaPesquisa = new PieChartModel();
        List<CandidatoPesquisa> listaCandidatos = candidatoPesquisaDAO
            .getByPesquisa(pesquisaSelecionada);
        for (CandidatoPesquisa cp : listaCandidatos) {
            graficoPizzaPesquisa.set(cp.get Candidato().getNomeCandidato(),
                                     cp.getIntencaoVoto());
        }
        graficoPizzaPesquisa.setTitle("Resultados da Pesquisa");
        graficoPizzaPesquisa.setLegendPosition("w");
        return graficoPizzaPesquisa;
    } else {
        return null;
    }
}
```

Para a geração de cada gráfico é necessário adicionar um método no managed bean de pesquisas. A função desses métodos é recuperar os dados do relacionamento entre **Candidato** e **Pesquisa** e gerar o modelo de dados adequado para cada tipo de gráfico. A **Listagem 15** mostra o código do método **getGraficoPizzaPesquisa()**.

Nesse código, inicialmente é verificado se alguma pesquisa foi selecionada na tabela de listagem de pesquisas. Caso positivo, é criado um modelo de dados para gráficos de pizza com a **PieChartModel**, classe do PrimeFaces. Em seguida, são recuperados todos os candidatos participantes da pesquisa selecionada utilizando o método **getByPesquisa()** da classe **CandidatoPesquisaDAO** e é feita uma iteração sobre esses candidatos para adicionar cada um deles no gráfico, através o método **set**, passando como parâmetros o nome e a quantidade de votos que ele recebeu. O gráfico de pizza calcula as porcentagens automaticamente. Ao final do método são definidos o título do gráfico e a posição da legenda (o valor **w** define que a legenda ficará à esquerda do gráfico).

De modo semelhante, para criar o gráfico de barras também é necessário definir um modelo de dados, neste caso utilizando a classe **BarChartModel**. O restante do código é praticamente igual ao da geração do gráfico de pizza (veja a **Listagem 16**). Note que para a geração desses dois gráficos é necessário escolher uma pesquisa — através de uma listagem — que será armazenada na variável **pesquisaSelecionada**.

Listagem 16. Método para a geração do gráfico de barras.

```
public BarChartModel getGraficoBarraPesquisa() {
    if (pesquisaSelecionada != null) {
        BarChartModel model = new BarChartModel();
        List<CandidatoPesquisa> listaCandidatos = candidatoPesquisaDAO
            .getByPesquisa(pesquisaSelecionada);
        ChartSeries candidatos = new ChartSeries();
        candidatos.setLabel("Candidatos");
        for (CandidatoPesquisa cp : listaCandidatos) {
            candidatos.set(cp.get Candidato().getNomeCandidato(), cp.getIntencaoVoto());
        }
        model.addSeries(candidatos);
        return model;
    } else {
        return null;
    }
}
```

Para a implementação do último gráfico utilizamos os métodos que recuperam todos os candidatos e todas as pesquisas cadastradas, **pesquisaDAO.getAll()** e **candidatoDAO.getAll()**, respectivamente. Em seguida, é feita uma iteração sobre todos os candidatos e, para cada um, é recuperado o seu desempenho e é calculada a porcentagem de votos em cada pesquisa.

Para a geração do modelo de dados do gráfico de linhas é utilizada a classe **LineChartModel**. A **Listagem 17** mostra esse código.

Com esses métodos implementados, exibir os gráficos na nossa página é bastante simples: basta utilizar a tag **<p:chart>** do

PrimeFaces e configurar dois atributos: **type**, que indica o tipo de gráfico que deve ser gerado (ex.: **bar**, **pie** e **line**); e **model**, que indica o método do managed bean que retorna o modelo de dados. Além desses atributos, também é utilizado o **rendered**, que indica que os gráficos de pizza e de barras não devem ser gerados até que o usuário selecione uma pesquisa.

A **Listagem 18** mostra o código da tela que apresenta as pesquisas cadastradas e os gráficos. Notem que o gráfico de linhas foi inserido logo abaixo da listagem de pesquisas e os gráficos de pizza e barras foram inseridos dentro de uma caixa de diálogo, que é apresentada quando o usuário seleciona a pesquisa que deseja analisar. O resultado pode ser verificado na **Figura 4**.

Listagem 17. Método para a geração do gráfico de linhas que mostra a evolução dos candidatos.

```
public LineChartModel getGraficoLinhaTempo() {
    LineChartModel lineModel = new LineChartModel();
    lineModel.setShowPointLabels(true);
    List<Pesquisa> pesquisas = pesquisaDAO.getAll();
    List<Candidato> candidatos = candidatoDAO.getAll();
    for (Candidato candidato : candidatos) {
        ChartSeries linha = new ChartSeries();
        linha.setLabel(candidato.getNomeCandidato());
        for (int i = 0; i < pesquisas.size(); i++) {
            Pesquisa pesquisa = pesquisas.get(i);
            CandidatoPesquisa cp = candidatoPesquisaDAO
                .getByPesquisaAndCandidato(pesquisa, candidato);
            // calcula a porcentagem de cada candidato para a pesquisa.
            linha.set(pesquisa.getId(), (cp.getIntencaoVoto() * 100 /
                pesquisa.getNumeroEleitores()));
        }
        lineModel.addSeries(linha);
    }
    lineModel.setLegendPosition("w");
    return lineModel;
}
```

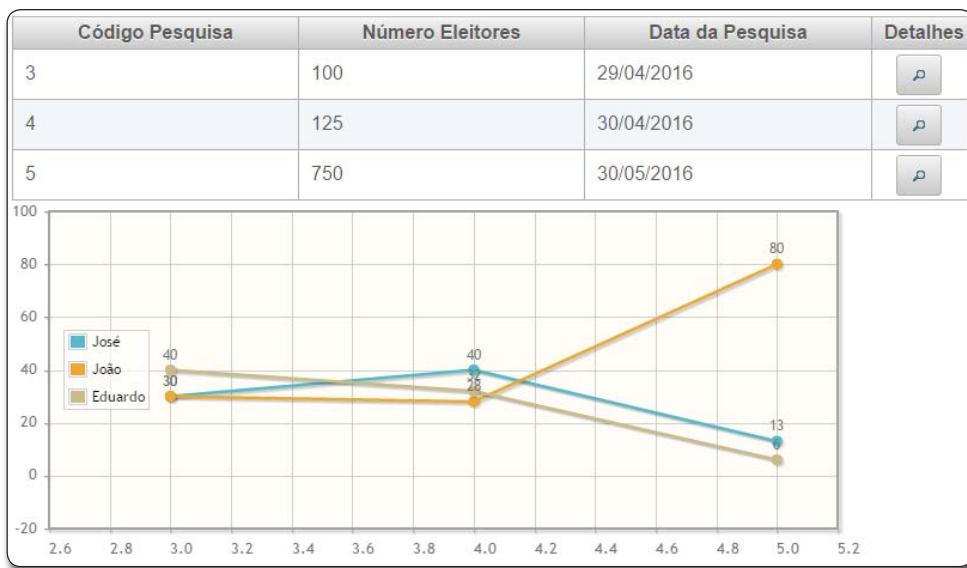


Figura 4. Tela de listagem de pesquisas e o gráfico de linha

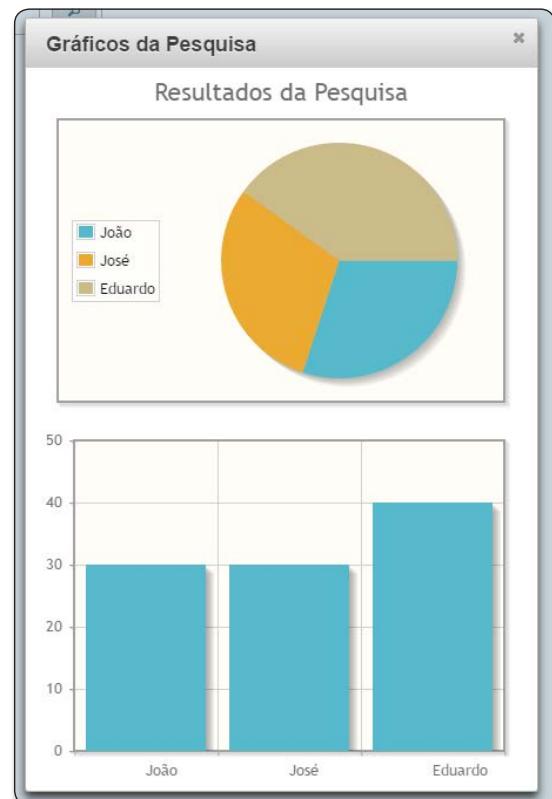


Figura 5. Gráficos de pizza e barras para cada pesquisa

Já a **Figura 5** mostra os gráficos de pizza e de barras, específicos para cada pesquisa. Esses gráficos são exibidos em um pop-up assim que o usuário clica no botão de detalhes de uma pesquisa.

Opções para os gráficos

O PrimeFaces disponibiliza ainda alguns recursos para a visualização dos gráficos, como a possibilidade de dar zoom em áreas específicas e adicionar efeitos de animação. A **Listagem 19** mostra o código que habilita o zoom e adiciona o efeito de animação no gráfico de linha. Como é possível notar, pouquíssimo código foi necessário para a inclusão dessas funcionalidades.

Apenas foi preciso chamar os métodos **setAnimate()** e **setZoom()** da classe **LineChartModel** passando o valor **true** como parâmetro.

Outra opção é a combinação de dois gráficos em uma só figura. Por exemplo, no gráfico de linhas pode ser adicionado um gráfico de barras mostrando o número de eleitores que participaram da pesquisa.

Aprenda a trabalhar com gráficos no PrimeFaces

Listagem 18. Tela de listagem de pesquisas (onde são exibidos os gráficos).

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">

<h:head>
</h:head>
<h:body>
<h:form id="form">
    <p:dataTable var="pesquisa" value="#{PesquisaMB.pesquisas}"
        style="width:800px;">
        <p:column headerText="Código Pesquisa">
            <h:outputText value="#{pesquisa.id}" />
        </p:column>
        <p:column headerText="Número Eleitores">
            <h:outputText value="#{pesquisa.numeroEleitores}" />
        </p:column>
        <p:column headerText="Data da Pesquisa">
            <h:outputText value="#{pesquisa.dataPesquisa}">
                <f:convertDateTime pattern="dd/MM/yyyy" />
            </h:outputText>
        </p:column>
    </p:dataTable>
    <h:commandButton update="form:graficos"
        oncomplete="PF('pesquisaDialog').show()" icon="ui-icon-search"
        title="View">
        <f:setPropertyActionListener value="#{pesquisa}" target="#{PesquisaMB.pesquisaSelecionada}" />
    </h:commandButton>
</h:body>
</html>
```

A Listagem 20 mostra o código para a combinação de dois gráficos. Para essa função, é utilizada a classe **CartesianChartModel**, que possibilita a adição de dados em diversos modelos. Depois, basta criar cada um dos gráficos desejados como exemplificado nessa listagem, que no começo cria o gráfico de barras e depois, o gráfico de linhas.

Listagem 19. Adicionando zoom e animação no gráfico de linhas.

```
public LineChartModel getGraficoLinhaTempo() {
    LineChartModel lineModel = new LineChartModel();
    lineModel.setShowPointLabels(true);
    List<Pesquisa> pesquisas = pesquisaDAO.getAll();
    List<Candidato> candidatos = candidatoDAO.getAll();
    for (Candidato candidato : candidatos) {
        ChartSeries linha = new ChartSeries();
        linha.setLabel(candidato.getNomeCandidato());
        for (int i = 0; i < pesquisas.size(); i++) {
            Pesquisa pesquisa = pesquisas.get(i);
            CandidatoPesquisa cp = candidatoPesquisaDAO.getByPesquisaAndCandidato(pesquisa, candidato);
            // calcula a porcentagem de cada candidato para a pesquisa.
            linha.set(pesquisa.getId(), (cp.getIntencaoVoto() * 100 / pesquisa.getNumeroEleitores()));
        }
        lineModel.addSeries(linha);
    }
    lineModel.setLegendPosition("w");
    // adiciona zoom e animação aos gráficos
    lineModel.setAnimate(true);
    lineModel.setZoom(true);
    return lineModel;
}
```

Listagem 20. Método que cria o modelo de dados dos gráficos de barras e linhas.

```
public CartesianChartModel getGraficoBarraLinha() {
    CartesianChartModel combinedModel = new BarChartModel();
    List<Pesquisa> pesquisas = pesquisaDAO.getAll();
    // cria gráfico de barras
    BarChartSeries barra = new BarChartSeries();
    barra.setLabel("Pesquisas");
    for (Pesquisa p : pesquisas) {
        barra.set(p.getId(), p.getNumeroEleitores());
    }
    combinedModel.addSeries(barra);
    List<Candidato> candidatos = candidatoDAO.getAll();
    // cria linhas para cada candidato da pesquisa
    for (Candidato candidato : candidatos) {
        LineChartSeries linha = new LineChartSeries();
        linha.setLabel(candidato.getNomeCandidato());
        for (int i = 0; i < pesquisas.size(); i++) {
            Pesquisa pesquisa = pesquisas.get(i);
            CandidatoPesquisa cp = candidatoPesquisaDAO.getByPesquisaAndCandidato(pesquisa, candidato);
            // calcula a porcentagem de cada candidato para a pesquisa.
            linha.set(pesquisa.getId(), (cp.getIntencaoVoto() * 100 / pesquisa.getNumeroEleitores()));
        }
        // adiciona a linha referente a cada candidato.
        combinedModel.addSeries(linha);
    }
    combinedModel.setTitle("Número de Eleitores e Candidatos");
    combinedModel.setLegendPosition("ne");
    return combinedModel;
}
```

Para exibir o gráfico na tela, basta adicionar a tag `<p:chart>` chamando o método criado. O trecho a seguir mostra o XHTML que adiciona o gráfico na tela:

```
<p:chart type="bar" model="#{PesquisaMB.graficoBarraLinha}"  
style="height:300px; width:700px;" />
```

Já a **Figura 6** mostra os dois gráficos combinados. Note que a legenda é unificada. Em azul, temos os dados das pesquisas apresentados através de barras, e em laranja e bege, os dados dos candidatos apresentados através de linhas.

Além dos gráficos apresentados neste artigo, existem outros, como o **MeterGauge**, que exibe informações utilizando um modelo que se assemelha ao velocímetro de um carro para mostrar, por exemplo, a velocidade da conexão com a Internet, e o **Donut** (Rosquinha), que é um tipo de gráfico com as mesmas características do gráfico de pizza, mas que pode exibir mais de um conjunto de dados ao mesmo tempo, como o resultado de várias pesquisas, possibilitando a comparação entre esses conjuntos.

Por fim, saiba que apesar de termos recuperado as informações de um banco de dados PostgreSQL, outras fontes podem ser utilizadas, como soluções NoSQL e serviços disponíveis na Internet, o que possibilita a geração de gráficos alimentados por dados obtidos em tempo real.

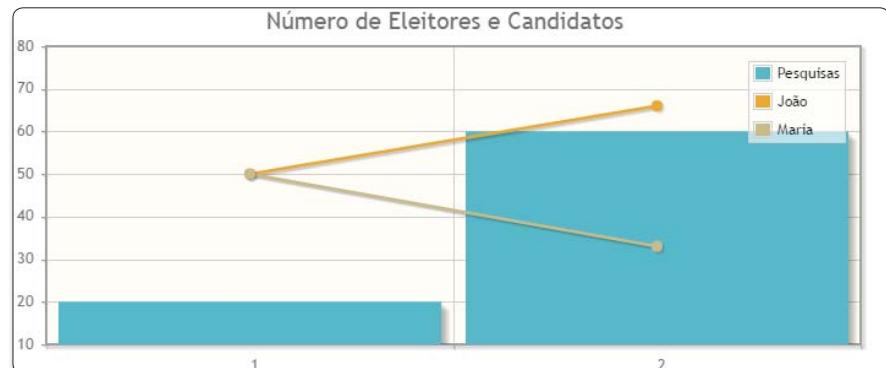


Figura 6. Imagem que combina um gráfico de barras e um de linhas

Autor



Luiz Henrique Zambom Santana

lhzsantana@gmail.com

É bacharel e mestre em Ciência da Computação. Atualmente cursa doutorado também em Ciência da Computação na UFSC. Possui mais de 10 anos de experiência em programação Java e há dois anos atua com tecnologias de Big Data. Trabalhou em projetos para grandes empresas no Brasil, Argentina e Alemanha. Atualmente é consultor de Elasticsearch.



Links:

Implementação de referência do JSF.

<https://glassfish.java.net/downloads/ri/>

Site oficial do PrimeFaces.

<http://primefaces.org/>

Fórum oficial do PrimeFaces.

<http://forum.primefaces.org/>

Show case com exemplos de todos os componentes do PrimeFaces.

<http://www.primefaces.org/showcase/>

Guia de usuário do PrimeFaces.

http://www.primefaces.org/docs/guide/primefaces_user_guide_5_1.pdf

Autor



Eduardo Felipe Zambom Santana

ezambomsantana@gmail.com



É bacharel e mestre em Ciência da Computação pela UFSCar. Possui mais de 10 anos de experiência em programação. Atualmente é aluno de doutorado na USP e professor na Universidade Anhembi Morumbi.

Criando aplicações com Spring e AngularJS para a nuvem

Aprenda neste artigo como desenvolver e colocar na nuvem uma aplicação utilizando Spring Boot e AngularJS

No desenvolvimento de aplicações, seja para web, desktop ou dispositivos móveis, além de todos os requisitos funcionais, a equipe de desenvolvimento precisa se preocupar também com requisitos não funcionais, como segurança, desempenho e escalabilidade, itens esses que constituem o alicerce da aplicação e que, na maioria dos casos, precisam ser levados em consideração desde o início, sob a pena de ser necessário reconstruir toda a aplicação.

No entanto, a modelagem de uma arquitetura robusta, escalável, que permita à equipe alta produtividade, desenvolvimento ágil e que faça uso de recursos de infraestrutura da nuvem em seu favor demanda tempo e tem impacto no orçamento do projeto. Esses itens (tempo e orçamento) nem sempre estão disponíveis, principalmente para quem está desenvolvendo uma aplicação como hobby e para pequenas startups. Por outro lado, o tempo e o dinheiro investidos nessa etapa reduzem o prazo e simplificam o desenvolvimento e a manutenção. Além disso, criar uma estrutura a partir do zero ou com pouquíssimo material prévio para cada novo projeto é moroso e improdutivo.

Uma das primeiras e mais difíceis tarefas quando se está iniciando um novo projeto é a escolha das tecnologias que serão utilizadas. Isso porque, com raras exceções, essas decisões acompanharão o projeto por toda a sua vida e mudar a tecnologia utilizada normalmente sai caro.

E quando o assunto são as tecnologias que servirão de base para o projeto, isso inclui linguagens, plataformas utilizadas, servidor de aplicações, o modo de armazena-

Fique por dentro

Este artigo apresenta uma proposta de arquitetura para aplicações web com foco principalmente em produtividade e escalabilidade, sem deixar de lado a segurança. Para isso, a arquitetura está baseada em dois frameworks de grande relevância no mercado, documentação vasta e suporte das comunidades: AngularJS e Spring. O primeiro é inteiramente desenvolvido em JavaScript e executa 100% no lado cliente, reduzindo a capacidade de processamento necessária do lado servidor. O segundo é um framework Java para aplicações web extremamente dinâmico e fácil de usar. Essas características permitem entregar uma solução altamente escalável e que pode ser utilizada em praticamente qualquer solução web.

mento dos dados, entre outros itens funcionais e não funcionais que merecem atenção.

Felizmente, é possível que uma mesma implementação de arquitetura sirva como referência para outras aplicações ou mesmo que seja reutilizada integralmente pela maioria das aplicações web. E é isso que este artigo irá mostrar: uma proposta de arquitetura que pode ser utilizada no modelo mais comum de desenvolvimento web, que se apoia em frameworks e ferramentas consolidadas e que está pronta para ser implantada na nuvem. A intenção aqui não é apresentar a “arquitetura definitiva para aplicações web”, mas sim um conjunto de soluções de problemas comuns a esse tipo de desenvolvimento, além de fornecer conhecimento para que os desenvolvedores possam adaptar as ideias às suas necessidades e aos requisitos dos seus projetos.

Frameworks

Frameworks são conjuntos de abstrações de códigos e estruturas genéricas que devem servir como suporte para uma complementação que possa criar funcionalidades específicas. Os frameworks são responsáveis por comandar o fluxo de execução das aplicações ou atividades. Não os confunda com bibliotecas, que são códigos completos que não ditam o fluxo da aplicação e não precisam de complementação para funcionar. Neste artigo serão utilizados dois frameworks de grande penetração no mercado, vasta documentação e amplo suporte da comunidade: AngularJS e Spring.

O desenvolvimento para web por muito tempo esbarrou em problemas de compatibilidade entre navegadores, quando utilizar JavaScript muitas vezes trazia mais problemas que soluções. No entanto, isso mudou bastante com o surgimento de bibliotecas como o jQuery e frameworks como o Dojo. Foi nessa onda que surgiu o AngularJS, framework JavaScript mantido principalmente pelo Google e usado para o desenvolvimento de aplicações web ricas, seguindo a arquitetura MVW (*Model-View-Whatever*). Os padrões arquiteturais normalmente utilizados nas aplicações front-end são MVC (*Model-View-Controller*), MVP (*Model-View-Presenter*) e MVVM (*Model-View-ViewModel*). Como o AngularJS funciona bem com todos eles, a equipe cunhou o termo MVW, que significa algo como Model-View-e o que funcionar no seu projeto.

Em resumo, o AngularJS estende as funcionalidades do HTML para simplificar e agilizar o desenvolvimento no lado cliente, permitindo a rápida criação de aplicações com excelente usabilidade, além de abstrair e facilitar o uso de recursos importantes, como chamadas a APIs REST e controle do fluxo de páginas.

Do lado servidor, a grande preocupação da maioria dos frameworks sempre foi fornecer ao desenvolvedor toda a infraestrutura possível para que ele pudesse manter o foco na codificação de funcionalidades importantes para o negócio e simplesmente utilizar recursos como segurança e controle transacional. Esse foi, desde o início, o grande apelo dos EJBs, mas a infraestrutura de servidor de aplicações e o ambiente pesado os mantiveram no mundo das grandes aplicações corporativas. O Spring Framework traz todos esses recursos em um ambiente muito mais leve, simples e acessível, permitindo que possa ser utilizado em praticamente qualquer aplicação.

Arquitetura

A arquitetura de um sistema computacional ou de uma aplicação diz respeito aos elementos que servem de base para a sua construção e como esses interagem entre si para satisfazer as necessidades de funcionamento, desempenho, segurança e usabilidade. É o tópico que trata das decisões de design que guiarão o desenvolvimento e o andamento do projeto.

Entre os requisitos não funcionais que devem ser observados pela arquitetura estão a reusabilidade e a escalabilidade. A primeira existe somente em tempo de desenvolvimento e trata da capacidade dos componentes de serem reutilizados (sejam métodos, classes ou serviços). Esse item impacta diretamente o desenvolvimento e a manutenção uma vez que, quanto maior

a reusabilidade, menor a quantidade de código a ser escrito e mantido e mais simples será a aplicação. Já a escalabilidade só faz sentido em tempo de execução e refere-se à capacidade do sistema ou aplicação de suportar o crescimento de trabalho de maneira uniforme, preferencialmente com o menor esforço possível.

O que será mostrado aqui é uma forma de utilizar esses conceitos na prática, com técnicas e padrões de projeto dos quais a equipe poderá se beneficiar, de acordo com os requisitos e necessidades do projeto.

Proposta

A proposta de arquitetura apresentada neste artigo está dividida em duas partes, bastante conhecidas pelos desenvolvedores: front-end e back-end. O front-end é a camada que interage com o usuário e, assim sendo, é nela onde são aplicados os conceitos de usabilidade e estética para que a aplicação seja bonita e simples. O back-end, por sua vez, é a camada responsável por executar as funcionalidades associadas ao negócio da aplicação e, dessa forma, é nela que são implementadas as regras de validação e execução, a persistência dos dados e as integrações com outros sistemas.

A ideia é que as duas camadas sejam desacopladas fisicamente, como se fossem duas aplicações implantadas em estruturas diferentes e utilizando recursos diferentes. A aplicação AngularJS é o cliente, e comporta as páginas HTML, os arquivos JavaScript, as imagens e outros recursos, que serão providos por um servidor de arquivos comum, como o Apache, NGINX ou outro que esteja disponível. Esses normalmente são mais rápidos e estáveis que os servidores de aplicações para a execução desse tipo de tarefa, onde os arquivos não sofrem modificações no seu conteúdo.

Já a aplicação back-end será implementada em Spring, tendo como base o projeto Spring Boot, que simplifica bastante o desenvolvimento, encapsula o uso de um servidor de aplicações e fornece uma API REST para que a aplicação cliente possa executar suas tarefas. Ao contrário das aplicações que utilizam páginas dinâmicas, como JSP e JSF, os serviços REST não precisam fazer a geração de páginas, o que os tornam significativamente mais leves, exigindo menos recursos do servidor e possibilitando um desenvolvimento mais simples e produtivo.

Os serviços back-end serão fornecidos somente como stateless (veja o **BOX 1**), o que significa que não precisarão ser criados e destruídos para cada cliente ou sessão, reduzindo o processamento e a quantidade de memória necessária no back-end. Além disso, por não terem necessidade de afinidade com o usuário da requisição, isto é, de uma instância do serviço não ser dedicada a um único cliente, a tarefa de balanceamento da carga de trabalho se torna muito mais simples.

Para garantir a segurança da aplicação, a autenticação será feita por meio de um token de segurança. Basicamente, na primeira requisição, a aplicação cliente solicita um token, informando um nome de usuário e uma senha. Se os dados estiverem corretos, o back-end fornece um token, que estará associado ao usuário e deverá ser informado em cada requisição. Enquanto o token for válido, a aplicação cliente terá autorização para executar.

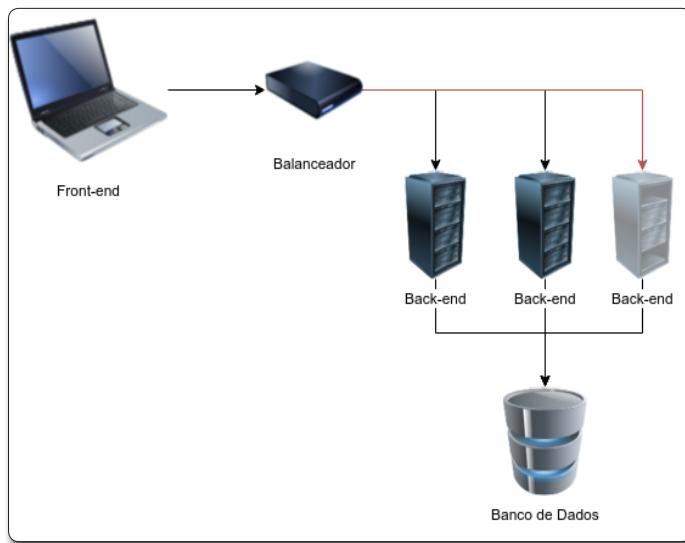


Figura 1. Implantação da aplicação

A Figura 1 mostra como poderá ser implantada a aplicação, incluindo, além do front-end e do back-end, elementos de infraestrutura, como平衡adores de carga e banco de dados.

Um balanceador de carga de trabalho (*workloader*) é o componente responsável por distribuir, de acordo com critérios estabelecidos, a execução do trabalho requisitado entre os recursos disponíveis. Assim, um balanceador pode, por exemplo, entregar uma requisição para cada servidor existente, em um sistema de rodízio.

BOX 1. Serviço stateless

Um serviço stateless é aquele que, como o nome sugere, não tem estado, isto é, tudo que é necessário para sua execução está nos dados da requisição e não é preciso que um outro serviço seja executado antes ou depois dele para que a sua tarefa se complete.

Back-end

Normalmente, ao desenvolver uma aplicação web, assim como para a implantação de aplicações em ambientes de produção, é preciso configurar um servidor de aplicações (Tomcat ou Jetty, por exemplo) e implantar o pacote WAR nesse servidor. Na proposta de arquitetura deste artigo, será mostrado o uso do Spring Boot, solução que facilita a criação de aplicações web standalone de forma realmente muito simples, sem a necessidade de utilização explícita de um servidor de aplicações (o framework cuida disso), sem arquivos de configuração, sem geração de código e sem a necessidade de deploy.

A criação de um novo projeto é sempre uma tarefa chata e repetitiva, quando o desenvolvedor precisa criar estruturas de pastas e pacotes, arquivos de build e outros recursos, normalmente muito parecidos. Uma excelente ferramenta para começar um novo projeto é o Spring Initializr (veja a seção **Links**), uma aplicação web do projeto Spring que realiza esse trabalho repetitivo rapidamente e com um número mínimo de parâmetros. Ao acessar o Initializr é possível selecionar as preferências do projeto, como o gerenciador

de dependências, a versão do framework Spring, os demais módulos que se deseja incluir, como segurança e persistência, e, por fim, ao clicar no botão *Generate Project*, gerar um esqueleto do projeto com um *pom.xml* (vide **Listagem 1**) e uma classe de inicialização da aplicação (vide **Listagem 2** com um método *main()*).

Listagem 1. Código do pom.xml gerado.

```
<groupId>com.example</groupId>
<artifactId>demo</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>demo</name>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.5.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Listagem 2. Código da classe de entrada.

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Além do que é criado por padrão, serão necessárias mais algumas dependências para as funcionalidades desejadas, como a capacidade de conversar no protocolo REST, utilização do formato JSON e acesso a bancos de dados. Isso pode ser feito de duas formas: selecionando as dependências desejadas no próprio Initializr, ou adicionando essas dependências manualmente no *pom.xml*, como na **Listagem 3**. A vantagem da primeira opção é que não é necessário conhecer o *groupId* ou o *artifactId* do Maven específicos para o módulo desejado, sendo possível fazê-lo somente com o seu nome. A escolha de qual opção utilizar fica a cargo do desenvolvedor.

Uma das preocupações da arquitetura aqui proposta, como visto anteriormente, é a simplicidade e reusabilidade do código, com o objetivo de aumentar a produtividade e melhorar a manutenibilidade. Sendo assim, a aplicação back-end será dividida nas camadas mostradas na **Figura 2**. A separação de responsabilidades é um conceito de grande importância e, por isso, deve servir de guia para manter a arquitetura da aplicação nos trilhos.

Uma excelente ideia para o projeto é o padrão chamado *Layer SuperType*, que orienta a criação de um supertipo para cada camada da aplicação. Segundo Martin Fowler, não é incomum que todas as classes de uma mesma camada tenham métodos e atributos que se repetem com pouca ou nenhuma variação. Logo, o melhor a fazer é colocar todo esse comportamento comum no supertipo da camada, reduzindo o código escrito e replicado.

Listagem 3. Dependências adicionais a serem incluídas no pom.xml.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-entitymanager</artifactId>
</dependency>
<dependency>
<groupId>org.codehaus.jackson</groupId>
<artifactId>jackson-mapper-asl</artifactId>
<version>1.9.10</version>
</dependency>
<dependency>
<groupId>joda-time</groupId>
<artifactId>joda-time</artifactId>
</dependency>
```

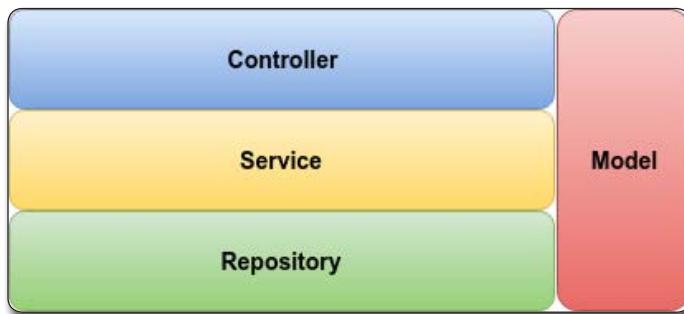


Figura 2. Camadas do back-end

Modelo

Na camada de modelo estão as entidades mapeadas do banco de dados, que servirão de parâmetro na definição dos supertipos das demais camadas da aplicação. Além dos mapeamentos, é uma boa ideia utilizar os recursos do Hibernate Validator para fazer as validações básicas da entidade — obrigatoriedade, valores mínimos e máximos, tamanho das strings, etc. — colocando anotações na própria classe, pois o seu uso é prático e essas restrições estão intimamente relacionadas à integridade da entidade. A **Listagem 4** mostra o supertipo dessa camada.

Listagem 4. Supertipo da camada de modelo.

```
package com.exemplo.demo.modelo;

@MappedSuperclass
public abstract class BaseModelo implements Serializable {
    private static final long serialVersionUID = 1L;
}
```

Inicialmente, essa classe não tem nenhum código que será herdado pelas outras classes, mas atributos comuns e seus mapeamentos podem ser adicionados. Algumas aplicações podem ter um ID incremental, a data e hora de inserção e da última atualização em

todas as entidades, por exemplo. Por ser uma classe abstrata, ela precisa ser anotada com `@MappedSuperclass` para indicar que não deve ser interpretada como uma entidade concreta e, desse modo, que deverá ser estendida por todas as demais entidades, garantindo o mesmo tipo e comportamento, quando necessário, a todas elas.

Repositórios

Agora que a superclasse que servirá como parâmetro às demais camadas está pronta, pode-se definir a próxima camada na pilha da aplicação: a camada de acesso aos dados, que como o nome indica, é responsável por recuperar, incluir e alterar os dados.

Certamente, muito do código escrito nessa camada para executar as tarefas mais comuns é repetitivo e pode ser parametrizado. No entanto, o Spring é capaz de fazer melhor e reduzir drasticamente a quantidade de código escrito, sendo necessário somente definir a interface do repositório. A partir disso, fica a cargo framework, em tempo de execução, gerar a implementação.

Listagem 5. Supertipo da camada de acesso aos dados.

```
package com.example.demo.repositorios;

@NoArgsConstructor
public interface BaseRepository<T extends BaseModelo> extends
PagingAndSortingRepository<T, Long> { }
```

Para que essa “mágica” aconteça, é claro que algumas regras devem ser seguidas para informar ao Spring qual código deve ser gerado, começando pela hierarquia da interface. Embora não seja obrigatório, é uma boa ideia estender interfaces do framework para herdar suas definições, reduzindo o código e mantendo um padrão de contrato, mesmo em projetos diferentes. Por isso, o supertipo dessa camada, mostrado na **Listagem 5**, estende `PagingAndSortingRepository`, que define métodos de listagem com suporte à paginação e ordenação, e também os métodos para salvar, obter e excluir, que podem ser vistos em mais detalhes na **Listagem 6**. Além disso, o supertipo também é parametrizado com uma entidade T que estende de `BaseModelo`, permitindo que somente as classes que pertencem a essa hierarquia possam ter seus próprios repositórios.

Nota

Como a interface `BaseRepository` é o supertipo da camada de repositórios e não irá gerar uma classe concreta, ela deve ser anotada com `@NoArgsConstructor` para evitar um erro na inicialização da aplicação. Isso ocorre porque o parâmetro T só será definido pelas interfaces que estendem `BaseRepository` e o Spring não saberia como gerar a classe sem ele.

Serviços

A próxima camada apresentada é a de serviços. Responsável pela lógica de negócios da aplicação, é nela que estarão as regras de validação mais complexas, como quais as condições pré-existentes para que os dados sejam persistidos e a interação entre entidades, além

de orquestrar diferentes serviços para a execução de uma tarefa. Em uma aplicação de venda de passagens, por exemplo, o serviço responsável pela reserva deverá orquestrar os serviços necessários para verificação da disponibilidade de voo, reserva de acentos e pagamentos. Por ser a detentora das regras de negócio e coordenar a interação entre outros serviços, essa classe também é a responsável por interagir com o repositório para acessar ou persistir os dados.

Listagem 6. Métodos da interface PageAndSortingRepository.

```
Iterable<T> findAll(Sort sort);
Page<T> findAll(Pageable pageable);
<S extends T> S save(S entity);
<S extends T> Iterable<S> save(Iterable<S> entities);
T findOne(ID id);
boolean exists(ID id);
Iterable<T> findAll();
Iterable<T> findAll(Iterable<ID> ids);
long count();
void delete(ID id);
void delete(T entity);
void delete(Iterable<? extends T> entities);
void deleteAll();
```

Na superclasse da camada de serviços (veja a **Listagem 7**) estarão os métodos comuns, públicos e protegidos a todos os serviços da aplicação. Esses devem ser implementados de forma genérica, deixando para as classes concretas fornecer o código específico de cada caso. Por exemplo, o método *save* pode invocar um método **protected** que faz a validação dos dados antes de persisti-los. Nesse caso, a implementação padrão pode não fazer validação alguma, deixando essa tarefa a critério das subclasses. As classes concretas, por sua vez, além de complementar o código do supertipo, precisam ser anotadas com **@Service** e **@Transactional**. A primeira anotação serve para identificá-la como um service para o framework e a segunda, para indicar que existe controle transacional nesse componente. Isso é necessário porque essas classes realizarão tarefas de alteração no banco de dados.

Nos métodos que realizam consulta ao banco de dados sem realizar qualquer alteração é uma boa ideia utilizar a anotação **@Transactional(propagation=Propagation.NOT_SUPPORTED)**. Isso indica ao framework que não precisa haver controle transacional para essa operação, economizando recursos tanto no servidor de aplicações quanto no banco de dados.

Controladores

A camada de controle deve ser responsável por receber as requisições da aplicação cliente, realizar transformações e repassar as requisições à camada de serviços. Assim como ocorre com outros componentes, criar controllers no Spring é fácil e pode ser feito anotando a classe com **@RestController**, o que sinaliza que esse controle conversa com as aplicações clientes por meio do protocolo REST. Nesse caso, não é necessária nenhuma anotação ou configuração adicional para informar ao Spring que tanto o *response* quanto o *request* devem utilizar o formato JSON. Assim como nas outras camadas, deve haver uma superclasse para essa também (vide **Listagem 8**).

Listagem 7. Esqueleto da superclasse de serviços.

```
package com.example.demo.services;
public abstract class BaseServicos<T extends BaseModelo> {
    public void delete(Long id) { ... }

    @Transactional(propagation=Propagation.NOT_SUPPORTED)
    public T get(Long id) { ... }

    @Transactional(propagation=Propagation.NOT_SUPPORTED)
    public Page<T> list(Integer page) { ... }

    public T save(T object) { ... }
}
```

Listagem 8. Supertipo da camada de controle.

```
package com.example.demo.controladores;

public abstract class BaseControlador<T extends BaseModelo> {

    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
    public void delete(@PathVariable Long id, HttpServletRequest request,
    HttpServletResponse response) { ... }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public T get(@PathVariable("id") Long id, HttpServletRequest request,
    HttpServletResponse response) { ... }

    @RequestMapping(value = "", method = RequestMethod.GET)
    public Page<T> list(@RequestParam(value = "p",
    required = false, defaultValue = "0") Integer page, HttpServletRequest request,
    HttpServletResponse response) { ... }

    @RequestMapping(value = "", method = RequestMethod.POST)
    public T save(@Validated @RequestBody T object,
    HttpServletRequest request, HttpServletResponse response) { ... }
}
```

Como nos demais casos de superclasses de camada, essa é uma classe abstrata e por isso não possui a anotação de implementação (**@RestController** neste caso) que deve ser colocada nas classes concretas. A anotação **@RequestMapping**, por sua vez, define a URI de acesso ao controller e deve ser declarada tanto na classe concreta, para especificar o recurso manipulado por ela, quanto nos métodos, para especificar em quais condições cada método será acionado.

Seguindo as convenções de nomenclatura de recursos para APIs REST, a URI de cada controller deve ser o nome do recurso por ele manipulado, no plural. Logo, tomando o exemplo de uma aplicação de venda de passagens, a URI do controller que manipula a entidade **Reserva** seria */reservas*. Ainda de acordo com a convenção REST, os verbos HTTP devem ser utilizados para definir as operações realizadas em um determinado recurso, sendo o método GET para recuperar e os métodos POST e/ou PUT para gravar. Assim, se a URI */reservas* for usada com o método GET, o controller deve retornar a lista de reservas, mas se o método utilizado for POST, o controller irá salvar o objeto JSON recebido. Da mesma forma, se a URI */reservas/1234* for utilizada com o método GET, o controller retornará essa reserva no formato JSON, se for utilizado o POST, os dados da reserva serão atualizados, e se for DELETE, a reserva será excluída.

Configurações

Agora que todas as camadas do back-end estão criadas, é necessário configurar a aplicação para que os componentes sejam carregados pelo Spring. Nas versões anteriores do framework, essa tarefa era feita em arquivos XML nos lugares mais diversos e com uma sintaxe complicada. Por outro lado, as versões mais atuais dão suporte à configuração por meio de anotações, como acontece nos componentes. Essa abordagem facilita o desenvolvimento e a manutenção da aplicação, já que tudo está no mesmo lugar. Porém, alterar essa configuração em tempo de execução não é possível, pois o código precisa ser recompilado e reimplantado.

Para informar ao Spring onde estão os componentes, deve ser utilizada a anotação **@ComponentScan** na classe de entrada da aplicação. Essa anotação tem o parâmetro **basePackages**, que deve ser preenchido com a lista de pacotes, separados por vírgula, que serão varridos à procura dos componentes devidamente anotados com **@Service**, **@RestController** e **@Component**. Nesse caso, o trecho que será adicionado à classe deve ser o seguinte:

```
@ComponentScan(basePackages = "com.example.demo.controladores,com.example.demo.servicos")
```

Nota

A anotação **@Component** deve ser utilizada para componentes que não são controladores e nem serviços, em classes que fornecem métodos para execução de tarefas genéricas, que não têm relação direta com o negócio da aplicação, como ler os dados de um arquivo ou converter valores. A vantagem de anotar a classe, ao invés de criar uma classe com métodos estáticos, é que, dessa forma, é possível usufruir dos recursos do framework, como injeção de dependência.

Embora permita o uso de anotações, algumas das configurações do comportamento do framework só podem ser feitas por meio de properties, e para isso o Spring conta com um engenhoso recurso. Se houver um arquivo chamado *application.properties* no classpath da aplicação, esse arquivo será lido e suas propriedades serão carregadas. Acontece que é bastante comum que se utilize um valor no ambiente de desenvolvimento e outro no ambiente de produção, e isso pode causar problemas. O desenvolvedor acaba esquecendo de alterar as configurações e o arquivo de desenvolvimento passa a ser utilizado em produção ou o inverso. Por isso, o mecanismo de configurações tem o conceito de perfil, parecido com o do Maven.

Assim como no arquivo *pom.xml*, no arquivo de properties é possível definir qual o perfil padrão utilizado em tempo de execução. Isso é feito por meio da propriedade **spring.profiles.active**. Caso ela tenha um valor, o framework irá carregar as propriedades do arquivo *application-<perfil>.properties*, se ele existir, e sobrepor os seus valores no arquivo original. Assim, se a propriedade **spring.profiles.active** tiver o valor **dev**, por exemplo, e existir um arquivo chamado *application-dev.properties*, esse será carregado e seus valores irão sobrepor os valores do arquivo *application.properties*.

O Spring possui um grande número de propriedades para configurar os mais diversos comportamentos da aplicação, como a porta usada para acesso HTTP. Uma propriedade que merece

ser mencionada aqui é a **security.sessions=NEVER**. Com ela, nenhuma sessão será criada no lado servidor, o que economiza recursos e simplifica a escalabilidade da aplicação.

Deve-se evitar o armazenamento de dados relativos ao cliente na sessão do servidor de aplicações. Essa prática, além de onerar o servidor, complica o escalonamento da aplicação, uma vez que exige o compartilhamento de sessões. Uma boa alternativa é utilizar um banco de dados NoSQL, do tipo document base ou key-value, de preferência em memória.

Como uma das premissas da arquitetura proposta é que a aplicação front-end e a back-end fiquem fisicamente em servidores diferentes, é possível que, dependendo do local de hospedagem, o acesso a esses servidores, ou conjunto de servidores, se dê por meio de domínios diferentes. Isso leva a um problema comum, conhecido por **CORS** (*Cross-Origin Resource Sharing*). Para habilitar esse recurso na aplicação, deve-se definir, no arquivo *application.properties*, as propriedades com prefixo **endpoints.cors** de acordo com as necessidades.

O mecanismo de CORS permite restringir o acesso a APIs entre sites de domínios diferentes. Isso se dá porque os navegadores modernos impedem que uma chamada via AJAX seja feita para um servidor em um domínio que não seja o do site acessado, a menos que o servidor explice a autorização por meio de um cabeçalho HTTP.

Segurança

Com a aplicação do lado servidor estruturada, é preciso pensar na segurança. Nesse caso, a aplicação utilizará um mecanismo de autenticação OAuth2 baseado em token para as chamadas à API REST.

Nesse modelo, o fluxo de autenticação funciona da seguinte forma: a aplicação cliente envia as credenciais do usuário, normalmente login e senha, para o servidor. O servidor valida essas credenciais e, em caso positivo, retorna à aplicação cliente um token. A partir daí, a cada requisição enviada pelo cliente, o token deve ser enviado junto, no cabeçalho HTTP, para ser validado no servidor.

Para que isso tudo funcione corretamente, deve-se configurar o módulo de segurança no Spring. Isso pode ser feito adicionando a dependência do módulo às configurações do Maven ao criar o esqueleto do projeto no Initializr, selecionando **Security** na lista de dependências, ou adicionando manualmente o trecho de código da [Listagem 9](#) ao arquivo *pom.xml*.

Agora é hora de utilizar os recursos de configuração programática para definir como a segurança irá funcionar. Em uma nova classe, que aqui será chamada de **OAuth2ServerConfiguration**, especificaremos as configurações necessárias. A classe deve ficar no pacote **com.example.demo.security** e ser anotada com **@Configuration**. No entanto, para que ela seja corretamente lida pelo Spring, é preciso adicionar esse pacote à lista de pacotes na anotação **@ComponentScan** da classe de entrada.

Uma boa maneira de organizar as configurações é adicionar duas classes estáticas à **OAuth2ServerConfiguration**: uma para a configuração da autorização, onde são definidos os recursos protegidos da aplicação; e outra para configurar o gerenciamento dos tokens de autenticação.

Na **Listagem 10** está um exemplo de como a configuração pode ser feita. Na documentação do framework pode-se verificar os detalhes e configurações que satisfazem às necessidades de cada projeto. É possível, por exemplo, definir um serviço responsável por recuperar as informações do usuário a partir das credenciais, caso o projeto precise de um modelo mais elaborado para representar o usuário autenticado.

Listagem 9. Dependência do Spring Security.

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-core</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.security.oauth</groupId>
<artifactId>spring-security-oauth2</artifactId>
</dependency>
```

Listagem 10. Configuração de segurança – código da classe OAuth2Server Configuration

```
package com.example.demo.security;
@Configuration
public class OAuth2ServerConfiguration {
    @Configuration
    @EnableResourceServer
    protected static class ResourceServerConfiguration extends
        ResourceServerConfigurerAdapter { ... }
    @Configuration
    @EnableAuthorizationServer
    protected static class AuthorizationServerConfiguration extends
        AuthorizationServerConfigurerAdapter {
        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpoints)
            throws Exception { ... }
    }
}
```

Quando um usuário é autenticado com sucesso e um token é gerado, esse token deve ser armazenado em algum lugar para poder ser validado depois. É no método **configure()** da classe **AuthorizationServerConfigurerAdapter** que esse lugar é informado. Para isso, **configure()** recebe como argumento um objeto do tipo **AuthorizationServerEndpointsConfigurer**, que pode ser utilizado para alterar diversas configurações de autenticação, entre elas o *tokenStore*, que é o mecanismo de armazenamento de tokens.

O framework oferece, por padrão, dois tipos de armazenamento de tokens: em memória ou em um banco de dados por meio de acesso JDBC. Obviamente o primeiro é mais rápido e deve ser utilizado em ambiente de desenvolvimento. O segundo, por sua vez, pode ser adotado em ambiente de produção, pois permite que vários servidores compartilhem os mesmos tokens de forma bastante simples. Uma boa opção para esse armazenamento é utilizar um banco de dados em memória, como o **H2**.

Front-End

Há vários anos, no início do desenvolvimento das aplicações web, o JavaScript era usado somente para realizar algumas validações no lado cliente ou executar alguma animação. Na maioria das vezes, coisas puramente estéticas, sem impacto real no funcio-

namento da aplicação. Isso porque, além de ter um desempenho ruim, as diferenças entre navegadores tornavam as coisas bastante complexas para fazer algo importante, uma vez que era de responsabilidade do desenvolvedor resolver essas diferenças.

Com o aumento do poder de processamento nos computadores pessoais e a guerra dos navegadores, os fabricantes passaram a investir mais nos mecanismos de execução de JavaScript, melhorando o seu desempenho e seguindo um caminho para a redução das diferenças entre eles. A linguagem passou então a ganhar mais importância e relevância e começaram a surgir bibliotecas e frameworks que aproveitam melhor suas capacidades.

Entre eles, surgiu o AngularJS, que é um framework que segue um novo paradigma de aplicações web, chamado SPA (*Single Page Application*), oferecendo suporte a interfaces de usuário ricas, sem recarregar a página inteira e preocupadas com a beleza e a usabilidade. As funcionalidades e a modularização do framework o tornam uma excelente opção para esse tipo de aplicação, entregando ao desenvolvedor facilidade de uso e alta produtividade.

Embora não costume ter uma interface web como o Spring Initializr, o AngularJS possui várias ferramentas para criar o esqueleto de um novo projeto, como o Angular Seed, que não é mantido pela equipe do AngularJS, e o Yeoman, que também pode ser usado para outros tipos de projetos, como Node.js. Essas opções são simples de usar e não deve haver dificuldades para o desenvolvedor. Por isso, nenhuma delas será abordada neste artigo.

Como explicado anteriormente, o AngularJS é um framework MVW, o que significa que ele funciona bem com qualquer padrão de projeto MVC, MVP ou MVVM. Assim sendo, aqui será adotado o padrão MVC. Ao iniciar uma aplicação construída sobre o AngularJS, a página HTML inicial é carregada, assim como o seu script. Esse script contém, entre outras coisas, a definição de states (ou rotas, dependendo de qual recurso for utilizado), que sinaliza qual view deve ser carregada de acordo com a situação (*state*) da aplicação. Desse modo, pode haver, por exemplo, um *state* chamado *home* e outro chamado *about*, de forma que a navegação entre as páginas se dá por meio das mudanças de *state*.

Do mesmo modo que ocorre no back-end, a lógica de execução deve ser implementada nos controllers, uma vez que esses componentes são os responsáveis por alterar estados e controlar o uso da interface do usuário. Assim como há, normalmente, uma classe *RestController* para cada recurso no back-end, um bom nível de granularidade para essa camada, pelo menos inicialmente, é que haja um controller para cada view. À medida que os componentes são reutilizados, é possível perceber que alguns dos controladores podem ser divididos para melhorar a coesão e aumentar o reuso.

Ainda que seja comum executar chamadas às APIs REST nos controladores, vê-se como uma boa prática encapsular essas chamadas em services. Para facilitar isso, o framework oferece uma boa abstração para a execução de chamadas remotas via AJAX, semelhante ao jQuery, o que facilita bastante a vida do desenvolvedor. Ao colocar as chamadas às APIs em serviços, obviamente um para cada resource, torna-se possível que um determinado controller reutilize diferentes serviços na mesma tela, por exemplo.

Além disso, o framework ainda coloca à disposição uma série de funcionalidades que ajudam significativamente no desenvolvimento, livrando o programador da preocupação com as minúcias de baixo nível da linguagem JavaScript.

Autenticação

Da mesma forma que foi necessário incluir um mecanismo para garantir a segurança no back-end, é importante que algo parecido seja feito no front-end também. Contudo, desta vez, isso não implica exatamente em uma questão de segurança, mas sim de usabilidade, dependendo do funcionamento da aplicação. Isso porque, como o back-end irá verificar a autenticação da aplicação cliente, as requisições serão negadas caso o front-end não verifique. Sendo assim, para evitar que erros de autenticação apareçam para o usuário que ainda não efetuou login ou que teve a sessão expirada, é importante que o mecanismo esteja sincronizado.

Para isso, será criado um serviço de autenticação na aplicação cliente. A ideia é que ele seja responsável por tudo relacionado à aplicação, como verificar se a autenticação foi realizada e se está válida, realizar a atualização do token se necessário, recuperar os dados do usuário logado e, é claro, realizar login e logout.

Uma vez que o serviço de autenticação esteja criado, deve ser adicionado um event listener ao mecanismo de manutenção do estado da aplicação, o **\$stateProvider**. Ao receber a notificação de **stateChangeStart**, ou seja, de que a aplicação está iniciando a mudança para um novo estado, o listener solicita ao serviço de autenticação que verifique se há um usuário autenticado. Em caso negativo, o usuário é redirecionado para a página de login.

Lembre-se que o serviço de autenticação deve utilizar algum recurso de persistência para armazenar o token de autenticação, certificando-se que esse seja removido quando o usuário efetuar logout. Isso pode ser feito por meio de cookies. A vantagem de usar cookies ao invés do armazenamento de dados do navegador é que é possível definir um tempo para que esse expire, fazendo com que tenhamos logout automático. As **Listagens 11** e **12** mostram como se dá esse fluxo.

Ao realizar o login, o controller responsável por essa tela deverá utilizar o serviço de autenticação para executar a tarefa. Esse último, por sua vez, autentica o usuário na aplicação back-end, o qual só então deverá ser direcionado para a página inicial da aplicação.

Quando o serviço de autenticação é invocado para executar, deverá verificar se já existe um token armazenado em um cookie válido e, caso positivo, somente realizar a validação do token no servidor, ao invés de uma nova autenticação. Isso é feito definindo o parâmetro **grant_type** da requisição para **refresh_token** e o parâmetro **refresh_token** com o token de atualização, sendo que esse é retornado pelo servidor no momento da autenticação com login e senha.

No caso de não haver nenhum token disponível, uma nova autenticação deve ser feita, com o nome do usuário e a senha. O serviço de autenticação deve então definir os parâmetros **grant_type**, **scope**, **username** e **password** com os respectivos valores: "password", "read write", nome do usuário e senha do usuário.

Em ambos os casos, além dos parâmetros passados, o serviço

deve informar ainda o **client_id** e o **client_secret**. Esses valores são especificados na aplicação back-end, na classe de configuração da segurança. Há ainda o cabeçalho **Authorization**, que deve ter a string 'Basic' concatenada com o resultado da função **btoa(client_id + ':' + client_secret)**. Essa função transforma a sequência com o id do cliente, o caractere ":" e o código secreto do cliente em um hash base64.

Em seguida, os dados de autenticação devem ser enviados, via **HTTP POST**, para a URI **/oauth/token**, e o seu resultado, como dito anteriormente, será armazenado em formato JSON em um cookie no navegador. Em casos mais críticos ou se o desenvolvedor é mais preocupado com a segurança, é possível encriptar os dados da autenticação antes de armazená-los no cookie. É claro que isso fará necessária a decriptação dos mesmos quando forem recuperados.

Uma vez autenticado, é importante lembrar que a cada nova requisição para o back-end, o cabeçalho **Authorization: Bearer + access_token** deve ser enviado para que a autenticação seja

Listagem 11. Fluxo de autenticação.

```
$rootScope.$on('$stateChangeStart',
  function (event, toState) {
    if (!auth.isAuthorized() && toState.isLogin != true) {
      $state.go('access.signin');
      event.preventDefault();
    }
});
```

Listagem 12. Serviço de autenticação.

```
function isAuthorized() {
  try {
    if (header == null) {
      setAuthorization(getAuthorization());
    }
    return header != null;
  } catch (error) {
  }
}
function setAuthorization(authorization) {
  header = "Bearer" + authorization.access_token;
  $cookies.put(CONFIG.AUTH, JSON.stringify(authorization), {expires: moment().add(30, 'minutes').toDate()});
}
function getAuthorization() {
  var authorization = null;
  try {
    authorization = JSON.parse($cookies.get(CONFIG.AUTH));
  } catch (Exception) {
  }
  return authorization;
}
```

Nota

Há um bug relacionado a esse fluxo aberto no GitHub do componente ui-router. Para contorná-lo, deve ser utilizado o seguinte código, logo após a configuração dos estados (assumindo que **app.home** é o estado padrão da aplicação):

```
$urlRouterProvider.otherwise(function ($injector, $location) {
  var $state = $injector.get("$state");
  $state.go("app.home");
});
```

validada, do contrário o servidor negará a requisição. Isso pode ser feito configurando o cabeçalho padrão no `$httpProvider` para evitar que eventuais esquecimentos se tornem uma dor de cabeça.

Utilize o recurso de definição de constantes do AngularJS para guardar configurações, como o `client_id`, o `client_secret`, endereços de servidores remotos ou outros valores que normalmente se repetem no código. Para isso, há o método `constant` no módulo da aplicação que define uma chave e um valor, que pode ser numérico, string, um array ou um objeto. Assim, quando necessário, basta injetar a constante definida pelo nome. Como exemplo, suponha que foi definida uma constante com o nome `CLIENTE`. Desse modo, basta que se use a injeção de dependências do framework com o nome `CLIENTE` e ela estará disponível.

Implantação

Há vários anos esse era um problema muito chato para quem desenvolve aplicações web baseadas em Java, pois não havia hospedagem barata e de qualidade. Os provedores de hospedagem de baixo custo, que forneciam estruturas muito pequenas, embora suficientes para muitos tipos de projetos, não davam suporte a ambientes com servidores de aplicações, porque isso requeria um suporte mais qualificado e mais caro. Esse cenário acabava por inviabilizar e desestimular o desenvolvimento de pequenos projetos e relegava a plataforma Java a ambientes corporativos, que possuíam orçamento para aluguel de servidores dedicados e poderiam configurá-los conforme as necessidades da aplicação.

Com o surgimento e, principalmente, com o crescimento e popularização da computação em nuvem, a demanda aumentou e o preço ficou cada vez mais acessível, tornando a realização de pequenos projetos muito menos impeditiva e com custos de uma pequena fração de outrora. Agora, é possível ter uma máquina de pequeno porte, com um processador, 512 megabytes de memória RAM e alguns gigabytes de disco por US\$ 5 por mês. Certamente um hardware capaz de executar, sem dificuldades, uma aplicação implementada sobre a arquitetura vista neste artigo.

Um dos maiores e mais conhecidos players de computação na nuvem do mercado é o Amazon Web Services. A empresa fornece uma enorme gama de serviços e infraestrutura no modelo *Pay As You Go*, ou seja, pague conforme o uso, além de sistemas

operacionais, tamanhos de hardware e orçamentos variados. Particularmente, os serviços oferecidos pela Amazon Web Services necessários para a implantação de uma aplicação seguindo a arquitetura proposta são: S3, EC2 e RDS.

O Amazon S3, ou *Amazon Simple Storage Service*, provê armazenamento com confiabilidade, segurança, escalabilidade e alto desempenho. É bastante simples de usar, embora tenha recursos mais avançados, como políticas de aposentadoria de dados, e também é muito barato, já que a cobrança é feita pelo volume de dados armazenado. O mais interessante nesse caso é que o S3 permite que os arquivos nele armazenados sejam servidos via HTTP diretamente, sem a necessidade de um outro servidor. Por esse motivo, todo front-end da aplicação deve ser implantado nele.

O EC2, que significa *Elastic Compute Cloud*, é o tipo de serviço mais comum: um servidor dedicado. Ele permite que o cliente escolha qual sistema operacional e qual versão irá executar e em minutos é possível ter disponível um novo servidor exclusivo, com uma capacidade que pode variar de um processador e meio gigabyte de memória a até 36 processadores com 244 gigabytes de memória.

O último, *Relational Database Service* (RDS), é o serviço que provê bancos de dados na nuvem, com um custo acessível, recursos confiáveis, seguro e redimensionável. A Amazon dispõe de seis implementações diferentes de bancos de dados: Amazon Aurora, Oracle, MS SQL Server, PostgreSQL, MySQL e MariaDB.

A melhor notícia é que toda essa estrutura está disponível em um modo de degustação por um período de tempo limitado para novos usuários. Ou seja, é possível implantar a aplicação para testes ou piloto gratuitamente e ela não será removida ao final do prazo, mas será preciso começar a pagar pelo uso.

Front-end

Para implantar o front-end no S3 é preciso criar um novo bucket (que é simplesmente uma forma de organizar os arquivos) informando um nome que o identifique unicamente — `demoapp`, por exemplo — e uma região (geográfica) na qual os arquivos serão hospedados. Em seguida, deve-se habilitar o suporte ao *Static Web Hosting*, que está disponível nas propriedades do bucket e permite que os arquivos sejam acessados diretamente por qualquer pessoa, funcionando como um servidor web comum — veja na **Figura 3** a

seta indicando o endereço de acesso. Além disso, é necessário informar o arquivo `index` do bucket a fim de evitar que a lista de arquivos do site seja exibida ao invés da aplicação (veja a **Figura 3**). Ainda podemos informar uma página de erro 404, que será exibida quando o arquivo solicitado não existir.

Depois de criado e habilitado o serviço de hospedagem do bucket, basta fazer o upload dos arquivos diretamente para ele. Visto que ele funcionará como o servidor web da aplicação, é importante que as raízes do bucket e da aplicação coincidam ou os arquivos não serão localizados corretamente.

Banco de dados

Como o endereço do servidor de banco de dados normalmente é utilizado no back-end, é uma boa ideia configurar

You can host your static website entirely on Amazon S3. Once you enable your bucket for static website hosting, all your content is accessible to web browsers via the Amazon S3 website endpoint for your bucket.

Endpoint: `demoapp1981.s3-website-sa-east-1.amazonaws.com` 

Each bucket serves a website namespace (e.g. "www.example.com"). Requests for your host name (e.g. "example.com" or "www.example.com") can be routed to the contents in your bucket. You can also redirect requests to another host name (e.g. redirect "example.com" to "www.example.com"). See our walkthrough for how to set up an Amazon S3 static website with your host name.



Figura 3. Criando um bucket para armazenamento do front-end

o banco de dados antes do back-end. Portanto, ao acessar o painel do RDS, clique em *Launch DB Instance* e escolha qual sistema gerenciador de bancos de dados será utilizado (é importante prestar atenção às indicações de quais são elegíveis para o período de gratuidade).

Feito isso, as configurações, como o tipo de hardware, espaço em disco, nome da instância, usuário e senha, devem ser selecionadas. Logo após, informamos o nome do banco de dados e escolhemos a localização do servidor, que deverá ser a mesma do servidor back-end.

Além disso, é permitido realizar configurações de rede (que só devem ser alteradas se o desenvolvedor souber o que está fazendo) e políticas de backup.

Depois de clicar para que a instância do servidor de banco de dados seja criada, leva alguns minutos até que ela fique disponível. Uma vez pronta, pode-se conectar usando o cliente apropriado para o fornecedor escolhido e executar os scripts de criação e povoamento dos dados iniciais do banco.

Back-end

Para executar o back-end da aplicação será necessário um servidor que disponha de uma máquina virtual Java. Sendo assim, no painel de controle do EC2, basta clicar em *Launch Instance*, escolher a opção *Amazon Linux* e depois o porte do servidor (há uma indicação das máquinas que são elegíveis para o período de gratuidade). Revisadas as configurações, ao clicar no botão *Launch* o servidor será iniciado logo após selecionarmos o par de chaves SSH que será utilizado para acessá-lo. A Figura 4 mostra onde localizar o nome e IP para acesso ao servidor.

Nota

É muito importante selecionar o par de chaves correto, pois sem ele não é possível acessar o servidor e será necessário destruí-lo e criar um novo. Caso ainda não haja um par criado, é possível fazê-lo nesse momento, selecionando a opção *Create a new key pair*.

A aplicação back-end, construída sobre o framework Spring Boot, agora precisa ser empacotada usando o comando *mvn package*, o que irá gerar um arquivo JAR no diretório *target* do projeto. Concluída essa etapa, o JAR deve ser enviado para o servidor, o que pode ser feito via SCP:

```
scp -i caminho_do_arquivo_chave target/arquivo.jar ec2-user@nome-do-host:/home/ec2-user
```

Com o arquivo copiado, basta iniciar a aplicação. Isso deve ser feito via console, utilizando a mesma chave conforme os comandos a seguir:

```
ssh -i caminho_do_arquivo_chave ec2-user@nome-do-host
java -jar arquivo.jar &
```

Pronto! Agora a aplicação está na nuvem.

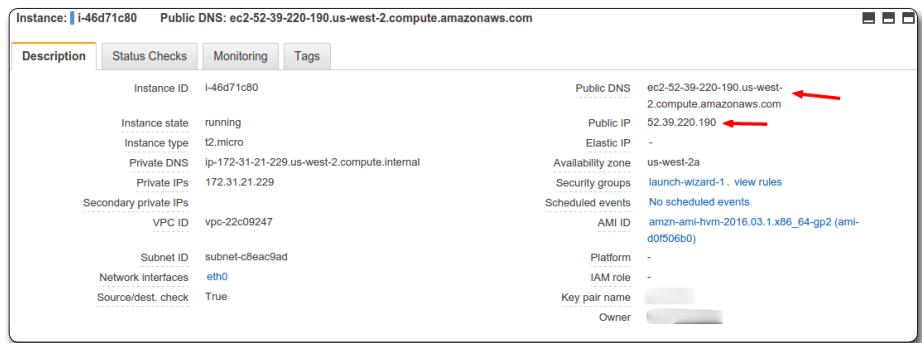


Figura 4. Criando uma instância do Linux no Amazon Web Services

Nota

Caso não seja possível acessar a aplicação back-end depois de iniciada, é preciso verificar nas configurações da instância do servidor, em grupos de segurança, entrada, se a porta utilizada está disponível.

Há inúmeras outras formas de implementar uma arquitetura que siga os mesmos conceitos, com as mesmas preocupações, ainda que utilizando diferentes tecnologias (ou as mesmas). Esse mesmo conceito pode, por exemplo, ser empregado em aplicativos móveis, com uma abordagem não nativa com o framework Ionic.

O que é importante é a atenção aos requisitos, funcionais e não funcionais, o respeito às diretrizes a serem seguidas no projeto e no desenvolvimento, assim como a aplicação de conceitos simples que facilitam e aceleram o trabalho. Ademais, é preciso ter cuidado para não cair na armadilha da otimização precoce, criando soluções caras e complexas que talvez nunca sejam necessárias de fato. Às vezes é melhor ter um desempenho 15% pior do que um atraso de 10% no prazo. Lembre-se que a diferença entre o remédio e o veneno é a dose.

Autor



Jarbas Lima

jjocenio@gmail.com - www.linkedin.com/in/jjocenio
Bacharel em Ciência da Computação pela Universidade Federal de Campina Grande, trabalha com Java há 15 anos desenvolvendo aplicações em todos os segmentos. Tem bastante curiosidade por outras tecnologias e possui as certificações SCJP e SCBCD.



Links:

Página do projeto Spring Initializr.

<https://start.spring.io/>

Projeto Spring Boot.

<http://projects.spring.io/spring-boot/>

Projeto Spring Security OAuth.

<http://projects.spring.io/spring-security-oauth/>

Página do projeto AngularJS.

<https://angularjs.org/>

Endereço da Amazon AWS.

<http://aws.amazon.com/>

Apache Camel: Um guia completo – Parte 1

Conheça esse poderoso framework que simplifica a integração entre sistemas

ESTE ARTIGO FAZ PARTE DE UM CURSO

ATI se encaixa como uma grande aliada para suportar os negócios das empresas e assegurar que todo o ecossistema globalizado e interligado funcione de forma satisfatória. Portanto, há um grande esforço por parte da TI para construir, integrar e manter os diversos sistemas que compõem todo o fluxo de negócio da empresa.

Dante dessas condições, o artigo apresentará em detalhes os problemas que geralmente são enfrentados nesse contexto de integração de sistemas, além de desafios que constantemente colocam sistemas à prova, e, as soluções/estratégias que ao longo do tempo surgiram para simplificar todo esse cenário.

Além disso, será apresentado o framework Apache Camel, que foi desenvolvido com base nessas soluções e cuja principal função é facilitar a construção de sistemas voltados especialmente para integrações.

Para iniciar o estudo do tema proposto, a próxima seção apresentará em detalhes os problemas e desafios que são largamente enfrentados pelos desenvolvedores quando eles lidam com requisitos de integração.

Problemas e desafios para a integração

Uma integração efetiva é um pré-requisito fundamental para o sucesso dos negócios, mas os desafios relacionados à mesma, como tempo, custo e problemas técnicos, podem acabar com grandes investimentos e levar ao fracasso muitos projetos. Esses problemas incluem:

- Sistemas legados baseados no desenvolvimento “tapa buraco”;
- Sistemas com arquitetura de uma única camada;
- Sistemas legados que não foram projetados para atender novos requisitos de qualidade e que são diretamente

Fique por dentro

Este artigo apresenta o Apache Camel, um framework para integração entre sistemas baseado em padrões. Esse consolidado framework open source diminui as dificuldades existentes nas integrações encapsulando código complexo através de uma linguagem simples e intuitiva, e com a utilização de componentes pré-existentes garante maior agilidade e eficiência no desenvolvimento. Portanto, aqueles que desejam aprender como o Apache Camel funciona, seus conceitos, recursos e como utilizá-lo em suas aplicações, encontrarão neste artigo um material de referência completo.

afetados pela necessidade de interoperabilidade, desempenho e segurança;

- Sistemas que foram projetados para trabalhar com os dados restritos ao uso interno;
- Decisões tomadas sem as devidas análises de arquitetura de software; e
- Falta de definições adequadas referentes ao escopo e esforço necessários para a integração.

Além dos problemas mencionados, existem alguns desafios que devem ser considerados durante a integração entre sistemas. São eles:

- Redes são instáveis e as soluções de integração implicam no transporte de dados por meio de redes que muitas vezes não são resilientes. Portanto, os sistemas integrados devem estar preparados para lidar com problemas relacionados à rede;
- Como as redes são lentas, o envio de dados por meio destas é mais lento que chamadas locais. Sendo assim, pensar em uma solução distribuída da mesma forma que se pensa em uma solução isolada pode acarretar em sérios problemas relacionados a desempenho;
- Visto que os sistemas são implementados com as mais diversas tecnologias, uma solução de integração necessita transmitir e receber dados de sistemas que utilizam diferentes linguagens de

programação, plataformas, frameworks e formatos de dados. Portanto, a solução a ser construída deve ser capaz de atender a esse cenário;

- Como mudanças são inevitáveis, sistemas são alterados a todo momento. Sendo assim, uma solução de integração deve garantir seu correto funcionamento e, consequentemente, o fluxo de negócios, independentemente das alterações realizadas nos sistemas que se conecta.

Felizmente os problemas e desafios apresentados até o momento têm sido superados pelos desenvolvedores ao longo do tempo, através de estilos de integração que foram adotados amplamente pelo mercado. Para nos aprofundarmos nesses estilos, a seção a seguir trará uma análise com mais detalhes.

Estilos de integração

Diane de todas as dificuldades e necessidades identificadas anteriormente, as integrações precisam ocorrer não apenas entre soluções homogêneas, mas também entre soluções estruturadas em plataformas distintas, separadas geograficamente dentro e fora do escopo da organização e que adotam as mais variadas tecnologias. Baseado nisso, as diferentes abordagens para que sistemas se comuniquem podem ser resumidas em quatro grandes estilos, a saber:

- **Transferência de arquivo (Electronic Data Interchange ou EDI):** um sistema escreve um arquivo de texto ou binário para que um outro sistema leia. Esse estilo exige conformidades no nome do arquivo, localização e formato (layout). Normalmente está associado ao uso do protocolo FTP (*File Transfer Protocol*). Na Figura 1 é exibido um diagrama exemplificando a integração por EDI;
- **Banco de dados compartilhado:** múltiplos sistemas utilizam um mesmo banco de dados físico para consultar e manipular dados. Com isso, não há duplicidade de informação. A Figura 2 ilustra esse estilo;
- **Invocação Remota de Procedimento (Remote Procedure Call ou RPC):** um sistema expõe suas funcionalidades para que sejam acessadas remotamente por



Figura 1. Integração entre sistemas através da transferência de arquivo

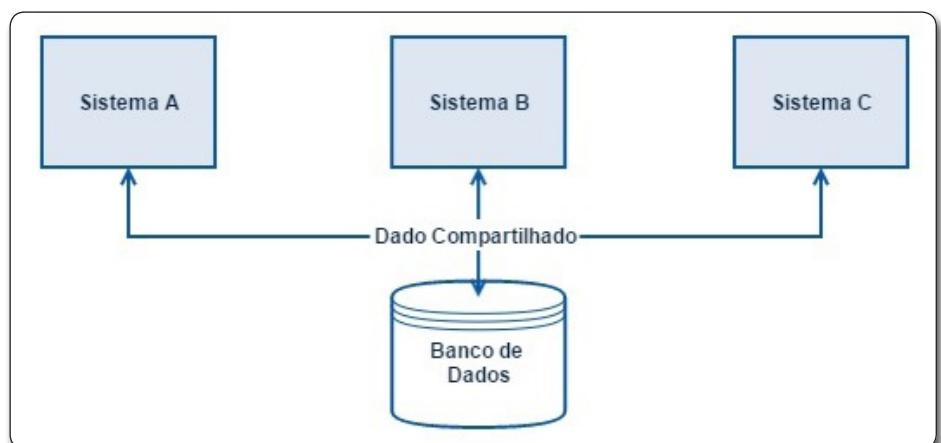


Figura 2. Integração entre sistemas através do dado compartilhado via banco de dados

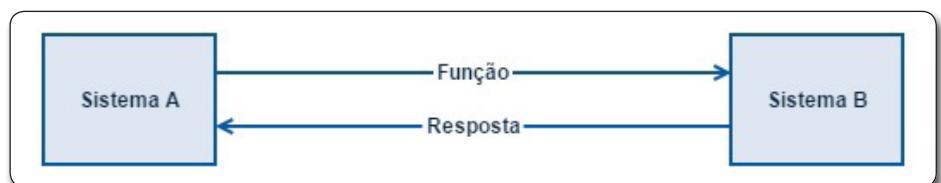


Figura 3. Integração entre sistemas através de RPC

outros sistemas. Nesse estilo a comunicação ocorre em tempo real, sincronamente e necessita que os sistemas integrados possuam compatibilidade em relação à tecnologia remota para que a comunicação funcione. A Figura 3 demonstra um cenário de integração com RPC;

- **Mensageria:** é a tecnologia que permite a entrega rápida e confiável de mensagens através da comunicação assíncrona. Nessa proposta, a mensagem é simplesmente um tipo de estrutura de dados como um texto (string), um arranjo de bytes, um registro ou um objeto, servindo como um meio de integrar sistemas. Tudo isso funciona através de um sistema que publica uma mensagem em um canal para que outro, que também possui acesso a esse canal, possa ler e interpretar essa mensagem. A Figura 4 apresenta um diagrama exibindo um exemplo de comunicação através desse estilo.

Apesar dos quatro estilos apresentados resolverem os mesmos problemas de integração, existem vantagens e desvantagens a serem consideradas na escolha de cada um. Para auxiliar essa escolha, foram definidos alguns critérios que devem ser analisados cuidadosamente. A Tabela 1 apresenta um comparativo entre os estilos de integração considerando esses critérios.

Observa-se que o estilo de mensageria é o que possui mais pontos positivos dentre os critérios comparados, ou seja, a mensageria é o estilo de integração que possui mais vantagens e por isso é considerada a melhor opção. Porém, os sistemas podem se integrar por meio de vários estilos, de forma que cada ponto da integração tire o melhor proveito de cada um.

Baseado na experiência dos desenvolvedores com o uso dos estilos apresentados, foram reconhecidos e definidos

padrões empresariais de integração que indicam soluções elegantes para problemas comumente identificados nesse contexto. A seguir, por constituírem a base do framework Apache Camel, esses padrões serão analisados.

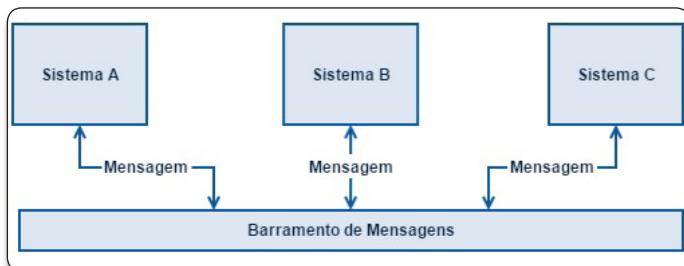


Figura 4. Integração entre sistemas através de mensagens

Enterprise Integration Patterns

Os Enterprise Integration Patterns ou EIPs foram criados para resolver uma variedade de problemas recorrentes do contexto de integração entre sistemas. Eles definem uma série de recomendações e boas práticas, independentes de plataforma e tecnologia, que foram documentadas no livro “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions”, dos autores Gregor Hohpe e Bobby Woolf (veja a seção [Links](#)).

O livro apresenta 65 padrões agrupados e classificados em seis categorias relacionadas a diferentes cenários da integração. Além disso, são indicadas as vantagens e desvantagens de cada padrão, um vocabulário e uma notação de modelagem a serem seguidos.

Mesmo divididos e categorizados, os padrões de integração podem ser combinados para atender cenários específicos do mundo real empresarial. Ademais, não são “receitas de bolo” que devem ser copiadas e colocadas para a resolução de um problema. Eles devem ser encarados como pequenos conselhos que descrevem soluções para problemas recorrentes e que por meio de tecnologia modelam padrões de negócio de forma elegante.

Visando garantir sistemas mais eficientes, flexíveis, adaptáveis e com baixo custo de manutenção, os EIPs são fortemente baseados no conceito de baixo acoplamento, onde o princípio fundamental é reduzir a dependência que os sistemas integrados têm um do outro quando trocam informações para então aumentar a testabilidade e rastreabilidade do código.

Para isso, os autores indicam o uso do estilo de integração baseado em mensageria.

Baseado em todo esse contexto de integração, juntamente com as boas práticas, padrões e mensageria, os autores do livro de EIPs propuseram diagramas formados por três elementos base com o objetivo de definir uma notação em linguagem visual que facilitasse o entendimento e representasse de forma mais clara os padrões. Essa notação acabou sendo difundida pela comunidade e consolidada como um padrão para representar e modelar visualmente a integração de sistemas. Portanto, entender os elementos base que constituem essa notação tornou-se bastante importante atualmente e sendo assim, na seção seguinte os mesmos serão apresentados.

Notação base para modelagem dos padrões

Conforme observa-se na **Figura 5**, os três elementos base das notações são: mensagem, conexão e componente. O elemento *Mensagem* representa os dados trafegando através das conexões e é constituído por uma estrutura em árvore onde o círculo sinaliza o nó raiz e os pequenos quadrados aninhados sinalizam os dados propriamente ditos. Já a *Conexão* representa o transporte dessas mensagens e o *Componente* é o item responsável por receber, processar e enviar essas mensagens.

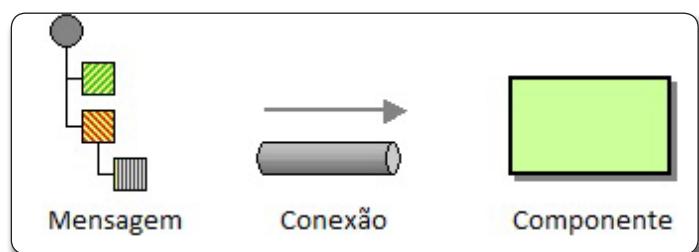


Figura 5. Elementos básicos da notação dos padrões de integração

A partir das notações exibidas é possível realizar toda a modelagem necessária para descrever um padrão. Porém, em grande parte dos cenários de integração os mesmos são utilizados em conjunto para atender uma determinada solução. Nesses casos, pode-se adotar a notação ou imagem que representa o próprio padrão conectada aos outros através de um símbolo de conexão.

| Critérios | Transferência de Arquivo | Banco de Dados Compartilhado | Invocação Remota de Procedimento | Mensageria |
|----------------------------------|--------------------------|------------------------------|----------------------------------|------------|
| Acoplamento entre os sistemas | ++ | --- | - | +++ |
| Nível de intrusão | ++ | --- | + | +++ |
| Escolha da tecnologia | ++ | ++ | ++ | - |
| Formato dos dados | + | ++ | ++ | +++ |
| Tempo para atualização dos dados | -- | +++ | ++ | ++ |
| Exposição das funcionalidades | --- | --- | +++ | ++ |
| Processamento assíncrono | +++ | ++ | ++ | +++ |
| Confiabilidade | -- | +++ | + | +++ |

Tabela 1. Comparativo de pontos positivos e negativos de cada estilo baseado em critérios comuns

Para facilitar o entendimento, a **Figura 6** apresenta um exemplo de padrões utilizados em conjunto.

Apresentados os conceitos envolvidos em um padrão e as notações básicas para representação visual, já é possível entender de forma mais clara a distribuição dos mesmos e seus propósitos específicos. Portanto, a seção a seguir abordará com mais detalhes essa classificação.

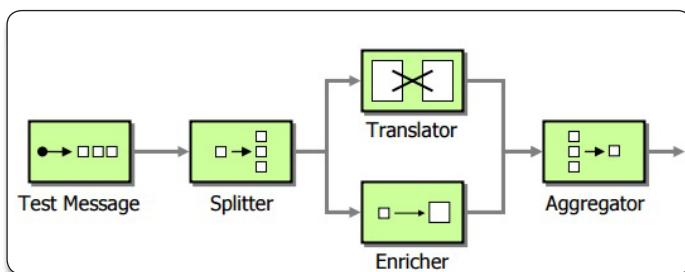


Figura 6. Padrões em conjunto para resolver um cenário de integração

Nota

Além dos elementos básicos apresentados para a modelagem visual de um padrão de integração, existe uma imagem específica e única que o representa, isto é, uma figura composta por elementos base que procuram indicar de forma direta o objetivo daquele padrão. Para a relação completa dos padrões com suas imagens de representação visual, verifique a seção [Links](#).

Classificações dos padrões

A integração entre sistemas abrange muitos problemas de domínio e negócio e por isso um bom recurso é dividir os padrões em categorias que refletem seu escopo e suas abstrações. Baseado nessa ideia, os autores do livro classificaram os padrões em seis categorias. Para apresentar a divisão proposta, a **Figura 7** exibe um diagrama completo com os padrões agrupados em suas respectivas categorias.

Conforme observa-se nessa imagem, os padrões estão classificados em:

- **Canais de Mensagens (Messaging Channels):** oferecem um meio para que aplicações possam trocar dados através de um canal conhecido, diminuindo bastante o acoplamento entre os envolvidos. A aplicação que envia o dado pode não saber exatamente qual sistema irá recebê-lo, mas há garantia de que este será enviado à aplicação correta em virtude da utilização de um canal em comum;
- **Construção de Mensagem (Message Construction ou Messaging Patterns):** oferecem soluções que descrevem as várias formas, funções e atividades que envolvem a criação e transformação da mensagem a ser trafegada entre os sistemas;
- **Roteamento de Mensagens (Messaging Routing):** oferecem mecanismos para direcionar mensagens de um remetente para um destinatário através de canais de comunicação como, por

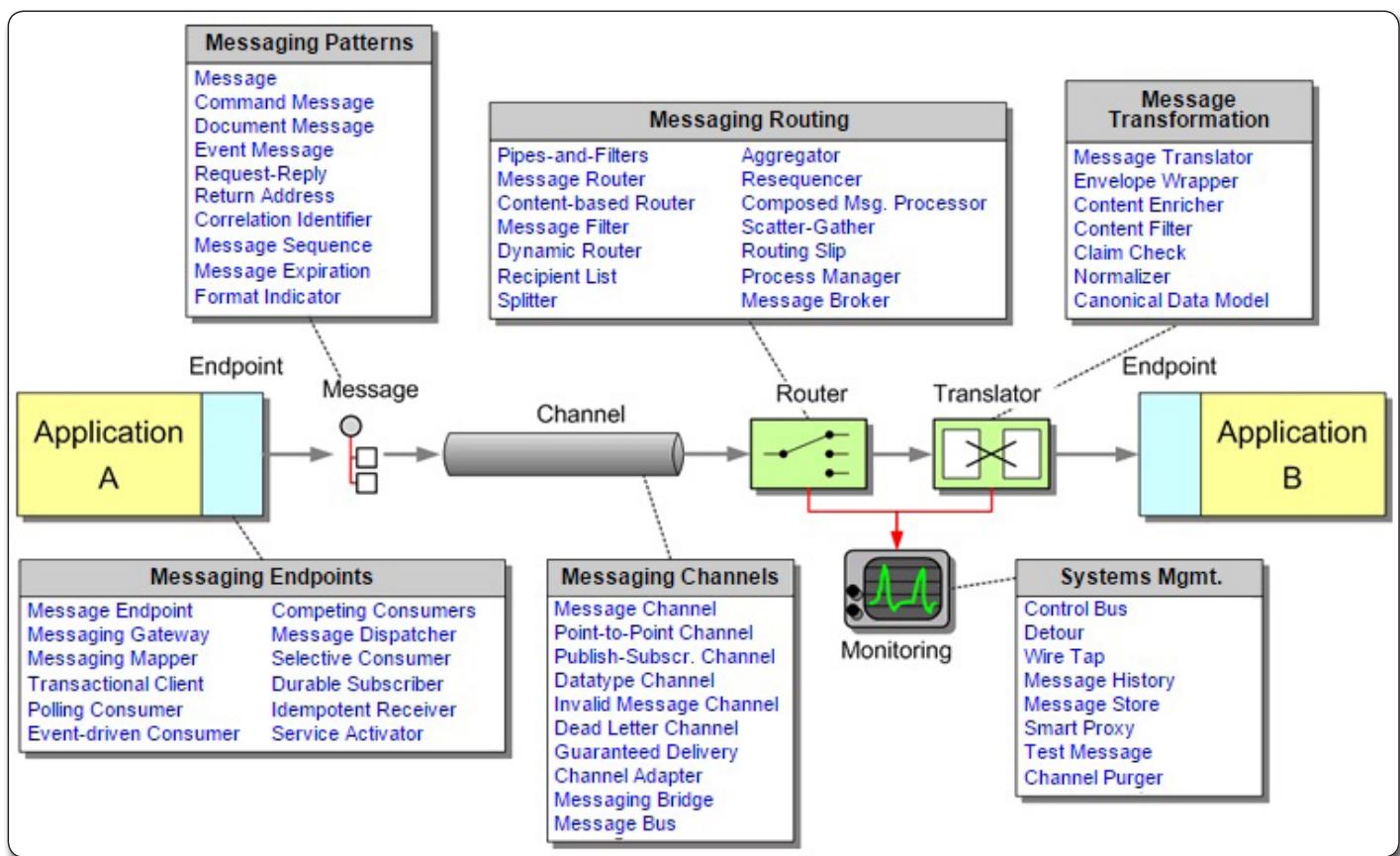


Figura 7. Categorias e seus respectivos padrões de integração

exemplo, filas e tópicos. Essas mensagens roteadas podem ser divididas, redirecionadas separadamente e reagrupadas posteriormente. Porém, vale ressaltar que esses padrões não alteram o conteúdo de uma mensagem. Eles apenas movem a mesma de um canal para outro;

- **Transformação de Mensagem (Message Transformation):** visam mudar o conteúdo de uma mensagem para diferentes necessidades de envio e recebimento. Em alguns cenários, dados precisam ser retirados ou adicionados para que as mensagens sejam reconstruídas;

- **Terminais de Mensagens (Messaging Endpoints):** esses padrões oferecem uma interface entre a aplicação e a API do servidor de mensageria (vide **BOX 1**), servindo como um “ponto de ligação”. Com isso, a aplicação pouco conhece sobre os formatos de mensagens e canais, pois o padrão encapsula as particularidades do servidor de mensageria;

- **Gerenciamento do Sistema (System Management):** proveem ferramentas para manter e gerenciar um complexo sistema baseado em mensageria funcionando, haja vista que existem soluções que podem processar bilhões de mensagens por dia. Os padrões dessa categoria lidam com condições de erro, gargalos de desempenho e mudanças nos sistemas.

BOX 1. Servidor de mensageria

O transporte de mensagens em canais de comunicação é tipicamente provido por um sistema de software separado, chamado de servidor de mensageria ou middleware orientado a mensagem (MOM – Message Oriented Middleware). O motivo pelo qual há a necessidade de se utilizar um sistema de mensageria é garantir que a mensagem seja entregue de um sistema a outro, pois as redes que os conectam são incertas e podem falhar a qualquer momento.

Exibidas as categorias e seus respectivos objetivos, o próximo passo é se aprofundar e apresentar em detalhes os principais padrões de integração, incluindo o(s) problema(s) que o mesmo busca atender e a solução proposta.

Nota

Todas as figuras que serão apresentadas durante o estudo sobre os principais padrões foram retiradas do próprio site dos EIPs, disponível na seção [Links](#).

Canais de mensagens (Messaging Channels)

Os principais padrões dessa categoria são:

- **Message Channel:** busca solucionar o problema de como uma aplicação pode comunicar-se com outra usando mensageria. A solução consiste em utilizar um canal de mensagens comum, onde uma aplicação grava a informação no canal e a outra lê. A **Figura 8** ilustra o conceito desse padrão;

- **Point-to-Point Channel:** busca solucionar o problema de como o remetente pode ter certeza que apenas um destinatário irá receber a mensagem ou executar um comando. A solução consiste em enviar a mensagem através de um canal ponto-a-ponto, onde há a garantia de que apenas um destinatário receberá a mensagem.

A **Figura 9** ilustra o conceito do padrão;

- **Publish-Subscribe Channel:** busca solucionar o problema de como um remetente pode transmitir uma mensagem a todos os destinatários interessados. A solução consiste em enviar a mensagem para um canal (tópico) que entrega uma cópia da mesma para cada destinatário. A **Figura 10** ilustra o conceito do padrão;

- **Channel Adapter:** busca solucionar o problema de como conectar uma aplicação ao servidor de mensageria para que ela possa enviar e receber mensagens. A solução consiste em acessar a API da aplicação ou seus dados, publicar mensagens em um canal baseado nesses dados, receber mensagens e chamar funcionalidades dentro da aplicação. A **Figura 11** ilustra o conceito do padrão.

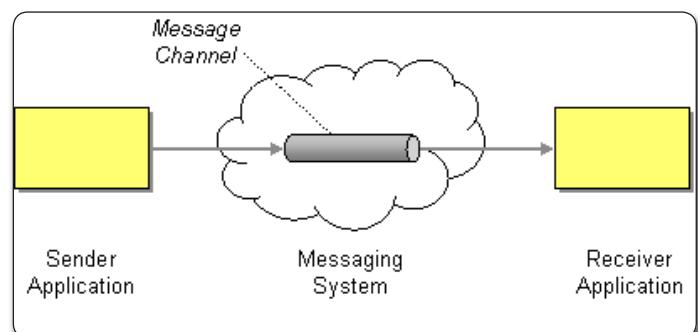


Figura 8. Message Channel

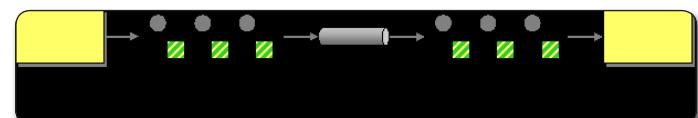


Figura 9. Point-to-Point Channel

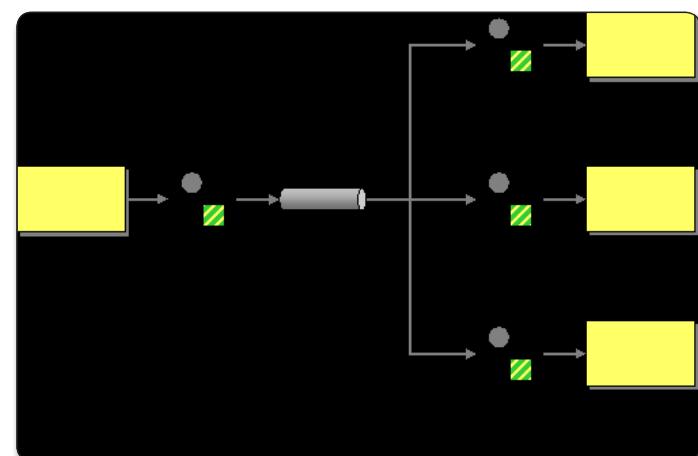


Figura 10. Publish-Subscribe Channel

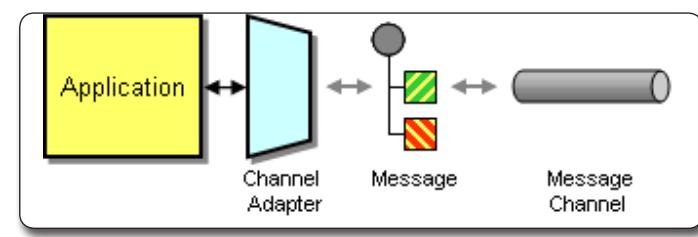


Figura 11. Channel Adapter

Construção de mensagens (Message Construction ou Messaging Patterns)

Os padrões com maior destaque na categoria de construção de mensagens são:

- **Message:** busca solucionar o problema de como duas aplicações conectadas por um canal de mensagens podem trocar informação. A solução consiste em empacotar a informação dentro de uma mensagem, ou seja, um registro de dados que o sistema de mensageria pode transmitir através de um canal de mensagens. A Figura 12 exemplifica o funcionamento desse padrão;

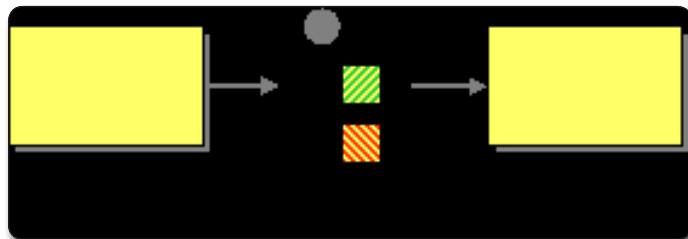


Figura 12. Message

- **Request-Reply:** busca solucionar o problema de como uma aplicação pode obter uma resposta do servidor ao enviar uma mensagem. A solução consiste em enviar um par de mensagens request-response em canais diferentes, isto é, a mensagem de envio é publicada em um canal e a resposta do servidor é publicada em outro, para que os sistemas envolvidos na integração possam identificar onde deve enviar/receber uma mensagem de request-response. A Figura 13 exemplifica o funcionamento desse padrão;

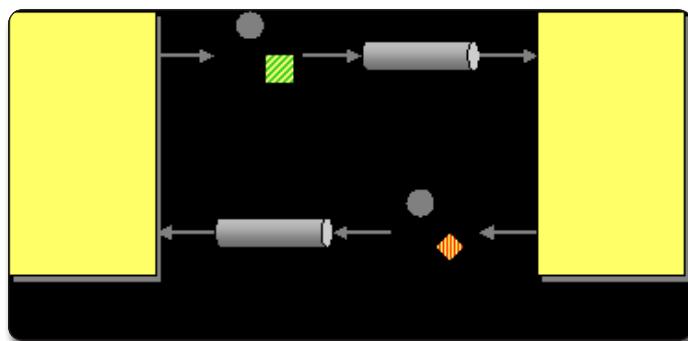


Figura 13. Request-Reply

- **Command Message:** busca solucionar o problema de como uma aplicação pode invocar uma função de outra aplicação. A solução consiste em usar uma mensagem de comando para chamar, de forma confiável, uma função em outra aplicação. A Figura 14 exemplifica o funcionamento desse padrão;

- **Message Expiration:** busca solucionar o problema de como o remetente da mensagem pode indicar quando a mesma deve ser descartada ou ignorada. A solução consiste em definir um limite de tempo para que a mensagem seja considerada válida. A Figura 15 exemplifica o funcionamento desse padrão.

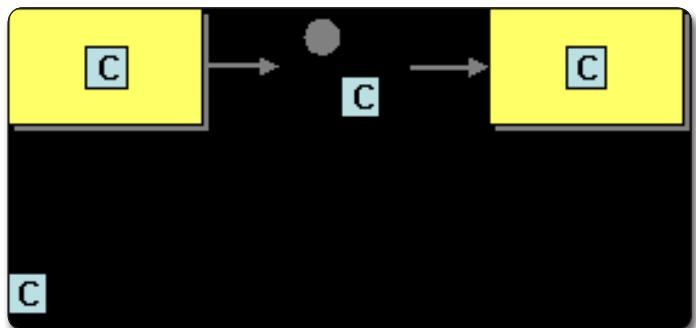


Figura 14. Command Message

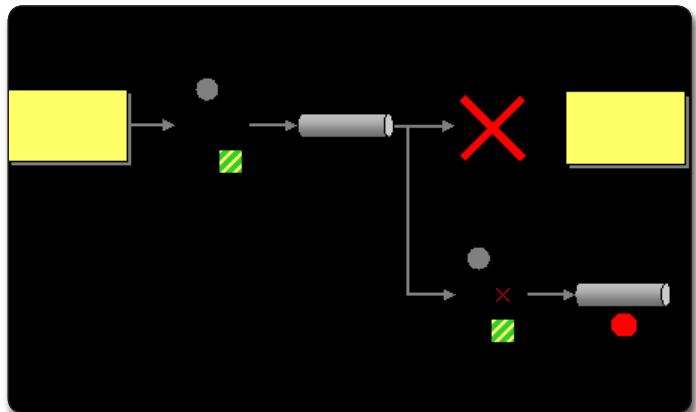


Figura 15. Message Expiration

Roteamento de mensagens (Messaging Routing)

Atualmente, os padrões mais utilizados para atender cenários onde o roteamento de mensagens é necessário são:

- **Message Router:** busca solucionar o problema de como desacoplar passos individuais de processamento de forma que mensagens possam ser passadas para filtros diferentes dependendo de uma série de condições. A solução consiste em utilizar um roteador que consome a mensagem de um canal e publica em outro dependendo de uma série de condições. Para conceituar a solução proposta, a Figura 16 ilustra o padrão;

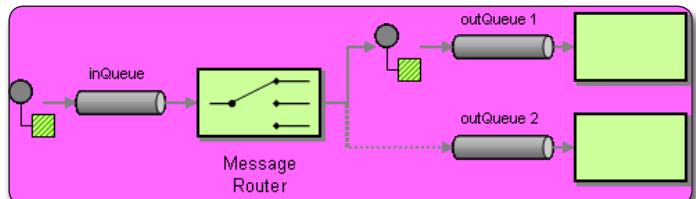


Figura 16. Message Router

- **Content-Based Router:** busca solucionar o problema de como uma aplicação deve lidar com o roteamento de uma mensagem quando existem vários destinatários disponíveis para a receberem. Dessa forma, a solução consiste em rotear cada mensagem para o receptor de acordo com o conteúdo da mensagem. Para conceituar a solução proposta, a Figura 17 ilustra o padrão;

Apache Camel: Um guia completo – Parte 1

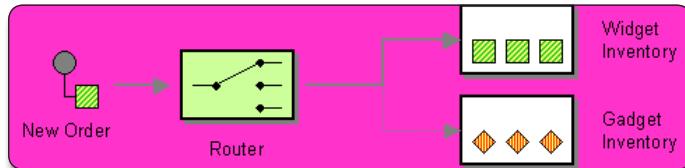


Figura 17. Content-Based Router

- **Recipient List:** busca solucionar o problema de como rotear uma mensagem para uma lista de destinatários especificados dinamicamente. A solução consiste em definir um canal de comunicação para cada destinatário, inspecionar a mensagem entrante com base em uma lista de destinatários desejados e então repassar a mensagem para todos os canais associados com os destinatários da lista. Para conceituar a solução proposta, a Figura 18 ilustra o padrão;

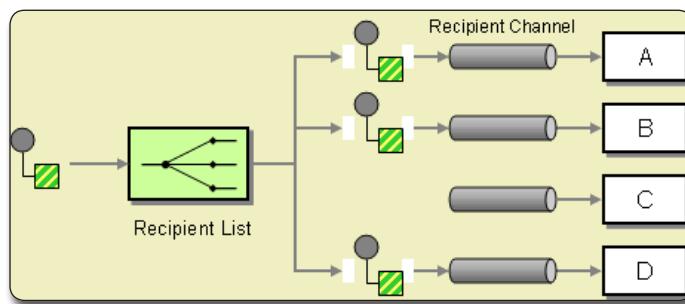


Figura 18. Recipient List

- **Splitter:** busca solucionar o problema de como uma mensagem pode ser processada quando possui muitos elementos que devem ser processados de maneira diferente. A solução consiste em dividir a mensagem em uma série de outras mensagens individuais, cada qual contendo dados relacionados. Para conceituar a solução proposta, a Figura 19 ilustra o padrão;

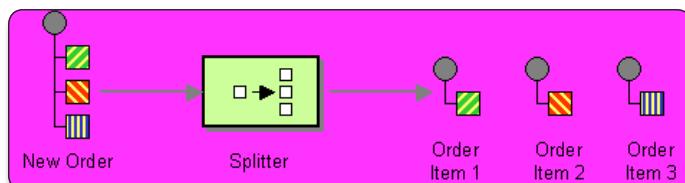


Figura 19. Splitter

- **Aggregator:** busca solucionar o problema de como agrupar mensagens individuais que estão relacionadas e devem ser processadas como uma só. A solução consiste em coletar, armazenar e agrupar todas as mensagens individuais relacionadas e então publicar uma única mensagem. Para conceituar a solução proposta, a Figura 20 ilustra o padrão;
- **Message Filter:** busca solucionar o problema de como uma aplicação pode evitar o recebimento de mensagens indesejadas. A solução consiste em utilizar um filtro para eliminar mensagens indesejadas de um canal com base em um conjunto de critérios. Para conceituar a solução proposta, a Figura 21 ilustra o padrão.

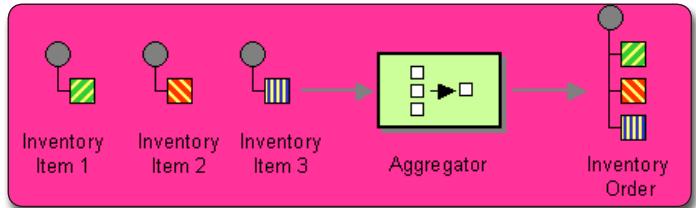


Figura 20. Aggregator

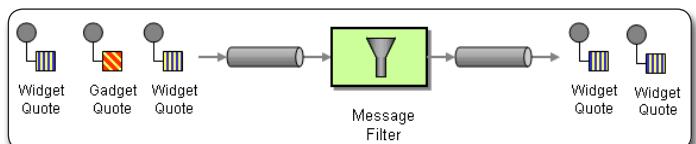


Figura 21. Message Filter

Transformação de mensagens (Message Transformation)

Na categoria de transformação de mensagens os padrões de maior relevância são:

- **Message Translator:** busca solucionar o problema de como realizar a comunicação usando mensageria para aplicações que usam formatos de dados diferentes. A solução consiste em utilizar um filtro para traduzir de um formato de dado específico para o outro. A Figura 22 retrata esse padrão;

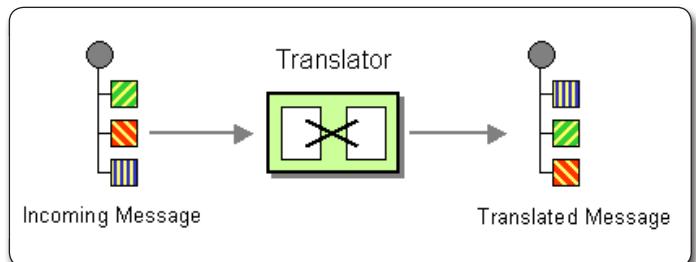


Figura 22. Message Translator

- **Content Enricher:** busca solucionar o problema de como uma aplicação vai trafegar a mensagem se a mesma não possui todos os dados necessários para o destinatário seguinte. A solução consiste em buscar num repositório externo os dados faltantes da mensagem. A Figura 23 retrata esse padrão;

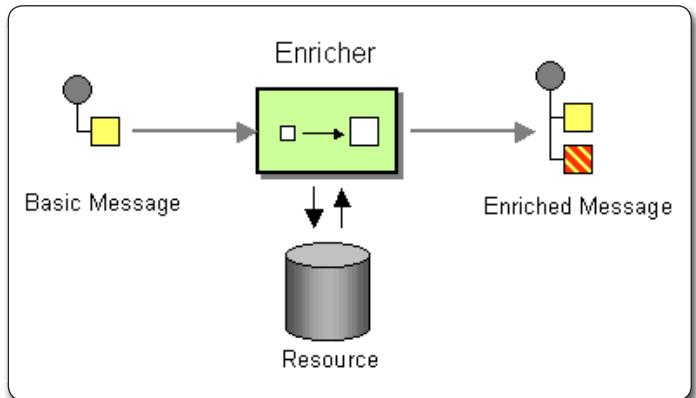


Figura 23. Content Enricher

- **Content Filter:** busca solucionar o problema de como diminuir o tamanho de uma mensagem, uma vez que nem todos os dados da mensagem são importantes. A solução consiste em utilizar um filtro para remover os dados que não são necessários. A Figura 24 retrata esse padrão;

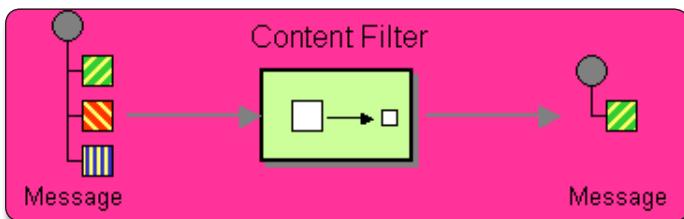


Figura 24. Content Filter

- **Claim Check:** busca solucionar o problema de como reduzir o tamanho de uma mensagem sem danificar o seu conteúdo. A solução consiste em guardar os dados da mensagem num repositório persistente e passar um comprovante da requisição para as aplicações seguintes para que quando necessário as mesmas possam utilizar esse comprovante para recuperar os dados armazenados. A Figura 25 retrata esse padrão.

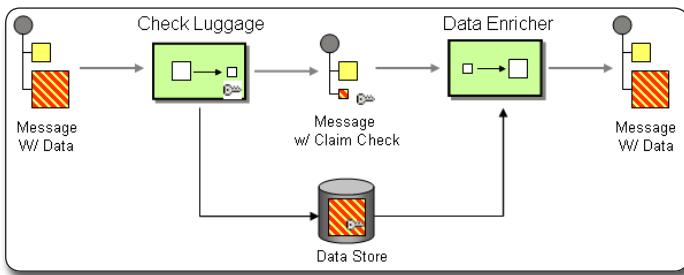


Figura 25. Claim Check

Terminais de mensagens (Messaging Endpoints)

Os padrões de terminais de mensagens (endpoints) com maior relevância são:

- **Message Endpoint:** busca solucionar o problema de como uma aplicação pode conectar-se num canal de mensageria para enviar e receber mensagens. A solução consiste em utilizar um cliente ou API do sistema de mensageria para enviar e receber mensagens. A Figura 26 ilustra o padrão;

- **Messaging Gateway:** busca solucionar o problema de como isolar o acesso ao sistema de mensageria do restante da aplicação. A solução consiste em utilizar uma classe que encapsula chamadas específicas ao sistema de mensageria e expõe uma interface com métodos específicos ao domínio da aplicação. A Figura 27 ilustra o padrão;

- **Service Activator:** busca solucionar o problema de como projetar um serviço que possa ser chamado de forma síncrona (sem o uso de mensageria) ou de forma assíncrona (com o uso de mensageria). A solução consiste em projetar um ativador de serviços que conecta as mensagens do canal ao serviço da aplicação. A Figura 28 ilustra o padrão;

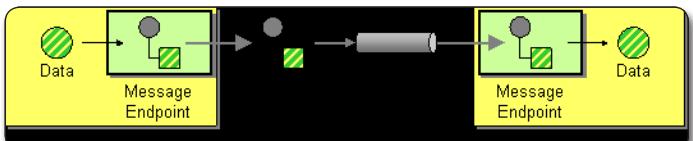


Figura 26. Message Endpoint

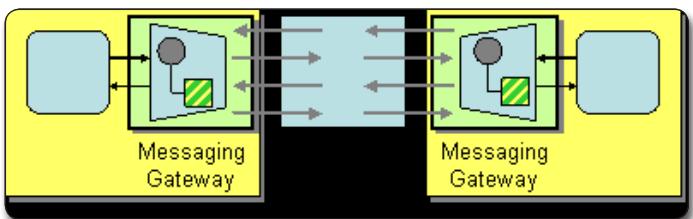


Figura 27. Messaging Gateway

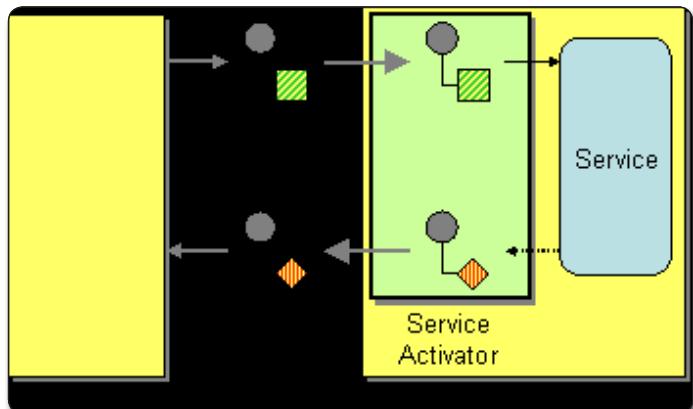


Figura 28. Service Activator

Gerenciamento do sistema (System Management)

Por fim, os padrões para gerenciamento do sistema em maior evidência atualmente são:

- **Message History:** busca solucionar o problema de como analisar e depurar de forma efetiva todo o fluxo da mensagem. A solução consiste em anexar junto a mensagem a lista de todas as aplicações em que a mesma passou desde sua origem. A Figura 29 apresenta o padrão;

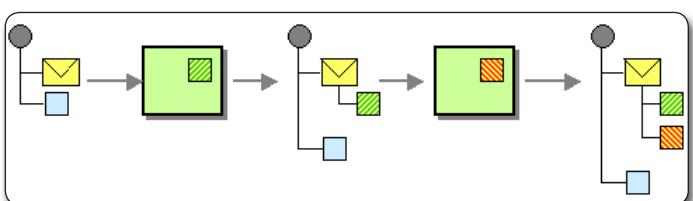


Figura 29. Message History

- **Control Bus:** busca solucionar o problema de como administrar e monitorar de forma efetiva todo o sistema de mensageria, muitas vezes distribuído em plataformas e localizações geográficas distintas. A solução consiste em aproveitar o mesmo mecanismo de mensageria das aplicações para publicar durante o fluxo mensagens de monitoramento em um canal específico. A Figura 30 apresenta o padrão;

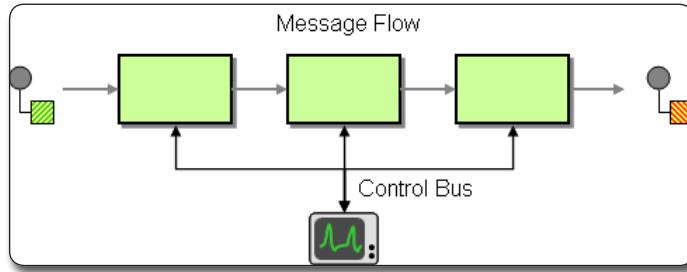


Figura 30. Control Bus

- **Detour:** busca solucionar o problema de como executar funções de validação, teste ou depuração, quando necessário, nas mensagens. A solução consiste em utilizar um roteador com um controle de condição para desviar a mensagem para etapas ou componentes intermediários antes de publicá-la de fato no canal de destino. A Figura 31 apresenta o padrão.

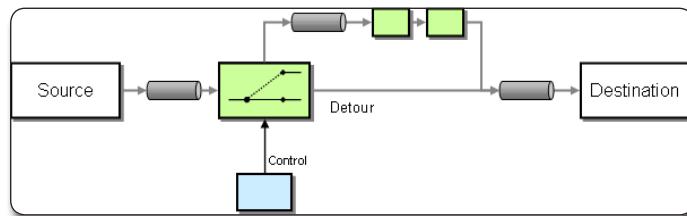


Figura 31. Detour

Apresentados os detalhes e funcionamento dos principais padrões de integração, é interessante analisarmos agora as tecnologias que estão em maior destaque no mercado e que se baseiam fortemente nos conceitos mencionados até aqui. É sobre isso que trataremos na seção a seguir.

Tecnologias para integração entre sistemas

Atualmente já estão consolidados no mercado vários frameworks e ESBs (vide BOX 2) para auxiliar os desenvolvedores na construção de sistemas voltados para integração, diminuindo a complexidade e garantindo que boas práticas identificadas e amadurecidas ao longo de anos sejam respeitadas.

BOX 2. ESB

ESB ou Enterprise Service Bus é uma plataforma de integração baseada em normas que combina as funcionalidades de mensageria, web services, transformação de dados e roteamento inteligente para conectar e coordenar com segurança a interação de um número significativo de sistemas diversos por meio do conceito de corporação estendida, isto é, sistemas que ligam as unidades locais da corporação, os parceiros de negócios, fornecedores e clientes. Para oferecer as funcionalidades mencionadas anteriormente, um ESB tem como premissa o baixo acoplamento na comunicação, além da utilização de redes distribuídas que possam ser facilmente escaláveis.

Alguns exemplos são:

- **Microsoft BizTalk Server:** conhecido também como BizTalk, trata-se de um ESB que através do uso de adaptadores permite que sistemas diferentes possam se comunicar e automatizar processos de negócio;

- **IBM WebSphere Message Broker:** o WMB é um ESB da IBM pertencente à família de produtos WebSphere. Este possibilita que as informações de negócio sejam trafegadas entre os diversos sistemas dos mais variados hardwares e plataformas de software. O produto opera sob uma Arquitetura Orientada a Serviços (vide BOX 3);

BOX 3. Arquitetura Orientada a Serviços

Arquitetura Orientada a Serviços ou simplesmente SOA é uma abordagem para definir arquiteturas de integração baseadas no conceito de serviço, sendo serviço uma função de um sistema computacional que é disponibilizada para outro sistema. Para atingir esse objetivo, SOA baseia-se na orientação a objetos, componentização e padrões de integração, visando trazer para a solução os benefícios do baixo acoplamento e encapsulamento.

- **Mule ESB:** é um ESB e também framework para integração entre sistemas capaz de lidar com sistemas desenvolvidos com tecnologias totalmente diferentes e também que disponibilizam formas de comunicação distintas para troca de informação. A plataforma é baseada em Java, mas permite interações entre outras plataformas, como .NET, usando serviços web e sockets;

- **Apache Camel:** é um framework para integração entre sistemas cuja principal função é servir como um motor de mediação e roteamento baseado em regras. Escrito em Java e de código fonte aberto sob a licença da Apache 2, implementa a maioria dos EIPs;

- **Niklas Integration Platform:** Niklas é uma plataforma de integração corporativa comercial open source. Suporta a transformação e transporte entre os vários tipos de dados e pode processar arquivos de formato livre, assim como EDIs padronizados e XML;
- **Spring Integration:** é um framework open source para integração entre sistemas que possui uma estrutura leve que se baseia no núcleo do famoso framework Spring.

No cenário atual o Apache Camel se destaca por ser uma solução gratuita, extremamente flexível, baseada nos EIPs e ser uma excelente opção para substituir um ESB, pois é muito mais simples e não perde em comparação aos recursos. Outro diferencial é que ele é de fácil integração com o Spring, acrescentando assim todo o potencial do já consagrado framework. Diante dessas vantagens, a seguir será apresentado em detalhes o Apache Camel, para que o desenvolvedor possa conhecê-lo melhor e dessa forma acrescentá-lo em seu repertório de soluções.

Apache Camel

Concebido inicialmente como um subprojeto do Apache ActiveMQ (vide BOX 4), o Apache Camel foi criado por um grupo de engenheiros de software e lançado no dia 2 de julho de 2007. Completando quase dez anos, esta solução ganhou mercado, foi base para a criação de muitos sistemas de integração, evoluiu e hoje é um framework maduro e consolidado. Por estas razões, possui atualmente uma comunidade extremamente forte, competente e ativa.

Camel é o acrônimo para *Concise Application Message Exchange Language* e recebeu esse nome principalmente pelo fato do nome

Camel ser pequeno e fácil de ser lembrado. Porém, há também outros motivos que levaram a essa escolha, incluindo o fato de um dos criadores gostar de cigarros da marca americana Camel. A lista completa com todos os motivos pode ser encontrada na seção [Links](#).

BOX 4. Apache ActiveMQ

O Apache ActiveMQ é um dos mais populares servidores de mensageria (MOM) de código fonte aberto. Escrito em Java, implementa a especificação JMS 1.1 (Java Message Service) e pode ser utilizado em modo standalone, embarcado na própria aplicação ou em um servidor Java EE. Além disso, é rápido, facilmente integrado aos EIPs, possui suporte a várias linguagens e protocolos de comunicação.

Com poucas linhas de código ou configuração é possível processar arquivos, invocar web services, publicar mensagens em um canal de mensageria, entre diversos outros recursos que precisariam de muito código, bibliotecas externas e controle interno. Portanto, nota-se que o Apache Camel provê simplicidade nas soluções de sistemas complexos, tornando a integração bastante produtiva, principalmente por trazer de forma embutida e de fácil aplicação os padrões EIPs. A ideia principal é que, com ele, os desenvolvedores foquem na implementação das regras e fluxos de negócio.

Ademais, o Apache Camel é uma excelente opção para oferecer roteamento, transformação de dados e regras de mediação na integração entre sistemas, sendo um dos mais completos frameworks de código fonte aberto para essa finalidade. Portanto, conhecer as principais características e as vantagens em utilizar o Camel é extremamente importante.

Principais características do Apache Camel

As principais características do Camel são:

- **Engine de roteamento e mediação:** representa a sua característica principal, ou seja, é o núcleo do framework. Esse mecanismo encaminha mensagens seletivamente, com base na configuração de rotas. As rotas indicam de qual endpoint a mensagem deve ser consumida e para qual deve ser enviada, além da possibilidade de serem criadas com uma combinação de implementações dos EIPs e uma linguagem específica de domínio (DSL);
- **Padrões empresariais de integração (EIPs):** implementa grande parte dos padrões de integração do livro de Gregor Hohpe e Bobby Woolf. Para verificar a lista completa dos padrões implementados pelo Apache Camel, verifique a seção [Links](#);
- **Linguagem específica de domínio:** do inglês DSL (*Domain Specific Language*), trata-se de uma linguagem criada para resolver os problemas de um domínio específico, ao contrário de linguagens de propósito geral, que são criadas para lidar com vários contextos. O Apache Camel possui uma DSL específica para trabalhar com a integração entre sistemas, sendo a principal e mais robusta baseada em Java, mas não se resume a isso. Possui também suporte a linguagens como Scala, Groovy e XML;
- **Biblioteca de componentes:** possui uma ampla biblioteca, contendo atualmente cerca de 220 componentes para as mais variadas

tecnologias, recursos e formatos de dados. Também abrange, de forma adicional, cerca de 23 linguagens de expressão e predicados, tais como JavaScript, Python, Ruby, BeanShell e Spring Expression Language, para que os mesmos auxiliem a implementação dos EIPs nas rotas em combinação com a DSL. Ademais, é possível customizar e desenvolver componentes;

- **Roteamento flexível:** pode rotear qualquer mensagem, pois não está restrito ao conteúdo desta. Essa facilidade permite que não seja necessário transformar o conteúdo da mensagem em um formato canônico para facilitar o roteamento;
- **Arquitetura modular e plugável:** o Apache Camel possui uma arquitetura modular e plugável, permitindo que qualquer um de seus componentes seja carregado independentemente de ser distribuído junto com as bibliotecas do framework, ser um componente de terceiros ou mesmo uma customização própria;
- **Modelo baseado em POJO:** Plain Old Java Objects são os elementos principais para a extensão do framework e podem ser usados em qualquer lugar e a qualquer momento no sistema. Isso possibilita estender funcionalidades internas do Camel com código personalizado;
- **Configuração simplificada:** segue o paradigma da convenção sobre configuração, que consiste em, sempre que possível, minimizar os requisitos necessários para configuração. Para isso, o Apache Camel utiliza uma configuração fácil e intuitiva;
- **Framework leve (Lightweight core):** o núcleo do Apache Camel é leve. A biblioteca total possui poucos megabytes e poucas dependências externas. Com isso, é possível incorporá-lo em qualquer aplicação standalone, aplicação Java Web ou Java EE, Google App Engine, entre outras;
- **Conversores automáticos de tipos:** possui um mecanismo interno de conversão de tipos onde não há necessidade de configurar regras de conversão;
- **Kit para testes:** disponibiliza um kit de testes que torna mais fácil testar as aplicações Camel. Inclusive é usado extensivamente para testar o próprio código;
- **Comunidade ativa:** o projeto possui uma comunidade atuante e competente, que contribui oferecendo suporte, sanando dúvidas e implementando novas funcionalidades. Para acessar a página da comunidade, veja o endereço indicado na seção [Links](#).

Nota

Vale ressaltar que o Apache Camel não é um ESB, embora contenha algumas funcionalidades deste. O Apache Camel roda embutido em uma aplicação, enquanto um ESB é um container/servidor separado e um produto complexo com mais funcionalidades, tais como: monitoramento e gerenciamento de processos, balanceamento de carga, alta disponibilidade, registro de serviços, segurança e disponibilização de ferramentas (tooling), entre outras. Por sua vez, o Apache Camel foca apenas no roteamento e mediação de mensagens.

Arquitetura do Apache Camel

A [Figura 32](#) apresenta a arquitetura do Apache Camel em alto nível. Observe que a Engine de Roteamento usa as rotas definidas em DSL para saber para onde as mensagens devem ser enviadas.

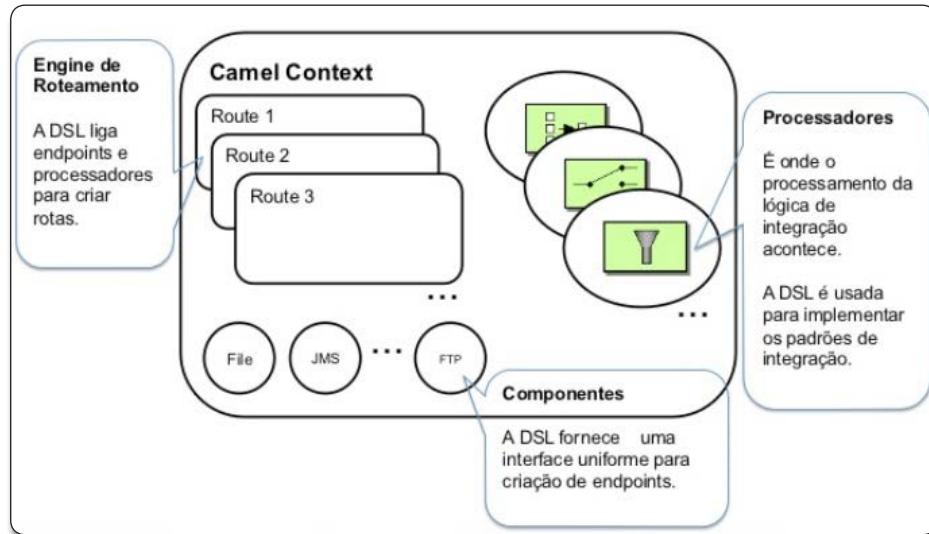


Figura 32. Arquitetura do Apache Camel

Na sequência, os Processadores são utilizados para transformar e manipular as mensagens durante o roteamento e também para implementar os EIPs; e os Componentes, para fazer a comunicação com outros sistemas. Por fim, para que o framework funcione corretamente, é necessário juntar os elementos citados em um contexto padrão do Apache Camel, para que cada um possa desempenhar sua função.

Como verificado, essa figura apresenta muitos conceitos importantes sobre o funcionamento e estrutura do Apache Camel, tais como: contexto, processadores, engine de roteamento de mensagens, endpoints e DSL. Para entender a fundo esses conceitos, os tópicos a seguir analisarão cada um, além de trazer uma explicação sobre o modelo de mensagens disponível no framework.

Modelo de mensagens

O Apache Camel oferece dois tipos de modelo de mensagem para que sejam utilizados conforme o cenário ou contexto da integração. A escolha do tipo deve ser adequada para representar o objetivo da troca de informações (comunicação) entre os sistemas envolvidos. O primeiro modelo, denominado Message, é designado para lidar com dados e fluxos simples, de modo que a mensagem é recebida pelo componente, seus dados são lidos e utilizados e na sequência é simplesmente “descartada”, enquanto o segundo, conhecido como Exchange, tem como principal função lidar com “conversas”, ou seja, informações que são trocadas entre componentes e que carregam conteúdo relevante capturado ao longo do fluxo. Na sequência, os dois modelos serão detalhados:

- **Message (org.apache.camel.Message):** é a entidade fundamental, que contém os dados que serão trafegados pelas rotas. Possui um body, headers, anexos (opcional), um identificador único e uma flag para sinalizar algum erro (*fault flag*). A Figura 33 apresenta a estrutura desse modelo de mensagem;
- **Exchange (org.apache.camel.Exchange):** é uma abstração para troca de mensagens entre rotas que incorpora o padrão MEPs (*Message Exchange Patterns*) para diferenciar comunicações assíncronas

(*InOnly*) e síncronas (*InOut*). Esse tipo de modelagem encapsula uma **Message** para representar a mensagem de entrada (*In message*) e outra para a de saída (*Out message*), uma exceção que venha ocorrer durante o fluxo, propriedades que são armazenadas durante todo o tempo de troca de mensagens e um identificador único chamado de **Exchange ID**. A Figura 34 apresenta a estrutura desse modelo de mensagem.

Dados os dois modelos de mensagens, o funcionamento interno do Apache Camel para trabalhar com ambos se inicia quando um consumidor de requisições da aplicação é invocado, isto é, um consumidor recebe mensagens. Nesse momento, automaticamente um Exchange é criado com uma Message incorporada em sua estrutura. Essa Message contém em seu body os dados da mensagem como, por exemplo, um JSON.

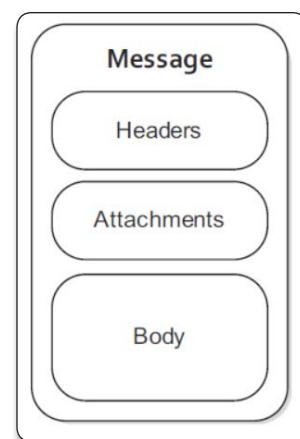


Figura 33. Estrutura de modelagem de uma Message

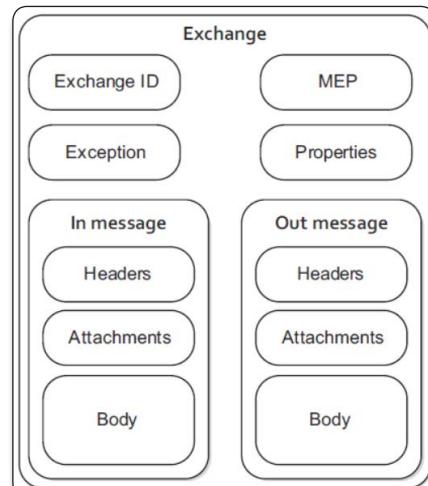


Figura 34. Estrutura de modelagem de uma Exchange

Na sequência, o Exchange inicia seu ciclo de vida e será utilizado e manipulado por toda a cadeia de processadores da rota, considerando a seguinte regra: a mensagem de saída (*Out message*) produzida pelo processador anterior é definida como a mensagem de entrada (*In message*) para o próximo. Se a mensagem de saída não for definida, então a mensagem de entrada continuará sendo repassada. Ao final da cadeia de processadores, a última mensagem de saída ou entrada (dependendo do cenário) será enviada de volta para o chamador original da requisição.

CamelContext

Conforme pode-se observar na **Figura 32**, o CamelContext é um container ou sistema de execução que mantém todos os outros recursos e mecanismos necessários juntos e conectados, provendo serviços úteis para o funcionamento do framework. A **Figura 35** mostra esses serviços e a **Tabela 2** descreve os detalhes dos mesmos.

Engine de roteamento

A engine ou mecanismo de roteamento é o que realmente move as mensagens e faz todo o trabalho pesado garantindo que as mensagens sejam encaminhadas corretamente. No entanto, ela não é exposta aos programadores.

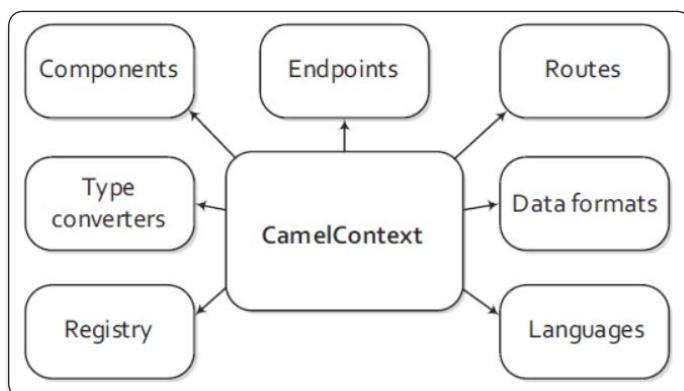


Figura 35. Serviços que o CamelContext provê

| Serviço | Descrição |
|-------------------|---|
| Componentes | Contém os componentes utilizados. Permite que os componentes sejam carregados na medida em que forem necessários através de detecção automática no classpath. |
| Endpoints | Contém os endpoints que serão criados. |
| Rotas | Contém as rotas que serão adicionadas. |
| Conversores | Contém os conversores que viabilizam que tipos de dados sejam convertidos de forma manual ou automática. |
| Formato dos dados | Contém os formatos de dados carregados. |
| Registro | Contém um registro que, por padrão, possibilita localizar os beans através de JNDI (vide BOX 5). |
| Linguagens | Contém as linguagens de expressões e predicados carregados. É permitido que diferentes linguagens sejam utilizadas para criar expressões. |

Tabela 2. Serviços e suas descrições

BOX 5. JNDI

JNDI ou Java Naming and Directory Interface possibilita a associação de um nome (ou uma representação alternativa mais simples) a recursos computacionais como: endereços de memória, de rede, de serviços, objetos e código em geral. Suas funções básicas são associar (mapear) um nome a um recurso e localizar um recurso a partir de seu nome.

A engine trabalha com o conceito de rotas – abstração base para o Camel – que são definidas através do uso de DSLs e consistem basicamente na declaração de um ou mais endpoints cuja lógica entre eles pode ser ou não a representação de um dos EIPs. Logo, a utilização de rotas pode oferecer algumas vantagens para aplicações que utilizam mensagens. Dentre essas vantagens, vale ressaltar:

- Desvinculam clientes de servidores;
- Desvinculam os produtores de mensagens dos consumidores;
- Podem decidir de forma dinâmica qual servidor um cliente deve invocar;
- Permitem que sistemas clientes e servidores sejam desenvolvidos de forma independente;
- Permitem que clientes de servidores sejam substituídos por mocks para realização de testes;
- Permitem promover as melhores práticas de design para conectar diferentes sistemas;
- Melhoram as funcionalidades e recursos de alguns sistemas.

Além das vantagens citadas, as rotas também possuem um identificador único usado para log, debug, monitoramento, inicialização e finalização do funcionamento da própria rota.

Linguagem específica de domínio

A linguagem específica de domínio ou DSL é uma linguagem de programação específica para solucionar os problemas de um domínio em particular. Por exemplo, o Apache Camel possui uma DSL baseada na linguagem Java com notações e definições próprias voltadas para atender o cenário de integração entre sistemas, facilitando dessa forma o entendimento e o foco na solução do problema.

Como um dos seus diferenciais, o Apache Camel oferece múltiplas DSLs para diversas linguagens de programação, como Java e Scala, além de permitir que as regras de roteamento de mensagens sejam especificadas em um arquivo XML com o auxílio do framework Spring. Esse variado suporte, como recém-mencionado, é uma das grandes vantagens que essa solução oferece em relação aos seus concorrentes.

Para exemplificar o uso de DSLs, na **Tabela 3** são exibidos exemplos baseados em diferentes linguagens com funcionalidades equivalentes.

Processador

Conhecido também como Processor (do inglês), é definido através da implementação da interface **Processor** (`org.apache.camel.Processor`), que disponibiliza o método `process(Exchange)`

a ser sobreescrito e a lógica de processamento seja implementada, incluindo as devidas manipulações no conteúdo da mensagem. Assim, os processadores podem ou não implementar EIPs para atender um cenário específico de integração.

| Camel DSL com Java | Camel DSL com XML e auxílio do Spring | Camel DSL com Scala |
|---|--|-------------------------------------|
| from("file:/tmp"). to("jms:aQueue"); | <route> <from uri="file:/tmp"/> <from uri="jms:aQueue"/> </route> | from "file:/tmp" -> "jms:aQueue" |

Tabela 3. Exemplos de DSLs em diferentes linguagens

Componentes

São as implementações de mais baixo nível no Apache Camel, pois são os responsáveis pela definição do protocolo ou tecnologia a ser utilizada para que um sistema se conecte a um endpoint. Para realizar tal tarefa o componente necessita de uma referência do **CamelContext** (`org.apache.camel.CamelContext`) para que seja possível criar o endpoint e estabelecer de fato a comunicação. A Figura 36 exibe um diagrama que demonstra o relacionamento e funcionamento de um componente.

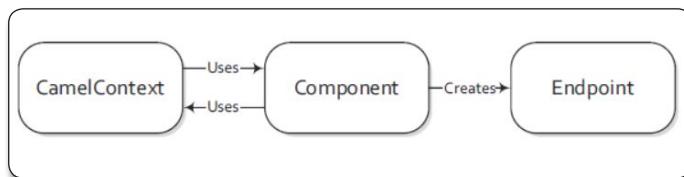


Figura 36. Funcionamento básico de um componente

Diante dos conceitos apresentados e da possibilidade de criação/customização, a lista de componentes é ampla e abrange mecanismos de conectividade, acesso a banco de dados, automação de tarefas, log, notificação, entre outros, além de cada vez mais crescer e se consolidar. Para se ter uma ideia, na versão 2.5.0 do Camel haviam cerca de 80 componentes. Já na versão mais atual, a 2.16.1 (até a escrita desse artigo), existem aproximadamente 220.

Considerando toda essa lista de componentes, a Tabela 4 apresenta, de forma sucinta, os mais utilizados pelas aplicações atualmente. Para a relação e descrição completa, além de exemplos dos componentes, veja o endereço indicado na seção Links.

Endpoint

No Apache Camel, um endpoint é uma abstração que modela o fim de um canal de comunicação e é através dele que as mensagens são enviadas e recebidas pelos sistemas. Para que isso ocorra, um endpoint deve ser criado utilizando uma URI (*Uniform Resource Identifier*).

Para demonstrar como um endpoint é declarado e criado programaticamente e também qual é sua composição básica, a Figura 37 apresenta um código de exemplo a partir do qual arquivos são lidos de um determinado diretório – no caso, `/data/inbox` – em intervalos de cinco segundos.

| Recurso | Componente |
|-------------------------------------|----------------------|
| Leitura e escrita de arquivos (I/O) | File |
| Mensagens assíncronas | JMS |
| Web services | CXF |
| Redes | MINA FTP |
| Banco de Dados | JDBC JPA |
| Mensagens em memória | Direct SEDA VM |
| Automação de tarefas | Timer Quartz |
| Log | Log |

Tabela 4. Principais componentes do Apache Camel

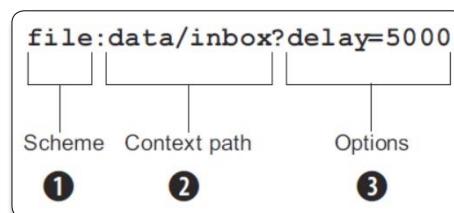


Figura 37. Exemplo e detalhes da composição de um endpoint

Nessa figura, observa-se que o endpoint (que é formado pela linha inteira) foi dividido em três partes para facilitar a explicação. A primeira parte, representada por **Scheme**, refere-se ao tipo de componente que o Apache Camel deve utilizar; neste caso, trata-se do **File**, para manipulação de arquivos. Na segunda parte temos o **Context path**, que se trata do diretório que o componente anterior deve considerar como base para a leitura de arquivos. Por fim, na terceira parte, denominada de **Options**, é especificado um parâmetro para que a leitura ocorra em intervalos de cinco segundos.

Produtor

Conhecido também como Producer (do inglês), é a abstração responsável por, internamente no framework, criar e enviar mensagens para um endpoint. Para conhecer como um produtor funciona e como está relacionado aos conceitos do Apache Camel, observe a Figura 38.

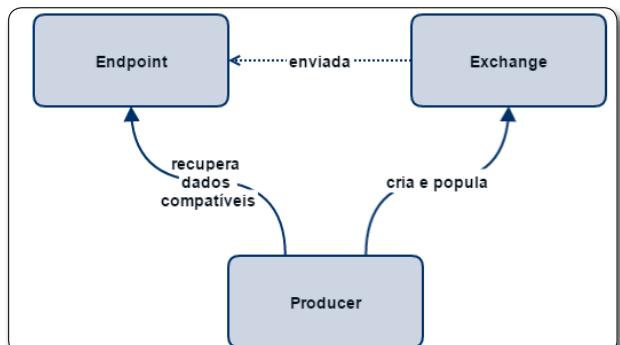


Figura 38. Ilustração de como um produtor funciona no Apache Camel

Veja que é possível observar que o produtor cria e popula um exchange com dados compatíveis que foram recuperados do endpoint para o qual a mensagem será enviada posteriormente. Essa funcionalidade é um recurso importante do Apache Camel porque abstrai toda a complexidade necessária de interação/implementação do desenvolvedor com os mecanismos de transporte de mensagem. Logo, basta configurar uma rota para enviar uma mensagem a um endpoint que o produtor fará o trabalho pesado.

Consumidor

Conhecido também como Consumer (do inglês), é a abstração responsável por receber as mensagens enviadas pelo produtor e para auxiliar nesse processo, existem dois tipos de consumidores, a saber:

- **Event-Driven Consumer (consumidor dirigido a evento):** é um consumidor assíncrono que fica aguardando que um cliente envie uma mensagem em um canal de comunicação e quando a mesma chega, imediatamente a consome e envia para processamento, ficando, logo em seguida, disponível novamente para consumir outra mensagem. Para ilustrar seu funcionamento, a Figura 39 é exibida;
- **Pooling Consumer (consumidor centralizador):** é um consumidor síncrono e em contraste com o tipo dirigido a evento, fica aguardando ativamente que um cliente envie uma mensagem em um canal de comunicação e quando uma mensagem chega, consome a mesma e não aceita nenhuma outra até que o processamento atual seja concluído. Para ilustrar seu funcionamento, a Figura 40 é exibida.

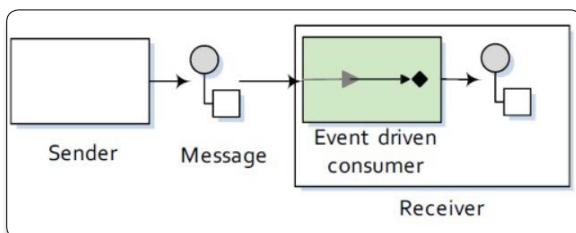


Figura 39. Ilustração de um consumidor do tipo dirigido a evento

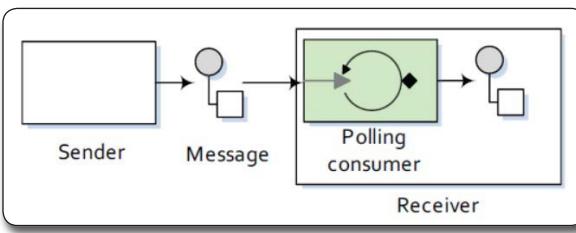


Figura 40. Ilustração de um consumidor do tipo centralizador

Segurança

A segurança é um dos requisitos mais importantes dentro de qualquer sistema e quando o cenário é integração, temas como confidencialidade, autenticidade, disponibilidade e integridade são de extrema importância e devem ser devidamente atendidos. Nessas condições, o Apache Camel oferece muitas formas e níveis de segurança que podem ser utilizados no roteamento de mensa-

gens, inclusive, empregadas em conjunto. As principais formas e níveis de segurança são aplicáveis em:

- **Roteamento das mensagens:** oferece serviços de autenticação e autorização que podem ser adicionados em rotas e segmentos de rotas. Para isso, é utilizado um padrão de estratégia para aplicar interceptadores nos processadores de mensagens. Os principais componentes dessa categoria são Shiro Security e Spring Security;
- **Conteúdo das mensagens:** oferece serviços de criptografia e descriptografia para assegurar o conteúdo das mensagens ou até mesmo partes desse conteúdo. Os principais componentes dessa categoria são XMLSecurity DataFormat, XML Security component, Crypto DataFormat e Crypto component;
- **Endpoint:** alguns componentes utilizados para comunicação e troca de mensagens oferecem recursos para proteger seus endpoints através de interceptadores, autenticação e autorização. Os principais componentes dessa categoria são Jetty, CXF, Spring Web Services, Netty, MINA, CometD e JMS;
- **Arquivos de configuração:** oferece um componente de propriedades chamado Properties para externalizar os valores e configurações para arquivos. Os valores dessas propriedades podem conter nomes de usuário e senhas, por exemplo, e por isso são criptografados e descriptografados automaticamente.

Erros e exceções

Em sistemas simples, cujo o controle do fluxo está totalmente dentro de seu contexto e ambiente, é fácil capturar erros e tratá-los, mas quando o cenário é de integração, um grande desafio é lidar com erros e problemas inesperados.

Consequentemente, vários são os riscos associados, como: a rede pode ficar indisponível, o sistema remoto pode responder em um tempo inadequado ou simplesmente pode ocorrer um evento de falha que a princípio apresenta razões obscuras. Além disso, problemas que não estão diretamente relacionados ao contexto de integração também podem ocorrer, como: disco cheio no servidor, falta de memória e lentidão devido ao alto consumo do processador.

Efetivamente, sistemas devem estar preparados para lidar com erros, problemas e situações inesperadas. Nessas condições, quando ocorrer um erro será possível dividí-lo em recuperável e irrecuperável. Um erro irrecuperável acontece quando, independentemente da quantidade de tentativas que o sistema faça para realizar uma tarefa ou função específica, ele sempre vai ocorrer. Um exemplo desse tipo de erro é verificado quando o sistema tenta acessar uma tabela que não existe no banco de dados. Já um erro recuperável é temporário, ou seja, ele pode não causar um problema na próxima tentativa. Um bom exemplo são as instabilidades de rede, que podem levar à perda de conexão em certos momentos, mas depois voltam a funcionar. Para contextualizar os dois tipos abordados, a Figura 41 apresenta um cenário.

Com base nisso, o Apache Camel orienta que um erro recuperável é derivado de **Throwable** (`java.lang.Throwable`) ou **Exception** (`java.lang.Exception`) e deve ser definido através do uso de um método exclusivo chamado `setException(Throwable cause)` em um **Exchange**.

Já um erro irrecuperável é representado por uma **Message** (que está contida em um **Exchange**) com uma mensagem de texto no body indicando o motivo ou descrição do erro e também pela definição de uma flag para indicar que houve falha, sendo o valor dessa flag informado através do método `setFault(true)`.

Para auxiliar no processo de tratamento de erros, o Apache Camel possui alguns manipuladores ou *error handlers* para as exceções do tipo recuperável. Assim, através da exceção definida em Exchange, conseguem recuperar dados e tratar o erro. Já para as exceções do tipo irrecuperável não há como utilizar os manipuladores, pois são apenas mensagens com uma flag habilitada. Para conhecer os manipuladores mais básicos de erros recuperáveis, observe a **Tabela 5**.

Além de todos esses recursos para o tratamento de erros, o Apache Camel suporta escopos que podem ser utilizados para definir diferentes níveis de manipuladores de erro. As opções de escopos são: de con-

texto (*context scope*), cujo acesso é global, ou seja, toda a aplicação pode herdar/acessar; e também o de rota (*route scope*), que está restrito a uma rota específica.

Para complementar toda essa estrutura, o Apache Camel permite ainda aplicar regras e políticas de exceções através da captura de erros com a declaração do método `onException()` na própria rota, cuja finalidade é interceptar exceções específicas a serem tratadas.

Testes

Em projetos cujo foco é a integração entre sistemas, os testes são fundamentais para aumentar ainda mais as chances de sucesso, justamente devido a todo o processo de integração, que na maioria das vezes é complexo.

Visando atender essa situação, o Apache Camel oferece um kit para tornar a implementação mais rápida utilizando como base o JUnit. A **Figura 42** exibe, em alto nível, a estrutura desse kit.

Observa-se a existência de três elementos

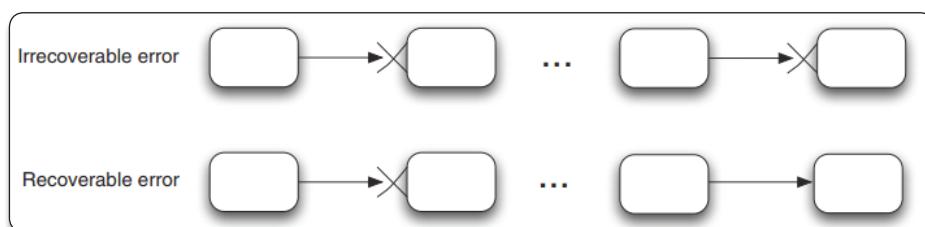


Figura 41. Classificação dos erros e conceito de retentativa com insucesso e sucesso

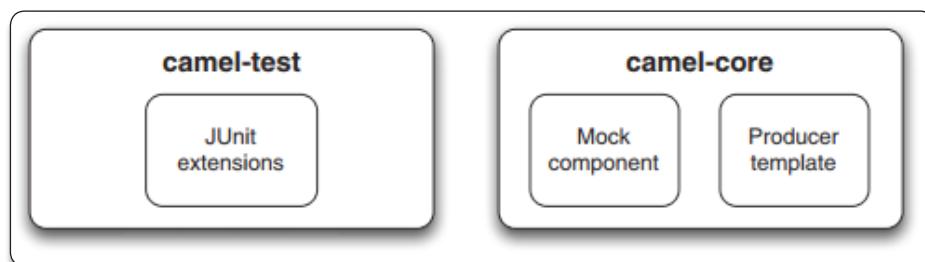


Figura 42. Kit de testes disponibilizado em dois arquivos JAR

| Manipulador de Erro | Descrição |
|---------------------------|---|
| DefaultErrorHandler | Esse é o manipulador de erro padrão. Está previamente habilitado, para o caso de outro manipulador não ser configurado. |
| DeadLetterChannel | Manipulador que implementa o padrão Dead Letter Channel (EIP). |
| K!TransactionErrorHandler | Essa opção estende o manipulador padrão e está associado a erros em transações. |
| NoErrorHandler | Manipulador que desabilita todos os manipuladores de erro. |
| LoggingErrorHandler | Manipulador que apenas faz o log da exceção ocorrida. |

Tabela 5. Manipuladores de erro oferecidos pelo Apache Camel

nessa imagem: JUnit extensions, que representa as classes do framework JUnit que facilitam os testes; Mock component, que como o próprio nome sinaliza, serve para simular componentes; e ProducerTemplate (producer), que é o mecanismo que permite enviar mensagens durante os testes.

Porém, não basta ter todo esse ferramental em mãos. É necessário saber a melhor maneira de escrever testes unitários no Apache Camel e esta maneira é enviando uma mensagem através de um canal e verificando se a mesma foi corretamente roteada pela aplicação. A **Figura 43** demonstra esse conceito de teste, onde uma mensagem com o conteúdo XXX é enviada para uma aplicação (Application) que, por sua vez, a converte para outro formato, no caso YY, e a retorna.

Diante de toda essa estrutura, vale ressaltar que assim como em código de produção, os códigos escritos para testes unitários também devem seguir boas práticas. No contexto do Apache Camel, algumas delas são:

- Escrever testes unitários com o kit de testes desde o começo do projeto;
- Utilizar o Mock component, pois é completo, simples e oferece tudo que é necessário para a escrita dos testes, sendo a sua maior contribuição voltada para a simulação e substituição de componentes (endpoints);
- Não esquecer ou desconsiderar os testes baseados em cenários alternativos, pois a integração entre sistemas é difícil e muitos problemas podem ocorrer. Portanto, verificar se a aplicação lida corretamente com falhas, por exemplo, é uma boa prática;
- Escrever testes de integração para validar se o comportamento da aplicação será conforme o esperado quando realmente estiver integrada a sistemas reais.

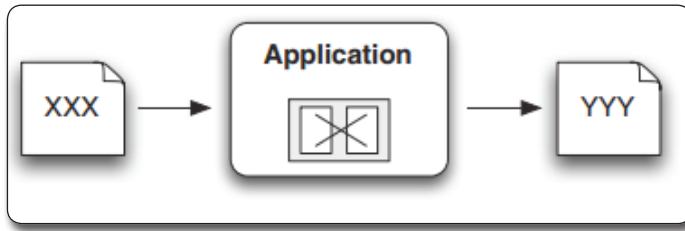


Figura 43. Exemplo de um cenário de teste

Apresentadas as características, os conceitos, vantagens, recursos e arquitetura do Apache Camel, é importante ter ciência dos desafios que o mesmo deverá enfrentar nos próximos anos para que seu sucesso continue aumentando e também para que o desenvolver fique ligado no que está por vir nas próximas versões do framework.

Desafios para o futuro

O Apache Camel está em constante evolução, seja oferecendo novas funcionalidades, componentes, recursos, suporte a formatos de dados, seja na correção de problemas e bugs. Isso tudo é reflexo de uma comunidade atuante e dedicada, que em quase 10 anos conseguiu elevar o patamar do projeto e consolidá-lo no mercado.

Apesar de todo esse avanço, o Apache Camel possui muitos desafios pela frente. Um dos principais é continuar garantindo que a agilidade e performance sejam um diferencial para qualquer sistema que preza pelo sucesso dos negócios. Portanto, o Apache Camel deve apresentar no futuro novos recursos e um suporte refinado para os seguintes tópicos:

- Escalabilidade horizontal;
- Edição de regras por analistas de negócio;
- Inteligência estatística;
- Big Data.

Esta primeira parte do artigo, através da apresentação em detalhes dos conceitos, fundamentos, estrutura e funcionamento do framework, demonstrou que o Apache Camel pode ser utilizado como solução para praticamente todos os cenários em que há integração entre sistemas, mas é indicado principalmente para aqueles em que tecnologias diferentes, plataformas e formatos de dados distintos são obstáculos e cuja complexidade não é pequena e nem grande.

Para cenários em que há poucas integrações e com tecnologias e plataformas iguais, o uso de bibliotecas específicas e destinadas a atender determinado cenário de integração é mais vantajoso, pois elimina-se o tempo de aprendizado e adaptação para utilização do Camel, tornando o desenvolvimento mais rápido e fácil, além de simplificar a arquitetura do sistema, evitando a inclusão de um framework nos quais os objetivos e funcionalidades vão muito além do que uma simples integração.

Já para projetos grandes, com muitos fluxos, integrações e sistemas com tecnologias heterogêneas, o uso do Apache Camel pode ser substituído por um ESB, pois o mesmo oferece de forma nativa muitas funcionalidades adicionais importantes para um

cenário mais complexo que, com o uso do Camel, teriam que ser adaptadas e até mesmo implementadas, gerando assim um esforço muito grande e um alto custo para manutenção.

Diante de tudo o que foi apresentado, pode-se afirmar que o futuro do Apache Camel é bastante promissor, pois dentro das empresas o número de sistemas que necessitam de integração está aumentando e as tecnologias, cada vez mais diversificadas. Nesse cenário, o Apache Camel se encaixará muito bem, pois sua finalidade principal é justamente servir como um integrador entre aplicações e serviços para atender necessidades de negócios.

Autor



Rodrigo Cunha Santana

rodcunhasantana@gmail.com

Tecnólogo em Processamento de Dados pela FATEC de Americana/SP, com Especialização em Engenharia de Software pela Unicamp, MBA em Arquitetura de Software pelo IGTI e MBA em Gestão e Governança de TI pelo Senac de São Paulo. Trabalha com Java há oito anos, entusiasta do mundo ágil e apaixonado por qualidade e design de código. Possui as certificações OCJA 5/6, SCJP 6, OCWCD 5, CSD, CSPO e CSM.



Links e Referências:

Site dos Padrões Empresariais de Integração.

<http://www.enterpriseintegrationpatterns.com>

Site do Apache Camel.

<http://camel.apache.org>

Página com a relação completa dos Padrões Empresariais de Integração.

<http://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html>

Página com os motivos da escolha do nome Camel.

<http://camel.apache.org/why-the-name-camel.html>

Página com os Padrões Empresariais de Integração suportados pelo Apache Camel.

<http://camel.apache.org/enterprise-integration-patterns.html>

Página da comunidade do Apache Camel.

<http://camel.apache.org/community.html>

Página com a lista completa de componentes do Apache Camel.

<http://camel.apache.org/components.html>

Livro sobre Padrões Empresariais de Integração.

HOHPE, Gregor; WOOLF, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional, 2003.
Livro *Camel in Action*.

IBSEN Claus; ANSTEY Jonathan. Camel in Action.

Manning Publications, 2011.
Livro *Apache Camel Developer's Cookbook*.

CRANTON Scott; KORAB Jakub. Apache Camel Developer's Cookbook.

Packt Publishing, 2013.

Livro Mastering Apache Camel.

ONOFRE Jean-Baptiste. *Mastering Apache Camel.* Packt Publishing, 2015.

Especificando e testando em Java com BDD através do Spock

Saiba como praticar o BDD utilizando o Spock como framework de especificações e testes em seus projetos

A adoção de metodologias ágeis, como Extreme Programming (XP), Scrum e Kanban, reposiciona e redimensiona o teste a uma das atividades fundamentais no desenvolvimento de software, não só pela garantia de qualidade do produto final, mas também para a própria especificação do mesmo. Para melhor definir e suportar essa atividade foram sendo criadas novas práticas e métodos, das quais podemos destacar o TDD, ATDD e o BDD. Esses, por sua vez, impulsionaram o nascimento de frameworks de testes como o JUnit, RSpec, Cucumber e o Spock, sobre o qual discorremos aqui.

O Spock é um framework de especificação e testes para Java e Groovy. Através dele podemos aplicar em nosso desenvolvimento conceitos, notação e estilos do BDD, que, como será visto, definem requisitos de sistema de forma concisa, expressiva e bastante legível, mesmo para não programadores, através de especificações e cenários.

Neste artigo veremos como o Spock pode ser aplicado no desenvolvimento e teste de software utilizando um projeto para uma aplicação web composta de serviços REST que fornecem informações geostatísticas de regiões, estados e municípios do Brasil a partir de dados do IBGE. Em sua implementação utilizaremos o Spring, mais especificamente o Spring Boot, como tecnologia base e o Maven como ferramenta de building e gerenciamento de dependências. Dessa forma, também exploraremos como o Spock pode ser naturalmente integrado tanto à fase de testes do Maven quanto aos recursos de testes do Spring.

Fique por dentro

Este artigo mostra como o Spock, um framework de especificações e testes para Java e Groovy, pode ajudá-lo na prática do BDD (Behavior-Driven Development), para o desenvolvimento e garantia da qualidade de software em seus projetos. No decorrer da leitura você também entenderá as diferenças entre os modos reativo e criativo de testes, qual a relação desse último com processos como TDD, ATDD e BDD e seu impacto sobre o papel do teste no desenvolvimento de software.

O artigo apresenta, ainda, um projeto Maven de uma pequena aplicação Spring composta de serviços REST, onde se demonstra o uso do Spock na especificação e execução de testes, geração de relatórios com os resultados em HTML e sua integração tanto com a fase de testes do Maven, através do Surefire, quanto com a infraestrutura de testes do Spring

Contudo, antes de abordarmos o desenvolvimento, vamos entender melhor o que se quer dizer com concisão, expressividade e legibilidade de testes, conforme mencionado anteriormente, para melhor avaliar o legado do BDD e do Spock. Para isso, vamos relembrar a necessidade, função e evolução do teste no desenvolvimento de software em Java, desde os simples unitários até as práticas de desenvolvimento orientado a testes, como o TDD, ATDD e o próprio BDD.

Testes unitários, integrados e o JUnit

Desde sempre houve a necessidade de se testar no desenvolvimento de software, seja em Java ou qualquer outra linguagem/ambiente. Testes visam assegurar que determinado requisito esteja implementado conforme o esperado após o desenvolvimento.

Além disso, eles não precisam ser, necessariamente, artefatos construídos por desenvolvedores, podendo ser formulados e conduzidos por, ou com o auxílio, de uma equipe de testes. Neste artigo, entretanto, abordaremos o teste como uma das responsabilidades do desenvolvedor, o qual pode cumpri-la das seguintes formas:

- Escrevendo testes *ad hoc*, ou seja, da forma e onde lhe parecer apropriado, o que em Java costuma resultar em um **main()** escrito na própria classe que se quer testar;
- Escrevendo classes de teste que podem ser executadas de maneira automatizada a qualquer momento com a ajuda de um framework de testes como o JUnit e uma ferramenta de building, como o Ant, Maven, Gradle, entre outras.

Testes *ad hoc* atendem única e exclusivamente a necessidades pontuais de asserção de qualidade do desenvolvedor que os escreve, e não há como automatizá-los de uma forma simples. Para ilustrar, vamos imaginar que para implementar o requisito “fornecer um texto com a data e hora no formato DD/MM/YYYY HH:MM:SS”, tenha sido criada a classe **Clock** como apresentada na **Listagem 1**.

Um teste *ad hoc* para o método **Clock.now()** poderia se resumir à inserção de um **main()** na classe, como mostra a **Listagem 2**.

Listagem 1. Código da classe Clock.

```
01 package br.com.devmedia.jm.woki.ibgez.misc;
02
03 import java.text.SimpleDateFormat;
04 import java.util.Date;
05 import java.util.Locale;
06
07 public final class Clock {
08     private Clock() {
09         // util
10     }
11
12     public static final String DATEFORMAT = "dd/MM/yyyy HH:mm:ss";
13
14     public static String now() {
15         return new SimpleDateFormat(DATEFORMAT, Locale.getDefault())
16             .format(new Date());
17     }
18 }
```

Listagem 2. Código da classe Clock com um teste ad hoc.

```
01 public final class Clock {
02     ...
03     public static void main(String[] args) {
04         System.out.println(Clock.now());
05     }
06 }
```

Note que o método de asserção do desenvolvedor foi simplesmente checar se o resultado de **System.out.println()** corresponde ao esperado, ou seja, o teste é um artifício feito em tempo de desenvolvimento para ser executado manualmente — e essa é a natureza do teste *ad hoc*.

Agora, para ilustrar um teste utilizando o JUnit, observemos a classe de teste da **Listagem 3**.

Listagem 3. Uma classe de teste com JUnit para Clock.

```
01 package br.com.devmedia.jm.woki.ibgez.misc;
02
03 import org.junit.Assert;
04 import org.junit.Test;
05
06 import java.text.DateFormat;
07 import java.text.SimpleDateFormat;
08 import java.util.Date;
09 import java.util.Locale;
10
11 public class ClockTest {
12
13     @Test
14     public void testNow() throws Exception {
15         DateFormat format = new SimpleDateFormat(Clock.DATEFORMAT,
16             Locale.getDefault());
17         String nowStr = Clock.now();
18         // aferir se o retorno de Clock.now() pode ser convertido em um Date,
19         // caso contrário haverá um ParseException e o teste falha
20         Date now = format.parse(nowStr);
21         // assertNotNull() aqui é redundante em termos de assercao,
22         // mas melhora a comprehensao
23         Assert.assertNotNull(now);
24         Assert.assertThat(nowStr, RegexMatcher.matches(
25             "^(\\d{2})\\/(\\d{2})\\/(\\d{4}) \\d{2}:\\d{2}:\\d{2}$"));
26     }
27 }
```

Ao contrário do exemplo *ad hoc*, nesse código encontramos mecanismos de asserção que expressam o critério de aceitação e que falharão caso o resultado não atenda o esperado. Note que não necessitamos de um **main()** em **ClockTest**, pois classes de teste são identificadas e executadas de forma orquestrada pelo JUnit, ou seja, elas não precisam de um ponto de entrada para execução.

O JUnit foi concebido para ser utilizado de forma automatizada através de ferramentas de building, como Ant, Maven e Gradle. No entanto, além disso, ele introduziu alguns conceitos e novas terminologias, a saber:

- Unit test (teste unitário): é um teste de uma única classe;
- Test case (caso de teste): testa como responde um único método a diversas entradas. Um teste unitário normalmente é composto de um ou mais casos de teste;
- Test suite (suíte de testes): é uma coleção logicamente agrupada de casos de teste;
- Test fixture: mecanismo de preparação dos dados necessários para o teste. Por exemplo, se estamos testando um código que atualiza uma lista de telefones, precisaremos de uma massa de dados contendo números de telefones que possa ser utilizada pelos diversos casos de teste;
- Test runner (executor de teste): é o componente central do framework, responsável pela identificação e execução orquestrada dos testes.

Além dos testes unitários, o JUnit também pode nos auxiliar com os integrados — que são aqueles que envolvem a asserção de contratos entre duas ou mais classes, subsistemas ou sistemas — através da categorização, utilizando a anotação **@Category**, e,

Especificando e testando em Java com BDD através do Spock

então, através de uma ferramenta de building, como o Maven, executar apenas os testes de determinada categoria. Dessa forma é possível agrupar e realizar os testes integrados isoladamente, com relação aos unitários. Apesar de útil, a categorização não é muito utilizada, pois costuma-se separar os dois tipos de teste com a própria ferramenta de building. O que importa, de qualquer forma, é não misturar os dois tipos em uma execução, pois o unitário deve aferir apenas o comportamento do nosso código, ou seja, não deve falhar pela ausência de resposta de um sistema externo ou banco de dados, por exemplo. Para ilustrar a diferença entre teste integrado e unitário, vejamos outra implementação da classe **Clock**, que chamamos **XClock**, na **Listagem 4**.

Listagem 4. Código da classe XClock.

```
01 package br.com.devmedia.jm.woki.ibgez.misc;
02
03 import org.springframework.beans.factory.annotation.Autowired;
04 import org.springframework.stereotype.Component;
05
06 import java.text.SimpleDateFormat;
07 import java.util.Locale;
08
09 @Component
10 public class XClock {
11     @Autowired
12     private TimeService timeService;
13
14     public String now() {
15         return new SimpleDateFormat(Clock.DATEFORMAT, Locale.getDefault())
16             .format(
17                 timeService.now()
18             );
19     }
20
21     public void setTimeService(TimeService timeService) {
22         this.timeService = timeService;
23     }
```

Observe, na linha 16 desse código, que a fonte de tempo não é mais a própria JVM, como em **Clock**, mas o retorno do método **now()** do serviço **TimeService**. Temos, com isso, uma dependência com outra classe que impacta o teste unitário, dado que se deve aferir apenas se nosso **now()**, e não **TimeService.now()**, comporta-se da forma esperada.

Na **Listagem 5** vemos uma classe de teste para **XClock**. Observe que, para efeitos ilustrativos, ela possui um teste integrado, o primeiro, e um unitário, o segundo, mas como foi dito anteriormente, a execução deles deve ser separada, qualquer que seja o artifício utilizado para tal.

Nessa listagem podemos notar no segundo caso de teste, **testNowUnitarioComMock()**, o uso de mocking, através do Mockito. O mocking pode ser entendido como emulação, e no caso apresentado é essencial emularmos **TimeService** para testarmos apenas nosso código, ou seja, viabilizar o teste unitário. **TimeService** é propositalmente uma interface e não uma implementação, pois dessa forma simplificamos sua emulação em nosso teste.

Listagem 5. Código da classe de teste para XClock.

```
01 package br.com.devmedia.jm.woki.ibgez.misc;
02
03 import br.com.devmedia.jm.woki.ibgez.Application;
04 import org.junit.Assert;
05 import org.junit.Test;
06 import org.junit.runner.RunWith;
07 import org.mockito.Mock;
08 import org.mockito.Mockito;
09 import org.mockito.MockitoAnnotations;
10 import org.springframework.beans.factory.annotation.Autowired;
11 import org.springframework.boot.test.SpringApplicationConfiguration;
12 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
13
14 import java.text.DateFormat;
15 import java.text.SimpleDateFormat;
16 import java.util.Date;
17 import java.util.Locale;
18
19 @RunWith(SpringJUnit4ClassRunner.class)
20 @SpringApplicationConfiguration(Application.class)
21 public class XClockTest {
22     @Autowired
23     XClock clock;
24
25     @Test
26     public void testNowIntegrado() throws Exception {
27         DateFormat format = new SimpleDateFormat(Clock.DATEFORMAT,
28             Locale.getDefault());
29         String nowString = clock.now();
30         Assert.assertThat(nowString, RegexMatcher.matches(
31             "\\\d{2}\\\\d{2}\\\\d{4} \\\\d{2}\\\\d{2}\\\\d{2}$"));
32     }
33
34     @Mock
35     TimeService timeService;
36
37     @Test
38     public void testNowUnitarioComMock() {
39         MockitoAnnotations.initMocks(this);
40         clock.setTimeService(timeService);
41         Mockito.when(timeService.now()).thenReturn(new Date(0));
42         String nowString = clock.now();
43         Assert.assertThat(nowString, RegexMatcher.matches
44             ("^31/12/1969 21:00:00$"));
45 }
```

O desacoplamento através da componentização e ligação de dependências por interfaces (contratos) resultam em testes unitários mais simples e inteligíveis, que são indicadores de boa arquitetura e desenho de software. Assim, ao escrevermos testes antes da implementação, prezando por sua simplicidade e legibilidade, induzimos boas práticas de desenvolvimento, sendo esse o principal motivo da criação do JUnit, como veremos adiante.

TDD, ATDD e o JUnit

A história do JUnit está fortemente ligada com a do XP, a primeira metodologia de desenvolvimento ágil relevante e de ampla adoção. Erich Gamma, um dos autores do legendário livro “Design Patterns: Elements of Reusable Object-Oriented Software”, e Kent Beck, o próprio criador do XP, escreveram o JUnit visando estabelecer um framework que viabilizasse a abordagem de testes da metodologia — ao mesmo tempo revolucionária e

imperativa. Revolucionária porque determina que os testes devem ser escritos antes do código da própria implementação, e imperativa porque assume como inválido e não operante qualquer código não coberto por um caso de teste automatizado.

O JUnit em si, no entanto, não impõe política de uso e pode ser utilizado independentemente da adoção do XP. Se não fosse assim, código legado ou escrito sem processo de qualidade formal por testes não poderiam usufruir de seus benefícios. Não é incomum, de fato, encontrarmos testes escritos depois da implementação, mas ainda assim de grande valia à garantia de qualidade de um produto.

Pode-se dizer que, quando testes são escritos após a implementação, eles estão em modo reativo, pois aferem código preexistente. Por outro lado, quando precedem a implementação, estão em modo criativo, visto que especificam e impactam arquitetura e desenho da implementação. O modo criativo é a essência de práticas de desenvolvimento como TDD (*Test Driven Development*) e o ATDD (*Acceptance Test Driven Development*).

O TDD é uma formalização das premissas citadas anteriormente — teste antes do código e automatização. Ele especifica e define software através de testes. Essa é, teoricamente, uma solução muito robusta para a garantia de qualidade, mas na prática pode se mostrar insuficiente, pois qualidade não pode ser sempre reduzida apenas ao atendimento de casos de teste, não importa seu número, já que os mesmos podem não estar captando corretamente o que está sendo demandado para o desenvolvimento.

Com o objetivo de diminuir os desentendimentos originados a partir da demanda feita pelo cliente, surgiu o ATDD, onde especificações são escritas não com o JUnit, mas com uma linguagem cujo domínio se aproxima de nossas linguagens naturais e que posteriormente será traduzida para casos e suítes de testes pelo desenvolvedor. Para ilustrar, vejamos na **Listagem 6** um exemplo de especificação ATDD.

Listagem 6. Exemplo de especificação ATDD.

```
01 Dado um usuário válido no sistema
02 Quando ele solicita a hora
03 Então o sistema
04 verifica se há um Locale específico na sua configuração
05 caso positivo
06 retorna a hora formatada para esse Locale
07 caso negativo
08 retorna a hora no Locale do seu cadastro
09 Casos:
10 1. locale = BR, locale da configuração do usuário = null: formato brasileiro
11 2. locale = US, locale da configuração do usuário = null: formato americano
12 3. locale = BR, locale da configuração do usuário = US: formato americano
13 4. locale = null: erro, não deve acontecer, falar com suporte
14 ...
```

Como podemos verificar, o ATDD facilita a cooperação do demandante na especificação do software, mas como o mapeamento de suas especificações para artefatos de teste não é automático, ainda pode haver imprecisão durante o processo de tradução executado pelo desenvolvedor. A solução para esse ponto fraco é o que acaba dando origem ao BDD, como veremos a seguir.

BDD

O BDD, *Behavior Driven Development*, é uma compilação de melhores práticas do TDD e ATDD com certa influência do DDD, *Domain Driven Design*. Na prática, ele melhora e se diferencia do ATDD por:

1. Definir uma terminologia a ser utilizada como linguagem de domínio para especificações, aproveitando-se de termos gerais e não técnicos, como anteriormente já introduzido pelo ATDD;
2. Definir a geração automática de artefatos de teste a partir de especificações.

Apesar de não ser a definição e implementação de uma ferramenta, e sim de um processo, sem a ferramenta o BDD não existe de fato. Por isso ele já nasce com uma, o JBehave, para Java. Além dessa, no entanto, existem outras, como o Cucumber, RSpec, entre outras, não só para Java, mas também para todas as principais linguagens/ambientes de programação, como Python, Groovy, C#, Ruby, etc. Independentemente disso, todo framework de BDD e ferramentas associadas devem, direta ou indiretamente, implementar e suportar o fluxo:

Especificação → Parser → Artefatos de Teste + Execução orquestrada dos testes gerados.

A terminologia também é algo muito importante no BDD. Ela precisa ser não só definida — para permitir sua interpretação e transformação automatizadas —, mas também se restringir a um jargão comum a todos os envolvidos no processo de construção do software, e não apenas aos desenvolvedores. No geral, as seguintes afirmações se aplicam a todos os frameworks de BDD:

- No BDD não dizemos teste, e sim especificação ou estória (*Specification/Story*);
- No BDD uma especificação é um conjunto de um ou mais cenários (*Scenario*);
- No BDD utilizamos a estrutura dado/quando/então/espera-se (*given/when/then/expect*) para descrevermos cenários. Exemplo: dada a página de login, quando o usuário entrar com credenciais válidas, então será apresentada sua página principal.

Spock

O Spock é um framework de especificações e testes para Java e Groovy que possui uma linguagem para especificações elegante, altamente expressiva e que atende o que se espera com o uso do BDD. Por possuir um JUnit runner, o Spock é compatível com as IDEs mais importantes, como o Eclipse, IntelliJ e NetBeans, assim como com ferramentas de building, como o Maven e Gradle e, consequentemente, com os servidores de integração contínua, como o Jenkins/Hudson/TeamCity.

O que diferencia o Spock de outros frameworks de BDD, como o Cucumber, é o fato da própria classe de teste funcionar como uma especificação. Isso é possível porque o Spock é escrito em Groovy, que nos permite utilizar uma string como o nome de um método, como se pode observar na **Listagem 7**, onde apresentamos um trecho da especificação **RegiaoSpec**, presente em nosso projeto exemplo.

Especificando e testando em Java com BDD através do Spock

Listagem 7. Trecho da especificação RegiaoSpec.groovy com o cenário de teste para a região centro-oeste do Brasil.

```
01 package br.com.devmedia.jm.woki.ibgez.spec
02
03 import spock.lang.Specification
04
05 import static br.com.devmedia.jm.woki.ibgez.persistence.model.Regiao.*
06 import static br.com.devmedia.jm.woki.ibgez.persistence.model.UnidadeFederal.*
07
08 class RegiaoSpec extends Specification {
09     def "regiao centro-oeste contem UFs conforme esperado"() {
10         given: "se a regiao e CO"
11         def regiao = CENTRO_OESTE
12
13         expect:"espera-se que Regiao.getUfs() retorne MS, MT, GO e DF"
14         getUfs(regiao).size() == 4
15         getUfs(regiao).contains(MS)
16         getUfs(regiao).contains(MT)
17         getUfs(regiao).contains(GO)
18         getUfs(regiao).contains(DF)
19     }
20
21     def "regiao norte contem UFs conforme esperado"() {
22         given: "se a regiao e N"
23         def regiao = NORTE
24
25         expect:"espera-se que Regiao.getUfs() retorne RO, AC, AM, RR, PA, AP e TO"
26         getUfs(regiao).size() == 7
27         getUfs(regiao).contains(RO)
28
29         getUfs(regiao).contains(AC)
30         getUfs(regiao).contains(AM)
31         getUfs(regiao).contains(RR)
32         getUfs(regiao).contains(PA)
33         getUfs(regiao).contains(AP)
34         getUfs(regiao).contains(TO)
35     }
36     def "regiao nordeste contem UFs conforme esperado"() {
37         given: "se a regiao e NE"
38         def regiao = NORDESTE
39
40         expect:"espera-se que Regiao.getUfs() retorne MA, PI, CE, RN, PB, PE, AL, SE e BA"
41         getUfs(regiao).size() == 9
42         getUfs(regiao).contains(MA)
43         getUfs(regiao).contains(PI)
44         getUfs(regiao).contains(CE)
45         getUfs(regiao).contains(RN)
46         getUfs(regiao).contains(PB)
47         getUfs(regiao).contains(PE)
48         getUfs(regiao).contains(AL)
49         getUfs(regiao).contains(SE)
50         getUfs(regiao).contains(BA)
51     }
52
53     ...
}
```

Além do nome do método que descreve o cenário, temos os labels **given** e **expect** (dado/espera-se) para definir-lo e apresentar seus critérios de aceitação, respectivamente. Labels, tanto para o Groovy quanto para o Java, apenas possuem a função estrutural de marcação, sendo que no Groovy podemos ainda adicionar a ele uma descrição, como visto na **Listagem 7**. Para o Spock, entretanto, essas marcações representam blocos funcionalmente diferentes durante a execução do cenário. Como se vê no exemplo apresentado, o label **given** corresponde à preparação do estado/contexto do teste, enquanto **expect** apresenta uma ou mais asserções a serem executadas.

Apesar do conhecimento de Groovy ser recomendável ao utilizarmos o Spock, ele não é essencial para desenvolvedores Java. Além da linguagem Groovy ser muito próxima e familiar, não precisamos utilizar nenhuma construção ou artifício dela que não seja facilmente assimilável nas especificações de nosso projeto exemplo.

Construindo e testando uma aplicação com BDD/Spock e Spring Boot

Neste tutorial iremos especificar, construir e testar uma aplicação web de serviços REST utilizando o BDD, através do Spock, e colocaremos em prática o que abordamos até aqui. Nós a chamaremos de ibgez, em referência ao IBGE, pois é através do relatório de estatísticas populacionais de municípios, estados e regiões desse instituto que extrairemos as informações que irão popular o banco de dados da aplicação. Uma instalação do Maven 3, Java 8 e conexão com a Internet — pois o Maven irá descarregar as dependências necessárias — são os únicos requisitos desse projeto.

Caso você já possua o Java 8 instalado em seu ambiente de desenvolvimento e uma IDE como o Eclipse ou IntelliJ, basta importar o código disponibilizado para download como um projeto Maven e você estará pronto para manipulá-lo. Mas, antes, vejamos seus requisitos e especificação.

Visão geral da aplicação

A ibgez deve fornecer dados geoestatísticos sobre nossos municípios, estados e regiões através de dois tipos de serviços REST, que responderão a requisições HTTP GET:

- Os serviços de consulta para município, que retornam dados de um município em específico ou do conjunto de municípios de um estado ou região em formato JSON.
- Os serviços de consulta para população, que retornam a população total de um estado, região ou do país e também os n municípios mais ou menos populosos do país.

O contexto raiz da aplicação será /ibgez, ou seja, em nosso ambiente local seus serviços serão acessados através de endereços como <http://localhost:8080/ibgez/municipio/nome/sao%20paulo>.

O conteúdo do retorno para municípios, em caso de sucesso, será uma string JSON contendo um array de objetos compostos do identificador único do registro no banco de dados, código IBGE, nome, população e UF do município, como em:

```
[{"id":3832,"codigo":"50308","nome":"São Paulo","populacao":11967825,"uf":"SP"}]
```

O conteúdo do retorno dos serviços de população, por outro lado, será um número, com exceção dos n municípios mais/menos populosos, onde também será retornado um array.

Todos os serviços, além do conteúdo, devem responder com o código HTTP 200 em caso de sucesso, 404 em caso de recurso não encontrado e 400 em caso de parâmetro inválido.

Os serviços podem ser consumidos com um navegador, como o Chrome, Internet Explorer, etc. — nesse caso não é necessário substituir os caracteres de espaço por %20, como no exemplo **são%20paulo** — ou com uma ferramenta como o **curl**, na qual será necessário tomar cuidado com caracteres sujeitos à transformação por codificação URL.

Especificação dos serviços

A partir das informações gerais apresentadas já possuímos o necessário para escrevermos nossas especificações. Em nosso projeto temos uma para os serviços de município e outra para os de população. A seguir, analisaremos o código de uma delas, apresentada na **Listagem 8**, mas antes, observe que:

1. Não há regras a serem seguidas quanto ao número e divisão das especificações, mas é recomendável respeitar um agrupamento lógico. Aqui temos uma para cada tipo de serviço. Isso reduz o escopo, simplifica, dá maior legibilidade e automaticamente induz a melhores práticas na implementação;
2. Na especificação não devemos pensar em como o sistema será implementado, mas apenas como queremos que ele responda. Veremos, adiante, que usamos um cliente REST para invocarmos as URIs de cada cenário e validamos a resposta sem precisar ter qualquer ideia de como a mesma está sendo gerada;
3. Se executarmos as especificações — e veremos como fazer isso em breve — apenas serão reportados erros, como esperado, já que a implementação ainda não existe.

Como se pode notar, o código contém apenas um trecho de *MunicipioSpec.groovy*, por questão de brevidade. A análise dele, entretanto, revelará todos os mecanismos necessários para compreender a versão completa da especificação, que é encontrada no projeto disponível para download. Vejamos:

- **Linha 13: MunicipioSpec extends Specification** — toda especificação deve estender essa classe base para que o Spock a execute como tal;
- **Linha 14: def client = new RESTClient('http://localhost:8080')** — instanciação de um cliente REST para consumo dos serviços. Como ele está sendo definido fora de qualquer método, é visível a todos eles, tanto aos que implementam cenários quanto a qualquer outro. Obviamente, o cliente poderia ser instanciado nos métodos, mas seria uma má prática, já que se trata de um recurso comum aqui;
- **Linha 15: def buildUri = { String... args -> '/ibgez/municipio' + args.join('/') }** — um método utilitário para a formação parametrizada de URIs. Ele concatena um array de strings separados por "/" à raiz /ibgez/municipio.
- **Linha 17: def "HTTP 200 para busca de dado municipio valido"()** — a definição de nosso primeiro cenário. Note que o Groovy nos permite fazer de uma string o nome de um método, ou seja, nos permite maior expressividade para que nos aproximemos da linguagem natural da especificação;

Listagem 8. Especificação com os cenários para os serviços de municípios, *MunicipioSpec.groovy*.

```
01 package br.com.devmedia.jm.woki.ibgez.spec
02
03 import br.com.devmedia.jm.woki.ibgez.Application
04 import groovyx.net.http.RESTClient
05 import org.springframework.boot.test.SpringApplicationContextLoader
06 import org.springframework.boot.test.WebIntegrationTest
07 import org.springframework.test.context.ContextConfiguration
08 import spock.lang.Specification
09
10 @ContextConfiguration(loader = SpringApplicationContextLoader.class,
11   classes = [Application.class])
12 @WebIntegrationTest
13 class MunicipioSpec extends Specification {
14   def client = new RESTClient('http://localhost:8080')
15   def buildUri = { String... args -> '/ibgez/municipio'+ args.join('/') }
16
17 def "HTTP 200 para busca de dado municipio valido"() {
18   when:
19   def res = client.get(path: buildUri(subctx, uf, cod))
20
21   then:
22   res.status == 200
23   res.data.nome == nome
24
25   where:
26   subctx | uf | cod | nome
27   '/uf' | 'sp' | '43907' | 'Rio Claro'
28   '/uf' | 'sp' | '46009' | 'Santa Branca'
29 }
30
31 def "HTTP 400 para busca de municipio com UF invalida"() {
32   client.handler.'400'= { 400 }
33   given:
34   def res = client.get(path: buildUri('/uf', 'aa', '00000'))
35
36   expect:
37   res == 400
38 }
39
40 def "HTTP 404 para busca de municipio com codigo invalido"() {
41   client.handler.'404'= { 404 }
42   given:
43   def res = client.get(path: buildUri('/uf', 'sp', '00000'))
44
45   expect:
46   res == 404
47 }
48
49 ...
```

- **Linhas 18 e 21: when: / then: (quando/então)** — são apenas labels para o Groovy, mas obviamente possuem significado especial para o Spock. **when:** apresenta a ação e/ou condição inicial do teste. No caso, quando nosso cliente executa um HTTP GET na URI desejada, **then:**, então, aferiremos que o Status HTTP da resposta é 200 e que o atributo **nome** do JSON retornado é conforme o esperado. Saiba que **when: / then:** podem ser substituídos por **given: / expect:**;
- **Linha 25: where:** — tabela de dados para ser utilizada nos testes. Trata-se de um recurso valioso do Spock e do BDD em geral, pois nos permite resumir um conjunto de condições e critérios de aceitação sem a necessidade de fazer um teste para

cada um deles ou termos muitas linhas de código em um único método que testa todas as condições. Cada uma das colunas pode ser referenciada como uma variável dentro do código, como no método `buildUri(subctx, uf, cod)` presente na linha 19, o qual irá criar a URI para o cliente REST, gerando `http://localhost:8080/ibgez/municipio/uf/sp/43907`, por exemplo, onde `subctx` é `/uf`, `uf` é `sp` e `cod` é `43907`. A resposta esperada para essa chamada é a variável `nome` = Rio Claro, no caso da primeira linha da tabela. Note que o Spock sabe, através de `where:`, que deve repetir o teste para cada uma das linhas da tabela;

- Linha 31: def “HTTP 400 para busca de município com UF invalida”() — nosso segundo cenário. Note que aqui utilizamos a construção `given: / expect:` ao invés de `when: / then:`. Como citado anteriormente, elas são intercambiáveis. Nesse método não temos a necessidade de utilizar uma tabela de dados, mas, por outro lado, precisamos criar um handler em nosso cliente para o Status HTTP 400. Essa é uma particularidade do framework que escolhemos utilizar para o consumo de REST, que pode não existir ou ser diferente em outros, mas o que importa é mantermos nossas aferições as mais simples e diretas possíveis para ganho de legibilidade.

Todos os outros cenários utilizam os mesmos mecanismos que apresentamos até aqui, mas se observarmos o trecho de `PopulacaoSpec.groovy`, apresentado na **Listagem 9**, podemos notar uma diferença importante: os labels `given:`, `expect:` e `where:` são seguidos de strings explicativas. Apesar de não serem obrigatórias, seu uso faz com que os relatórios de execução do Spock sejam muito mais expressivos, conforme podemos observar nas **Figuras 1 e 2**.

Note como o relatório para `PopulacaoSpec`, na **Figura 2**, é muito mais expressivo simplesmente porque adicionamos aos labels uma string explicativa. Essa melhoria de apresentação é importante quando consideramos que o uso do BDD objetiva melhor comunicação e responsabilização quanto ao resultado final do produto entre todas as áreas envolvidas no desenvolvimento de software. Felizmente, como vimos, com o Spock basta não esquecermos de documentar nossos labels.

Integrando o Spock a um projeto Maven/Spring

Até aqui vimos como são escritas as especificações com o Spock e Groovy. Precisamos agora entender como utilizá-las em nosso projeto de forma transparente e integrada ao Maven e ao Spring/Spring Boot. Felizmente, como veremos, essa é uma tarefa muito simples, onde devem ser considerados dois aspectos.

O primeiro deles é a integração do Spock com o Maven, que já possui uma fase de testes, como sabemos, conduzida pelo plugin Surefire, responsável pela execução dos testes unitários do projeto através dos executores de testes do JUnit. Como dissemos anteriormente, o Spock possui um executor de testes JUnit para invocá-lo, ou seja, ele pode ser utilizado naturalmente na fase de testes do Maven/Surefire, desde que sejam satisfeitas três condições:

1. O Spock deve ser declarado como uma das dependências do projeto através do POM;

2. O Maven deve ser instruído em como compilar código em Groovy;
3. O Surefire deve saber onde encontrar as especificações a serem executadas.

Todas essas três condições são resolvidas dentro do POM do projeto. A **Listagem 10** mostra a declaração de dependência do Spock no POM de um projeto Spring/Spring Boot, como o nosso. Note que a declaração explícita de versão para a dependência é opcional, pois trata-se de um artefato disponível no gerenciamento de dependências do Spring Boot.

| HTTP 200 para busca de dado municipio valido | | | | | | Return |
|--|------------------|-----------------|--------------------|---------------------------|-----------------|-------------------------|
| When: | --- | | | | | |
| Then: | --- | | | | | |
| Where: | --- | | | | | |
| Examples: | <code>/uf</code> | <code>sp</code> | <code>43907</code> | <code>Rio Claro</code> | <code>OK</code> | <code>2/2 passed</code> |
| | <code>/uf</code> | <code>sp</code> | <code>46009</code> | <code>Santa Branca</code> | <code>OK</code> | |

Figura 1. Trecho do relatório de execução do Spock para a especificação `MunicipioSpec`

| testa resposta para populacao de UF valida | | | | | | Return |
|--|---|-----------------|-----------------------|-----------------|-----------------|-------------------------|
| Given: | ao invocar /ibgez/populacao/uf/{uf} com \${uf} valida | | | | | |
| Expect: | obtem-se resposta positiva (HTTP 200) e como conteudo a populacao total da UF | | | | | |
| Where: | para as seguintes UF | | | | | |
| Examples: | <code>/uf</code> | <code>sp</code> | <code>44396484</code> | <code>OK</code> | <code>OK</code> | <code>2/2 passed</code> |
| | <code>/uf</code> | <code>rj</code> | <code>16550024</code> | | | |

Figura 2. Trecho do relatório de execução do Spock para a especificação `PopulacaoSpec`

Listagem 9. A especificação para os serviços de população, `PopulacaoSpec.groovy`.

```
01 package br.com.devmedia.jm.woki.ibgez.spec
02
03 import br.com.devmedia.jm.woki.ibgez.Application
04 import groovyx.net.http.RESTClient
05 import org.springframework.boot.test.SpringApplicationConfiguration
06 import org.springframework.boot.test.WebIntegrationTest
07 import org.springframework.test.context.ContextConfiguration
08 import spock.lang.Specification
09
10 @ContextConfiguration(loader = ApplicationContextLoader.class,
11   classes = [Application.class])
12 @WebIntegrationTest
13 class PopulacaoSpec extends Specification {
14   def client = new RESTClient('http://localhost:8080')
15   def buildUri = { String... args -> '/ibgez/populacao' + args.join('/') }
16
17   def "testa resposta para populacao de UF valida"() {
18     given:'ao invocar /ibgez/populacao/uf/{uf} com ${uf} valida'
19     def res = client.get(path: buildUri(subctx, uf))
20
21     expect:'obtem-se resposta positiva (HTTP 200) e como conteudo a populacao total da UF'
22     res.status == 200
23     res.data instanceof Integer
24     res.data == total
25
26     where:'para as seguintes UF'
27     subctx | uf | total
28     '/uf' |'sp'| 44396484
29     '/uf' |'rj'| 16550024
30   }
31
32   ...
}
```

O plugin **gmavenplus-plugin**, necessário para a compilação de arquivos **.groovy** em **.class**, é declarado conforme mostrado na **Listagem 11**. Observe que ele é parametrizado de forma a executar apenas para a fase de testes, através dos goals **addTestSources** e **testCompile**.

Finalmente, a **Listagem 12** mostra como instruímos o Surefire para a execução de testes terminados em ***Spec.class**, o que inclui nossas especificações, além dos tradicionais ***Test.class**, testes escritos em Java com o JUnit.

Listagem 10. Declarando a dependência do Spock no POM.

```
141 ...
142 <dependency>
143   <groupId>org.spockframework</groupId>
144   <artifactId>spock-spring</artifactId>
145   <scope>test</scope>
146 </dependency>
147 ...
```

Listagem 11. Configurando o Groovy no POM.

```
34 ...
35 <build>
36   <plugins>
37     <plugin>
38       <groupId>org.codehaus.gmavenplus</groupId>
39       <artifactId>gmavenplus-plugin</artifactId>
40       <version>1.5</version>
41       <executions>
42         <execution>
43           <goals>
44             <goal>addTestSources</goal>
45             <goal>testCompile</goal>
46           </goals>
47         </execution>
48       </executions>
49     </plugin>
50   ...
```

Listagem 12. Configurando o Surefire para considerar e executar nossas especificações Spock.

```
50 ...
51   <plugin>
52     <groupId>org.apache.maven.plugins</groupId>
53     <artifactId>maven-surefire-plugin</artifactId>
54     <configuration>
55       <includes>
56         <include>**/*Test.class</include>
57         <include>**/*Spec.class</include>
58       </includes>
59     </configuration>
60   </plugin>
61 </plugins>
62 </build>
63 ...
```

Assim como testes escritos em Java residem em **src/test/java**, nossas especificações em Groovy devem ficar em **src/test/groovy**. Feito isso, elas serão executadas na fase de testes do Maven, do mesmo modo que qualquer outro teste unitário em Java, seja pela invocação direta da fase através de **mvn test** ou indiretamente pelas fases posteriores, como **package** ou **install**.

Contudo, antes de executarmos os testes precisamos abordar o segundo aspecto: a integração do Spock com a infraestrutura de testes do Spring/Spring Boot, que é abordada na **Listagem 13**, a qual demonstra o que devemos adicionar a nossas especificações para que as mesmas possam testar e utilizar beans instanciados pelo contexto da aplicação. Note, no entanto, que elas só precisam ter ciência desse contexto se estiverem testando artefatos gerenciados pelo Spring — como é nosso caso, pois estamos aferindo o comportamento que será, de alguma forma, implementado pelas diversas camadas da aplicação, a qual, portanto, deve estar instanciada no momento dos testes.

Listagem 13. Anotações necessárias para que nossas especificações trabalhem de forma integrada com o Spring.

```
01 @ContextConfiguration(loader = ApplicationContextLoader.class,
02   classes = [Application.class])
03 @WebIntegrationTest
04 class MunicipioSpec extends Specification {
05   def client = new RESTClient('http://localhost:8080')
06   def buildUri = { String... args -> '/ibgez/municipio' + args.join('/') }
07 ...
```

As anotações **@ContextConfiguration** e **@WebIntegrationTest**, basicamente, fazem com que o Spring instancie a aplicação, preparando-a para os testes, tanto através de especificações com o Spock quanto com qualquer outro artefato de teste que precise aferir as camadas MVC ou outras, como, por exemplo, a de persistência.

Para concluir, vejamos como o Spock pode gerar relatórios de execução dos testes em HTML muito mais legíveis que os relatórios textuais ou em XML gerados pelo Surefire.

Utilizando um plugin de reports para o Spock

No projeto ibgez fazemos uso do plugin **Spock Reports Extension**, através do qual o Spock gera um conjunto de páginas em HTML pelas quais podemos visualizar os resultados da execução de cada especificação. Para aprimorar a aparência desse conteúdo, aspectos como folhas de estilo CSS e elementos estruturais de visualização, como TOC (*Table of Contents*), são configuráveis através de um arquivo de propriedades que deve estar no classpath, em **META-INF/services/com.athaydes.spockframework.report.IreportCreator.properties**.

Em nosso projeto utilizamos os valores default para as configurações de visualização e apenas alteramos a propriedade **outputDir**, declarando o valor **target/spock-reports** para que o Spock gere relatórios sob esse diretório sempre que executado.

O BDD é uma prática de especificação e testes relativamente nova, mas cuja adoção tem sido crescente, e não sem motivos, pois, como vimos, apesar de abordar a questão da qualidade de uma maneira aparentemente mais simples e informal, é, não só mais precisa com relação à definição do escopo e critérios de aceitação, mas também potencialmente muito mais efetiva, visto que induz à cooperação e responsabilização entre demandante e executor do desenvolvimento de software.

Especificando e testando em Java com BDD através do Spock

O Spock é um dos frameworks que nos possibilitam implementar essa prática. É uma alternativa ao Cucumber — talvez o mais utilizado entre os frameworks de BDD —, que pretende ser mais direta, pois se especifica na própria classe de teste Groovy sem perdas de expressividade e conformidade com o jargão do BDD, e leve, como consequência da ausência de artefatos adicionais entre a especificação e seu código.

Autor



Willian Oki

willian.oki@gmail.com

Trabalha com desenvolvimento de software desde 1997. Já desenvolveu em C/C++ e Perl, mas especializou-se em arquitetura e desenvolvimento Java EE desde 2005. Atuou principalmente no setor financeiro e de telecomunicações e atualmente é engenheiro de software na Adyen Latin America.



Links:

Página principal do Spock.

<https://github.com/spockframework>

Guia de Spock – Spock Primer.

http://spockframework.github.io/spock/docs/1.0/spock_primer.html

Spock Reports Extension.

<https://github.com/renatoathaydes/spock-reports>

Página principal do Groovy.

<http://www.groovy-lang.org>

Introdução sobre o BDD.

<http://dannorth.net/introducing-bdd>

Estimativa populacional do IBGE.

http://www.ibge.gov.br/home/estatistica/populacao/estimativa2015/estimativa_dou.shtml



Guia HTML 5

Um verdadeiro manual de referência com tudo que você precisa sobre HTML!



DEVMEDIA

<http://www.devmedia.com.br/guias/guia-html/3>

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486