



Edição 156 :: R\$ 14,90

Spring Data e o padrão Specification
Simplifique a construção e o reuso de consultas

Por que os processos de teste falham?

Saiba o que pode levar um processo
de teste à falha e como evitar

Apache Camel: Um guia completo

Evoluindo a solução com novas
integrações a sistemas externos

 DEVMEDIA



JAVA EE COM CDI

Como e quando utilizar
Interceptors e Decorators

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

EditorEduardo 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.

Publicidadepublicidade@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

Destaque – Boas Práticas

Conteúdo sobre Boas Práticas

06 – Spring Data e o padrão Specification: Simplifique a construção e o reuso de consultas

[*Marcio Ballem de Souza*]

Conteúdo sobre Boas Práticas

16 – Java EE CDI: Como e quando utilizar Interceptors e Decorators

[*Gabriel Novais Amorim*]

Artigo no estilo Curso

24 – Apache Camel: Um guia completo – Parte 3

[*Rodrigo Cunha Santana*]

Destaque – Reflexão

Artigo do tipo Engenharia de Software

41 – Por que os processos de teste falham?

[*Renata Eliza*]

Programador Java: Por onde começar?

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



DEVMEDIA

Spring Data e o padrão Specification: Simplifique a construção e o reuso de consultas

Conheça neste artigo o padrão Specification e, junto com Spring Data, descubra como construir diferentes consultas a partir do reuso

Spring Framework é uma ferramenta de grande sucesso dentro do mundo Java. Para alcançar esse reconhecimento, vários dos seus recursos são projetados com a finalidade de simplificar o dia a dia dos programadores, fazendo com que eles utilizem princípios básicos de orientação a objetos, como herança, polimorfismo, baixo acoplamento, alta coesão e reuso. Com a intenção de fornecer facilidades de implementação aos desenvolvedores, o Spring teve sua arquitetura desenvolvida com base em diversos padrões de projeto. Ao mesmo tempo, ele incentiva seus usuários na adoção de padrões para que eles obtenham uma maior qualidade e produtividade no desenvolvimento de suas aplicações. Esse incentivo pode ser visto em recursos como Injeção de Dependências, Inversão de Controle, Model View Controller (por meio do Spring MVC) e o padrão Repository (por meio do Spring-Data).

Esse último é uma excelente opção para desenvolver a camada de acesso a dados em aplicações Java. O padrão Repository, no Spring-Data, faz com que o desenvolvedor programe apenas interfaces com as assinaturas dos métodos de consulta. Deste modo, o Spring-Data se encarregará da implementação desses métodos em tempo de execução. Para isso, as consultas são basicamente escritas por palavras-chave adicionadas ao nome do método, ou então, por código JPQL incluído em uma

Fique por dentro

Este artigo tem como objetivo introduzir o leitor ao padrão de projeto Specification, elaborado por Eric Evans para a camada de persistência de dados. O objetivo central desse padrão é desacoplar os critérios de seleção de uma entidade de modo que eles possam ser usados separadamente ou de forma combinada para se montar consultas específicas com maior flexibilidade.

A partir de uma implementação já predefinida no Spring-Data JPA para o padrão Specification, se torna mais simples seu uso, bastando ao desenvolvedor fornecer elementos básicos no processo de cada consulta que deve ser criada.

Assim, este artigo é útil por demonstrar como trabalhar com o Specification junto ao Spring-Data JPA em busca de uma maior produtividade no desenvolvimento de aplicações com acesso a banco de dados.

anotação do tipo `@Query` em sua assinatura. As consultas via Repository podem, ainda, serem substituídas ou desenvolvidas paralelamente com o padrão de projeto Specification.

Esse padrão tem como objetivo flexibilizar as regras existentes na construção de consultas a bancos de dados. Por exemplo, um método básico de consulta poderia receber como argumento um objeto do tipo `Integer` para recuperar uma entidade por um `id`. Mas esse mesmo método, tendo um `Integer` como argumento, não poderia receber como parâmetro um objeto `String` na tentativa

de recuperar uma entidade por **nome**. Dessa forma, um novo método de consulta precisaria ser implementado para aceitar um parâmetro do tipo **String**.

Pensando nisso, o Specification é empregado como uma solução que visa separar as regras usadas na criação das consultas, possibilitando ter, por exemplo, um único método na camada de persistência que execute as consultas planejadas. Nesse cenário, as regras são encapsuladas em objetos chamados de predicados, que executam um papel específico dentro do padrão: representar um argumento do método de consulta na camada de persistência.

Aproveitando essa facilidade, o Spring-Data JPA já oferece parte do padrão Specification implementado. Assim, se torna mais fácil para o desenvolvedor trabalhar com ele, sendo necessário apenas criar algumas classes que definem mais especificamente quais ações devem ser executadas. Esse processo será apresentado durante o artigo, para que o leitor tenha uma introdução ao uso do padrão de projeto Specification junto com o Spring-Data JPA.

Padrões de Projeto

Como este artigo tem o objetivo de abordar o padrão Specification, elaborado por Eric Evans, é interessante saber um pouco mais sobre o que são padrões de projeto. Apresentados pela primeira vez em 1977, os padrões foram criados por Christopher Alexander, que publicou um catálogo com cerca de 250 padrões que discutiam questões comuns da arquitetura civil, descrevendo em detalhes o problema e as justificativas de cada solução.

Com o tempo, a área de desenvolvimento de software também passou a ter os seus próprios padrões. Tais elementos surgiram com objetivos semelhantes aos propostos por Alexander, a saber: explorar um problema, buscar uma solução e fornecer um modelo a ser seguido por outros desenvolvedores para o uso dessa solução. Essa necessidade ocorreu devido à baixa qualidade encontrada nos códigos fontes, que demonstravam ser de difícil interpretação, reutilização e manutenção.

Os padrões de projeto se tornaram ainda mais populares com o advento da orientação a objetos, onde seu uso é considerado uma boa prática a ser seguida na busca por uma maior qualidade no desenvolvimento de software. Quando um software é construído com qualidade, ele vai proporcionar uma maior flexibilidade e organização, tornando mais fácil sua manutenção e atualizações futuras, assim como favorece a técnica de reuso, conceito muito importante que possibilita o reaproveitamento de código. Exatamente por isso existem vários padrões que primam por esse conceito, como é o caso do DAO (*Data Access Object*), Service Layer, Model-View-Controller, Repository, Specification, entre outros.

Predicados em JPA

No padrão de projeto Specification há uma regra definindo que as consultas devem ser construídas com base em predicados, um tipo de objeto que tem como objetivo armazenar uma condição ou um tipo de critério qualquer. Por exemplo, se um valor é igual, menor, maior ou igual a outro.

Antes de explorarmos o uso desse padrão junto ao Spring-Data JPA, vamos analisar como trabalhar com predicados em consultas criadas com a API Criteria da JPA (veja o **BOX 1**). Essa abordagem é importante para que seja possível perceber o quanto a implementação de Specification no Spring-Data facilita o uso de predicados.

Para isso, vamos supor que exista uma entidade chamada **User**, e dentro dela, dois atributos, nomeados como **name** e **age**, que serão parâmetros em uma consulta baseada em predicados via API Criteria, conforme o pseudocódigo encontrado na **Listagem 1**.

BOX 1. API Criteria

A API Criteria é uma API que faz parte da especificação JPA com o objetivo de definir consultas ao banco de dados de forma alternativa às consultas do modelo JPQL. Assim, enquanto uma JPQL é escrita via String, a consulta por Criteria é baseada em métodos de interface e classes. A principal vantagem da API Criteria é que os erros na construção da consulta podem ser detectados mais cedo, isto é, em tempo de compilação, ao passo que as consultas baseadas em JPQL têm os erros de escrita detectados apenas em tempo de execução. Apesar dessa diferença, as consultas elaboradas com JPQL ou API Criteria são consideradas iguais em relação à desempenho e eficiência.

Listagem 1. Consulta via API Criteria em JPA.

```
1 EntityManager em = ...;
2 CriteriaBuilder builder = em.getCriteriaBuilder();
3 CriteriaQuery<User> query = builder.createQuery(User.class);
4 Root<User> root = query.from(User.class);
5 Predicate pName = builder.equal(root.get("name"), name);
6 Predicate pAge = builder.lessThan(root.get("age"), age);
7 query.where(builder.and(pName, pAge));
8 em.createQuery(query.select(root)).getResultList();
```

Analizando o código dessa listagem, podemos compreender cada linha da seguinte maneira:

1. O objeto **EntityManager** é criado a partir de uma fábrica de **EntityManager** omitida no código;
2. **CriteriaBuilder** é uma interface que fornece vários métodos que representam diferentes tipos de condições, como: **and**, **or**, **between**, entre outras;
3. **CriteriaQuery** é instanciada para criar um objeto de consulta para a entidade do tipo **User**;
4. **Root** define uma variável que representa a cláusula **FROM** da consulta;
5. Um predicado é criado para um atributo do tipo **name**;
6. Um predicado é criado para um atributo do tipo **age**;
7. Os predicados são adicionados à cláusula **WHERE** com a condição **AND**;
8. A consulta é executada a partir das regras definidas nas linhas anteriores.

O principal problema com esse código é que os predicados não são fáceis de exteriorizar e, portanto, reutilizar, pois você precisa, em primeiro lugar, configurar o **CriteriaBuilder**, **CriteriaQuery** e **Root**. Isso significa que seria muito complicado utilizar o mesmo método para aceitar consultas com diferentes

parâmetros e critérios. Além disso, a legibilidade do código é considerada pobre e, à primeira vista, de difícil interpretação, devido à declaração de tantos objetos diferentes.

É nesse ponto que o Spring-Data JPA passa a ser um grande aliado do desenvolvedor. Ele abstrai a complexidade do código da **Listagem 1** e fornece uma forma mais simples de se trabalhar com o uso de predicados via JPA, como será visto logo mais.

O Spring-Data JPA

Agora que já sabemos um pouco sobre predicados, vamos passar a conhecer algumas características do Spring-Data JPA. Essa solução representa um dos principais projetos da família Spring-Data, a qual também é composta pelo Spring-Data MongoDB, Spring-Data Redis, Spring-Data REST, entre outros.

O Spring-Data JPA visa melhorar significativamente a camada de acesso a bancos de dados relacionais. Por meio de interfaces do tipo Repository, tal solução omite do desenvolvedor a complexidade do uso de classes que envolvem a especificação JPA, tornando mais fácil a construção de consultas, processos de paginação e auditoria.

Para o desenvolvedor, resta a responsabilidade de criar nos repositórios as assinaturas dos métodos de consulta. Feito isso, o Spring-Data JPA fornecerá a implementação desses métodos automaticamente ao interpretar palavras-chave declaradas nos nomes dos métodos, tais como: **BY, AND, OR, LIKE, BEFORE, NOT, IN**, entre outras, combinadas com os nomes das propriedades de uma entidade. Por exemplo, um método nomeado como **findByNameAndIdade()** usa duas palavras-chave: **BY** e **AND**. O Spring-Data as identifica e as combina com os nomes das propriedades (**Nome** e **Idade**) da entidade. Assim, a consulta é criada automaticamente em tempo de execução.

Outra forma de trabalhar com consultas no Spring-Data JPA é declarando a anotação **@Query**. Esse recurso recebe como valor uma **String** que contém uma consulta na linguagem JPQL. Quando optamos por essa solução, deixa de ser necessário o uso de palavras-chave na assinatura do método.

Para saber mais sobre o Spring-Data JPA, veja o endereço indicado na seção **Links**.



Classe de entidade

Cada tabela no banco de dados recebe a denominação de entidade, e uma entidade é representada no código de uma aplicação por uma classe de entidade. Quando usamos JPA, as entidades são mapeadas por meio de anotações, para que um framework do tipo ORM, como o Hibernate, possa relacionar a classe de entidade e seus atributos com a tabela e suas colunas no banco de dados.

A classe de entidade também é importante para o Spring-Data JPA, já que cada repositório na camada de persistência deve ser específico a uma única classe de entidade. Dessa forma, o Spring-Data JPA consegue identificar e relacionar os nomes das propriedades da entidade com as propriedades digitadas como parte do nome dos métodos, ou mesmo, da consulta JPQL atribuída a uma anotação do tipo **@Query**.

Nota

Um framework ORM (Object-Relational Mapping ou mapeamento objeto-relacional) é uma ferramenta com o objetivo de aproveitar ao máximo o paradigma de orientação a objetos no desenvolvimento de aplicações com acesso a banco de dados. Uma das suas responsabilidades é criar uma ponte entre o modelo relacional e o modelo orientado a objetos a partir de um mapeamento via arquivos XML ou anotações. Além disso, os frameworks oferecem métodos já implementados para operações triviais, como salvar, alterar e remover, enquanto os métodos de consulta devem ser desenvolvidos pelos programadores, embora haja algumas consultas já predefinidas, como uma seleção por chave-primária ou que retorne todas as entidades de uma tabela. Entre os frameworks ORM mais famosos, temos o Hibernate, Eclipse Link e Top Link.

Um exemplo de classe de entidade pode ser visualizado na **Listagem 2**, que mostra o código da classe **Contato**. Assim, cada objeto referente a essa entidade representará uma linha na tabela *contatos* de um banco de dados.

Listagem 2. Código da classe/entidade Contato.

```
@Entity  
@Table(name = "CONTATOS")  
public class Contato implements Serializable {  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    @Column(name = "nome")  
    private String nome;  
    @Column(name = "sobrenome")  
    private String sobrenome;  
    @Column(name = "idade")  
    private Integer idade;  
    @Column(name = "sexo")  
    private String sexo;  
  
    // métodos getters/setters omitidos...  
}
```

Analisando esse código é possível observar o uso de algumas anotações da especificação JPA, as quais são encontradas no pacote **javax.persistence**. A seguir descrevemos o objetivo ou o papel de cada uma delas para o mapeamento:

- **@Entity** – determina que a classe representa uma entidade gerenciada pelo framework ORM;
- **@Table** – configura o nome da tabela no banco de dados;
- **@Id** – indica que o atributo **id** é a chave primária;
- **@GeneratedValue** – configura o tipo de geração de chaves utilizado pelo banco de dados;
- **@Column** – indica o nome da coluna no banco de dados que o atributo da classe representa.

Repositórios

Ao adotar o Spring-Data JPA, automaticamente faremos uso do padrão de projeto Repository, o qual já possui boa parte de sua implementação predefinida pelo Spring. Assim, fica a cargo do desenvolvedor criar uma interface que vai conter as assinaturas dos métodos de consulta ao banco de dados. Essa interface é denominada repositório e estará sempre vinculada a uma classe de entidade. Já os chamados métodos de escrita não precisam ser declarados, pois já são implementados pelo framework, como já mencionado.

Para transformar a interface que contém as assinaturas dos métodos de consulta em um repositório do Spring-Data JPA, é preciso estender a interface **org.springframework.data.jpa.repository.JpaRepository**. Essa interface será responsável por fornecer, por meio de herança, métodos de escrita e também algumas implementações básicas de métodos de consulta, como o **findAll()**, que localiza todos os registros de uma tabela, ou o **findOne()**, que localiza um único registro. Outra característica importante que será obtida por meio dessa interface é a transformação do repositório em um bean gerenciado pelo processo de injeção de dependências e inversão de controle do Spring Framework. Dessa forma, deixa de ser necessário o uso da anotação **@Repository**, destinada a esse propósito.

A **Listagem 3** apresenta um exemplo de repositório para a entidade **Contato**, exposta na **Listagem 2**.

Listagem 3. Repositório da entidade Contato.

```
public interface ContatoRepository extends JpaRepository<Contato, Long> {
    @Query("select c from Contato c where c.sexo like ?1")
    List<Contato> findBySexo(String sexo);

    @Query("select c from Contato c where c.idade >= 18")
    List<Contato> findByMaioridade();

    @Query("select c from Contato c where c.sexo like ?1 and c.idade >= 18")
    List<Contato> findBySexoAndMaioridade(String sexo);
}
```

Observe que em **ContatoRepository** existem três métodos de consulta, os quais usam a anotação **@Query** com uma instrução em linguagem JPQL. O método **findBySexo()**, por exemplo, tem um argumento do tipo **String** que deve receber um valor equivalente a **F** ou **M** para selecionar o gênero dos contatos a serem localizados. E por meio de alguns recursos do Spring-Data JPA,

o valor contido no parâmetro **sexo** será combinado à JPQL, substituindo a expressão **?1** automaticamente em tempo de execução. Na sequência, temos o método **findByMaioridade()**, o qual não possui nenhum argumento em sua assinatura, mas localiza no banco de dados os contados com maioridade, ou seja, idade maior ou igual a 18 anos. Como pode ser visto, esse critério foi fixado na instrução JPQL. Já no método **findBySexoAndMaioridade()**, há um argumento **String** para o tipo de gênero dos contatos e, na JPQL, temos o critério que vai selecionar os contatos por maioridade. Assim, serão retornados pela consulta todos os contatos que sejam do sexo feminino e maiores de 18 anos; ou do sexo masculino e maiores de 18 anos, dependendo do valor atribuído como critério ao parâmetro **sexo**.

Por fim, para especificar **ContatoRepository** como um repositório gerenciado pelo Spring-Data, foi estendida a interface **JpaRepository** com dois generics. O primeiro generic é **Contato**, o qual vai vincular esse repositório ao modelo ou à entidade **Contato**. Já o segundo tem como objetivo informar ao Spring-Data que o tipo da chave primária no mapeamento da classe de entidade foi definido como **Long**.

O padrão Specification no Spring-Data JPA

O padrão de projeto Specification foi elaborado para situações em que diferentes regras, usadas na seleção de um objeto, possam ser combinadas com o uso de elementos da lógica booleana. Por conta disso, o padrão vem sendo utilizado com mais frequência em operações relacionadas a banco de dados, possibilitando uma maneira mais concisa de expressar certos tipos de regras que são necessárias ao elaborar uma consulta. De modo geral, podemos entender que essas regras são formadas por elementos relacionados à construção da consulta, como a lista de argumentos de um método, os critérios e até mesmo os operadores lógicos. A partir disso, pode-se dizer que o objetivo básico desse padrão é separar cada uma dessas regras em um objeto próprio, de modo que elas possam ser combinadas a qualquer momento. A esses objetos damos o nome de predicado.

Nesse contexto, cada predicado é uma regra específica que vai representar uma parte da instrução que compõe a consulta. Assim, em um predicado existirá um parâmetro e um tipo de critério; por exemplo, para um parâmetro **nome**, poderíamos ter o critério **LIKE**. Enquanto outro predicado poderia ser formado por um parâmetro **idade** e o critério **equal**. Com isso, um método de consulta pode receber qualquer tipo de predicado, ou mesmo uma combinação de diferentes predicados obtida por uma lógica condicional. Em contrapartida, um método de consulta que não usa predicados como argumento estaria preso a um tipo de dado, como um objeto **Long**. Dessa forma, ele só poderá localizar os dados a partir de um parâmetro do tipo **Long** e recuperar os dados conforme sua lógica condicional, que também estaria fixada na consulta.

Para possibilitar esse diferencial, no Spring-Data JPA, a implementação do padrão Specification disponibiliza uma interface central, de mesmo nome do padrão, que contém apenas um

Spring Data e o padrão Specification: Simplifique a construção e o reuso de consultas

método: **toPredicate()**. Esse método, como podemos observar na **Listagem 4**, retorna um objeto do tipo **Predicate**, que exerce o papel de um predicado em uma consulta.

Listagem 4. Interface Specification no Spring-Data JPA.

```
public interface Specification<T> {  
    Predicate toPredicate(Root<T> var1, CriteriaQuery<?> var2, CriteriaBuilder var3);  
}
```

Analisando o **toPredicate()**, pode-se observar que sua lista de argumentos contém alguns dos objetos da API Criteria, como **Root**, **CriteriaQuery** e **CriteriaBuilder**. O leitor mais atento irá lembrar que todos esses argumentos e também o tipo de retorno (**Predicate**) foram abordados na **Listagem 1**, onde estão descritas as funções de cada um desses objetos.

Como o Spring-Data JPA já fornece uma pré-implementação de **Specification**, é preciso seguir algumas regras para habilitar seu uso. A primeira delas é estender no repositório a interface **org.springframework.data.jpa.repository.JpaSpecificationExecutor**, conforme o código da **Listagem 5**. Essa interface tem a responsabilidade de conectar o recurso **Specification** ao recurso **Repository** do Spring-Data JPA.

Listagem 5. Repositório habilitado para o uso de Specification.

```
public interface ContatoRepository extends  
    JpaRepository<Contato, Long>, JpaSpecificationExecutor<Contato> {  
  
    @Query("select c from Contato c where c.sexo like ?1")  
    List<Contato> findBySexo(String sexo);  
  
    @Query("select c from Contato c where c.idade >= 18")  
    List<Contato> findByMaioridade();  
  
    @Query("select c from Contato c where c.sexo like ?1 and c.idade >= 18")  
    List<Contato> findBySexoAndMaioridade(String sexo);  
}
```

Ao estender a interface **JpaSpecificationExecutor**, alguns métodos serão herdados e podem ser conferidos na **Listagem 6**. Veja que todos eles têm como argumento um objeto do tipo **Specification**. Isso porque esse objeto vai armazenar todas as regras da consulta para que ela possa ser executada pelo respectivo método que o recebeu.

Listagem 6. Código da interface JpaSpecificationExecutor.

```
1 public interface JpaSpecificationExecutor<T> {  
2     T findOne(Specification<T> var1);  
3     List<T> findAll(Specification<T> var1);  
4     Page<T> findAll(Specification<T> var1, Pageable var2);  
5     List<T> findAll(Specification<T> var1, Sort var2);  
6     long count(Specification<T> var1);  
7 }
```

A escolha do método a ser chamado para a execução da consulta é feita conforme o tipo de retorno desejado, que pode ser:

1. Um objeto único;
2. Uma lista de objetos;
3. Uma lista de objetos para consultas paginadas;
4. Uma lista de objetos com um tipo de ordenação definido pelo parâmetro **Sort**;
5. Um **long** com a quantidade de objetos que satisfazem as regras da consulta.

Especificando a entidade Contato

Agora que já conhecemos a interface **JpaSpecificationExecutor**, o próximo passo é definir os predicados. Segundo o padrão **Specification**, é necessário criar uma implementação para a interface **Specification** e seu método **toPredicate()**, ao qual será adicionada parte da regra da consulta, contendo os parâmetros e critérios necessários.

Porém, dessa forma teríamos que criar uma classe para cada novo método, já que cada classe deve ter no máximo uma implementação de **toPredicate()**. Para contornar esse detalhe, o ideal é desenvolver uma classe que represente uma entidade específica e adicionar a ela classes anônimas (veja o **BOX 2**) que representem cada um dos métodos de consulta que se deseja trabalhar. Um exemplo disso pode ser visto na **Listagem 7**, onde temos a classe **ContatoSpecification** e o método **maioridade()**.

Conforme essa listagem, ao instanciar a interface **Specification** é criada uma classe anônima e o método **toPredicate()** é implementado.

BOX 2. Classe Anônima

Classe anônima é uma classe sem nome, por isso chamada de anônima, que não é declarada explicitamente no código com a palavra reservada **class**. Na verdade, esse tipo de classe é definido dentro de um método ou mesmo como um argumento de um método.

A classe anônima pode ser usada como uma forma diferente de implementar uma interface, o que a transformaria em uma subclasse. Porém, o acesso a esse tipo de classe é restrito ao método em que ela é definida. Outra característica é que uma classe anônima não pode implementar mais de uma interface e nem estender uma classe ao mesmo tempo em que implementa uma interface.



Listagem 7. Busca maioridade por especificação.

```
public class ContatoSpecification {  
  
    public static Specification<Contato> maioridade() {  
        return new Specification<Contato>() {  
            @Override  
            public Predicate toPredicate(Root<Contato> root,  
                CriteriaQuery<?> query, CriteriaBuilder builder) {  
  
                return builder.greaterThanOrEqualTo(root.get("idade"), 18);  
            }  
        };  
    }  
}
```

Usamos, então, o objeto **criteriaBuilder** para acessar o método **greaterThanOrEqualTo()**. Esse método vai testar se um valor é maior ou igual a outro; neste caso, a uma idade de 18 anos. Como parâmetros do método **greaterThanOrEqualTo()**, devemos informar o atributo da classe **Contato** que será usado como parte do critério dessa consulta (esse atributo é adicionado por meio do método **get()** do objeto **root**) e o valor que será avaliado pelo critério.

Note como ficou mais simples trabalhar com os objetos da API Criteria, em relação ao código da **Listagem 1**. Dessa vez, basicamente em uma linha definimos a consulta.

Neste momento é importante citar que a partir do Java 8 as classes anônimas podem ser substituídas por um novo recurso, denominado Expressões Lambda (veja o **BOX 3**). Assim, com o intuito de explorar essa nova opção, os próximos exemplos de métodos que serão adicionados à classe **ContatoSpecification** utilizam Lambda, como mostra o código da **Listagem 8**.

BOX 3. Expressões Lambda

Expressões lambda são uma funcionalidade comum em muitas linguagens de programação, em particular as que seguem o paradigma de Programação Funcional, e recentemente foram introduzidas no Java através do Projeto Lambda, conforme a JSR 335. Em resumo, com o uso desse recurso há a possibilidade de se trabalhar com expressões que não requerem a declaração de um nome, de um tipo de retorno ou mesmo um modificador de acesso, o que viabiliza uma redução na quantidade de código necessária para a escrita de algumas funções, como, por exemplo, as classes anônimas, bastante utilizadas em listeners e threads.

Listagem 8. Consulta por tipo de sexo via especificação.

```
public static Specification<Contato> sexo(String sexo) {  
    return (root, query, builder) -> builder.like(root.get("sexo"), sexo);  
}
```

Com relação à essa listagem, observe que o método **sexo()** possui como argumento um atributo do tipo **String** para definir o gênero dos resultados a serem retornados. Para isso, usamos o método **like()**, de **CriteriaBuilder**, para criar o critério da consulta. Esse método realiza uma operação idêntica à instrução **LIKE** do SQL e, assim, vai retornar todas as entidades do sexo feminino ou masculino, dependendo do parâmetro informado.

Agora é possível demonstrar como implementaríamos uma consulta do tipo **findBySexoAndMaioridade()** sem a necessidade de criar um terceiro método de consulta no repositório. Para isso, suponha que em seu projeto haja uma classe do tipo **ContatoService**, a qual tem acesso ao repositório **ContatoRepository**, via injeção de dependências, como apresentado na **Listagem 9**.

Para ser um bean gerenciado pelo controle de injeção de dependências do Spring Framework, a classe **ContatoService** é anotada com **@Service**. E para injetar nessa classe o objeto **ContatoRepository**, declaramos a anotação **@Autowired**. Assim, vamos ter acesso a qualquer método da interface **JpaSpecificationExecutor** disponível para acesso no service.

Listagem 9. Método de consulta por sexo e maioridade com Specification.

```
@Service  
public class ContatoService {  
  
    @Autowired  
    private ContatoRepository repository;  
  
    public List<Contato> findBySexoAndMaioridade(String sexo) {  
  
        return repository.findAll(  
            Specifications  
                .where(ContatoSpecification.sexo(sexo))  
                .and(ContatoSpecification.maioridade())  
        );  
    }  
}
```

Conforme o código apresentado, veja que o corpo do método **findBySexoAndMaioridade()** possui uma chamada a **findAll()** de **JpaSpecificationExecutor**, via objeto **repository**. Por isso, atribuímos a ele um objeto do tipo **Specifications**, que é uma classe do Spring-Data JPA que implementa a interface **Specification** e nos fornece alguns métodos importantes, como o **where()** e o **and()**, que representam a lógica condicional da consulta.

Os métodos de **Specifications** são usados com o objetivo de receber os predicados, de **ContatoSpecification**, na forma de parâmetros. Deste modo, as regras da consulta passam a ser combinadas uma a uma. Como há o interesse em mesclar os critérios sexo e maioridade, foi atribuído ao **where()** o método **sexo()**, de **ContatoSpecification**, e no **and()**, o método **maioridade()**.

Como você pode verificar, a classe **Specifications** oferece o chamado *glue-code* para combinar uma cadeia de métodos com diferentes predicados. Então, o Spring-Data JPA, junto ao padrão Specification, combina essas regras como se existisse uma única instrução JPQL, conforme a seguir:

```
select c from Contato c where c.sexo like ?1 and c.idade >= 18
```

A classe **Specifications** está disponível no pacote **org.springframework.data.jpa.domain** e sua lista de métodos é composta por:
• **where()** – usado para definir o primeiro predicado da consulta;

- **and()** – usado após o **where()**, **and()**, **or()** ou **not()** e tem função idêntica ao **AND** do SQL;
- **or()** – usado após o **where()**, **and()**, **or()** ou **not()** e tem função idêntica ao **OR** do SQL;
- **not()** – nega o predicado da especificação e é usado como o primeiro método da consulta.

Normalmente, o ideal é adicionar na classe que contém os predicados, como a **ContatoSpecification**, os métodos de consulta referentes a todos os atributos da classe de entidade. Deste modo, é possível combinar os predicados, referentes a cada atributo, entre si. Com base nisso, vamos incluir na classe **ContatoSpecification** mais alguns métodos, descritos na **Listagem 10**.

Listagem 10. Especificações para consultas por nome e por sobrenome.

```
public static Specification<Contato> nome(String nome) {  
    return (root, query, builder) -> builder.like(root.get("nome"), nome);  
}  
  
public static Specification<Contato> sobrenome(String sobrenome) {  
    return (root, query, builder) -> builder.like(root.get("sobrenome"), sobrenome);  
}
```

Veja que os novos métodos incluídos em **ContatoSpecification** são consultas bastante simples, que localizam resultados individualmente por atributos como **nome** e **sobrenome**. Com esses métodos, mais o **sexo()** e **maioridade()**, é possível realizar combinações diversas, via **Specifications**, para a obtenção de diferentes resultados.

A **Listagem 11** apresenta um método que poderia ser adicionado em **ContatoService** para localizar um único contato usando o nome e sobrenome como critérios. Assim, o método indicado para uso, entre aqueles da interface **JpaSpecificationExecutor**, é o **findOne()**.

Listagem 11. Busca de contato por nome e sobrenome.

```
public Contato findByNomeCompleto(String nome, String sobrenome) {  
    return repository.findOne(  
        Specifications  
            .where(ContatoSpecification.nome(nome))  
            .and(ContatoSpecification.sobrenome(sobrenome))  
    );  
}
```

Como podemos notar, o método **findByNomeCompleto()** tem dois argumentos do tipo **String**: **nome** e **sobrenome**. Então, **nome** e **sobrenome** são usados como parâmetros para complementar as regras dos critérios definidos nos métodos **nome()** e **sobrenome()** da classe **ContatoSpecification**. Por fim, devemos criar a lógica condicional, onde **nome()** é adicionado ao **where()**, de **Specifications**, e **sobrenome()** é adicionado ao **and()**. Com isso, os predicados são combinados pela instrução **AND**, que retorna apenas o contato que possui o nome e o sobrenome referentes aos argumentos do método **findByNomeCompleto()**.

Conforme já mencionado, a interface **JpaSpecificationExecutor** fornece, também, o método **count()**. Esse método pode ser usado, por exemplo, para buscar a quantidade de entidades no banco de dados que possuam uma determinada idade, ou um determinado gênero. A **Listagem 12** apresenta uma consulta que retornaria o total de contatos por um gênero qualquer.

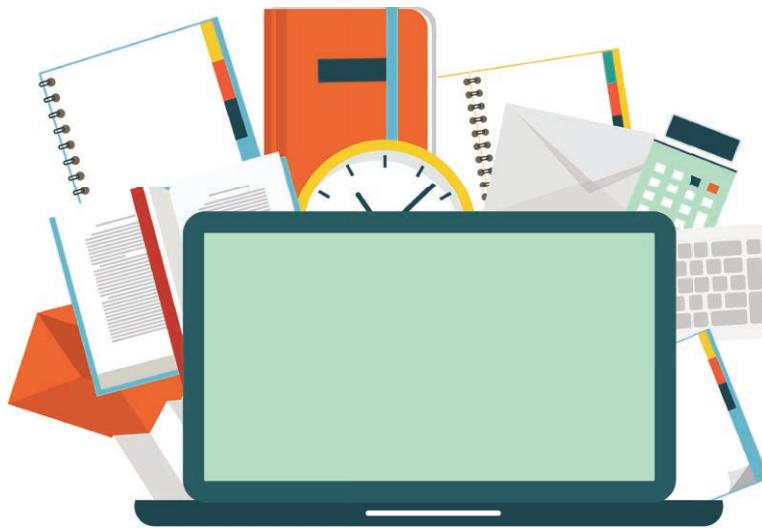
Listagem 12. Busca a quantidade total de contatos por gênero.

```
public long countByGenero(String sexo) {  
    return repository.count(ContatoSpecification.sexo(sexo));  
}
```

Note que **countByGenero()** recebe como parâmetro o gênero do contato ("M" ou "F"). Em seguida, usando o método **sexo()**, da classe **ContatoSpecification**, passamos a trabalhar com um predicado que tem como critério selecionar todos os contatos referentes ao sexo recebido como parâmetro. Já o método **count()**, da interface **JpaSpecificationExecutor**, recebe esse predicado e retorna a quantidade total de contatos que foram obtidos conforme o critério estabelecido.

Concluído esse exemplo, vejamos outra possibilidade de consulta. Para isso, suponha que você queira obter como retorno uma lista de contatos que seja de um determinado gênero e menores de 18 anos. Se estivéssemos trabalhando sem Specification, teríamos que criar um novo método no repositório com essas regras. Porém, como estamos usando Specification, e na classe **ContatoSpecification** já existe o método **maioridade()**, é preciso apenas negá-lo no método que será adicionado em **ContatoService**, como no código exemplificado na **Listagem 13**.

O método **findBySexoAndMenoridade()** tem como parâmetro um gênero, que pode ser "M" ou "F". Já o método **sexo()** recebe esse parâmetro e retorna um predicado, o qual será atribuído ao método **and()** da classe **Specifications**. Antes disto, no entanto,



foi declarado o método **not()**, também de **Specifications**, para negar os resultados de uma consulta por maioridade. Deste modo, **maioridade()**, ao invés selecionar os contados com idade maior ou igual a 18 anos, vai selecionar os contados com idade menor que 18 anos. Essa seleção é combinada, via condicional **AND**, aos contatos referentes ao tipo de sexo. Assim, se o gênero informado é “M”, todos os contatos de menoridade e gênero “M” serão retornados pelo método **findAll()**. Se o critério for “F”, serão retornados todos os contatos de gênero feminino e de menoridade.

Listagem 13. Consulta com o método de negação **not()**.

```
public List<Contato> findBySexoAndMenoridade(String sexo) {
    return repository.findAll(
        Specifications
            .not(ContatoSpecification.maioridade())
            .and(ContatoSpecification.sexo(sexo))
    );
}
```

Algo importante a ressaltar aqui é o quanto o Spring favorece o uso do padrão Specification. Uma forma de se notar esse favorecimento é quando utilizamos, por exemplo, os métodos **findOne()**, **findAll()** e **count()** da interface **JpaSpecificationExecutor**, assim como os métodos da classe **Specifications** (**where()**, **and()**, **or()** e **not()**). Se o padrão tivesse que ser programado em sua totalidade pelo desenvolvedor, a lógica de todos esses métodos, classes e interfaces também deveria ser criada por ele, o que tornaria a tarefa muito mais difícil.

Outro fato que deve ser levado em consideração sobre as consultas via padrão Specification é a possibilidade de escrever opções mais complexas do que as abordadas até aqui. Se você domina ou tem conhecimento mais avançado na API Criteria, pode usar os objetos da lista de argumentos do método **toPredicate()** para implementá-las. Por exemplo, veja na **Listagem 14** o método **idade()**. Esse método usa o objeto da interface **CriteriaBuilder** para localizar no banco de dados os contatos que possuem uma determinada idade. Porém, note que o objeto **criteriaQuery** foi utilizado com o **orderBy()** para criar uma ordenação do resultado conforme os atributos **nome** e **sobrenome**. Assim, se uma consulta for realizada por uma idade que possua vários contatos como retorno, eles serão ordenados primeiramente por nome, de forma ascendente, e se houver nomes idênticos, a ordenação será por sobrenome, também com classificação ascendente.

Observe que o exemplo apresentado no método **idade()** não é tão complexo, mas mostra que os métodos que formam os predicados podem conter mais de um critério. Deste modo, verifica-se que a complexidade da consulta é algo variável e está relacionada ao que cada projeto necessita.

Com o intuito de possibilitar queries mais complexas, a interface **QueryBuilder** fornece mais alguns métodos, além do **orderBy()**, por exemplo:

- **groupBy()** – tem a função de agrupar os resultados por determinados campos. É idêntico ao **GROUP BY** do SQL;

- **having()** – especifica um critério de pesquisa para um grupo ou uma agregação. Função idêntica ao **HAVING** do SQL;
- **distinct()** – elimina os resultados redundantes ou duplicados de uma consulta. Sua função é a mesma que o **DISTINCT** do SQL.

Listagem 14. Explorando a API Criteria.

```
public class ContatoSpecification {
    // métodos já abordados foram omitidos...

    public static Specification<Contato> idade(Integer idade) {
        return (root, query, builder) -> {
            query.orderBy(
                builder.asc(root.get("nome")),
                builder.asc(root.get("sobrenome"))
            );
            return builder.equal(root.get("idade"), idade);
        };
    }
}
```

Desafio ao leitor

Levando em consideração os exemplos apresentados até aqui, vamos adicionar à entidade **Contato** um atributo do tipo **java.util.Date**, conforme a **Listagem 15**, para representar a data de cadastro dos contados.

Listagem 15. Entidade Contato com atributo Date.

```
@Entity
@Table(name = "CONTATOS")
public class Contato implements Serializable {

    // demais atributos e métodos omitidos...

    @Temporal(TemporalType.DATE)
    @Column(name = "data_cadastro")
    private Date dtCadastro;
}
```

A partir dessa nova informação será possível localizar contatos por meio das datas em que eles foram cadastrados.

Com base nisso e no conteúdo exposto neste artigo, elabore duas consultas conforme os seguintes requisitos:

1. Buscar no banco de dados a quantidade total de contatos cadastrados em um determinado ano, o qual será informado como parâmetro;
2. Localizar uma lista de contatos pelo campo data de cadastro usando como parâmetros o mês e o dia.

Conforme os requisitos citados para cada consulta, desenvolva os predicados necessários na classe **ContatoSpecification**, e, em seguida, em **ContatoService**, defina dois novos métodos para executar cada uma das consultas solicitadas.

Spring Data e o padrão Specification: Simplifique a construção e o reuso de consultas

Respostas do desafio

De acordo com o desafio proposto, o passo inicial é definir os predicados que serão empregados nas consultas. Como uma data é formada por três elementos (ano, mês e dia), e cada consulta solicitada usa diferentes elementos, o mais indicado neste caso é criar um predicado para cada um deles, conforme a **Listagem 16**.

Listagem 16. Predicados para datas.

```
public static Specification<Contato> byAno(Integer ano) {
    return (root, query, builder) ->
        builder.equal(
            builder.function("year", Integer.class, root.get("dtCadastro")),
            ano
        );
}

public static Specification<Contato> byMes(Integer mes) {
    return (root, query, builder) ->
        builder.equal(
            builder.function("month", Integer.class, root.get("dtCadastro")),
            mes
        );
}

public static Specification<Contato> byDia(Integer dia) {
    return (root, query, builder) ->
        builder.equal(
            builder.function("day", Integer.class, root.get("dtCadastro")),
            dia
        );
}
```

Agora que os predicados estão especificados, vamos analisar como criar a primeira consulta, aquela em que devemos buscar a quantidade de contatos a partir de um determinado ano. Para isso, na classe **ContatoService** adicionamos o método **countByAno()** — conforme a **Listagem 17** —, o qual tem como argumento a variável **ano**, que vai conter o valor referente ao ano e que será utilizada como parâmetro na consulta por meio do predicado **byAno()**. Por fim, invocamos o método indicado para a consulta; neste caso, o **count()**, acessível por meio do objeto **repository**.

Listagem 17. Método que retorna a quantidade de contatos cadastrados por ano.

```
@Service
public class ContatoService {

    // demais métodos e atributos omitidos...

    public long countByAno(int ano) {
        return repository.count(ContatoSpecification.byAno(ano));
    }
}
```

Uma consulta similar em SQL pode ser escrita conforme a seguir:

```
select count(*) from Contatos where extract(year from data_cadastro) = 2015
```

Veja que na operação do **select** há uma função chamada **extract**, a qual tem como objetivo isolar os elementos da data e usá-los separadamente como critério. Neste caso, o elemento em questão é o ano (**year**) da (**from**) coluna **data_cadastro**. O elemento extraído pela função é então comparado ao valor do parâmetro (**2015**). Esse processo é o mesmo usado em todos os predicados da **Listagem 16**, porém, nos predicados utilizamos os métodos da API Criteria, como o **builder.function()**, que representa a função **extract** do SQL, e **builder.equal()**, que realiza a comparação de igualdade entre o elemento da data e o parâmetro.

Já na segunda consulta do desafio precisamos trabalhar com dois parâmetros, o mês e o dia, e retornar uma lista de contatos. Como não será necessário criar novos predicados para essa consulta, pois isso já foi feito na **Listagem 16**, vamos adicionar em **ContatoService** o método **findByMesAndDia()**, conforme a **Listagem 18**.

Listagem 18. Método que retorna uma lista de contatos por mês e dia.

```
@Service
public class ContatoService {

    // demais métodos e atributos omitidos...

    public List<Contato> findByMesAndDia(int mes, int dia) {
        return repository.findAll(
            Specifications
                .where(ContatoSpecification.byMes(mes))
                .and(ContatoSpecification.byDia(dia))
        );
    }
}
```

Analizando **findByMesAndDia()**, veja que ele possui dois argumentos, um para o mês e outro para o dia. No corpo do método, pelo objeto **repository**, invocamos **findAll()**, que vai receber como parâmetro os predicados requeridos. Como temos dois predicados, vamos usar os métodos da classe **Specifications** nesta operação. Assim, no método **where()** é adicionado o predicado relativo ao mês, e em **and()** inserimos o predicado para o dia.

Neste momento podemos avaliar a importância de separar cada elemento da data em um predicado diferente.



Com essa prática, viabilizamos a construção de qualquer consulta que precise conter um ou mais elementos de uma data. Nos desafios, solicitamos apenas duas, mas com esses três predicados podemos implementar muitas outras opções que venham a ser necessárias.

Como vimos, o padrão Specification possibilita uma abordagem diferente na construção de consultas, e com o auxílio do Spring-Data JPA, esse processo se torna ainda mais simples, devido aos recursos oferecidos pelo framework. Isso pode ser observado a partir dos exemplos de código apresentados neste artigo, onde expomos consultas consideradas básicas, mas que demonstram as vantagens dessa nova opção.

Como um importante complemento, sugerimos um estudo mais aprofundado sobre a API Criteria, da especificação JPA, através da documentação oficial da Oracle (veja o endereço indicado na seção **Links**), pois é possível, e muitas vezes será necessário, criar consultas mais complexas.

Por fim, saiba que embora o texto tenha abordado o padrão Specification já desenvolvido pelo Spring-Data, é perfeitamente possível, se você não for um usuário de soluções Spring, trabalhar com esses conceitos em um projeto que adote apenas o JPA na camada de persistência. Para isso, o indicado é pesquisar mais à fundo, no livro Domain Driven Design, as regras necessárias para implementar o padrão de projeto Specification.

Autor



Marcio Ballem de Souza

@mballem – www.mballem.com

Bacharel em Sistemas de Informação e especialista em aplicações para a web pela FURG. Tem experiência em desenvolvimento Java e certificação OCPJP 6. É autor do livro Desvendando o MongoDB, do Mongo Shell ao Java Driver.



Links:

Spring Data JPA – Guia de Referência.

<http://docs.spring.io/spring-data/jpa/docs/1.9.4.RELEASE/reference/html/>

Spring Data JPA – Specifications.

<https://docs.spring.io/spring-data/jpa/docs/1.9.4.RELEASE/reference/html/#specifications>

The Java Tutorials – Lambda Expressions.

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

Using the Criteria API to Create Queries.

<https://docs.oracle.com/javaee/6/tutorial/doc/gjtv.html>



Java EE CDI: Como e quando utilizar Interceptors e Decorators

Aprenda neste artigo como utilizar Interceptors e Decorators do CDI

CDI é a especificação do Java EE para lidar com contextos (diferentes estados da aplicação e seus objetos relacionados) e injeção de dependências. Essa especificação se integra tão bem e de forma tão natural ao restante da plataforma Java que, muitas vezes, seus recursos são utilizados mesmo quando o desenvolvedor não conhece os detalhes da especificação.

Quando o assunto é CDI, para aqueles que conhecem apenas o básico da especificação, o primeiro assunto que vem à cabeça são os recursos relacionados à injeção de dependências, como: a injeção de dependência em si, por meio da anotação `@Inject`; a produção de beans, por meio da anotação `@Produces`; e a definição de escopo dos beans por meio da anotação `@SessionScoped`. CDI, no entanto, oferece muitos outros recursos que podem melhorar ainda mais a arquitetura da aplicação, tanto em termos de acoplamento, quanto em termos de coesão e granularidade, de forma a evitar que o código se torne complexo e custoso de se manter.

Entre os recursos que muitas vezes passam despercebidos pela maioria dos desenvolvedores estão os interceptors e os decorators. Com base nisso, nos próximos tópicos analisaremos esses recursos através de diferentes cenários nos quais eles são indicados.

Interceptors

Interceptors, ou interceptadores, em CDI, são filtros que permitem executar operações antes ou depois das chamadas a métodos e que podem, inclusive, substituir

Fique por dentro

Quando se fala em CDI, na maioria das vezes são enfatizados apenas os aspectos relacionados à injeção de dependências, visto que essa especificação utiliza o termo dependency injection em seu nome. Entretanto, CDI está repleta de recursos e vai muito além da injeção de dependências, podendo facilitar e muito a implementação dos mais variados requisitos de um software. Os recursos Interceptors e Decorators, por exemplo, temas deste artigo, são muito úteis e importantes na criação de soluções robustas e aderentes às práticas mais modernas de arquitetura de software ao proporcionar mecanismos para extensão da funcionalidade da aplicação sem alterações bruscas no código fonte, tornando o desenvolvimento mais rápido e confiável.

a chamada ao método original. Os interceptadores se baseiam em anotações, métodos e classes para definir ou não a sua execução, e são úteis em muitos cenários, como incluir um cabeçalho padrão em todas as respostas HTTP. Esse mecanismo simplifica a implementação de pequenos trechos de código em diferentes pontos da aplicação com o mínimo de alterações.

Como será possível notar a partir de agora, existem vários tipos de interceptadores, entre eles: `@AroundInvoke`, que intercepta métodos determinados pelo desenvolvedor por meio de anotações; `@AroundTimeout`, que intercepta métodos invocados pelo serviço de *timer* do Java EE (veja a seção **Links** para saber mais sobre o serviço de *timer*); e interceptadores de ciclo de vida, que interceptam a construção e destruição de beans, como o `@AroundTimeout`.

Nota

Além dos Interceptors da CDI, também existem os Interceptors da API de Servlets. Ambos seguem o mesmo conceito, entretanto, os Interceptors da CDI interceptam chamadas a métodos, enquanto os Interceptors da API de Servlets interceptam requisições HTTP.

A **Figura 1** ilustra o funcionamento de alguns interceptadores em uma invocação a um método. Quando se habilita a CDI no projeto, a chamada a qualquer método sempre será executada em um contexto CDI. Dessa forma, caso o método invocado ou a classe na qual ele foi implementado sejam anotados com um interceptor, esse interceptor será disparado, encapsulando e gerenciando a invocação do método. Esse mecanismo é excelente para definir requisitos transversais, como a implementação de logs ou de segurança em toda a aplicação. Para saber mais sobre requisitos transversais, veja o **BOX 1**.

BOX 1. Requisitos transversais

Requisito transversal é um termo que vem, pouco a pouco, substituindo o termo requisito não funcional. Um requisito transversal representa comportamentos transversais do sistema, ou seja, comportamentos que deveriam ser implementados em todos os módulos, definindo restrições, atributos de qualidade e/ou a forma de operação.

Exemplos de requisitos transversais incluem segurança, desempenho e escalabilidade. Eles dizem respeito a toda a aplicação, por exemplo: não faz sentido definir um controle de acesso por meio de uma tela de login sem um mecanismo que controle o acesso a todas as telas do sistema, ou ainda definir requisitos de escalabilidade ou desempenho sem considerar a aplicação como um todo.

Tendo em vista a importância dos requisitos transversais, muitas abordagens surgiram para melhorar a implementação dos mesmos pela aplicação. O esforço mais conhecido nessa área é a programação orientada a aspectos, implementada por frameworks como AspectJ e Spring AOP, que, inclusive, foram grandes responsáveis pela popularização do termo requisitos transversais. Além disso, mecanismos como os Interceptors na API de Servlet e na CDI oferecem boas alternativas para a implementação de requisitos transversais em aplicações Java.

Interceptador @AroundInvoke

Esta opção atua interceptando apenas alguns ou todos os métodos de uma classe. Um exemplo de uso de `@AroundInvoke` seria a adição de um mecanismo de log na aplicação, de forma a gerar log de todas (ou quase todas) as operações executadas.

A princípio, a implementação de um log padrão em todos os métodos parece guiar para uma solução bastante trabalhosa,

que consiste em percorrer todo o código fonte e, em cada um dos métodos, adicionar as linhas de log necessárias. Com o uso de interceptadores, esse requisito pode ser implementado de forma rápida e sem grande impacto no código. A **Listagem 1** mostra um interceptor que adiciona o mecanismo de log aos métodos ou classes que receberem tal anotação.

A definição de um interceptor é bastante simples. Primeiro, é necessário utilizar a anotação `@Interceptor` sobre a classe, indicando que a mesma será um interceptor. Segundo, deve-se utilizar a anotação `@Priority`, para definir a prioridade do interceptor quando houver mais de um elegível para interceptação; no caso, é utilizada uma constante de prioridade padrão, `Priority.APPLICATION`, que equivale ao inteiro 2000. Além da `Priority.APPLICATION`, há, ainda, `Priority.LIBRARY_AFTER`, `Priority.LIBRARY_BEFORE`, `Priority.PLATFORM_AFTER` e `Priority.PLATFORM_BEFORE`. Quanto menor a prioridade, mais cedo o interceptor será executado em uma cadeia de interceptadores. A definição da prioridade via anotação é opcional, entretanto, caso não seja definida dessa forma, é preciso registrar o interceptor no arquivo `beans.xml` de modo que a implementação de CDI saiba da prioridade por ordem de registro nesse arquivo. Terceiro, ter um método anotado com `@AroundInvoke`, para defini-lo como um método interceptor.

O método definido pela anotação `@AroundInvoke` se interpõe entre a requisição e o método alvo, tomando o controle e decidindo como completar a requisição. De acordo com o código da **Listagem 1**, antes do método alvo ser executado, é adicionado um trecho de log avisando que o interceptor está executando. Em seguida, por meio do `InvocationContext`, são obtidas informações sobre a chamada de método em questão, como o nome do método chamado, o objeto e os parâmetros. Após o interceptor registrar as informações de log, ele invoca o método alvo por meio da chamada `context.proceed()`, que executa e obtém o retorno do método alvo. Se a chamada `context.proceed()` não acontecer, o método alvo não será executado. É nesse ponto que é possível definir se a chamada a esse método será concretizada ou se vamos executar outro em seu lugar. Em nosso exemplo, depois da execução do método alvo, são feitos mais alguns registros de log e o método interceptor retorna o resultado da invocação a quem originou a chamada.

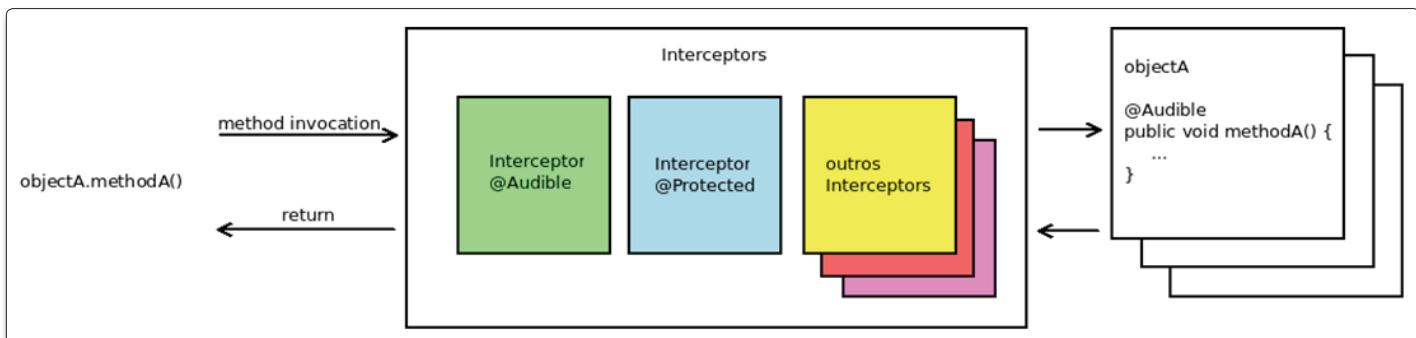


Figura 1. Funcionamento dos Interceptors da CDI

Java EE CDI: Como e quando utilizar Interceptors e Decorators

Listagem 1. Código do interceptador que adiciona mecanismo de log transversal.

```
package com.gabrielamorim.cdi.interceptors;

import java.lang.reflect.Method;
import java.util.logging.Logger;

import javax.annotation.Priority;
import javax.inject.Inject;
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Log @Interceptor @Priority(Interceptor.Priority.APPLICATION)
public class LogInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    public Object log(InvocationContext context) throws Exception {
        logger.info("Interceptor before the execution");

        Method method = context.getMethod();
        Object target = context.getTarget();
        Object[] params = context.getParameters();

        logger.info(String.format("Interceptor - method: '%s'"
                + "from this object: '%s'"
                + "with the following parameters: '%s'",
                method,
                target,
                params));

        logger.info("Interceptor calling original method");
        Object object = context.proceed();

        logger.info("Interceptor after the execution");

        return object;
    }
}
```

Para utilizar o interceptador **LogInterceptor** de forma a interceptar chamadas de métodos em determinados objetos, é preciso fazer a ligação do mesmo às classes em que ele deve atuar, o que é feito por meio dos *interceptor bindings*, isto é, anotações colocadas no interceptor e nas classes ou métodos que serão interceptados.

O exemplo da **Listagem 1** utiliza a anotação `@Log` como *interceptor binding*, definida na **Listagem 2**. Essa anotação serve para vincular objetos da classe interceptada ao interceptador. Para isso, a anotação precisa declarar ser um *interceptor binding* por meio da anotação `@InterceptorBinding`. Note, ainda, que ela é anotada com `@Inherited`, para que as classes que estenderem classes anotadas com `@Log` herdem o mecanismo de log.

Listagem 2. Definição da anotação de interceptor binding.

```
package com.gabrielamorim.cdi.interceptors;

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

import javax.interceptor.InterceptorBinding;

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

Tendo a classe interceptadora e a anotação de vinculação, qualquer classe ou método anotados com `@Log` serão interceptados por **LogInterceptor**. O mais interessante nessa abordagem é que, a partir de agora, é possível adicionar funcionalidades a diversos pontos da aplicação apenas adicionando uma anotação. Assim, todas as classes da aplicação poderiam ser interceptadas pelo **LogInterceptor** e passar a registrar o log com uma alteração mínima, a adição de uma anotação.

A **Listagem 3** mostra uma servlet anotada com `@Log`. Dessa forma, todo método invocado nessa servlet será interceptado e registrado nos logs. O resultado de uma requisição nesse caso produz os registros apresentados na **Listagem 4**.

Observe que são quatro registros de log adicionados após uma requisição, exatamente o número de logs colocados no método `@AroundInvoke` do interceptador. O primeiro registro trata-se de um ponto antes da execução do método, enquanto o segundo mostra informações sobre o método que está sendo invocado que, neste caso, é o `HttpServlet.service()`. O terceiro registro foi colocado imediatamente antes da invocação do método alvo original. Por fim, o quarto registro é criado após a execução do método alvo, mostrando que a chamada a ele aconteceu sem problemas.



Listagem 3. Exemplo de uso da anotação @Log para interceptar a execução de métodos dessa servlet.

```
package com.gabrielamorim.cdi.servlet;

//parte dos imports omitidos

import com.gabrielamorim.cdi.interceptors.Log;

@WebServlet("/Servlet") @Log
public class Servlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Servlet() {
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().append("Served at:").append(request.getContextPath());
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Listagem 4. Resultado em log da implementação do requisito transversal.

```
113:39:55,827 INFO [com.gabrielamorim.cdi.interceptors.LogInterceptor]
(default task-8) Interceptor before the execution
13:39:55,828 INFO [com.gabrielamorim.cdi.interceptors.LogInterceptor]
(default task-8) Interceptor - method:'public void javax.servlet.http.HttpServlet
.service(javax.servlet.ServletRequest,javax.servlet.ServletResponse) throws javax.
servlet.ServletException,java.io.IOException' from this object:'com.gabrielamorim.
cdi.servlet.Servlet$Proxy$_$_WeldSubclass@349ec39a' with the following par-
ameters:[Ljava.lang.Object;@3d3e9a4d'
13:39:55,828 INFO [com.gabrielamorim.cdi.interceptors.LogInterceptor] (default
task-8) Interceptor calling original method
13:39:55,828 INFO [com.gabrielamorim.cdi.interceptors.LogInterceptor] (default
task-8) Interceptor after the execution
```

Interceptador @AroundTimeout

Já os interceptadores **@AroundTimeout** trazem um conceito bastante simples. Basicamente, eles são utilizados para interceptar a execução dos métodos invocados pelo serviço de *timer* do Java EE, aqueles métodos anotados com **@Timeout** ou **@Schedule**.

A **Listagem 5** mostra a definição de um interceptor **@AroundTimeout**. Note que ela é semelhante à dos interceptadores **@AroundInvoke**. A diferença está na anotação utilizada no método **timer()**, que no caso foi anotado com **@AroundTimeout**.

A **Listagem 6** mostra um exemplo de uso do **TimerInterceptor**. Essa implementação possui um método anotado com **@Schedule**. Dessa forma, um timer é criado e disparado de acordo com o tempo configurado; nesse caso, a cada minuto. Além disso, uma mudança substancial em relação ao exemplo anterior está na forma de ligar o método interceptador à classe interceptada. Essa ligação não foi feita com um *interceptor binding*, como no exemplo anterior, mas sim definindo uma classe interceptadora na classe interceptada por meio de **@Interceptors**.

O resultado do *timer* configurado pela classe **TimerSchedule** e interceptado pela classe **TimerInterceptor**, em três minutos de execução, é apresentado na **Listagem 7**.

Listagem 5. Exemplo de interceptador **@AroundTimeout**. Intercepta métodos do serviço de timer.

```
package com.gabrielamorim.cdi.interceptors;

import javax.interceptor.AroundTimeout;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
public class TimerInterceptor {

    @Inject
    private Logger logger;

    @AroundTimeout
    public Object timer(InvocationContext context) throws Exception {
        logger.info("Interceptor before timeout method");
        Object retorno = context.proceed();
        logger.info("Interceptor after timeout method");
        return retorno;
    }
}
```

Listagem 6. Utilização do interceptador **TimerInterceptor**.

```
package com.gabrielamorim.cdi.timer;

import javax.ejb.Schedule;
import javax.inject.Inject;
import javax.interceptor.Interceptors;

import com.gabrielamorim.cdi.interceptors.TimerInterceptor;

@Stateless @Interceptors({TimerInterceptor.class})
public class TimerSchedule {

    @Inject
    private Logger logger;

    @Schedule(hour="*", minute="*")
    public void scheduledMethod() {
        logger.info("TimerSchedule.scheduledMethod()");
    }
}
```

Listagem 7. Resultado em log do interceptador **@AroundTimeout**.

```
11:53:00,002 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 7) Interceptor before timeout method
11:53:00,002 INFO [com.gabrielamorim.cdi.timer.TimerSchedule] (EJB default - 7)
TimerSchedule.scheduledMethod()
11:53:00,002 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 7) Interceptor after timeout method
11:54:00,002 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 1) Interceptor before timeout method
11:54:00,002 INFO [com.gabrielamorim.cdi.timer.TimerSchedule] (EJB default - 1)
TimerSchedule.scheduledMethod()
11:54:00,002 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 1) Interceptor after timeout method
11:55:00,001 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 8) Interceptor before timeout method
11:55:00,001 INFO [com.gabrielamorim.cdi.timer.TimerSchedule] (EJB default - 8)
TimerSchedule.scheduledMethod()
11:55:00,002 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 8) Interceptor after timeout method
```

Note que esses registros coincidem com os logs colocados no interceptor e na classe interceptada, mostrando a ordem exata em que cada método é chamado e executado. O método **TimerSchedule.scheduledMethod()**, por exemplo, é invocado a cada minuto pelo *timer*. Como essa classe possui uma classe configurada para interceptar métodos invocados pelo *timer*, o método **TimerInterceptor.timer()** é invocado e assume o controle da execução do método invocado inicialmente. Em seguida, **TimerInterceptor.timer()** registra um log antes de invocar o método alvo, chama esse método com **InvocationContext.proceed()** e registra um log após a finalização dele.

Interceptadores de ciclo de vida

Além dos interceptadores **@AroundInvoke** e **@AroundTimeout**, existem também os interceptadores de ciclo de vida: **@PostConstruct**, **@AroundConstruct** e **@PreDestroy**. Esses interceptadores captam eventos do ciclo de vida dos beans, como sua construção e destruição.

Uma das anotações para eventos de ciclo de vida é a **@AroundConstruct**, que define um interceptor para chamadas aos construtores dos beans interceptados. Já **@PostConstruct** define um interceptor que é invocado quando o construtor do bean interceptado finaliza sua execução. Por sua vez, **@PreDestroy** especifica um interpretador a ser invocado antes do bean ser removido da memória.

A **Listagem 8** adiciona os interceptadores **@AroundConstruct**, **@PostConstruct** e **@PreDestroy** na classe **TimerInterceptor**, de forma a interceptar também as etapas do ciclo de vida do objeto **TimerSchedule**, como a sua construção e pós-construção.



Listagem 8. Exemplo de uso dos interceptadores de ciclo de vida.

```
package com.gabrielamorim.cdi.interceptors;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.interceptor.AroundConstruct;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
public class TimerInterceptor {

    @Inject
    private Logger logger;

    @AroundTimeout
    public Object timer(InvocationContext context) throws Exception {
        logger.info("Interceptor before timeout method");
        Object retorno = context.proceed();
        logger.info("Interceptor after timeout method");
        return retorno;
    }

    @AroundConstruct
    public void catchConstructor(InvocationContext context) throws Exception {
        logger.info("Interceptor before constructor");
        context.proceed();
        logger.info("Interceptor after constructor");
    }

    @PostConstruct
    public void postConstruct(InvocationContext context) throws Exception {
        context.proceed();
        logger.info("bean construction finished");
    }

    @PreDestroy
    public void preDestroy(InvocationContext context) throws Exception {
        context.proceed();
        logger.info("bean destroyed");
    }
}
```

Após a implementação dos métodos que receberam essas anotações, como demonstrado nessa listagem, os registros de log do exemplo ficarão conforme apresentado na **Listagem 9**.

Quando a aplicação é iniciada e o primeiro *timer* é disparado, uma instância da classe **TimerSchedule** é criada. Nesse momento, o interceptor é acionado e gerencia a chamada ao construtor por meio do método **TimerInterceptor.catchConstructor()**, anotado com **@AroundConstruct**, registrando no log as duas primeiras linhas da **Listagem 9**. Quando a criação da instância de **TimerSchedule** é finalizada, mais uma vez o interceptor é acionado e o método anotado com **@PostConstruct** é invocado, dessa vez devido ao evento do ciclo de vida que indica a finalização da construção de um bean. Após esses eventos, a execução se dá conforme implementado nos exemplos anteriores. Note que o método interceptor **@PreDestroy** não foi invocado. Isso se deve ao fato que a instância de **TimerSchedule** não foi des-

truída, permanecendo em memória para atender aos próximos disparos do *timer*.

Assim como exposto nos outros exemplos, percebe-se que Interceptors é um recurso muito importante da API de CDI e que pode facilitar muitas tarefas, como a inclusão de requisitos transversais ou filtros de chamadas de métodos, adicionando mais uma camada de segurança ou regras de negócios temporais.

Listagem 9. Resultado em log da implementação de interceptadores de ciclo de vida.

```
14:18:52,920 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 1) Interceptor before constructor
14:18:52,921 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 1) Interceptor after constructor
14:18:52,953 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 1) bean construction finished
14:18:52,973 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 1) Interceptor before timeout method
14:18:52,980 INFO [com.gabrielamorim.cdi.timer.TimerSchedule] (EJB default - 1)
TimerSchedule.scheduledMethod()
14:18:52,980 INFO [com.gabrielamorim.cdi.interceptors.TimerInterceptor]
(EJB default - 1) Interceptor after timeout method
```

Decorators

Decorator é um padrão de projeto que permite adicionar comportamento a um objeto já existente, sendo uma alternativa à herança de classes, já que o comportamento é adicionado a um objeto em tempo de execução e não a uma classe, em tempo de implementação.

A **Figura 2** apresenta um diagrama UML que retrata a estrutura de classes na implementação desse padrão. Note que é criada uma interface base para definir o objeto que terá comportamento adicionado em tempo de execução.

No diagrama, essa interface é chamada de **Component** e define o comportamento que será adicionado por meio do método **operation()**. Já o objeto que será decorado deve implementar a interface base e é chamado **ConcreteComponent**. Além disso, uma classe abstrata, para definir os objetos que adicionarão comportamento ao objeto decorado, precisa ser criada e implementar **Component**. Essa classe é chamada de **Decorator** e deve possuir um atributo do tipo **Component**, que conterá o objeto a ser decorado. Agora, basta implementar os objetos decoradores concretos que estendem **Decorator** e adicionar o comportamento no método **addedBehavior()**.

Com esse padrão, o objeto **ConcreteComponent** nunca será utilizado sem estar encapsulado em um dos objetos decoradores, pois é isso que permite a adição do comportamento em tempo de execução. Quando o método **operation()** do objeto decorador é chamado, ele deve invocar primeiro o método **operation()** do objeto decorado e depois o método **addedBehavior()** para adicionar o comportamento extra.

CDI possibilita a implementação do padrão Decorator de forma bastante simples, por meio de anotações. Quando essa especificação faz uma injeção de dependência declarada por meio da anotação **@Inject**, antes de injetar um objeto do tipo definido é feita uma busca por algum bean anotado com **@Decorator** que seja do mesmo tipo do objeto a ser injetado. Caso exista um objeto decorador, o objeto original é adicionado ao decorador como atributo e o próprio objeto decorador é injetado, ao invés do objeto original.

A melhor maneira de entender o uso dos Decorators na CDI é com um exemplo. Sendo assim, suponha que em uma empresa exista uma calculadora de empregados com um método que retorna uma lista dos melhores funcionários. A lista a seguir retrata um exemplo:

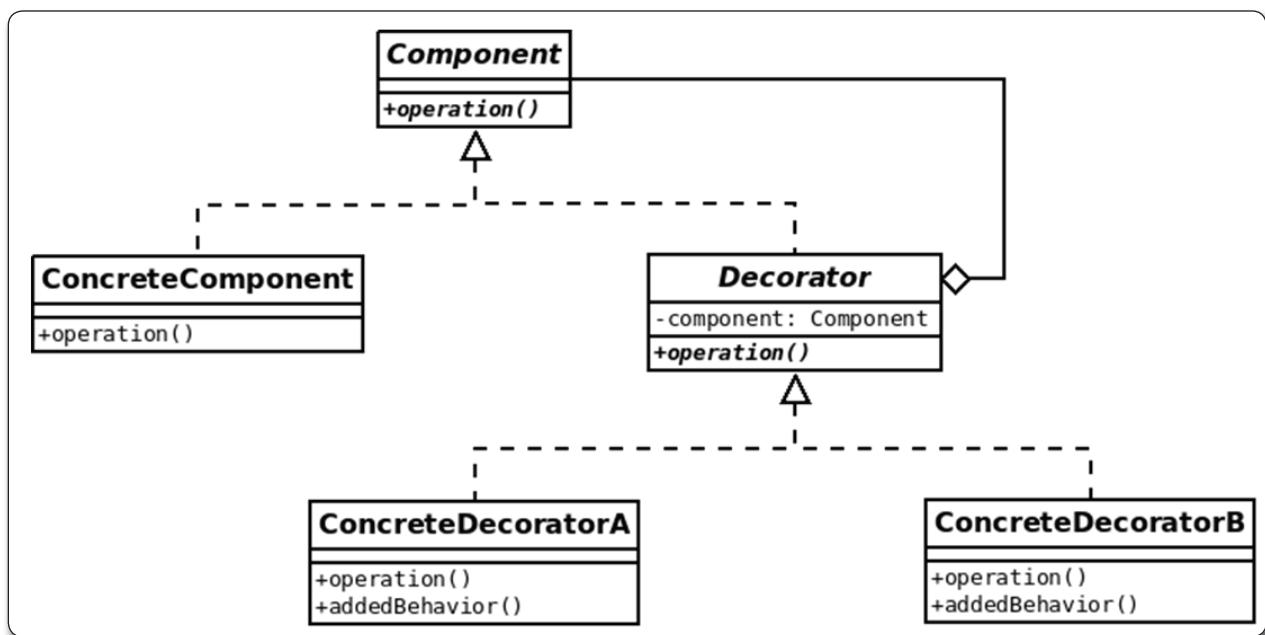


Figura 2. Diagrama UML ilustrando a estrutura de classes do padrão Decorator

1 - John
2 - Mary
3 - Adam
4 - Davy
5 - Debi

Atualmente, essa lista mostra apenas os melhores empregados, mas foi solicitado que ela passe a mostrar, também, a pontuação de cada um. Um detalhe a considerar, no entanto, é que esse requisito é temporário, pois é um experimento para descobrir se a produtividade e a qualidade tenderão a aumentar caso os empregados, além de verem apenas o ranking, vejam também a pontuação.

Como é um requisito temporário que irá adicionar uma funcionalidade a algo já existente, ao invés de alterar o código original que gera a lista, pode-se optar por um Decorator que altere a lista original, adicionando os pontos de cada empregado.

A **Listagem 10** apresenta a implementação do padrão Decorator com CDI nesse cenário. Esse código contém a interface que define a calculadora de empregados, chamada **EmployeesCalculator**, a implementação original da calculadora de empregados, chamada **BestEmployeesCalculator**, e o decorador para adicionar a pontuação de cada empregado, chamado **BestEmployeesCalculatorDecorator**.

O tipo decorador, que no caso é o **BestEmployeesCalculatorDecorator**, deve ser do mesmo tipo do objeto decorado, **BestEmployeesCalculator**. Por isso, ambas as classes devem implementar a interface **EmployeesCalculator**. Nesse caso, a classe decoradora deve ser anotada com **@Decorator** e injetar um objeto de mesmo tipo que não seja um Decorator. Esse objeto injetado é denominado **delegate**, e será ele o utilizado pelo decorador para obter o resultado original e fazer as devidas alterações. O objeto **delegate** deve, ainda, ser anotado com **@Delegate**. São esses os requisitos para a CDI entrar em ação e identificar que há um Decorator para uma classe.



Listagem 10. Implementação do padrão Decorator com CDI.

```
package com.gabrielamorim.cdi.decorators;

public interface EmployeesCalculator {
    public String calculateBestEmployees();
}

package com.gabrielamorim.cdi.decorators;

import javax.enterprise.context.Dependent;

@Dependent
public class BestEmployeesCalculator implements EmployeesCalculator {

    public String calculateBestEmployees() {
        return "1 - John\n" +
            "2 - Mary\n" +
            "3 - Adam\n" +
            "4 - Davy\n" +
            "5 - Debi";
    }
}

package com.gabrielamorim.cdi.decorators;

import javax.decorator.Decorator;
import javax.decorator.Delegate;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent @Decorator
public class BestEmployeesCalculatorDecorator implements EmployeesCalculator {

    @Inject @Delegate
    private EmployeesCalculator delegate;

    public String calculateBestEmployees() {
        String originalEmployeeRanking = delegate.calculateBestEmployees();
        String lines[] = originalEmployeeRanking.split("\n");

        String newEmployeeRanking = "New Employee's ranking:\n";

        int points = 500;
        for(String line : lines) {
            newEmployeeRanking += line + "\t" + points + "\n";
            points = points - 20;
        }

        return newEmployeeRanking;
    }
}
```

A **Listagem 11** demonstra, na prática, o conceito apresentado no parágrafo anterior. Para isso, declara uma servlet que utiliza um **EmployeesCalculator** para exibir o ranking de empregados. Nesse caso, o ranking de empregados é obtido por meio da chamada ao método **calculateBestEmployees()**, de uma classe do tipo **EmployeesCalculator**. Mas, se **EmployeesCalculator** é uma interface, o que será injetado nesse ponto? Como CDI sabe o que

deverá ser injetado? Em tempo de execução, a CDI identifica que há um bean que satisfaz essa injeção de dependência, o **BestEmployeesCalculator**. Além disso, a CDI identifica um Decorator para este tipo, **BestEmployeesCalculatorDecorator**, por meio dos tipos dos objetos. Como o Decorator possui um **BestEmployeesCalculator** como delegate, o bean que seria injetado na servlet é então injetado no Decorator, que, por sua vez, é injetado na servlet. Assim, o método **calculateBestEmployees()** é chamado sob uma instância do decorator, que obtém o resultado original por meio do delegate, realiza as alterações no resultado obtido e retorna o resultado decorado. No caso do nosso exemplo, temos a seguinte resposta:

New Employee ranking:

1 - John	500
2 - Mary	480
3 - Adam	460
4 - Davy	440
5 - Debi	420

Listagem 11. Utilização do decorator implementado na [Listagem 10](#).

```
@WebServlet("/EmployeesCalculator") @Log
public class Servlet2 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Inject
    private EmployeesCalculator bestEmployeesCalculator;

    public Servlet2() {
    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().append(bestEmployeesCalculator
            .calculateBestEmployees());
    }

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Note que não houve alteração na ordem dos empregados, apenas a adição da sua pontuação.

Um último detalhe sobre o funcionamento dos Decorators na CDI é a habilitação desse recurso por meio da declaração dos mesmos no arquivo *beans.xml*. A **Listagem 12** mostra como deve ficar esse arquivo.

Agora que os recursos de Interceptors e Decorators foram examinados em detalhes, cabe uma observação importante sobre o propósito dos dois. Ambos são empregados para estender as funcionalidades da aplicação, porém, em domínios diferentes. Os Interceptors são utilizados para requisitos transversais, ou seja, requisitos não funcionais que geralmente cobrem todo o

sistema. Os Decorators, por sua vez, devem ser utilizados para estender a aplicação, adicionando novas regras de negócio em pontos específicos do código.

Listagem 12. Declaração do decorator no arquivo *beans.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
    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/beans_1_0.xsd">
    <decorators>
        <class>com.gabrielamorim.cdi.decorators.BestEmployeesCalculatorDecorator
        </class>
    </decorators>
</beans>
```

Os recursos apresentados neste artigo são muito úteis para facilitar e melhorar a implementação de vários tipos de requisitos, por exemplo a extensão de funcionalidades, como visto nos exemplos com Decorators, ou a implementação de requisitos transversais, como visto nos exemplos com Interceptors.

Esses mecanismos trazem muitas vantagens ao programador e à equipe de desenvolvimento como um todo, pois são práticas e padrões voltados para o aumento da produtividade tanto em relação ao tempo de projeto quanto em relação ao tempo de operação e manutenção do sistema. Quando esses assuntos são dominados pela equipe, o grau de qualidade tende a aumentar significativamente, pois o código se torna mais organizado e com menos pontos de repetição.

Autor



Gabriel Novais Amorim

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

Engenheiro de software, trabalha com desenvolvimento de sistemas há sete anos e atualmente tem trabalhado com soluções SOA na plataforma Java. Possui as certificações OCJP, OCEJWCD, OCEEJBD, OCEJWSD, TOGAF, IBM-OOAD, IBM-RUP, CompTIA Cloud Essentials e SOACP.

Seus interesses incluem arquitetura de software e metodologias ágeis. Tecnólogo em Análise e Desenvolvimento de Sistemas (Unifeso) e especialista (MBA) em Engenharia de Software (FIAP).



Links:

Especificação da CDI.

<http://www.cdi-spec.org/>

Tutorial de Interceptors - Java EE 6.

<http://docs.oracle.com/javaee/6/tutorial/doc/gkhjx.html>

Tutorial de Decorators - Java EE 6.

<http://docs.oracle.com/javaee/6/tutorial/doc/gkhqf.html>

Using the Timer Service - Java EE 6.

<http://docs.oracle.com/javaee/6/tutorial/doc/bnboy.html>

Apache Camel: Um guia completo – Parte 3

Evoluindo a solução de pagamento bancário através de novas integrações com sistemas externos

ESTE ARTIGO FAZ PARTE DE UM CURSO

Na parte 2 desta série de artigos foi apresentado o exemplo prático a ser implementado. Este exemplo consiste na construção de uma solução integradora para pagamento bancário com cartão de crédito via aplicativo mobile. Além disso, os requisitos funcionais e não funcionais que devem ser atendidos também foram indicados, juntamente com o fluxo de negócio a ser seguido.

A partir dessas informações, foi definido que a arquitetura da solução será baseada em componentes com comunicação interna utilizando mensageria no formato JSON, permitindo assim baixo acoplamento, processamento assíncrono, performance, flexibilidade, tolerância a falhas, escalabilidade e baixo custo de manutenção. Ademais, foram apresentados conceitos e funcionalidades básicas do Apache Camel através da implementação do primeiro fluxo de negócio.

Nesta terceira parte, novos recursos do framework serão introduzidos, com destaque para a implementação dos EIPs, de componentes para mensageria e a continuação da codificação do roteamento e manipulação de mensagens visando atender as regras de negócio.

Para exercitar e aprender esses recursos do Camel, os requisitos definidos no artigo anterior continuarão a serem implementados, em sua sequência natural, para que as integrações iniciais com os sistemas externos bancários e de segurança do cartão sejam realizadas ao passo que a arquitetura da solução também é evoluída.

Fique por dentro

Este artigo apresenta recursos importantes do Apache Camel através da implementação dos requisitos da solução de pagamento bancário. Fundamentado na implementação desses requisitos, será possível constatar as facilidades que o framework oferece para encapsular código complexo capaz de integrar sistemas heterogêneos. Portanto, aqueles que desejam aprender na prática os recursos fundamentais para a construção de um sistema integrado com o auxílio do Apache Camel encontrarão neste artigo códigos valiosos para o alcance desse objetivo.

Implementando os requisitos RF-02-SISBAN e RNF-02-SISBAN

Com os primeiros requisitos já atendidos, será iniciada a implementação do RF-02-SISBAN, que menciona que, após receber os dados da requisição de pagamento do Aplicativo Mobile, a Solução de Pagamento Bancário deve enviar o número do cartão e o CPF do titular para o Sistema Bancário para validação, e também do RNF-02-SISBAN, que define que a comunicação com o Sistema Bancário deve ser realizada via mensageria com o formato JSON.

Para atender os novos requisitos, será necessário continuar com o desenvolvimento do componente **MobileIntegracao**, iniciando pelas alterações na classe **MobileIntegracaoRouteBuilder**, responsável por definir todas as rotas do respectivo componente. Sendo assim, precisamos alterar a classe **MobileIntegracaoRouteBuilder** conforme a codificação da **Listagem 1**.

Basicamente, as modificações realizadas foram na rota **seda:postPagamento**, definida nas linhas 29 a 35, que é uma rota interna responsável por consumir as mensagens enviadas pela rota de entrada (id *rotaEntradaWS*). Esta última, por sua vez, é a fonte de entrada de dados na Solução de Pagamento Bancário.

Portanto, uma variável de nome **converteParaPagamentoBancarioComumProcessor**, do tipo **ConverteParaPagamentoBancarioComumProcessor**, foi declarada na linha 15 e utilizada na linha 31

Listagem 1. Alterações no código da classe MobileIntegracaoRouteBuilder.

```
01 package br.com.devmedia.mobile.integracao.rota;
02
03 import org.apache.camel.LoggingLevel;
04 import org.apache.camel.builder.RouteBuilder;
05 import org.apache.camel.model.rest.RestBindingMode;
06 import org.springframework.beans.factory.annotation.Autowired;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.mobile.integracao.comum.PagamentoRequisicao;
10 import br.com.devmedia.mobile.integracao.processor.ConvertePara
    PagamentoBancarioComumProcessor;
11
12 @Component
13 public class MobileIntegracaoRouteBuilder extends RouteBuilder {
14
15     private ConverteParaPagamentoBancarioComumProcessor convertePara
    PagamentoBancarioComumProcessor;
16
17     @Override
18     public void configure() throws Exception {
19
20         rest("/pagamento").id("rotaEntradaWS")
21             .description("Serviço para efetuar pagamento bancário")
22             .consumes("application/json")
23             .produces("application/json")
24         .post()
25         .bindingMode(RestBindingMode.json)
26         .type(PagamentoRequisicao.class)
27         .to("seda:postPagamento");
28
29         from("seda:postPagamento")
30             .log(LoggingLevel.INFO, "[MobileIntegracao] Nova requisição de
    pagamento bancário")
31             .process(convertParaPagamentoBancarioComumProcessor)
32             .setHeader("FLUXO", constant("NOVA_REQUISICAO_PAGAMENTO
    _BANCARIO"))
33             .log(LoggingLevel.INFO, "[MobileIntegracao] para [PagamentoBancario]
    Nova requisição de pagamento bancário")
34             .to("activemq:queue:PagamentoBancario?deliveryPersistent=false")
35         .end();
36     }
37
38     @Autowired
39     public void setConverteParaPagamentoBancarioComumProcesso
        (ConverteParaPagamentoBancarioComumProcessor convertePara
    PagamentoBancarioComumProcessor) {
40         this.converteParaPagamentoBancarioComumProcessor =
        converteParaPagamentoBancarioComumProcessor;
41     }
42 }
```

para que o respectivo processor manipule e trate a mensagem recebida. O processor **ConverteParaPagamentoBancarioComumProcessor** será desenvolvido nos próximos passos.

Na sequência, a linha 32 define no header da mensagem (Exchange) uma constante de nome **FLUXO** e valor **NOVA_REQUISICAO_PAGAMENTO_BANCARIO** para que quando o componente seguinte receber a mensagem consiga identificar de qual fluxo a mesma pertence e, então, tratar e dar o destino correto. Essa abordagem é uma implementação do EIP Content-Based Router.

Adiante, verifica-se na linha 33 a inclusão de log com a mensagem “[MobileIntegracao] para [PagamentoBancario] Nova requisição de pagamento bancário”, para registro do componente de origem e destino da mensagem. Essa abordagem é uma boa prática pois em sistemas componentizados e em cenários de integração é muito importante indicar de onde e para onde a mensagem ou fluxo está caminhando, para que seja fácil a análise posterior.

Finalizando esse trecho, a linha 34 faz de fato o envio da mensagem para uma fila, que, no caso, se chama **PagamentoBancario**. Para isso é utilizado o componente **activemq-camel**, que foi previamente configurado no arquivo *application-context.xml* do componente **MobileIntegracao**. Ainda nessa mesma linha, observa-se o uso do parâmetro **deliveryPersistent**, para indicar que as mensagens não precisam ser persistidas em disco quando o servidor de mensageria desligar, pois em nosso cenário não há necessidade de manter as mensagens. Além disso, essa configuração deixará o processo de mensageria mais rápido. Uma dica interessante é que não é preciso criar a fila **PagamentoBancario** previamente no servidor de mensageria, pois no carregamento da aplicação a mesma é criada automaticamente.

Por fim, as linhas 38 a 41 implementam o método setter da variável **converteParaPagamentoBancarioComumProcessor**, para que, juntamente com a anotação **@Autowired**, o Spring faça a injeção e gerenciamento do processor.

Realizadas as modificações em **MobileIntegracaoRouteBuilder**, é preciso criar o processor **ConverteParaPagamentoBancarioComumProcessor**, que define a última palavra de sua nomenclatura como **Processor** para que um padrão seja respeitado com o objetivo de facilitar a identificação de classes desse tipo. Na sequência, será necessário criar o pacote **br.com.devmedia.mobile.integracao.processor** e implementar a classe **ConverteParaPagamentoBancarioComumProcessor**, conforme a **Listagem 2**.

Observando a implementação anterior, nota-se que na linha 14 é incluída a anotação **@Component** para que o Spring consiga fazer a injeção de dependência. Na sequência, a linha 15 declara de fato o processor **ConverteParaPagamentoBancarioComumProcessor** através da implementação da interface **Processor** do Camel, isto é, com a implementação dessa interface é que a classe em questão é transformada em um processor. Na linha 18 o método **void process(Exchange)** é sobreescrito para que a lógica de manipulação de mensagens seja realizada.

Na linha 19, o objeto **PagamentoRequisicao**, que representa os dados da requisição de pagamento bancário — feita através do web service de entrada da solução — é recuperado, e a partir dele um novo objeto, do tipo **PagamentoBancarioComum**, é instanciado e populado. Nas seções seguintes, a classe **PagamentoBancarioComum** será implementada e mais detalhes serão apresentados. Neste momento, saiba que o objetivo com a criação dessa classe é servir como um POJO e trafegar os dados entre os componentes da solução.

Listagem 2. Código da classe ConverteParaPagamentoBancarioComumProcessor.

```
01 package br.com.devmedia.mobile.integracao.processor;
02
03 import java.util.Date;
04
05 import org.apache.camel.Exchange;
06 import org.apache.camel.Processor;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.mobile.integracao.comum.PagamentoBancarioComum;
10 import br.com.devmedia.mobile.integracao.comum.PagamentoRequisicao;
11 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
12 import br.com.devmedia.pagamento.bancario.processor.ProcessorUtil;
13
14 @Component
15 public class ConverteParaPagamentoBancarioComumProcessor implements
    Processor {
16
17     @Override
18     public void process(Exchange exchange) throws Exception {
19         PagamentoRequisicao pagamentoRequisicao = exchange.getIn().getBody(PagamentoRequisicao.class);
20
21         PagamentoBancarioComum pagamentoBancario =
22             new PagamentoBancarioComum();
23         populaPagamentoBancarioComum(pagamentoBancario, pagamentoRequisicao);
24         ProcessorUtil.setObjetoHeader(exchange, pagamentoBancario);
25     }
26
27     private void populaPagamentoBancarioComum(PagamentoBancarioComum
28         pagamentoBancario, PagamentoRequisicao pagamentoRequisicao) {
29         pagamentoBancario.setIdTransacao(pagamentoRequisicao.getIdTransacao());
30         pagamentoBancario.setCpfTitularCartao(pagamentoRequisicao
31             .getCpfTitularCartao());
32         pagamentoBancario.setNumeroCartao(pagamentoRequisicao
33             .getNumeroCartao());
34         pagamentoBancario.setCodigoSegurancaCartao(pagamentoRequisicao
35             .getCodigoSegurancaCartao());
36         pagamentoBancario.setDataInicialTransacao(new Date());
37         pagamentoBancario.setValorCompra(pagamentoRequisicao.getValorCompra());
38     }
39 }
```

Continuando, na linha 21 é invocado o método **populaPagamentoBancarioComum(PagamentoBancarioComum, PagamentoRequisicao)**, cujo objetivo é simplesmente popular um objeto **PagamentoBancarioComum** a partir dos dados de um objeto **PagamentoRequisicao**. A implementação da lógica contida nesse método é exibida nas linhas 27 a 34.

Na sequência, a linha 22 apresenta o código para incluir no header da mensagem (Exchange) que está sendo manipulada o objeto da classe **PagamentoBancarioComum**, para que ele seja salvo e posteriormente recuperado quando qualquer componente da solução precisar de dados da transação. Para que essa abordagem funcione corretamente, é necessário que o objeto em questão tenha seus dados incluídos e atualizados no decorrer do fluxo pelos próprios componentes conforme as integrações ocorrem.

Ainda na linha 22, nota-se a utilização de uma biblioteca auxiliar através da classe **ProcessorUtil** para incluir um objeto qualquer no header da mensagem com o uso do método **setObjetoHeader(Exchange, Object)**. Essa biblioteca, nomeada de **ComponenteUtil**, será desenvolvida para conter código comum que será utilizado por todos os demais componentes da solução e, assim, evitar repetição, isto é, seguir boas práticas de Engenharia de Software. A implementação da biblioteca será detalhada em breve.

Por fim, a linha 23 faz a conversão do objeto da classe **PagamentoBancarioComum** para o formato JSON através do método **converteObjetoParaJson(Object)**, contido na classe **ConversorJson**, respeitando o formato em que as mensagens deverão ser trafegadas entre os componentes. Na linha 24, o JSON é de fato incluído no body do Exchange e assim está apto a ser recebido pelo próximo componente do fluxo.

Desta forma, para que a classe **ConverteParaPagamentoBancarioComumProcessor** compile e funcione corretamente, será necessária a criação da biblioteca auxiliar **ComponenteUtil** e

também da biblioteca comum **PagamentoBancarioComum**. Sendo assim, as seções a seguir detalharão a criação de ambas.

Implementando a biblioteca ComponenteUtil

Conforme mencionado anteriormente, a biblioteca **ComponenteUtil** será responsável por prover código comum que todos os outros componentes da solução necessitam utilizar, evitando, dessa forma, a repetição de código. Para isso, vamos criar uma estrutura de projeto/módulo Java Standalone (JAR) dentro do diretório do projeto agregador com o nome de **ComponenteUtil**. Feito isso, adicione nesse novo projeto um arquivo *pom.xml* (vide **Listagem 3**) para indicar as configurações básicas e as dependências externas de bibliotecas.

Listagem 3. Arquivo pom.xml da biblioteca ComponenteUtil.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04   http://maven.apache.org/xsd/maven-4.0.0.xsd">
05   <modelVersion>4.0.0</modelVersion>
06
07   <parent>
08     <groupId>br.com.devmedia</groupId>
09     <artifactId>solucao-pagamento-bancario</artifactId>
10     <version>1.0.</version>
11   </parent>
12
13   <name>ComponenteUtil</name>
14   <artifactId>componente-util</artifactId>
15   <packaging>jar</packaging>
16
17   <dependencies>
18     <dependency>
19       <groupId>org.apache.camel</groupId>
20       <artifactId>camel-jackson</artifactId>
21     </dependency>
22   </dependencies>
23
24 </project>
```

Observa-se que é utilizada a tag **parent** nas linhas 5 a 9 para informar que as definições feitas no *pom.xml* do projeto agregador devem ser utilizadas nesse módulo. Dessa forma, não é necessário definir o **groupId** e o **version** para o próprio módulo, pois essas informações serão obtidas a partir do *SolucaoPagamentoBancario*.

Além disso, a tag **packaging** é definida como JAR, pois assim um JAR dessa biblioteca será gerado para que seja incorporado e utilizado pelos outros componentes. Por fim, nas linhas 15 a 20 é declarada a dependência da biblioteca **camel-jackson**, que auxilia nas conversões de JSON para objeto e vice-versa.

Seguindo com a implementação, deve-se criar um pacote com o nome **br.com.devmedia.pagamento.bancario.conversor** para conter a classe **ConversorJson**, responsável por realizar conversões de objetos em formato JSON. Seu código é apresentado na **Listagem 4**.

Listagem 4. Código da classe ConversorJson, módulo ComponenteUtil.

```
01 package br.com.devmedia.pagamento.bancario.conversor;
02
03 import java.io.IOException;
04 import java.io.StringWriter;
05
06 import com.fasterxml.jackson.core.JsonGenerator;
07 import com.fasterxml.jackson.databind.DeserializationFeature;
08 import com.fasterxml.jackson.databind.MappingJsonFactory;
09 import com.fasterxml.jackson.databind.ObjectMapper;
10
11 public abstract class ConversorJson {
12
13     public static String converteObjetoParaJson(final Object objeto)
14         throws Exception {
15         final StringWriter sw = new StringWriter();
16         final ObjectMapper mapper = new ObjectMapper();
17         final MappingJsonFactory jsonFactory = new MappingJsonFactory();
18         final JsonGenerator jsonGenerator = jsonFactory.createGenerator(sw);
19         try {
20             mapper.writeValue(jsonGenerator, objeto);
21             return sw.getBuffer().toString();
22         } catch (Exception e) {
23             throw e;
24         } finally {
25             sw.close();
26         }
27
28     public static <T> T converteJsonParaObjeto(final String json, final Class<T>
29         objeto) throws IOException, InstantiationException, IllegalAccessException {
30         final ObjectMapper mapeador = new ObjectMapper();
31         mapeador.configure(DeserializationFeature.FAIL_ON_UNKNOWN
32         _PROPERTIES, false);
33         final T obj = mapeador.readValue(json, objeto);
34     }
35 }
```

Verificando essa classe, nota-se nas linhas 13 a 26 a criação do método **converteObjetoParaJson(Object)** para realizar conversões de objeto para JSON. Para tal tarefa, é utilizada a biblioteca Jackson, já conhecida e consolidada no mundo Java. Já as linhas 28 a 33 demonstram a criação do método **converteJsonParaObjeto(String, Class<T>)** utilizando **Generics** para converter um JSON em objeto,

também através da biblioteca Jackson. Vale ressaltar que na linha 30 é indicada a configuração **DeserializationFeature.FAIL_ON_UNKOWN_PROPERTIES** com o valor **false** para que atributos contidos no JSON que não existam na classe final sejam ignorados e, assim, erros não sejam gerados durante a conversão.

Continuando com o desenvolvimento da biblioteca **ComponenteUtil**, crie o pacote **br.com.devmedia.pagamento.bancario.processor** e implemente a classe **ProcessorUtil**, conforme a **Listagem 5**. Essa classe será responsável por conter os métodos úteis que são utilizados pelos demais componentes para salvar e recuperar dados de uma mensagem.

Analisando o código apresentado, observa-se que nas linhas 9 a 11 existe a implementação do método **setObjetoHeader(Exchange, Object)**, cuja finalidade é incluir no header de uma mensagem do tipo Exchange um POJO em formato JSON contendo os dados que serão utilizados pelos componentes da Solução de Pagamento Bancário.

Listagem 5. Código da classe ProcessorUtil, módulo ComponenteUtil.

```
01 package br.com.devmedia.pagamento.bancario.processor;
02
03 import java.io.IOException;
04 import org.apache.camel.Exchange;
05 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
06
07 public abstract class ProcessorUtil {
08
09     public static void setObjetoHeader(final Exchange exchange,
10         final Object objeto) throws Exception {
11         exchange.getIn().setHeader(objeto.getClass().getName(),
12             ConversorJson.converteObjetoParaJson(objeto));
13     }
14
15     public static void setObjetoHeader(final Exchange exchange,
16         final String chaveAdicional, final Object objeto) throws Exception {
17         exchange.getIn().setHeader(objeto.getClass().getName()
18             + chaveAdicional, ConversorJson.converteObjetoParaJson(objeto));
19     }
20
21     public static <T> T getObjetoHeader(final Exchange exchange,
22         final Class<T> tipoClass)
23     throws IOException, InstantiationException, IllegalAccessException {
24         final String tipoStr = exchange.getIn().getHeader(tipoClass.getName(),
25             String.class);
26         if (tipoStr != null) {
27             final T tipo = ConversorJson.converteJsonParaObjeto(tipoStr, tipoClass);
28             return tipo;
29         }
30         return null;
31     }
32
33     public static <T> T getObjetoHeader(final Exchange exchange,
34         final String chaveAdicional, final Class<T> tipo)
35     throws IOException, InstantiationException, IllegalAccessException {
36         final String tipoStr = exchange.getIn().getHeader(tipo.getName()
37             + chaveAdicional, String.class);
38         if (tipoStr != null) {
39             return ConversorJson.converteJsonParaObjeto(tipoStr, tipo);
40         }
41         return null;
42     }
43 }
```

Já o método das linhas 13 a 15 possui o mesmo objetivo, porém permite que uma chave adicional seja incluída no header para situações em que há mais de um objeto do mesmo tipo a ser salvo. Continuando, o método das linhas 17 a 25 é complementar aos anteriores e recupera um objeto do header a partir da indicação de uma classe. Por fim, o último método, indicado nas linhas 27 a 34, tem como objetivo recuperar um objeto do header baseado em uma classe específica, mas com a diferença de utilizar a chave adicional.

Finalizada a implementação da biblioteca, é necessário incluir a mesma nas dependências do componente **MobileIntegracao**, para isso, adicione no *pom.xml* do respectivo componente a mesma declaração da **Listagem 6**. Deve-se, também, incluir a biblioteca no *pom.xml* do projeto agregador, da mesma forma como está sendo feito a seguir:

```
<module>ComponenteUtil</module>
```

Listagem 6. Dependência a ser incluída no pom.xml do componente MobileIntegracao

```
<dependency>
  <groupId>${project.parent.groupId}</groupId>
  <artifactId>componente-util</artifactId>
  <version>${project.parent.version}</version>
</dependency>
```

Após todos esses passos, o processador **ConverteParaPagamentoBancarioComumProcessor** está quase concluído, faltando apenas a criação da biblioteca comum **PagamentoBancarioComum**.

Implementando a biblioteca PagamentoBancarioComum

Para a implementação da biblioteca **PagamentoBancarioComum**, cuja finalidade é representar os dados que serão trafegados entre o componente **PagamentoBancario** e os demais componentes internos da solução, é necessário primeiramente criar um novo projeto/módulo do tipo Java Standalone (JAR) dentro do diretório do projeto agregador com o nome **PagamentoBancarioComum**. Feito isso, adicione a esse novo projeto um arquivo *pom.xml* com o código da **Listagem 7**.

Prosseguindo com a codificação da biblioteca, deve-se criar o pacote **br.com.devmedia.pagamento.bancario.comum** e implementar a classe **PagamentoBancarioComum** conforme o código da **Listagem 8**.

Nesse código é possível notar que foram definidos os atributos que serão utilizados durante todas as etapas da transação do pagamento bancário.

Na sequência, deve-se incluir a nova biblioteca **PagamentoBancarioComum** nas dependências do projeto **MobileIntegracao** conforme a **Listagem 9**, para que ela seja a responsável por trafegar os dados e fazer a comunicação com o componente **PagamentoBancario**, e também adicioná-la na declaração de módulos do *SolucaoPagamentoBancario*, como expõe a seguir:

```
<module>PagamentoBancarioComum</module>
```

Listagem 7. Arquivo pom.xml do módulo PagamentoBancarioComum.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
02   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
03
04   <modelVersion>4.0.0</modelVersion>
05
06   <parent>
07     <groupId>br.com.devmedia</groupId>
08     <artifactId>solucao-pagamento-bancario</artifactId>
09     <version>1.0.</version>
10   </parent>
11
12   <name>PagamentoBancarioComum</name>
13   <artifactId>pagamento-bancario-comum</artifactId>
14   <packaging>jar</packaging>
15
16 </project>
```

Listagem 8. Código da classe PagamentoBancarioComum.

```
01 package br.com.devmedia.mobile.integracao.comum;
02
03 import java.io.Serializable;
04 import java.math.BigDecimal;
05 import java.util.Date;
06
07 public class PagamentoBancarioComum implements Serializable {
08
09   private static final long serialVersionUID = 5594233232137106383L;
10
11   private String idTransacao;
12   private String cpfTitularCartao;
13   private String numeroCartao;
14   private String codigoSegurancaCartao;
15   private BigDecimal valorCompra;
16   private Date dataInicialTransacao;
17   private Date dataFinalTransacao;
18   private String status;
19
20   public PagamentoBancarioComum() {}
21
22   // Getters e Setters dos atributos omitidos.
23
24   // Método toString() omitido.
25
26 }
```

Listagem 9. Dependência a ser incluída no pom.xml do componente MobileIntegracao.

```
<dependency>
  <groupId>${project.parent.groupId}</groupId>
  <artifactId>pagamento-bancario-comum</artifactId>
  <version>${project.parent.version}</version>
</dependency>
```

Com as duas bibliotecas prontas, o processador **ConverteParaPagamentoBancarioComumProcessor** está concluído. Portanto, já é possível continuar com a implementação dos requisitos e iniciar a construção do próximo componente da solução, o **PagamentoBancario**. Sua responsabilidade é orquestrar o fluxo e deter as regras de negócio, sendo o componente central da solução. Assim, **PagamentoBancario** possuirá a biblioteca **PagamentoBancarioComum**, a qual será compartilhada entre os demais componentes a fim de viabilizar a troca de mensagens.

Implementando o componente PagamentoBancario

Para a implementação do componente **PagamentoBancario**, é necessário criar uma estrutura de projeto Java Web (WAR) dentro do diretório do projeto agregador com o nome **PagamentoBancario**. Na sequência, deve-se adicionar esse novo componente nos módulos do projeto agregador e incluir, no arquivo *pom.xml*, as dependências para as bibliotecas **ComponenteUtil** e **PagamentoBancarioComum**.

Prosseguindo, deve-se ajustar o arquivo *web.xml*, localizado no diretório *src/main/webapp/WEB-INF*, conforme a **Listagem 10**, para que a aplicação carregue (ao iniciar) as configurações do Spring, Camel e ActiveMQ. Além disso, é preciso incluir um novo arquivo denominado *application-context.xml* — semelhante à **Listagem 11** — no diretório *src/main/resources*, pois são através das declarações dos beans (gerenciados pelo Spring) contidas nesse arquivo que a conexão com o ActiveMQ é efetuada e o carregamento inicial das rotas do Camel é feito.

Analizando mais a fundo, as linhas 14 a 16 informam para o Camel qual a classe que contém as rotas a serem iniciadas quando a aplicação carregar. Nesse caso, trata-se da **PagamentoBancarioRouteBuilder**, referenciada pela tag *camel:routeBuilder* na linha 15.

Adiante, na linha 18, é definida uma configuração do Spring para que o mesmo faça uma busca e injeção de dependência de forma automática a partir do pacote **br.com.devmedia.pagamento.bancario**. Com isso, as classes anotadas com **@Component** e que estejam nesse pacote serão automaticamente gerenciadas pelo Spring. Essa configuração é valiosa, pois evita que seja necessário indicar classe por classe no arquivo XML.

Prosseguindo com a análise da listagem, as linhas 20 a 22 criam um bean chamado **jmsConnectionFactory** para fazer a conexão com o ActiveMQ. Vale ressaltar que na linha 21 é indicada a URL para conexão com o servidor de mensageira, incluindo a opção **failover** para disponibilizar redundância nas conexões, ou seja, se a conexão ou comunicação falhar com o servidor da primeira URL, automaticamente é estabelecida uma conexão com o segundo servidor através da outra URL. O parâmetro **randomize** indica, nesse caso, que a conexão com os servidores deve acontecer conforme a ordem da configuração, e não de forma aleatória.

Na sequência do arquivo, é declarado nas linhas 24 a 27 mais um bean, com o nome **pooledConnectionFactory**, para ser um pool de conexões com o ActiveMQ baseado nas configurações do bean anterior. Esse bean permitirá no máximo oito conexões em uso ao mesmo tempo, conforme a propriedade **maxConnections** da linha 25.

Já nas linhas 29 a 32, outro bean é criado com o nome **jmsConfig** para indicar a quantidade de consumidores para cada conexão do ActiveMQ definida anteriormente. Essa configuração é realizada na linha 31 através da propriedade **concurrentConsumers**.

Finalizando a criação do arquivo *application-context.xml*, as linhas 34 a 36 definem um bean chamado **activemq**, cuja implementação é o componente do Camel **camel-activemq**. Esse bean inclui em suas configurações o bean anterior, **jmsConfig**, conforme observa-se na linha 35, e com esse último bean é que será possível enviar mensagens para as filas nas rotas do Camel.

Listagem 10. web.xml do módulo PagamentoBancario.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xmlns="http://java.sun.com/xml/ns/javaee"
04   xmlns:web="http://java.sun.com/
05   xml/ns/javaee/web-app_2_5.xsd"
06   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
07   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
08   id="WebApp_ID" version="2.5" metadata-complete="true">
09   <display-name>Pagamento Bancario</display-name>
10  <!-- Listener -->
11  <listener>
12    <listener-class>org.springframework.web.context.ContextLoaderListener
13    </listener-class>
14  </listener>
15  <!-- Listener -->
16  <listener>
17    <listener-class>org.springframework.web.context.request.
18      RequestContextListener</listener-class>
19  </listener>
20  <!-- Context-param -->
21  <context-param>
22    <param-name>contextConfigLocation</param-name>
23    <param-value>classpath:application-context.xml</param-value>
24  </context-param>
25
26 </web-app>
```

Listagem 11. application-context.xml do componente PagamentoBancario.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xmlns:camel="http://camel.apache.org/schema/spring"
05   xmlns:context="http://www.springframework.org/schema/context"
06   xsi:schemaLocation="http://www.springframework.org/schema/context
07   http://www.springframework.org/schema/context/spring-context-3.0.xsd
08   http://www.springframework.org/schema/beans
09   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
10   http://camel.apache.org/schema/spring
11   http://camel.apache.org/schema/spring/camel-spring.xsd">
12
13 <camel:camelContext id="pagamentoBancario">
14   <camel:routeBuilder ref="pagamentoBancarioRouteBuilder"/>
15 </camel:camelContext>
16
17 <context:component-scan base-package="br.com.devmedia.pagamento
18 .bancario"/>
19
20 <bean id="jmsConnectionFactory" class="org.apache.activemq
21 .ActiveMQConnectionFactory">
22   <property name="brokerURL" value="failover:(tcp://localhost:61616,
23   tcp://127.0.0.1:61616)?randomize=false"/>
24 </bean>
25
26 <bean id="pooledConnectionFactory" class="org.apache.activemq.pool.
27 PooledConnectionFactory" init-method="start" destroy-method="stop">
28   <property name="maxConnections" value="8"/>
29   <property name="connectionFactory" ref="jmsConnectionFactory"/>
30 </bean>
31
32 <bean id="jmsConfig" class="org.apache.camel.component.jms
33 .JmsConfiguration">
34   <property name="connectionFactory" ref="pooledConnectionFactory"/>
35   <property name="concurrentConsumers" value="10"/>
36 </bean>
37
38 </beans>
```

Apache Camel: Um guia completo – Parte 3

Concluída a criação dos dois arquivos anteriores, deve-se incluir o pacote **br.com.devmedia.pagamento.bancario.rota** e codificar a classe **PagamentoBancarioRouteBuilder** de acordo com a **Listagem 12**.

Listagem 12. Código inicial da classe PagamentoBancarioRouteBuilder.

```
01 package br.com.devmedia.pagamento.bancario.rota;
02
03 import org.apache.camel.LoggingLevel;
04 import org.apache.camel.Predicate;
05 import org.apache.camel.builder.PredicateBuilder;
06 import org.apache.camel.builder.RouteBuilder;
07 import org.springframework.beans.factory.annotation.Autowired;
08 import org.springframework.stereotype.Component;
09
10 import br.com.devmedia.pagamento.bancario.processor.PopulaBancoComumProcessor;
11
12 @Component
13 public class PagamentoBancarioRouteBuilder extends RouteBuilder {
14
15     private PopulaBancoComumProcessor populaBancoComumProcessor;
16
17     @Override
18     public void configure() throws Exception {
19
20         from("activemq:queue:PagamentoBancario?concurrentConsumers=5")
21             .id("rotaEntradaPagamentoBancario")
22             .choice()
23                 .when(novaRequisicaoPagamentoBancario)
24                     .log(LoggingLevel.INFO, "[PagamentoBancario]
Nova requisição de pagamento bancário")
25                     .to("seda:enviaDadosCartaoValidacao")
26             .endChoice()
27         .end();
28
29         from("seda:enviaDadosCartaoValidacao")
30             .log(LoggingLevel.INFO, "[PagamentoBancario] Preparando os dados do
cartão para validação")
31             .process(populaBancoComumProcessor)
32             .log(LoggingLevel.INFO, "[PagamentoBancario] para [BancoIntegracao]
Enviando os dados do cartão para validação")
33             .to("activemq:queue:BancoIntegracao?deliveryPersistent=false")
34     }
35
36     private final Predicate novaRequisicaoPagamentoBancario =
37     PredicateBuilder.and(header("FLUXO").isEqualTo
38     ("NOVA_REQUISICAO_PAGAMENTO_BANCARIO"));
39
40     @Autowired
41     public void setPopulaBancoComumProcessor(PopulaBancoComumProcessor
42         populaBancoComumProcessor) {
43         this.populaBancoComumProcessor = populaBancoComumProcessor;
44     }
45 }
```

Observa-se que as linhas 20 a 26 definem uma rota para receber mensagens da fila **PagamentoBancario** e repassá-las para outra fila interna, chamada **enviaDadosCartaoValidacao**. Na linha 20, o código declara cinco consumidores em paralelo para a fila **PagamentoBancario** com o auxílio do parâmetro **concurrentConsumers**, aumentando, dessa forma, o poder de processamento. Prosseguindo, as linhas 21 a 25 implementam o EIP Content-Based Router para que a mensagem recebida seja roteada para a

fila correspondente levando-se em consideração o seu conteúdo. Assim, baseado no conteúdo da mensagem, a mesma é direcionada para um fluxo específico. Para realizar essa verificação é utilizado um recurso do Camel, chamado de Predicado (ou Predicate, em inglês), que funciona como uma condição. Esse predicado é definido na linha 36 e sua lógica é verificar se existe no header da mensagem uma constante chamada **FLUXO** com o valor **NOVA_REQUISICAO_PAGAMENTO_BANCARIO**. Apenas para recordar, essa constante foi definida no componente **MobileIntegracao**.

Avançando com o entendimento do código é possível verificar que entre as linhas 28 e 33 é definida uma rota interna chamada **enviaDadosCartaoValidacao**, sendo declarado na linha 30 a inclusão de um processador do tipo **PopulaBancoComumProcessor** para manipular as mensagens. Na linha 32 o código faz o envio das mensagens para a fila **BancoIntegracao**, isto é, a fila em que o próximo componente da solução consumirá as mensagens.

Portanto, é necessário codificar o processador **PopulaBancoComumProcessor**, responsável por transformar os dados contidos em um objeto **PagamentoBancarioComum** em **BancoIntegracaoComum**, isto é, o objeto que o próximo componente do fluxo utilizará. Para isso, crie um novo pacote com o nome **br.com.devmedia.pagamento.bancario.processor** e implemente a classe **PopulaBancoComumProcessor** nesse mesmo pacote conforme a **Listagem 13**.

Listagem 13. Código da classe PopulaBancoComumProcessor.

```
01 package br.com.devmedia.pagamento.bancario.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.springframework.stereotype.Component;
06
07 import br.com.devmedia.banco.integracao.comum.BancoIntegracaoComum;
08 import br.com.devmedia.pagamento.bancario.comum
09 .PagamentoBancarioComum;
09 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
10
11 @Component
12 public class PopulaBancoComumProcessor implements Processor {
13
14     @Override
15     public void process(Exchange exchange) throws Exception {
16         PagamentoBancarioComum pagamentoBancario = ConversorJson.
17             converteJsonParaObjeto(exchange.getIn().getBody().toString(),
18             PagamentoBancarioComum.class);
19         BancoIntegracaoComum bancoIntegracao = new BancoIntegracaoComum();
20         bancoIntegracao.setPagamentoBancario(pagamentoBancario);
21         String json = ConversorJson.converteObjetoParaJson(bancoIntegracao);
22         exchange.getIn().setBody(json);
23
24     private void populaBancoIntegracao(PagamentoBancarioComum
25         pagamentoBancario, BancoIntegracaoComum bancoIntegracao) {
26         bancoIntegracao.setCpfTitularCartao(pagamentoBancario
27             .getCpfTitularCartao());
28         bancoIntegracao.setNumeroCartao(pagamentoBancario.getNumeroCartao());
29     }
30 }
```

O código dessa listagem segue a mesma proposta do processador **ConverteParaPagamentoBancarioComumProcessor**, ou seja, instanciar um POJO — neste caso, do tipo **BancoIntegracaoComum** — contendo os dados que serão necessários ao próximo componente do fluxo. Para isso, o método das linhas 15 a 21 popula os atributos do POJO **BancoIntegracaoComum** (CPF do titular e número do cartão) a partir do objeto **PagamentoBancarioComum**, recuperado através da mensagem de entrada.

Após a conclusão desse código, para seu correto funcionamento, ainda é necessária a criação da biblioteca **BancoIntegracaoComum**.

Implementando a biblioteca BancoIntegracaoComum

A implementação da biblioteca **BancoIntegracaoComum**, cuja finalidade é representar os dados que serão trafegados entre os componentes internos da solução e também com o sistema externo bancário, deve-se iniciar com a criação de um novo projeto/módulo do tipo Java Standalone (JAR) dentro do diretório do projeto agregador com o nome **BancoIntegracaoComum**. Em seguida, crie nesse novo projeto um arquivo *pom.xml* com o código da **Listagem 14** para definir as configurações básicas da biblioteca.

Listagem 14. Arquivo *pom.xml* da biblioteca BancoIntegracaoComum.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
02
03   <modelVersion>4.0.0</modelVersion>
04
05   <parent>
06     <groupId>br.com.devmedia</groupId>
07     <artifactId>solucao-pagamento-bancario</artifactId>
08     <version>1.0.</version>
09   </parent>
10
11   <name>BancoIntegracaoComum</name>
12   <artifactId>banco-integracao-comum</artifactId>
13   <packaging>jar</packaging>
14
15 </project>
```

Logo após, devemos criar o pacote **br.com.devmedia.banco.integracao.comum** e inserir a classe **BancoIntegracaoComum** conforme a **Listagem 15**. Essa classe representa um POJO que contém os dados que serão enviados ao Sistema Bancário.

Analizando essa listagem é possível notar que a classe possui os dois atributos (CPF do titular e número do cartão) que serão enviados ao sistema externo. Portanto, deve-se incluir essa nova biblioteca na declaração de módulos do *SolucaoPagamentoBancario*, para que ela seja incorporada à solução, e também nas dependências do componente **PagamentoBancario**, para que o mesmo popule o POJO com os dados necessários e transmita-o ao componente **BancoIntegracao**.

Concluída a implementação da biblioteca **BancoIntegracaoComum**, é a vez da construção do componente **BancoIntegracao**.

Listagem 15. Código da classe **BancoIntegracaoComum**.

```
01 package br.com.devmedia.banco.integracao.comum;
02
03 import java.io.Serializable;
04
05 public class BancoIntegracaoComum implements Serializable {
06
07   private static final long serialVersionUID = -5978319920262469013L;
08
09   private String cpfTitularCartao;
10   private String numeroCartao;
11
12   public BancoIntegracaoComum() {}
13
14   // Getters e Setters dos atributos omitidos.
15
16   // Método toString() omitido.
17 }
```

Esse novo componente será responsável por fazer a comunicação com o Sistema Bancário e, por isso, todas as regras e necessidades voltadas a essa integração devem ficar concentradas nele.

Implementando o componente BancoIntegracao

Para a implementação do componente **BancoIntegracao**, é necessário criar uma estrutura de projeto Java Web (WAR) dentro do diretório do projeto agregador com o nome **BancoIntegracao**. Na sequência, deve-se adicionar esse novo componente nos módulos do projeto agregador e incluir no arquivo *pom.xml* as dependências para as bibliotecas **ComponenteUtil** e **BancoIntegracaoComum**.

Prosseguindo, deve-se ajustar o *web.xml*, localizado no diretório *src/main/webapp/WEB-INF*, conforme a **Listagem 16**, para que a aplicação carregue (ao iniciar) as configurações do Spring, Camel e ActiveMQ. Além disso, é preciso incluir um novo arquivo denominado *application-context.xml* — semelhante à **Listagem 17** — no diretório *src/main/resources*, pois é através das declarações dos beans (gerenciados pelo Spring) contidas nesse arquivo que a conexão com o ActiveMQ é efetuada e o carregamento inicial das rotas do Camel é feito.

As configurações definidas no arquivo *application-context.xml* da **Listagem 17** seguem o mesmo padrão do *application-context.xml* do componente **PagamentoBancario**. O diferencial deste arquivo está nas linhas 14 a 16, que informam ao Camel que a classe que contém as rotas a serem iniciadas quando a aplicação carregar é a **BancoIntegracaoRouteBuilder**, referenciada pela tag **camel:routeBuilder** na linha 15.

Continuando com o desenvolvimento do componente, após a criação dos arquivos deve-se adicionar o pacote **br.com.devmedia.banco.integracao.rota** e incluir a classe **BancoIntegracaoRouteBuilder** conforme a **Listagem 18**.

Explorando o código apresentado, observa-se nas linhas 18 a 23 a definição de uma rota para receber mensagens da fila **BancoIntegracao**. Nesse trecho de código onde é definida a rota, um processador do tipo **PreparaDadosCartaoParaValidacaoProcessor** é declarado na linha 20, com a finalidade de criar mensagens compatíveis com a esperada pelo Sistema Bancário.

Apache Camel: Um guia completo – Parte 3

Por fim, na linha 22 é feito o envio da mensagem para a fila **SistemaExternoBancario**, para que o Sistema Bancário receba a mensagem. A partir desse ponto, a mensagem está fora dos domínios da Solução de Pagamento Bancário.

Em cenários reais, a definição e criação dessa fila necessita ser alinhada com a equipe responsável pelo outro sistema, para que a comunicação entre ambos seja realizada com sucesso.

Listagem 16. web.xml do componente BancoIntegracao.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5" metadata-complete="true">
03 <display-name>Banco Integracao</display-name>
04
05 <listener>
06   <listener-class>org.springframework.web.context.ContextLoaderListener
     </listener-class>
07 </listener>
```

```
08
09 <listener>
10   <listener-class>org.springframework.web.context.request
     .RequestContextListener</listener-class>
11 </listener>
12
13 <context-param>
14   <param-name>contextConfigLocation</param-name>
15   <param-value>classpath:application-context.xml</param-value>
16 </context-param>
17
18 </web-app>
```

Listagem 17. application-context.xml do componente BancoIntegracao.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xmlns:camel="http://camel.apache.org/schema/spring"
04   xmlns:context="http://www.springframework.org/schema/context"
05   xsi:schemaLocation="
06     http://www.springframework.org/schema/context
07     http://www.springframework.org/schema/context/spring-context-3.0.xsd
08     http://www.springframework.org/schema/beans
09     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
10    http://camel.apache.org/schema/spring
11    http://camel.apache.org/schema/spring/camel-spring.xsd">
12
13 <camel:camelContext id="bancoIntegracao">
14   <camel:routeBuilder ref="bancoIntegracaoRouteBuilder"/>
15 </camel:camelContext>
16
17 <context:component-scan base-package="br.com.devmedia.banco.integracao"/>
18
19 <bean id="jmsConnectionFactory" class="org.apache.activemq
  .ActiveMQConnectionFactory">
```

```
21   <property name="brokerURL" value="failover:(tcp://localhost:61616,
22     tcp://127.0.0.1:61616)?randomize=false"/>
23
24 <bean id="pooledConnectionFactory" class="org.apache.activemq.pool
  .PooledConnectionFactory" init-method="start" destroy-method="stop">
25   <property name="maxConnections" value="8"/>
26   <property name="connectionFactory" ref="jmsConnectionFactory"/>
27 </bean>
28
29 <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
30   <property name="connectionFactory" ref="pooledConnectionFactory"/>
31   <property name="concurrentConsumers" value="10"/>
32 </bean>
33
34 <bean id="activemq" class="org.apache.activemq.camel.component
  .ActiveMQComponent">
35   <property name="configuration" ref="jmsConfig"/>
36 </bean>
37
38 </beans>
```

Listagem 18. Código inicial da classe BancoIntegracaoRouteBuilder.

```
01 package br.com.devmedia.banco.integracao.rota;
02
03 import org.apache.camel.LoggingLevel;
04 import org.apache.camel.builder.RouteBuilder;
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.stereotype.Component;
07
08 import br.com.devmedia.banco.integracao.processor
  .PreparaDadosCartaoParaValidacaoProcessor;
09
10 @Component
11 public class BancoIntegracaoRouteBuilder extends RouteBuilder {
12
13   private PreparaDadosCartaoParaValidacaoProcessor preparaDadosCartaoPara
  ValidacaoProcessor;
14
15   @Override
16   public void configure() throws Exception {
17
```

```
18     from("activemq:queue:BancoIntegracao?concurrentConsumers=5")
        .id("rotaEntradaBancoIntegracao")
        .log(LoggingLevel.INFO, "[BancoIntegracao] Envia os dados do cartão
          para validação")
        .process(preparaDadosCartaoParaValidacaoProcessor)
        .log(LoggingLevel.INFO, "[BancoIntegracao] para [SistemaExternoBancario]
          Enviando os dados do cartão para validação")
        .to("activemq:queue:SistemaExternoBancario?deliveryPersistent=false")
        .end();
24
25   }
26
27   @Autowired
28   public void setPreparaDadosCartaoParaValidacaoProcessor(PreparaDadosCartaoPara
  ValidacaoProcessor preparaDadosCartaoParaValidacaoProcessor) {
29     this.preparaDadosCartaoParaValidacaoProcessor = preparaDadosCartaoPara
  ValidacaoProcessor;
30   }
31 }
```

Prosseguindo, ainda é necessário desenvolver o processor **PreparaDadosCartaoParaValidacaoProcessor**, que está declarado na linha 20 da **Listagem 18**. Portanto, implemente este processador iniciando pela criação do pacote **br.com.devmedia.banco.integracao.processor** e pela definição da classe **PreparaDadosCartaoParaValidacaoProcessor** conforme a **Listagem 19**.

Listagem 19. Código da classe PreparaDadosCartaoParaValidacaoProcessor.

```

01 package br.com.devmedia.banco.integracao.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.slf4j.Logger;
06 import org.slf4j.LoggerFactory;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.banco.integracao.comum.BancoIntegracaoComum;
10 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
11
12 @Component
13 public class PreparaDadosCartaoParaValidacaoProcessor implements Processor {
14
15     private static final Logger LOGGER = LoggerFactory.getLogger(
16         (PreparaDadosCartaoParaValidacaoProcessor.class));
17
18     @Override
19     public void process(Exchange exchange) throws Exception {
20         BancoIntegracaoComum bancoIntegracao =
21             ConversorJson.converteJsonParaObjeto(exchange.getIn().getBody()
22             .toString(), BancoIntegracaoComum.class);
23         String json = ConversorJson.converteObjetoParaJson(bancoIntegracao);
24         LOGGER.info("Json enviado:" + json);
25         exchange.getIn().setBody(json);
26     }
27 }
```

O código dessa listagem basicamente converte o POJO **BancoIntegracaoComum**, que possui os dados do CPF do titular e número do cartão, para uma **String** em formato JSON e, na sequência, transforma essa mesma **String** em uma mensagem a ser enviada ao sistema externo. Nesse trecho de código é válido ressaltar a utilização de log para registrar o que está sendo enviado para o sistema externo. Essa abordagem é uma boa prática a ser seguida em qualquer integração de sistema para que a análise e entendimento do fluxo seja mais fácil.

Terminada a codificação, os requisitos RF-02-SISBAN e RNF-02-SISBAN foram implementados com sucesso. Dessa forma, a situação atual do fluxo de negócio já implementado e da arquitetura da Solução de Pagamento Bancário podem ser conferidas nas **Figuras 1** e **2**, respectivamente.

No tópico seguinte, será codificado o requisito RF-03-SISBAN para receber a resposta do Sistema Bancário com a validação do número do cartão e do CPF do titular.

Implementando o requisito RF-03-SISBAN

A implementação do requisito RF-03-SISBAN é bem simples e consiste, basicamente, em permitir que a Solução de Pagamento Bancário receba a resposta do Sistema Bancário com o resultado da validação do CPF do titular e do número do cartão através de uma mensagem contendo a flag **isValido**. Para isso, será implementado o EIP **Request-Reply**, no qual a requisição é enviada em uma fila e a resposta é dada em outra. No cenário da Solução de Pagamento, a fila **SistemaExternoBancario** é a de envio e a **BancoIntegracaoResposta** é a de resposta do sistema externo.

Portanto, precisamos incluir no método **configure()** da classe **BancoIntegracaoRouteBuilder**, do componente **BancoIntegracao**,

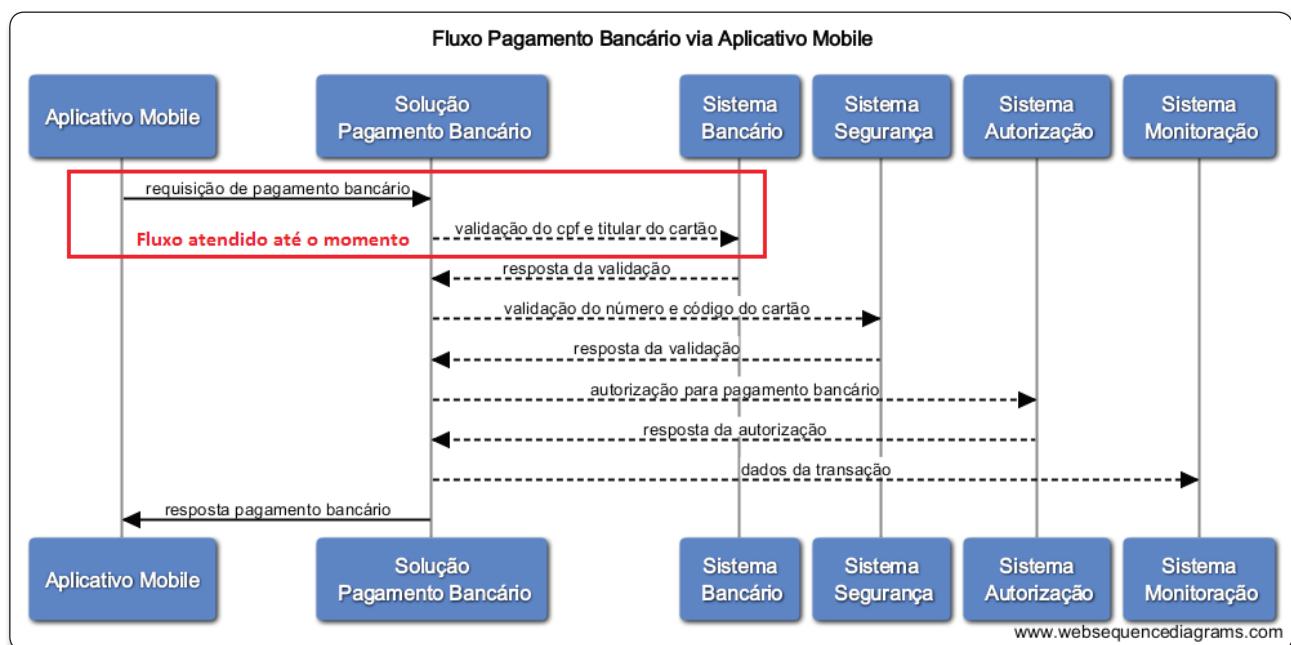


Figura 1. Primeiro e segundo fluxos de negócio atendidos.

Apache Camel: Um guia completo – Parte 3

uma nova rota para consumir mensagens da fila de resposta do Sistema Bancário. Esse novo código deve ser equivalente ao da [Listagem 20](#).

Listagem 20. Código para consumir mensagens de resposta do Sistema Bancário.

```
01 from("activemq:queue:BancoIntegracaoResposta?  
concurrentConsumers=5").id("rotaRespostaBancoIntegracao")  
02 .log(LoggingLevel.INFO, "[BancoIntegracao] Recebendo resposta da  
03 validação dos dados do cartão")  
04 .process(preparaRespostaValidacaoDadosCartaoProcessor)  
05 .setHeader("FLUXO", constant("RESPOSTA_VALIDACAO_DADOS_CARTAO"))  
06 .log(LoggingLevel.INFO, "[BancoIntegracao] para [PagamentoBancario]  
07 Enviando resposta da validação dos dados do cartão")  
08 .to("activemq:queue:PagamentoBancario?deliveryPersistent=false")  
09 .end();
```

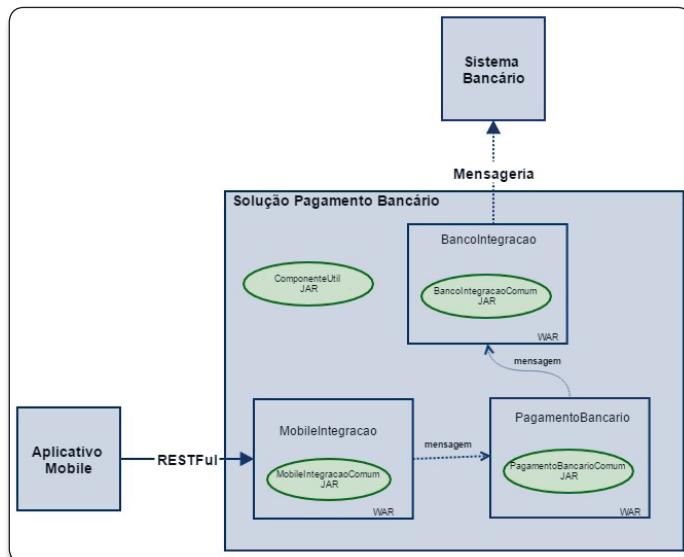


Figura 2. Novo diagrama da arquitetura da Solução de Pagamento Bancário

Verificando essa listagem, observa-se na linha 1 o código para consumir mensagens da fila **BancoIntegracaoResposta**. Em seguida, a linha 3 inclui um processor do tipo **PreparaRespostaValidacaoDadosCartaoProcessor** para tratar a mensagem de resposta do sistema externo. Adiante, a linha 4 define uma constante no header da mensagem com a chave **FLUXO** e o valor **RESPOSTA_VALIDACAO_DADOS_CARTAO**, para que o fluxo de negócio que está em execução possa ser identificado pelo próximo componente e assim rotear a mensagem ao destinatário correto. Por fim, a linha 6 faz o envio da mensagem para o **PagamentoBancario**.

Na sequência, é preciso codificar o processor **PreparaRespostaValidacaoDadosCartaoProcessor**. Sendo assim, crie uma classe com esse mesmo nome no pacote **br.com.devmedia.banco.integracao.processor** do projeto **BancoIntegracao**. Seu código é apresentado na [Listagem 21](#).

Esse código simplesmente recupera a mensagem em formato JSON e a converte para o POJO **RespostaSistemaBancario**. Logo após, converte esse mesmo POJO para JSON e o inclui no body da mensagem, para que ela seja enviada posteriormente ao próximo componente do fluxo.

Listagem 21. Código da classe **PreparaRespostaValidacaoDadosCartaoProcessor**.

```
01 package br.com.devmedia.banco.integracao.processor;  
02  
03 import org.apache.camel.Exchange;  
04 import org.apache.camel.Processor;  
05 import org.slf4j.Logger;  
06 import org.slf4j.LoggerFactory;  
07 import org.springframework.stereotype.Component;  
08  
09 import br.com.devmedia.banco.integracao.comum.RespostaSistemaBancario;  
10 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;  
11  
12 @Component  
13 public class PreparaRespostaValidacaoDadosCartaoProcessor  
14     implements Processor {  
15  
16     private static final Logger LOGGER = LoggerFactory.getLogger(  
17         (PreparaRespostaValidacaoDadosCartaoProcessor.class));  
18  
19     @Override  
20     public void process(Exchange exchange) throws Exception {  
21         LOGGER.info("Json recebido: " + exchange.getIn().getBody());  
22         RespostaSistemaBancario respostaSistemaBancario =  
23             ConversorJson.converteJsonParaObjeto(exchange.getIn()  
24             .getBody().toString(), RespostaSistemaBancario.class);  
25         String json = ConversorJson.converteObjetoParaJson  
26             (respostaSistemaBancario);  
27         exchange.getIn().setBody(json);  
28     }  
29 }
```

Listagem 22. Código da classe **RespostaSistemaBancario**.

```
01 package br.com.devmedia.banco.integracao.comum;  
02  
03 import java.io.Serializable;  
04  
05 public class RespostaSistemaBancario implements Serializable {  
06  
07     private static final long serialVersionUID = 7477511445650453586L;  
08  
09     private String cpfTitularCartao;  
10     private String numeroCartao;  
11     private Boolean valido;  
12  
13     public RespostaSistemaBancario() {}  
14  
15     // Getters e Setters dos atributos omitidos.  
16  
17     // Método toString() omitido.  
18  
19 }
```

Para finalizar a implementação do requisito, é necessário criar um POJO denominado **RespostaSistemaBancario** para representar os dados de resposta do Sistema Bancário. Deste modo, crie o pacote de nome **br.com.devmedia.banco.integracao.comum** e implemente a classe **RespostaSistemaBancario** na biblioteca **BancoIntegracaoComum** conforme o código da [Listagem 22](#).

Finalizada a criação dessa classe, o requisito foi atendido e mais um fluxo de negócio implementado. Além disso, mais um pequeno passo foi dado para a construção da solução, que agora está apta a receber as respostas do Sistema Bancário e enviar a resposta para o componente seguinte. Para um melhor entendimento da situação atual, observe as [Figuras 3 e 4](#).

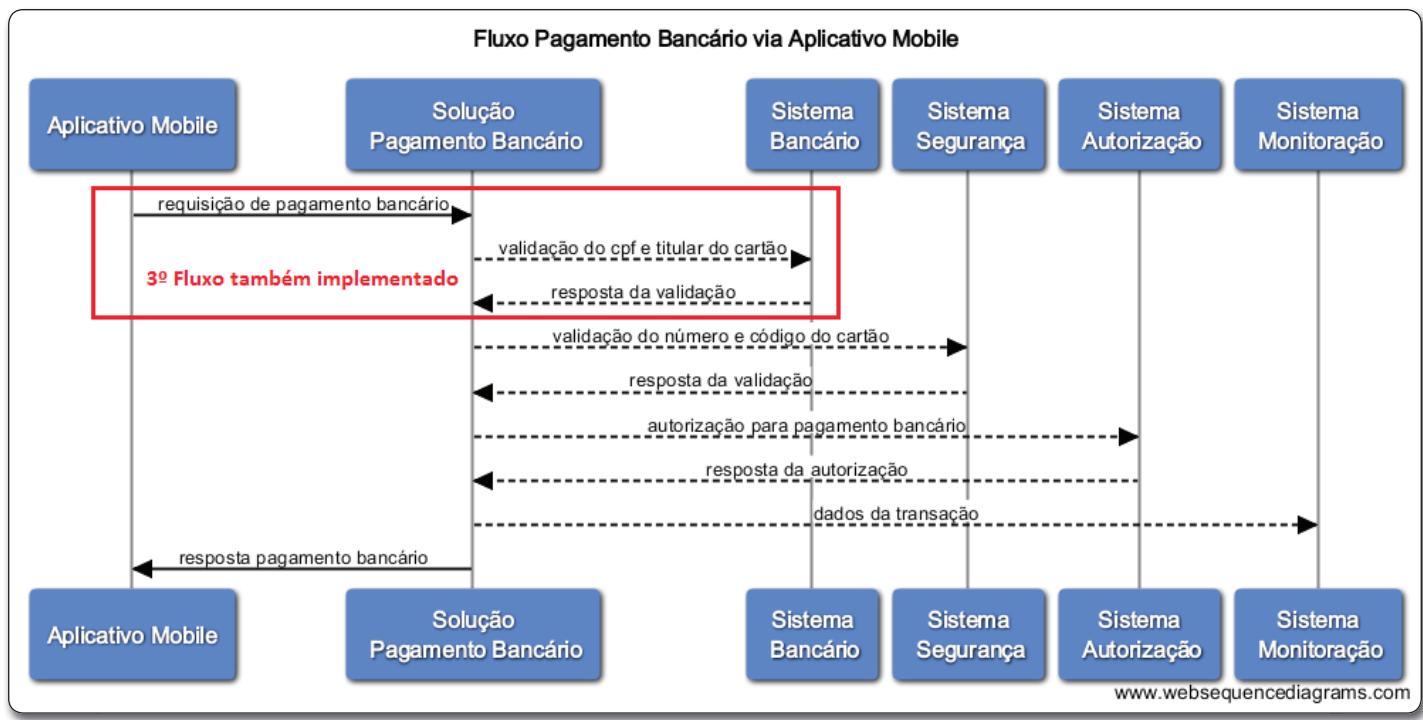


Figura 3. Terceiro fluxo de negócio implementado

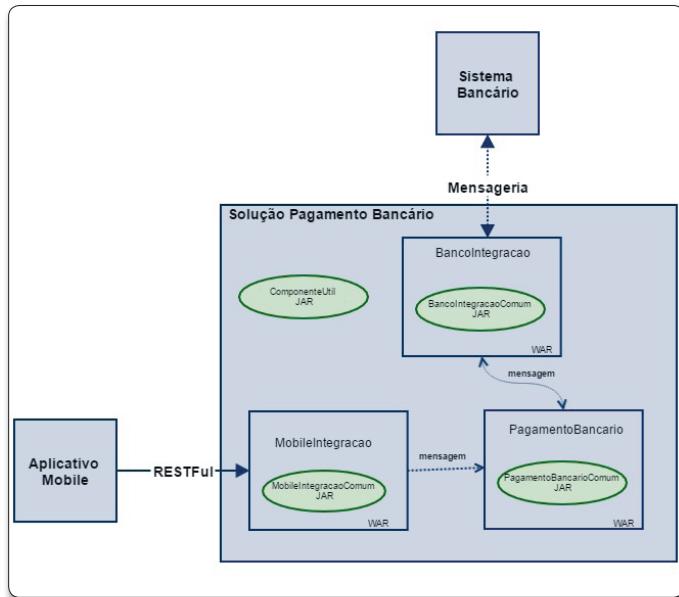


Figura 4. Solução apta a receber mensagens do Sistema Bancário e repassar a resposta

Implementando os requisitos RF-04-SISSEG e RNF-03-SISSEG

A partir de agora será iniciada a implementação do requisito RF-04-SISSEG, que corresponde a receber a resposta do sistema bancário, avaliar a flag indicativa de que os dados são válidos e, se forem, enviar, para o sistema de segurança, o número e o código de segurança do cartão para validação. Além desse requisito, o RNF-03-SISSEG também será atendido e seu objetivo basicamente é ditar que a comunicação com o sistema de segurança seja realizada via mensageria com o formato JSON.

Assim sendo, será necessário complementar o código da classe **PagamentoBancarioRouteBuilder**, do componente **PagamentoBancario**, para implementar os requisitos anteriores. As alterações são apresentadas na **Listagem 23**.

Listagem 23. Alterações no código da classe PagamentoBancarioRouteBuilder.

```

01 from("activemq:queue:PagamentoBancario?concurrentConsumers=5")
02 .id("rotaEntradaPagamentoBancario")
03 .choice()
04 .when(novaRequisicaoPagamentoBancario)
05 .log(LogLevel.INFO, "[PagamentoBancario]")
06 Nova requisição de pagamento bancário
07 .to("seda:enviaDadosCartaoValidacao")
08 .when(respostaValidacaoDadosCartao)
09 .log(LogLevel.INFO, "[PagamentoBancario]")
10 Resposta de validação dos dados do cartão
11 .to("seda:respostaValidacaoDadosCartao")
12 .endChoice()
13 .end();
14 
15 from("seda:respostaValidacaoDadosCartao")
16 .log(LogLevel.INFO, "[PagamentoBancario]")
17 Verificando resposta da validação dos dados do cartão "
18 .process(verificaRespostaValidacaoDadosCartaoProcessor)
19 .log(LogLevel.INFO, "[PagamentoBancario]")
20 Preparando código de segurança do cartão para validação"
21 .process(populaSegurancaComumProcessor)
22 .log(LogLevel.INFO, "[PagamentoBancario] para [SegurancalIntegracao]")
23 Enviando o código de segurança do cartão para validação"
24 .to("activemq:queue:SegurancalIntegracao?deliveryPersistent=false")
25 .end();

```

Apache Camel: Um guia completo – Parte 3

Baseado no código apresentado, é necessário incluir na rota de entrada mais um predicado para verificar se a mensagem que está sendo recebida pertence ao fluxo de resposta de validação do Sistema Bancário, para então seguir o fluxo correto. Portanto, crie uma nova condição conforme as linhas 6 a 8 e também um novo predicado, denominado **respostaValidacaoDadosCartao**.

Após essa etapa, codifique uma nova rota interna com o nome **respostaValidacaoDadosCartao**, conforme as linhas 12 a 19. Essa rota deve incluir dois processors. A função do primeiro é verificar a resposta de validação do Sistema Bancário e a função do segundo é preparar a mensagem para o próximo componente. Por fim, na linha 18 é realizado o envio da mensagem, a ser consumida pelo próximo componente do fluxo, através da fila **SegurancaIntegracao**.

Feito isso, crie no pacote **br.com.devmedia.pagamento.bancario**.**rota** um novo processor chamado **VerificaRespostaValidacaoDadosCartaoProcessor** (vide [Listagem 24](#)), ainda no componente **PagamentoBancario**.

Listagem 24. Código da classe VerificaRespostaValidacaoDadosCartaoProcessor.

```
01 package br.com.devmedia.pagamento.bancario.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.slf4j.Logger;
06 import org.slf4j.LoggerFactory;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.banco.integracao.comum.RespostaSistemaBancario;
10 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
11
12 @Component
13 public class VerificaRespostaValidacaoDadosCartaoProcessor implements
    Processor {
14
15     private static final Logger LOGGER = LoggerFactory.getLogger(
        VerificaRespostaValidacaoDadosCartaoProcessor.class);
16
17     @Override
18     public void process(Exchange exchange) throws Exception {
19         RespostaSistemaBancario respostaSistemaBancario =
            ConversorJson.converteJsonParaObjeto(exchange.getIn()
                .getBody().toString(), RespostaSistemaBancario.class);
20         if (respostaSistemaBancario.getisValido()) {
21             LOGGER.info("Dados do cartão são válidos");
22         }
23     }
24 }
```

O código dessa listagem transforma a resposta enviada pelo componente **BancoIntegracao**, em formato JSON, para um objeto **RespostaSistemaBancario** e verifica se a resposta foi positiva. Nesse caso, se a resposta foi positiva, apenas é logada uma mensagem de sucesso. Em um cenário real, no entanto, é necessário tratar o fluxo de insucesso.

Continuando, devemos implementar o processor **PopulaSegurancaComumProcessor**, a ser criado no pacote **br.com.devmedia.pagamento.bancario.processor**. Seu código é apresentado na [Listagem 25](#).

Listagem 25. Código da classe PopulaSegurancaComumProcessor.

```
01 package br.com.devmedia.pagamento.bancario.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.springframework.stereotype.Component;
06
07 import br.com.devmedia.pagamento.bancario.comum
    .PagamentoBancarioComum;
08 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
09 import br.com.devmedia.seguranca.integracao.comum
    .SegurancaIntegracaoComum;
10
11 @Component
12 public class PopulaSegurancaComumProcessor implements Processor {
13
14     @Override
15     public void process(Exchange exchange) throws Exception {
16         PagamentoBancarioComum pagamentoBancario = ProcessorUtil
            .getObjetoHeader(exchange, PagamentoBancarioComum.class);
17         SegurancaIntegracaoComum segurancaIntegracao =
            new SegurancaIntegracaoComum();
18         segurancaIntegracao.setCodigoSegurancaCartao(pagamentoBancario
            .getCodigoSegurancaCartao());
19         segurancaIntegracao.setNumeroCartao(pagamentoBancario
            .getNumeroCartao());
20         exchange.getIn().setBody(segurancaIntegracao);
21     }
22
23     private void populaSegurancaIntegracao(SegurancaIntegracaoComum
        segurancaIntegracao, PagamentoBancarioComum pagamentoBancario) {
24         segurancaIntegracao.setCodigoSegurancaCartao(pagamentoBancario
            .getCodigoSegurancaCartao());
25         segurancaIntegracao.setNumeroCartao(pagamentoBancario
            .getNumeroCartao());
26     }
27 }
```

Nota-se na linha 16 que o objeto incluído no header da mensagem durante a execução do fluxo no componente **MobileIntegracao** é recuperado para que seja utilizado. Na próxima linha, 17, um objeto do tipo **SegurancaIntegracaoComum** é instanciado para que o código da linha 18 popule, com base justamente neste objeto recuperado no header da mensagem, o **segurancaIntegracao**.

Por fim, as linhas 19 e 20 transformam o objeto **SegurancaIntegracaoComum** para uma **String** em formato JSON e, na sequência, transformam essa mesma **String** em uma mensagem a ser enviada ao componente seguinte do fluxo. Para que o processor **PopulaSegurancaComumProcessor** compile e funcione de forma correta, é necessário criar a biblioteca **SegurancaIntegracaoComum**, além de um POJO com o mesmo nome. A criação dessa biblioteca, juntamente com o componente **SegurancaIntegracao**, será detalhada a seguir.

Implementando os componentes SegurancaIntegracaoComum e SegurancaIntegracao

A implementação da biblioteca **SegurancaIntegracaoComum**, cuja finalidade é representar os dados que serão trafegados entre os componentes internos da solução e também com o sistema externo de segurança, deve-se iniciar com a criação de um novo projeto/módulo do tipo Java Standalone (JAR) dentro do diretório do projeto agregador.

Feito isso, crie nesse novo projeto um *pom.xml* com o código da **Listagem 26** para definir as configurações básicas da biblioteca.

Na sequência, deve-se criar o pacote **br.com.devmedia.seguranca.integracao.comum** e adicionar o POJO **SegurancaIntegracaoComum** para representar os dados a serem enviados para o sistema externo de segurança. A classe deve ser implementada em conformidade com a **Listagem 27**.

Listagem 26. Arquivo pom.xml da biblioteca SegurancaIntegracaoComum.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
02   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
03
04   <modelVersion>4.0.0</modelVersion>
05
06   <parent>
07     <groupId>br.com.devmedia</groupId>
08     <artifactId>solucao-pagamento-bancario</artifactId>
09     <version>1.0</version>
10   </parent>
11
12   <name>SegurancaIntegracaoComum</name>
13   <artifactId>Seguranca-integracao-comum</artifactId>
14   <packaging>jar</packaging>
15
16 </project>
```

Listagem 27. Código da classe SegurancaIntegracaoComum.

```
01 package br.com.devmedia.seguranca.integracao.comum;
02
03 import java.io.Serializable;
04
05 public class SegurancaIntegracaoComum implements Serializable {
06
07   private static final long serialVersionUID = -6188988876132780644L;
08
09   private String numeroCartao;
10   private String codigoSegurancaCartao;
11
12   public SegurancaIntegracaoComum() {}
13
14   // Getters e Setters dos atributos omitidos.
15
16   // Método toString() omitido.
17 }
```

Verifica-se nesse código que a classe possui os dois atributos (número e código de segurança do cartão) que serão enviados ao sistema externo de segurança. Continuando, é necessário adicionar essa nova biblioteca nas dependências do projeto **PagamentoBancario** e também na declaração de módulos do projeto agregador.

Com a biblioteca **SegurancaIntegracaoComum** implementada, é a vez da criação do novo componente, chamado de **SegurancaIntegracao**, responsável por fazer a comunicação com o Sistema de Segurança e manter concentrada todas as regras e necessidades voltadas a essa integração.

Para a implementação desse componente, é preciso criar uma estrutura de projeto Java Web (WAR) dentro do diretório do projeto agregador com o nome **SegurancaIntegracao**.

Na sequência, deve-se adicionar esse novo componente nos módulos do projeto agregador e incluir no *pom.xml* as dependências para as bibliotecas **ComponenteUtil** e **SegurancaIntegracaoComum**.

Prosseguindo, deve-se ajustar o arquivo *web.xml*, localizado no diretório *src/main/webapp/WEB-INF*, conforme a **Listagem 28**, para que a aplicação carregue (ao iniciar) as configurações do Spring, Camel e ActiveMQ.

Listagem 28. web.xml do componente SegurancaIntegracao.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5" metadata-complete="true">
03   <display-name>Segurança Integracao</display-name>
04
05   <listener>
06     <listener-class>org.springframework.web.context.ContextLoaderListener
      </listener-class>
07   </listener>
08
09   <listener>
10     <listener-class>org.springframework.web.context.request
      .RequestContextListener</listener-class>
11   </listener>
12
13   <context-param>
14     <param-name>contextConfigLocation</param-name>
15     <param-value>classpath:application-context.xml</param-value>
16   </context-param>
17
18 </web-app>
```

Além disso, é preciso incluir um novo arquivo denominado *application-context.xml* — semelhante à **Listagem 29** — no diretório *src/main/resources*, pois são através das declarações dos beans (gerenciados pelo Spring) contidas nesse arquivo que a conexão com o ActiveMQ é efetuada e o carregamento inicial das rotas do Camel é feito.

Examinando a **Listagem 29**, é possível reparar que a lógica e recursos declarados no arquivo *application-context.xml* seguem o mesmo padrão do respectivo arquivo do componente **PagamentoBancario**. O diferencial neste caso está nas linhas 14 a 16, que informam ao Camel que a classe que contém as rotas a serem iniciadas quando a aplicação carregar é a **SegurancaIntegracaoRouteBuilder**, referenciada pela tag **camel:routeBuilder** na linha 15.

Finalizada a criação dos arquivos *web.xml* e *application-context.xml*, deve-se adicionar o pacote **br.com.devmedia.seguranca.integracao.rota** e codificar a classe de rotas **SegurancaIntegracaoRouteBuilder** conforme a **Listagem 30**.

Observa-se que o código entre as linhas 18 e 23 define uma rota para receber mensagens da fila **SegurancaIntegracao**. Na linha 20, um processador do tipo **PreparaCodigoSegurancaCartaoValidacaoProcessor** é incluído para criar uma mensagem compatível com a esperada pelo Sistema de Segurança.

Apache Camel: Um guia completo – Parte 3

Listagem 29. application-context.xml do componente SegurancalIntegracao.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xmlns:camel="http://camel.apache.org/schema/spring"
05   xmlns:context="http://www.springframework.org/schema/context"
06   xsi:schemaLocation="
07     http://www.springframework.org/schema/context
08     http://www.springframework.org/schema/context/spring-context-3.0.xsd
09     http://www.springframework.org/schema/beans
10     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
11     http://camel.apache.org/schema/spring
12     http://camel.apache.org/schema/spring/camel-spring.xsd">
13
14 <camel:camelContext id="segurancalIntegracao">
15   <camel:routeBuilder ref="segurancalIntegracaoRouteBuilder"/>
16 </camel:camelContext>
17
18 <context:component-scan base-package="br.com.devmedia.seguranca
19   .integracao"/>
20 <bean id="jmsConnectionFactory" class="org.apache.activemq
21   .ActiveMQConnectionFactory">
22
23
24 <bean id="pooledConnectionFactory" class="org.apache.activemq.pool
25   .PooledConnectionFactory" init-method="start" destroy-method="stop">
26   <property name="maxConnections" value="8"/>
27   <property name="connectionFactory" ref="jmsConnectionFactory"/>
28 </bean>
29
30 <bean id="jmsConfig" class="org.apache.camel.component.jms
31   .JmsConfiguration">
32   <property name="connectionFactory" ref="pooledConnectionFactory"/>
33   <property name="concurrentConsumers" value="10"/>
34 </bean>
35
36 <bean id="activemq" class="org.apache.activemq.camel.component
37   .ActiveMQComponent">
38   <property name="configuration" ref="jmsConfig"/>
39 </bean>
40
41 </beans>
```

Listagem 30. Código inicial da classe SegurancalIntegracaoRouteBuilder.

```
01 package br.com.devmedia.seguranca.integracao.rota;
02
03 import org.apache.camel.LoggingLevel;
04 import org.apache.camel.builder.RouteBuilder;
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.stereotype.Component;
07
08 import br.com.devmedia.seguranca.integracao.processor
09   .PreparaCodigoSegurancaCartaoValidacaoProcessor;
10
11 @Component
12 public class SegurancalIntegracaoRouteBuilder extends RouteBuilder {
13
14   private PreparaCodigoSegurancaCartaoValidacaoProcessor prepara
15    CodigoSegurancaCartaoValidacaoProcessor;
16
17   @Override
18   public void configure() throws Exception {
19     from("activemq:queue:SegurancalIntegracao?concurrentConsumers=5")
20       .id("rotaEntradaSegurancalIntegracao")
21       .log(LoggingLevel.INFO, "[SegurancalIntegracao] Envia código de segurança
22         do cartão para validação")
23       .process(preparaCodigoSegurancaCartaoValidacaoProcessor)
24       .log(LoggingLevel.INFO, "[SegurancalIntegracao] para [SistemaExterno
25         SegurancaCartao] Enviando código de segurança do cartão para validação")
26       .to("activemq:queue:SistemaExternoSegurancaCartao?deliveryPersistent=
27         false")
28       .end();
29   }
30 }
```

Por fim, na linha 22 é feito o envio da mensagem para a fila **SistemaExternoSegurancaCartao**, para que o Sistema de Segurança receba a mensagem.

Para concluir, ainda é necessário desenvolver o processor **PreparaCodigoSegurancaCartaoValidacaoProcessor**. Portanto, crie o pacote **br.com.devmedia.seguranca.integracao.processor** e adicione o processor **PreparaCodigoSegurancaCartaoValidacaoProcessor** conforme o código da Listagem 31.

Essa listagem converte o objeto **SegurancaIntegracaoComum** para uma **String** em formato JSON e na sequência inclui essa mesma **String** no body da mensagem para que ela seja enviada ao sistema externo posteriormente. Com essa codificação, os requisitos RF-04-SISSEG e RNF-03-SISSEG foram atendidos e os fluxos de negócio implementados.

As **Figuras 5 e 6** apresentam os fluxos de negócio já implementados e a arquitetura atual da Solução de Pagamento Bancário.

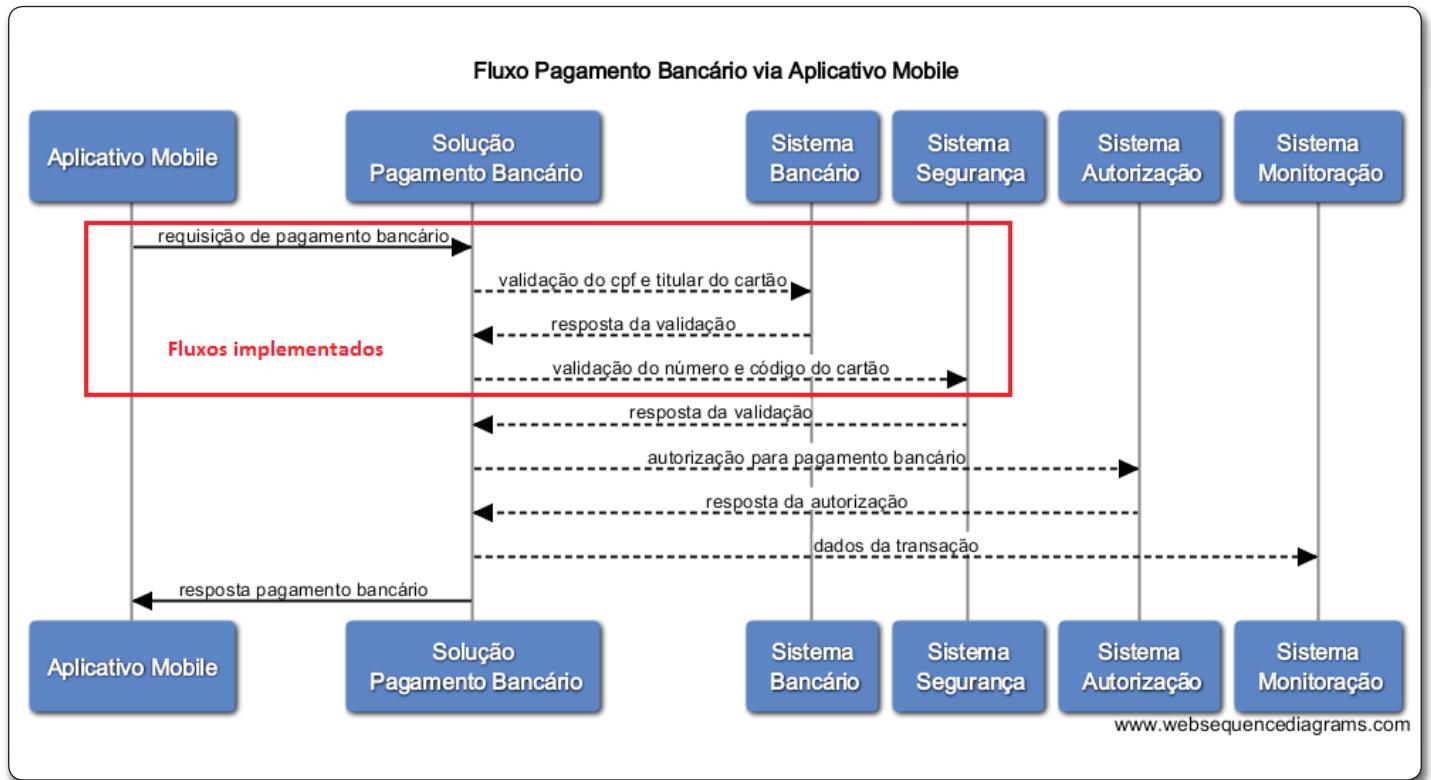


Figura 5. Fluxos de negócio já implementados

Listagem 31. Código da classe PreparaCodigoSegurancaCartaoValidacaoProcessor.

```

01 package br.com.devmedia.seguranca.integracao.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.slf4j.Logger;
06 import org.slf4j.LoggerFactory;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
10 import br.com.devmedia.seguranca.integracao.comum
    .SegurancalIntegracaoComum;
11
12 @Component
13 public class PreparaCodigoSegurancaCartaoValidacaoProcessor
    implements Processor {
14
15     private static final Logger LOGGER = LoggerFactory.getLogger(
        PreparaCodigoSegurancaCartaoValidacaoProcessor.class);
16
17     @Override
18     public void process(Exchange exchange) throws Exception {
19         SegurancalIntegracaoComum segurancalIntegracao =
            ConversorJson.converteJsonParaObjeto(exchange.getIn()
                .getBody().toString(), SegurancalIntegracaoComum.class);
20         String json = ConversorJson.converteObjetoParaJson(segurancalIntegracao);
21         LOGGER.info("Json enviado:" + json );
22         exchange.getIn().setBody(json);
23     }
24 }

```

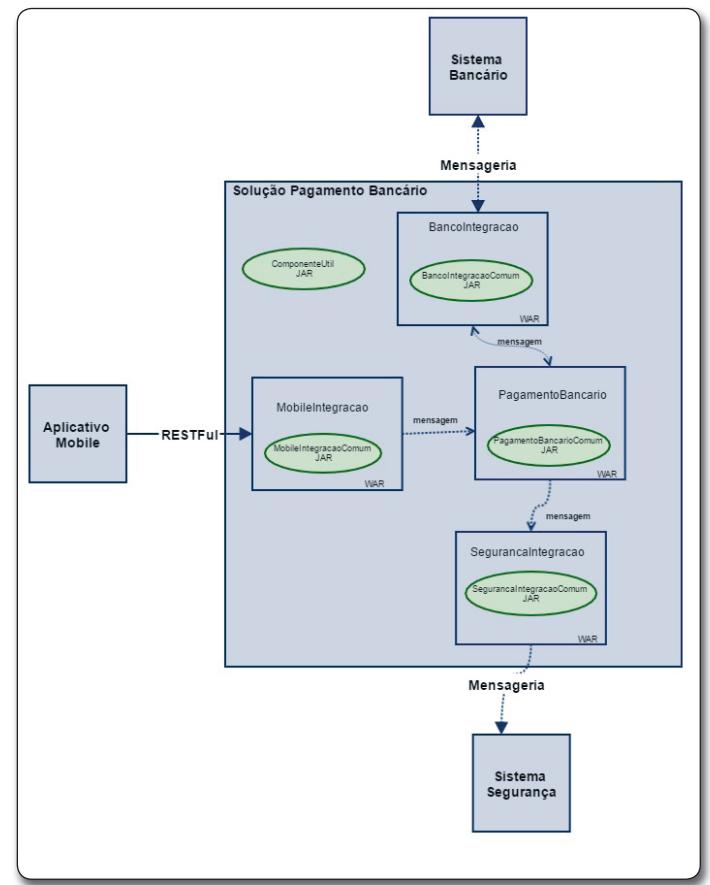


Figura 6. Arquitetura atual da Solução de Pagamento Bancário

Nesta terceira parte do artigo, continuamos com a construção e codificação da solução para integrar um aplicativo mobile a sistemas externos que são necessários para que uma transação de pagamento bancário com cartão de crédito seja realizada com sucesso.

Deste modo, foi possível demonstrar na prática os principais recursos do Apache Camel, como o roteamento de mensagens, padrões EIPs e o uso de componentes pré-existentes para a integração com outros sistemas.

Além das formas de integração apresentadas, é válido mencionar que também é possível utilizar outros tipos de integração, como arquivos, web services SOAP, WebSocket e TCP, conforme as necessidades de negócio e recursos tecnológicos. Para todas essas opções o Camel possui componentes que facilitam o trabalho do desenvolvedor, e a utilização de cada um segue o mesmo padrão dos demais componentes apresentados até aqui.

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:

Site sobre o Apache Camel.

<http://camel.apache.org>

Página com os componentes do Apache Camel.

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

Página com os EIPs implementados pelo Apache Camel.

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

Site sobre o Apache ActiveMQ.

<http://activemq.apache.org>

Site sobre os EIPs.

<http://www.enterpriseintegrationpatterns.com>

Por que os processos de teste falham?

Conheça nesse artigo alguns fatores que podem levar um processo de teste a falhar e descubra como evitar o fracasso do projeto

A busca contínua pela qualidade de software tem sido cada vez mais intensa, mas o caminho para alcançá-la tem se tornado cada vez mais difícil. Isso está acontecendo porque o avanço da tecnologia fornece continuamente novas ferramentas que podem viabilizar uma série de benefícios ao software, porém, podem surgir diferentes problemas na aplicação; por exemplo, na validação de algumas funcionalidades. Portanto, torna-se fundamental contorná-los para evitar que um software seja entregue sem atender as necessidades especificadas e satisfaça a todos os envolvidos.

Um passo fundamental nessa busca por um sistema com maior confiabilidade é fazer com que o processo de teste e o processo de desenvolvimento sejam executados paralelamente, desde o início do ciclo de vida do software.

Para viabilizar a qualidade do software, o objetivo do processo de teste deve ser bem claro, visando estruturar todos os itens que envolvem os testes em um projeto, possibilitando organizar e controlar todo o ciclo de teste e minimizar os riscos.

Essa estruturação permite uma melhor gestão dos testes, garantindo mais efetividade na redução dos erros detectados. Porém, para tornar isso factível, é necessário, além de definir de maneira clara os objetivos do teste, selecionar as técnicas de teste mais adequadas ao tipo do projeto, bem como captar profissionais treinados e qualificados para desempenharem os respectivos papéis dentro do processo.

O que ocorre hoje em dia é que fatores como a rápida evolução das ferramentas de desenvolvimento, a complexidade das aplicações desenvolvidas, a falta de uma gestão efetiva para a equipe, ou até mesmo a falta de capacitação dos profissionais envolvidos nas

Fique por dentro

São tantos os fatores que podem levar um processo de teste de software a falhar que ao longo desse artigo destacaremos o impacto negativo que alguns deles podem gerar ao projeto, assim como analisaremos o que pode ser feito para evitá-los e os benefícios que podem ser obtidos, caso eles sejam efetivamente implantados desde o início do ciclo de desenvolvimento.



Por que os processos de teste falham?

atividades, tendem a aumentar consideravelmente as falhas em um processo de teste.

Diante desse cenário, o esforço do teste de software precisa ser redobrado e contínuo para viabilizar a usabilidade, a disponibilidade e a confiabilidade do projeto em questão, independentemente da tecnologia ou metodologia utilizada no seu desenvolvimento. Além destes, outros fatores também devem ser observados:

- O controle na gestão de mudanças da aplicação;
- A segurança, desde o acesso indevido à aplicação, até a garantia da integridade do software, deve ser tratada como premissa importante para os testes;
- Os riscos inerentes ao tipo do software.

Para evitar problemas que possam levar a falhas no processo de teste, todo o planejamento deve ser minuciosamente elaborado através de um levantamento completo e detalhado sobre o ciclo de desenvolvimento. O objetivo é cercar o projeto com as principais técnicas e tipos de teste que poderão ser utilizados para evitar lacunas no planejamento que poderiam culminar em falhas ao longo da construção do software.

O problema ocorre quando o planejamento dos testes não é seguido ou não contempla efetivamente as necessidades do projeto, fazendo com que a garantia da qualidade do software fique extremamente ameaçada. Diante disso, com o intuito de evitar a iminência de um fracasso do projeto, ao longo desse artigo serão apresentados alguns motivos recorrentes que tendem a levar um processo de teste à falha, como a ausência da capacitação técnica e a deficiência no planejamento dos testes. Ademais, apresentaremos os benefícios que a correção de alguns desses motivos pode agregar ao processo.

Ausência de gerência de qualidade independente

A simples falta de uma gerência voltada exclusivamente para a qualidade de software pode ser um fator relevante para que um processo de teste falhe.

A partir do momento em que uma empresa opta por deixar o gerenciamento da qualidade à cargo de outra área, automaticamente abre mão de uma série de benefícios que podem envolver a utilização mais efetiva dos recursos físicos, recursos humanos, metodologias e técnicas.

Quando o gerenciamento de toda essa infraestrutura física, intelectual e humana é impreciso, o teste pode acabar sendo aplicado tarde no software, geralmente quando o seu desenvolvimento já está parcialmente concluído ou até mesmo finalizado.

Desse modo, por mais recursos que estejam disponíveis ou por mais ampla que seja a estrutura fornecida para execução dos testes, pode ocorrer de as atividades voltadas para o teste de software não obterem tanta atenção quanto necessária, inabilitizando um melhor gerenciamento de toda essa infraestrutura e minimizando os possíveis benefícios que poderiam ter sido obtidos para todo o ciclo de desenvolvimento.

É importante observar que ao optar por uma gerência independente, voltada para garantir a qualidade do software, significa que o processo de teste será independente do processo de desenvolvimento, porém integrado.

Contudo, a independência de uma gerência voltada para a qualidade do software não garante por si só o sucesso do processo de teste, mas agrega considerável valor ao projeto como um todo através da ênfase dada às atividades voltadas para o teste de software, o que possivelmente não acontece com o gerenciamento compartilhado de equipes. O que ocorre é que quando existe uma gerência de qualidade independente, uma série de benefícios podem ser obtidos para o projeto. Para exemplificar, destacamos:

- É possível priorizar e direcionar melhor os recursos humanos da equipe em relação às atividades de teste;
- Possibilita implantar uma metodologia específica e totalmente voltada para a realização das atividades inerentes ao processo de teste;



- Viabiliza a utilização de recursos tecnológicos visando exclusivamente a garantia da qualidade;
- Permite garantir a conformidade dos requisitos especificados de acordo com a evolução da implementação do software;
- Reduz consideravelmente os riscos relacionados a soluções não padronizadas;
- Amplia o controle na manutenção dos softwares legados;
- Possibilita gerenciar a execução e analisar os resultados obtidos com os testes;
- Permite acompanhar a evolução dos relatórios acerca da qualidade dos aplicativos testados;
- Estabelece um modelo de gestão de demandas de testes manuais e automatizados;
- Agrega valor em todo o processo de desenvolvimento.

Ausência de profissionais capacitados

Algumas empresas insistem em utilizar os profissionais já existentes na equipe de desenvolvimento para estruturar e organizar o processo de teste. Optar por essa solução não é totalmente errado, desde que haja uma prévia avaliação dos candidatos visando identificar a aptidão do profissional para exercer uma ou várias funções da área de teste.

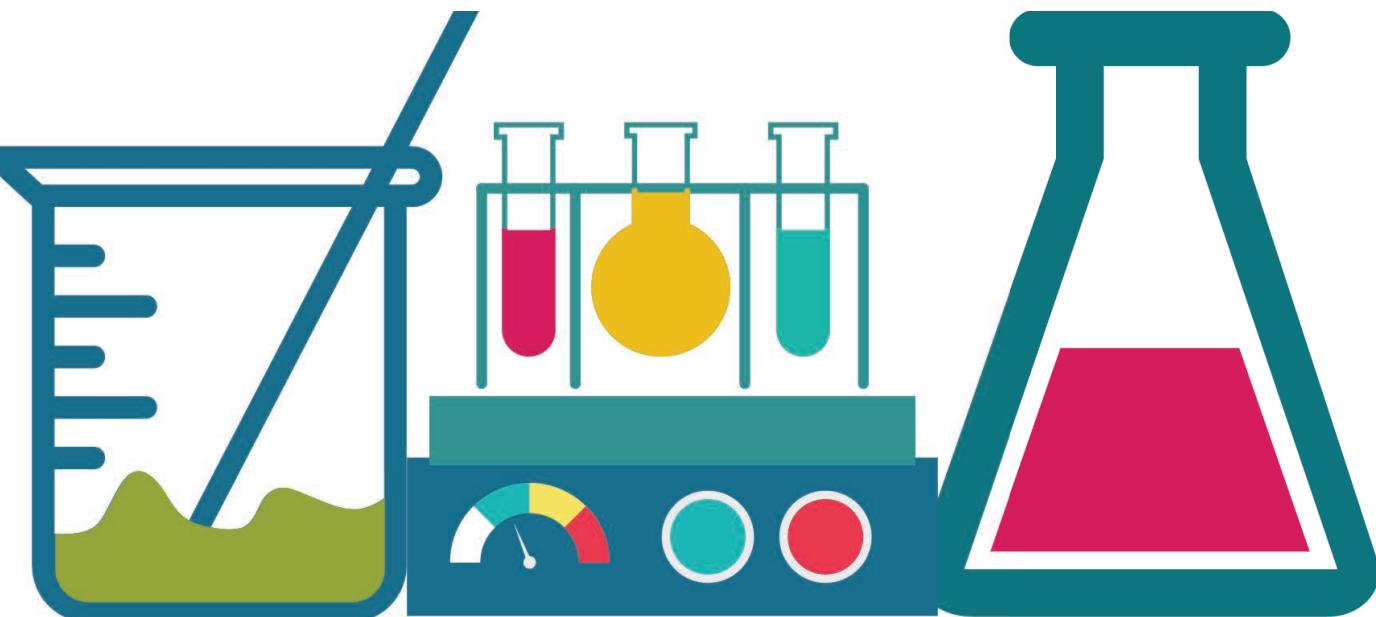
Quando um profissional não capacitado é alocado, a qualidade de todo o ciclo de desenvolvimento pode ser prejudicada. Isso ocorre porque as perspectivas de um desenvolvedor e de um testador em relação à qualidade de um software podem ser muito diferentes. Geralmente, o foco do desenvolvedor é entregar uma aplicação funcional. Já o objetivo de um testador é avaliar a solidez da aplicação que foi entregue. De uma maneira bem sucinta, é possível dizer que o profissional de teste tem um perfil crítico por natureza.

Saiba que o teste de software requer perfis com boa percepção e experiência para a execução dos diferentes tipos de testes, o que propicia mais eficiência na detecção de erros. É importante, ainda, ser claro, objetivo, conciso, bom ouvinte, bom negociador, ser crítico e, fundamentalmente, ser comprometido.

Para auxiliar na melhor distribuição dos recursos humanos envolvidos na garantia de qualidade do processo, o teste de software possui uma série de funções bem detalhadas e definidas de acordo com as atividades que englobam todo o ciclo de vida do teste.

O objetivo dessas funções consiste em organizar as responsabilidades de execução das atividades dentro do processo de teste. Nesse momento cabe ressaltar que as definições das atribuições a cada perfil geralmente são adaptadas à realidade das empresas, e, além disso, deve-se considerar as habilidades exigidas por cada um, analisados a seguir:

- **Testador:** também conhecido como Homologador ou Tester, é o profissional que tem como principal função executar os testes planejados pelo Analista de Teste, reportando defeitos quando encontrados;
- **Analista de Teste:** também intitulado Analista de Homologação ou Projetista de Teste, precisa ter um pouco mais de experiência nas atividades da área de teste, chegando a ser primordial que tenha exercido a função de Testador. É o responsável pela elaboração dos casos de teste e, em alguns casos, também põe a mão na massa para executá-los;
- **Automatizador:** conhecido também como Analista de Teste de Automação, é quem transforma os casos de teste elaborados pelo analista de teste em scripts automatizados. Além de automatizar os testes funcionais, é responsável pelos testes de desempenho, performance e carga;
- **Arquiteto de Teste:** também chamado de Engenheiro de Teste, é responsável pela criação e manutenção de toda a infraestrutura de teste, que inclui o ambiente, a arquitetura e as ferramentas;
- **Líder de Teste:** também intitulado Coordenador de Teste, é o profissional responsável pela organização dos testes. É ele quem estima os esforços, os recursos e o tempo de teste. É também quem cria e mantém o artefato plano de teste e atribui as tarefas aos demais membros da equipe; e



Por que os processos de teste falham?

- **Gerente de Teste:** é o responsável por todos os assuntos relacionados ao teste de software. Normalmente é quem define a política de teste adotada na organização, incluindo planejamento, documentação, controle e monitoramento dos testes, aquisição de ferramentas, participação em inspeções e revisões do trabalho de teste.

Conforme apresentado, a definição dos papéis é bem clara. Porém, é importante ressaltar que, apesar de não ser recomendável, não existe objeção para que uma pessoa assuma mais de um papel no projeto.

Diante desse fato, o primeiro questionamento a ser feito é: será que acumular responsabilidades em um mesmo profissional é viável? Outra questão: será que ter profissionais capacitados em teste é o suficiente para que um processo de teste não falhe? Certamente a resposta para essas perguntas é "não", afinal, evitar o fracasso do processo pode ir muito além da capacitação dos envolvidos nas atividades de teste.

Por outro lado, a qualificação dos profissionais e a experiência podem exercer grande diferença na obtenção da qualidade de um projeto, propiciando muitos benefícios, a saber:

- Uma equipe qualificada pode ser capaz de valorizar ainda mais os testes;
- Permite direcionar as atividades do processo de teste de acordo com o papel exercido por cada profissional;
- Possibilita um ganho de produtividade no projeto, em função da correta distribuição das atividades que compõem o ciclo de vida do teste;
- Garante um maior alcance da qualidade do software, obtida através da visão crítica dos profissionais de teste.

Ausência de um ambiente de teste isolado

Para testar um software, uma exigência que se torna primordial é ter um ambiente exclusivo para a realização dos testes, isto é, isolado de qualquer interferência dos demais. Esse ambiente deve contar com algumas características capazes de garantir a integridade dos testes realizados, por exemplo:

- Ser restrito à utilização da equipe de teste;
- Deve viabilizar a criação e manutenção de uma massa de testes com dados representativos;
- Deve ser mantido sempre atualizado, para viabilizar testes mais próximos à realidade da aplicação;
- Deve possuir um processamento independente, mas características similares aos ambientes de desenvolvimento e produção.

Como esperado, a ausência de um ambiente de testes isolado aumenta as chances de falha de um processo de teste. Porém, quando esse item é posto em primeiro plano, uma série de benefícios pode ser alcançada, a saber:

- Permite manter a integridade dos dados;
- Amplia o controle sobre todo o ambiente de teste;
- Mantém a massa de teste controlada;
- Garante a utilização das normas e dos padrões especificados no planejamento dos testes;
- Garante a efetividade dos resultados dos testes por não sofrer a interferência de compartilhamento do ambiente com outras áreas.

Ausência de procedimentos de testes automatizados

É sabido que não é viável ter um processo de teste 100% automatizado, pois nem todas as funcionalidades são suscetíveis à automação, seja pela viabilidade técnica ou pelo alto custo. Porém, também não é aconselhável deixar a garantia da qualidade de software totalmente sob responsabilidade dos testes manuais.

Esse tipo de teste requer um grande esforço de repetição sempre que uma alteração é realizada no software, o que aumenta consideravelmente a probabilidade de novos erros serem inseridos na aplicação. O ideal, portanto, é achar um ponto de equilíbrio que seja viável para o projeto, mesclando esses dois tipos de teste.

Neste ponto é válido ressaltar que quando a execução dos testes de um projeto é feita de forma manual e não há um planejamento bem definido, ou até mesmo quando falta expertise ao responsável pela execução, torna-se delicado garantir que todas as possibilidades de teste sejam efetivamente verificadas e validadas.

Nota

A função do planejamento de testes é definir detalhadamente todos os artefatos que serão gerados e executados ao longo do processo de teste, por exemplo, o plano de teste, o caso de teste e o roteiro de teste.

Com o intuito de disponibilizar a automação de um processo de teste de software, o primeiro passo é partir do pressuposto de que as funcionalidades mais factíveis à automação são aquelas que envolvem a execução de tarefas repetitivas e cansativas, facilmente suscetíveis a erros, ou aquelas mais complexas de serem efetuadas.



Para viabilizar a automação, as ferramentas mais comuns são as que utilizam a técnica *Record & Play*. Essa opção consiste em gravar as ações realizadas pelo usuário enquanto ele interage com a interface gráfica, convertendo-as em scripts de teste para posterior execução.

Além dessas, existem ferramentas que têm como objetivo a execução de outras técnicas ou tipos de teste de maneira automatizada, por exemplo: ferramentas que permitem a simulação de invasões e ataques ao software. Há, também, aquelas que são voltadas para analisar e auditar o código da aplicação, ferramentas para viabilizar a geração de uma grande massa de testes, entre outras.

Entretanto, por envolver custos mais altos e normalmente ser um investimento de longo prazo, se comparado com os testes manuais, a automação de testes geralmente é deixada em segundo plano. Outros fatores que influenciam essa decisão são o custo de desenvolvimento e manutenção dos scripts de automação, que pode ser alto e requerer mais tempo, principalmente quando há mudanças nos requisitos do software, e a curva de aprendizado, que também é alta.

A complexidade técnica do projeto também pode pesar na opção da empresa em definir se irá ou não automatizar o processo de teste. Além desses fatores, a falta de um processo de desenvolvimento bem definido e a ausência de apoio de toda a equipe envolvida no ciclo de vida do software poderão tornar inviável a opção pela automação das atividades de teste.

Ainda relacionado isso, uma questão relevante a ser levantada pela empresa e pelo projeto é: será que é viável utilizar diferentes tipos de ferramentas de automação? A resposta dependerá diretamente do tipo do projeto, do valor disponível para investimento, do prazo, dos profissionais envolvidos, entre outros fatores.

Mais um questionamento: será que a ausência de procedimentos de testes automatizados pode fazer um processo de teste falhar? Sim, principalmente se também estiverem ausentes outros itens apresentados ao longo do artigo. Ademais, se for viabilizada a implantação de testes automatizados, uma série de vantagens pode ser obtida, a saber:

- Propicia que a alta produtividade e a qualidade caminhem juntas;
- Aumenta da confiabilidade na qualidade do processo de teste;
- Amplitude da abrangência da cobertura dos testes;
- O retorno rápido dos resultados possibilita que as falhas sejam encontradas mais rapidamente, agilizando o processo de correção dos erros;
- Reduz significativamente o tempo necessário para executar um ciclo de testes. A automação pode ser capaz de executar centenas ou até mesmo milhares de vezes mais rápido;
- A maioria das ferramentas possibilita a reutilização dos scripts de teste em diferentes versões do software;
- A automação permite programar testes mais sofisticados;
- E pode viabilizar também a execução de testes mais complexos, difíceis de reproduzir manualmente.

Qualidade aplicada tarde

Quando dizemos que o teste de software deve caminhar juntamente com o ciclo de desenvolvimento, não é apenas um simples clichê. Essa é uma necessidade básica para que a busca contínua pela qualidade alcance o objetivo esperado para o projeto.

Por mais que algumas ferramentas ou técnicas possam ser utilizadas para a execução das atividades de teste, dependendo do momento em que o processo de teste for iniciado ao longo do desenvolvimento da aplicação, ele estará altamente suscetível a falhar em função do início tardio.

O que ocorre é que quando a empresa opta por amadurecer o processo de desenvolvimento e somente iniciar os testes quando há alguma parte do software concluída ou o desenvolvimento já foi finalizado, provavelmente será demandado mais tempo para corrigir os problemas identificados e consequentemente haverá um custo maior do que se teria se esses fossem corrigidos logo que inseridos no código.

Deste modo, o teste aplicado tarde, na maioria das vezes, não gera um produto satisfatório e possivelmente irá ao encontro da Regra 10 de Myers, que indica que quanto mais tarde é identificado um defeito, maior será o custo de sua correção.



Por que os processos de teste falham?

Nota

O americano Glenford James Myers publicou em 1979 uma das mais importantes obras da área de Teste: "The Art of Software Testing" ("A arte de teste de software"). Nesse livro foram apresentados conceitos que movimentaram profundamente o ciclo de desenvolvimento de software, definindo como o Teste de Software pode ser capaz de reduzir os custos de um projeto. Entre tantas contribuições, uma das mais importantes para área foi a chamada "Regra 10 de Myers".

Por sua vez, o que aconteceria caso o teste fosse antecipado e efetivamente iniciado juntamente com o ciclo de desenvolvimento? Será que antecipar os testes garante que o processo de teste não irá falhar? Não há como garantir que o êxito seja alcançado no projeto somente em função desse pilar ser implantado, porém uma série de benefícios pode ser obtida, a saber:

- A organização dos testes no início do ciclo de desenvolvimento possibilita saber o nível de qualidade do software durante todo o período de execução dos testes;
- É possível antecipar o impacto que os erros podem gerar na aplicação como um todo;
- Permite a tomada de decisões de acordo com o status atual da qualidade do software;
- Considerável aumento da confiança do cliente.

Deficiência no planejamento dos testes

O ciclo de vida de um processo de teste consiste em uma série de etapas dependentes que visam estruturar as atividades e definir como os testes serão conduzidos no projeto.

Essas etapas podem variar de acordo com a metodologia utilizada e cada uma delas tem um tempo estimado de duração. Por exemplo, o planejamento, fase inicial do processo de teste, geralmente é responsável pela utilização de 10% do tempo destinado aos testes.

Lembre-se que é no planejamento que especificamos o objetivo dos testes, registrado no artefato plano de teste. Além disso, essa atividade também é responsável pela identificação das técnicas de teste que serão aplicadas ao projeto, assim como:

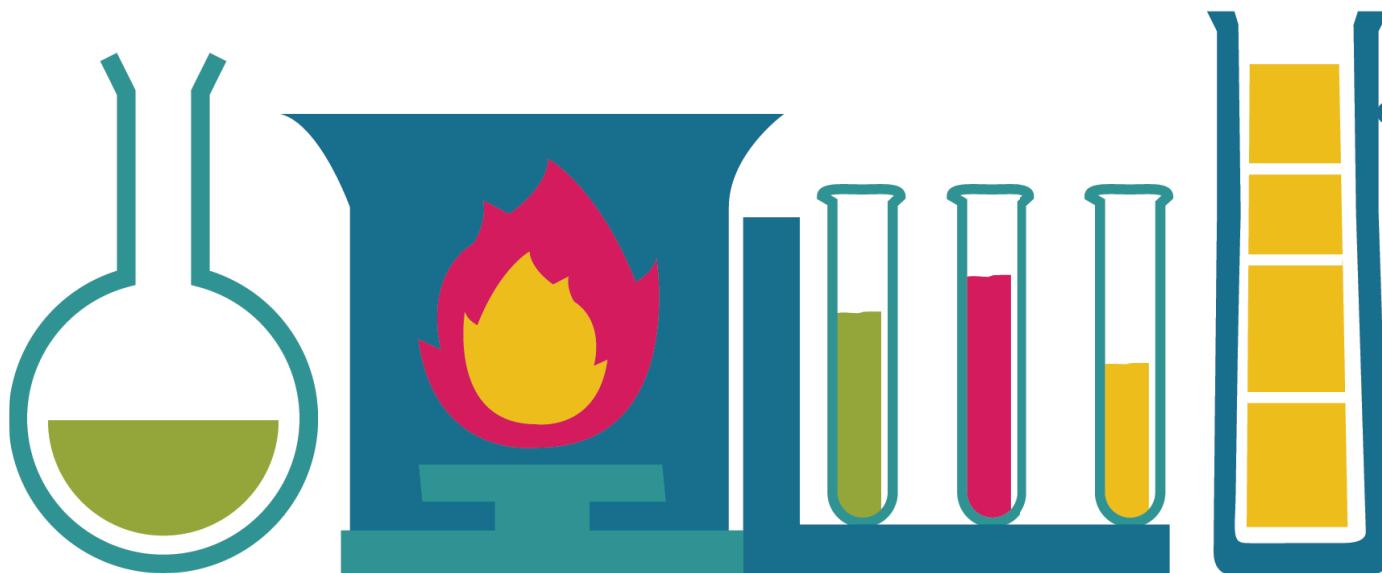
- Definição do escopo dos testes;
- Definição dos métodos e técnicas de teste;
- Planejamento das atividades de teste;
- Planejamento do ambiente de teste;
- Designação dos recursos envolvidos no teste;
- Definição dos riscos do projeto de teste; e
- Definição das métricas para monitorar e controlar os testes.

O planejamento gera como principal artefato o plano de teste, um documento que define o nível de cobertura, completo ou reduzido, que deverá ser alcançado pelos testes. Esse documento tem como finalidade fornecer uma visão geral do projeto, contendo as diretrizes que possibilitarão a execução do processo de teste. Além disso, é ele o responsável pela definição de todo o escopo dos testes, certificando que os mesmos possam ser repetidos e controlados.

Esse artefato representa todo o planejamento para a execução do teste, incluindo os recursos (pessoas, software, hardware), estratégias, cronograma das atividades e as funcionalidades que serão testadas.

Mas será que existe o risco de o processo de teste falhar caso haja alguma deficiência no planejamento dos testes do projeto? A resposta, obviamente, é sim. O planejamento é a etapa inicial e pode ser considerada como o coração do processo de teste. É nele que ficam contidas todas as informações inerentes aos testes que deverão ser realizados. Se existem informações conflitantes ou incompletas, o teste provavelmente irá falhar. Ademais, por mais minucioso que seja um planejamento, problemas podem surgir.

Logo, se o planejamento está incompleto, consequentemente o teste terá alguma parte importante comprometida, inviabilizando uma maior cobertura e/ou uma maior eficácia na busca pela garantia da qualidade. Em resumo, é possível afirmar que se o planejamento começa corretamente, as demais etapas do processo de teste tendem a se manter mais consistentes.



Segurança da aplicação deixada em segundo plano

Segundo o dicionário, segurança é uma qualidade ou condição de quem ou do que está livre de perigos, incertezas, assegurado de danos e riscos eventuais; situação em que nada há a temer. Quando se trata de um processo de teste de software, a segurança é representada através do teste de segurança.

Esse tipo de teste tem como meta garantir que as premissas de segurança da aplicação estejam funcionando exatamente como especificado.

Quando deixada em segundo plano, a segurança de um software propicia o surgimento de uma série de problemas, possibilitando as mais diversas tentativas ilegais de acesso. Essas brechas na segurança da aplicação podem ser identificadas através de diferentes maneiras, por exemplo:

- Por meio da utilização de metodologias ou frameworks que permitem avaliar possíveis lacunas na segurança, viabilizando inclusive avaliar o nível de cobertura dos testes;
- Através da verificação e validação de cada um dos itens recomendados pelas metodologias ISO/IEC-15408, OSSTMM, NIST e OWASP, por exemplo, na busca por possíveis falhas de segurança.

Nota

Com grande reconhecimento internacional, o OWASP (Open Web Application Security Project) é uma documentação gratuita e aberta a qualquer pessoa interessada em melhorar a segurança do seu software. Ela propõe um framework com atividades a serem executadas ao longo do ciclo de vida do desenvolvimento, tendo como objetivo a segurança da aplicação.

Quando bem elaborado, o teste de segurança possibilita que muitas vulnerabilidades sejam sanadas, agregando confidencialidade, integridade e disponibilidade das informações tratadas pelo sistema. Ademais, viabiliza a identificação precoce de brechas no software, o que proporciona uma maior confiança para a aplicação, independentemente da realidade do projeto e da equipe.

Foco apenas em testes progressivos

Os testes progressivos são realizados levando em consideração apenas as atualizações ou implementações desenvolvidas. Assim, se houve a correção de um erro na aplicação, por exemplo, o teste ocorre somente naquela funcionalidade que foi modificada ou implementada.

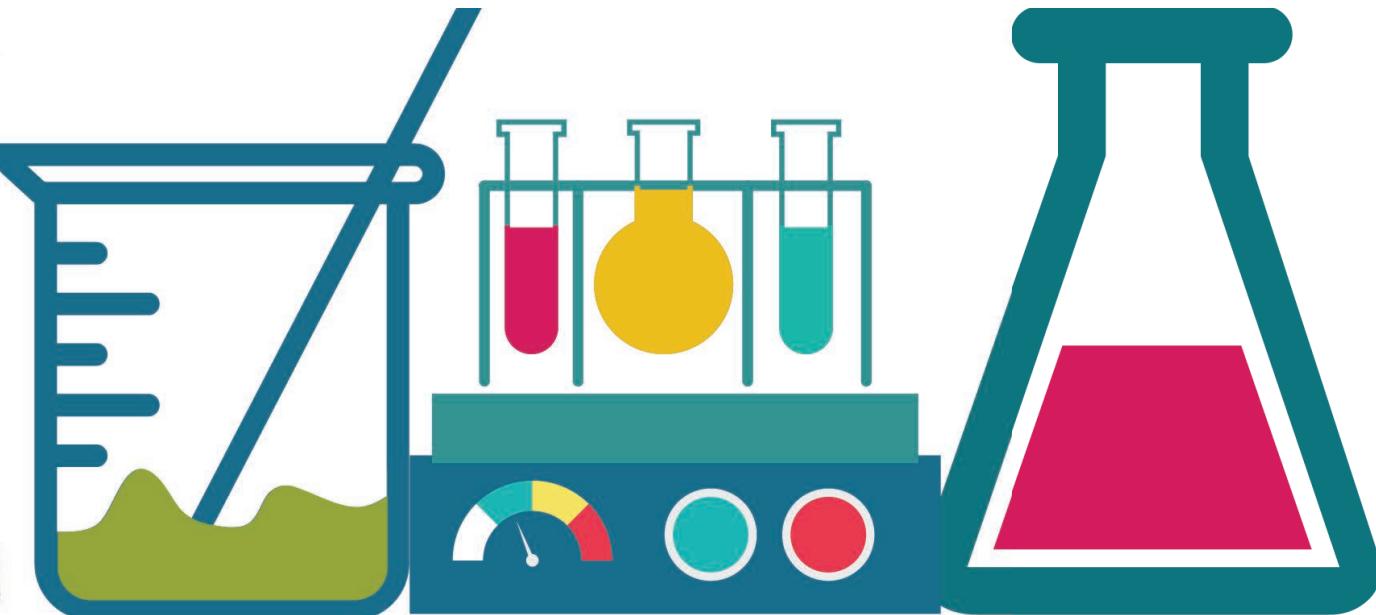
Executar esse tipo de teste pode demandar menos tempo, porém aumenta significativamente os riscos de a aplicação falhar, afinal, mudanças no código podem inserir novos defeitos em funcionalidades já validadas. Dito isso, para que a equipe de teste consiga detectar os erros gerados em função de uma atualização ou implementação, é preciso executar um teste mais completo.

Portanto, o risco de o processo de teste falhar caso o foco seja dado apenas à execução de testes progressivos torna-se altíssimo. Isso ocorre porque não há garantia do correto funcionamento do que já havia sido implementado e validado. Diante disso, para evitar a falha do processo, o primeiro passo é executar testes de regressão.

Esse tipo de teste deve ser executado sempre que uma nova versão do software for disponibilizada, ou mesmo quando surgir a necessidade de executar um novo ciclo de testes. O objetivo da regressão é eliminar as chances de falhas na aplicação em função de mudanças no código, garantindo que os requisitos se mantenham de acordo com o que foi especificado e acordado com o cliente.

Quando o teste de regressão é realizado é possível assegurar pelo menos que novos erros não foram inseridos na aplicação, o que não dispensa a realização de outros testes, mas provavelmente já propicia uma maior confiabilidade.

Enfim, executar testes progressivos para obter respostas imediatas acerca de uma nova implantação ou correção no software é, sim, algo válido. Porém, é importante ressaltar que a execução desses nunca deve vir sozinha, ou seja, deve sempre vir acompanhada pelo menos da execução de um conjunto de testes de regressão.



Por que os processos de teste falham?

Riscos não planejados

Em um processo de testes, analisar um risco tem como objetivo identificar fatores que podem gerar algum dano ao software. No entanto, será que o não planejamento dos riscos pode fazer o processo de teste falhar? Dependendo da complexidade do projeto, do tamanho da organização, a resposta mais provável é sim. Isso porque um risco pode ocasionar grandes perdas, tanto para o projeto quanto para a empresa.

Para realizar a análise de riscos, o ideal é listar as probabilidades de ameaça à qualidade do software antes mesmo de avaliar a documentação dos testes e se o plano de teste está sendo seguido, para, a partir daí, identificar os riscos associados ao projeto de teste, avaliá-los e documentá-los. Depois disso, os riscos devem ser priorizados de acordo com a probabilidade de ocorrência e o potencial impacto para o processo de teste.

Apenas para deixar mais claro, quando bem implementada, a análise de riscos pode auxiliar bastante na garantia da qualidade, o que acarreta em inúmeros benefícios à aplicação, por exemplo:

- A alocação e realocação de esforço para os testes é baseada na lista de probabilidade de problemas levantados durante a fase de análise e identificação dos riscos;
- É possível avaliar as consequências em caso de falha do software;
- O esforço não é desperdiçado em funções não críticas ou de baixo risco;
- Os testes tornam-se proativos, viabilizando conhecer com antecedência o que pode dar errado no software.

Ausência de um modelo corporativo de qualidade

Esse fator está diretamente relacionado à ausência de uma gerência de qualidade específica para a equipe de testes e isso tende a fazer com que cada projeto encontre sua própria maneira de estruturar o processo de garantia da qualidade.

É importante frisar que cada projeto possui suas peculiaridades, mas um modelo corporativo de qualidade possibilita que um padrão global seja adotado em todos eles. Isso evita discrepâncias como a existência de projetos em uma mesma empresa que mantêm a segurança do software em primeiro plano e projetos que sequer executam esse tipo de teste.

A ausência de um modelo corporativo de qualidade implica diretamente na falta de padrões, fazendo com que cada etapa

do projeto seja gerenciada de uma maneira diferente, correndo o sério risco de não se tornar operacional. Para evitar esse cenário, e consequentemente a falha do processo de teste, é preciso que alguns itens sejam implementados, por exemplo:

- Criação de um modelo corporativo de qualidade que seja efetivamente utilizado em todos os projetos;
- Planejamento corporativo das atividades de teste;
- Capacitação dos profissionais da equipe de teste;
- Compartilhamento contínuo dos conceitos que serão exigidos ao longo das etapas do processo de teste.

Transferir o planejamento dos testes ao analista de sistemas

Diretamente ligado a outro fator já apresentado, veja o tópico "Ausência de profissionais capacitados", este tópico tem como objetivo apresentar alguns motivos para evitar que um analista de sistemas seja o responsável pela elaboração do planejamento de testes, do mesmo modo que não é recomendado que profissionais sem expertise executem as atividades do processo de teste.

Quando um profissional de desenvolvimento se responsabiliza pela estruturação e organização dos testes, a eficiência e eficácia desses pode ser comprometida, possibilitando inclusive o desgaste do processo como um todo, em função de um esforço ser aplicado e não necessariamente viabilizar a garantia da qualidade.

Isso ocorre porque o foco do analista de sistemas normalmente está voltado para o desenvolvimento, assim como para compreender as exigências especificadas pela área de requisitos e as disponibilidades ou restrições tecnológicas que farão com que o software se molde às necessidades do cliente. Entre as atribuições desse analista, podemos citar:

- Avaliar a documentação especificada para o projeto, visando evitar falhas na implementação;
- Atentar à qualidade da solução implementada, garantindo a flexibilidade para suportar possíveis mudanças;
- Acompanhar as evoluções tecnológicas para manter o produto novo tanto tecnologicamente quanto conceitualmente.

Enfim, o analista de sistemas tem uma série de preocupações e funções que tornam impraticável a absorção de novas atribuições. Desse modo, evite incluir o planejamento dos testes às suas atividades.



Acesso restrito do analista de testes ao software

A partir do momento em que o profissional de teste não possui acesso ao software a ser testado, fica comprometida toda a qualidade da aplicação. Essa restrição pode ocorrer, por exemplo, quando questões contratuais inviabilizam ou limitam o acesso do analista de teste à aplicação.

Quando a equipe de teste pode utilizar o software é possível identificar regras de negócio ou exceções que não foram implementadas, campos obrigatórios que se tornaram dispensáveis, interfaces que foram parcialmente desenvolvidas, brechas na segurança da aplicação, assim como permite a execução de diferentes tipos e técnicas de teste.

Além das vantagens apresentadas, ao acolher o acesso do profissional de teste ao software, outros benefícios podem ser obtidos:

- Antecipação da detecção de inconformidades entre os requisitos e o software;
- Avaliação gradual das implementações do software;
- Melhoria na massa de teste, que passa a ser mais voltada para as necessidades da aplicação;
- Execução completa do planejamento dos testes.

Pressão

Ao tratar de teste de software, a pressão está relacionada à redução dos prazos pré-estabelecidos para a execução das atividades, sejam elas as mais fundamentais para a garantia da qualidade do projeto ou não. A pressão também pode ter relação com a entrega de uma aplicação, principalmente quando são exigidos que os objetivos definidos para a qualidade do software sejam mantidos, sem que necessariamente todo o planejamento dos testes tenha sido executado.

Apresentamos, então, o último pilar que pode levar um processo de teste a falhar: a intensa e rotineira pressão sofrida pela equipe de teste.

Por diferentes motivos, seja em função do contínuo avanço das tecnologias de desenvolvimento de software, seja em função da crescente demanda por sistemas com mais qualidade, as empresas de desenvolvimento consentem cada vez mais com a pressão, que envolve, por exemplo, a falta de tempo hábil para desenvolver novos softwares ou melhorar os já existentes, redução de prazos pré-estabelecidos, mudanças de cenário, de escopo, entre outros, e vêm aumentando consideravelmente nos últimos anos.

Nesse contexto, a área que sofre mais impacto geralmente é a de teste. Mas, por que isso acontece? O que ocorre é que se o projeto tem um prazo reduzido ou já curto desde a definição do seu escopo e algumas das etapas que antecedem os testes atrasem, serão os testes que irão arcar com essa diferença de tempo.

No entanto, note que esse problema depende também das escolhas acerca da metodologia de desenvolvimento adotada no projeto, bem como das técnicas de teste utilizadas e, ainda, se essas são iniciadas ou não juntamente com o ciclo de desenvolvimento. Neste ponto é válido ressaltar que algumas dessas opções podem requerer uma aplicação ou parte dela já desenvolvida para viabilizar os testes e a validação do software.

O certo é que com menos tempo para a execução das atividades planejadas a qualidade possivelmente será impactada e poderá levar o processo de teste a falhar. Para evitar que isso aconteça, é possível adotar algumas ações que podem colaborar para amenizar a pressão sofrida pela equipe de teste. São elas:

- Viabilizar continuamente a antecipação dos testes;
- Analisar os riscos;
- Conduzir de perto a gestão de defeitos, cobrando ininterruptamente as correções propostas;
- Implantar, acompanhar e incluir os artefatos provenientes da gestão de mudanças no planejamento dos testes; e
- Compartilhar responsabilidades entre todos os envolvidos no ciclo de vida de desenvolvimento de software.

Um processo de teste pode falhar por inúmeros motivos. No entanto, é possível reverter esse cenário quando as razões que o levam ao fracasso são analisadas durante o seu planejamento. Isso permite uma correta implementação do processo e viabiliza a garantia da qualidade do software, assim como agrupa confiança e credibilidade à aplicação.

Para isso, é importante salientar que a implantação do processo de teste deve ser minuciosamente estudada, evitando basear as atividades desse processo em poucos pilares, pois isso pode não ser suficiente para obter os resultados desejados.



Por que os processos de teste falham?

É valido frisar, ainda, que é preciso definir como será a utilização dos recursos disponíveis, adequando-os de acordo com a complexidade do projeto, as exigências do cliente e o tamanho da equipe.

Avaliados os pontos específicos do projeto e da empresa, implementar alguns benefícios apresentados ao longo desse artigo poderá prevenir que certos problemas sejam os responsáveis pelo fracasso dos testes, mesmo quando há um processo e um planejamento bem definidos, o ambiente de teste configurado e os profissionais capacitados alocados para cada atividade.

Para concluir, saiba que ao adotar esses passos é possível evitar que um processo de teste falhe. Porém, deve-se advertir que o processo também precisa ser revisto continuamente, o que possibilitará um controle cada vez maior acerca de todas as atividades de teste, levando a resultados mais confiáveis e eficazes.

Autora



Renata Eliza

renataeliza@gmail.com - @RenataEliza



Atua na área de Teste de Software há mais de dez anos. Tecnóloga em Processamento de Dados, MBA em Teste de Software e ISTQB Certified Tester Foundation Level. Mantém o blog www.asespecialistas.com.

Livros e Links:

BARTIÉ, Alexandre. Garantia da Qualidade de Software, Rio de Janeiro: Campus, 2002.

RIOS, Emerson; MOREIRA, Trayahú. Teste de Software. 2.ed. São Paulo: Alta Book, 2006.

Artigo sobre ambiente de teste.

<http://www.tiespecialistas.com.br/2015/12/ambiente-de-teste>

Monografia sobre automação de testes em softwares e sites web

<http://www.esab.edu.br/wp-content/uploads/monografias/peron-rezende-de-sousa-es.pdf>

Artigo sobre a gestão da qualidade.

http://www2.ifma.edu.br/proen/arquivos/artigos.php/gestao_da_qualidade.pdf

Apresentação sobre garantia da qualidade de software.

<http://pt.slideshare.net/AlexandreBartie/XZone-Garantia-da-Qualidade-de-Software>

Aula que aborda o tema qualidade de software.

http://www.proftoninho.com/Docs/QS_aula_01.pdf

Artigo sobre qualidade de software.

<http://xa.yimg.com/kq/groups/46090643/1016566756/name/Aula+Incial+QS.pdf>



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