



Edição 155 :: R\$ 14,90

 DEVMEDIA

Persistência de dados: JPA ou Spring Data JPA?

Conheça as diferenças entre a especificação e o módulo Spring

Apache Camel: Um guia completo – Parte 2

Domine o framework na prática  
através de um exemplo real

# MACHINE LEARNING NA PRÁTICA

Recomendação de  
conteúdo com Mahout  
e com Hadoop



Java EE: Introdução ao CDI  
Reduza o acoplamento e  
melhore a arquitetura



# MVP

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

APlique esse investimento  
na sua carreira...

E mostre ao mercado  
quanto você vale!

CONFIRA TODO O MATERIAL  
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS  
**R\$ 69,90\*** mensais

\*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR  
MAIS DO MERCADO!

 **DEVMEDIA**

## EXPEDIENTE

### Editor

Eduardo Spínola ([eduspinola@gmail.com](mailto:eduspinola@gmail.com))

**Consultora Técnica** Anny Caroline ([annycarolinegnr@gmail.com](mailto: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](http://www.devmedia.com.br/central), ou se preferir entre em contato conosco através do telefone 21 3382-5038.

### Publicidade

[publicidade@devmedia.com.br](mailto:publicidade@devmedia.com.br) – 21 3382-5038

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

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

### Fale com o Editor!

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

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



**EDUARDO OLIVEIRA SPÍNOLA**

[eduspinola.wordpress.com](http://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](http://www.devmedia.com.br/forum)

# Sumário

Destaque – Reflexão

Conteúdo do tipo Boas Práticas

## 06 – Persistência de dados: JPA ou Spring Data JPA?

[ Alexandre Gama ]

Destaque – Vanguarda

Artigo sobre Novidades

## 16 – Introdução à recomendação de conteúdo com Apache Mahout e Hadoop

[ Everton Gago ]

## 28 – Contextos e Injeção de Dependência (CDI) para Java EE

[ Daniel Monteiro ]

Artigo no estilo Curso

## 39 – Apache Camel: Um guia completo – Parte 2

[ Rodrigo Cunha Santana ]

# Programador Java: Por onde começar?

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



**DEVMEDIA**

# Persistência de dados: JPA ou Spring Data JPA?

Conheça as diferenças entre a especificação JPA e o módulo Spring Data JPA e facilite o seu dia a dia na implementação de funcionalidades relacionadas à persistência de dados

Durante longos anos, nós, desenvolvedores, tivemos que lidar com as dificuldades de mapeamento e manipulação de objetos em Java para as suas respectivas representações em um banco de dados, registros em uma tabela. O mundo relacional do banco de dados é incompatível com o mundo orientado a objetos porque eles lidam com os dados de maneiras distintas. Enquanto em Java os registros são representados por objetos que possuem comportamento, no banco de dados os registros são representados de forma textual e sem comportamento.

Para diminuir a incompatibilidade entre eles e permitir que objetos sejam representados no banco de dados, foi criada a especificação Java Database Connectivity API, o famoso JDBC. Apesar dos benefícios conquistados, também vieram muitas dificuldades, como problemas de conexão com bancos de dados diversos, casting excessivo de objetos, código complexo para ações simples, ausência de cache de objetos, etc.

Para solucionar os problemas originados com o JDBC, diversos frameworks foram criados e ganharam espaço na comunidade. Um dos mais relevantes foi o **Hibernate**, que rapidamente se tornou padrão nos pequenos e grandes projetos. Infelizmente essas soluções, conhecidas como frameworks ORM, não seguiam nenhum padrão de desenvolvimento e traziam algumas dificuldades, como: pouco reaproveitamento do conhecimento de um framework em relação a outro, impossibilidade de migração entre os frameworks, quando um deles possuía mais funcionalidades que outros, entre outras questões.

Para resolver esse problema, a Sun lançou, em 2006, o primeiro release de sua nova especificação, a **JPA**, ou

## Fique por dentro

Este artigo visa a comparação entre duas excelentes opções de frameworks para o mapeamento entre objetos Java e dados relacionais de banco de dados, o Spring Data JPA e a própria especificação JPA. Ao longo do artigo serão apresentados diversos exemplos que mostrarão como o Spring Data JPA simplifica ainda mais o desenvolvimento de projetos que envolvam operações entre objetos e dados de uma tabela. Para todos os exemplos serão feitas comparações com a especificação JPA e serão mostrados os benefícios de unir os dois frameworks no mesmo projeto.

Java Persistence API, para a padronização do mapeamento entre o mundo orientado a objetos e o mundo relacional. A partir desse momento, os frameworks mais utilizados, como o Hibernate, o EclipseLink e o TopLink, passaram a seguir as regras dessa especificação, o que possibilitou a migração de uma solução para a outra de forma fácil, além de uma API simples e padronizada, permitindo o reaproveitamento do conhecimento de um framework em relação a outro.

Apesar das incríveis funcionalidades existentes na especificação JPA, muitas delas são repetitivas, fazendo com que o mesmo código seja reescrito diversas vezes, ou exigem muito código para operações simples. Tendo em vista esse cenário, o Spring criou o framework Spring Data JPA, cujo principal objetivo é permitir que o desenvolvimento com JPA se torne mais fácil, prático e menos repetitivo.

### Nota

É importante saber que o Spring Data JPA utiliza as próprias funcionalidades da especificação JPA, mas encapsula os seus recursos.

A seguir listamos algumas das principais funcionalidades do Spring Data JPA, que serão analisadas ao longo do artigo, comparando também com o que há de equivalente na especificação JPA:

- Operações de criação, remoção, modificação e seleção, conhecidas como CRUD;
- Query Methods;
- Method Names;
- Query Annotation;
- Named Queries;
- Ordenação.

## Configurando o Spring Data JPA

O Spring Data JPA pode ser configurado de diversas formas, por exemplo: fazendo o download manual do seu JAR; usando um gerenciador de dependências, como o Gradle ou o Maven; utilizando o projeto Spring Boot; e até mesmo o projeto JBoss Forge. Para este artigo optamos pelo gerenciador de dependências Maven. Para que o Spring Data JPA seja configurado no Maven, basta adicionar no arquivo pom.xml as linhas indicadas na **Listagem 1**.

**Listagem 1.** Declaração da dependência do Spring Data JPA.

```
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-jpa</artifactId>
<version>1.9.4.RELEASE</version>
</dependency>
```

## Implementando um CRUD com JPA

É comum que projetos em sua fase inicial comecem com operações básicas de criação, remoção, modificação e seleção de objetos. Para o nosso primeiro exemplo, criaremos um CRUD de funcionários e veremos a primeira comparação entre o framework Spring Data JPA e a especificação JPA.

Na **Listagem 2** apresentamos o código da classe **Funcionario**, já com as anotações da JPA para que seja possível o mapeamento entre essa classe e a tabela *funcionario* no banco de dados.

A seguir temos a descrição de cada anotação da JPA que foi utilizada:

- **@Table:** É utilizada para informar o nome da tabela relacionada a essa entidade no banco de dados;
- **@Entity:** Indica que a classe será mapeada e passará a ser gerenciada pela JPA;
- **@Column:** Nos possibilita fazer algumas configurações, como o nome da coluna no banco de dados, tamanho da coluna, obrigatoriedade de campos, etc.;
- **@Id:** Indica que o atributo da classe será gerenciado pela JPA e terá o seu valor automaticamente incrementado;
- **@GeneratedValue:** Permite que o banco utilize a estratégia de gerar automaticamente os valores da coluna **id**, que representa a chave primária da tabela.

Com a classe **Funcionario** mapeada, podemos criar uma outra classe, que será responsável por conter as ações necessárias para

um CRUD de funcionários. Nesses casos é comum a criação de uma classe DAO — *Data Access Object* — ou de uma classe Repository que contenha tais operações encapsuladas. Na **Listagem 3** apresentamos o código da classe **FuncionarioRepositoryJPA**.

**Listagem 2.** Classe Funcionario sendo mapeada pela JPA.

```
@Table(name = "funcionario")
@Entity
public class Funcionario {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

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

    @Column(name = "sobrenome")
    private String sobrenome;

    @Column(name = "cargo")
    private String cargo;

    @Column(name = "departamento")
    private String departamento;

    @Deprecated
    Funcionario() {
    }

    public Funcionario(String nome, String sobrenome, String cargo,
String departamento) {
        this.nome = nome;
        this.sobrenome = sobrenome;
        this.cargo = cargo;
        this.departamento = departamento;
    }

    // getters e setters omitidos
}
```

**Listagem 3.** Classe com as operações de CRUD usando JPA.

```
public class FuncionarioRepositoryJPA {

    private EntityManager manager;

    public FuncionarioRepositoryJPA(EntityManager manager) {
        this.manager = manager;
    }

    public void save(Funcionario funcionario) {
        manager.persist(funcionario);
    }

    public Funcionario findById(Long id) {
        return manager.find(Funcionario.class, id);
    }

    public void update(Funcionario funcionario) {
        manager.merge(funcionario);
    }

    public void delete(Funcionario funcionario) {
        manager.remove(funcionario);
    }
}
```

# Persistência de dados: JPA ou Spring Data JPA?

Note que cada método representa uma operação no CRUD. Para que as modificações feitas no objeto **funcionario** sejam refletidas no banco de dados, é necessária a utilização da interface **Entity-Manager**, da JPA, cujo objetivo principal é gerenciar os objetos que serão mapeados para o banco.

Com a classe **FuncionarioRepositoryJPA** implementada, chegou a hora de utilizá-la. Para isso, criaremos uma nova classe, **FuncionarioController**, para que possamos fazer uso dos métodos adicionados no repositório de funcionários. A **Listagem 4** apresenta esse código, onde empregamos as seguintes anotações:

- **@Controller**: Indica que a classe será um controller gerenciado pelo Spring e poderá receber chamadas web;
- **@RequestMapping**: Indica que o método poderá ser invocado a partir de uma URL no browser, por exemplo, por um formulário de cadastro de funcionários;
- **@Autowired**: Uma das anotações mais conhecidas do Spring, é usada para injetar uma dependência, comumente chamada de bean.

**Listagem 4.** Classe controller — utiliza a classe FuncionarioRepositoryJPA e seus métodos de CRUD.

```
@Controller  
public class FuncionarioController {  
  
    @Autowired  
    private FuncionarioRepositoryJPA funcionarioRepository;  
  
    @RequestMapping("/salvar")  
    public void salvar(Funcionario funcionario) {  
        funcionarioRepository.save(funcionario);  
    }  
  
    @RequestMapping("/remover")  
    public void remover(Funcionario funcionario) {  
        funcionarioRepository.remove(funcionario);  
    }  
  
    @RequestMapping("/atualizar")  
    public void atualizar(Funcionario funcionario) {  
        funcionarioRepository.update(funcionario);  
    }  
  
    @RequestMapping("/buscar")  
    public Funcionario buscaPorId(Long id) {  
        return funcionarioRepository.findById(id);  
    }  
}
```

Vale destacar que **FuncionarioController** é somente uma classe de teste para que possamos invocar os métodos do repositório de funcionários. Ao longo do artigo não faremos mais uso dela, pois vamos focar somente nas funcionalidades da JPA e do Spring Data JPA.

Por fim, apesar de **FuncionarioRepositoryJPA** ser bem simples e ter pouco código, note que mesmo para operações básicas, como as relacionadas ao CRUD, foi escrito muito código para que a JPA faça as operações de persistência de um funcionário no banco de dados.

## Implementando um CRUD com Spring Data JPA

O Spring Data JPA utiliza diversos padrões de projeto internamente e um deles é o próprio Repository, que permite fazer a persistência de dados de forma mais simples e intuitiva do que fizemos no exemplo anterior. Para que seja possível fazer tal operação de persistência de dados, o Spring Data JPA contém diversas interfaces, como:

- **Repository**: Interface de marcação para indicar que uma classe é um repositório gerenciado pelo Spring. Ela é usada internamente por outras interfaces do próprio Spring e dificilmente será necessário implementá-la diretamente;
- **CrudRepository**: Como o próprio nome indica, essa interface possibilita a persistência e gerenciamento de objetos em operações de CRUD. É uma das interfaces mais utilizadas em projetos com Spring Data JPA;
- **PagingAndSortingRepository**: Interface que estende **CrudRepository** e possibilita operações de CRUD com paginação e ordenação de resultados;
- **JpaRepository**: Interface que herda os métodos da interface **PagingAndSortingRepository** e é uma extensão específica para a especificação JPA.

Para o nosso primeiro exemplo com Spring Data JPA, vamos usar a interface **CrudRepository** com o objetivo de criar as mesmas operações de CRUD vistas na **Listagem 3**. A nova interface foi criada na **Listagem 5**, com o nome de **FuncionarioRepositorySpringDataJPA**, e contém as mesmas operações de **FuncionarioRepositoryJPA**.

**Listagem 5.** FuncionarioRepositorySpringDataJPA estendendo a interface CrudRepository para operações de CRUD.

```
public interface FuncionarioRepositorySpringDataJPA extends  
    CrudRepository<Funcionario, Long> {  
}
```

Pronto! Essa interface executa exatamente as mesmas operações vistas na classe **FuncionarioRepositoryJPA**, mas como você pode notar, sem a necessidade de escrever um método. Neste momento você pode estar pensando: Como isso é possível, se não implementamos um método?

Essa é a primeira facilidade que ganhamos quando optamos pelo Spring Data JPA. A interface **CrudRepository** já oferece 11 métodos diferentes para que seja possível executarmos diversas combinações de operações de CRUD, enquanto a classe **FuncionarioRepositoryJPA** possui apenas quatro, escritos por nós mesmos.

Para analisar como funciona internamente a interface de CRUD do Spring Data JPA, na **Listagem 6** apresentamos os três métodos da interface **CrudRepository**, que correspondem às quatro operações de CRUD que implementamos na classe **FuncionarioRepositoryJPA**. Existem somente três métodos porque o método **save()** possui dois objetivos, o de salvar e o de atualizar um objeto.

**Listagem 6.** Métodos para operações de CRUD da interface CrudRepository – Spring Data JPA.

```
<S extends T> S save(S entity);  
T findOne(ID id);  
void delete(T entity);
```

Para que façamos uso do exemplo da **Listagem 5**, na **Listagem 7** trocamos a classe **FuncionarioRepositoryJPA** pela interface **FuncionarioRepositorySpringDataJpa** na nossa classe **FuncionarioController**.

**Listagem 7.** FuncionarioController com Spring Data JPA para o mapeamento do repositório de funcionários.

```
@Controller  
public class FuncionarioController {  
  
    @Autowired  
    private FuncionarioRepositorySpringDataJPA funcionarioRepository;  
  
    @RequestMapping("/salvar")  
    public void salvar(Funcionario funcionario) {  
        funcionarioRepository.save(funcionario);  
    }  
  
    @RequestMapping("/remover")  
    public void remover(Funcionario funcionario) {  
        funcionarioRepository.delete(funcionario);  
    }  
  
    @RequestMapping("/atualizar")  
    public void atualizar(Funcionario funcionario) {  
        funcionarioRepository.save(funcionario);  
    }  
  
    @RequestMapping("/buscar")  
    public Funcionario buscaPorId(Long id) {  
        return funcionarioRepository.findOne(id);  
    }  
}
```

Neste momento os métodos da classe **FuncionarioController** foram modificados para que as invocações sejam feitas na interface do Spring Data JPA, de forma muito semelhante aos métodos das operações de CRUD que criamos na classe **FuncionarioRepositoryJPA**, mas com a vantagem de não ter sido necessário criar um método na interface **FuncionarioRepositorySpringDataJpa**.

## Spring Data JPA e Query Methods

O Spring Data JPA chama de Query Methods os métodos que são adicionados na classe que implementa a interface **CrudRepository** e que executam uma determinada query. Esses métodos devem seguir uma convenção de nomes do próprio Spring Data JPA para que as queries sejam criadas internamente pelo Spring e executadas.

Para que vejamos as diferenças entre a JPA e Spring, faremos comparações entre a forma que a JPA cria queries e a forma que o Spring Data cria as mesmas queries, mas usando Query Methods.

Deste modo, os próximos tópicos cobrirão detalhadamente o uso de query methods para a criação de queries comuns, facilmente encontradas em um projeto real.

### Query Methods simples

Para ilustrar o nosso primeiro exemplo com Query Methods, vamos pensar na seguinte consulta: retornar todos os funcionários cujo nome seja “Alexandre”. Essa é uma query bastante simples, pois envolve somente um comando de **select** na query. Na **Listagem 8** apresentamos sua implementação usando JPA.

**Listagem 8.** Consulta via JPA que retorna todos os funcionários filtrados por um nome.

```
public List<Funcionario> buscaPorNome(String nome) {  
    String jpql = "select f from Funcionario f where f.nome = :nome";  
    TypedQuery<Funcionario> query = manager.createQuery(jpql,  
    Funcionario.class);  
    query.setParameter("nome", nome);  
  
    return query.getResultList();  
}
```

Nesse código foi criado o método **buscaPorNome()** contendo uma variável do tipo **String** com o nome **jpql**, e como já mencionado, a consulta tem o objetivo de recuperar todos os funcionários de determinado nome. Para a variável foi dado o nome **jpql** somente por convenção, para indicar que estamos utilizando a linguagem de persistência da JPA, a *Java Persistence Query Language*, que é uma forma própria do JPA para criar queries através de strings. Note que a string da variável **jpql** é muito semelhante às queries nativas que executamos em bancos de dados.

Em seguida, criamos o objeto **TypedQuery** através da invocação do método **createQuery()** da variável **manager**. Esse objeto, representado pela variável **query**, permitirá que passemos o nome “Alexandre” como parâmetro para a query de busca de funcionários, através do método **setParameter()**. Por último, invocaremos o método **getResultSet()**, cujo objetivo é executar a consulta e retornar a lista de funcionários encontrados com o nome especificado. Apesar de simples, observe que escrevemos bastante código para implementar essa consulta.

Na **Listagem 9**, criamos a mesma operação de busca de funcionários pelo seu nome, porém, usando o Spring Data JPA, para compararmos as duas opções.

**Listagem 9.** Consulta via Spring Data JPA que retorna todos os funcionários filtrados por um nome.

```
public interface FuncionarioRepositorySpringDataJpa extends  
    CrudRepository<Funcionario, Long> {  
  
    List<Funcionario> findByNome(String nome);  
}
```

# Persistência de dados: JPA ou Spring Data JPA?

Nessa última listagem, apenas declaramos o método **findByNome()**. Note que não foi necessária nenhuma implementação para o método. Mas, se não existe nenhuma implementação, como o Spring Data JPA executará uma query para retornar os mesmos resultados do código da **Listagem 8**?

Assim como muitos dos projetos Spring, o Spring Data JPA utiliza convenções para montar as consultas ao banco de dados. Nesse método fizemos uso de uma das convenções de nome. Repare que em sua assinatura existe o nome **findBy**. Esse pequeno trecho é uma convenção do Spring que indica ao Spring Data que gostaríamos de buscar um registro baseado em algum atributo mapeado na classe **Funcionario**.

Em seguida, finalizamos o nome do método com a palavra-chave **nome**. Isso significa que vamos utilizar o atributo **nome** da classe **Funcionario**, ou seja, o método **findBy** precisa ter, também, o nome do atributo que se deseja pesquisar. Com essa convenção, o Spring Data monta uma consulta JPQL que executa a mesma operação da **Listagem 8**, mas sem a necessidade de escrita da query, ou melhor, sem a necessidade de implementar o corpo no método.

Na **Listagem 10** apresentamos mais alguns métodos que retornam funcionários a partir de outros atributos da classe **Funcionario**.

**Listagem 10.** Métodos que buscam funcionários a partir dos atributos nome, sobrenome e cargo.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    List<Funcionario> findByNome(String nome);  
  
    List<Funcionario> findBySobrenome(String sobrenome);  
  
    List<Funcionario> findByCargo(String cargo);  
  
}
```

Note como foi simples implementar filtros de funcionários por nome, sobrenome e cargo. Para tanto, basta adicionar o atributo desejado para que o Spring Data crie internamente a query com o filtro.

## Query Methods com consultas JPQL

Outra opção interessante do Spring Data JPA é que podemos escrever uma consulta personalizada em um método qualquer de um Repository. Como exemplo, criaremos o método **findNomeByCargo()**, que conterá a consulta SQL que gostaríamos de executar quando o método for invocado. Para isso, imagine que precisamos buscar um funcionário pelo seu cargo e retornar somente o seu nome, ao invés do objeto **Funcionario** inteiro.

Na **Listagem 11** temos o código para isso utilizando JPA e JPQL.

Observe que adotamos a opção padrão do JPA para retornar um registro. Observe também que fizemos uso da cláusula

**where**, responsável, neste caso, pelo filtro dos funcionários através do **cargo**. Já na **Listagem 12**, temos essa mesma operação, mas explorando os recursos do Spring Data JPA.

**Listagem 11.** Busca por funcionário pelo seu cargo e retorna o seu nome usando JPA e JPQL.

```
public String buscaNomePeloCargo(String cargo) {  
    String jpql = "select f.nome from Funcionario f where f.cargo = :cargo";  
    TypedQuery<String> query = manager.createQuery(jpql, String.class);  
    query.setParameter("cargo", cargo);  
  
    return query.getSingleResult();  
}
```

**Listagem 12.** Busca por funcionário pelo seu cargo e retorna o seu nome usando Spring Data JPA.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    @Query(value = "select f.nome from Funcionario f where f.cargo = :cargo")  
    String findNomeByCargo(@Param("cargo") String cargo);  
  
}
```

Nesse código utilizamos a anotação **@Query** e passamos como valor do atributo **value** a **String** que representa a consulta que deverá ser executada quando o método **findNomeByCargo()** for invocado. Repare que é bem parecido com o que é feito no JPA, porém, de forma mais simples, pois não é necessário lidar com o parâmetro da cláusula **where**.

Saiba, no entanto, que essa operação só é possível graças à outra anotação que usamos, a **@Param**, que é responsável por fazer a ligação entre a variável **cargo** e o parâmetro **cargo** da query, indicado por **:cargo** na string JPQL. Quando esse método for executado, retornará o mesmo resultado da **Listagem 11**.

## Tipos possíveis de objetos retornados a partir de um Query Method

O leitor mais atento deve ter notado, nos exemplos anteriores, que sempre retornamos um objeto do tipo **List**, no entanto, é importante ressaltar que o Spring Data disponibiliza três tipos de objetos para isso:

- **Tipo básico:** O objeto retornado poderá ser um tipo básico do Java, como **String**, **Integer**, **Float**, etc. ou **null**. Esses objetos também podem ser retornados em um objeto que implementa ou estende a interface **Collection**, como **List**, **LinkedList**, **Set**, etc.;
- **Entity:** O objeto retornado será do tipo usado no Repository. No nosso caso, a classe **Funcionario**;
- **Optional:** O objeto retornado poderá ser do tipo **Optional**, tanto do Java 8 quanto da biblioteca do Google Guava.

Para demonstrar os possíveis tipos retornados a partir de um query method, na **Listagem 13** foram criados três métodos na interface **FuncionarioRepositorySpringDataJPA**, cada um com um tipo diferente de retorno.

**Listagem 13.** Interface FuncionarioRepositorySpringDataJPA, com métodos usando os três tipos de retorno disponibilizados pelo Spring Data JPA.

```
public interface FuncionarioRepositorySpringDataJPA extends  
CrudRepository<Funcionario, Long> {  
  
    @Query(value = "select f.nome from Funcionario f where f.cargo = :cargo")  
    String findNomeByCargo(@Param("cargo") String cargo);  
  
    @Query(value = "select f.nome from Funcionario f where f.codigo = :codigo")  
    Optional<String> findNomeByCodigo(@Param("codigo") String codigo);  
  
    Funcionario findByld(Long id);  
  
    Optional<Funcionario> findByDepartamento(String departamento);  
}
```

Note que basta indicar qual é o tipo de retorno desejado na assinatura do método e o Spring Data JPA fará a conversão automaticamente. O mesmo não pode ser feito com JPA de forma tão fácil, pois, como vimos na **Listagem 11**, é necessário indicar para a JPA qual o tipo de retorno desejado através da interface **TypedQuery**. Ademais, se for necessário ter um **Optional** como retorno, a JPA não tem suporte e esse processo deve ser feito de forma manual.

#### *Tipos possíveis de retorno de um Query Method para uma lista*

Quando é necessário ter uma lista de objetos como retorno, o Spring Data disponibiliza dois tipos diferentes de objetos:

- **List**: O resultado é colocado em um objeto do tipo **List** ou é retornado um objeto do tipo **List** vazio;
- **Stream**: O resultado é uma **stream**, um tipo de objeto adicionado ao Java 8 e que permite operações de filtro, redução, conversão em uma lista, entre outras.

O código da **Listagem 14** contém dois métodos adicionados ao repositório de funcionários para mostrar os dois exemplos anteriores: o primeiro, **findByCargo()**, exemplificando o resultado em forma de uma lista, usando o objeto **List**; e o segundo, **findByCidade()**, retornando uma stream de objetos do tipo **Funcionario**.

A vantagem de o retorno ser uma **stream** é que podemos fazer operações de filtro, redução e conversão diretamente no objeto e em seguida transformá-lo em um **List**. A JPA só nos possibilita usar o tipo **List**, fazendo com que os filtros, redução e conversão tenham que ser feitos diretamente na query, tornando assim a nossa query menos genérica e aproveitável.

#### *Query Methods com parâmetros nos métodos do repositório*

Até agora os nossos exemplos foram simples, criando queries que só recebiam um parâmetro em seus métodos. Nos próximos casos serão criadas queries contendo duas ou mais colunas, utilizando parâmetros nas consultas.

**Listagem 14.** Métodos que retornam List ou Stream.

```
public interface FuncionarioRepositorySpringDataJPA extends  
CrudRepository<Funcionario, Long> {  
  
    List<Funcionario> findByCargo(String cargo);  
  
    Stream<Funcionario> findByCidade(String cidade);  
}
```

Para o nosso primeiro exemplo de query method com parâmetros, vamos buscar todos os funcionários de determinado **cargo** e **departamento**. A **Listagem 15** mostra a implementação do código JPA.

**Listagem 15.** Busca por funcionários que são de um determinado cargo e departamento usando JPA.

```
public class FuncionarioRepositoryJPA {  
  
    public List<Funcionario> buscaPeloCargoEDepartamento  
(String cargo, String departamento) {  
        String jpql = "select f from Funcionario f  
        where f.cargo = :cargo and f.departamento = :departamento";  
        TypedQuery<Funcionario> query = manager.createQuery  
(jpql, Funcionario.class);  
        query.setParameter("cargo", cargo);  
        query.setParameter("departamento", departamento);  
  
        return query.getResultList();  
    }  
}
```

Esse código continua simples e apenas passa as variáveis **cargo** e **departamento**, do tipo **String**, como parâmetros. Em seguida, usamos esses parâmetros na query JPQL de consulta de funcionários, através do método **setParameter()** da variável **query**. Dessa forma, a JPA criará internamente uma consulta com o **cargo** e **departamento** na cláusula **where**, que indica que somente os funcionários do cargo e departamento desejados sejam buscados no banco de dados. Já na **Listagem 16** temos a mesma operação, mas utilizando Spring Data JPA.

**Listagem 16.** Busca por funcionários que são de um determinado cargo e departamento usando Spring Data JPA.

```
public class FuncionarioRepositorySpringDataJpaTest {  
    List<Funcionario> findByCargoAndDepartamento(String cargo,  
    String departamento);  
}
```

Conforme visto nos exemplos, o Spring faz uso intensivo de convenções de nomes para os métodos e parâmetros. Na operação de busca de funcionários mostrada na **Listagem 16**, declaramos também a palavra-chave **And**, entre **Cargo** e **Departamento**. O Spring fará a leitura dessa palavra e, a partir disso, entenderá que deve criar uma query com a cláusula **where** usando o **Cargo** e **Departamento** para realizar o filtro de funcionários.

# Persistência de dados: JPA ou Spring Data JPA?

A query gerada pelo Spring é a mesma query criada na [Listagem 15](#), mas usando somente convenção de nomes.

## Parâmetros de métodos para queries nativas

Outra funcionalidade interessante do Spring Data é a possibilidade de executar queries nativas do banco de dados. Para que isso seja possível, devemos utilizar novamente a anotação `@Query` e escolher um dos dois tipos de passagem de parâmetro para o método: o tipo por posição ou o tipo por nome.

Na [Listagem 17](#) temos um exemplo de como usar parâmetros posicionais.

**Listagem 17.** Busca de funcionários que são de um determinado cargo e cidade usando parâmetros posicionais.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    @Query(value = "select * from Funcionario f where f.cargo = ?1  
    and f.cidade = ?2", nativeQuery = true)  
    List<Funcionario> findByCargoAndCidade(String cargo, String cidade);  
}
```

Nesse código, o parâmetro `nativeQuery` foi adicionado à anotação `@Query` para indicar que o Spring deverá executar a query como nativa e não como JPQL. Note que o atributo `value` contém uma string, representando a nossa query desejada, e utiliza o símbolo de interrogação para sinalizar a posição na qual os parâmetros `cargo` e `cidade` devem ser inseridos, respectivamente. Assim, a posição 1, representada pelo símbolo `?1`, informa que será utilizado o parâmetro `cargo`, e a posição 2, representada pelo símbolo `?2`, informa que será utilizado o parâmetro `cidade`.

Apesar de possível, a prática de escrever queries nativas não é recomendada, pois a consulta pode perder compatibilidade com outros bancos, caso o desenvolvedor dependa, por exemplo, de funções que são proprietárias de um banco de dados específico. O mesmo exemplo pode ser feito sem query nativa, usando somente JPQL, e com parâmetros posicionais, como mostra a [Listagem 18](#).

**Listagem 18.** Busca de funcionários que são de um determinado cargo e cidade usando Spring Data JPA e parâmetros posicionais.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    @Query(value = "select f from Funcionario f where f.cargo = ?1  
    and f.cidade = ?2")  
    List<Funcionario> findByCargoAndCidade(String cargo, String cidade);  
}
```

É válido ressaltar que os parâmetros posicionais são fáceis de utilizar e igualmente fáceis de proporcionarem erros. Imagine que em uma mudança de código na [Listagem 18](#) você decida trocar a ordem dos parâmetros `cargo` e `cidade` no método.

Como os dois são do tipo `String`, o código continuará compilando, mas sabemos que a funcionalidade estará errada, dado que o cargo será pesquisado como cidade e vice-versa. Por esse motivo o seu uso é desencorajado, sendo preferível trabalhar com parâmetros nomeados.

Os parâmetros nomeados são mais intuitivos e com menor possibilidade de erros, já que normalmente possuem o mesmo nome que os parâmetros do método. Na [Listagem 19](#) você pode acompanhar a mesma funcionalidade do código da [Listagem 18](#), mas usando parâmetros nomeados.

**Listagem 19.** Busca de funcionários que são de um determinado cargo e cidade usando Spring Data JPA, parâmetros nomeados e query nativa.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    @Query(value = "select * from Funcionario f where f.cargo = :cargo  
    and f.cidade = :cidade", nativeQuery = true)  
    List<Funcionario> findByCargoAndCidade(@Param("cargo") String cargo,  
                                            @Param("cidade") String cidade);  
}
```

Nesse exemplo usamos dois parâmetros com nomes na string de consulta, `:cargo` e `:cidade`, que fazem referência, respectivamente, aos parâmetros `cargo` e `cidade`. Note que se trocarmos a ordem dos parâmetros no método, a query continuará funcionando. Quem assegura esse funcionamento é a anotação `@Param`, que é usada justamente para fazer a ligação entre o nome do parâmetro no método e na query.

## Method names em Query Methods

Até o momento apresentamos diversas queries, mas em todas elas passamos no máximo dois parâmetros para os métodos. Note, ainda, que na maioria das queries o operador `And` do Spring Data esteve presente. Contudo, existem muitos outros operadores a serem utilizados em queries, como o `Or`, `Group By`, `Order By` e `Like`.

Dito isso, nossos próximos exemplos estarão relacionados à criação de métodos que executam queries com as seguintes características:

- Busca por funcionário que esteja na cidade de São Paulo **ou** que trabalhe no departamento de Engenharia;
- Busca por funcionário que trabalhe em algum departamento que **contenha** a palavra “Engenharia”;
- Busca pelos **três primeiros** funcionários que trabalhem em São Paulo.

Na [Listagem 20](#) temos a classe `FuncionarioRepositoryJPA` com a resolução desses três problemas usando somente a especificação JPA.

Foram pelo menos 24 linhas de código para implementar as três consultas com JPA. Na [Listagem 21](#), o mesmo resultado é alcançado, mas utilizando Spring Data JPA e muito menos linhas.

**Listagem 20.** Classe com os métodos que mostram a resolução das três situações propostas.

```
@Repository  
public class FuncionarioRepositoryJPA {  
  
    public List<Funcionario> buscaPorDepartamentoOuCidade  
(String departamento, String cidade) {  
        String jpql = "select f from Funcionario f where  
f.departamento = :departamento or f.cidade = :cidade";  
        TypedQuery<Funcionario> query = manager.createQuery  
(jpql, Funcionario.class);  
        query.setParameter("departamento", departamento);  
        query.setParameter("cidade", cidade);  
  
        return query.getResultList();  
    }  
  
    public List<Funcionario> buscaPorDepartamentoContendoPalavra  
Caselnsensitive(String departamento) {  
        String jpql = "select f from Funcionario f where f.departamento like :departamento";  
        TypedQuery<Funcionario> query = manager.createQuery(jpql,  
Funcionario.class);  
        query.setParameter("departamento", "%" + departamento + "%");  
  
        return query.getResultList();  
    }  
  
    public List<Funcionario> buscaOsTresPrimeirosPelaCidade(String cidade) {  
        String jpql = "select f from Funcionario f where f.cidade = :cidade order by asc";  
        TypedQuery<Funcionario> query = manager.createQuery  
(jpql, Funcionario.class);  
        query.setParameter("cidade", cidade);  
        query.setMaxResults(3);  
  
        return query.getResultList();  
    }  
}
```

**Listagem 21.** Interface FuncionarioRepositorySpringDataJpa com a implementação dos três problemas propostos.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    List<Funcionario> findByCidadeOrDepartamento(String cidade,  
String departamento);  
  
    List<Funcionario> findByDepartamentoAllIgnoreCase(String departamento);  
  
    List<Funcionario> findFirst3ByCidade(String cidade);  
}
```

Observe que foi escrito um número muito menor de linhas com Spring Data JPA, e novamente o Spring utilizou a convenção de nomes para facilitar a criação das queries.

A novidade no primeiro método é o uso da palavra de convenção **Or**, que será usada para criar a query com a cláusula **or** no **where**. No segundo método foi usada a convenção **AllIgnoreCase**, que indica que a query será case insensitive, ou seja, não levará em consideração a distinção entre letras maiúsculas e minúsculas na cláusula **where**. E no terceiro método usamos a convenção **First3**, que limitará a quantidade de registros retornados pela query.

Essa palavra reservada é o equivalente ao método **setMaxResults()**, usado pelo terceiro método da **Listagem 20**. O interessante é que basta trocar o número 3 por outros números para que o limite seja modificado.

### Named Query com JPA e Spring Data JPA

Named Queries são queries criadas na própria entidade (no nosso caso, na entidade **Funcionario**) e que possuem um nome que será usado para construir a query pelo **EntityManager** da JPA. Como exemplo, na **Listagem 22** adicionamos o código contendo uma Named Query na classe **Funcionário** responsável por buscar funcionários a partir do nome de um departamento.

**Listagem 22.** Classe Funcionario com uma Named Query configurada.

```
@NamedQuery(name = "Funcionario.buscaPorDepartamento", query = "select f  
from Funcionario f where f.departamento = :departamento")  
public class Funcionario {  
}
```

Observe que esse código recebeu a anotação **@NamedQuery**, a qual possui dois atributos obrigatórios: **name**, que indica qual será o nome da query; e **query**, que contém a query JPQL propriamente dita.

O uso de Named Query é interessante por dois motivos:

- O Hibernate consegue fazer cache dos parâmetros antes da execução da query. Isso significa que quando a query for executada, o Hibernate a executará mais rápido, já que os seus parâmetros já são conhecidos pelo cache;
- Permite que o código na classe DAO fique mais limpo e claro.

Para que seja visto o Named Query em ação com JPA, na **Listagem 23** foi criado um método na classe **FuncionarioRepositoryJPA** que executa o código da Named Query contida na classe **Funcionario**.

Observe que invocamos o método **createNamedQuery()** do **EntityManager**, indicando que será criada uma nova Named Query. Como parâmetros, **createNamedQuery()** recebe o nome que desejamos para a Named Query e o nome da classe.

**Listagem 23.** Execução da Named Query de Funcionario usando somente JPA.

```
@Repository  
public class FuncionarioRepositoryJPA {  
  
    public List<Funcionario> buscaFuncionariosPorDepartamento  
(String departamento) {  
        TypedQuery<Funcionario> query = manager.  
createNamedQuery("Funcionario.buscaPorDepartamento", Funcionario.class);  
        query.setParameter("departamento", departamento);  
  
        return query.getResultList();  
    }  
}
```

# Persistência de dados: JPA ou Spring Data JPA?

Visto que na **Listagem 22** o nome dado foi *Funcionario.buscaPorDepartamento*, esse deve ser o nome passado como parâmetro para **createNamedQuery()**. A partir desse momento chamamos, como de costume, o método **setParameter()**, para montar a query com o parâmetro especificado; no caso, o nome do departamento.

**Listagem 24.** Interface com o método *buscaPorDepartamento()*, que usará a Named Query criada na classe Funcionario.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    List<Funcionario> buscaPorDepartamento(@Param("departamento")  
    String departamento);  
  
}
```

No caso da **Listagem 24**, não foi necessário criar uma JPQL e nos beneficiamos de outra convenção de nomes do Spring. Note que o nome do método é o mesmo nome que foi dado para a Named Query da classe **Funcionario** na **Listagem 22**, removendo somente a palavra **Funcionario**. Dessa forma, o Spring Data consegue entender que existe uma Named Query com o nome **buscaPorDepartamento**. Para a passagem do parâmetro, foi usada a anotação **@Param**, que já vimos anteriormente, permitindo a ligação do parâmetro do método com o parâmetro da Named Query.

## Ordenação de dados com Spring Data JPA

Uma das operações mais comuns no dia a dia é a ordenação de uma determinada lista de objetos. Isso pode ser feito de duas maneiras: de forma ascendente e de forma descendente. Para exemplificar essas opções, vamos resolver os requisitos listados a seguir:

1. Buscar todos os Funcionários e ordená-los de forma ascendente pelo seu nome;
2. Buscar todos os Funcionários do departamento de Engenharia, ordenando-os de forma ascendente pelo nome e em seguida de forma descendente pelo cargo.

Para iniciar a comparação entre JPA e Spring Data, a **Listagem 25** mostra a implementação das duas situações utilizando somente JPA.

Para o primeiro caso, a query executada é bem simples: criamos a variável **jpql** e o seu valor contém a query de busca de todos os funcionários. Nessa consulta, informamos as palavras-chave **order by** e **asc** para indicar que a query deve ser executada ordenando de forma ascendente o resultado, usando o **nome** como objeto de ordenação.

No segundo exemplo, a query é bastante semelhante, mas foi adicionado mais um atributo na ordenação, o atributo **cargo**. Dessa forma, será feita a ordenação pelo **nome** e em seguida pelo **cargo**.

Para encerrar, na **Listagem 26** temos a implementação dos mesmos dois exemplos, usando Spring Data JPA.

Note como é simples o código com Spring Data JPA. Não precisamos nos preocupar com a query, pois novamente o Spring

Data utiliza a convenção de código. Nesse último exemplo, novas convenções de nome foram adotadas, como a **findAll**, para buscar todos os registros da tabela de funcionários, e a convenção **OrderByNome**, para ordenar a lista de funcionários pelo atributo **nome**.

**Listagem 25.** Métodos com as consultas para as duas situações utilizando somente JPA.

```
@Repository  
public class FuncionarioRepositoryJPA {  
  
    public List<Funcionario> buscaTodosOrdenadosPeloNome() {  
        String jpql = "select f from Funcionario f order by f.name asc";  
        TypedQuery<Funcionario> query = manager.createQuery(jpql, Funcionario.class);  
  
        return query.getResultList();  
    }  
  
    public List<Funcionario> buscaPeloDepartamentoOrdenadosPeloNome  
    AscendenteECargoDescendente(String departamento) {  
        String jpql = "select f from Funcionario f where f.departamento =  
        :departamento order by f.name asc, f.cargo desc";  
        TypedQuery<Funcionario> query = manager.createQuery(jpql, Funcionario.class);  
        query.setParameter("departamento", departamento);  
  
        return query.getResultList();  
    }  
}
```

**Listagem 26.** Métodos com as consultas para as duas situações utilizando Spring Data JPA.

```
public interface FuncionarioRepositorySpringDataJpa extends  
CrudRepository<Funcionario, Long> {  
  
    List<Funcionario> findAllByOrderByNameAsc();  
  
    List<Funcionario> findByDepartamentoOrderByNomeAscCargoDesc  
(String departamento);  
}
```

Observe que a string **nome** precisa ser igual ao nome do atributo **nome** da classe **Funcionário**. O método da **Listagem 26** contém, ainda, o termo **Asc**, para indicar que a ordenação será feita de forma ascendente. Se você não indicar que tipo de ordenação deseja para a sua lista, o Spring, por padrão, fará a ordenação de forma ascendente, não necessitando, portanto, que a palavra reservada **Asc** seja adicionada.

Para que a ordenação seja feita de forma descendente, basta adicionar o termo **Desc** no nome do método. Por último, utilizamos a convenção de nome **findByDepartamento** no segundo método, sinalizando que deve ser feita uma consulta com um filtro; no caso, o departamento.

A persistência de dados, como sabemos, é um dos assuntos mais importantes e discutidos em um projeto. O Spring Framework, além dos diversos módulos que facilitam o desenvolvimento de

software, conseguiu simplificar ainda mais a persistência de dados, usando diversas convenções de nomes que nos possibilitam criar somente a assinatura dos métodos, sem se preocupar com as queries, fazendo com que não precisemos lidar com a relativa complexidade da API da especificação JPA.

Além do que foi visto até aqui, o Spring Data contém muitos outros módulos, como o Spring Data MongoDB, excelente opção que facilita o desenvolvimento com o MongoDB, Spring Data Redis e Spring Data Solr, assim como módulos não oficiais, que foram criados pela comunidade, como o Spring Data Cassandra, Spring Data DynamoDB e Spring Data Couchbase.

## Autor



### Alexandre Gama

[alexandre.gama.lima@gmail.com](mailto:alexandre.gama.lima@gmail.com)

Trabalha como engenheiro de software no Elo7 e instrutor na Caelum. Além de cometer em alguns projetos open source, é palestrante em eventos nacionais como o TDC. É apaixonado por integração entre sistemas e está se aventurando no mundo DevOps.



## Links:

### Documentação do Spring Data JPA.

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

### Página principal do módulo Spring Data JPA.

<http://projects.spring.io/spring-data-jpa/>

### Guia inicial do Spring Data JPA.

<https://spring.io/guides/gs/accessing-data-jpa/>



# Introdução à recomendação de conteúdo com Apache Mahout e Hadoop

Como utilizar a API de machine learning da Apache e um sistema de processamento distribuído para criar um recomendador de conteúdo escalável

Com uma grande quantidade de informações e uma extensa variedade de produtos e serviços, cada vez mais nos deparamos com dificuldades para escolher entre as alternativas apresentadas. Frente a este cenário, geralmente confiamos nas recomendações que são passadas por outras pessoas e tomamos como base sua satisfação em relação àquilo que estão nos recomendando. Essas recomendações acontecem de muitas formas, normalmente através de jornais, revistas, revisores de filmes e livros, entre outros.

O objetivo de um sistema de recomendação é melhorar a capacidade do processo de indicação, muito comum na relação social entre seres humanos. Nesses sistemas, os usuários fornecem as recomendações como entrada e o sistema as direciona para os indivíduos potencialmente interessados; como acontece no Netflix, quando classificamos um filme indicando o número de estrelas. Um dos grandes desafios desse tipo de sistema é realizar o casamento correto entre as pessoas que estão recomendando e as pessoas que estão recebendo a recomendação. Esse relacionamento é conhecido como relacionamento de interesse. Entre as principais técnicas para recomen-

## Fique por dentro

Cada vez mais os cenários de recomendação de conteúdo são explorados por portais de conteúdo e e-commerce, possibilitando a identificação automática das preferências dos usuários com o objetivo de sugerir novos itens ou conteúdos de seu interesse. Neste artigo, apresentaremos uma introdução a esse assunto utilizando a API de machine learning da Apache (Apache Mahout) e um sistema de processamento distribuído (Hadoop) para construir um recomendador de vídeos inteligente e escalável.

dação de conteúdo, podemos destacar a filtragem demográfica e a filtragem colaborativa.

A filtragem demográfica utiliza a descrição de um indivíduo para aprender o relacionamento entre um item em particular e o tipo de indivíduo que poderia se interessar por ele. Nessa abordagem, os dados pessoais do usuário são requisitados através de formulários e combinados com o seu perfil de consumo, permitindo estabelecer um relacionamento de interesse para cada tipo de indivíduo. Já na filtragem colaborativa, esse relacionamento é determinado através do comportamento co-

num de diferentes usuários, ou seja, a filtragem colaborativa considera que existe um perfil de consumo comum entre as pessoas que gostam das mesmas coisas. Essa abordagem é vantajosa porque não precisa coletar mais informações sobre o usuário, além das informações sobre o seu comportamento de consumo no próprio portal.

É muito comum um usuário gostar de diversos itens do mesmo portal, e ao mapear todos esses interesses, geramos um grande volume de dados. Isso acontece porque os portais oferecem uma grande quantidade de produtos, e cada produto visitado precisa gerar um log de acesso. Lidar com esse volume de dados nos leva a um cenário de Big Data, quando muitas vezes temos a necessidade de realizar processamento paralelo e distribuído. Isso se torna ainda mais importante quando aplicamos algoritmos de machine learning, que normalmente são caros do ponto de vista computacional. Felizmente, para isso, podemos contar com a ajuda do Apache Mahout e do Hadoop.

O Apache Mahout é uma biblioteca de machine learning de código aberto cujos principais objetivos são: processar recomendações, classificações e agrupamentos. Mantido pela Apache Software Foundation, o Mahout nasceu em 2008 como um subprojeto do Apache Lucene, outra ferramenta de código aberto destinada a problemas de busca e recuperação de informações. Em 2010 o Apache Mahout se tornou um projeto de software independente, que visa escalabilidade e eficiência. Por isso compatibilizou seus algoritmos com o Hadoop.

O Hadoop, por sua vez, é uma ferramenta de código aberto que implementa o paradigma Map-Reduce, introduzido pelo Google e criado para realizar processamento paralelo e distribuído. Assim, o Hadoop é capaz de processar grandes conjuntos de dados dividindo uma tarefa em pequenas partes e processando essas partes em máquinas distintas.

Este artigo mostra como utilizar o Apache Mahout e o Hadoop para construir um recomendador de vídeos inteligente e escalável. Em nosso experimento utilizaremos a base de dados aberta MovieLens, que é provida por um grupo de estudo especialista em recomendação de conteúdo da Universidade de Minnesota.

## Apache Mahout

Como visto anteriormente, o Mahout é uma biblioteca de machine learning mantida pela Apache Software Foundation cujo objetivo é facilitar o uso de algoritmos de aprendizado de máquina quando utilizados em sistemas de processamento distribuído. Quando falamos em machine learning ou aprendizado de máquina, em português, estamos nos referindo a um conjunto de técnicas que permitem a uma máquina melhorar suas análises a partir de resultados obtidos anteriormente. Essas técnicas são muito exploradas pela mineração de dados e usam principalmente métodos estatísticos e probabilísticos, bem como reconhecimento de padrões e outras ferramentas matemáticas. Embora não seja uma área de estudo nova, está em pleno crescimento, tanto que grandes corporações, como Amazon, Facebook e Google, utilizam algoritmos desse tipo em muitas de suas aplicações.

Esses algoritmos são implementados em diversos tipos de aplicações, como: jogos, sistemas de detecção de fraudes, análise da bolsa de valores, forecast de preço, entre outros. Eles também são muito comuns em sistemas de recomendação, como os da Amazon e Netflix, que sugerem produtos aos usuários com base em compras/visualizações anteriores.

O aprendizado de máquina permite solucionar problemas a partir de duas abordagens: o aprendizado supervisionado e o aprendizado não supervisionado.

### Aprendizado supervisionado

O aprendizado supervisionado consiste em aprender uma função a partir de um conjunto de dados de treinamento, previamente rotulados. Após o treinamento, a máquina se torna apta a classificar um novo dado com base nas regras aprendidas durante o treino.

Muitos problemas podem ser solucionados a partir do aprendizado supervisionado, tais como: classificar mensagens de e-mail como spam, rotular páginas da web de acordo com o gênero, reconhecimento de imagens, etc. Para solucionar esses problemas podemos utilizar diversos algoritmos, por exemplo: Redes Neurais Artificiais, Máquinas de Vetor de Suporte (SVM) e classificadores Bayesianos.

### Aprendizado não supervisionado

No aprendizado não supervisionado não existe uma fase inicial de treinamento, pois seu propósito é indicar o significado dos dados. Essa técnica normalmente é utilizada para segmentar os dados e formar grupos lógicos, possibilitando identificar novas tendências e comportamentos.

Os algoritmos de aprendizado não supervisionado geralmente são utilizados para mapear padrões de consumo, bem como efetuar recomendações. Entre os principais algoritmos de aprendizado não supervisionado, podemos destacar: algoritmos de agrupamento e filtragem colaborativa.

### Recursos do Apache Mahout

Os algoritmos oferecidos pelo Mahout se dividem em quatro grupos: filtragem colaborativa, algoritmos de classificação, algoritmos de agrupamento e algoritmos de redução de dimensionalidade. Todos eles podem ser utilizados via API e a maior parte deles também pode ser utilizada via linha de comando, através de shell interativo.

## Hadoop

Como vimos anteriormente, o Hadoop é uma ferramenta de Map-Reduce que permite realizar processamento paralelo e distribuído. Ele é formado por dois componentes principais: o Hadoop Distributed File System (HDFS), que armazena e manipula os dados que serão processados pelas máquinas que compõem o cluster; e o Map-Reduce, que gerencia todo o processamento realizado pelo cluster de máquinas. A **Figura 1** ilustra esses dois componentes.

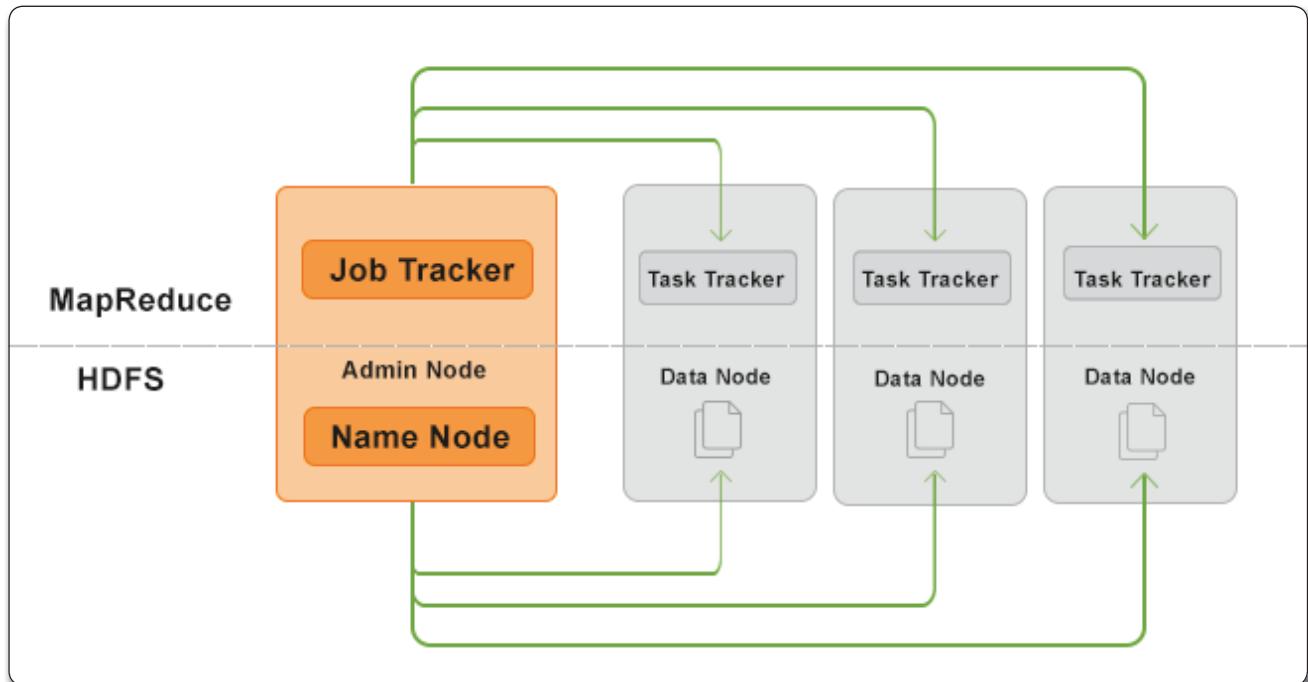


Figura 1. Componentes do Hadoop

### Hadoop Distributed File System (HDFS)

O HDFS é um sistema de arquivos distribuído baseado no Google File System (GFS). Esse tipo de sistema de arquivos é necessário quando o volume de dados a ser processado é grande demais para ficar em uma única máquina, fazendo com que toda a complexidade de um ambiente de rede entre em cena. Tal sistema armazena todos os arquivos em blocos de até 64MB e permite que os arquivos tenham réplicas que auxiliem no processamento paralelo e distribuído. Como um dos seus diferenciais, o HDFS possui dois tipos de nós: namenodes (master) e datanodes (slaves).

O namenode gerencia o namespace do sistema de arquivos, fazendo o mapeamento entre todos os blocos e onde estão armazenados. Já o datanode armazena os dados na forma de blocos e notifica o namenode sobre os arquivos que está armazenando. Esse primeiro nó (namenode) é um ponto crucial e qualquer falha nele pode impossibilitar o acesso aos arquivos. Para contornar esse problema, o Hadoop oferece um namenode secundário (*secondarynamenode*), que pode ser utilizado para reiniciar o sistema caso haja alguma falha no principal.

### Map-Reduce

O Map-Reduce permite especificar cada tarefa de processamento em duas funções: mapeamento e redução. Essas funções são executadas em paralelo no cluster e dependem de um sistema de arquivos distribuído, pois cada vez que uma máquina termina de processar uma janela de dados, ela deve pegar a próxima janela não processada e tratá-la.

A única forma de garantir que duas máquinas não realizem um trabalho redundante (processar a mesma janela), é se todos os

computadores do cluster notificarem o sistema de arquivos sobre as operações que estão realizando. Relacionado a isso, existem dois componentes responsáveis pela divisão e processamento dos dados: o job tracker e o task tracker.

O job tracker é um orquestrador de tarefas que notifica o namenode sobre o início ou término de um processamento. Isso permite a ele atribuir novos processamentos ao task tracker. O task tracker, por sua vez, processa os dados e, de tempos e tempos, notifica o job tracker, informando que o processamento ainda está sendo realizado. Caso a notificação não aconteça, o task tracker é considerado morto e suas tarefas de processamento são designadas a outro nó. Quando um task tracker recebe uma tarefa de processamento, ele cria um processo independente. Assim, consegue garantir que o nó não será comprometido caso algum processo falhe.

### Recursos em nuvem

Já há algum tempo o Hadoop tem sido oferecido como uma solução baseada em serviço na nuvem, cuja principal vantagem é o encapsulamento de seus componentes (namenodes, datanodes, job trackers e task trackers). Esse encapsulamento provê um gerenciamento automático dos recursos do Hadoop, permitindo um desenvolvimento mais ágil e seguro.

Uma das vantagens em utilizar a nuvem é que podemos fazer configurações personalizadas dos serviços, por exemplo: podemos escolher entre utilizar uma configuração tradicional (armazenamento e processamento) ou apenas o módulo de processamento, como é o caso do serviço AWS Elastic Map-Reduce. Fazer essa configuração fora da nuvem é mais difícil, pois teríamos que customizar o Hadoop.

### Filtragem colaborativa

A filtragem colaborativa é uma das técnicas mais indicadas para a construção de recomendadores de conteúdo, pois utiliza o histórico de interesses do usuário para sugerir novos itens. Ela foi popularizada pela Amazon e hoje está presente na maioria dos portais de e-commerce e conteúdo, permitindo indicar livros, músicas, filmes, notícias e páginas da web.

Esse tipo de algoritmo consiste em ter um conjunto de usuários e seu histórico de interesses, que pode ser composto de links visitados ou produtos comprados. O objetivo é encontrar outros usuários que manifestaram os mesmos interesses e estabelecer um perfil em comum. Quando isso acontece, costumamos dizer que encontramos um relacionamento de interesse. Esse relacionamento pode ser determinado de duas formas: recomendação baseada no usuário ou baseada no item.

Na recomendação baseada no usuário, os itens são sugeridos através da relação entre usuários e sua forma de consumo. Essa relação, no entanto, não é fácil de ser obtida, uma vez que os usuários possuem uma natureza dinâmica e podem mudar seus gostos e preferências ao longo do tempo. Um dos principais mecanismos para contornar a natureza dinâmica dos usuários é considerar um histórico recente de consumo. Já a recomendação baseada

em itens não sofre tantas mudanças e, em alguns casos, pode ser calculada com menor frequência. Isso passa a ser melhor quando a quantidade de usuários é maior que a quantidade de itens.

### Implementando um recomendador de conteúdo

Vejamos agora como criar um recomendador de conteúdo simples e de processamento não distribuído. Aqui, vamos apresentar e exemplificar alguns recursos e características da API Apache Mahout, bem como os parâmetros necessários para utilizar um algoritmo de filtragem colaborativa.

A partir da versão 0.7 da API, o Apache Mahout disponibiliza tanto a filtragem colaborativa para recomendações baseadas no usuário quanto para recomendações baseadas no item. Neste artigo optamos pela versão 0.10 da API e gerenciamos as dependências do projeto através do Apache Maven.

A **Listagem 1** apresenta o comando do Apache Maven a ser executado para criar o nosso projeto.

---

#### Listagem 1. Criação do projeto Java a partir do Apache Maven.

```
mvn archetype:generate  
-DgroupId=recomendador  
-DartifactId=recomendador  
-DarchetypeArtifactId=maven-archetype-quickstart  
-DinteractiveMode=false
```

Feito isso, note que será criado um diretório com o nome do projeto e, dentro dele, um arquivo chamado *pom.xml*. Esse arquivo representa uma especificação do projeto onde são configuradas todas as dependências, bem como informações sobre versão e empacotamento.

Para adicionar a dependência do Apache Mahout, basta editar esse arquivo e acrescentar o trecho de código apresentado na **Listagem 2**.

---

#### Listagem 2. Dependência do Apache Mahout.

```
<dependency>  
    <groupId>org.apache.mahout</groupId>  
    <artifactId>mahout-mr</artifactId>  
    <version>0.10.0</version>  
</dependency>
```

Observe que para configurar a dependência é necessário especificar três informações: **groupId**, **artifactId** e **version**, definições dessas que podem ser encontradas no repositório do Maven, onde devemos buscar pelo nome da API e pela versão que desejamos utilizar.

O trecho de código apresentado na **Listagem 3** ilustra como deve ficar o arquivo *pom.xml* após a inclusão da dependência do Mahout.

Encerrada essa etapa, execute o comando *mvn clean install* no diretório do projeto. Isso fará com que o Maven descarregue todas as dependências declaradas no *pom.xml* e as associe ao projeto, o que pode demorar alguns minutos.

# Introdução à recomendação de conteúdo com Apache Mahout e Hadoop

Listagem 3. Configuração do arquivo pom.xml.

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>recomendador</groupId>
  <artifactId>recomendador</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>recomendador</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>org.apache.mahout</groupId>
      <artifactId>mahout-mr</artifactId>
      <version>0.10.0</version>
    </dependency>
  </dependencies>
</project>
```

Para se certificar que as dependências foram configuradas corretamente, observe a mensagem de sucesso apresentada ao final do processo.

A partir deste momento podemos importar o projeto para uma IDE e iniciar a implementação do recomendador. Neste artigo, organizamos o código em duas classes: **FileManager** e **Recomendador**. A classe **FileManager** trata da importação dos dados para o Mahout, enquanto a classe **Recomendador** implementa dois exemplos de recomendação, um baseado no usuário e outro baseado no item.

Para facilitar a manipulação das informações, o Mahout disponibiliza um conjunto de classes voltadas à importação dos dados. Essas classes definem como os dados serão disponibilizados para os objetos que processam as recomendações.

O código da **Listagem 4** mostra como é simples importar os dados de um arquivo para o Mahout.

Listagem 4. Importação dos dados para o Mahout.

```
01 public class FileManager {
02   public static DataModel getFileDataModel() throws IOException {
03     File file = new File("caminho-do-arquivo");
04     return new FileDataModel(file);
05   }
06 }
```

Nesse código, criamos a classe **FileManager** para simplificar a importação dos dados. Ela possui um método estático que cria uma referência para o arquivo onde os dados estão armazenados e retorna um objeto do tipo **DataModel**, que traduz os dados para o Mahout.

Para que os dados sejam importados, eles precisam estar em um formato específico. Esse formato é necessário para que o algoritmo de filtragem colaborativa consiga estabelecer o relacionamento de interesse entre os usuários e os itens.

Esse arquivo foi extraído da base de dados aberta MovieLens (veja a seção **Links**), que, como dito no início do artigo, é provida por um grupo de estudos em recomendação de conteúdo da Universidade de Minnesota.

A **Tabela 1** apresenta o formato dos dados esperados pelo Mahout.

ID USUÁRIO	ID ITEM	CLASSIFICAÇÃO	TIMESTAMP
1	101	5	881250949
1	102	3	891717742
1	103	2.5	878887116
2	102	2	880606923
2	102	2.5	886397596
2	103	5	884182806
2	104	2	881171488
3	101	2.5	891628467
3	104	4	886324817
3	105	4.5	883603013
3	107	5	879372434
4	101	5	879781125
4	103	3	876042340
4	104	4.5	891035994
4	106	4	888104457
5	101	4	879485318
5	102	3	879270459
5	103	2	879539794
5	104	4	874834944
5	105	3.5	892079237
5	106	4	886176814

Tabela 1. Formato de entrada dos dados para o Mahout

Como podemos observar, os dados de entrada devem conter o identificador do usuário, o identificador do item, a classificação do conteúdo (entre 1 e 5) e o timestamp. Em geral, esses dados representam o acesso do usuário a um item de interesse, por exemplo: A primeira linha indica que o usuário de identificador 1 acessou o item de identificador 101 e deu a nota 5.

Normalmente, essas informações são coletadas durante a navegação do usuário no portal e armazenadas para processar as recomendações. O timestamp é um dado opcional, mas é importante colocar essa informação porque os algoritmos de filtragem colaborativa atribuem maior relevância para os registros mais novos. Isso permite acompanhar as mudanças de gosto e preferência dos usuários.

## Recomendação baseada no usuário

Agora que já sabemos como importar os dados a partir do Mahout, vejamos, na **Listagem 5**, como fazer recomendações baseadas no usuário.

#### Listagem 5. Recomendação baseada no usuário.

```
01 public void userBased() throws IOException, TasteException {  
02     long userId = 1;  
03     int vizinhos = 10;  
04     int totalRecomendacoes = 5;  
05  
06     // leitura dos dados.  
07     FileDataModel dataModel = FileManager.getFileDataModel();  
08  
09     // medida de similaridade e raio de vizinhança.  
10     UserSimilarity similarity = new PearsonCorrelationSimilarity(dataModel);  
11     UserNeighborhood neighborhood;  
12     neighborhood = new NearestNUserNeighborhood  
           (vizinhos, similarity, dataModel);  
13  
14     // processando as recomendações.  
15     GenericUserBasedRecommender recommender;  
16     recommender = new GenericUserBasedRecommender  
           (dataModel, neighborhood, similarity);  
17  
18     // obtendo lista de itens recomendados e imprimindo.  
19     List<RecommendedItem> recommendations = recommender.recommend  
           (userId, totalRecomendacoes);  
20     for (RecommendedItem item: recommendations) {  
21         System.out.println(item);  
22     }  
23 }
```

Para desenvolver uma recomendação baseada no usuário, o Mahout disponibiliza a interface **UserSimilarity**, que possui várias implementações. Para utilizar essa interface, precisamos calcular a similaridade de perfil entre os usuários e o raio de vizinhança, ou seja, o grau de distanciamento entre os usuários sobre os quais as buscas serão realizadas. O raio de vizinhança pode ser calculado através da interface **NearestNUserNeighborhood**, que nos retorna um **UserNeighborhood**. Ao instanciar **NearestNUserNeighborhood** indicamos, além dos dados de entrada, o número máximo de vizinhos (observe o atributo **vizinhos**) e o fator de similaridade. A especificação do número de vizinhos é importante, principalmente, para o controle de processamento. Quando maior o raio de vizinhança, maior será o consumo de CPU.

Uma vez definidos a similaridade entre os usuários e o raio de vizinhança, podemos gerar a lista de recomendação para um usuário, o que é feito através de **GenericUserBasedRecommender**, que recebe em seu construtor os dados, o raio de vizinhança e a similaridade. Em seguida, a partir do método **recommender.recommend()**, que recebe o identificador do usuário para o qual desejamos fazer uma recomendação e a quantidade máxima de itens que pretendemos recomendar, obtemos a lista de recomendação. No exemplo de código apresentado na **Listagem 5**, estamos fazendo um máximo de cinco recomendações.

Os resultados obtidos são exibidos no seguinte formato:

```
RecommendedItem[item:104, value:5.0]  
RecommendedItem[item:106, value:4.0]
```

Esses dados mostram os itens recomendados para o usuário de identificador 1, sendo possível notar que cada item possui um

valor de classificação, indicando o fator de aderência entre o item e o usuário em questão. Quanto maior for esse fator, mais aderente aos interesses do usuário esse item é.

Em casos onde a lista de recomendação é muito grande, podemos utilizar essa informação para ordenar os itens antes de apresentá-los ao usuário.

#### Recomendação baseada no item

A **Listagem 6** apresenta um exemplo de implementação para a recomendação de conteúdo baseada no item. Para isso, criamos o método **itemBased()**.

#### Listagem 6. Recomendação baseada no item.

```
01 public void itemBased() throws IOException, TasteException {  
02     long userId = 1;  
03     int totalRecomendacoes = 5;  
04  
05     // leitura dos dados.  
06     FileDataModel dataModel = FileManager.getFileDataModel();  
07  
08     // medida de similaridade.  
09     ItemSimilarity similarity = new TanimotoCoefficientSimilarity(dataModel);  
10  
11     // processando as recomendações.  
12     GenericBooleanPrefItemBasedRecommender itemRecommender;  
13     recommender = new GenericBooleanPrefItemBasedRecommender  
           (dataModel, similarity);  
14  
15     // obtendo lista de itens recomendados e imprimindo.  
16     List<RecommendedItem> recommendations = recommender.recommend  
           (userId, totalRecomendacoes);  
17     for (RecommendedItem item: recommendations) {  
18         System.out.println(item);  
19     }  
20 }
```

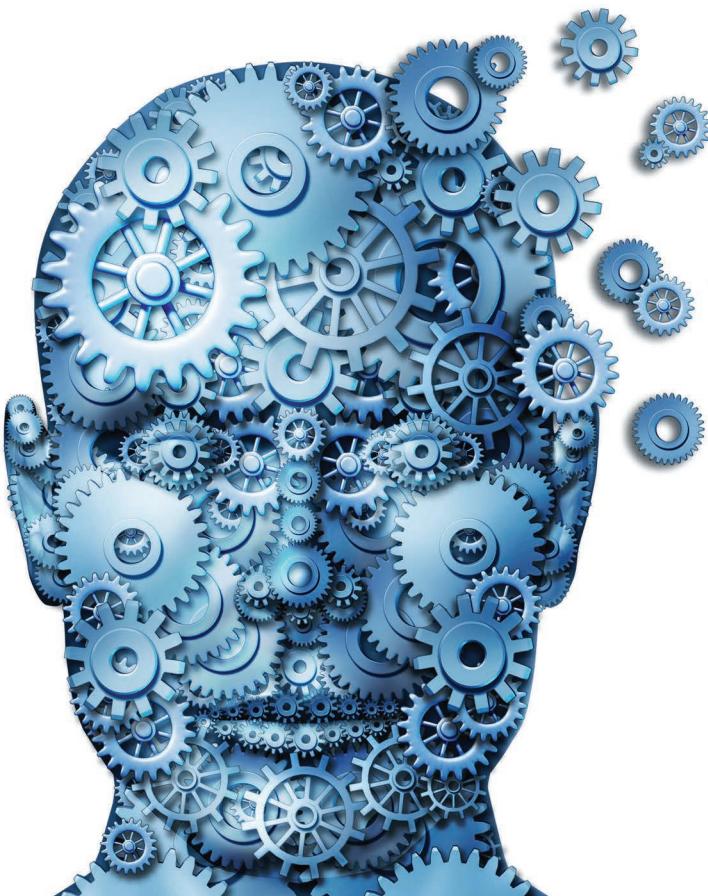


Para fazer as recomendações baseadas no item, o Mahout disponibiliza a interface **ItemSimilarity**, que assim como a interface **UserSimilarity**, possui várias implementações. Diferentemente da recomendação baseada no usuário, no entanto, a recomendação baseada no item não precisa calcular o raio de vizinhança, tornando a parametrização do algoritmo mais simples.

Quando se trata da recomendação por item, a classe **GenericBooleanPrefItemBasedRecommender** é responsável pelo cálculo de similaridade entre os dados. Depois de processar os dados, a lista de recomendação gerada segue o mesmo formato apresentado anteriormente, com o identificador dos itens e suas respectivas classificações, conforme o exemplo:

```
RecommendedItem[item:104, value:1.8]
RecommendedItem[item:106, value:1.15]
RecommendedItem[item:105, value:0.85]
RecommendedItem[item:107, value:0.2]
```

Note que tanto a recomendação baseada no usuário quanto a recomendação baseada no item produziram resultados muito semelhantes, divergindo apenas nas métricas e na forma com que foram calculadas. Assim como na recomendação baseada no usuário, os itens 104 e 106 foram os mais recomendados, muito embora os critérios para escolha tenham sido diferentes.



O tempo de processamento também pode variar de um método para o outro, dependendo da quantidade de registros. Como exemplo, vamos tomar uma situação onde a quantidade de itens seja muito maior do que a quantidade de usuários. Nesse caso, a recomendação baseada no usuário seria mais performática. Outro fator que também influencia no tempo de processamento é a quantidade de itens a serem recomendados. Quanto maior a quantidade de itens, maior será o custo computacional.

### Identificação do usuário

Uma característica muito importante para os sistemas de recomendação é conhecer o histórico de consumo dos usuários. Para isso, é preciso registrar todas as ações desses no portal, como: links e conteúdos acessados, produtos visualizados, etc. Essas informações devem ser armazenadas ao longo do tempo, e quanto mais informações coletarmos, melhor será a precisão do recomendador.

Para registrar todas essas informações é necessário identificar o usuário sempre que ele retornar ao site, mas isso pode ser um problema, principalmente quando ele não está autenticado. Uma alternativa para isso é adotar um mecanismo de Cookie Pool, que permite identificar cada usuário através de um número único. Assim, sempre que um usuário entrar no portal, basta verificar se ele é conhecido, buscando sua identificação no cookie. Um ponto sensível dessa solução é que o usuário pode fazer uma navegação anônima, ou até mesmo limpar os cookies do navegador, impossibilitando sua identificação.

Portanto, a identificação do usuário é um ponto crucial para um sistema de recomendação de conteúdo, de modo que o sistema passa a aprender sobre o seu perfil e mapear suas preferências. Esse mapeamento permitirá à solução de machine learning fazer recomendações mais efetivas e aderentes com aquilo que o usuário gosta.

### Efeito bolha

Para que os algoritmos de filtragem colaborativa consigam estabelecer um relacionamento de interesse entre os usuários e os itens, eles precisam que cada item receba uma classificação. Essa classificação pode ser feita de duas formas: classificação explícita ou classificação baseada no número de ocorrências. Na classificação explícita, os usuários classificam os itens de acordo com seus próprios interesses, como acontece no Netflix quando o usuário avalia um filme informando o número de estrelas.

A classificação também pode ser baseada no número de ocorrências, o que está relacionado à quantidade de acessos que um conteúdo recebe. Esse tipo de classificação é mais suscetível a falhas, uma vez que não expressa os interesses diretos de um ou outro usuário. Isso pode ser muito perigoso, uma vez que pode gerar um problema conhecido com efeito bolha.

Quando a classificação do conteúdo é baseada no número de acessos que ele tem, partimos do princípio que o conteúdo mais acessado é o mais bem recomendado. A questão é: É mais recomendado porque é o mais acessado, ou é mais acessado porque é mais recomendado?

Esse comportamento pode colocar o usuário em uma espécie de bolha, onde apenas as informações mais vistas sejam apresentadas a ele. Para contornar esse problema podemos fazer um balanceamento das classificações dos conteúdos.

Balancear essas classificações consiste em normalizar o número de acessos de cada conteúdo em um intervalo previamente estabelecido, normalmente entre 1 e 5. O problema dessa normalização é que ela não resolve as discrepâncias entre os valores, que em alguns casos podem ser muito grandes. Vejamos um exemplo na **Tabela 2**.

CONTEÚDO	NÚMERO DE ACESSOS	CLASSIFICAÇÃO
Link de conteúdo 1	600	5
Link de conteúdo 2	412	3.35
Link de conteúdo 3	201	1.67
Link de conteúdo 4	152	1.26
Link de conteúdo 5	137	1.14
Link de conteúdo 6	130	1.08

**Tabela 2.** Normalização do número de acessos para criar a classificação

Como podemos observar, a normalização consiste em atribuir a classificação 5 para o conteúdo mais acessado e calcular uma proporcionalidade para os demais conteúdos. Veja que essa classificação não é muito justa, pois existem dois links com um número de acessos muito maior. Quando optamos por essa classificação, os conteúdos menos acessados recebem uma avaliação de recomendação muito baixa, proporcionando novamente o efeito bolha, que leva o conteúdo mais votado a continuar sendo o recomendado.

Para tornar o cálculo da classificação mais justo, podemos utilizar o desvio padrão dos dados e criar faixas de valores normais, atribuindo a classificação máxima para os conteúdos mais acessados e normalizando os valores dos conteúdos menos acessados. A **Tabela 3** apresenta um exemplo.

CONTEÚDO	NÚMERO DE ACESSOS	CLASSIFICAÇÃO
Link de conteúdo 1	600	5
Link de conteúdo 2	412	5
Link de conteúdo 3	201	4
Link de conteúdo 4	152	3
Link de conteúdo 5	137	2.70
Link de conteúdo 6	130	2.56

**Tabela 3.** Classificação normalizada pelo desvio padrão

Note que atribuímos a classificação 5 para os dois primeiros itens, a classificação 4 para o terceiro item e consideramos uma proporcionalidade de acesso a partir do quarto item. Assim, a classificação geral dos itens foi melhorada, possibilitando que mais conteúdos sejam recomendados ao usuário.

### Experiência dos usuários (UX)

Para incorporar um mecanismo de recomendação de conteúdo, normalmente o portal precisa passar por ajustes na experiência

do usuário. O objetivo desses ajustes é separar o conteúdo recomendado dos demais conteúdos do portal, proporcionando mais facilidade para que o usuário encontre os assuntos de seu interesse.

Quando revisamos a experiência do usuário com o nosso portal, devemos considerar uma série de premissas: Um portal de receitas, por exemplo, poderia ser dividido em área de recomendação e área editorial. Os conteúdos exibidos na área de recomendação seriam sugeridos pelos algoritmos de machine learning. Já os conteúdos apresentados na área editorial poderiam ser relacionados a novidades sobre o portal, entrevistas recentes ou boas práticas culinárias.

Outro cenário em que esse tipo de abordagem faz bastante sentido é quando seu portal traz um conteúdo estruturado que depende de uma sequência. Como exemplo, vamos tomar um cenário de videoaulas. Nesse caso, a área de recomendação continuaria apresentando os vídeos sugeridos pelos algoritmos de machine learning, mas a área editorial apresentaria os demais vídeos da sequência. Isso porque é muito comum uma videoaula ser dividida em várias partes, o que torna importante deixar acessível ao usuário os próximos vídeos dessa sequência.

### Avaliando seu recomendador

Embora o Mahout disponibilize a interface **RecommendedEvaluator**, que auxilia na avaliação dos resultados, ela se restringe a fazer análises estatísticas sobre as recomendações produzidas pelos algoritmos. A verdadeira avaliação dos resultados deve ser feita através do monitoramento dos acessos, de modo que seja possível medir os resultados trazidos pelo portal antes e depois do recomendador.

Esses resultados podem ser analisados de diversas formas e em vários contextos. O mais comum é avaliar a taxa de conversão trazida pelas recomendações. Em um portal de conteúdos, por exemplo, podemos medir a frequência com que os usuários acessam os links recomendados e se isso influenciou sua forma de uso do portal.

Além disso, responder algumas perguntas pode nos ajudar a medir a eficiência do recomendador. Vejamos algumas delas:

- Dos itens recomendados, quantos foram acessados por cada usuário?
- O tempo de permanência do usuário no portal aumentou ou diminuiu?

- Se o tempo aumentou, devemos verificar se esse aumento é porque o usuário descobriu conteúdos que não conhecia ou se ele demorou mais tempo para localizar o que estava procurando;

- Se o tempo diminuiu, pode ser porque ele tenha encontrado mais rapidamente os assuntos de seu interesse.

- Com que frequência os usuários retornam ao portal?

As respostas a essas perguntas podem indicar possíveis melhorias a serem realizadas no portal, melhorias essas que podem estar relacionadas diretamente à qualidade da recomendação ou não.

Em muitos casos, chegamos à conclusão de que é preciso repensar a experiência do usuário com o portal, tornando o conteúdo recomendado mais acessível e evidente.

## Nota

A cada ajuste, seja na experiência do usuário ou no recomendador, novas medições devem ser realizadas para avaliar a efetividade das mudanças.

Existem diversas ferramentas que podem auxiliar durante o processo de medições. A mais popular é o Google Analytics. Essa solução é um serviço oferecido pelo Google que permite associar uma etiqueta (tag) a cada conteúdo que se deseja extrair medições. Com isso, é possível compreender o que os visitantes fazem dentro do portal, bem como a taxa de conversão de diferentes áreas do site. Esse tipo de informação é muito importante, pois pode indicar a melhor área do portal onde colocar o recomendador de conteúdos.

## Processamento distribuído com o Hadoop

Os exemplos de código das **Listagens 5 e 6** foram implementados a partir da API do Mahout e representam um cenário onde os dados são processados de forma não distribuída, ou seja, são exemplos de código que tratam de problemas onde todo o conjunto de dados pode ser carregado na memória de um único computador. Infelizmente, essa não é a realidade da maioria dos cenários onde os recomendadores de conteúdo são aplicados, uma vez que coletar informações sobre os acessos dos usuários produz uma grande quantidade de dados.

Quando precisamos processar muitos dados é comum utilizar técnicas de processamento distribuído, mas isso traz complexidades para o desenvolvimento do projeto. Esses desafios, no entanto, podem ser simplificados com o uso do Hadoop, que permite realizar processamento distribuído e provê performance e escalabilidade para a aplicação.

Para utilizar o Hadoop em um projeto precisamos especificar dois parâmetros para o sistema de arquivos, o que pode ser feito editando o arquivo `<diretório-hadoop>/conf/core-site.xml`. Esse arquivo é responsável por toda a configuração do HDFS, além dos daemons de Map Reduce (Job tracker e task tracker).

A configuração do HDFS depende de dois parâmetros específicos: **name** e **value**, conforme apresentado na **Listagem 7**.

### Listagem 7. Configurando o sistema de arquivos do Hadoop.

```
<?xml version="1.0"?>
<?xmlstylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

Nesse código, configuraremos o sistema de arquivos com o alias padrão `fs.default.name` e indicamos a URL que o Hadoop utilizará para acessar o HDFS: `hdfs://localhost:9000`. Essa URL será adotada pelo Hadoop para acessar o sistema de arquivos sempre que precisar gravar ou ler alguma informação durante o processamento dos dados.

Realizada a configuração, o sistema de arquivos precisa ser formatado. Esse processo criará toda a estrutura necessária para comportar os namenodes e datanodes do Hadoop. A **Listagem 8** traz um exemplo de como formatar o sistema de arquivos.

### Listagem 8. Formatando o sistema de arquivos do Hadoop.

```
$ hadoop namenode -format
INFO namenode.NameNode: STARTUP_MSG:
/*****
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = 187a6099907f/172.17.0.2
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 1.2.1
STARTUP_MSG: java = 1.7.0_80
*****
INFO util.GSet: Computing capacity for map BlocksMap
INFO util.GSet: VM type      = 64-bit
INFO util.GSet: 2.0% max memory = 932184064
INFO util.GSet: capacity     = 2^21 = 2097152 entries
INFO util.GSet: recommended=2097152, actual=2097152
INFO namenode.FSNamesystem: fsOwner=root
INFO namenode.FSNamesystem: supergroup=supergroup
INFO namenode.FSNamesystem: isPermissionEnabled=true
INFO namenode.FSNamesystem: dfs.block.invalidate.limit=100
INFO namenode.FSEditLog: dfs.namenode.edits.toleration.length = 0
INFO namenode.NameNode: Caching file names occurring more than 10 times
INFO common.Storage: Image file /tmp/hadoop-root/dfs/name/current/fsimage
of size 110 bytes.
INFO common.Storage: Storage directory /tmp/hadoop-root/dfs/name has been
successfully formatted.
namenode.NameNode: SHUTDOWN_MSG:
/*****
SHUTDOWN_MSG: Shutting down NameNode at 187a6099907f/172.17.0.2
*****
```

Realizada essa etapa, serão apresentadas algumas informações importantes sobre as diretivas de acesso aos dados. Entre as diretivas podemos destacar as permissões de leitura e escrita no HDFS, que foram concedidas ao usuário que executou a formatação; no nosso caso, o usuário root. Isso pode ser observado através do parâmetro `fsOwner`. Também podemos notar que foi criada uma imagem do HDFS, que poderá ser restaurada em caso de falha. O caminho dessa pode ser observado através do parâmetro `Image File`.

Com o sistema de arquivos formatado, podemos executar o Hadoop. Antes disso, porém, precisamos configurar três variáveis de ambiente, conforme o código a seguir:

```
$ export HADOOP_PREFIX=<hadoop_home>
$ export HADOOP_CONF_DIR="$HADOOP_PREFIX/conf"
$ export PATH="$PATH:$HADOOP_PREFIX/bin"
```

Essas variáveis permitem ao Hadoop localizar os arquivos de configuração e os scripts de execução do Map Reduce, normalmente disponíveis em `<hadoop_home>/bin` e `<hadoop_home>/conf`. O script `start-dfs.sh` inicia os serviços do sistema de arquivos (namenode, datanode e secondarynamenode), enquanto o script `start-mapred.sh` inicia os serviços de Map-Reduce (jobtracker e tasktracker).

Com o intuito de simplificar a inicialização dos serviços, o Hadoop disponibiliza o script `start-all.sh`. Esse arquivo inicializa todos os serviços necessários para subir o Hadoop, em sua devida ordem. Um exemplo da execução do script `start-all.sh` pode ser visto na **Listagem 9**.

#### Listagem 9. Inicializando o Hadoop.

```
01 $ start-all.sh
02
03 starting namenode, logging to <hadoop_home>/logs/
  hadoop-root-namenode-187a6099907f.out
04 ECDSA key fingerprint is 5e:78:1b:7b:80:0c:01:20:c8:c5:d8:2e:30:b9:5f:b5.
05 Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
06 starting datanode, logging to <hadoop_home>/logs/
  hadoop-root-datanode-187a6099907f.out
07 starting secondarynamenode
08 jobtracker
09 starting tasktracker
```

Depois de inicializar os serviços do Hadoop é importante verificar se todos os seus daemons foram executados (*namenode*, *datanode*, *secondarynamenode*, *jobtracker* e *tasktracker*). Outro fator importante a ser observado é que o Hadoop efetuou uma conexão SSH, o que pode ser verificado na linha 04 da **Listagem 9**. Essa é a forma como cada máquina do cluster se comunica com as demais, por isso a importância das diretivas SSH estarem devidamente configuradas.

Agora que o Hadoop já está executando, devemos persistir no sistema de arquivos os dados a serem processados. Para isso, utilizaremos o comando `hadoop fs -put <nome-do- arquivo-local> <nome-do- arquivo-remoto>`.

Para verificar se os dados foram persistidos corretamente, podemos consultar o HDFS através do comando `hadoop fs -ls`. A **Listagem 10** traz um exemplo de como persistir os dados no sistema de arquivos e como consultá-los.

Neste momento podemos notar que o arquivo *u.data* foi persistido e está sob o domínio do usuário root, responsável por realizar a formatação do sistema de arquivos. Esse grupo oferece diversas bases de dados, mas nesse exemplo optamos pela base ml-100k, que como o próprio nome sugere, possui 100 mil registros que representam acessos a conteúdos em vídeo. É importante ressaltar que todas as bases de dados disponibilizadas pela MovieLens estão no formato de entrada exigido pelo Mahout.

Agora que já adicionamos os dados de entrada no HDFS, devemos executar o algoritmo de filtragem colaborativa para processar a lista de recomendações. A **Listagem 11** mostra como fazer isso.

#### Listagem 10. Persistindo e consultando os dados do sistema de arquivos distribuído (HDFS).

```
$ hadoop fs -put u.data u.data
$ hadoop fs -ls

Found 1 items
-rw-r--r-- 3 root supergroup 1979173 /user/root/u.data
```

#### Listagem 11. Executando o algoritmo de filtragem colaborativa como job do Hadoop.

```
$ hadoop jar <mahout_home>/mahout-core-0.9-job.jar
org.apache.mahout.cf.taste.hadoop.item.RecommenderJob -s SIMILARITY_COOC-
CURRENCE --input u.data
--output output

INFO mapred.JobClient: map 100% reduce 100%
INFO mapred.JobClient: Job complete: job_local696313686_0001
INFO mapred.JobClient: Counters: 22
INFO mapred.JobClient: File Output Format Counters
INFO mapred.JobClient: Bytes Written=20361
INFO mapred.JobClient: File Input Format Counters
INFO mapred.JobClient: Bytes Read=1979173
INFO mapred.JobClient: FileSystemCounters
INFO mapred.JobClient: FILE_BYTES_READ=25668657
INFO mapred.JobClient: HDFS_BYTES_READ=3958346
INFO mapred.JobClient: FILE_BYTES_WRITTEN=26013586
INFO mapred.JobClient: HDFS_BYTES_WRITTEN=20361
INFO mapred.JobClient: Map-Reduce Framework
INFO mapred.JobClient: Reduce input groups=1682
INFO mapred.JobClient: Map output materialized bytes=5319
INFO mapred.JobClient: Combine output records=1682
INFO mapred.JobClient: Map input records=100000
INFO mapred.JobClient: Reduce shuffle bytes=0
INFO mapred.JobClient: Physical memory (bytes) snapshot=0
INFO mapred.JobClient: Reduce output records=1682
INFO mapred.JobClient: Spilled Records=3364
INFO mapred.JobClient: Map output bytes=382876
INFO mapred.JobClient: Total committed heap usage (bytes)=562036736
INFO mapred.JobClient: CPU time spent (ms)=0
INFO mapred.JobClient: Virtual memory (bytes) snapshot=0
INFO mapred.JobClient: SPLIT_RAW_BYTES=103
INFO mapred.JobClient: Map output records=100000
INFO mapred.JobClient: Combine input records=100000
INFO mapred.JobClient: Reduce input records=1682
```



## Nota

É importante destacar que o comando exposto na **Listagem 11** executa um algoritmo de recomendação baseada no usuário. Para efetuar uma recomendação baseada no item, devemos utilizar o algoritmo **ItemSimilarityJob**.

Veja que é necessário informar o algoritmo de recomendação (**RecommenderJob**), o tipo de similaridade e os arquivos de entrada e saída (acessados a partir do HDFS). No caso específico do arquivo de saída, estamos nos referindo à lista de recomendação produzida pelo Mahout.

Durante a execução do algoritmo, o Hadoop apresenta detalhes sobre o processo de Map-Reduce e indica a quantidade de bytes de entrada (leitura) e saída (escrita) do HDFS. Isso nos permite observar, na prática, o funcionamento dos job trackers, que estão lendo os dados do sistema de arquivos e registrando os resultados do processamento.

Encerrado o processo, podemos extrair o arquivo de saída do HDFS e analisar os resultados produzidos pelo Mahout. Esse arquivo pode ser recuperado através do comando `hadoop fs -getmerge <nome-do-arquivo-no-hdfs> <nome-do-arquivo-local>`, conforme o exemplo da **Listagem 12**.

## Listagem 12. Recuperando a lista de recomendação.

```
$ hadoop fs -getmerge output output.data  
$ head output.data
```

```
01 1 [234:5.0,347:5.0,237:5.0,47:5.0,282:5.0,275:5.0,88:5.0,515:5.0,514:5.0,121:5.0]  
02 2 [282:5.0,210:5.0,237:5.0,234:5.0,347:5.0,121:5.0,258:5.0,515:5.0,462:5.0,79:5.0]  
03 3 [137:5.0,284:5.0,248:4.87234,14:4.8125,508:4.769231,845:4.75,124:4.703704,  
319:4,150:4.68,591:4]  
04 4 [347:5.0,210:5.0,234:5.0,275:5.0,121:5.0,255:5.0,237:5.0,895:5.0,282:5.0,79:5.0]  
05 5 [275:5.0,79:5.0,514:5.0,282:5.0,121:5.0,12:5.0,258:5.0,237:5.0,234:5.0,117:5.0]  
06 6 [282:5.0,210:5.0,515:5.0,234:5.0,275:5.0,121:5.0,129:5.0,258:5.0,237:5.0,117:  
5.0]  
07 7 [240:5.0,315:5.0,403:5.0,517:5.0,176:5.0,418:5.0,682:5.0,209:5.0,137:  
5.0,235:5.0]  
08 8 [275:5.0,79:5.0,514:5.0,282:5.0,121:5.0,12:5.0,515:5.0,237:5.0,234:5.0,117:5.0]  
09 9 [275:5.0,79:5.0,514:5.0,282:5.0,121:5.0,12:5.0,258:5.0,237:5.0,234:5.0,117:5.0]  
10 10 [462:5.0,275:5.0,515:5.0,514:5.0,234:5.0,282:5.0,129:5.0,258:5.0,237:5.0,  
121:5.0]
```

No arquivo resultante, podemos notar que o identificador do usuário é seguido dos conteúdos e suas respectivas notas para recomendação. Por exemplo, a terceira linha diz para recomendarmos ao usuário 3 os seguintes vídeos: 137, 284, 248, 14, 508, 845, 124, 319, 150 e 591, cada um acompanhado de sua respectiva nota. A classificação que acompanha os vídeos (valor entre 1 e 5) indica a aderência entre o vídeo e o usuário, de modo que quanto maior a classificação, mais aderente esse conteúdo é. Essa informação pode ser utilizada para ordenar os conteúdos no momento da exibição.

A cada dia o mundo digital ganha novos usuários, que ficam mais exigentes com o passar do tempo. Esses usuários consomem diferentes tipos de produtos e serviços, de roupas e acessórios até transações bancárias. Com tanta informação e o tempo cada vez mais apertado, buscamos solucionar nossos problemas de forma rápida, eficiente e simples. Nesse sentido, as soluções baseadas em recomendação de conteúdo são muito importantes, pois apresentam de maneira simples as informações e produtos de interesse dos usuários.

Com as facilidades trazidas por ferramentas como Hadoop e Mahout, criar esse tipo de solução se torna uma tarefa simples e ágil, visto que não precisamos lidar com as complexidades trazidas pelos sistemas de processamento paralelo e distribuído. Através dessas ferramentas, podemos rapidamente incorporar um sistema de recomendação a qualquer portal, seja ele voltado para conteúdo, produtos ou serviços.

Além dessas facilidades, podemos executar os algoritmos do Mahout pelo shell interativo e integrar essas soluções com diversas tecnologias. Como exemplo, vejamos um cenário em nuvem: através de rotinas automatizadas, podemos criar uma instância em nuvem que processe a lista de recomendações e a armazene em um banco de dados. Logo após, podemos disponibilizar uma camada de serviços que consulte esse banco de dados e mostre as sugestões aos usuários. Essa plataforma pode ser fornecida como serviço (SaaS) e atender a diversos portais.



Outra grande vantagem é o suporte do Mahout aos algoritmos de filtragem colaborativa, que permitem mapear os interesses do usuário conhecendo apenas seu histórico de acessos. Isso passa a ser muito interessante, uma vez que não é preciso coletar outras informações sobre o usuário, tais como faixa etária, sexo, entre outras. Saiba, ou lembre-se, que poucos usuários se dispõem a preencher formulários, seja para não se expor ou por uma questão de tempo.

Entre as soluções apresentadas para identificar o usuário, podemos destacar a identificação baseada em um serviço de Cookie Pool, que não necessita da autenticação do usuário. Através desse tipo de serviço podemos atribuir um identificador único para cada usuário e registrar todos os seus itens de interesse, armazenando seu histórico de navegação no site. Isso também permite identificar o usuário quando ele retornar ao site, para efetuar novas recomendações. Entre os problemas associados ao cookie pool, podemos destacar a navegação anônima e a possibilidade de o usuário limpar os cookies de seu navegador. Outro problema é que não existe histórico de consumo quando o usuário visita o site pela primeira vez, e nesse caso não sabemos o que recomendar a ele. Em situações como esta, é comum apresentar os itens mais acessados do portal.

Para encerrar, quanto às ferramentas, podemos destacar como pontos fracos o fato do Hadoop não oferecer um bom monitoramento de tarefas, assim como o tempo gasto para consultar os dados do HDFS, que está sujeito à latência da rede e pode comprometer o desempenho das consultas.

## Autor



**Everton Gago**

@evertongjúnior

É pesquisador na área de inteligência artificial e machine learning. Seu principal interesse é em algoritmos matemáticos que reproduzem os aspectos biológicos da aprendizagem e da auto-organização, quando aplicados em problemas de mineração de dados.



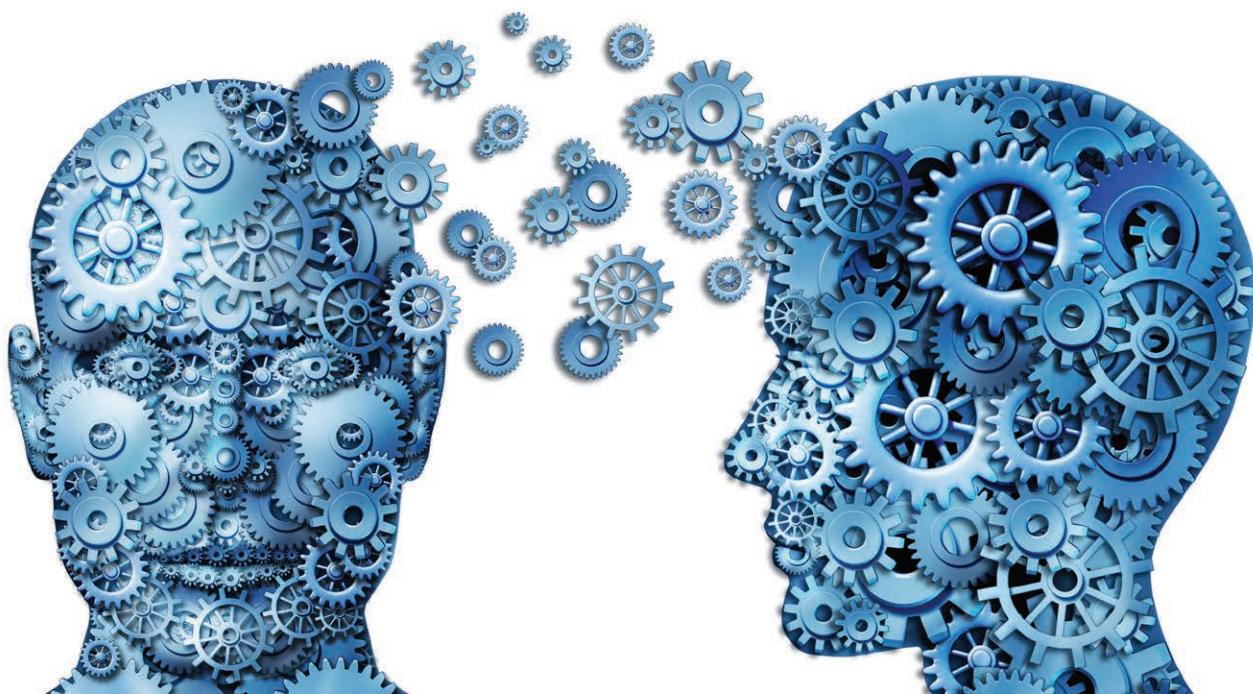
## Links:

### Dados da MovieLens.

<http://grouplens.org/datasets/movielens/100k/>

### Repositório Maven do Apache Mahout, versão 0.10.

<http://mvnrepository.com/artifact/org.apache.mahout/mahout/0.10.0>



# Contextos e Injeção de Dependência (CDI) para Java EE

**Uma introdução aos serviços disponíveis na especificação CDI que podem ajudar na estruturação de sua aplicação e redução do acoplamento entre componentes**

Este artigo apresenta uma visão geral de conceitos relacionados à injeção de dependência e como seu uso pode ser útil na construção de aplicações Java simples e bem estruturadas. Veremos como é possível construir soluções onde o gerenciamento de conexões entre componentes é feito de forma simplificada e flexível, reduzindo o acoplamento entre as diversas partes do sistema.

Além disso, vamos ver como a injeção de dependência (DI) disponível na plataforma Java EE facilita a organização de componentes. Organizar e gerenciar o relacionamento entre componentes de uma solução é uma condição fundamental para manter a mesma de forma organizada e lógica.

Como sabemos, um projeto de software consiste em transformar um conjunto de requisitos em uma solução. E para atingir esse objetivo, muitas vezes separamos esses requisitos em diversos elementos, ou componentes, que possuem um conjunto de responsabilidades e colaboram para viabilizar as diferentes funcionalidades. Observe que usamos aqui a noção de componente de forma bem abrangente (e vamos continuar utilizando no texto a seguir), não implicando na adoção de uma determinada estrutura ou tecnologia, mas apenas a ideia de que a realização dos requisitos seja feita fragmentando e realizando os mesmos em diversos módulos, ao invés de em um único elemento.

## Fique por dentro

Este artigo é útil para o desenvolvedor que queira entender como aplicar a especificação CDI no gerenciamento de dependências entre elementos de uma aplicação. Essa especificação pode facilitar o objetivo da construção de soluções com alta coesão e baixo acoplamento, através do gerenciamento de dependências de forma declarativa e diversos serviços de suporte ao ciclo de vida de componentes.

Outro nome para essa separação é modularização, técnica usada há bastante tempo como parte da estratégia de separar grandes problemas em problemas menores e mais fáceis de serem resolvidos, alocando responsabilidades específicas e relacionadas a determinados componentes.

Ao realizarmos essa divisão de responsabilidades entre diversos componentes da solução, podemos avaliar a qualidade dessa separação e da interação entre os componentes através de duas medidas: coesão e acoplamento. De forma geral, devemos buscar projetar e construir componentes com alto grau de coesão e baixo acoplamento, pois isso facilita a compreensão e manutenção da solução.

## Coesão e acoplamento

Coesão e acoplamento não são conceitos novos no desenvolvimento de software e muito menos exclusivos da plataforma Java. Ao contrário, são conceitos que estão por aí desde meados de 1960

e que, volta e meia, são alvo de novas e engenhosas propostas de solução. Para entender como a injeção de dependência se encaixa nesse contexto, vamos falar um pouco sobre coesão, acoplamento e outros conceitos relacionados.

E por que esses conceitos são importantes e recorrentes? Porque nenhum componente é uma ilha: Qualquer sistema, a partir de um grau mínimo de complexidade, será composto por diversas peças que colaboram entre si, e gerenciar essa complexidade em diversos níveis é um problema recorrente no desenvolvimento de software. Considere as dependências que existem entre métodos, classes, componentes, bibliotecas e aplicações. Para cada um desses tipos de relacionamento podemos ter estratégias distintas para facilitar a redução de acoplamento e aumentar a coesão.

### Coesão

Coesão mede o quanto as responsabilidades de um determinado componente estão relacionadas ao mesmo. Quanto maior a coesão, melhor o projeto do elemento.

Como definimos um componente de forma ampla, vamos usar como exemplo um método de uma classe. Uma forma de avaliar o nível de coesão do mesmo seria a seguinte: Todas as linhas de código desse método são relacionadas ao objetivo do mesmo? Você consegue dizer que o método possui uma única responsabilidade, e que toda a sua estrutura contribui para essa responsabilidade, ou consegue identificar diversas responsabilidades distintas nesse método?

Por exemplo, em um mesmo método, você pode acessar um banco de dados, validar as informações contra um serviço externo, formatar os dados para exibição em uma tela e verificar se o usuário dessa aplicação possui perfil de administrador ou usuário comum. Muita coisa? Sim, e você pode identificar facilmente métodos assim, verificando, por exemplo, os que são mais extensos ou que fazem diversas “coisas”. Esses são métodos que comumente apresentam baixa coesão, característica normalmente encontrada onde diversas responsabilidades são “amarradas” em um mesmo elemento.

De maneira mais ampla, dessa vez analisando a classe, considere se os métodos da mesma são relacionados. Uma mesma classe é responsável pelos diversos aspectos descritos anteriormente, ou você consegue determinar a responsabilidade de uma classe como única? A facilidade com que consegue responder a essa pergunta pode determinar se sua classe é coesa ou não. E qual a vantagem de ter uma classe coesa? Quanto menos você precisar navegar na sua estrutura de projeto para tratar de um determinado assunto, mais fluente e prática é a manutenção do seu sistema.

### Tipos de coesão

Entre os tipos de coesão que podemos citar, temos:

- **Acidental:** Esse tipo de coesão, como sugerido pelo nome, ocorre sem que seja feito qualquer planejamento. As responsabilidades são agrupadas não de forma projetada, mas sim por conveniência, como, por exemplo, uma classe utilitária com diversas funções não relacionadas a um único assunto;

- **Comunicação:** Aqui, os elementos são agrupados em função de atualizarem a mesma estrutura de dados ou produzirem os mesmos dados de saída (por exemplo, diversos métodos que geram relatórios correlatos);

- **Funcional:** Caso os elementos existam no componente para atender a uma mesma função. Para esse tipo de coesão, deve ser possível descrever a função do componente de forma única, mesmo que existam diversos elementos para atender a essa função. Por exemplo, um módulo que possui um conjunto de funções relacionadas entre si para manter os dados de um catálogo de produtos;

- **Lógica:** Podemos dizer que um módulo possui coesão lógica quando o mesmo realiza funções similares. Considere, por exemplo, um módulo responsável apenas por funções de validação de dados ou tratamento de erros;

- **Procedural:** Para existir coesão procedural é necessário que os elementos do componente sejam parte de um algoritmo ou procedimento executado em etapas para se chegar a um fim. Por exemplo, o processo de geração de um arquivo, que segue passos como: abrir o arquivo, posicionar um cursor no ponto de escrita, escrever o conteúdo e fechar o arquivo;

- **Sequencial:** Uma forma semelhante à coesão procedural, com o detalhe de que na coesão sequencial os dados de saída de um elemento do componente são os dados de entrada para um próximo elemento, e assim sucessivamente;

- **Temporal:** Nesse tipo de coesão, são agrupadas ações realizadas dentro de um mesmo período de tempo. Um exemplo comum para esse tipo de coesão é o agrupamento de funções de inicialização e encerramento de um sistema.

### Acoplamento

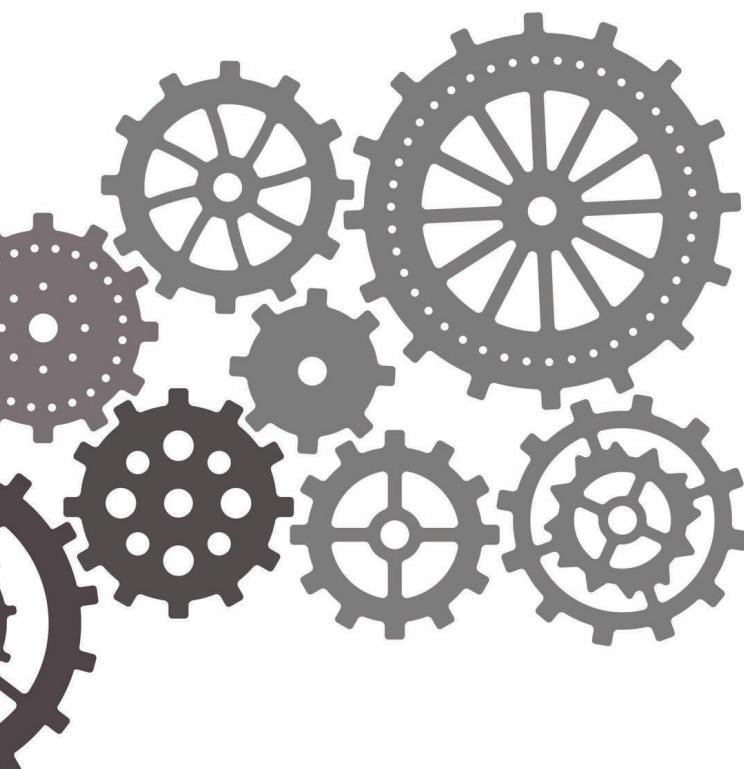
Enquanto a coesão mede o nível de dependência entre os elementos de um determinado componente, o acoplamento mede o nível de interdependência entre os componentes em um contexto (sistema, programa, aplicação ou qualquer que seja a estrutura da solução). Quanto maior o nível de acoplamento, maior é a probabilidade de que a alteração em um elemento da aplicação afete outros. Logo, quanto menor o acoplamento, melhor a solução. Entretanto, como falamos anteriormente, nenhum componente (útil) é uma ilha, e deve existir algum acoplamento para que o mesmo possa ser utilizado.

Uma forma prática e simples (porém não absoluta) para medir o acoplamento de uma classe Java é observar os imports da mesma: Quanto mais elementos externos (classes, métodos estáticos) uma classe importa, mais ela depende de elementos externos para realizar suas funções. No exemplo que falamos na seção sobre coesão, aquele método que “faz muita coisa” provavelmente geraria a necessidade de importação de diversos outros pacotes ou classes para realizar tanto. E se fizer uso de bibliotecas externas para tal, a tendência é só aumentar.

## Tipos de acoplamento

Alguns tipos de acoplamento são conforme a seguir:

- **Conteúdo:** Quando um componente depende de forma direta de dados ou do funcionamento interno de outro. Pode ocorrer, por exemplo, em um cenário em que um componente “configura” um outro componente para utilizar o mesmo;
- **Comum:** Nesse tipo de acoplamento, os componentes compartilham dados através de variáveis globais ou algo equivalente. Mudar esse repositório de dados comum afeta todos os módulos dependentes do mesmo;
- **Externo:** Nesse tipo os componentes compartilham um formato de dados comum, normalmente definido por um protocolo de comunicação ou interface;
- **Controle:** Esse tipo de acoplamento ocorre quando um componente controla o fluxo de execução de outro, muitas vezes definindo através de “flags” qual deve ser o fluxo de execução de um determinado procedimento. É muito comum entre componentes que se comunicam utilizando parâmetros booleanos para fazer uma coisa caso o valor seja verdadeiro, ou outra, caso o valor seja falso;
- **Dados:** Um tipo de acoplamento muito comum, em que componentes compartilham dados, por exemplo, através de parâmetros passados de um método para outro. Um tipo mais específico desse tipo de acoplamento são estruturas usadas para a passagem de dados entre diversos componentes, em que cada componente usa apenas parte das estruturas.



## Injeção de dependência

Como falamos anteriormente, devemos projetar componentes com alto grau de coesão e baixo acoplamento, pois isso nos permite ter estruturas simples, de mais fácil manutenção. A injeção de dependências nos auxilia justamente nos aspectos relacionados ao acoplamento. E mesmo não provendo recursos para atingirmos a coesão desejada em nossos módulos, nos permite identificar componentes cuja coesão seja deficiente, através, por exemplo, da identificação de muitos elementos externos para se realizar alguma atividade.

Para utilizar a injeção de dependência de forma efetiva, devemos separar as diversas responsabilidades entre os componentes do sistema, tendo em vista, entre outras coisas, o tipo de coesão desejada. A partir daí, cada componente poderá fazer uso dos demais elementos da aplicação, o que irá estabelecer as conexões necessárias entre os mesmos.

Para ilustrar o acoplamento entre classes antes do uso do CDI, veja o exemplo da **Listagem 1**, onde ilustramos duas classes que colaboram para mostrar o último evento registrado em um repositório de dados.

### Listagem 1. Exemplo de classes acopladas.

```
01 public class EventViewer {  
02     03     private EventRepository eventStore;  
04  
05     06     public EventViewer() {  
06         eventStore = new FastEventRepository();  
07     }  
08  
09     10     public void showLastEvent() {  
10         System.out.println(eventStore.getLastEvent());  
11     }  
12 }  
13  
14 public interface EventRepository {  
15     String getLastEvent();  
16 }  
17  
18 public class FastEventRepository implements EventRepository {  
19  
20     @Override  
21     public String getLastEvent() {  
22         return "Nothing happened today";  
23     }  
24 }
```

Nesse código, a classe **EventViewer** é responsável por exibir o último evento registrado em um repositório (método **showLastEvent()**). Para obter esse evento, ela utiliza um objeto da classe **FastEventRepository** (linha 10). Dessa forma, separamos a responsabilidade de obter a informação da de exibir a informação em uma tela, e o resultado é uma estrutura coesa, sem diversas responsabilidades em cada elemento.

Observe que usamos uma boa prática de programação utilizando interfaces, quando declaramos em **EventViewer** (linha 03) um

atributo do tipo **EventRepository** e não de uma implementação específica. Essa prática reduz o acoplamento, pois a classe cliente (**EventViewer**) depende de uma interface (**EventRepository**), não importa qual seja a sua implementação. Dessa forma, os métodos que dependem de uma determinada interface não estão acoplados a uma implementação da mesma.

Até aqui, conseguimos desacoplar os métodos de uma classe cliente (**EventViewer**) da sua dependência (**EventRepository**), mas a classe ainda está acoplada a uma implementação, pois quando instanciamos o atributo **eventStore** (linha 06) referenciamos diretamente **FastEventRepository**, e a classe **EventViewer** passa a “saber” da existência dessa implementação, o que mantém o acoplamento.

Declarar o atributo **eventStore** como **EventRepository** reduz o acoplamento, pois a instância concreta é mencionada uma única vez e poderia ser substituída em um único ponto. Mas e se essa e outras interfaces forem usadas em diversos pontos do sistema, sendo que cada classe que utiliza uma interface precisa instanciar uma implementação? Nesse caso, esse “único ponto” de acoplamento iria se multiplicar em vários outros espalhados pelo sistema.

Antes do uso de injeção de dependências, alguns padrões comumente usados para resolver esse problema eram denominados genericamente de Factory (**Listagem 2**), para objetos criados dentro da própria aplicação, e Locator (**Listagem 3**), para dependências externas à aplicação, como um DataSource obtido a partir do servidor de aplicação via JNDI.

Na **Listagem 2**, utilizamos uma classe factory (**RepositoryFactory**) responsável por criar a instância da interface **EventRepository**, da qual nosso **EventViewer** depende (linha 06).

Ena **Listagem 3**, utilizamos um service locator, de forma idêntica a uma classe factory (linha 06), mas cuja implementação se distingue em função do service locator obter o objeto que precisamos a partir de um serviço externo, como um contexto JNDI.

Os dois exemplos mostrados nessas listagens resultam no mesmo resultado, onde temos uma classe (**EventViewer**) dependente de uma interface (**EventRepository**), mas não acoplada a uma implementação (**FastEventRepository**). Para isso adicionamos um novo elemento na aplicação (Service Locator ou Factory), responsável por instanciar ou localizar as diversas implementações de interfaces utilizadas no sistema.

## Contextos e Injeção de Dependências – CDI

No caso de aplicações Java, apresentamos princípios associados a essa busca, como o uso de interfaces e padrões como factory e service locator.

A especificação CDI provê serviços que ajudam na construção de estruturas mais simples e desacopladas sem aumentar de forma significativa a complexidade do código. Ela padroniza técnicas de redução de acoplamento, como o uso de interfaces, factory, service locator, e introduz o conceito de objetos gerenciados (chamados de beans), que podem ser injetados em objetos através de referências incluídas nos mesmos de forma declarativa.

---

### Listagem 2. Exemplo do padrão factory method.

---

```
01 public class EventViewer {  
02     03     private EventRepository eventStore;  
04  
05     06     public EventViewer() {  
06         eventStore = RepositoryFactory.newEventRepository();  
07     }  
08  
09     10    public void showLastEvent() {  
10         System.out.println(eventStore.getLastRecordEvent());  
11     }  
12 }  
13  
14 public class RepositoryFactory {  
15  
16     17     public static EventRepository newEventRepository() {  
17         return new FastEventRepository();  
18     }  
19 }
```

---

### Listagem 3. Exemplo do padrão service locator.

---

```
01 public class EventViewer {  
02     03     private EventRepository eventStore;  
04  
05     06     public EventViewer() {  
06         eventStore = ServiceLocator.newEventRepository();  
07     }  
08  
09     10    public void showLastEvent() {  
10         System.out.println(eventStore.getLastRecordEvent());  
11     }  
12 }  
13  
14 public class ServiceLocator {  
15  
16     17     public static EventRepository newEventRepository() {  
17         Context context;  
18         try {  
19             context = new InitialContext();  
20             return (EventRepository) context.lookup("EventRepository");  
21         } catch (NamingException e) {  
22             throw new IllegalStateException("Erro ao obter EventRepository", e);  
23         }  
24     }  
25 }
```



# Contextos e Injeção de Dependência (CDI) para Java EE

Algumas das vantagens do uso de CDI em aplicações Java são:

- Simplicidade para a definição de objetos gerenciados (beans): quase todas as classes podem ser usadas como objetos gerenciados, desde que tenham um construtor sem parâmetros ou que possuam a anotação `@Inject`, não sendo necessário implementar interfaces ou utilizar anotações da especificação CDI para a definição de beans. Além disso, EJBs também podem ser utilizados como beans, utilizando tanto as anotações da especificação EJB quanto da especificação CDI. Dessa forma o uso do CDI não é intrusivo;
- Permite aos elementos da aplicação indicar suas dependências de forma declarativa, e à aplicação, selecionar diferentes implementações dessas dependências;
- Diversas tecnologias da plataforma Java possuem integração com CDI, como, por exemplo, JSF e JSP, que permitem a referência a objetos gerenciados através da Unified Expression Language;
- Padronização do gerenciamento do ciclo de vida de componentes, com a definição de escopo para os objetos criados;
- Inclui outros padrões utilizados na composição de aplicações de forma desacoplada, como Decorators, Interceptors e Events, usados para estender a funcionalidade de beans de forma não intrusiva.

## CDI básico: a anotação `@Inject`

A anotação `@Inject` é usada para indicar que nossa classe possui uma determinada dependência. A partir daí o ambiente de execução CDI tem condições de tentar obter um objeto que atenda a essa dependência. Usando o mesmo exemplo de classes que colaboraram para mostrar o último evento registrado em um repositório de dados, o resultado do uso de `@Inject` pode ser visto na [Listagem 4](#).

**Listagem 4.** Exemplo de uso da anotação `@Inject`. Sinaliza que a classe depende de um componente.

```
01 import javax.inject.Inject;
02
03 public class EventViewer {
04
05     private EventRepository eventStore;
06
07     @Inject
08     public EventViewer(EventRepository eventStore) {
09         this.eventStore = eventStore;
10     }
11
12     public void showLastEvent() {
13         System.out.println(eventStore.getLastRecordEvent());
14     }
15 }
```

Nesse código, o uso da anotação `@Inject` (linha 07) é suficiente para que o runtime do CDI tente obter um objeto que implemente a interface `EventRepository` quando uma instância de `EventViewer` for criada. Não são necessárias alterações na interface que define nosso componente (`EventRepository`) e nem na implementação da mesma (`FastEventRepository`). Apenas a classe dependente (`EventViewer`) declara qual a sua necessidade.

De forma mais detalhada, o runtime CDI irá procurar por uma classe que implemente a interface `EventRepository` e criar uma instância da mesma para “injetar” esse objeto no atributo `eventStore`, através do construtor da classe `EventViewer` (linha 09). Mais à frente, veremos como o CDI determina a classe que será instanciada, no caso de mais uma classe implementar tal interface.

A injeção de dependência pode ser feita de três formas distintas, a saber:

- **Via construtores:** Como vimos na [Listagem 4](#), podemos injetar dependências através de um construtor anotado com `@Inject`. Uma classe pode ter apenas um construtor com essa anotação;
- **Via métodos de inicialização:** Outra forma de declarar dependências em uma classe é definir um método de inicialização com parâmetros de injeção, indicando quais são os objetos dos quais o nosso componente depende, conforme a [Listagem 5](#), onde o método de inicialização `setEventRepository()` é anotado com `@Inject` para que as dependências da classe sejam injetadas naquele ponto;

**Listagem 5.** Injeção de dependência com método de inicialização.

```
01 public class EventViewer {
02
03     private EventRepository eventStore;
04
05     @Inject
06     public void setEventRepository(EventRepository eventStore) {
07         this.eventStore = eventStore;
08     }
09 }
```

- **Via injeção direta de atributos:** Além de construtores e métodos específicos, podemos também utilizar a anotação `@Inject` diretamente em atributos de uma classe, como demonstra a [Listagem 6](#).

**Listagem 6.** Injeção de dependência direta em atributos.

```
01 public class EventViewer {
02
03     @Inject
04     private EventRepository eventStore;
05
06 }
```

O estilo de injeção a ser utilizado é uma questão de preferência, mas o uso de um construtor ou método de inicialização deve ser preferido, visto que deixa explícito na interface do componente quais são as suas dependências, além de facilitar a substituição dos objetos injetados durante testes unitários, por exemplo.

## Escopo e ciclo de vida

Para criar os objetos dos quais as classes de uma aplicação dependem, o ambiente de execução do CDI precisa identificar o escopo no qual o objeto criado se encaixa e qual o ciclo de vida do mesmo, de forma que os objetos criados existam pelo tempo necessário para atender a um determinado contexto.

Escopo	Anotação	Círculo de vida
Requisição	@RequestScoped	O existe para uma única requisição HTTP.
Sessão	@SessionScoped	O objeto gerenciado existe para diversas requisições, em uma sessão HTTP.
Aplicação	@ApplicationScoped	O objeto gerenciado é compartilhado por todas as sessões HTTP da aplicação.
Dependente	@Dependent	Nesse caso, o objeto gerenciado tem o mesmo escopo de outro componente que o utiliza. Esse é o escopo padrão, caso nenhum outro seja definido para um determinado componente.
Conversação	@ConversationScoped	Esse escopo é semelhante ao escopo de sessão, pois o objeto gerenciado existe para diversas requisições. Entretanto, o ciclo de vida desse escopo é controlado pelo desenvolvedor, que define sua criação e quando o mesmo é finalizado de forma explícita.

**Tabela 1.** Escopos disponíveis para objetos criados no ambiente de execução CDI

Isso faz muito sentido em uma aplicação web, onde alguns objetos só precisam existir durante uma requisição do usuário, enquanto outros são diretamente rastreáveis para a sessão do mesmo, ou ainda compartilhados por toda a aplicação. A **Tabela 1** descreve os escopos disponíveis e onde são adequados.

Outros tipos de componentes da plataforma Java EE, como EJBs ou servlets, não possuem um escopo definido. Eles podem ser objetos com ou sem estado (*Stateful* ou *Stateless*) ou ainda compartilhados por todos os clientes (como EJB Singleton). Contudo, uma vez que esses componentes sejam usados como objetos gerenciados, serão associados a um contexto e ciclo de vida de forma automática e bem definida.

Além dos contextos pré-definidos, descritos na **Tabela 1**, a especificação CDI também suporta a criação de novos escopos, definidos pelo desenvolvedor. Isso pode ser feito através da criação de anotações específicas para os novos escopos, conforme demonstra a **Listagem 7**.

**Listagem 7.** Definição de um novo escopo, através da anotação @ScopeType.

```
01 @ScopeType
02 @Retention(RUNTIME)
03 @Target({TYPE, METHOD})
04 public @interface CacheAwareScoped { }
```

Para que o novo escopo possa ser usado, é necessário definir um contexto para o mesmo. Essa é uma parte da especificação voltada para o desenvolvimento de frameworks, visto que os escopos pré-definidos já atendem à maioria dos casos de uso para os quais essa tecnologia foi desenvolvida.

Além das anotações para declaração de dependências e escopo, podemos utilizar **@PostConstruct** e **@PreDestroy** nos objetos gerenciados para realizar ações vinculadas ao ciclo de vida dos mesmos, em sua inicialização e finalização, como ilustrado na **Listagem 8**. Essas anotações fazem parte de outra especificação, a JSR-250, *Common Annotations for the Java Platform*, e se integram de forma transparente aos objetos gerenciados através de CDI.

Para realizar ações em fases do ciclo de vida dos objetos, o processo é simples: para inicialização do seu objeto, crie um método e aplique a anotação **@PostConstruct**. Esse método será chamado antes do objeto ser disponibilizado pelo ambiente de execução CDI para seus dependentes.

De forma análoga, para realizar alguma limpeza antes de um objeto gerenciado ser destruído, por exemplo, basta criar um método e aplicar a anotação **@PreDestroy**. Assim, esse método será executado antes do objeto ser destruído pelo runtime do CDI.

**Listagem 8.** Anotações de ciclo de vida.

```
01 @PostConstruct
02 public void init () {
03     this.openConnection();
04 }
05
06 @PreDestroy
07 public void shutdown () {
08     this.cleanup();
09 }
```

#### Beans pré-definidos e resources

Além de quase todo tipo de classe que pode ser utilizada como objeto gerenciado, existem alguns beans pré-definidos, disponibilizados pelo CDI, a saber:

- Classes da Java Transaction API, como **UserTransaction** e **Principal**;
- Classes da especificação Bean Validation;
- Recursos de ambiente, como DataSources, e objetos de mensageria, como Topics e Queues JMS;
- Contextos de persistência, que fazem parte da especificação Java Persistence API (JPA).

Enfim, a variedade de objetos de outras especificações do Java ilustra o nível de integração entre o CDI e demais componentes da plataforma.



## Produtores e qualificadores

O principal aspecto do projeto de um componente é sua interface, a qual define os serviços que o mesmo deve prover ou suas responsabilidades. A realização desse componente pode ser feita por diversas classes que implementam a interface do mesmo. Para criar instâncias gerenciadas dessas classes podemos utilizar métodos denominados produtores, que nada mais são do que a versão do CDI para o padrão factory, que discutimos anteriormente.

Para definir qual das implementações de um componente é aplicável a um contexto do sistema, podemos utilizar qualificadores, que funcionam para selecionar uma das implementações de forma declarativa. Em combinação com os métodos produtores, temos um modelo flexível de criação de componentes, onde a criação de objetos é desacoplada do seu uso.

Por exemplo, considere o uso de um componente para obter o endereço de um serviço utilizado por uma aplicação. Podemos ter mais de um endereço para esse serviço, e o que determina qual devemos utilizar é o tipo de classe que utiliza esse endereço, como uma classe de teste, ou um serviço da aplicação (vide [Listagem 9](#)).

Na classe **ApplicationService**, o endereço que deve ser injetado (linha 06) é o do ambiente de execução da aplicação, enquanto a classe **TestCase** deve utilizar o endereço de testes (linha 16).

Para definir qual configuração utilizar, aplicamos um dos qualificadores na declaração da dependência; neste caso, **@RuntimeConfig** (linha 05), indicando que queremos que seja injetado um valor classificado com esse qualificador. Caso não seja especificado nada no ponto de injeção da dependência (linha 15), será usado o default.

### Listagem 9. Exemplo de uso de qualificadores de componentes.

```
01 @Singleton
02 public class ApplicationService {
03
04     @Inject
05     @RuntimeConfig
06     private String serviceAddress;
07
08     public void doSomethingNice() {
09         System.out.println(serviceAddress);
10    }
11}
12
13 public class TestCase {
14
15     @Inject
16     private String serviceAddress;
17
18     @Test
19     public void doSomethingNice() {
20         System.out.println(serviceAddress);
21    }
22}
```

Observe também que não há nenhuma referência para a classe **ApplicationConfig** nas classes **ApplicationService** e **TestCase**, que utilizam as configurações fornecidas por essa classe. Dessa forma, os produtores de objetos gerenciados são desacoplados de quem os utiliza.

A mesma coisa pode ser feita para se produzir e injetar objetos complexos com base no ambiente de execução, como um **DataSource** para o ambiente de produção, ou banco de dados em memória para testes integrados.

Para diferenciar os métodos responsáveis por produzir os endereços utilizados, aplicamos os mesmos qualificadores na classe **ApplicationConfig** (vide [Listagem 10](#)), responsável pela produção dos endereços: **RuntimeConfig** (linha 14) e **TestConfig** (linha 07). Esses qualificadores indicam o tipo de ambiente ao qual os endereços retornados por esses métodos se destinam. Observe que a classe **ApplicationConfig** produz Strings em dois métodos distintos, um deles anotado com **@RuntimeConfig**, definido pelo desenvolvedor, e outro com **@Default**, que faz parte do pacote **javax.enterprise.inject**.

Os qualificadores são anotações usadas para classificar um componente onde o mesmo é produzido ou para selecionar uma determinada implementação do componente usando o qualificador como critério de seleção. Alguns exemplos de qualificadores são apresentados na [Listagem 11](#).

### Listagem 10. Exemplo de classe produtora de objetos gerenciados.

```
01 import javax.enterprise.inject.Default;
02 import javax.enterprise.inject.Produces;
03
04 public class ApplicationConfig {
05
06     @Produces
07     @TestConfig
08     @Default
09     public String simulationServiceAddress() {
10         return "http://simulation.coolapplication";
11     }
12
13     @Produces
14     @RuntimeConfig
15     public String productionServiceAddress() {
16         return "http://production.coolapplication";
17     }
18}
```

### Listagem 11. Exemplos de qualificadores.

```
01 @Qualifier
02 @Retention(RUNTIME)
03 @Target({ FIELD, TYPE, METHOD, PARAMETER })
04 public @interface RuntimeConfig {} 
05
06 @Qualifier
07 @Retention(RUNTIME)
08 @Target({ FIELD, TYPE, METHOD, PARAMETER })
09 public @interface TestConfig {}
```

Outro aspecto demonstrado nesse exemplo é que os métodos produtores de objetos gerenciados não estão limitados à construção de objetos gerenciados. Eles também podem ser utilizados para produzir objetos que não são gerenciados, como Strings ou outras classes que nem possuem anotações relacionadas a CDI. Dessa forma, essa especificação não impõe restrições como a obrigatoriedade de anotar todas as classes cuja interdependência você queira gerenciar. Isso pode ser útil, por exemplo, para gerenciar dependências para classes que não fazem parte dos fontes do seu projeto, como uma biblioteca externa.

### JSF e Expression Language

Algo bastante interessante na especificação CDI é que existem poucas restrições para os tipos de classes que podem usar ou serem usadas como beans gerenciados. Além disso, a tecnologia possui um bom nível de integração com outros componentes da plataforma Java EE, inclusive se adaptando ao contexto. Por exemplo, podemos utilizar objetos gerenciados por CDI em um contexto onde expression language esteja disponível, como uma página construída com Facelets. Para isso, aplicamos a anotação **@Named** em uma classe, como ilustrado na [Listagem 12](#). Um pequeno detalhe é que ao usar **@Named** sem nenhum parâmetro, o nome do bean será o nome simples da classe, com a primeira letra em minúsculo.

Ainda no exemplo da [Listagem 12](#), usamos a anotação CDI **@RequestScoped** para configurar o ciclo de vida do componente, indicando que deve ser criado um novo objeto para atender cada requisição HTTP feita a uma view que utilize o mesmo, de forma integrada ao contexto da aplicação web.

#### Listagem 12. Anotação @Named, para uso de objeto gerenciado com JSF.

```
01 @Named
02 @RequestScoped
03 public class EventViewer {
04
05     private EventRepository eventStore;
06
07     @Inject
08     public setEventRepository(EventRepository eventStore) {
09         this.eventStore = eventStore;
10    }
11    public String getLastEvent() {
12        return eventStore.getLastRecordEvent();
13    }
14}
```

Assim, o objeto gerenciado pode ser referenciado diretamente no Facelet conforme o código:

```
<h:form id="myform">
Last recorded Event <p><h:outputText value="#{eventViewer.lastEvent}"></p>
</h:form>
```

### Estereótipos

Um estereótipo é uma anotação que agrupa um conjunto de anotações, nos permitindo aplicar uma série de classificadores

aos componentes da aplicação utilizando uma única anotação, de maneira consistente e sem repetição. Por exemplo, na [Listagem 13](#), definimos uma anotação **@ViewBean** agrupando as anotações **@Named** e **@RequestScoped**. Assim, ao anotarmos uma classe com **@ViewBean**, serão aplicadas todas as anotações que fazem parte do estereótipo.

#### Listagem 13. Exemplo de uso de estereótipos, recurso que permite o agrupamento de anotações.

```
01 @Named
02 @RequestScoped
03 @Stereotype
04 @Target(TYPE)
05 @Retention(RUNTIME)
06 public @interface ViewBean{}
07
08 @ViewBean
09 public class EventViewer {...}
```

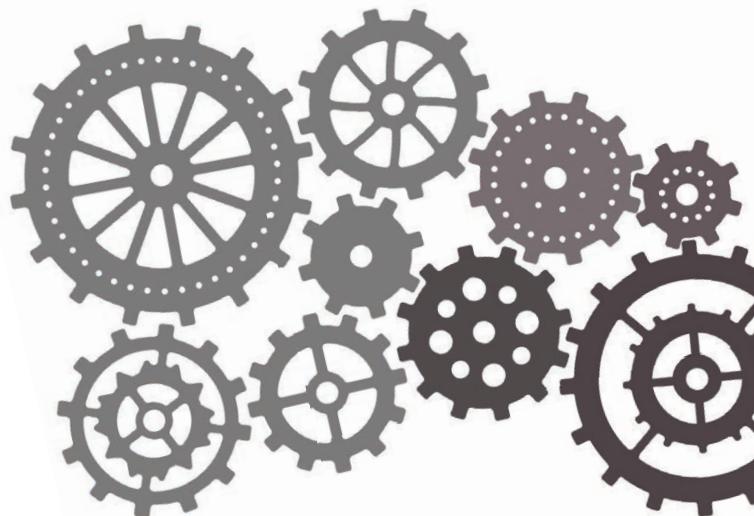
### EJB

Falamos anteriormente que os objetos gerenciados criados via CDI se adaptam ao contexto, mas o que isso significa?

Ilustramos a resposta a essa pergunta na [Listagem 14](#), onde utilizamos a anotação **@Inject** para injetar o EJB **BusinessService** em **ApplicationFacade**, ao invés da anotação **@EJB**. Nesse caso, estamos gerenciando a dependência entre os componentes EJB através do uso de CDI. Além disso, também podemos fazer uso de outros recursos disponibilizados por essa especificação, como qualificadores e estereótipos.

#### Listagem 14. Injeção de EJBs através do uso de anotações CDI.

```
01 @Stateless
02 public class ApplicationFacade {
03     @Inject
04     private BusinessService service;
05 }
06
07 @Stateless
08 public class BusinessService {...}
```



# Contextos e Injeção de Dependência (CDI) para Java EE

## JAX-RS

A integração entre CDI e JAX-RS não é diferente da integração entre CDI e EJBs ou entre EJBs e JAX-RS. Na **Listagem 15**, podemos ver um recurso JAX-RS com uma dependência injetada através da anotação `@Inject`.

### Listagem 15. Injeção de dependências com CDI em recursos JAX-RS.

```
01 @Path("/myresource")
02 public class MyResource {
03
04     @Inject
05     private ApplicationService service;
06
07 }
08
09 @RequestScoped
10 public class ApplicationService {...}
```

Nesse caso, será injetado um objeto **ApplicationService**, com escopo de requisição, no recurso **MyResource**.

Em resumo, podemos utilizar as APIs específicas para os serviços que estamos desenvolvendo, como serviços RESTful JAX-RS, componentes EJB ou ainda POJOs, e gerenciar a dependência entre os mesmos através de anotações CDI.

## Eventos

Outro recurso disponível para aplicações que utilizam CDI são os eventos, uma opção que permite que objetos gerenciados interajam sem acoplamento direto ou dependência em tempo de compilação, uma implementação bem interessante do padrão de projeto Observer.

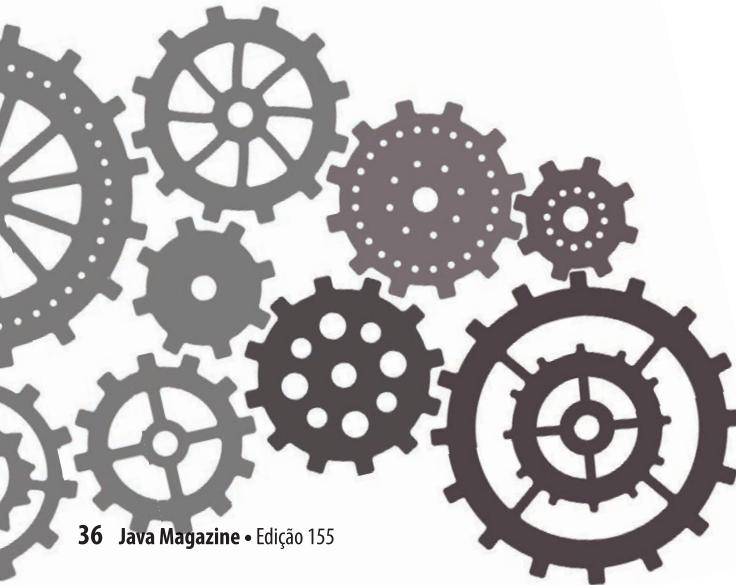
Para utilizar eventos em uma aplicação CDI, precisamos de um elemento que defina o evento (**Listagem 16**) e também injetar um objeto do tipo `javax.enterprise.event.Event` na classe que irá disparar o evento, **Corretora**. Nessa classe, apresentada na **Listagem 17**, injetamos o objeto gerenciado `events` (linha 8), tipado com o conteúdo do evento que queremos disparar, como uma notificação de compra de uma ação. Na linha 16, o evento é construído e disparado, através do método `fire()`.

### Listagem 16. Exemplo de evento.

```
01 public class CompraAcao {
02
03     private final String codigo;
04     private final int quantidade;
05     private final BigDecimal preco;
06
07     public CompraAcao(String codigo, int quantidade, BigDecimal preco) {
08         this.codigo = codigo;
09         this.quantidade = quantidade;
10         this.preco = preco;
11     }
12
13     public String getCodigo() {
14         return codigo;
15     }
16
17     public int getQuantidade() {
18         return quantidade;
19     }
20
21     public BigDecimal getPreco() {
22         return preco;
23     }
24}
```

### Listagem 17. Classe responsável por disparar os eventos que interessam outros elementos da aplicação.

```
01 import java.math.BigDecimal;
02 import javax.ejb.Stateless;
03 import javax.enterprise.event.Event;
04 import javax.inject.Inject;
05
06 @Stateless
07 public class Corretora {
08     private Event<CompraAcao> events;
09
10     @Inject
11     public void setEvents(Event<CompraAcao> events) {
12         this.events = events;
13     }
14
15     public void comprar() {
16         events.fire(new CompraAcao("XPTO", 1000, BigDecimal.valueOf(2L)));
17     }
18 }
```



Para observar o evento disparado pela classe **Corretora**, definimos classes com métodos anotados com `@Observes` (vide **Listagem 18**). Esses métodos são executados quando são disparados eventos do tipo observado.

## Interceptadores

Existem aspectos de uma aplicação que não são diretamente relacionados a seus requisitos de negócio. Entre alguns desses aspectos podemos citar segurança, log de auditoria e monitoramento, os quais são candidatos a componentes que podem ser construídos especificamente para cada um desses assuntos (coesão, lembra?). Isso nos permite manter o código não específico da aplicação separado das regras de negócios.

**Listagem 18.** Classe que “observa” eventos disparados por outros componentes da aplicação.

```
01 @Stateless
02 class MesaOperacoesListener {
03
04     public void processar(@Observes CompraAcao operacao){}
05 }
06
07 @Stateless
08 class AuditoriaListener {
09
10    public void processar(@Observes CompraAcao operacao){}
11 }
```

Para esse fim foram criados Java EE interceptors, originalmente como parte da especificação Enterprise JavaBeans, mas que podem ser utilizados em objetos gerenciados por CDI, sendo um elemento muito interessante para aplicações integradas a essa API.

No exemplo da **Listagem 19** demonstramos como adicionar um aspecto comum a um processo de negócio. Nessa listagem, definimos a classe **Corretora** (linha 8), que não possui instruções para log de auditoria, a classe **Auditor** (linha 23), que é responsável pelo registro de log, e a anotação **@Auditavel** (linha 19), que realiza a ligação entre **Corretora** e **Auditor**.

A classe **Corretora**, que no nosso cenário representa uma classe de negócio sem regras de auditoria, contém a anotação **@Auditavel** (linha 7), a qual marca a classe para ser interceptada por qualquer interceptador que possua a mesma anotação. E para fazer a ligação entre a classe de negócio e o interceptador, incluímos **@InterceptorBinding** na anotação **@Auditavel** (linha 16), sinalizando que **@Auditavel** fará a ligação (binding) entre o elemento onde a mesma aparece, seja um método ou classe, e os interceptadores da aplicação. A classe **Auditor**, por sua vez, é anotada com **@Interceptor** (linha 22), que define o tipo de componente interceptador.

Assim, sempre que um método de uma classe anotada com **@Auditavel** for executado, o método **auditar()** do interceptador também será. A anotação **@AroundInvoke** (linha 25) faz com que o método **auditar()** seja executado antes do início do método auditado. Além disso, o parâmetro **invocationContext** (linha 26) nos permite ter acesso ao método interceptado (linhas 28 e 29). Após realizar as ações específicas do **Auditor**, a chamada a **invocationContext.proceed()** dá sequência ao método que foi interceptado.

Uma crítica a mecanismos como interceptadores é que o seu uso pode gerar comportamentos “invisíveis” em uma aplicação. Se olhar apenas para o código de uma funcionalidade achando que ele contém todo o fluxo que é executado em um determinado processo, você pode ignorar comportamentos que estão localizados em um interceptador, também executado como parte do fluxo, mas de forma desacoplada.

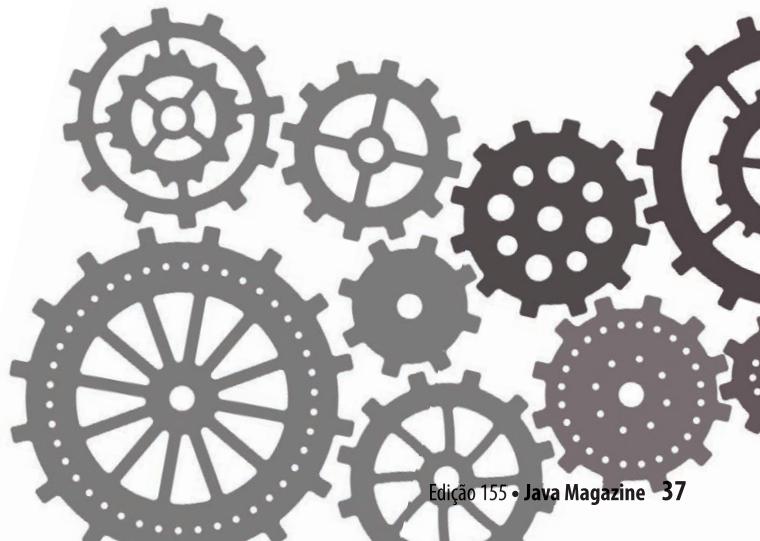
A principal falha no uso de interceptadores, entretanto, é misturar nos mesmos componentes a lógica de negócio e aspectos comuns da aplicação. Por exemplo, se no log de auditoria começarmos a incluir lógica como: “faça um log X, mas se a ação for realizada por um usuário sem perfil administrador, também crie

uma solicitação de aprovação para continuar o processo”, logo estaremos com uma aplicação sem coesão e com lógica distribuída por elementos onde não deveria estar presente.

É possível utilizar interceptadores tanto para lógica de negócio quanto para aspectos ditos de infraestrutura de código, mas essa separação deve ser planejada e realizada pelo próprio desenvolvedor. Como qualquer tecnologia, CDI não previne o mau uso.

**Listagem 19.** Exemplo de uso do interceptador **AroundInvoke** para log de auditoria de um método.

```
01 import javax.interceptor.AroundInvoke;
02 import javax.interceptor.Interceptor;
03 import javax.interceptor.InterceptorBinding;
04 import javax.interceptor.InvocationContext;
05
06 @Stateless
07 @Auditavel
08 public class Corretora {
09 ...
10
11    public void comprar() {
12        events.fire(new CompraAcao("XPTO", 1000, BigDecimal.valueOf(2L)));
13    }
14}
15
16 @InterceptorBinding
17 @Retention(RetentionPolicy.RUNTIME)
18 @Target({ ElementType.METHOD, ElementType.TYPE })
19 @interface Auditavel {}
20
21 @Auditavel
22 @Interceptor
23 class Auditor {
24
25    @AroundInvoke
26    public Object auditar(InvocationContext invocationContext) throws Exception {
27
28        Method method = invocationContext.getMethod();
29        System.out.println("Evento :" + method.getName() + " na classe " + method.getDeclaringClass().getName());
30        return invocationContext.proceed();
31
32    }
33}
```



## Alternativas e considerações

A implementação de referência do CDI na plataforma Java EE é o Weld, sob o guarda-chuva da JBoss. Os conceitos mostrados aqui estão disponíveis na plataforma Java EE desde a versão 6, e vêm sendo refinados a cada nova versão. Essas ideias estão disponíveis há muito tempo em frameworks como Spring e Guice, com suas diferenças, mas objetivos similares: simplificar a vida do desenvolvedor com recursos que facilitam a busca por alta coesão e baixo acoplamento.

Neste artigo, exploramos os recursos de injeção de dependência que podem ser utilizados para simplificar a estrutura de aplicações Java e reduzir o acoplamento entre os componentes. As formas como a especificação CDI nos ajudam a atingir esses objetivos são:

- Disponibilizando um mecanismo simplificado de injeção de dependências, o CDI nos permite remover códigos utilizados para realizar a conexão entre componentes que fazem parte de uma aplicação;

- Ao retirar da aplicação o código necessário para a montagem de componentes, podemos construir os mesmos com foco na responsabilidade que possuem, tornando-os mais coesos;
- Com o uso de interceptadores, qualificadores e escopos, podemos não só configurar uma aplicação de forma flexível, mas também separar aspectos não específicos da aplicação em componentes cuja manutenção pode ser separada da lógica de negócios;
- A integração com outras tecnologias da plataforma Java EE e a padronização de conceitos já disponíveis em outros frameworks consolida o uso das mesmas dentro desse ecossistema e nos permite reutilizar o conhecimento existente, ou adquiri-lo com uma curva de aprendizado menos íngreme.

Enfim, existem diversas estratégias e alternativas para reduzirmos o acoplamento entre componentes de uma aplicação, e o uso de CDI facilita esse objetivo. É possível separar as responsabilidades

de classes de forma coesa e utilizar os mecanismos disponíveis para conectar as mesmas de forma organizada. Deste modo, não há desculpa para a criação de classes gigantes, monolíticas e emaranhadas, uma vez que temos recursos para separar e compor nossas estruturas de código de forma racional.

As tecnologias disponíveis na plataforma Java EE buscam simplificar o modelo de desenvolvimento, nos permitindo reduzir o código necessário para compor as aplicações. E quanto menos poluição temos em uma aplicação, mais fácil fica para que sua estrutura e objetivos transpareçam para o desenvolvedor.

## Autor



Daniel Monteiro

danielmonteiro.java@gmail.com

Oracle Certified Master, Java EE Enterprise Architect. Bacharel em Ciências da Computação, atua no desenvolvimento de software desde 2002.



## Links e Referências:

### Contexts and Dependency Injection for Java EE.

<http://docs.oracle.com/javaee/7/tutorial/partcdi.htm#GJBNR>

### Inversion of Control Containers and the Dependency Injection pattern.

<http://www.martinfowler.com/articles/injection.html>

### Weld - CDI reference Implementation.

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

### Contexts and Dependency Injection for Java EE specification.

<http://jcp.org/en/jsr/detail?id=346>

Yourdon, Edward; Constantine, Larry L. (1975). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press. ISBN 0138544719.

# Apache Camel: Um guia completo – Parte 2

## Conheça na prática o framework através de um exemplo baseado em um cenário real

ESTE ARTIGO FAZ PARTE DE UM CURSO

**N**a parte 1 desta série de artigos foram levantados os principais problemas e desafios que são enfrentados nas integrações de sistemas. Esses problemas estão diretamente relacionados com sistemas legados, arquiteturas mal definidas e decisões de projeto erradas. Além do mais, alguns desafios, como instabilidade e lentidão de redes, sistemas heterogêneos e mudanças de requisitos, também estão incluídos nesse contexto e devem ser considerados.

Porém, esses problemas e desafios foram superados pelos desenvolvedores ao longo do tempo, através de estilos de integração conhecidos como: transferência de arquivo (*Electronic Data Interchange* ou EDI), banco de dados compartilhado, invocação remota de procedimento (*Remote Procedure Call* ou RPC) e mensageria, sendo a mensageira o que possui mais vantagens e benefícios.

Baseando-se na experiência dos desenvolvedores adquirida com a adoção dos estilos, foram reconhecidos e definidos padrões empresariais de integração com o uso de mensageria para indicar soluções elegantes para problemas comumente identificados nesse contexto. Esses padrões foram apresentados em detalhes no artigo anterior.

Exibidos os detalhes e funcionamento dos padrões, foram indicadas as principais tecnologias para integração que utilizam esses padrões como base, e o destaque ficou para o Apache Camel. Essa tecnologia busca simplificar a integração entre sistemas permitindo que com poucas linhas de código e com pouca configuração seja possível processar arquivos, invocar web services, publicar mensagens em um canal de mensageria, entre diversos outros recursos que precisariam de muito código, bibliotecas externas e controle interno.

### Fique por dentro

Este artigo apresenta através de um exemplo prático o Apache Camel, um framework para integração entre sistemas baseado em padrões. A partir da implementação desse exemplo será possível constatar as facilidades que o framework oferece para encapsular código complexo por meio de uma linguagem simples e intuitiva, e também, com a utilização de componentes pré-existentes, verificar o ganho em agilidade e eficiência no desenvolvimento. Portanto, aqueles que desejam aprender na prática como o Apache Camel funciona encontrarão neste artigo demonstrações fundamentais.

Apresentados todos esses conceitos, padrões e principalmente o Apache Camel como um framework em destaque atualmente para integração de sistemas, é importante a elaboração de um exemplo prático para consolidação do conhecimento adquirido. Sendo assim, nesta série de artigos será abordada a implementação de uma solução completa para pagamento bancário.

### Solução para pagamento bancário

Visando consolidar o que foi explicado na primeira parte desta série, será desenvolvida uma solução para integrar um aplicativo mobile aos sistemas necessários para que uma transação de pagamento bancário com cartão de crédito disparada por esse aplicativo seja realizada com sucesso. Para construir essa solução, os requisitos funcionais, descritos na **Tabela 1**, e os requisitos não funcionais, descritos na **Tabela 2**, foram criados com a finalidade de enriquecer o exemplo a ser implementado.

Baseado nos requisitos apresentados, é possível definir o fluxo de negócio que a solução de pagamento bancário deverá atender, incluindo os sistemas externos com necessidade de integração. E foi exatamente isso que foi feito na **Figura 1**. Ela exibe o diagrama de sequência com todas as etapas necessárias para o funcionamento do nosso exemplo.

Com o auxílio dessa imagem e das **Tabelas 1 e 2**, observa-se que o fluxo iniciará com o aplicativo mobile enviando uma requisição

# Apache Camel: Um guia completo – Parte 2

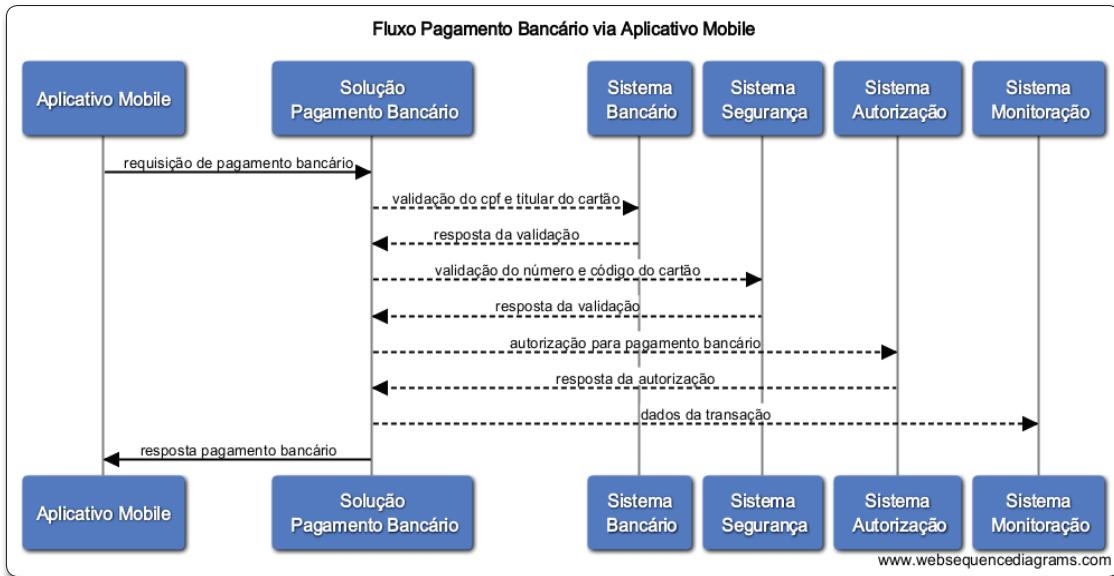


Figura 1. Diagrama de sequência para o fluxo de pagamento bancário

Código - requisito funcional	Descrição
RF-01-APPMOB	A solução de pagamento bancário deve expor um web service para receber do aplicativo mobile requisições com os seguintes dados: id da transação, CPF e nome do titular do cartão, número e código de segurança do cartão e valor da compra.
RF-02-SISBAN	A solução de pagamento bancário, após receber os dados da requisição de pagamento do aplicativo mobile, deve enviar o número do cartão e o CPF do titular para o sistema bancário para validação.
RF-03-SISBAN	A solução de pagamento deve aceitar do sistema bancário a resposta contendo os mesmos atributos de envio, além de uma flag indicando se os dados são válidos.
RF-04-SISSEG	A solução de pagamento, após receber a resposta do sistema bancário, deve avaliar a flag indicativa dos dados válidos. Se positiva, enviar para o sistema de segurança o número e o código de segurança do cartão para validação.
RF-05-SISSEG	A solução de pagamento deve aceitar do sistema de segurança a resposta contendo os mesmos atributos de envio, além de uma flag indicando se os dados são válidos.
RF-06-SISAUT	A solução de pagamento, após receber a resposta do sistema de segurança, deve avaliar a flag indicativa dos dados válidos. Se positiva, enviar para o sistema de autorização o número do cartão e o valor da compra para aprovação do pagamento.
RF-07-SISAUT	A solução de pagamento deve aceitar a resposta do sistema de autorização com os atributos: número do cartão, valor da compra, status e código de autorização do pagamento, mensagem complementar do status e data de autorização.
RF-08-SISMON	Após a conclusão de uma transação de pagamento, ou seja, após a resposta do sistema de autorização, a solução de pagamento bancário deve enviar uma mensagem com os dados da transação, baseada no layout do requisito RF-11-SISMON, para o sistema de monitoração.
RF-09-APPMOB	A solução de pagamento deve enviar a resposta da autorização para o aplicativo mobile com os atributos: id da transação, status e código de autorização do pagamento e data de autorização.
RF-10-SISMON	A solução de pagamento deve enviar uma mensagem para o sistema de monitoração baseada no layout do requisito RF-12-SISMON caso ocorra qualquer erro inesperado durante um fluxo de pagamento bancário.
RF-11-SISMON	O layout da mensagem de conclusão da transação de pagamento deve seguir o padrão: id da transação + data inicial da transação + data final da transação + CPF do titular do cartão + número do cartão + status do pagamento. Os campos referentes a datas devem ser enviados no formato yyyyMMddHHmmss.
RF-12-SISMON	O layout da mensagem de erro inesperado da transação de pagamento deve seguir o padrão: data inicial da transação + mensagem de erro. O campo referente à data deve ser enviado no formato yyyyMMddHHmmss.

Tabela 1. Requisitos funcionais

Código - requisito não funcional	Descrição
RNF-01-APPMOB	A solução de pagamento bancário deve expor web services RESTful.
RNF-02-SISBAN	A comunicação com o sistema bancário deve ser realizada via mensageria com o formato JSON.
RNF-03-SISSEG	A comunicação com o sistema de segurança deve ser realizada via mensageria com o formato JSON.
RNF-04-SISAUT	A comunicação com o sistema de autorização deve ser realizada via mensageria com o formato JSON.
RNF-05-SISMON	A comunicação com o sistema de monitoração deve ser realizada via mensageria com o formato texto baseado em layout pré-definido.
RNF-06-APPMOB	A solução de pagamento bancário deve responder ao aplicativo mobile em até 20 segundos.

Tabela 2. Requisitos não funcionais

para a solução de pagamento bancário com o id da transação, CPF e nome do titular do cartão, número e código de segurança do cartão e valor da compra.

Com a posse desses dados, a solução de pagamento bancário enviará o número do cartão e o CPF do proprietário para o sistema bancário para verificar se ambos são válidos. Se os dados estiverem corretos, o número do cartão e o código de segurança serão enviados ao sistema de segurança para verificação.

Na sequência, se a resposta do sistema de segurança for positiva, a solução de pagamento bancário enviará para o sistema de autorização o número do cartão e o valor da compra para que, de fato, o pagamento seja efetuado. Por fim, a solução de pagamento bancário enviará ao sistema de monitoração os dados da transação e também a resposta final ao aplicativo mobile.

Com todas as informações mencionadas, ou seja, requisitos e fluxo de negócio conhecidos, já é possível definir a arquitetura da solução, de forma que a mesma possa atender e corresponder ao cenário esperado.

## Arquitetura da solução

Além de procurar atender a todos os requisitos indicados, a escolha da arquitetura precisa considerar as necessidades de baixo acoplamento, processamento assíncrono, performance, flexibilidade e tolerância a falhas. Para isso, foi definida uma arquitetura baseada em componentes com comunicação interna utilizando mensageria com o formato JSON. Dessa forma, além de cobrir os requisitos e necessidades de negócio, haverá grande poder de escalabilidade e também baixo custo de manutenção. Por fim, para compor o restante da solução, os frameworks e ferramentas listados a seguir foram escolhidos (para realizar o download de cada um, verifique a seção **Links**):

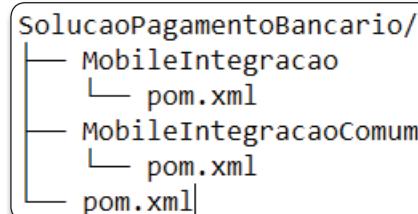
- **Apache Camel (versão 2.16.1):** framework para integração entre os componentes e sistemas;
- **Apache ActiveMQ (versão 5.6.0):** servidor de mensageria para suportar a solução;
- **Apache Maven (versão 3.3):** ferramenta para gerenciamento de dependências e construção automática de build;
- **Spring Framework (versão 4.2.4.RELEASE):** framework para injeção de dependências;
- **WildFly (versão 9.0.2-final):** servidor de aplicação Java EE para execução da solução.

Definida a arquitetura, a partir de agora iniciaremos a implementação da solução de forma gradual, de acordo com os requisitos e fluxo de negócio.

## Iniciando a construção da solução

Conforme relatado na seção anterior, onde foi definida uma arquitetura baseada em componentes, será preciso criar uma estrutura de projeto para atender às necessidades arquiteturais mencionadas. Como foi escolhido o Maven para auxiliar no gerenciamento de dependências e build, os componentes serão organizados em projetos na estrutura multimódulos.

Para ajudar no entendimento de como é essa organização multimódulos, a **Figura 2** exibe a estrutura a ser utilizada no caso da solução de pagamento bancário, mostrando que o diretório *SolucaoPagamentoBancario* é onde estão localizados o projeto agregador e os diretórios dos módulos *MobileIntegracao* e *MobileIntegracaoComum*, que são exemplos de componentes da própria solução. Note que esses módulos devem ficar dentro do diretório do projeto agregador e também conter seu próprio *pom.xml*.



**Figura 2.** Exemplo da estrutura inicial de multimódulos do Maven para a solução a ser construída

Explicada a estrutura da solução, podemos iniciar a criação do projeto agregador e de seu respectivo *pom.xml*. Para isso, crie um diretório com o nome *SolucaoPagamentoBancario* e, dentro do mesmo, o arquivo *pom.xml* conforme as **Listagens 1, 2 e 3**.

Na **Listagem 1**, observa-se nas linhas de 4 a 10 que são definidas as características básicas para um projeto gerenciado pelo Maven, tais como: **name**, **groupId**, **artifactId**, **packaging**, **version**, **modelVersion** e **description**. Como diferencial, vale ressaltar que a tag **packaging** possui o valor **pom** para indicar que esse projeto agregador servirá como base para os demais módulos, isto é, as características básicas e dependências declaradas serão herdadas diretamente.

Já nas linhas 13 a 15 são definidas propriedades cujos valores são as versões das bibliotecas externas que serão utilizadas e que estão declaradas dentro da parte de gerenciamento de dependências.



# Apache Camel: Um guia completo – Parte 2

**Listagem 1.** Arquivo pom.xml do projeto agregador — definições relativas às características da solução, versões de dependências e módulos.

```
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/maven-v4_0_0.xsd">
02
03   <modelVersion>4.0.0</modelVersion>
04   <name>SolucaoPagamentoBancario</name>
05   <groupId>br.com.devmedia</groupId>
06   <artifactId>solucao-pagamento-bancario</artifactId>
07   <version>1.0.</version>
08   <packaging>pom</packaging>
09   <description>Solução de Pagamento Bancário utilizando Apache Camel
10   </description>
11
12   <properties>
13     <camel.version>2.16.1</camel.version>
14     <spring.version>4.2.4.RELEASE</spring.version>
15     <activemq.version>5.6.0</activemq.version>
16     <maven.compiler.plugin.version>3.3</maven.compiler.plugin.version>
17     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
18     <project.reporting.outputEncoding>UTF-8</project.reporting
19       .outputEncoding>
20   </properties>
21
22   <modules>
23     <module>MobileIntegracao</module>
24     <module>MobileIntegracaoComum</module>
25   </modules>
```

**Listagem 2.** Arquivo pom.xml do projeto agregador — declaração das dependências de bibliotecas externas.

```
01 <dependencyManagement>
02   <dependencies>
03     <dependency>
04       <groupId>org.apache.camel</groupId>
05       <artifactId>camel-core</artifactId>
06       <version>${camel.version}</version>
07     </dependency>
08     <dependency>
09       <groupId>org.apache.camel</groupId>
10      <artifactId>camel-spring</artifactId>
11      <version>${camel.version}</version>
12    </dependency>
13    <dependency>
14      <groupId>org.apache.camel</groupId>
15      <artifactId>camel-jackson</artifactId>
16      <version>${camel.version}</version>
17    </dependency>
18    <dependency>
19      <groupId>org.apache.camel</groupId>
20      <artifactId>camel-jms</artifactId>
21      <version>${camel.version}</version>
22    </dependency>
23    <dependency>
24      <groupId>org.apache.activemq</groupId>
25      <artifactId>activemq-camel</artifactId>
26      <version>5.13.0.</version>
27    </dependency>
28    <dependency>
29      <groupId>org.apache.activemq</groupId>
30      <artifactId>activemq-core</artifactId>
31      <version>${activemq.version}</version>
32    </dependency>
33    <dependency>
34      <groupId>org.apache.activemq</groupId>
35      <artifactId>activemq-pool</artifactId>
36      <version>5.13.0.</version>
37    </dependency>
38  </dependencies>
39 </dependencyManagement>
```

Essa prática é recomendada porque, caso haja necessidade de alteração nas versões utilizadas, não é preciso replicar essa mudança em todos os lugares onde as mesmas estão indicadas. Além do mais, evita que erros ocorram ao declarar dependências com versões erradas ou trocadas.

Ainda nas definições de propriedades da **Listagem 1**, verifica-se na linha 16 uma propriedade a ser utilizada para indicar qual ver-

**Listagem 3.** Arquivo pom.xml do projeto agregador — informações para build e geração de pacotes.

```
01   <build>
02     <plugins>
03       <plugin>
04         <groupId>org.apache.maven.plugins</groupId>
05         <artifactId>maven-compiler-plugin</artifactId>
06         <version>${maven.compiler.plugin.version}</version>
07         <configuration>
08           <source>1.6</source>
09           <target>1.6</target>
10         </configuration>
11       </plugin>
12     </plugins>
13     <finalName>${project.artifactId}</finalName>
14   </build>
15 </project>
```



são do plugin **maven-compiler-plugin** o Maven deve utilizar para auxiliar na compilação de toda a solução. Continuando, nas linhas 17 e 18 são definidas propriedades específicas para o Maven com o objetivo de indicar o tipo correto de encode a ser utilizado na manipulação de arquivos para evitar erros e alertas com acentuação de palavras — no caso da solução de pagamento bancário será utilizado o UTF-8. Por fim, a **Listagem 1** possui nas linhas 21 a 24 a tag **modules**, que é usada para definir os módulos que serão agregados à solução de pagamento bancário; nesse caso, estão incluídos inicialmente o **MobileIntegracao** e o **MobileIntegracaoComum** a fim de atender à primeira parte do fluxo de negócios. Os detalhes da implementação desses componentes serão descritos na próxima seção.

Proseguindo com a criação do arquivo POM do projeto agregador, será necessário incluir o código da **Listagem 2** para declarar as dependências de bibliotecas externas que serão utilizadas pela solução. Basicamente, existem as dependências do Camel e do ActiveMQ, com a definição da versão através das propriedades criadas na **Listagem 1**.

Já na **Listagem 3** são declaradas as informações de build e geração de pacotes com a utilização do plugin **maven-compiler-plugin**. A função desse plugin é bem simples e consiste basicamente em auxiliar na compilação dos projetos, como se pode notar nas linhas 8 e 9, onde é especificada a escolha pela versão 1.6 do Java.

Por fim, a linha 13 revela que o artefato gerado (WAR ou JAR) deve ter o mesmo nome que foi declarado na tag **artifactId** do *pom.xml*, para simplificar o nome do artefato. Sem essa configuração os mesmos seriam gerados com o número da versão declarado na tag **version**.

## Implementando os requisitos RF-01-APPMOB e RNF-01-APPMOB

Com a estrutura básica da solução pronta, será apresentada a implementação dos primeiros requisitos, identificados pelos códigos RF-01-APPMOB e RNF-01-APPMOB. Logo, será criado o módulo **MobileIntegracaoComum**, cuja finalidade é representar os dados que serão enviados pelo aplicativo mobile na requisição de pagamento através de web services, além de servir como um intermediário na comunicação com os demais componentes internos da solução, isto é, ele será o responsável por trafegar os dados entre os componentes. Sua implementação é muito simples e consiste basicamente de um POJO com os atributos que serão enviados.

### Criando o módulo MobileIntegracaoComum

Desse modo, crie um novo módulo/projeto do tipo Java Standalone (com o nome **MobileIntegracaoComum**) dentro do diretório do projeto agregador para que a estrutura fique conforme a **Figura 2**. Após isso, crie dentro desse novo projeto um arquivo *pom.xml* com o código da **Listagem 4**.

Observa-se nessa listagem que é utilizada a tag **parent** entre linhas 6 e 10 para determinar que esse módulo possui ligação

com o projeto agregador e que, portanto, as informações referentes a **groupId** e **version** serão herdadas e obtidas automaticamente. 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.

**Listagem 4.** Arquivo *pom.xml* do módulo **MobileIntegracaoComum**.

```
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>MobileIntegracaoComum</name>
13   <artifactId>mobile-integracao-comum</artifactId>
14   <packaging>jar</packaging>
15
16 </project>
```

Seguindo com a implementação da biblioteca, devemos criar um pacote com o nome **br.com.devmedia.mobile.integracao.comum** para conter a classe **PagamentoRequisicao**, que representará o POJO. O código dessa classe é apresentado na **Listagem 5**.



# Apache Camel: Um guia completo – Parte 2

**Listagem 5.** Código da classe PagamentoRequisicao.

```
01 package br.com.devmedia.mobile.integracao.comum;
02
03 import java.io.Serializable;
04 import java.math.BigDecimal;
05
06 public class PagamentoRequisicao implements Serializable {
07
08     private static final long serialVersionUID = 2672206963968858712L;
09
10    private String idTransacao;
11    private String cpfTitularCartao;
12    private String nomeTitularCartao;
13    private String numeroCartao;
14    private String codigoSegurancaCartao;
15    private BigDecimal valorCompra;
16
17    public PagamentoRequisicao() {
18    }
19
20    // Getters e setters dos atributos omitidos.
21
22    @Override
23    public String toString() {
24        StringBuilder builder = new StringBuilder();
25        builder.append("PagamentoRequisicao [");
26        if (idTransacao != null) {
27            builder.append("idTransacao=");
28            builder.append(idTransacao);
29            builder.append(",");
30        }
31        if (cpfTitularCartao != null) {
32            builder.append("cpfTitularCartao=");
33            builder.append(cpfTitularCartao);
34            builder.append(",");
35        }
36        if (nomeTitularCartao != null) {
37            builder.append("nomeTitularCartao=");
38            builder.append(nomeTitularCartao);
39            builder.append(",");
40        }
41        if (numeroCartao != null) {
42            builder.append("numeroCartao=");
43            builder.append(numeroCartao);
44            builder.append(",");
45        }
46        if (codigoSegurancaCartao != null) {
47            builder.append("codigoSegurancaCartao=");
48            builder.append(codigoSegurancaCartao);
49            builder.append(",");
50        }
51        if (valorCompra != null) {
52            builder.append("valorCompra=");
53            builder.append(valorCompra);
54            builder.append(",");
55        }
56        builder.append("]");
57        return builder.toString();
58    }
59}
```

Analisando esse código, verifica-se nas linhas 10 a 15 a declaração dos atributos **idTransacao**, **cpfTitularCartao**, **nomeTitularCartao**, **numeroCartao**, **codigoSegurancaCartao** e **valorCompra**, que representarão os dados enviados pela requisição de pagamento. Já nas linhas 22 a 58 é implementado o método **toString()**, para que os dados do objeto sejam registrados de forma estruturada em logs. Por fim, vale ressaltar que os métodos getters e setters dos atributos foram omitidos por questão de espaço, mas devem ser implementados.

## Criando o módulo MobileIntegracao

Finalizada a implementação da primeira versão do componente **MobileIntegracaoComum**, é a vez do **MobileIntegracao**, responsável por realizar a integração com o aplicativo mobile e manter todas as regras e necessidades voltadas a essa integração.

Para isso, crie um novo módulo/projeto do tipo Java Web dentro do diretório do projeto agregador com o nome **MobileIntegracao**, assim como realizado anteriormente para o **MobileIntegracaoComum**.

Caso seja necessário algum auxílio com a estrutura dos projetos, verifique novamente a **Figura 2**. Feito isso, no novo projeto, crie um arquivo **pom.xml** com o código das **Listagens 6, 7 e 8**.

**Listagem 6.** pom.xml do módulo MobileIntegracao — definições do projeto agregador, nomes, propriedades e repositórios.

```
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
06    <modelVersion>4.0.0</modelVersion>
07
08    <parent>
09      <groupId>br.com.devmedia</groupId>
10      <artifactId>solucao-pagamento-bancario</artifactId>
11      <version>1.0</version>
12    </parent>
13
14    <name>MobileIntegracao</name>
15    <artifactId>mobile-integracao</artifactId>
16    <packaging>war</packaging>
17
18    <properties>
19      <restlet.version>2.3.4</restlet.version>
20      <jackson.version>2.7.0</jackson.version>
21    </properties>
22
23    <repositories>
24      <repository>
25        <id>maven-restlet</id>
26        <name>Public online Restlet repository</name>
27        <url>http://maven.restlet.org</url>
28      </repository>
29    </repositories>
```

**Listagem 7.** pom.xml do módulo MobileIntegracao — declaração de dependências relacionadas ao Camel e outras bibliotecas.

```
01 <dependencies>
02   <dependency>
03     <groupId>${project.parent.groupId}</groupId>
04     <artifactId>mobile-integracao-comum</artifactId>
05     <version>${project.parent.version}</version>
06   </dependency>
07   <dependency>
08     <groupId>org.apache.camel</groupId>
09     <artifactId>camel-core</artifactId>
10   </dependency>
11   <dependency>
12     <groupId>org.apache.camel</groupId>
13     <artifactId>camel-spring</artifactId>
14   </dependency>
15   <dependency>
16     <groupId>org.apache.camel</groupId>
17     <artifactId>camel-jms</artifactId>
18   </dependency>
19   <dependency>
20     <groupId>org.apache.camel</groupId>
21     <artifactId>camel-restlet</artifactId>
22     <version>${camel.version}</version>
23   </dependency>
24   <dependency>
25     <groupId>org.apache.camel</groupId>
26     <artifactId>camel-jackson</artifactId>
27   </dependency>
```

Na **Listagem 6**, observa-se a declaração, nas linhas 6 a 10, das informações do projeto agregador, nas linhas 12 a 14, os dados específicos do módulo, de 16 a 19, as propriedades para indicar a versão das bibliotecas externas que serão utilizadas e, por fim, de 21 a 27, a inclusão da tag **repositories** para indicar o repositório externo onde a biblioteca Restlet será encontrada. A mesma será utilizada para implementação do web service RESTful.

Continuando com a criação do *pom.xml*, a **Listagem 7** apresenta parte das bibliotecas externas que serão necessárias, e o restante está detalhado na **Listagem 8**. Na **Listagem 7** são declaradas as dependências para inclusão do **MobileIntegracaoComum** e também para utilização do Camel. Um detalhe nessa listagem é a inclusão do **camel-restlet**, que se trata de um componente do próprio Camel para integração e funcionamento com a biblioteca Restlet, conforme se observa nas linhas 19 a 23. Por fim, na **Listagem 8**, encontram-se as dependências para bibliotecas do Spring, Restlet e ActiveMQ.

Uma observação válida sobre essas listagens é que para as bibliotecas que foram declaradas no *pom.xml* do projeto agregador não é necessário incluir a versão, pois a mesma será recuperada automaticamente pelo Maven.

Na sequência é essencial alterar o arquivo *web.xml*, localizado no diretório *WEB-INF*, para que a aplicação carregue (ao iniciar) as configurações do Spring, Camel, ActiveMQ e Restlet (web service). Portanto, o arquivo deve ficar conforme a **Listagem 9**.

Analizando esse arquivo, verifica-se na linha 7 a declaração da tag **display-name** para informar o nome da aplicação web, nesse

**Listagem 8.** pom.xml do módulo MobileIntegracao — declaração de dependências relacionadas ao Spring e outras bibliotecas.

```
01   <dependency>
02     <groupId>org.springframework</groupId>
03     <artifactId>spring-expression</artifactId>
04     <version>${spring.version}</version>
05   </dependency>
06   <dependency>
07     <groupId>org.springframework</groupId>
08     <artifactId>spring-beans</artifactId>
09     <version>${spring.version}</version>
10   </dependency>
11   <dependency>
12     <groupId>org.springframework</groupId>
13     <artifactId>spring-context</artifactId>
14     <version>${spring.version}</version>
15   </dependency>
16   <dependency>
17     <groupId>org.springframework</groupId>
18     <artifactId>spring-web</artifactId>
19     <version>${spring.version}</version>
20   </dependency>
21   <dependency>
22     <groupId>org.restlet.jee</groupId>
23     <artifactId>org.restlet.ext.spring</artifactId>
24     <version>${restlet.version}</version>
25   </dependency>
26   <exclusions>
27     <exclusion>
28       <artifactId>spring-asrn</artifactId>
29     </exclusion>
30   </exclusions>
31 </dependency>
32 <dependency>
33   <groupId>org.restlet.jee</groupId>
34   <artifactId>org.restlet.ext.servlet</artifactId>
35   <version>${restlet.version}</version>
36 </dependency>
37 <dependency>
38   <groupId>org.apache.activemq</groupId>
39   <artifactId>activemq-camel</artifactId>
40 </dependency>
41 <dependency>
42   <groupId>org.apache.activemq</groupId>
43   <artifactId>activemq-core</artifactId>
44 </dependency>
45 <dependency>
46   <groupId>org.apache.activemq</groupId>
47   <artifactId>activemq-pool</artifactId>
48 </dependency>
49 </dependencies>
50 </project>
```



# Apache Camel: Um guia completo – Parte 2

Listagem 9. web.xml do módulo MobileIntegracao.

```
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/xml/ns/javaee/web-app_2_5.xsd"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
07   id="WebApp_ID" version="2.5" metadata-complete="true">
08
09   <display-name>Mobile Integração</display-name>
10
11   <listener>
12     <listener-class>org.springframework.web.context.ContextLoaderListener
13     </listener-class>
14   </listener>
15
16   <listener>
17     <listener-class>org.springframework.web.context.request
18       .RequestContextListener</listener-class>
19   </listener>
20
21   <context-param>
22     <param-name>contextConfigLocation</param-name>
23     <param-value>classpath:application-context.xml</param-value>
24   </context-param>
25
26   <servlet>
27     <servlet-name>RestletServlet</servlet-name>
28     <servlet-class>org.restlet.ext.spring.SpringServerServlet</servlet-class>
29     <init-param>
30       <param-name>org.restlet.component</param-name>
31       <param-value>RestletComponent</param-value>
32     </init-param>
33   </servlet>
34
35 </web-app>
```



caso, “Mobile Integração”. Já as linhas 9 a 11 indicam a utilização do listener **ContextLoaderListener** do Spring para criar um contexto geral para o componente e incluí-lo no **ServletContext**. Com isso, será possível que o Spring gerencie os beans que serão utilizados no componente.

As linhas 17 a 20 indicam que o arquivo *application-context.xml* deve ser carregado e interpretado ao iniciar a aplicação para que as configurações presentes no mesmo sejam incluídas no contexto geral da aplicação. É nesse arquivo que serão declarados os beans a serem utilizados.

Concluindo com o arquivo *web.xml*, nas linhas 22 a 33 é definido o servlet **RestletServlet** da biblioteca Restlet para compor o web service. Seu funcionamento ocorrerá baseado em um servlet Java EE, onde, dada uma requisição para uma determinada URL, o mesmo será invocado. Em especial, observa-se nas linhas 30 a 33 que a URL mapeada para o servlet é */webservice/\**, isto é, qualquer requisição que iniciar com o path */webservice* será processada por ele.

Na sequência, devemos incluir, ainda no diretório *WEB-INF*, o arquivo *jboss-web.xml* conforme a **Listagem 10**.

Listagem 10. jboss.xml do módulo MobileIntegracao.

```
01 <jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
04   http://www.jboss.org/j2ee/schema/jboss-web_5_1.xsd">
05
06   <context-root></context-root>
07
08 </jboss-web>
```

A criação desse arquivo com a configuração indicada é apenas para declarar ao servidor de aplicação — no caso, o WildFly — que a aplicação web será acessada pelo contexto do próprio endereço do servidor de aplicação. Por exemplo, ao invés da aplicação ser acessada através do endereço *http://localhost:8080/mobile-integracao* será apenas *http://localhost:8080/*. Essa configuração foi incluída para simplificar o acesso ao web service, para não expor detalhes da solução e diminuir a dependência que o sistema externo teria ao vincular o nome da aplicação com o web service.

Ainda ajustando as configurações de **MobileIntegracao**, é preciso criar o arquivo *application-context.xml* no diretório *src/main/resources* de acordo com a **Listagem 11**.

As configurações implementadas nesse arquivo são muito importantes, pois são através das declarações dos beans (gerenciados pelo Spring) que a conexão com o ActiveMQ é efetuada, o carregamento inicial das rotas do Camel é feito, além da utilização da biblioteca Restlet. Analisando mais a fundo, as linhas 12 a 14 informam para o Camel qual a classe que contém as rotas a serem iniciadas quando a aplicação carregar. Nesse caso, trata-se da **MobileIntegracaoRouteBuilder**, a ser implementada neste artigo, referenciada pela tag **camel:routeBuilder** na linha 13.

Já a linha 16 declara um bean chamado **RestletComponent**, que é uma implementação do próprio Restlet a ser utilizado para o web service. Na sequência, as linhas 18 a 20 indicam a declaração de outro bean, denominado **RestletComponentService**, relacionado

**Listagem 11.** application-context.xml do módulo MobileIntegracao.

```
01 <beans xmlns="http://www.springframework.org/schema/beans"
02   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="mobileIntegracao">
14     <camel:routeBuilder ref="mobileIntegracaoRouteBuilder"/>
15   </camel:camelContext>
16 
17   <bean id="RestletComponent" class="org.restlet.Component"/>
18 
19   <bean id="RestletComponentService" class="org.apache.camel.component
20     .restlet.RestletComponent">
21     <constructor-arg ref="RestletComponent"/>
22   </bean>
23 
24   <bean id="jmsConnectionFactory" class="org.apache.activemq
25     .ActiveMQConnectionFactory">
26     <property name="brokerURL"
27       value="failover:(tcp://localhost:61616,tcp://127.0.0.1:61616)?
28         randomize=false"/>
29   </bean>
30 
31   <bean id="pooledConnectionFactory" class="org.apache.activemq.pool
32     .PooledConnectionFactory"
33     init-method="start" destroy-method="stop">
34     <property name="maxConnections" value="8"/>
35     <property name="connectionFactory" ref="jmsConnectionFactory"/>
36   </bean>
37 
38   <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
39     <property name="connectionFactory" ref="pooledConnectionFactory"/>
40     <property name="concurrentConsumers" value="10"/>
41   </bean>
42 
43   <context:component-scan base-package="br.com.devmedia.mobile.integracao"/>
44 
45 </beans>
```

ao componente **camel-restlet** do Camel, que, por sua vez, necessita do bean anterior para ser construído.

Prosseguindo com a análise da listagem, as linhas 22 a 25 criam um bean chamado **jmsConnectionFactory** para fazer a conexão com o ActiveMQ. Vale ressaltar que na linha 24 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 27 a 32 mais um bean, com o nome de **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 29.

Já nas linhas 34 a 37, outro bean é criado, com o nome **jmsConfig**, para customizar e indicar um número de consumidores que estarão funcionando em paralelo para cada conexão do ActiveMQ definida anteriormente. Essa alteração visa uma melhor performance para o processamento das mensagens, pois o padrão é 1 consumidor apenas. Essa configuração é realizada na linha 36 através da propriedade **concurrentConsumers**.

Finalizando a criação do arquivo, as linhas 39 a 41 definem um bean chamado **activemq**, cuja implementação é o componente do Camel **camel-activemq**. Esse bean inclui em suas configurações o bean mencionado anteriormente, **jmsConfig**, conforme observa-se na linha 40, e com esse último bean é que será possível enviar



mensagens para as filas nas rotas do Camel. Por último, na linha 43 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.mobile.integracao`. Com isso, as classes anotadas com `@Component` 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.

Concluída a implementação do `application-context.xml`, é hora de criarmos a classe que será a detentora das rotas do Camel, ou seja, a classe que vai manipular o fluxo. Essa é a principal classe do componente e as rotas definidas nela são inicializadas no momento em que a aplicação é carregada. Para isso, crie o pacote `br.com.devmedia.mobile.integracao.rota` e, depois, a classe `MobileIntegracaoRouteBuilder`, conforme a **Listagem 12**.

Analizando esse código, observa-se na linha 10 a declaração da anotação `@Component` para indicar que essa classe será gerenciada pelo Spring. Logo após, a linha 11 cria uma classe chamada `MobileIntegracaoRouteBuilder`. Atente para a parte final do nome, o trecho `RouteBuilder`, uma boa prática para que classes específicas, detentoras de rotas, sejam padronizadas e localizadas facilmente. Ainda na linha 11 é possível verificar que `MobileIntegracaoRouteBuilder` estende a classe abstrata do Camel `RouteBuilder`, o que a faz tornar-se uma classe específica para definição e construção de rotas. Com isso, é necessário sobrescrever o método `void configure()` para, então, definir em seu interior as rotas.

Já as linhas 16 a 23 declararam uma rota com o auxílio da DSL do Restlet para compor o web service RESTful responsável por receber as requisições de pagamento. Nesse código, a linha 16 permite a criação de um web service do tipo REST cujo path é `/pagamento`. Assim, a URL final para exposição do web service será formada da seguinte forma: `http://localhost:8080/webservice/pagamento`.



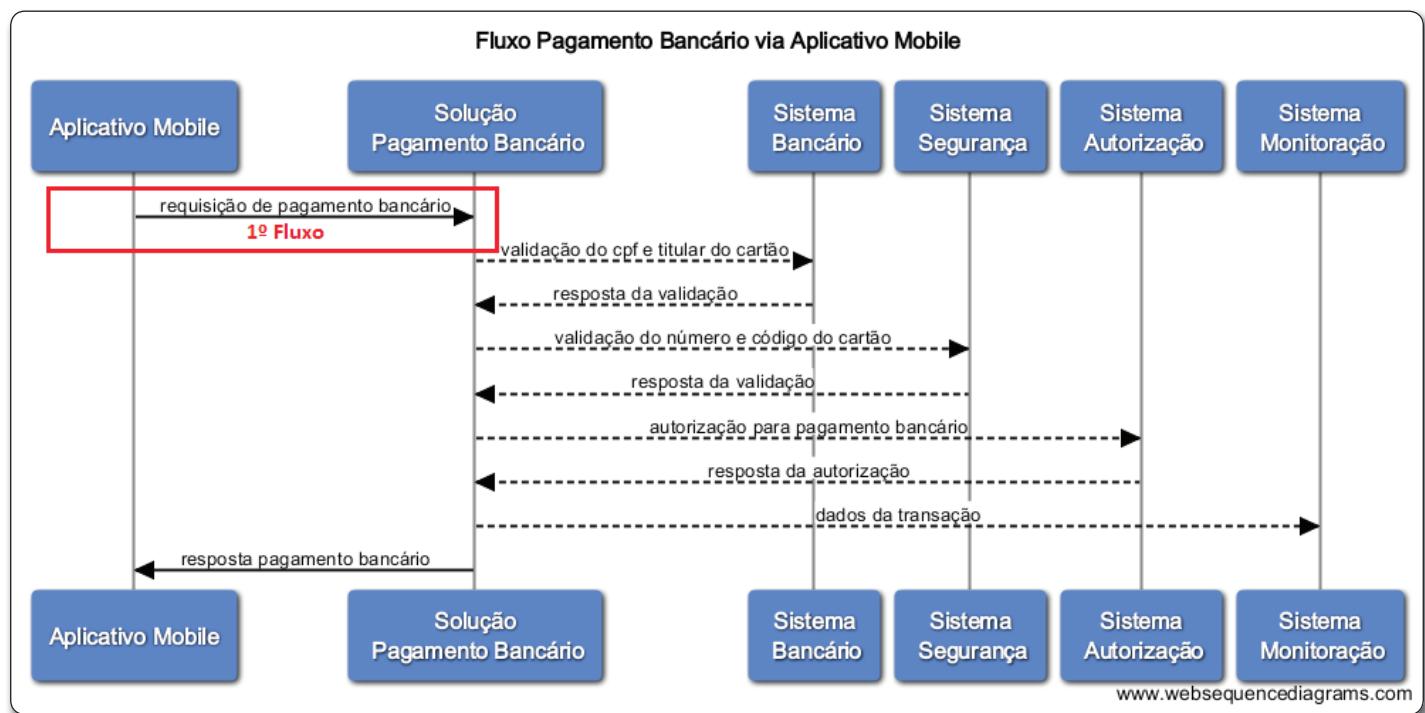
mento, onde `localhost:8080` é o endereço padrão do servidor de aplicação, `/webservice` origina-se do servlet denominado `RestletServlet` mapeado no arquivo `web.xml` da aplicação e `/pagamento` é a declaração do path na classe em questão. Por fim, a inclusão do id “`rotaEntradaWS`” ao final da linha é uma boa prática para identificar e nomear rotas no contexto geral do Camel.

Continuando com a análise da **Listagem 12**, a linha 17 é apenas uma descrição da finalidade do web service. As linhas 18 e 19, por sua vez, indicam que o mesmo consumirá requisições com o conteúdo em JSON e produzirá respostas também com conteúdo em JSON. Já a linha 20 declara que o método desse web service será do tipo POST. Na sequência, as linhas 21 e 22 são incluídas para que, ao receber a requisição, o Camel faça a conversão automática do body da requisição para um objeto (POJO) `PagamentoRequisicao` da biblioteca `MobileIntegracaoComum`. Nessa conversão, o objeto já é incluído pelo próprio Camel na mensagem (Exchange) que está sendo trafegada na rota.

Na sequência, a linha 23 faz o envio da mensagem que está sendo manipulada ao longo da rota do web service para uma outra rota, chamada de `postPagamento`, que nesse caso é interna

**Listagem 12.** Código da classe `MobileIntegracaoRouteBuilder`, do módulo `MobileIntegracao`.

```
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.stereotype.Component;
07
08 import br.com.devmedia.mobile.integracao.comum.PagamentoRequisicao;
09
10 @Component
11 public class MobileIntegracaoRouteBuilder extends RouteBuilder {
12
13     @Override
14     public void configure() throws Exception {
15
16         rest("/pagamento").id("rotaEntradaWS")
17             .description("Serviço para efetuar pagamento bancário")
18             .consumes("application/json")
19             .produces("application/json")
20             .post()
21             .bindingMode(RestBindingMode.json)
22             .type(PagamentoRequisicao.class)
23             .to("seda:postPagamento");
24
25         from("seda:postPagamento")
26             .log(LoggingLevel.INFO, "[MobileIntegracao] Nova requisição de
27             pagamento bancário")
28             .end();
29 }
```



**Figura 3.** Primeiro fluxo de negócio atendido.

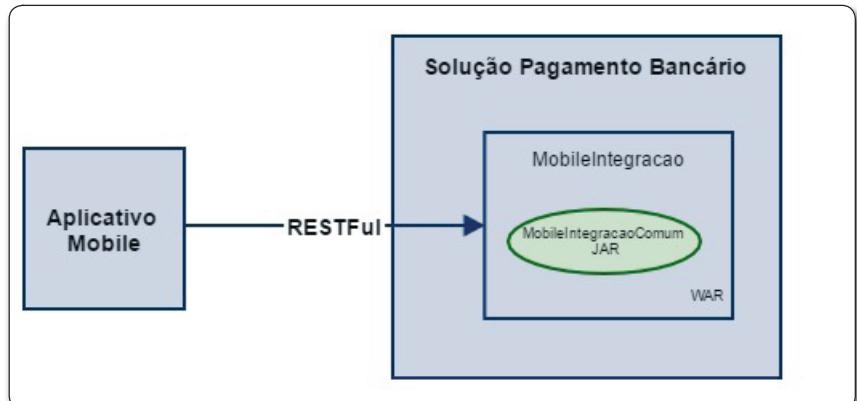
e específica do componente. Essa nova rota utiliza um componente do próprio Camel conhecido como SEDA, cuja finalidade é prover, através de uma fila, um comportamento assíncrono para uma arquitetura SEDA (vide **BOX 1**), de modo que as mensagens sejam trocadas em uma abordagem BlockingQueue (vide **BOX 2**) e os consumidores sejam chamados em uma thread separada do produtor.

Por último, as linhas 25 a 27 declaram uma outra rota para consumir as mensagens oriundas da primeira, que é a do web service. Para essa nova rota, chegará uma mensagem do tipo Exchange com o POJO devidamente incluído. Porém, na implementação desse primeiro requisito, apenas é logado, em modo INFO, que uma requisição de pagamento bancário foi enviada.

Com todas essas implementações, o primeiro fluxo de negócio foi atendido, conforme se observa na **Figura 3**, e a solução de pagamento bancário se encontra com uma arquitetura semelhante à **Figura 4**.

Nesta segunda parte do artigo, iniciamos a construção e codificação de uma solução baseada em um cenário real 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. Nessa primeira etapa prática, foi possível demonstrar, através da implementação do primeiro fluxo de negócio, todas as facilidades, vantagens e os recursos básicos do Apache Camel.

Portanto, pode-se destacar a facilidade de entendimento e compreensão do código através do uso de DSL para a declaração de



**Figura 4.** Diagrama atual da arquitetura da solução de pagamento bancário

#### BOX 1. SEDA

SEDA é a sigla para Staged Event-Driven Architecture e tem como proposta decompor uma complexa aplicação dirigida a eventos em componentes menores ligados por filas. Com essa arquitetura é possível evitar a alta sobrecarga de processamento que é associada à utilização de threads para tratar concorrência, além de servir como um modelo para desacoplar eventos e threads da camada lógica da aplicação.

#### BOX 2. BlockingQueue

**BlockingQueue** ou **Fila Bloqueante** é uma estrutura de dados que utiliza uma fila onde os elementos são removidos de forma ordenada, seguindo um modelo conhecido como FIFO (First in, first out), em que o primeiro elemento que chega é o primeiro a sair. Além do mais, nesse modelo, qualquer tentativa em adicionar elementos em um fila cheia ou retirar de uma vazia será recusada, e sua principal vantagem é garantir que uma thread obtenha uma única mensagem por vez. No contexto do Java existem várias implementações de **BlockingQueues**, pois se trata apenas de uma interface. As mais conhecidas são: **ArrayBlockingQueue**, **DelayQueue**, **LinkedBlockingQueue**, **PriorityBlockingQueue** e **SynchronousQueue**.

rotas e processadores. Além disso, a utilização de componentes pré-existentes, a fim de realizar a conexão e comunicação com outros sistemas e componentes, também é destaque. Por fim, a inclusão do framework Spring para gerenciar os recursos da aplicação e realizar as injetões de dependências também demonstrou a simplicidade e agilidade no desenvolvimento que o mesmo pode acrescentar a soluções que adotam o Apache Camel.

## Autor



Rodrigo Cunha Santana

[rodcunhasantana@gmail.com](mailto: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:

### Endereço para download do Apache Camel.

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

### Endereço para download do Apache ActiveMQ.

<http://activemq.apache.org/activemq-560-release.html>

### Endereço para download do Apache Maven.

<https://maven.apache.org/download.cgi>

### Endereço para download do Spring Framework.

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

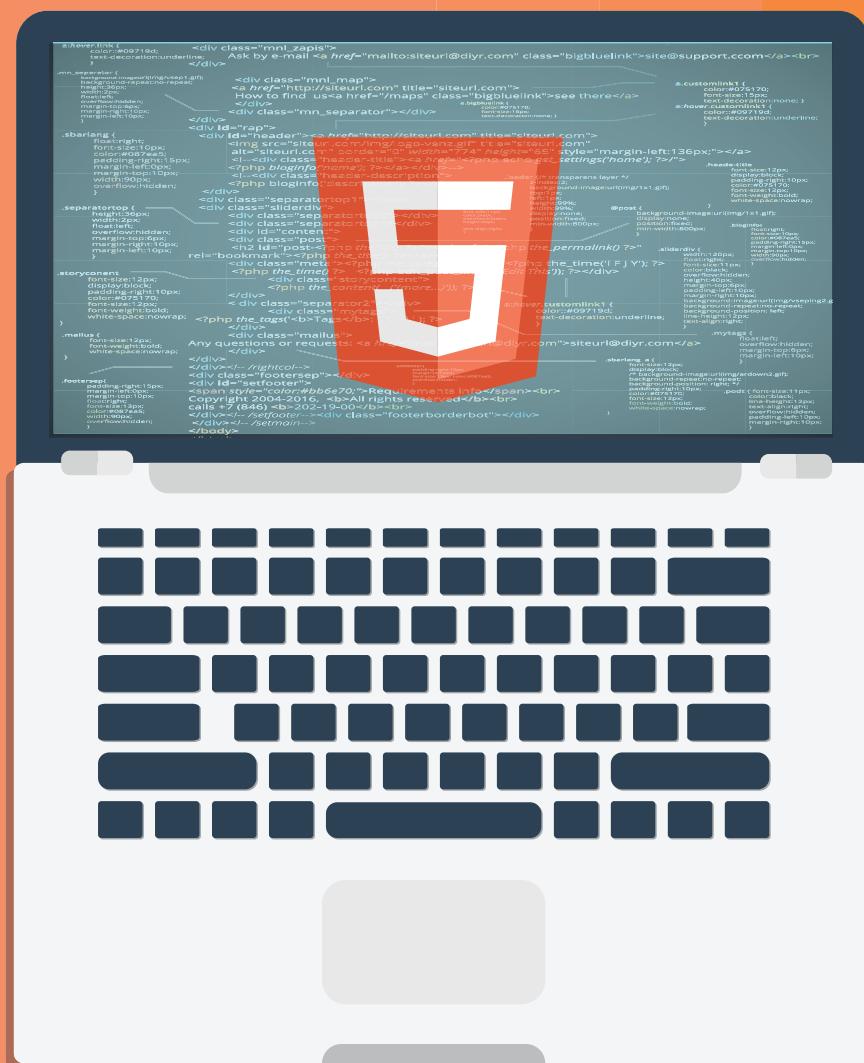
### Endereço para download do WildFly.

<http://wildfly.org/downloads>



# 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](http://porta80.com.br) | [comercial@porta80.com.br](mailto:comercial@porta80.com.br) | [twitter.com/porta80](http://twitter.com/porta80)

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