



Cloud computing com WebLogic Multitenant
Aplicações Java EE com controle e isolamento na JVM

**Testes funcionais e
automatizados com Cucumber**
**Como simplificar os
testes de web services com Java**

Apache Camel: Um guia completo
**Concluindo a solução de integração
para um sistema de pagamento**

 DEVMEDIA

Edição 157 :: R\$ 14,90

WEB SERVICES SEGUROS COM JWT

**Proteja sua API RESTful
com JSON Web Token**



MVP

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

APlique esse investimento
na sua carreira...

E mostre ao mercado
quanto você vale!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



Edição 157 • 2017 • ISSN 1676-8361



MVP

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

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultora Técnica Anny Caroline (annycarolinegnr@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araújo

Distribuição

FC Comercial e Distribuidora S.A

Rua Teodoro da Silva, 907, Grajaú - RJ

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

Atendimento ao leitor

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

Publicidade

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

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

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

Fale com o Editor!

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

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



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

Sumário

Destaque - Vanguarda

Conteúdo sobre Novidades

04 – WebLogic Multitenant: A nuvem dentro do servidor de aplicação

[Daniel Cicero Amadei]

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

17 – Web services RESTful: Como adicionar segurança com JWT

[Emanuel Mineda Carneiro]

Conteúdo sobre Boas Práticas

28 – Web services: testes funcionais e automatizados com Cucumber

[Gabriel Novais Amorim]

Artigo no estilo Curso

36 – Apache Camel: Um guia completo – Parte 4

[Rodrigo Cunha Santana]

WebLogic Multitenant: A nuvem dentro do servidor de aplicação

Consolidando suas aplicações Java EE com controle e isolamento na mesma JVM

Hoje em dia a maioria das corporações sofre com a proliferação de ambientes, que ocorre por questões de insegurança (ao colocarmos novas aplicações em ambientes já existentes e funcionais) ou por diferença de criticidade. O grande problema dessa proliferação de ambientes é que ela custa caro em termos de mais infraestrutura e pessoal e também aumenta o risco, pois cresce a dificuldade de manter todos esses ambientes atualizados.

A solução para esse cenário é a consolidação das aplicações em um número menor de ambientes, reduzindo a complexidade, demanda sobre o time de operações, riscos, etc. No entanto, tal consolidação nos expõe aos problemas que levam à proliferação, como, por exemplo, a possibilidade de uma aplicação impactar sobre outra. A consolidação com controle sobre o comportamento de cada aplicação e também com isolamento entre elas é o cenário ideal para colocar mais aplicações em um mesmo ambiente e ainda garantir que uma não cause impacto na outra. Isso só pode ser obtido através de uma abordagem Multitenant. Mas, o que é mesmo Multitenant?

Para que seja possível entender o que é uma abordagem Multitenant, por que ela permite a consolidação com controle e, depois, assimilar como isso se aplica ao mundo Java EE, é preciso dar um passo atrás e compreender o significado desse termo. Multitenancy significa “múltiplos inquilinos” e, quando falamos sobre o assunto, estamos tratando de consolidação e compartilhamento de recursos com o objetivo de colocar mais ativos no mesmo ambiente e nos beneficiar dos recursos comuns disponíveis.

No mundo do software, podemos dizer que um software ou plataforma é Multitenant quando uma instância

Fique por dentro

Este artigo será útil para que você conheça uma inovação interessante no mundo dos servidores de aplicação Java EE. Vamos apresentar a funcionalidade Multitenant do Oracle WebLogic Server, recentemente lançada como parte da versão 12.2 do produto. Você aprenderá quais benefícios essa abordagem cloud traz para o desenvolvimento e, principalmente, para a instalação e operação das suas aplicações. Além disso, veremos como uma abordagem Multitenant auxilia você a alavancar outras iniciativas, como DevOps e Microservices, sem ter que abandonar a robustez da plataforma Java Enterprise Edition.

serve a múltiplos consumidores, também conhecidos como tenants. Essa abordagem permite compartilhar o ambiente e tirar proveito de uma série de fatores devido a esse compartilhamento, como veremos ao longo do artigo.

Java EE e o compartilhamento de recursos

Neste momento você pode estar pensando nos projetos Java EE que possui, onde provavelmente já compartilha recursos através da instalação de múltiplas aplicações no mesmo ambiente, não é mesmo? Então, o que de tão interessante tem esse tal de Multitenant que você já não esteja fazendo, visto que na maioria dos servidores de aplicação é possível instalar diversas aplicações na mesma instância e compartilhar os recursos e serviços providos pela plataforma Java EE?

Isso tudo é verdade, mas até certo ponto. O Java EE realmente é uma plataforma que permite o compartilhamento de recursos. Você pode instalar múltiplas aplicações na mesma JVM, fazendo com que todas elas compartilhem memória, poder de processamento, conexões com a base de dados, adaptadores JCA, recursos JMS (filas, tópicos, connection factories), entre outros.

Porém, nem tudo é perfeito nesse cenário. O problema começa quando você precisa compartilhar o ambiente e a infraestrutura, mas ao mesmo tempo necessita de um isolamento forte entre os ‘tenants’. Os tenants podem ser aplicações Java EE, ou conjuntos delas, que demandam isolamento umas das outras. A motivação desse isolamento pode ser, por exemplo, a criticidade ou o fato de ser uma aplicação nova, não madura e que não se tem confiança sobre o impacto que pode causar no ambiente.

Isolamento é importante quando o assunto é compartilhamento

Já que multitenancy significa multi-inquilinato, podemos fazer uma analogia com o mundo real para exemplificar a importância do isolamento em cenários de compartilhamento de recursos. Por exemplo, ao pensar em um edifício de apartamentos, fica mais clara a demanda por compartilhamento com controle. Nesse cenário, podemos assumir que os usuários irão compartilhar algumas das facilidades do edifício, como elevadores, áreas comuns, piscinas, garagens, entre outros. No entanto, vão demandar também políticas de isolamento. Os apartamentos deverão ser isolados fisicamente uns dos outros por questões de segurança e privacidade. Será necessário também um controle de acesso que garanta que usuários de um apartamento não possam acessar outro apartamento sem autorização, entre outros exemplos.

Agora, imagine que para conseguir um isolamento físico mais forte ou um controle de acesso diferente por tenant, tenhamos que replicar os elevadores, a piscina, a garagem ou até mesmo o edifício para cada condômino. Imaginou? Pois é, muitas vezes é isso que acontece com as aplicações que precisam ser isoladas umas das outras, como veremos mais a frente.

A mesma demanda por um balanceamento entre o compartilhamento de recursos, o isolamento e o controle que temos em um condomínio se aplica a qualquer plataforma que se disponha a prover serviços compartilhados. É isso que o WebLogic Server Multitenant agrega ao que já existe na plataforma Java EE e o que veremos neste artigo.

A necessidade por isolamento e compartilhamento de recursos em Java EE

A necessidade do isolamento entre aplicações Java EE não é algo novo. Essa demanda ganha cada vez mais atenção à medida que as corporações tendem a consolidar mais aplicações no mesmo ambiente para reduzir a complexidade, consumo de recursos e simplificar a gestão e as operações.

Algumas abordagens para compartilhamento de recursos com isolamento já existem há bastante tempo. Essas soluções têm seu valor, e algumas podem ser utilizadas para complementar as funcionalidades providas pelo WebLogic Multitenant. Vamos analisar algumas delas para compreender quais problemas são possíveis de serem solucionados com o WebLogic Multitenant e que não eram antes.

Isolamento através da virtualização: múltiplas máquinas virtuais

A virtualização pode ser uma maneira de isolar aplicações e, ainda, tirar proveito do compartilhamento de recursos.

Isso é possível uma vez que as VMs compartilham o mesmo hardware físico, mas há o isolamento entre as máquinas virtuais em termos de consumo de recursos, como CPU, memória, disco, etc.

Máquinas virtuais são uma boa opção para a consolidação e divisão de servidores grandes em máquinas virtuais menores e mais especializadas. Você provavelmente fará a instalação das instâncias do WebLogic Multitenant em máquinas virtuais. O problema é que configurar uma máquina virtual somente para isolar uma única aplicação das demais pode ser um exagero.

As máquinas virtuais levam consigo todo o footprint de uma nova instância do sistema operacional e, por isso, pode ser um exagero utilizá-las como uma forma de isolamento de aplicações. Tecnologias de containers, como o Docker (veja o **BOX 1**), são uma alternativa para esse cenário. Para adicionar mais overhead à nossa equação, em cada máquina virtual você terá uma máquina virtual Java (JVM) e outra instância do servidor de aplicação. Caso você precise instalar e isolar uma nova aplicação, o mesmo overhead será replicado: máquina virtual, sistema operacional, JVM e servidor de aplicação. Além disso, esse cenário agrupa muita complexidade para a gestão dos ambientes, uma vez que, quanto mais aplicações demandam isolamento, maior é a demanda por mais máquinas virtuais, JVMs e instâncias de servidor de aplicação, o que leva a mais ambientes, mais recursos, mais tempo para aplicação de atualizações, etc.

BOX 1. Docker

É um projeto open-source que permite criar containers com tudo que sua aplicação precisa para executar. Através do uso de containers podemos ter um isolamento similar a uma máquina virtual, porém sem o overhead de executar um sistema operacional sobre outro. Isso ocorre porque o container não exige uma nova instância do sistema operacional, como no caso das máquinas virtuais, mas sim compartilha o kernel do sistema operacional onde está executando.

Isolamento com múltiplas JVMs

Outra forma bastante comum de isolar aplicações Java EE é executá-las em instâncias dedicadas do servidor de aplicação e JVM, mas na mesma infraestrutura base, que pode ser virtual ou física, ou seja, executando vários “nós” do servidor de aplicação na mesma máquina.

Dessa forma, as aplicações são completamente isoladas umas das outras sob o ponto de vista da JVM e servidor de aplicação, mas compartilham o mesmo sistema operacional. É mais uma abordagem para isolamento, mas que também vem com um preço a ser pago. Nesse caso, o overhead está relacionado à complexidade. Esse preço é um pouco menor do que o do cenário anterior, mas ainda é grande, pois temos a replicação da JVM e da instância de servidor de aplicação para cada aplicação que demande isolamento. Assim, à medida que adicionamos mais aplicações com essa característica, maior será o consumo de recursos no ambiente.

Isolamento limitando o número de threads e execuções concorrentes

A última opção que precisamos avaliar antes de mergulhar no mundo do Multitenant é a instalação de todas as aplicações na

mesma instância do servidor de aplicação e, consequentemente, na mesma JVM. Essa abordagem é boa em termos de compartilhamento de recursos, uma vez que todas as aplicações compartilham a mesma JVM e servidor de aplicações. Com isso, o footprint é “diluído” entre todas elas. Por outro lado, o risco passa a ser a falta de isolamento, visto que uma aplicação pode monopolizar todos os recursos disponíveis no servidor de aplicação e/ou JVM, levando a uma indisponibilidade das outras aplicações presentes no mesmo ambiente.

Para minimizar (ou tentar minimizar) tais riscos, muitos servidores de aplicação possuem a opção de configurarmos pools de thread dedicados por aplicação. Isso pode ser uma opção para evitar que uma aplicação lenta ou com um pico inesperado no número de acessos monopolize o servidor e deixe as demais indisponíveis.

Com pools de threads dedicados para cada aplicação em um servidor de aplicação Java EE, você pode instalar várias aplicações juntas e ainda garantir que uma que esteja lenta ou acessando um back-end lento, por exemplo uma base de dados com problema, não roube todas as threads do servidor. Essa abordagem pode ser uma opção, mas também tem algumas desvantagens, como veremos a seguir.

O primeiro inconveniente é como efetuar o dimensionamento dos pools de threads. Se você criar um pool muito grande, sobrará threads e você acabará consumindo recursos desnecessariamente. Por outro lado, se você subestimar o tamanho do pool, faltarão threads para processamento das requisições, ocorrerão enfileiramentos e o usuário final é quem sofrerá.

Outro problema dessa abordagem é o fato dos pools de threads serem separados e independentes uns dos outros. Isso não permite que uma aplicação ou requisição seja priorizada em relação às demais, pois não existe conhecimento por um dos pools que outro está processando requisições. Sendo assim, é impossível que uma requisição “passe a sua vez” para outra que seja mais prioritária.

0 caminho até o suporte a multitenancy no WebLogic

Em sua versão nove, lançada em 2006, o servidor de aplicações WebLogic fez uma mudança radical em sua arquitetura. O principal objetivo foi criar a fundação para acomodar múltiplas aplicações e ainda prover um profundo controle sobre como cada uma dessas aplicações se comporta e o quanto do ambiente elas podem utilizar. Foi o primeiro passo para um ambiente Multitenant.

Essa abordagem constituiu basicamente a capacidade do WebLogic Server trabalhar com um único pool de threads para todas as aplicações. Pode parecer estranho unificar o pool de threads quando estamos falando de isolamento e proteção, mas na verdade foi uma decisão arquitetural bastante interessante. Com um pool de threads único, criou-se também um conceito chamado work managers, que veremos em mais detalhes a seguir. Além disso, o pool de threads passou a ser auto-ajustável, mitigando os riscos sobre dimensionamento que mencionamos. Para mais informações sobre o que são e para que servem pools de threads, consulte o **BOX 2**.

BOX 2. Pools de Threads

Um dos serviços providos pelos servidores de aplicação é o que chamamos de pools de threads. Os pools de threads são importantes uma vez que a criação de threads é um processo bastante custoso. Com a abordagem de pools, as threads são criadas, mantidas “vivas” e reutilizadas para servir propósitos recorrentes, como atender às requisições dos clientes.

Work managers e sua importância no mundo Multitenant

Mesmo com um pool de threads único, o WebLogic permite que controlemos a concorrência utilizando um recurso chamado de “Work Managers”. Com essa opção, você tem um controle bastante granular sobre o comportamento de sua aplicação, quantidade de execuções concorrentes, entre outros aspectos. Assim, podemos obter os mesmos benefícios que teríamos em um cenário de thread pools dedicados por aplicação, ao mesmo tempo em que mantemos as vantagens de um único thread pool.

A ideia principal do uso de um único thread pool somado aos work managers é a possibilidade de ter uma visibilidade completa sobre as requisições de todas as aplicações. Com um thread pool único podemos priorizar aplicações ou requisições que são mais críticas do que as demais. Isso é extremamente importante para um cenário Multitenant e não seria possível se houvessem diferentes pools de threads com cada um atendendo sua aplicação. Além disso, os work managers permitem controlar o número de requisições concorrentes que cada aplicação pode atender, o quanto se deseja enfileirar, além de vários outros aspectos relacionados à concorrência das requisições.

O que mais é necessário para um ambiente Java EE Multitenant “de verdade”?

Como já mencionado, isolar requisições e controlar a concorrência é um excelente primeiro passo. Porém, para efetivamente chegar a um ambiente Java EE que possa ser compartilhado por múltiplas aplicações, é preciso atender a outros requisitos igualmente importantes quando o assunto é multitenancy. Vamos discutir um pouco sobre eles a seguir:

- **Isolamento da segurança:** um ambiente Multitenant deve prover isolamento dos mecanismos de segurança. Isso permite que as políticas aplicadas a uma aplicação não interfiram nas demais. Além disso, deve haver a possibilidade de isolar outros recursos relacionados à segurança, como usuários, papéis e grupos, entre os tenants que façam uso do ambiente;

- **Isolamento dos contextos e namespaces:** para que seja possível obter um alto nível de consolidação, é necessário o isolamento de contextos e namespaces envolvidos nas aplicações. No mundo Java EE isso geralmente está relacionado ao serviço *Java Naming and Directory Interface*, mais conhecido como JNDI. Deve ser possível, por exemplo, que duas aplicações instaladas na mesma instância do servidor de aplicações demandem recursos com o mesmo nome JNDI e que representem diferentes objetos;

- **Isolamento no consumo de recursos:** o controle de concorrência e o isolamento no processamento das requisições são muito importantes. No entanto, eles não eliminam a necessidade de um profundo controle de consumo dos recursos pelas aplicações

que vivem no mesmo ambiente. É crucial evitar que determinada aplicação “roube” todo o poder de processamento ou utilize toda a memória heap disponível na JVM. Isso pode ocorrer devido a um pico inesperado e anormal no número de acessos ou simplesmente porque a aplicação foi mal codificada.

Esses pontos, descritos como essenciais, estão presentes no WebLogic Multitenant.

Como funciona o WebLogic Multitenant?

Até aqui você já entendeu os benefícios e o que é necessário para um ambiente Java EE verdadeiramente Multitenant. Também já vimos o quanto difícil é compartilhar esse ambiente entre múltiplas aplicações sem o devido suporte para isolamento entre elas. Agora é hora de entender o funcionamento do WebLogic Multitenant e o que ele provê entre tudo o que analisamos conceitualmente até aqui.

Estudo sobre partições

Além das funcionalidades providas pelos Work Managers que já vimos, temos as **Partições**, que são o componente-chave para a arquitetura Multitenant do WebLogic. Uma ótima forma de entender o que é uma partição é pensar nela como um microcontainer que reside dentro do servidor de aplicação. Uma partição pode atuar como um delimitador de escopo para aplicações e recursos, e pode também controlar outros aspectos relacionados ao compartilhamento desses recursos, como o namespace JNDI, isolamento da segurança, uso de CPU, memória e file descriptors. A Figura 1 ilustra a relação entre as partições e a JVM do servidor de aplicação.

Quando criamos recursos dentro do servidor de aplicação, como data sources JDBC, filas JMS, etc., você tem duas opções em relação ao escopo deles: global ou de partição. Ao escolher global, o recurso fica disponível e visível para todas as aplicações. Em contrapartida, no escopo de partição, o recurso ficará disponível e visível somente para aplicações que sejam instaladas e executem na mesma partição. Nesse caso, é possível ter recursos com o mesmo nome JNDI em diferentes partições, sem a ocorrência de conflitos, uma vez que estarão com escopo de partição.

A segurança é outro aspecto que pode ter seu escopo associado à partição. Dessa forma, é possível configurar recursos relacionados

à segurança que se apliquem a somente uma partição, não sendo, portanto, compartilhados com todo o ambiente. Como exemplos do que pode ser isolado por partição podemos citar: usuários, grupos, papéis (roles), provedores de autenticação, entre outros aspectos.

Na Figura 2 você pode ver um exemplo onde temos três partições em uma única instância do servidor de aplicações. Uma coisa interessante e que ainda não havíamos mencionado é o fato das partições terem um estado (Status). Assim, você pode iniciar ou parar toda a partição, o que ativa ou desativa todos os seus recursos e aplicações. É como se você estivesse parando ou iniciando o servidor de aplicação, mas na verdade você fará isso apenas para um grupo de aplicações e recursos.

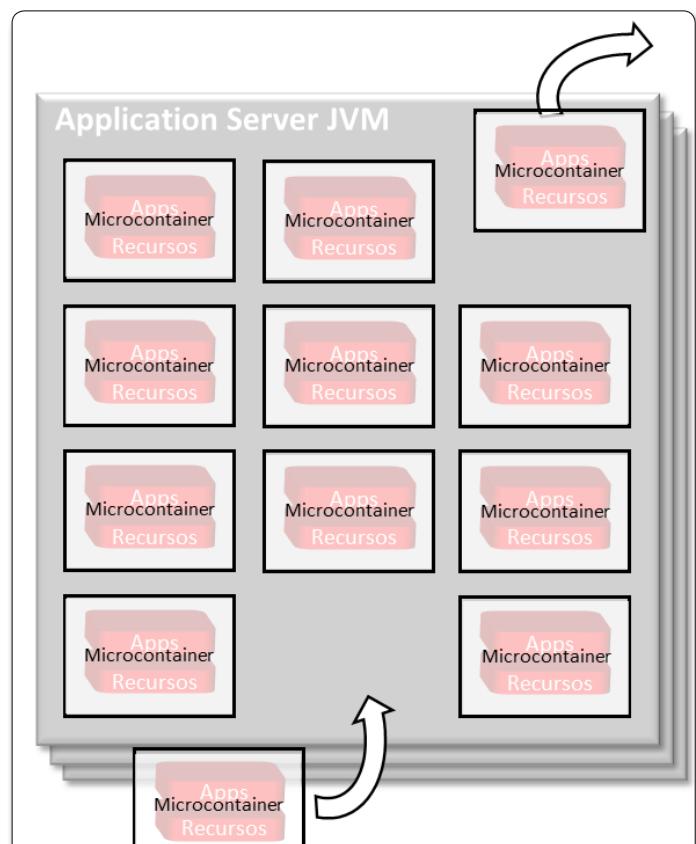


Figura 1. Partições dentro do servidor de aplicação

Partições de Domínio

Partições de domínio são blocos de construção de MT (multitenancy) do WebLogic Server. O recurso Multitenancy permite que várias organizações clientes compartilhem um domínio, aumentando a eficiência e reduzindo os custos operacionais. Antes de criar uma partição de domínio, crie um ou mais alvos virtuais. Consulte os tópicos de Conceitos Básicos para obter mais informações.

► Conceitos Básicos de Multitenancy

| Nome | Status | Estado | Partição do OTD | Realm | Alvos Padrão | Alvos Disponíveis | Grupos de Recursos |
|------------|--------|-------------|-----------------|---------|-----------------|-------------------|--------------------|
| Particao-A | ↑ | Em Execução | | Realm-A | VirtualTarget-A | VirtualTarget-A | particaoA-RG |
| Particao-B | ↑ | Em Execução | | Realm-B | VirtualTarget-B | VirtualTarget-B | particaoB-RG |
| Particao-C | ↑ | Em Execução | | Realm-C | VirtualTarget-C | VirtualTarget-C | particaoC-RG |

Partições de Domínio 3 de 3

Figura 2. Um domínio com três partições

WebLogic Multitenant: A nuvem dentro do servidor de aplicação

Como já mencionado, ao ter múltiplas partições no ambiente, podemos configurar recursos com o mesmo nome JNDI, desde que o seu escopo seja de partição, para que não ocorram conflitos. As aplicações terão acesso somente aos recursos de sua partição ou globais, que, como o nome indica, serão visíveis por todas as partições.

A **Figura 3** demonstra que podemos ter três data sources JDBC, todos eles com o mesmo nome JNDI e sem conflitos, uma vez que cada um tem um escopo de partição diferente.

O acesso aos recursos com escopo de partição ou recursos globais é transparente do ponto de vista de uma aplicação instalada na mesma partição que os recursos que ela quer acessar. Aqui, no entanto, é válido mencionar que é possível acessar recursos com escopo de partição a partir de clientes externos ao servidor de aplicações ou até mesmo a partir de outras partições. Neste caso, utilizamos uma sintaxe especial ao especificar a URL do provedor JNDI para conexão, especificando o nome da partição como a última parte da URL. A **Listagem 1** mostra como isso pode ser feito.

Além disso, também é válido destacar que é possível proteger os recursos para que não sejam acessados sem a devida autorização. Deste modo, esse acesso entre partições ou de um cliente externo para um recurso de uma partição pode ser controlado.

Por fim, saiba que podemos instalar a mesma aplicação em cada uma das partições, e cada uma fará uso dos recursos com escopo daquela partição, como os data sources JDBC, sem interferências de recursos ou aplicações de outras partições.

A **Figura 4** mostra esse cenário. Nela, vemos a aplicação CrmAPIs instalada três vezes, uma vez em cada partição. Essa aplicação faz uso do data source com nome JNDI *jdbc/appDS*, que vimos na **Figura 3**, e cada instância da aplicação verá apenas o nome JNDI disponibilizado em sua partição.

Listagem 1. Efetuando lookup JNDI de um objeto com escopo de partição.

```
Hashtable<String, String> env = new Hashtable<>();
env.put(Context.PROVIDER_URL, "t3://<ip>:<porta>/<partição>");
env.put(Context.SECURITY_PRINCIPAL, <usuário>);
env.put(Context.SECURITY_CREDENTIALS, <senha>);
Context ctx = new InitialContext(env);
Object c = ctx.lookup("jms/MinhaFila");
```

Controlando o consumo de recursos e priorizando as requisições

Outro aspecto importante que podemos obter com o uso de partições é a priorização das requisições ao servidor de aplicação, que pode, inclusive, ser baseada em questões de negócio, por exemplo: podemos priorizar a partição que abriga as aplicações mais críticas. Outra opção para a priorização é que ela pode ser ativada dependendo da quantidade de recursos sendo consumidos por cada uma das partições existentes no ambiente. No caso da priorização por recursos, podemos escolher entre:

- **Uso de CPU:** uso médio de CPU efetuado pela partição nos últimos minutos;
- **Memória heap utilizada:** total de memória heap da JVM utilizada pela partição e ainda não liberada para coleta pelo garbage collector;
- **File descriptors abertos:** número de descritores de arquivos mantidos abertos pela partição (veja o **BOX 3**).

BOX 3. File descriptors

Os file descriptors são ponteiros abstratos para acesso a qualquer mecanismo de entrada/saída (IO), como arquivos, pipes ou sockets, presentes em sistemas operacionais baseados em Unix e Linux.

| Origens de Dados JDBC | | | | | | | |
|---|--------------|----------|---|----------------------------|------------|-----------------|--|
| Esta página lista as origens de dados do sistema JDBC que foram criadas neste domínio. Você pode criar, configurar, testar, controlar ou excluir as origens de dados do sistema desta página. | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| Nome | Nome da JNDI | Tipo | Escopo | Grupo de Recursos / Modelo | Partição | Alvos | |
| appDS | jdbc/appDS | Genérico | Grupos de Recursos da Partição de Domínio | particaoA-RG | Particao-A | VirtualTarget-A | |
| appDS | jdbc/appDS | Genérico | Grupos de Recursos da Partição de Domínio | particaoB-RG | Particao-B | VirtualTarget-B | |
| appDS | jdbc/appDS | Genérico | Grupos de Recursos da Partição de Domínio | particaoC-RG | Particao-C | VirtualTarget-C | |

Figura 3. Data sources JDBC instalados em cada partição.

| Implantações | | | | | | | |
|--------------|-------------|--------|--------|-------------|---------------|---------------------|-----------------|
| | | | | | | | |
| | Nome | Status | Estado | Integridade | Tipo | Partição de Domínio | Alvos |
| ▶ | (@) CrmAPIs | ▲ | Active | OK | Aplicação Web | Particao-A | VirtualTarget-A |
| ▶ | (@) CrmAPIs | ▲ | Active | OK | Aplicação Web | Particao-B | VirtualTarget-B |
| ▶ | (@) CrmAPIs | ▲ | Active | OK | Aplicação Web | Particao-C | VirtualTarget-C |
| ▶ | opss-rest | ▲ | Active | OK | Aplicação Web | | AdminServer |

Figura 4. A mesma aplicação instalada em cada partição

Esse tipo de controle só é possível com uma forte parceria entre o servidor de aplicações e a JVM, e é exatamente assim que ele foi feito. Para que fosse possível levar em conta métricas mantidas pela JVM, a HotSpot JVM foi alterada para incluir tais capacidades e atuar em parceria com o servidor de aplicações, possibilitando um ambiente realmente multitenant.

Controlando o consumo de recursos de cada partição

Como vimos na seção anterior, a JVM foi estendida para manter um controle de qual partição está abrindo file descriptors, executando código (e consequentemente consumindo CPU) e também sobre o quanto de memória está sendo alocado por partição.

Para tirar proveito de tudo isso e fazer uso das opções de controle de consumo de recursos, você precisa utilizar a versão 8u40 ou superior da JVM HotSpot. Além disso, é necessário passar os seguintes parâmetros na linha de comando para habilitar a gestão de recursos na JVM e utilizar o algoritmo G1 (vide **BOX 4**) para garbage collection:

-XX:+UnlockCommercialFeatures -XX:+ResourceManagement -XX:+UseG1GC

BOX 4. Garbage First (G1)

O coletor Garbage First (G1) é um algoritmo de garbage collection para aplicações que executam em ambiente servidor. Esse algoritmo é parcialmente concorrente e trabalha com várias áreas de memória segregadas, realizando a coleta de cada uma dessas áreas de modo independenteumas das outras para evitar pausar completamente a aplicação.

A seguir, veremos como ter um controle total sobre os recursos utilizados em cada partição e, com isso, poder tomar ações a partir de tais situações.

Triggers

Quando você habilita a gestão de recursos (também conhecido como *resource management*), a JVM passa a armazenar as métricas relacionadas ao consumo dos mesmos por partição. A partir desse ponto, você pode configurar as ações que deseja tomar quando certas condições ocorrerem. Isso é útil para que você possa evitar que uma aplicação malcomportada danifique todo o ambiente e permitir proteger as partícões, o que é muito importante quando pensamos em consolidação.

A essa possibilidade de controlar o consumo de recursos e tomar uma ação dependendo do estado do ambiente, damos o nome de “Trigger” ou Gatilho, em português. Como já mencionado, as condições que podem ser usadas nesses gatilhos podem ser baseadas em consumo de CPU, memória heap retida e/ou número de file descriptors abertos. Por exemplo, podemos criar um gatilho para reduzir a prioridade de uma partição ou até mesmo pará-la por completo quando ela exceder limites pré-estabelecidos de consumo de CPU ou quando estiver usando mais memória do que deveria.

A **Figura 5** mostra como podemos configurar um gatilho em relação à memória heap retida pela partição. Esse gatilho vai notificar, via mensagens de alerta nos logs, se a partição estiver retendo mais do que 512MB de memória, vai reduzir a prioridade

da partição caso ela consuma mais do que 768MB de memória e irá parar a partição com seus recursos e aplicações por completo caso ela comece a consumir mais do que 1GB de memória heap.



Figura 5. Configurando um gatilho baseado no total de memória retida pela partição

Os gatilhos ou triggers são uma forma muito importante para proteger o ambiente. Outra forma de fazer isso é configurar como ele deve se comportar em momentos de contenção, por exemplo, priorizando uma partição em relação às demais.

Priorização e o compartilhamento de recursos

O WebLogic Multitenant suporta a definição de como requisições devem ser priorizadas entre as partícões. Assim, você pode instalar aplicações de diferentes graus de criticidade no mesmo ambiente e entregar mais recursos para aquelas consideradas críticas. Você prioriza as partícões que contêm aplicações mais críticas especificando uma “fatia” de recursos que devem ser dedicados a elas na forma de um “peso”. Essa funcionalidade é chamada de *Fair Share*.

Para entender melhor o conceito por trás da priorização, suponha que você possui três partícões em seu ambiente, como vemos na **Figura 6**.

Nesse cenário, sem configurações adicionais, cada partição terá seu próprio escopo para seus recursos, como JNDI, JMS, JDBC, entre outros. As partícões podem, ainda, ter isolamento de segurança.

Porém, não haverá isolamento ou priorização em relação ao consumo de recursos.

Agora, imagine que a partição A tenha aplicações mais críticas e/ou importantes que as demais. Nesse caso, faz sentido que a partição A tenha maior prioridade que as outras. É exatamente para isso que a funcionalidade Fair Share existe.

Através do Fair Share podemos indicar um peso à partição, o qual se tornará um percentual de prioridade da partição. Deve-se ressaltar que essa priorização só é válida em momentos de contenção no ambiente e, caso não exista contenção, as partícões serão tratadas do mesmo modo. A **Figura 7** ilustra um cenário no qual a partição A recebe 70% de prioridade.

WebLogic Multitenant: A nuvem dentro do servidor de aplicação



Figura 6. Seu servidor de aplicação com três partições



Figura 7. Priorização das partições no servidor de aplicação

A fatia de recursos que deve ser alocada para cada partição pode ser definida de acordo com o consumo de recursos, por exemplo: dedicar $x\%$ do uso de CPU para determinada partição.

As Figuras 8 e 9 mostram como podemos configurar o Fair Share baseado no consumo de recursos para as partições A e B (a configuração da partição C é idêntica à da partição B). Com essa configuração no ambiente, a partição A poderá consumir até 70% de CPU e as outras duas, 15% cada.

Com essa configuração, caso alguma dessas partições passe a consumir mais CPU do que o determinado em nossa configuração, ou seja, 70% no caso da partição A e 15% cada uma das outras duas, o gerenciador de recursos entrará em ação e reduzirá a prioridade da partição consumindo mais recursos do que o especificado, como veremos a seguir.

Evidenciando a priorização na prática

Para evidenciar o funcionamento da priorização de partições na prática, criamos uma aplicação composta de algumas APIs REST que foi instalada nas três partições de modo homogêneo. Para garantir que haveria contenção, limitamos o número de threads do WebLogic a cinco e criamos um teste de carga com nove threads.

Ao executar o teste de carga, após aproximadamente dois minutos é possível notar o gestor de recursos em ação. Ao analisar os logs, veremos uma mensagem indicando que a ação *Slow Action Quota* foi atingida e que as partições B e C deveriam ter sua prioridade reduzida, conforme expõe a Listagem 2.

Analizando os gráficos do JMeter, ferramenta utilizada para geração da carga, podemos ver a priorização em ação na Figura 10. Após a condição ser detectada, a partição A (linha vermelha) começa a ser priorizada em relação às demais. Isso ilustra que a configuração do gestor de recursos está funcionando conforme esperado. Essa figura mostra o gráfico sob a perspectiva de transações por segundo. Note que a partição A executa mais transações por segundo que as outras partições menos prioritárias do ambiente, por terem excedido o limite de consumo de CPU.

Podemos observar o mesmo comportamento através do tempo médio de resposta.



Figura 8. Configuração do Fair Share baseado no consumo de CPU para a partição A



Figura 9. Configuração do Fair Share baseado no consumo de CPU para a partição B

Listagem 2. Logs gerados durante condições de carga.

```
<[Slow Action Quota Reached For Partition: Particao-B] [Current Usage: 33]
[severity-value: 32] [Previous Usage: 0] [Was Required action to Slow the Partition
is executed?: true] [Resource Name: com.oracle.weblogic.rcm.framework.base.
ThreadCPUResourceAttributes] [partition-id: 6bd0194f-e5ea-4c0f-bdc9-
1c69 bee69d02] [partition-name: Particao-B] [rid: 0:69] > <BEA-2165800>
<Resource Consumption Management Slow Message: Given quota has been
reached for the partition and a slow action has been executed.OfType: 1>
```

```
<[Slow Action Quota Reached For Partition: Particao-C] [Current Usage: 32]
[severity-value: 32] [Previous Usage: 0] [Was Required action to Slow the
Partition is executed?: true] [Resource Name: com.oracle.weblogic.rcm.framework
.base.ThreadCPUResourceAttributes]
[partition-id: b15cb3f3-954f-4f2d-8a6f-67fe5a282104]
[partition-name: Particao-C] [rid: 0:73] > <BEA-2165800>
<Resource Consumption Management Slow Message: Given quota has
been reached for the partition and a slow action has been executed.OfType: 1>
```

A partição A passa a ter um tempo médio de resposta menor que as demais, visto que ela passou a ter mais prioridade. A Figura 11 ilustra isso com a partição A representada pela linha vermelha. Veja que o tempo médio de resposta dela é bem menor que o das outras partições, pois suas requisições são retiradas da fila de requisições mais rapidamente.

Quando a situação de carga termina e o consumo de CPU das partições B e C passa a ser menor que o valor estabelecido de 15%, a priorização é removida e todas as partições voltam a ter a mesma prioridade. Isso é detectado automaticamente pelo gestor de recursos (confira o log da Listagem 3).

Listagem 3. Logs indicando que a condição de carga cessou e a ação de “despriorizar” as partições B e C está sendo desligada.

```
<[Un-Slow Action For Partition: Particao-B]>
[Current Usage: 0] [severity-value: 32]
[Previous Usage: 0] [Resource Name:
com.oracle.weblogic.rcm.framework.base.
ThreadCPUResourceAttributes] [Was Required
action to Un-Slow the Partition is executed?: true]
[partition-id: 6bd00194f-e5ea-4c0f-bdc9
-1c69bee69d02] [partition-name: Particao-B]
[rid: 0:69] > <BEA-2165803>

<Resource Consumption Management Unslow
Message: Slow action is withdraw as quota was
reached below the given trigger value.>
<[Un-Slow Action For Partition: Particao-C]>
[Current Usage: 0] [severity-value: 32]
[Previous Usage: 0] [Resource Name: com.oracle.
weblogic.rcm.framework.base.
ThreadCPUResourceAttributes] [Was Required
action to Un-Slow the Partition is executed?: true]
[partition-id: b15cb3f3- 954f-4f2d-8a6f-
67fe5a282104] [partition-name: Particao-C]
[rid: 0:73] > <BEA-2165803>
<Resource Consumption Management Unslow
Message: Slow action is withdraw as quota was
reached below the given trigger value.>
```

Criando seu ambiente Multitenant

O primeiro passo para experimentar os recursos do WebLogic Multitenant é instalar um banco de dados que será o repositório de nossa aplicação. Pela simplicidade de instalação e configuração, vamos usar o MySQL. Conectaremos a esse banco de dados via JDBC com o usuário root do MySQL.

Realizada a instalação desse banco de dados, inicie sua instância do MySQL (vide diretório *bin* da instalação do SGBD) e acesse-a via comando *mysql*. O comando para acesso é *mysql -u root -p*. Criaremos uma base de dados para cada partição com o objetivo de armazenar o modelo de dados da nossa aplicação de exemplo. O script para isso está disponível no site da DevMedia.Com o usuário root, execute os seguintes comandos:

```
create database <NOME DA BASE DE DADOS>
use <NOME DA BASE DE DADOS>
source <caminho completo dos scripts>
/modelo-dados-mysql.sql
```

Por exemplo, para a base de dados **db1**, assumindo que os scripts estão em *c:\temp\scripts*, você deve executar:

```
create database db1
use db1
source c:\temp\scripts\modelo-dados-mysql.sql
```

Execute o mesmo processo para criar outras duas bases de dados: **db2** e **db3**.

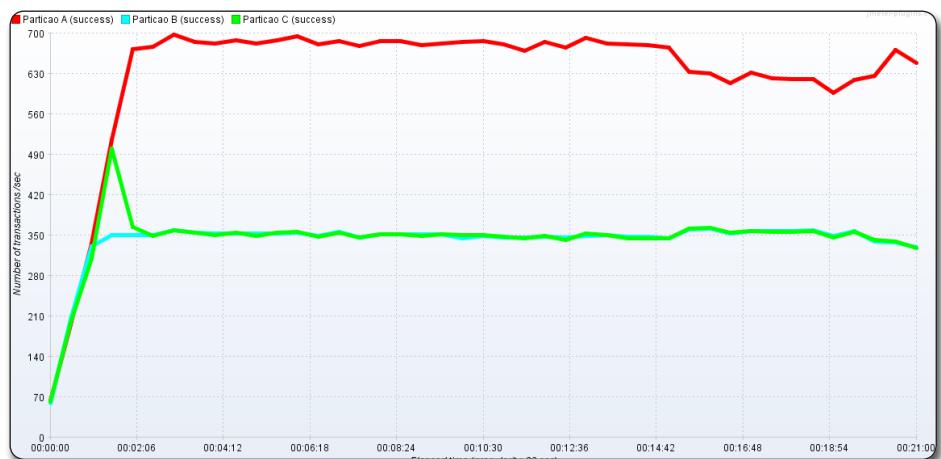


Figura 10. Transações por segundo da mesma aplicação instalada nas três partições

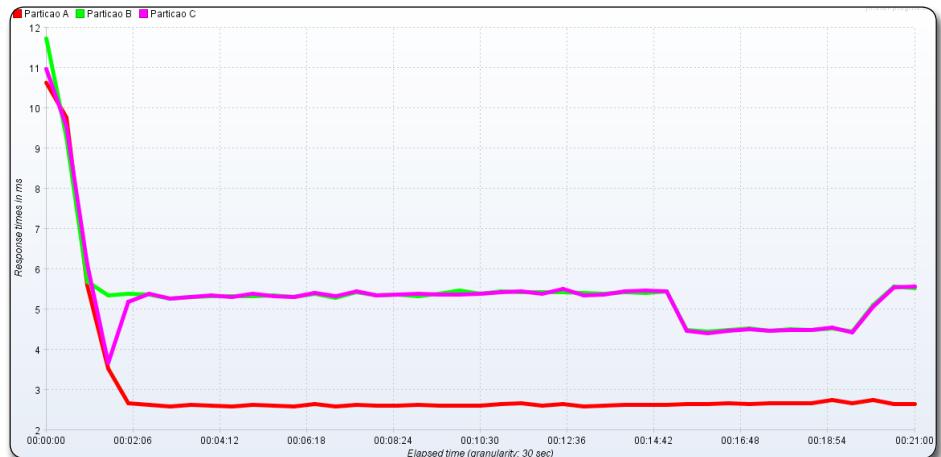


Figura 11. Tempo médio de resposta da mesma aplicação instalada nas três partições

A **Listagem 4** mostra os resultados esperados após a execução do script para criação do modelo de dados. Lembre-se de executar essas operações para as três bases de dados.

Mais à frente vamos conectar cada partição do WebLogic a uma base de dados diferente. Por isso precisamos das três idênticas em termos de modelo de dados, mas os dados armazenados serão diferentes, para exemplificar na prática o isolamento provido pela abordagem multitenant.

Instalando e configurando o WebLogic Multitenant

Para criar seu ambiente Multitenant, o próximo passo é instalar o JDK 8, preferencialmente o último update disponível. No momento em que este artigo foi escrito, o último update disponível era o JDK 8u91.

Com o JDK instalado, você deve efetuar o download do WebLogic Server 12.2.1.1 (ou mais recente) a partir da URL disponível

Listagem 4. Resultados da criação dos objetos no banco de dados (MySQL)

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.03 sec)
Query OK, 0 rows affected (0.02 sec)
Query OK, 0 rows affected (0.03 sec)
Query OK, 0 rows affected (0.03 sec)
Query OK, 0 rows affected (0.03 sec)
Records: 0 Duplicates: 0 Warnings: 0
Query OK, 0 rows affected (0.08 sec)
Records: 0 Duplicates: 0 Warnings: 0
Query OK, 0 rows affected (0.08 sec)
Records: 0 Duplicates: 0 Warnings: 0
Query OK, 0 rows affected (0.09 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

na seção **Links**. Ao acessar esse endereço, você verá que existem algumas opções disponíveis para download, como uma versão “quick start” para os desenvolvedores, versões específicas para sistemas operacionais e uma versão genérica.

WebLogic Multitenant: A nuvem dentro do servidor de aplicação

Para esse exemplo vamos baixar a versão genérica, que é um JAR portável para qualquer sistema operacional suportado.

Além do WebLogic Server, devemos instalar o Fusion Middleware Infrastructure 12.2, que dá suporte a uma série de recursos requeridos para o funcionamento do Multitenant. Vale notar que tanto o instalador do WebLogic quanto o do Fusion Middleware Infrastructure vêm compactados como arquivos .zip. Deste modo, extraia os JARs dos instaladores dos arquivos .zip que você efetuou o download.

Logo após, instale utilizando o executável do Java 8 conforme demonstrado a seguir (caso esteja no Windows, é necessário acessar o prompt de comando como administrador):

```
java -jar fmw_12.2.1.1.0_wls.jar
```

Durante o processo será solicitado o *Oracle Home*. O *Oracle Home* é o local onde os binários do WebLogic serão instalados. Tome nota desse diretório ou altere-o para um de sua preferência, pois precisaremos acessá-lo em breve. Para o restante da instalação, basta aceitar as configurações padrão até o fim.

Encerrada essa etapa, é hora de instalar o Fusion Middleware Infrastructure seguindo um processo muito similar ao que usamos para o WebLogic Server:

```
java -jar fmw_12.2.1.1.0_infrastructure.jar
```

A instalação do Fusion Middleware Infrastructure deve ser feita na mesma *Oracle Home* que você usou para o WebLogic. Isso adicionará novas funcionalidades e componentes necessários para o funcionamento do WebLogic Multitenant.

Criando o domínio

Após a execução dos dois instaladores, vamos criar um Domínio. Na documentação do WebLogic, um domínio é definido como “a unidade básica de administração para instâncias do WebLogic Server. Um domínio consiste de uma ou mais instâncias do WebLogic Server (e seus recursos associados) que você gerencia com um único Admin Server. Você pode definir múltiplos domínios baseado em diferentes responsabilidades dos administradores, fronteiras das aplicações ou localização geográfica dos servidores. Por outro lado, você pode utilizar um único domínio para centralizar todas as tarefas administrativas relacionadas ao WebLogic Server”.

Para criar o domínio, vá até o diretório onde instalamos o WebLogic, acesse o subdiretório *oracle_common\common\wlserver\bin* e execute o *config.cmd* caso esteja no Windows ou *config.sh* caso esteja no Linux/Unix.

Quando o assistente de configuração é iniciado, ele propõe um nome padrão para o domínio, que pode ser mantido para nosso exemplo como *base_domain*. Clique então em *Next* e, ao selecionar os *Templates*, mantenha a opção *Basic WebLogic Domain* marcada, e marque também: *Oracle Enterprise Manager – Restricted JRF*, *Oracle Restricted JRF* e *WebLogic Cluster Coherence Extension*, como visto na Figura 12.

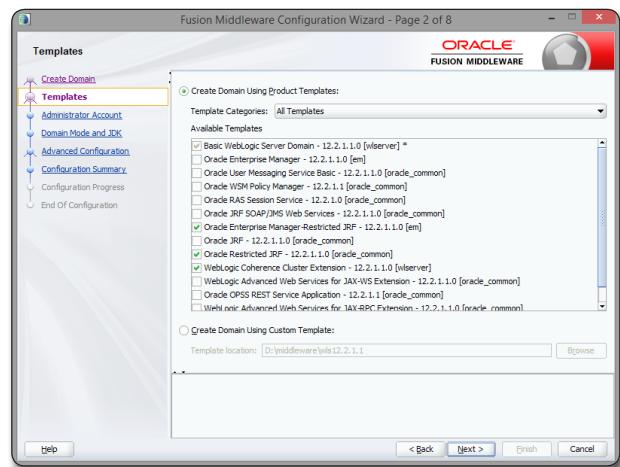


Figura 12. Seleção dos templates para o domínio

Seguindo com o processo de instalação, a única informação que será solicitada será a senha do usuário administrador, nomeado como *weblogic* por padrão. Esse é o usuário que será usado para gerenciar o domínio, servidores e partições, por isso é muito importante lembrar dessa senha depois. Lembre-se também de tomar nota do diretório onde o domínio foi criado. Acessaremos esse diretório para iniciar o servidor mais tarde.

Após o domínio ter sido criado, um checkbox estará disponível para iniciar o Admin Server (veja BOX 5). Você pode clicar nele ou acessar o diretório do domínio e executar o comando *startWebLogic.cmd* ou *startWebLogic.sh*, dependendo do SO onde você instalou o WebLogic. Para mais informações sobre como subir o WebLogic e acessar os consoles de administração, consulte o BOX 6.

Feito isso, você verá uma mensagem similar à apresenta a seguir, indicando que o servidor está no ar:

```
<May 14, 2016 5:12:44 PM BRT> <Notice> <WebLogicServer> <BEA-000365>
<Server state changed to RUNNING.>
```

BOX 5. Admin Server

O Admin Server é o servidor responsável pela administração do domínio, onde efetuamos as configurações do domínio, partições, clusters e servidores gerenciados (managed servers).

BOX 6. Subindo, configurando e gerenciando o servidor de aplicação

A seguir apresentamos as instruções para inicializar o servidor e também para configurar e gerenciar o domínio:

- Subindo seu servidor WebLogic Server: Para subir seu servidor de administração do WebLogic, acesse a pasta onde o domínio do servidor foi criado e execute o comando *startWebLogic.cmd*/*startWebLogic.sh*, dependendo do SO. Caso durante a subida do servidor seja solicitado o usuário, entre com o usuário *weblogic* e sua senha;
- Acessando o Console de Administração do WebLogic: Para acessar o console de administração do WebLogic Server, utilize o endereço <http://<ip:porta>/console>. Com uma instalação local no seu computador, provavelmente esse endereço será <http://localhost:7001/console>;
- Acessando o Enterprise Manager Fusion Middleware Control: Para acessar o EM Fusion Middleware Control, utilize o endereço <http://<ip:porta>/em>. Com uma instalação local no seu computador, provavelmente esse endereço será <http://localhost:7001/em>.

Agora, tente acessar o servidor via o endereço `http://localhost:7001/console` e efetue o login no console de administração utilizando o usuário `weblogic` e a senha que você informou durante a criação do domínio. Esse é o servidor de administração do WebLogic, também conhecido como Admin Server.

Nota

Por razões de simplificação, nesse artigo vamos instalar nossas aplicações no Admin Server, que na verdade é o único servidor de nosso domínio. Esse cenário é suficiente para mostrar como funcionam as partições e o ambiente Multitenant. Em um ambiente de produção, provavelmente você terá um cluster com vários servidores por questões de alta disponibilidade.

Criando os realms de segurança

Os realms são as unidades onde os recursos e elementos relacionados à segurança são configurados. O isolamento dos recursos relacionados à segurança é possível com a configuração de múltiplos realms e associação de cada um deles a uma partição diferente.

Dessa forma, cada partição poderá “enxergar” seus recursos de segurança sem que as outras tenham acesso a eles.

Para a definição do que é um realm de segurança, nada melhor do que a documentação oficial do produto: “um realm de segurança comprehende mecanismos para proteger recursos do WebLogic.

Cada realm de segurança consiste em um conjunto de provedores de segurança configurados, usuários, grupos, roles e políticas. Você utiliza realms para configurar a autenticação, autorização, mapeamento de roles (papéis) e credenciais, auditoria e outros serviços”.

Voltando ao tema, veremos o isolamento na prática com a configuração de diferentes realms para cada partição. Para iniciar a configuração, garanta que o Admin Server esteja no ar e acesse o console de administração do WebLogic.

No console de administração, clique em *Realms de Segurança* e então no botão *Novo* para criar o realm. Nomeie-o como “Realm-A” e selecione a opção *Criar provedores padrão dentro deste novo realm*, como exibido na **Figura 13**.

Por padrão, o WebLogic vem com um servidor LDAP incorporado. Quando marcamos a opção *Criar provedores padrão dentro deste novo realm*, estamos indicando que necessitamos de outra

instância do servidor LDAP incorporado ao novo Realm. Isso é excelente para o nosso exemplo, pois poderemos facilmente cadastrar diferentes usuários em cada um desses realms e demonstrar o isolamento de segurança. No mundo real, provavelmente, você apontará cada realm para um LDAP corporativo ou base de dados onde as credenciais estão armazenadas. Outra opção é utilizar um mesmo servidor LDAP, mas com árvores de busca dos usuários diferentes, uma para cada realm associado às diferentes partições.

Agora, clique no recém-criado realm e então na aba *Usuários e Grupos*. Logo após, clique no botão *Novo* para criar um novo usuário, atribua o nome “`usuarioA`” e entre com qualquer senha com mais de oito caracteres.

Repeta esse processo e crie mais dois realms: Realm-B, com o usuário `usuarioB`; e Realm-C, com o usuário `usuarioC`. Com isso, teremos três partições nesse ambiente, com cada partição apontando para um realm diferente. Lembre-se de tomar nota das senhas dos usuários, pois efetuaremos login com eles em breve.

Criando sua primeira partição

Agora é o momento de criarmos as partições, os componentes mais importantes para o cenário de multitenancy. Para isso, acesse o Enterprise Manager Fusion Middleware Control via `http://localhost:7001/em` com o usuário `weblogic` para logon. Você deve ver uma página similar à **Figura 14**. É nesse console que configuramos as partições, data sources e instalamos a aplicação.

Clique em *Partições de Domínio*, localizado no canto inferior esquerdo, e depois em *Ativar Lifecycle Manager*, para habilitar a edição de partições em seu domínio. Em seguida, reinicie seu servidor WebLogic.

A configuração que fizemos habilita a criação de partições no domínio. Assim, após o servidor no ar novamente, volte ao Enterprise Manager Fusion Middleware Control, clique no combo *Domínio do WebLogic > Ambiente > Alvos Virtuais* e depois em *Criar*, para criar um novo *Virtual Target* (ver **BOX 7**). Logo após, nomeie o virtual target como `VirtualTarget-A` e informe o prefixo de URI como `/particaoA`. Por fim, clique em *Próximo* e defina que o nosso virtual target deve ser implantado no servidor `AdminServer`.

Repeta esse processo e crie mais dois virtual targets, chamados `VirtualTarget-B` e `VirtualTarget-C`, com as URIs `/particaoB` e `/particaoC`, respectivamente, os dois implantados no `AdminServer`.



Figura 13. Criando um novo realm de segurança

WebLogic Multitenant: A nuvem dentro do servidor de aplicação

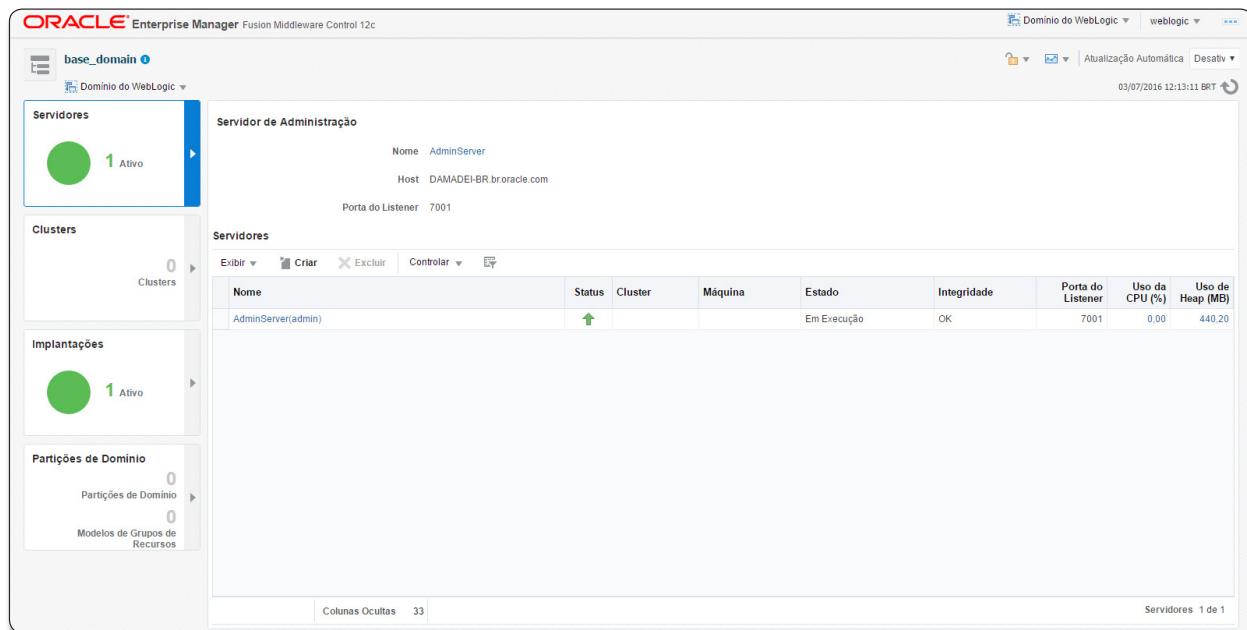


Figura 14. Enterprise Manager Fusion Middleware Control

BOX 7. Virtual Target

Virtual targets são similares a virtual hosts em web servers. O virtual target fornece uma abstração de um alvo físico para os recursos que são direcionados para o alvo virtual. Ele também provê um HTTP server separado, para que recursos que utilizam o virtual target utilizem um HTTP server diferente do HTTP server padrão, e fornece, ainda, um mapeamento das requisições de entrada para o virtual target, o que resulta em um direcionamento para uma partição específica do domínio. Para mais informações sobre virtual targets, a seção Links possui o endereço para um blog que apresenta uma explicação bastante detalhada sobre o tema.

Agora é a hora de criar as partições. Para isso, clique em *Partições de Domínio* e então em *Criar*. Nomeie a partição como “Particao-A”, selecione o Realm-A como realm de segurança da partição e clique em *Próximo*. Como cada partição precisa de pelo menos um alvo virtual (virtual target) associado a ela, que redirecionará as requisições para essa partição, faremos essa associação nesse passo, selecionando o VirtualTarget-A como o alvo virtual da partição. Então, marque a opção para que ele seja o alvo padrão, clicando em *Próximo* em seguida.

Além dessa associação com os alvos virtuais, as partições também possuem um elemento chamado Grupo de Recursos (ou *Resource Groups* em inglês), que, como o nome indica, agrupa recursos como data sources, objetos JMS, etc. e os associa à partição. É isso que chamamos de “escopo de partição”.

Voltando ao passo a passo da criação da partição, vamos agora criar o grupo de recursos da partição A, que deve se chamar particaoA-RG. Esse nome é especificado nesse momento, onde também devemos mover o VirtualTarget-A para a lista de alvos. Feito isso, clique em *Próximo* e então em *Criar*, para criar a partição.

Ao ser definida, a partição não é inicializada e, devido a isso, não conseguimos acessar as suas aplicações ou recursos. Para inicializar a partição, acesse o menu *Partições de Domínio*, clique

no item *Particao-A* > *Control* > *Iniciar*. Repita esses passos para criar as partições B e C e implante-as nos alvos VirtualTarget-B e VirtualTarget-C, respectivamente.

Criando os data sources

Com a partição e grupo de recursos criados, vamos definir os data sources para acesso ao repositório de dados da nossa aplicação. Esses data sources possuirão escopo de partição e serão instalados no grupo de recursos de sua partição.

Para prosseguir com a criação, acesse a partição Particao-A, *Grupos de Recursos* e selecione o grupo particaoA-RG. Aqui temos acesso a todos os elementos disponíveis no grupo de recursos. Desse modo, clique em *Serviços*, que é a seção que contém os serviços disponibilizados pelo grupo de recursos, mantenha-se na aba *JDBC*, para ter acesso aos serviços desse tipo, e clique em *Criar* > *Origem de Dados Genérica*. Feito isso, nomeie o data source como “appDS”, clique em *Selecionar...*, para alterar o tipo de banco de dados, e escolha MySQL, mantendo o driver que já vem especificado, conforme a Figura 15.



Figura 15. Alterando o tipo de banco de dados para MySQL

Após selecionar o driver JDBC, é hora de especificar o nome JNDI no passo seguinte da configuração. Para isso, vamos utilizar o valor “jdbc/appDS” no campo *Nome da JNDI*, pois esse será o valor para efetuar o lookup do data source.

Logo após, clique em próximo para prosseguirmos com a configuração.

Com os nomes do data source e JNDI configurados e o driver JDBC alterado para MySQL, chegou a hora de especificar as informações de conectividade com o banco de dados. Essa configuração é efetuada clicando no botão *Gerar URL e Propriedades* e preenchendo as informações conforme a **Figura 16**. Observe que estamos entrando com o nome do banco de dados (*db1*), e isso irá variar quando criarmos os data sources para as demais partições, uma vez que cada partição acessará uma base de dados diferente.

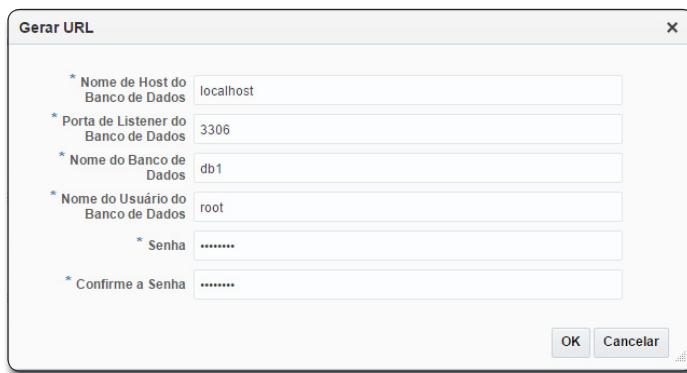


Figura 16. Gerando a URL para acesso ao banco de dados

Ao entrar com as informações de conexão com o banco de dados e clicar em *OK*, seremos direcionados para o último passo da criação do data source, que é o teste da conexão. Nessa janela, haverá um botão, chamado *Testar Conexão de Banco de Dados*, no qual devemos clicar para validar se o que especificamos está correto e se é possível conectar com o banco de dados. O teste deverá ser bem-sucedido, o que permite prosseguirmos com o assistente, clicando em *Próximo* e depois em *Criar*.

Neste momento, repita a operação para os outros grupos de recursos das demais partições conforme a **Tabela 1**, para que cada partição tenha um data source com o nome JNDI *jdbc/appDS*, mas apontando para diferentes bases de dados do MySQL.

| Partição | Grupo de Recursos | Nome JNDI | Base de Dados MySQL |
|------------|-------------------|------------|---------------------|
| Particao-B | particaoB-RG | jdbc/appDS | db2 |
| Particao-C | particaoC-RG | jdbc/appDS | db3 |

Tabela 1. Configuração dos data sources para as demais partições

Ao analisar os data sources do domínio, englobando todas as partições, temos os três data sources com o mesmo nome JNDI. Porém, como cada um deles está em um escopo de partição diferente, não há conflitos. A **Figura 17** ilustra isso.

Para concluir, vamos fazer o deploy da aplicação. Assim como no caso do data source, as aplicações são instaladas no grupo de recursos da partição. Para realizar a implantação, volte à página inicial do Enterprise Manager clicando no link *Enterprise Manager* ao lado do logo da Oracle, clique em *Partições de Domínio* no canto inferior esquerdo, *Grupo de Recursos* no segundo item do

menu e no único grupo de recursos da partição, por exemplo, *particao-ARG* no caso da partição A. Em seguida, acesse a aba *Implantações* e a opção de menu *Deployment > Implantar*.

Nesse ponto podemos o fazer upload do arquivo de instalação da aplicação ou indicar um arquivo já existente no servidor.

Faça o upload do arquivo clicando em *Escolher Arquivo* e selecione o WAR *CrmApis.war*, que você baixou do site da DevMedia. Logo após, prossiga com o assistente de implantação clicando em *Próximo* até o último passo e depois em *Implantar* quando o botão surgir. Instale a mesma aplicação nas partições B e C.

Isolamento na prática

Nesse ponto já temos as partições criadas e a aplicação instalada em cada uma delas. Já podemos, portanto, acessar a aplicação na partição via sua URL. Como exemplo, temos: <http://localhost:7001/particaoA/crm-apis> para acesso na partição A. Nessa URL podemos notar um ponto interessante, que é o prefixo do virtual target que utilizamos (*/particaoA*) e que possui a responsabilidade de redirecionar a requisição HTTP para a partição correspondente.

Ao acessar a aplicação, será solicitado o login via usuário e senha. Nesse momento será possível verificar o isolamento de segurança na prática. Para isso, efetue o login com o usuário *usuarioA*, que é o que está cadastrado no realm dessa partição. Ao realizar o acesso, você será direcionado para uma listagem de clientes vazia. Cadastre, então, um novo cliente e, ao finalizar, veja que ele aparece na lista de clientes cadastrados, conforme demonstra a **Figura 18**.

Agora faça logout e tente entrar com o usuário *usuarioB* nessa mesma URL. Você verá que não será bem-sucedido, pois estamos acessando a aplicação na partição A (observe o prefixo */particaoA* na URL) com um usuário que está em outro realm de segurança. Tente também efetuar logon com o usuário *weblogic* e veja que da mesma forma não será possível. Isso se deve ao fato do usuário *weblogic* ser administrador do ambiente, mas ele não consta no realm de segurança em uso pela partição e, por essa razão, não acessamos a aplicação com ele. Esse fato demonstra o isolamento em relação à segurança, pois cada partição está utilizando o seu realm e não enxerga nem mesmo o realm padrão do servidor de aplicação.

Dito isso, acesse a aplicação via URL <http://localhost:7001/particaoB/crm-apis>, para acessar a mesma aplicação na partição B, e utilize o usuário *usuarioB* para acesso. Note que agora ele funciona.

Aqui vemos outro ponto importante em nossa jornada pelo mundo Multitenant. Repare que apesar das aplicações adotarem o mesmo nome JNDI para acesso ao data source, a partição B acessa outro data source, diferente da partição A, e esse, por sua vez, utiliza uma outra base de dados do MySQL. Como ainda não cadastramos um cliente na aplicação instalada na partição B, a lista de clientes estará vazia, mostrando o isolamento de recursos na prática. Esse era o último ponto que faltava evidenciar: o isolamento do contexto e namespace. Como já vimos no decorrer do artigo, cada partição só tem visibilidade daquilo que está no seu escopo ou do que é global. A **Figura 19** ilustra o acesso à aplicação na partição B.

WebLogic Multitenant: A nuvem dentro do servidor de aplicação

Origens de Dados JDBC

Esta página lista as origens de dados do sistema JDBC que foram criadas neste domínio. Você pode criar, configurar, testar, controlar ou excluir as origens de dados do sistema desta página.

| | Nome | Nome da JNDI | Tipo | Escopo | Grupo de Recursos / Modelo | Partição | Alvos |
|---|-------|--------------|----------|---|----------------------------|------------|-----------------|
| 1 | appDS | jdbc/appDS | Genérico | Grupos de Recursos da Partição de Domínio | particaoA-RG | Particao-A | VirtualTarget-A |
| 2 | appDS | jdbc/appDS | Genérico | Grupos de Recursos da Partição de Domínio | particaoB-RG | Particao-B | VirtualTarget-B |
| 3 | appDS | jdbc/appDS | Genérico | Grupos de Recursos da Partição de Domínio | particaoC-RG | Particao-C | VirtualTarget-C |

Figura 17. Lista de data sources do domínio

Ola usuarioA

Listagem de Clientes

Listagem de clientes para demonstrar as capacidades do WebLogic Multitenant.

| ID | Nome | RG | CPF | Nascimento | Sexo |
|----|---------------|-----------|-------------|------------|-----------|
| 1 | Daniel Amadei | 121112223 | 12312312399 | 25/11/1980 | Masculino |

Logout

A Excluir

Figura 18. Lista de clientes cadastrados na aplicação da partição A

Ola usuarioB

Listagem de Clientes

Listagem de clientes para demonstrar as capacidades do WebLogic Multitenant.

| ID | Nome | RG | CPF | Nascimento | Sexo |
|----|------|----|-----|------------|------|
|----|------|----|-----|------------|------|

Logout

Novo

Figura 19. Lista de clientes cadastrados na aplicação da partição B

O que mais além de consolidação e isolamento?

Além dos benefícios da consolidação e isolamento, obtidos através dos recursos Multitenant, o conceito de microcontainers provido pelas partições também dá a possibilidade de desconectar as partições de um ambiente e conectá-las em outro, sem esforço. Esses recursos vão muito além do modelo tradicional de empacotamento de aplicações Java EE, que é caracterizado basicamente pela criação de arquivos JAR, WAR ou EAR com os artefatos da aplicação.

Essa possibilidade de exportar a partição e importá-la em outro ambiente se diferencia da abordagem de empacotar a aplicação, pois permite que todos os recursos da partição também sejam exportados. Por recursos, compreenda elementos que são configurados, mantidos e providos pelo servidor de aplicação, ou seja, além da aplicação, são levados recursos como data sources JDBC e filas JMS, que teriam que ser criados manualmente.

Além dessa funcionalidade, também podemos migrar uma partição de um cluster para outro. Nesse processo, a partição pode estar executando e as aplicações sendo acessadas. Quando a partição for migrada, as requisições serão direcionadas para o novo cluster e as sessões HTTP, se houverem, também serão migradas. Assim, o usuário final não terá qualquer ideia de que houve uma migração, uma vez que não haverá indisponibilidade.

No decorrer do artigo, vimos que existem vários elementos importantes que devem ser levados em consideração para podermos classificar um servidor de aplicação Java EE como sendo multitenancy “de verdade”. Não é apenas uma questão de instalar aplicações no mesmo ambiente e esperar pelo melhor. O servidor de aplicação deve estar preparado para isolar recursos, contextos e aplicações, prover diferentes configurações para cada tenant e, o mais importante, deve possibilitar o controle do consumo de recursos por cada tenant com o objetivo de proteger ou priorizar os mais críticos.

Autor



Daniel Cicero Amadei

daniel.amadei@gmail.com

Bacharel em Sistemas de Informação pelo Mackenzie e pós-graduado pela Fundação Vanzolini. Trabalha com TI desde 1999, já tendo atuado como Desenvolvedor, Analista e Arquiteto de Sistemas. Atualmente é consultor de vendas na Oracle com foco em Java, Middleware e Cloud.



Links:

Endereço para download do WebLogic Server.

<http://www.oracle.com/technetwork/middleware/weblogic/downloads/index.html>

Documentação do WebLogic Server.

<http://docs.oracle.com/middleware/12211/wls/index.html>

Blog sobre os Virtual Targets no WebLogic Multitenant.

https://blogs.oracle.com/dipol/entry/partition_targeting_and_virtual_targets

Web services RESTful: Como adicionar segurança com JWT

Aprenda como proteger sua API RESTful com JSON Web Token (JWT)

Segurança é um requisito fundamental em grande parte dos sistemas atuais, pois é necessário garantir que cada usuário possua acesso somente às funcionalidades que se adequem às suas competências, visando manter a integridade do sistema e das informações armazenadas.

Para aplicações web, a utilização da forma tradicional de segurança, que mantém, após uma autenticação, todas as autorizações de um usuário em sua sessão, geralmente é adequada. Serviços RESTful, por outro lado, são, por definição, stateless. Isso elimina essa forma como opção e nos obriga a buscar alternativas. Atualmente, duas das alternativas mais conhecidas são:

- **HTTP Basic Authentication (BA)** – Nesse tipo de autenticação, as credenciais de acesso (username/password) são enviadas, no header **Authorization**, em todas as requisições. Elas devem seguir o formato “username:password”, serem codificadas com base64 e precedidas por “Basic”. Por exemplo: para o username “Alladin” com password “open sesame”, o header seria “**Authorization**: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==”. Deve-se notar que, por conta de suas características, o sistema cliente deve guardar as credenciais do usuário para evitar solicitá-las em cada requisição, o que pode representar uma nova vulnerabilidade;

Nota

O padrão para o header **Authorization** foi apresentado pela W3C na definição do HTTP 1.0 (RFC 1945) e segue o formato “**Authorization**: <type> <credentials>”, onde <type> indica o tipo de autenticação.

Fique por dentro

Este artigo apresenta um novo padrão, denominado JSON Web Token (JWT), para a criação de tokens capazes de transportar informações de uma forma compacta e confiável. Além disso, aborda também uma maneira de utilizar esses tokens como uma alternativa stateless, de forma semelhante a utilizada por gigantes como Google e Microsoft, aos processos existentes de autenticação para APIs RESTful. No artigo, será analisada a implementação de uma API RESTful com segurança via JWT, baseada no framework Spring Security, para exemplificar todos os conceitos.

- **Token Based Authentication** – Nesse tipo de autenticação, o cliente realiza um login passando as credenciais do usuário e recebe um token, que possui uma assinatura que inviabiliza sua adulteração. As requisições subsequentes adicionam o token a um header e as informações contidas nele garantem acesso aos serviços desejados.

Neste artigo, focaremos em um tipo específico de *Token Based Authentication*, que utiliza um novo padrão de token, denominado JSON Web Token (JWT). Além de conhecer esse novo tipo de token, veremos como utilizá-lo em conjunto com o framework Spring Security para proteger uma API RESTful.

JWT

JSON Web Token, ou JWT, é um padrão aberto (RFC 7519), de uso geral, que possibilita a troca segura de informações entre partes, na forma de objetos JSON. Como vantagens, o JWT possui o fato de ser mais compacto que alternativas baseadas em XML e a capacidade de ser autocontido, ou seja, possuir todas as informações relevantes sobre um assunto, dispensando consultas adicionais a um eventual banco de dados para recuperar os dados.

Um JWT é composto por três partes: header, payload e signature (assinatura); todas codificadas com base64 (vide **BOX 1**) e separadas por pontos (.). A seguir, analisaremos cada uma delas em detalhes.

BOX 1. Base64

Base64 é um método para codificação de dados para transferência na internet definido pelo padrão RFC 4648, que, como seu nome indica, utiliza apenas 64 caracteres ([A-Z], [a-z], [0-9], "/" e "+"). O método consiste em, primeiramente, transformar o texto original em um número binário. Essa transformação leva em consideração a codificação original do texto (ASCII). Após isso, o número binário resultante é convertido, por meio de uma tabela, para base64. O texto em ASCII "teste", por exemplo, corresponde ao número binário "01110100 01100101 01110011 01110100 01100101", que, por sua vez, pode ser codificado em base64 como "dGVzdGU=".

Header

O header (cabeçalho) de um JWT é um JSON e, geralmente, contém o algoritmo de hashing utilizado na assinatura e o tipo de token (JWT).

A seguir é apresentado um exemplo de header não codificado:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Mesmo header codificado com base64:

```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9
```

Payload

O payload (ou corpo) de um JWT é também um JSON e contém as informações relevantes ao assunto, denominadas **claims**, na forma de pares chave/valor.

Existem três tipos de **claims**, a saber:

- **Registradas** – Um conjunto de **claims** cujo uso não é obrigatório, mas recomendado. São elas: **iss** (Issuer), identificador de quem gerou o JWT; **sub** (Subject), identificador único que representa o assunto do JWT; **aud** (Audience), identifica o recipiente do JWT; **exp** (Expiration Time), data/hora de expiração do JWT; **nbf** (Not Before), data/hora de ativação do JWT, a partir da qual ele será válido; **iat** (Issued At), data/hora de geração do JWT; e **jti** (JWT Id), identificador único do JWT. Aqui, é interessante notar que, para manter o token compacto, os nomes das **claims** possuem apenas três caracteres;

- **Públicas** – **Claims** criadas por usuários de JWTs. Para evitar colisões, os nomes dessas **claims** devem ser registrados na IANA ou definidos como URIs que contenham um namespace resistente a colisão (como um nome de domínio, por exemplo);

- **Privadas** – **Claims** criadas com nomes não registrados na IANA (vide **BOX 2**) e não resistentes a colisão. Devem ser utilizadas com cuidado, em comum acordo entre as partes envolvidas (consumidores e produtores).

Nota

Uma colisão ocorre quando um mesmo nome pode possuir diferentes significados/usos. Isso gera problemas de comunicação, fazendo com que informações sejam interpretadas incorretamente quando transferidas para diferentes sistemas ou módulos.

BOX 2. IANA

A IANA (Internet Assigned Numbers Authority) é uma organização privada, sem fins lucrativos, responsável pela atribuição de nomes e números globalmente únicos, utilizados em padrões técnicos que regem a Internet. Suas atividades podem ser agrupadas em três categorias: nomes de domínio, recursos numéricos e atribuições de protocolos.

Um exemplo de payload, contendo **claims** registradas e privadas, é apresentado na **Listagem 1**.

Listagem 1. Exemplo de payload com claims registradas e privadas.

```
{  
  "adm": "true",  
  "app": "Postal",  
  "iss": "br.com.javamagazine",  
  "sub": "Administrator",  
  "exp": 1477781160  
}
```

Nota

É importante mencionar que, geralmente, as informações contidas no payload não são criptografadas. Nada impede, no entanto, de se proteger informações sigilosas. Isso, contudo, pode aumentar a quantidade de processamento necessária para processar o token, causando problemas de desempenho.

Assinatura

A assinatura serve para identificar quem enviou o JWT e validar se ele não foi modificado no trajeto. Diversos algoritmos de chave simétrica e assimétrica (vide **BOX 3**) estão disponíveis para assinar um JWT, por exemplo: HMAC-SHA256, HMAC-SHA512, RSA-SHA256, RSA-SHA512, ECDSA-SHA256, ECDSA-SHA-512, entre outros. A escolha do algoritmo adequado depende das necessidades de segurança da aplicação desenvolvida.

BOX 3. Criptografia simétrica e assimétrica

Algoritmos de criptografia de chave simétrica utilizam uma chave secreta para codificar e decodificar um conteúdo. Esse tipo de algoritmo não é eficaz quando há a necessidade de troca de informações, ou seja, quando o responsável por decodificar um determinado conteúdo não é o mesmo que codificou. Isso ocorre porque, nesses casos, se faz necessário compartilhar a chave secreta, o que acrescenta uma nova vulnerabilidade ao processo.

Algoritmos de criptografia de chave assimétrica, por outro lado, utilizam dois tipos de chave, denominadas privada e pública, para codificar ou decodificar um conteúdo. Como forma de garantir a origem de uma informação, utiliza-se a chave privada para codificação e a pública para decodificação. Com essa configuração, é impossível codificar um conteúdo apenas com a chave pública, que pode ser distribuída livremente entre os receptores da informação.

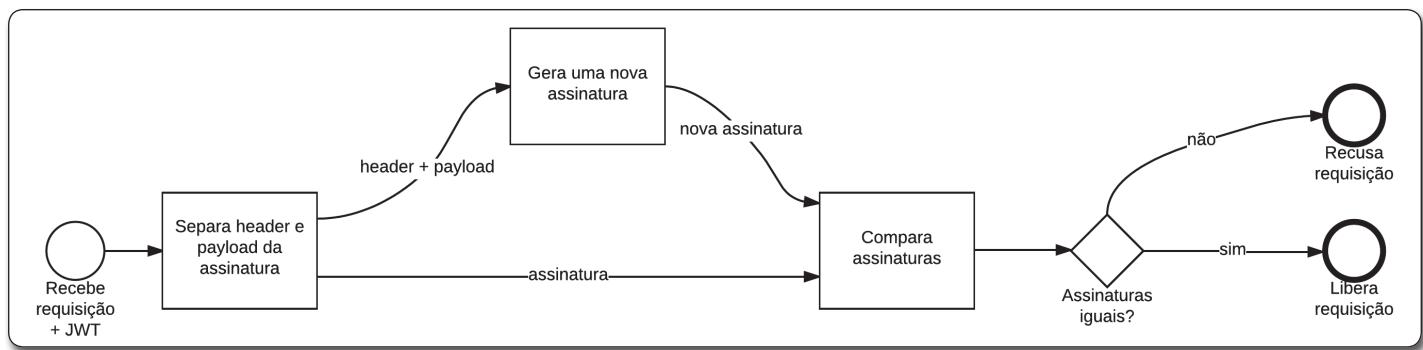


Figura 1. Processo de validação

Os algoritmos de codificação utilizam como entradas:

- O header codificado com base64 concatenado com o payload, também codificado com base64;
- Uma chave, definida pelo desenvolvedor do sistema. Chaves, idealmente, devem possuir uma quantidade de bits, gerados aleatoriamente, igual ou superior à do algoritmo escolhido. Um exemplo de chave, em hexadecimal, para um algoritmo de 256 bits, como o HMAC-SHA256, seria:

"620FD22D31267AFF7D788833E311DA62C1A6CEE8F6A5FB77307B65F2EB2890B7"

Em resumo, o pseudocódigo do processo de codificação se pareceria com a seguinte estrutura:

```

AlgoritmoCodificacao()
base64UrlEncoder(header) + "."
base64UrlEncoder(payload),
privateKey
)

```

O processo de validação, no servidor, consiste em gerar uma nova assinatura com o header e o payload recebidos e a comparar com a assinatura original. Qualquer alteração no conteúdo do header ou do payload de um JWT o tornará inválido, pois isso fará com que a nova assinatura gerada seja diferente da original. Todo esse processo de validação pode ser mais facilmente visualizado no diagrama apresentado na **Figura 1**.

Segurança com JWT

Por conta de suas características, o JWT é especialmente interessante para uma implementação de Token Based Authentication. Sua capacidade de ser autocontido permite transmitir todas as informações relevantes aos processos de autorização e autenticação. Isso, aliado ao seu formato compacto, faz com que seja possível realizar uma autenticação segura, completamente stateless, com uma carga mínima em cada requisição e sem expor qualquer informação privada do usuário.

Um efeito colateral positivo causado pelo fato do JWT ser autocontido consiste na possibilidade de gerá-lo em uma aplicação e utilizá-lo para autenticação em diversas outras. Essa característica faz com que o JWT seja frequentemente utilizado em OAuth (vide **BOX 4**).

BOX 4. OAuth

OAuth é um padrão aberto para autorização que permite a um usuário, denominado Resource Owner, compartilhar recursos de uma aplicação, denominada Resource Server, com uma terceira aplicação, denominada Client, sem expor suas credenciais.

O fluxo para uma implementação OAuth 2.0, como ilustrado na **Figura 2**, se inicia com o Client se registrando no Resource Server, fornecendo dados básicos e uma URI de redirecionamento, e recebendo um Client Id e um Client Secret. Logo após, quando um Resource Owner acessa o Client, ele é direcionado ao Resource Server, que solicita uma autorização para compartilhar determinados recursos. Uma vez que o Resource Owner aceita, o Resource Server fornece um token de acesso ao Client, por meio da URI de redirecionamento, com o qual ele tem acesso aos recursos solicitados para, entre outras coisas, autenticar o Resource Owner. O processo de logar com uma conta Google em uma aplicação, por exemplo, ilustra bem o fluxo do OAuth 2.0.

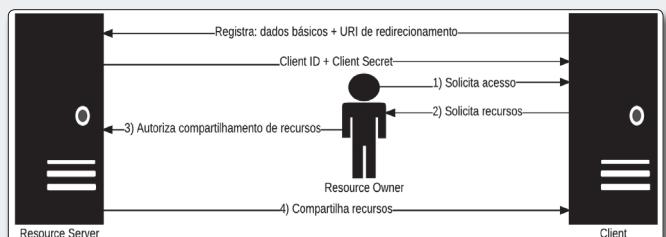


Figura 2. Fluxo do OAuth 2.0.

Nota

Um header de Authorization que contém um JWT geralmente segue o formato "Authorization: Bearer <token>", o mesmo utilizado pelo OAuth 2.0.

JWT com Spring Security

Nesta seção criaremos uma API RESTful com segurança via JWT, utilizando como base o framework Spring Security. O código-fonte completo do exemplo se encontra na área de downloads desta edição. Durante esse processo de construção, será assumido que o leitor possui conhecimento prévio em criação de APIs RESTful com o framework Spring MVC e configuração de autenticação/autorização convencional utilizando o framework Spring Security.

Inicialmente, vamos configurar o projeto com o Apache Maven. Para isso, especificaremos as dependências necessárias no arquivo *pom.xml*, apresentado na **Listagem 2**. Eis uma pequena descrição das principais dependências:

- Linhas 20 a 24 - O framework Spring Web MVC facilita a criação de APIs REST;

Web services RESTful: Como adicionar segurança com JWT

- Linhas 25 a 34 - O framework Spring Security cuida das funcionalidades de autenticação e autorização e, com algumas modificações, serve como base para nossa implementação de segurança via JWT;
- Linhas 35 a 39 - A serialização/desserialização JSON fica por conta da biblioteca Jackson;
- Linhas 40 a 44 - A biblioteca JJWT é responsável por realizar a criação, parsing e validação dos tokens.

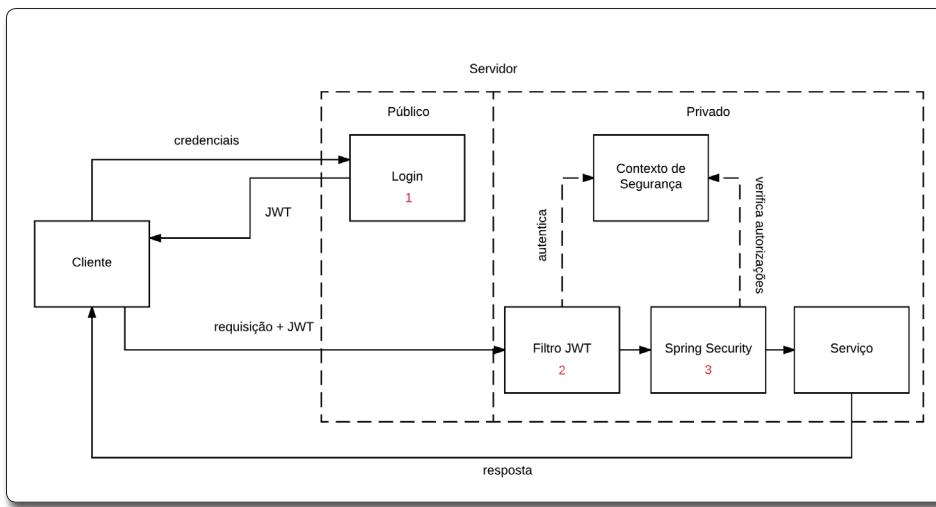


Figura 3. Estrutura da aplicação

A Figura 3 apresenta a estrutura desejada para nossa aplicação. O fluxo de acesso padrão ocorre quando o usuário realiza um login bem-sucedido por meio de uma API pública (1), recebe um JWT e passa a utilizá-lo nas requisições seguintes que envolvam serviços privados. Para cada uma dessas requisições, o filtro JWT (2) valida o token e insere os dados do usuário no contexto de segurança. A partir daí o filtro do Spring Security (3) realiza o processo de autorização, verificando se o usuário tem acesso ao recurso desejado. Nas próximas seções detalharemos cada um desses componentes, assim como as configurações necessárias para interligá-los.

Configuração

Iniciemos, então, com a configuração do sistema. Para isso, utilizaremos três arquivos: *web.xml*, para a configuração geral da aplicação web; *applicationContext.xml*, para a configuração dos elementos do framework Spring; e *applicationContext-security.xml*, para a configuração do framework Spring Security.

No arquivo *web.xml*, apresentado na **Listagem 3**, configuramos o servlet do Spring MVC (linhas 6 a 19), indicando que os arquivos de configuração de contexto

Listagem 2. Código do arquivo pom.xml.

```
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/maven-v4_0_0.xsd">
05   <modelVersion>4.0.0</modelVersion>
06   <groupId>br.gov.sp.fatec</groupId>
07   <artifactId>SpringRestSecurityJwt</artifactId>
08   <packaging>war</packaging>
09   <version>1.0-SNAPSHOT</version>
10  <name>SpringRestSecurityJwt Maven Webapp</name>
11  <url>http://maven.apache.org</url>
12  <properties>
13    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14    <org.springframework.version>4.2.6.RELEASE</org.springframework.version>
15    <org.springframework.security.version>4.0.3.RELEASE
16    <jackson.version>2.4.1</jackson.version>
17    <jjwt.version>0.6.0</jjwt.version>
18    <javax.servlet.version>2.5</javax.servlet.version>
19  </properties>
20  <dependencies>
21    <dependency>
22      <groupId>org.springframework</groupId>
23      <artifactId>spring-webmvc</artifactId>
24      <version>${org.springframework.version}</version>
25    </dependency>
26    <dependency>
27      <groupId>org.springframework.security</groupId>
28      <artifactId>spring-security-web</artifactId>
29      <version>${org.springframework.security.version}</version>
30    </dependency>
31    <dependency>
32      <groupId>org.springframework.security</groupId>
33      <artifactId>spring-security-config</artifactId>
34    </dependency>
35    <dependency>
36      <groupId>com.fasterxml.jackson.core</groupId>
37      <artifactId>jackson-databind</artifactId>
38      <version>${jackson.version}</version>
39    </dependency>
40    <dependency>
41      <groupId>io.jsonwebtoken</groupId>
42      <artifactId>jjwt</artifactId>
43      <version>${jjwt.version}</version>
44    </dependency>
45    <dependency>
46      <groupId>javax.servlet</groupId>
47      <artifactId> servlet-api</artifactId>
48      <version>${javax.servlet.version}</version>
49      <scope>provided</scope>
50    </dependency>
51  </dependencies>
52  <build>
53    <finalName>SpringRestSecurityJwt</finalName>
54    <plugins>
55      <plugin>
56        <groupId>org.apache.maven.plugins</groupId>
57        <artifactId>maven-compiler-plugin</artifactId>
58        <configuration>
59          <source>1.7</source>
60          <target>1.7</target>
61        </configuration>
62      </plugin>
63    </plugins>
64  </build>
65 </project>
```

do Spring seguem o padrão “/WEB-INF/applicationContext*.xml” (linha 11), e o filtro do Spring Security (linhas 21 a 33), indicando que qualquer requisição, antes de ser processada, deve passar por sua verificação (linha 30).

No arquivo *applicationContext.xml*, apresentado na **Listagem 4**, indicamos que o framework Spring deve procurar por classes anotadas a partir do pacote **br.com.javamazine** (linha 13). Além disso, habilitamos o uso de anotações do framework Spring MVC (linha 15).

Listagem 3. Código do arquivo web.xml.

```
01 <!DOCTYPE xml>
02 <web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
03   <display-name>Spring Rest Security</display-name>
04
05   <!-- Add Support for Spring -->
06   <servlet>
07     <servlet-name>spring</servlet-name>
08     <servlet-class>org.springframework.web.servlet.DispatcherServlet
09       </servlet-class>
10   <init-param>
11     <param-name>contextConfigLocation</param-name>
12     <param-value>/WEB-INF/applicationContext*.xml</param-value>
13   </init-param>
14   <load-on-startup>1</load-on-startup>
15 </servlet>
16 <servlet-mapping>
17   <servlet-name>spring</servlet-name>
18   <url-pattern>/</url-pattern>
19 </servlet-mapping>
20
21 <!-- Spring Security -->
22 <filter>
23   <filter-name>springSecurityFilterChain</filter-name>
24   <filter-class>
25     org.springframework.web.filter.DelegatingFilterProxy
26   </filter-class>
27 </filter>
28 <filter-mapping>
29   <filter-name>springSecurityFilterChain</filter-name>
30   <url-pattern>/*</url-pattern>
31   <dispatcher>FORWARD</dispatcher>
32   <dispatcher>REQUEST</dispatcher>
33 </filter-mapping>
34
35 </web-app>
```

Listagem 4. Código do arquivo applicationContext.xml.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
03   xmlns:context="http://www.springframework.org/schema/context"
04   xmlns:mvc="http://www.springframework.org/schema/mvc"
05   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
06   xsi:schemaLocation="http://www.springframework.org/schema/beans
07   http://www.springframework.org/schema/beans/spring-beans.xsd
08   http://www.springframework.org/schema/context
09   http://www.springframework.org/schema/context/spring-context.xsd
10  http://www.springframework.org/schema/mvc
11  http://www.springframework.org/schema/mvc/spring-mvc.xsd">
12
13  <context:component-scan base-package="br.com.javamazine"/>
14
15  <mvc:annotation-driven />
16
17 </beans>
```

Finalmente, no arquivo *applicationContext-security.xml*, apresentado na **Listagem 5**, habilitamos o uso das anotações **@PreAuthorize** e **@PostAuthorize** (linha 10); definimos um entry point customizado (linha 12); indicamos que a autenticação será stateless (linha 13), ou seja, que os dados do usuário autenticado não permanecerão na sessão; desabilitamos a proteção contra CSRF (linha 14), pois o JWT já previne esse tipo de fraude; incluímos um filtro customizado para tratar os JWTs antes do filtro correspondente ao form de login (linha 15); definimos um bean para nosso filtro customizado (linhas 18 a 20); e configuramos um Authentication Manager (linhas 22 a 26), que utiliza o bean **segurancaService** como um Authentication Provider customizado (linha 23) para recuperar informações de usuários, cujas senhas não se encontram codificadas (linha 24).

Nas próximas seções analisaremos cada um dos elementos configurados.

Listagem 5. Arquivo applicationContext-security.xml.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans:beans xmlns="http://www.springframework.org/schema/security"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xmlns:beans="http://www.springframework.org/schema/beans"
05   xsi:schemaLocation="http://www.springframework.org/schema/beans
06   http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
07   http://www.springframework.org/schema/security
08   http://www.springframework.org/schema/security/spring-security.xsd">
09
10  <global-method-security pre-post-annotations="enabled"/>
11
12  <http entry-point-ref="restAuthenticationEntryPoint"
13    create-session="stateless">
14    <csrf disabled="true"/>
15    <custom-filter before="FORM_LOGIN_FILTER" ref="jwtAuthenticationFilter"/>
16  </http>
17
18  <beans:bean id="jwtAuthenticationFilter"
19    class="br.com.javamagazine.security.JwtAuthenticationFilter">
20  </beans:bean>
21
22  <authentication-manager alias="authenticationManager">
23    <authentication-provider user-service-ref="segurancaService">
24      <password-encoder hash="plaintext"/></password-encoder>
25    </authentication-provider>
26  </authentication-manager>
27
28 </beans:beans>
```

Entry Point

O Entry Point é responsável por definir que ação tomar quando ocorre um acesso de usuário não autenticado (que não realizou login). O comportamento normal é redirecionar para a página de login, contudo, por se tratarem de APIs REST, esse comportamento seria indesejado. Tendo isso em vista, definimos um Entry Point customizado, apresentado na **Listagem 6**, que sempre retorna um erro 401 — UNAUTHORIZED (linha 19) para qualquer tipo de erro de autenticação. Caso seja necessário um tratamento mais elaborado, dependente do erro, os dados da requisição recusada e do erro se encontram disponíveis nos parâmetros **request** e **authException**, respectivamente (linhas 16 e 17).

Web services RESTful: Como adicionar segurança com JWT

Listagem 6. Código do Entry Point (RestAuthenticationEntryPoint).

```
01 package br.com.javamagazine.security;
02
03 import java.io.IOException;
04
05 import javax.servlet.http.HttpServletRequest;
06 import javax.servlet.http.HttpServletResponse;
07
08 import org.springframework.security.core.AuthenticationException;
09 import org.springframework.security.web.AuthenticationEntryPoint;
10 import org.springframework.stereotype.Component;
11
12 @Component
13 public class RestAuthenticationEntryPoint implements AuthenticationEntryPoint {
14
15     @Override
16     public void commence(HttpServletRequest request,
17             HttpServletResponse response, AuthenticationException authException)
18             throws IOException {
19         response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
20             "Unauthorized");
21     }
}
```

Authentication Provider

Para que o Spring Security possa gerenciar os processos de autenticação e autorização, precisamos definir como gerenciar usuários e suas permissões.

Para isso, utilizaremos as classes **User** e **Authority**, que implementam, respectivamente, as interfaces **UserDetails** e **GrantedAuthority** do Spring Security. A Figura 4 apresenta um diagrama com ambas as classes.

Nosso Authentication Provider customizado precisa implementar a interface **UserDetailsService** do Spring Security. Essa interface possui um único método, denominado **loadUserByUsername()**, que recebe como parâmetro o nome de usuário informado durante o login e deve retornar uma implementação da interface **UserDetails** contendo seus dados e suas autorizações. Idealmente, buscariamos tais informações no banco de dados, mas como esse não é o foco desse artigo, utilizaremos uma abordagem simplificada.

A Listagem 7 apresenta nossa implementação, que sempre retorna um **User** com username e password idênticos ao parâmetro recebido. Se o parâmetro possuir valor **admin**, a **Authority** associada a esse **User** será **ROLE_ADMIN**; caso contrário a **Authority** será **ROLE_USER**.

Até este ponto, realizamos apenas customizações básicas no Spring Security. Os próximos elementos, entretanto, são exclusivos do JWT e fogem do fluxo usual de autenticação/autorização, no qual, após uma autenticação (login) bem-sucedida, todos os acessos subsequentes utilizam as informações de autorização armazenadas na sessão.

Geração e parsing de tokens

Para gerar os JWTs durante o processo de autenticação e realizar o parsing deles durante as requisições, foi criada uma classe utilitária, denominada **JwtUtils**, cujo código se encontra na Listagem 8. Nesse projeto, utilizaremos um algoritmo de chave simétrica (HMAC SHA256) para gerar a assinatura.

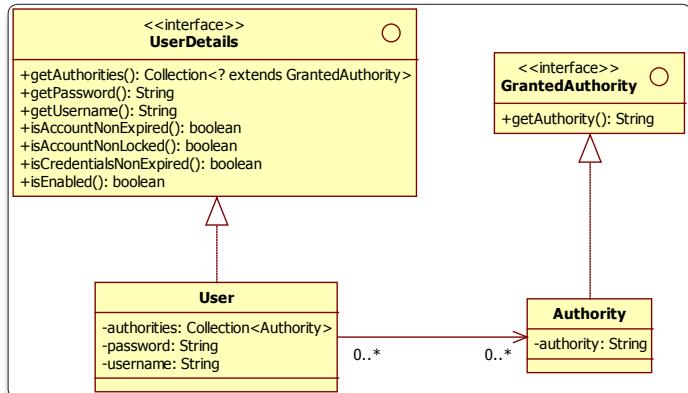


Figura 4. Diagrama de classes

Listagem 7. Código do Authentication Provider customizado.

```
01 package br.com.javamagazine.service;
02
03 import java.util.ArrayList;
04
05 import org.springframework.security.core.userdetails.UserDetails;
06 import org.springframework.security.core.userdetails.UserDetailsService;
07 import org.springframework.security.core.userdetails.UsernameNotFoundException;
08 import org.springframework.stereotype.Service;
09
10 import br.com.javamagazine.model.Authority;
11 import br.com.javamagazine.model.User;
12
13 @Service("segurancaService")
14 public class SegurancaServiceImpl implements UserDetailsService {
15
16     @Override
17     public UserDetails loadUserByUsername(String username)
18             throws UsernameNotFoundException {
19         if(username == null) {
20             throw new UsernameNotFoundException(username);
21         }
22         Authority authority = new Authority();
23         if(username.equals("admin")) {
24             authority.setAuthority("ROLE_ADMIN");
25         } else {
26             authority.setAuthority("ROLE_USER");
27         }
28         User user = new User();
29         user.setUsername(username);
30         user.setPassword(username);
31         user.setAuthorities(new ArrayList<Authority>());
32         user.getAuthorities().add(authority);
33         return user;
34     }
35
36 }
```

A chave privada se encontra definida na linha 20. Em projetos reais seria recomendável manter essa chave em um arquivo de configuração e gerá-la utilizando as melhores práticas (idealmente, ela é formada por caracteres aleatórios, com tamanho mínimo de 256 bits). Ao analisar a classe **JwtUtils**, são de especial interesse os métodos **generateToken()** e **parseToken()**, responsáveis por gerar e realizar o parsing dos tokens, respectivamente.

O método **generateToken()** utiliza a biblioteca **JWT** para gerar um token (linhas 28 a 32). Em sua construção, optamos por utilizar as claims reservadas **iss** (linha 29), **sub** (linha 30) e **exp** (linha 31). A claim **exp**, em especial, garante que a validade de nosso token é de apenas 1 hora após sua criação. O grande diferencial de nosso

token, entretanto, se encontra em nossa claim privada **usr** (linha 28), na qual inserimos a serialização (realizada pelo Jackson nas linhas 25 e 26) de um objeto do tipo **User**, contendo todas as características e autorizações do usuário autenticado. Essa claim permite transmitirmos, por meio do token, todas as informações necessárias para autenticar o usuário em requisições futuras, sem necessidade de qualquer acesso a um Authentication Manager e, consequentemente, a um eventual banco de dados, refletindo o caráter autocontido de um JWT. Finalmente, geramos a assinatura com o algoritmo HMAC SHA256 e a compactamos em base64 (linha 32).

Listagem 8. Código da classe utilitária JwtUtils.

```

01 package br.com.javamagazine.security;
02
03 import io.jsonwebtoken.JwtException;
04 import io.jsonwebtoken.SignatureAlgorithm;
05
06 import java.io.IOException;
07 import java.util.Date;
08
09 import org.springframework.security.core.userdetails.UserDetails;
10
11 import br.com.javamagazine.model.User;
12
13 import com.fasterxml.jackson.core.JsonParseException;
14 import com.fasterxml.jackson.core.JsonProcessingException;
15 import com.fasterxml.jackson.databind.JsonMappingException;
16 import com.fasterxml.jackson.databind.ObjectMapper;
17
18 public class JwtUtils {
19
20     private static final String secretKey = "j4v4_s3cr3t";
21
22     public static String generateToken(User user)
23         throws JsonProcessingException {
24         final Long hora = 1000L * 60L * 60L;
25         ObjectMapper mapper = new ObjectMapper();
26         String userJson = mapper.writeValueAsString(user);
27         Date agora = new Date();
28         return Jwts.builder().claim("usr", userJson)
29             .setIssuer("br.com.javamagazine")
30             .setSubject(user.getUsername())
31             .setExpiration(new Date(agora.getTime() + hora))
32             .signWith(SignatureAlgorithm.HS256, secretKey).compact();
33     }
34
35     public static UserDetails parseToken(String token)
36         throws JsonParseException, JsonMappingException, IOException {
37         ObjectMapper mapper = new ObjectMapper();
38         String userJson = Jwts.parser().setSigningKey(secretKey)
39             .parseClaimsJws(token).getBody().get("usr", String.class);
40         return mapper.readValue(userJson, User.class);
41     }
42
43 }
```

O método **parseToken()**, por sua vez, utiliza a biblioteca JJJWT para realizar o procedimento inverso, ou seja, validar um token e recuperar as informações do usuário de seu corpo (payload). Para tanto, ele faz uso da chave secreta para validar o token (linha 38). Após isso, ele recupera a informação desejada informando a chave da claim (**usr**) e seu tipo (**String**). Durante esse processo, exceptions podem ser geradas se o token for incorreto ou se a data/hora atual for superior à de sua expiração. Finalmente, utilizamos o Jackson para desserializar o JSON recuperado em um objeto do tipo **User** (linha 40).

Nota

A JJWT é uma biblioteca Java que utiliza uma interface fluida para construção e parsing de tokens. Segundo o site oficial do padrão, ela é capaz de validar todas as claims reservadas e suportar todos os algoritmos de criptografia disponíveis. Dentro de sua estrutura para encadeamento de métodos, os processos de construção e parsing são iniciados pela chamada aos métodos `builder()` e `parser()`, respectivamente.

Login

Devido ao fato de necessitarmos gerar e enviar um JWT quando ocorre uma autenticação bem-sucedida, iremos construir um mecanismo de login customizado, disponibilizado por meio da URL `"/login"`, conforme apresenta a **Listagem 9**. Esse serviço deve ser de acesso público, sem qualquer segurança, para possibilitar o acesso de usuários não autenticados.

Listagem 9. Código do mecanismo de login.

```

01 package br.com.javamagazine.controller;
02
03 import javax.servlet.http.HttpServletResponse;
04
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.beans.factory.annotation.Qualifier;
07 import org.springframework.security.authentication.AuthenticationManager;
08 import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
09 import org.springframework.security.core.Authentication;
10 import org.springframework.security.core.userdetails.UserDetails;
11 import org.springframework.web.bind.annotation.RequestBody;
12 import org.springframework.web.bind.annotation.RequestMapping;
13 import org.springframework.web.bind.annotation.RestController;
14
15 import br.com.javamagazine.model.User;
16 import br.com.javamagazine.security.JwtUtils;
17 import br.com.javamagazine.security.Login;
18
19 import com.fasterxml.jackson.core.JsonProcessingException;
20
21 @RestController
22 @RequestMapping(value = "/login")
23 public class LoginController {
24
25     @Autowired
26     @Qualifier("authenticationManager")
27     private AuthenticationManager auth;
28
29     public void setAuth(AuthenticationManager auth) {
30         this.auth = auth;
31     }
32
33     @RequestMapping(path = "")
34     public UserDetails login(@RequestBody Login login,
35         HttpServletResponse response) throws JsonProcessingException {
36         Authentication credentials = new UsernamePasswordAuthenticationToken(
37             login.getUsername(), login.getPassword());
38         User user = (User) auth.authenticate(credentials).getPrincipal();
39         user.setPassword(null);
40         response.setHeader("Token", JwtUtils.generateToken(user));
41         return user;
42     }
43
44 }
```

Como o *Authentication Manager* do Spring Security implementa toda a lógica de autenticação de usuários, vamos utilizá-lo (injeção nas linhas 25 e 26), ao invés de reescrever todo o processo.

Nosso serviço de login (linhas 33 a 42) receberá um objeto do tipo **Login** (linha 34) contendo o `username` e o `password` e repassará

esses dados para o Authentication Manager para autenticação. Em caso de sucesso, será recuperado do **Principal** um objeto do tipo **User**, contendo as características e autorizações do usuário autenticado (linha 38). Com base nessas informações recuperadas, com exceção do password, geramos um JWT e o carregamos no header **Token** da resposta (linhas 39 e 40).

Adicionalmente, retornamos também o objeto **User** (linha 41) para utilização no front-end (mostrar o nome do usuário logado, por exemplo).

Nota

No contexto de segurança, Principal é um objeto identificador que representa o usuário autenticado. Em Java, corresponde a uma implementação da interface `java.security.Principal`.

Filtro JWT

Precisamos agora incluir em nosso projeto um elemento capaz de utilizar o JWT, enviado em cada requisição, para realizar o processo de autenticação, liberando acesso aos recursos protegidos. Para isso, criaremos um filtro, **JwtAuthenticationFilter**, apresentado na **Listagem 10**. Nele, nós tentamos recuperar um JWT do header **Authorization** (linhas 27 e 28).

Caso ele exista, realizamos a validação e o parsing (linha 30), criamos um objeto **Authentication** com as informações recuperadas (linhas 31 a 35) e o carregamos no contexto de segurança (linha 36), efetivamente autenticando o usuário para essa requisição. Caso alguma exception ocorra durante a validação do JWT (validade expirada, assinatura inválida, etc.), retornamos um erro 401 - UNAUTHORIZED.

Nosso filtro aceita headers **Authorization** no formato OAuth 2.0 (com prefixo “Bearer”) ou que contenham apenas o JWT. Por esse motivo, na linha 30, eliminamos o texto “Bearer”, se existir, antes de processar o token. Podemos verificar também que utilizamos apenas as informações contidas no JWT, sem necessidade de acessar qualquer outro recurso para realizar a autenticação do usuário.

Após esse filtro, o Spring Security reconhecerá o usuário como autenticado, pulando todos os filtros de autenticação e validando somente se ele possui as autorizações necessárias para acessar os recursos desejados.

Nota

Por conta de ser autocontido, um JWT dispensa o uso de mecanismos tradicionais (Authentication Manager, Authentication Provider, etc.) para realização da autenticação. Isso pode gerar problemas em sistemas com modificações constantes de autorizações e, por esse motivo, é importante definir uma política de expiração adequada, capaz de garantir que a vida de um token possua uma duração mínima necessária para realizar seu propósito.

API privada

Nada do que criamos até agora faria sentido se não existissem serviços a proteger, com acesso privado, em nossa API. A classe **TesteController**, exposta na **Listagem 11**, apresenta dois desses serviços.

Listagem 10. Código do filtro JWT, JwtAuthenticationFilter.

```
01 package br.com.javamagazine.security;
02
03 import java.io.IOException;
04
05 import javax.servlet.FilterChain;
06 import javax.servlet.ServletException;
07 import javax.servlet.ServletRequest;
08 import javax.servlet.ServletResponse;
09 import javax.servlet.http.HttpServlet;
10 import javax.servlet.http.HttpServletRequest;
11
12 import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
13 import org.springframework.security.core.Authentication;
14 import org.springframework.security.core.context.SecurityContextHolder;
15 import org.springframework.security.core.userdetails.UserDetails;
16 import org.springframework.web.filter.GenericFilterBean;
17
18 public class JwtAuthenticationFilter extends GenericFilterBean {
19
20     private String tokenHeader = "Authorization";
21
22     @Override
23     public void doFilter(ServletRequest request, ServletResponse response,
24             FilterChain chain)
25             throws IOException, ServletException {
26
27         try {
28             HttpServletRequest servletRequest = (HttpServletRequest) request;
29             String authorization = servletRequest.getHeader(tokenHeader);
30             if (authorization != null) {
31                 UserDetails user = JwtUtils.parseToken(authorization.replaceAll(
32                     "Bearer", ""));
33                 Authentication credentials =
34                     new UsernamePasswordAuthenticationToken(
35                         user.getUsername(),
36                         user.getPassword(),
37                         user.getAuthorities());
38                 SecurityContextHolder.getContext().setAuthentication(credentials);
39             }
40             chain.doFilter(request, response);
41         } catch (Throwable t) {
42             HttpServletResponse servletResponse = (HttpServletResponse) response;
43             servletResponse.sendError(HttpServletResponse.SC_UNAUTHORIZED,
44             t.getMessage());
45         }
46     }
47 }
```

Listagem 11. Código exemplo da API privada.

```
01 package br.com.javamagazine.controller;
02
03 import org.springframework.security.access.prepost.PreAuthorize;
04 import org.springframework.web.bind.annotation.PathVariable;
05 import org.springframework.web.bind.annotation.RequestMapping;
06 import org.springframework.web.bind.annotation.RestController;
07
08 @RestController
09 @RequestMapping(value = "/api")
10 public class TesteController {
11
12     @RequestMapping(value = "/hello/{nome}")
13     @PreAuthorize("isAuthenticated()")
14     public String hello(@PathVariable("nome") String nome) {
15         return "Hello " + nome + "!";
16     }
17
18     @RequestMapping(value = "/helloAdmin")
19     @PreAuthorize("hasRole('ROLE_ADMIN')")
20     public String helloAdmin() {
21         return "Hello administrator!";
22     }
23
24 }
```

O primeiro deles, disponível pela URL `/api/hello/{nome}`, utiliza a anotação `@PreAuthorize` para restringir o acesso apenas a usuários autenticados (linha 13) e retorna uma mensagem de boas-vindas direcionada ao nome enviado como parâmetro na URL (mapeado com a ajuda da anotação `@PathVariable` na linha 14). O segundo serviço, disponível pela URL `/api/helloAdmin`, utiliza a anotação `@PreAuthorize` para restringir o acesso apenas a usuários autenticados que possuam a autorização `ROLE_ADMIN` (linha 19) e retorna uma mensagem de boas-vindas direcionada ao administrador.

Avaliação

Para avaliar nosso sistema, precisaremos enviar algumas requisições HTTP para os recursos privados da API. Existem diversas ferramentas que possibilitam isso, mas vamos utilizar o add-on Postman para o navegador Google Chrome, por sua facilidade de instalação e uso.

Os casos de teste apresentados nas seções a seguir visam garantir que nossa API possui as funcionalidades mínimas de segurança necessárias. Para todos os casos, assume-se que o nome da aplicação é `SpringRestSecurityJwt` e o servidor web escolhido utiliza a porta local 8080.

Acesso não autorizado

O primeiro teste a realizar consiste em tentar acessar um serviço sem qualquer autenticação. A Tabela 1 apresenta a configuração da requisição. O resultado esperado para esse teste é um erro de acesso não autorizado (401), pois, fora o login, nenhum serviço de nossa API aceita acessos não autenticados. É importante notar que o servidor utilizado utiliza a porta 8080. Portanto, isso deve ser alterado de acordo com o ambiente de desenvolvimento disponível.

Como esperado, essa requisição recebe um erro 401 – UNAUTHORIZED, conforme verificado na Figura 5, que apresenta a tela do Postman contendo a requisição e a resposta.

Login

Nosso próximo teste consiste em realizar um login utilizando nosso serviço customizado. A Tabela 2 apresenta a configuração dessa requisição, onde passamos os parâmetros no corpo (body) como um JSON e, por esse motivo, precisamos informar um header `Content-Type` com valor `application/json`. O resultado esperado consiste na recepção de um JWT correspondente ao usuário informado.

Essa requisição é processada com sucesso (Status 200 – OK), como podemos visualizar na Figura 6. O elemento mais importante da resposta, contudo, é o header `Token`, que contém nosso JWT.

Ao decodificarmos o segundo trecho do JWT (payload), obtemos o JSON apresentado na Listagem 12, onde podemos visualizar nossa claim privada `usr` e as claims registradas `iss`, `sub` e `exp`.

Acesso com JWT

Finalmente, realizaremos um acesso idêntico ao primeiro teste, mas utilizando o JWT como meio de autenticação.

A Tabela 3 apresenta a configuração dessa requisição. Para reproduzir o teste, o token deve ser alterado pelo recebido no processo de login.

| Tipo de requisição | GET |
|--------------------|--|
| URL | <code>http://localhost:8080/SpringRestSecurityJwt/api/hello/teste</code> |

Tabela 1. Requisição sem autenticação

| Tipo de requisição | POST |
|--------------------|---|
| URL | <code>http://localhost:8080/SpringRestSecurityJwt/login</code> |
| Headers | <code>Content-Type: application/json</code> |
| Body | <pre>{ "username": "admin", "password": "admin" }</pre> |

Tabela 2. Login para um usuário admin

| Tipo de requisição | GET |
|--------------------|---|
| URL | <code>http://localhost:8080/SpringRestSecurityJwt/api/hello/teste</code> |
| Headers | <code>Authorization: Bearer eyJhbGciOiJIUzI1NiJ9eyJ1c3IiOjJX-CJ1c2VybmFtZVwiOlwiYWRtaW5cixlnBhc3N3b3JkXCl6bnVsbCxclmF1dGhvcmloaWVzXCl6W3tclmF1dGhvcmloevWiO-lwiUk9M</code> <code>RV9BRE1JTiwifV19liwiaXNzIjoiYnluY29tLmphdmFtYWdhemuZlslslnN1Yil6lmFkbWluliwiZhwljoxNDc3NzxMTYwfQ.Xt_3mGTjHe7sX-SVc6E7KjfFA70ErWnPLYcsynUX4xw</code> |

Tabela 3. Requisição com JWT

Listagem 12. Payload decodificado do JWT.

```
{
  "usr": "{\"username\": \"admin\", \"password\": null, \"authorities\": [{\"authority\": \"ROLE_ADMIN\"}]}",
  "iss": "br.com.javamagazine",
  "sub": "admin",
  "exp": 1477781160
}
```

Ao contrário de nosso primeiro teste, temos um header `Authorization`, onde passamos o JWT recebido após a autenticação, prefixado por “Bearer”. Dessa vez, esperamos que a requisição seja processada com sucesso (Status 200 – Ok) e que recebamos uma mensagem de boas-vindas.

Confirmando a expectativa, a requisição é processada com sucesso (Status 200 – OK), o que pode ser observado na Figura 7. Isso indica que nosso filtro funciona e é capaz de autenticar o usuário apenas com o token.

Acesso com JWT inválido

Como um teste adicional, tentaremos a mesma requisição com um JWT modificado, mantendo a mesma assinatura, mas com um corpo diferente. A Tabela 4 apresenta a configuração dessa requisição.

Web services RESTful: Como adicionar segurança com JWT

| Tipo de requisição | GET |
|--------------------|---|
| URL | http://localhost:8080/SpringRestSecurityJwt/api/hello/teste |
| Headers | Authorization: Bearer eyJhbGciOiJIUzI1NiJ9eyJc1IiOjI7XCJ1c2VybmtFtZVwiOlwiYWRtaW5clixlnBhc3N3b3JkXCI6bnVsbcxclmF1dGhvcmloaWVzXCi6W3tclmF1dGhvcmloevwiOlwiUk9MRV9BRE1JTlwiF19liwiaXNzjoiYnluY29tLmphdmFtYVdhemluZSlslN1Yil6lmFkbWlwiZXhwljoxNDc3NzgxMTYwfQ.Xt_3mGTjHe7sX-SVc6E7KjfFA70ErWnPLYcsynUX4xw |

Tabela 4. Requisição com JWT inválido

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/SpringRestSecurityJwt/api/hello/teste`. The Authorization header contains an invalid JWT token. The response status is 401 Unauthorized, and the response body is an HTML error page from Apache Tomcat 7.0.53, indicating an unauthorized access attempt.

Figura 5. Requisição não autenticada

The screenshot shows the Postman interface with a POST request to `http://localhost:8080/SpringRestSecurityJwt/login`. The Content-Type header is set to application/json. The response status is 200 OK, and the response body contains a JSON object with user information and a token.

Figura 6. Requisição de login

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/SpringRestSecurityJwt/api/hello/teste`. The Authorization header contains a valid JWT token. The response status is 200 OK, and the response body is "Hello teste!".

Figura 7. Requisição com JWT

Para reproduzir o teste é necessário alterar o payload do token recebido no processo de login. O resultado esperado para esse teste é um erro de acesso não autorizado (401).

Essa requisição recebe um erro 401 – UNAUTHORIZED, como verificado na **Figura 8**, mostrando que nosso filtro valida o conteúdo, utilizando para isso a assinatura e a chave privada. É importante notar que a mensagem de erro que acompanha o erro indica que a assinatura do JWT não bate com a assinatura computada localmente, no servidor.

Acesso com JWT expirado

Neste outro teste adicional, tentaremos a mesma requisição com o JWT válido, mas após seu tempo de expiração (conforme configurado em seu corpo). Nesse caso, a requisição segue a mesma configuração previamente apresentada na **Tabela 3**, mas o resultado esperado dessa vez é um erro de acesso não autorizado (401). Ao executarmos a requisição, recebemos um erro 401 – UNAUTHORIZED, como verificado na **Figura 9**, mostrando que nosso filtro também valida a data de expiração do token, conforme a mensagem que acompanha o erro.

Acesso com JWT contendo autorizações insuficientes

Por fim, testaremos um caso onde o JWT possui autorizações insuficientes para o serviço acessado. Para tanto, geraremos um novo token para um usuário de nome **teste**, segundo os parâmetros indicados na **Tabela 5**.

O token gerado por essa requisição, presente no header **Token**, é o seguinte:

```
eyJhbGciOiJIUzI1NiJ9eyJc1IiOjI7XCJ1c2VybmtFtZVwiOlwidGVzdGVclixlnBhc3N3b-3JkXCI6bnVsbcxclmF1dGhvcmloaWVzXCi6W3tclmF1dGhvcmloevwiOlwiUk9MRV9VU-0VSXC9XX0iLCpc3MiOjici5jb20uamF2YW1hZ2F6aW5liiwc3ViljoidGVzdGUiLCJ-1eHAiOjE0Nzc4MjgxNTI9.2gGMeLIOq9UEv7ghoS80wNuf2plfvdzYzkvZEeGGuYk
```

| Tipo de requisição | POST |
|--------------------|---|
| URL | http://localhost:8080/SpringRestSecurityJwt/login |
| Headers | Content-Type: application/json |
| Body | { "username":"teste", "password":"teste" } |

Tabela 5. Login do usuário teste

Além disso, o corpo (body) da resposta contém um JSON com as características do usuário **teste**, como visualizado na **Listagem 13**.

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/SpringRestSecurityJwt/api/hello/teste`. The Authorization header is set to "Bearer eyJhbGciOiJIUzI1NiJ9eyJcIiOj7XClc2". The response status is 401 Unauthorized, and the response body contains an HTML error page with the message: "HTTP Status 401 - JWT signature does not match locally computed signature. JWT validity cannot be asserted and should not be trusted.".

Figura 8. Requisição com JWT inválido

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/SpringRestSecurityJwt/api/hello/teste`. The Authorization header is set to "Bearer eyJhbGciOiJIUzI1NiJ9eyJcIiOj7XClc2". The response status is 401 Unauthorized, and the response body contains an HTML error page with the message: "HTTP Status 401 - JWT expired at 2016-10-29T20:46:00-0200. Current time: 2016-10-29T20:48:27-0200".

Figura 9. Requisição com JWT expirado

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/SpringRestSecurityJwt/api/helloAdmin`. The Authorization header is set to "Bearer eyJhbGciOiJIUzI1NiJ9eyJcIiOj7XClc2". The response status is 403 Forbidden, and the response body contains an HTML error page with the message: "HTTP Status 403 - Access is denied".

Figura 10. Requisição com JWT contendo autorizações insuficientes

Listagem 13. JSON recebido no corpo da resposta.

```
{
  "username": "teste",
  "password": null,
  "authorities": [
    {
      "authority": "ROLE USER"
    }
  ]
}
```

| Tipo de requisição | GET |
|--------------------|---|
| URL | <code>http://localhost:8080/SpringRestSecurityJwt/api/helloAdmin</code> |
| Headers | Authorization: Bearer eyJhbGciOiJIUzI1NiJ9eyJcIiOj7XClc2 |

Tabela 6. Requisição com autorizações insuficientes

Utilizaremos esse token para acessar o serviço `helloAdmin`, com os parâmetros listados na **Tabela 6**. O resultado esperado é um erro 403 – FORBIDDEN, indicando que o usuário não possui a autorização necessária para acessar o recurso.

A **Figura 10** apresenta o resultado da requisição, o qual condiz com o esperado.

Este artigo apresentou uma abordagem simples e funcional, baseada em Spring Security, de um mecanismo de Token Based Authentication com JWT. Como indicado, entretanto, a implementação apresentada não busca completude e, por esse motivo, não se preocupa com a persistência dos usuários e suas autorizações ou com problemas de acesso, como CORS (*Cross-Origin Resource Sharing*).

A persistência pode ser realizada com o auxílio de outros frameworks, como o Spring Data JPA, e um filtro CORS também pode ser facilmente criado com o próprio framework Spring Security, utilizado neste artigo.

Por fim, outro ponto que merece atenção, agora que conhecemos JWT, é a possibilidade de criar um mecanismo de autenticação/autorização com OAuth 2.0. O Spring oferece uma solução completa para isso, inclusive com suporte nativo a JWT, com o framework Spring Security OAuth.

Autor



Emanuel Mineda Carneiro

emanuel.mineda@fatec.sp.gov.br

Professor na FATEC São José dos Campos. Trabalha com desenvolvimento de software desde 2005. Bacharel em Ciência da Computação pela Universidade Federal de Itajubá (UNIFEI). Mestre em Ciências pelo Programa de Pós-Graduação em Engenharia Eletrônica e Computação do Instituto Tecnológico de Aeronáutica (ITA).



Web services: testes funcionais e automatizados com Cucumber

Como simplificar os testes de web services com Java e Cucumber eliminando as etapas de parser de XML

Há muito tempo a utilização de web services vem aumentando gradativamente. Nesse cenário, atualmente, quase tudo é feito com o auxílio de web services. A preferência pela utilização de serviços deve-se, em grande parte, ao baixo acoplamento, manutenibilidade e, o mais importante, ao reuso e à rapidez no desenvolvimento que a adoção de web services propicia às organizações.

Em relação à análise e desenvolvimento de serviços, as organizações já atingiram um nível de maturidade bastante elevado. Além disso, a abordagem para esses temas na literatura técnica, como análise e projeto de serviços e criação de inventário de serviços, é bastante difundida. Um desafio, no entanto, surge em meio a essas já maduras técnicas de desenvolvimento orientadas a serviços: os testes.

Testar serviços pode ser uma tarefa complexa devido às particularidades que envolvem a criação de web services. Um serviço pode gerar dados em diferentes formatos, como XML e JSON, pode ter um contrato especificando a estrutura desses dados e ainda há a preocupação com o conteúdo gerado. Outro atenuante das dificuldades de se testar web services é a gama de tecnologias que podem ser - e na maioria das vezes são - utilizadas para sua implementação. Enfim, são muitas características técnicas e de negócio a serem levadas em consideração para realizar os testes de serviços.

Fique por dentro

A utilização de web services baseados em XML é bastante comum, e seu uso continua aumentando, especialmente por conta dos produtos e soluções SOA, que adotaram o XML como padrão. Executar testes nesses cenários é uma tarefa bastante complexa, ainda mais quando os testes se concentram na validação de dados de payloads XML. Em muitos desses cenários os testes de dados em payloads exigem o mapeamento do XML para POJOs a fim de recuperar os dados a serem validados, tornando o desenvolvimento mais lento e complexo.

Neste artigo será mostrado como a utilização do Cucumber pode melhorar e incrementar as possibilidades de testes de payloads XML em ambientes com uso extensivo de web services. A ideia central é criar uma infraestrutura para testes de forma a unir as facilidades que o Cucumber possibilita, como automatização de testes e criação de cenários, com as tecnologias relacionadas à linguagem XML, como XPath, de forma que as validações de payloads sejam feitas diretamente nos payloads XML, ao invés de implementar o mapeamento entre o payload e o objeto para a validação dos dados. Dessa forma, o desenvolvimento de testes se torna muito mais simples e rápido.

Testes manuais em web services são bastante custosos devido a tantas variáveis nesses cenários, tais como o comportamento de um ESB, que deve ser simulado para garantir um teste completo de uma mensagem até ela atingir seu destino final.

Nesses casos, a automação é o caminho a seguir, pois elimina a recorrência de erros comuns em ambientes de alta complexidade. Ainda assim, dependendo do uso que se faz dos web services e de como está projetada a arquitetura sob a qual os mesmos serão utilizados, esses testes podem se tornar muito complexos. Ademais, testar os clientes desses serviços também não foge à regra da complexidade, e podem, inclusive, ter um nível de dificuldade ainda maior, pois deve-se se preocupar com o mapeamento de todos os cenários funcionais possíveis para um serviço exposto.

Em sistemas mais complexos, onde todo o processamento das informações depende de web services, como orquestradores de serviços, os testes dos clientes se mostram mais difíceis de acordo com as tecnologias utilizadas. Imagine um mecanismo em um orquestrador de serviço que lê um payload em XML e o transforma com um XSL para enviar o payload de saída para algum serviço. A **Figura 1** ilustra a arquitetura desse sistema e a utilização de web services em seu processamento.

Nessa arquitetura, web services são imprescindíveis para o seu funcionamento pois, como são utilizados de diversas formas por meio de diferentes tecnologias (línguagens de programação), suas características de independência de plataforma e linguagem fazem muito mais sentido para implementação. Além disso, conceitualmente falando, do ponto de vista da arquitetura SOA, um serviço é uma função computacional disponibilizada por outro sistema de forma independente dos clientes. Na arquitetura apresentada na **Figura 1**, note como são utilizadas tecnologias de parsers para a geração de payloads em XML, diferentes protocolos de comunicação e vários tipos de modelos de dados. Tudo isso torna a arquitetura bastante complexa.

Este artigo aborda a aplicação de BDD e especificações funcionais ao processamento de dados gerados por meio de transformações (usualmente XSL) para serem consumidos por web services. Considerando essa situação, um dos principais problemas está relacionado à integridade dos dados. Como testar a integridade? Além disso, qual seria a melhor abordagem para escrever os testes? Deve-se fazer o parser dos XMLs de saída e compará-los com objetos ou criar arquivos XML de saída esperados? É muito difícil eliminar toda a complexidade presente em um cenário como esse. Contudo, podemos amenizar as dificuldades por meio da utilização de frameworks de testes automatizados.

Os próximos tópicos deste artigo abordarão como o Cucumber pode auxiliar na implementação da abordagem BDD, além de prover um ambiente pronto para automatização dos testes.

Cucumber

O Cucumber é uma ferramenta de testes construída sob o conceito de BDD que possui distribuições para diferentes linguagens e plataformas, tais como Ruby, JavaScript, .NET, PHP e, é claro, Java. Em relação à plataforma Java, a implementação do Cucumber é a Cucumber-JVM, que suporta as linguagens mais populares da JVM, como Java, Closure, Jython, Groovy, JRuby e Scala.

Inerentes a todas as versões do Cucumber estão presentes ideias próprias do framework, que são voltadas para a abordagem BDD (vide **BOX 1**). Esses conceitos de BDD, analisados a seguir, são uma referência para as diferentes versões do Cucumber e descrevem o modelo conceitual da ferramenta.

BOX 1. O que é BDD?

O sucesso ao término de um projeto de software depende, em grande parte, de um fator simples e inerente aos humanos, a comunicação. É por meio da comunicação que as necessidades e requisitos sistêmicos passam dos stakeholders para o time de desenvolvimento. Quando há falha no entendimento dessas necessidades e requisitos, o resultado é um produto de software funcional, porém inconsistente com os requisitos de negócios e com o que era esperado pelos stakeholders. A metodologia BDD (Behavior-Driven Development) surgiu para suprir essas falhas na comunicação de requisitos de negócios. A sua abordagem é baseada na orientação do desenvolvimento por meio de comportamentos que o sistema deve ter, facilitando assim a comunicação de analistas de negócio e desenvolvedores. Basicamente, o sistema passa a ser especificado em termos de comportamento, ou seja, qual o comportamento o sistema deve ter de acordo com um stakeholder.

As práticas do BDD incluem envolver os stakeholders, descrever em forma de exemplos o comportamento do sistema ou de partes dele, automatizar os exemplos fornecidos para agilizar testes de regressão, esclarecer responsabilidades, possibilitar o debate sobre funcionalidades do software e usar simuladores de testes (mocks, stubs, fakes, dummies, spies e afins) visando apoiar o desenvolvimento das funcionalidades do software.

Quando se avança no estudo dessa técnica em paralelo com o uso prático em conjunto com alguma ferramenta, como o Cucumber, percebe-se que a ferramenta atua como um contrato para garantir que a técnica de BDD será cumprida de acordo com seu propósito.

Gherkin

Gherkin é a base do Cucumber e pode ser considerada uma DRL (*Domain Specific Language*) para BDD, tendo como base a língua inglesa (ou outro idioma) e a definição de uma estrutura que

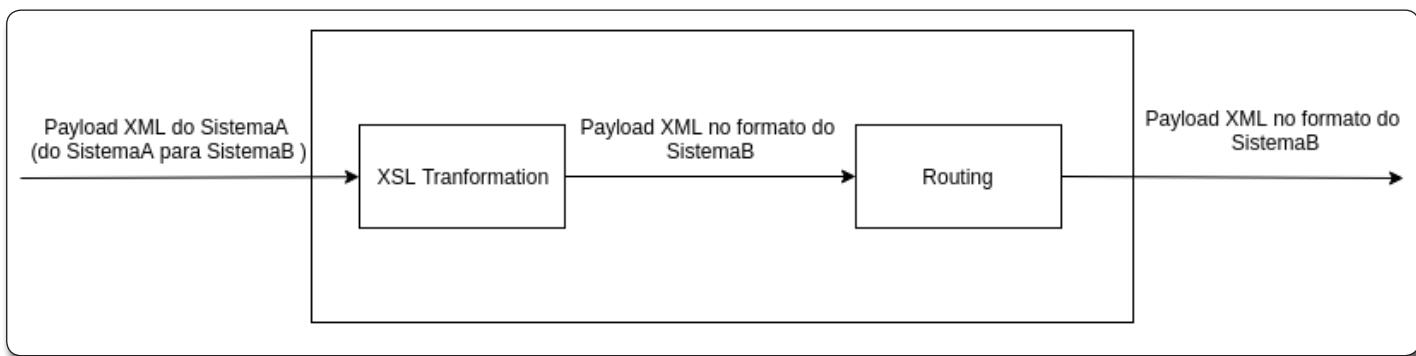


Figura 1. Exemplo de como um orquestrador de serviços funciona

Web services: testes funcionais e automatizados com Cucumber

especifica como o grupo de palavras dessa DRL pode ser utilizado. Gherkin foi projetada para ser facilmente aprendida por pessoas sem conhecimento técnico a fim de que essas pudessem escrever cenários de testes em uma linguagem próxima à natural. Além disso, a sua estrutura possibilita a descrição de regras de negócio nos mais diferentes domínios.

Essa linguagem inclui, entre outras, as seguintes palavras-chave: *Given*, *When*, *Then*, *And*, *But*, além de caracteres especiais. Outra característica importante é o fato do Gherkin ter como propósito ser uma documentação de especificações funcionais executáveis. Esse objetivo é atingido com o código Gherkin sendo escrito em arquivos com extensão *.feature*, que são executados pelo Cucumber durante os testes. A **Listagem 1** mostra um exemplo de arquivo *.feature* contendo uma especificação funcional para um cenário hipotético de ativação de linhas em uma telecom. Note como esse cenário apresenta uma leitura simples de ser compreendida.

Listagem 1. Exemplo de arquivo *.feature* com especificação funcional.

Feature: Line activation / deactivation

Scenario: Activate line

Given I have an order to activate a line with the 1198767653 number
When the order is received
Then the service invoked must be SVC_ACTIVATE_LINE
And the input to the legacy system should contains /data/service = 'ACTIVATE_LINE'

Em Gherkin, cada linha que não esteja vazia deve começar com uma palavra-chave seguida de algum texto que reflete o negócio sob o qual aquela especificação diz respeito. A **Tabela 1** mostra as principais palavras-chave da Gherkin e o que elas querem dizer. Os próximos tópicos irão detalhar o conteúdo dessa tabela, expondo as características dessas palavras e suas regras de uso.

Feature

O Cucumber define arquivos próprios que, conceitualmente, devem conter características particulares do sistema ou aspectos de determinadas características.

Na prática, são nesses arquivos que devem ser implementadas as especificações funcionais executáveis utilizando Gherkin.

Um arquivo *.feature* deve possuir a descrição de uma feature, ou seja, uma característica do sistema e todos os cenários relacionados que compõem essa determinada feature. Devido a esses conceitos relacionados às features, pode-se entender um arquivo desse tipo como um container de cenários.

A **Listagem 2** apresenta um exemplo de definição de uma feature. Note que a palavra-chave **Feature** é seguida pelo seu nome, que tem como objetivo fornecer um termo ou mais termos que expliquem rapidamente os aspectos tratados de um cenário.

Listagem 2. Exemplo de arquivo *.feature* e sua descrição.

Feature: Line migrations

The operators' system have to be able to change
the customer's linetype between a set of options and plans.

Rules:

- Clients can't to change to the same type of plan he was before

Descriptions

Algumas partes dos documentos Gherkin não precisam começar com palavras-chave. Ainda na **Listagem 2**, note que há uma descrição contendo duas linhas logo após a definição do nome da feature. Após as palavras-chave *Feature*, *Scenario*, *Scenario Outline* ou *Example*, pode-se escrever qualquer conteúdo a fim de documentar com um pouco mais de detalhe os aspectos tratados naquele ponto.

Esse é o conceito de *description* no Cucumber, algo altamente recomendado até pela natureza do arquivo *.feature*, cujo objetivo é ser uma especificação executável, mas, também, clara e de simples compreensão. O ideal é utilizar algumas linhas para descrever o elemento em questão. Cerca de três linhas é o suficiente para complementar e prover algum contexto para a especificação executável.

Scenarios

Um cenário (*scenario*) ilustra uma regra de negócio e é composto por uma sequência de passos (*steps*).

| Feature | Descreve uma característica única e de alto nível do sistema. Utilizada para identificar as características e agrupar cenários de teste. |
|-----------------------------|---|
| Scenario | Um cenário é um exemplo concreto que ilustra uma regra de negócio. |
| Given, When, Then, And, But | Essas palavras representam os steps do cenário de teste. Um step é a unidade básica que forma um cenário por meio de uma proposição. |
| Background | Define steps básicos para serem executados em um cenário. |
| Scenario Outline | Define templates para cenários de testes. Podemos entender templates como se fossem métodos na linguagem Java, já que realizam sempre as mesmas operações, tendo como diferença apenas os parâmetros. |
| Examples | Utilizada para definir exemplos para os templates especificados por meio de um Scenario Outline. |
| "" | As aspas indicam strings. |
| | O pipe é empregado para criar estruturas de dados em tabelas, de forma a separar o conteúdo em linhas e colunas para ser utilizado como parâmetro em um Scenario Outline. |
| @ | O caractere @ é utilizado para definir uma tag, que, por sua vez, agrupa cenários. |
| # | O caractere # é utilizado para indicar comentários. |

Tabela 1. Palavras-chave da linguagem Gherkin

Apesar de não haver limitação, o recomendado, por motivos de facilidade e coesão, é descrevê-lo utilizando entre três e cinco etapas.

Quando os cenários se tornam muito longos, eles perdem o poder de expressividade como documentação e especificação. Cenários curtos facilitam a leitura e o entendimento, assim como os métodos de linguagens de programação orientadas a objetos, que são muito mais fáceis de decifrar se forem implementados com poucas linhas.

Enfim, os cenários possuem papel central no Cucumber, pois, além de ilustrar uma regra de negócio, também atuam como testes do código desenvolvido. Todo cenário é uma especificação funcional do sistema e tem como padrão uma descrição do contexto inicial, um evento e a saída esperada. Esse padrão é ilustrado pelas palavras-chave *given*, *when* e *then*.

Steps

Um *step* é a atividade básica de um cenário e geralmente não tem nenhum significado especial ou sentido sem que esteja agrupado a outros steps em cenários. Fazendo uma analogia com algoritmos, o algoritmo com todas as suas instruções seria o cenário, enquanto cada instrução do mesmo seria um *step*. É essa relação, de *step* com instrução e cenário com algoritmo, que viabiliza o conceito de especificação executável no Cucumber, pois possibilita a execução dos cenários em uma sequência de passos.

Sempre que o Cucumber encontra uma das seguintes palavras-chave: *given*, *when*, *then*, *and* e *but*, ele considera o conteúdo subsequente como um *step*. Apesar de ele não diferenciar essas palavras como diferentes tipos de *steps*, cabe aos responsáveis pela definição dos cenários o uso correto e bem especificado, de forma a gerar sequências de *steps* logicamente interconectadas. O correto uso das palavras-chave que definem *steps* é importante para manter a consistência e legibilidade dos cenários como um todo.

Given

O *step Given* é utilizado para descrever um contexto inicial do sistema, como uma cena para o cenário. Geralmente esse contexto é escrito na forma de algo que aconteceu no passado, como no exemplo:

Given client bought an item for \$100

Dado que o cliente comprou um item por R\$100

Quando esse passo é executado, o Cucumber configura o contexto de execução de teste para um determinado estado, criando e instanciando objetos e adicionando dados em um banco de dados de teste, por exemplo. Saiba que podemos definir quantos *steps Given* forem necessários. Quando isso acontecer, a recomendação é que eles sejam separados com as palavras-chave *and* ou *but*, para melhorar a legibilidade.

When

O *step When* é utilizado para descrever um evento ou ação, como uma pessoa interagindo com o sistema ou algum evento externo

lançado por outro sistema. A **Listagem 3** mostra o uso do *When* em um cenário de ativação de linhas em uma telecom. Nesse exemplo ocorre um evento externo, o recebimento de uma ordem.

A recomendação é que se tenha apenas um *step When* por cenário, pois a necessidade de mais steps de evento é um sinal de que o cenário deve ser dividido em cenários menores.

Then

O step *Then* é indicado para descrever a saída esperada do cenário ou seu resultado. Nesse tipo de step é necessário utilizar uma asserção para comparar o resultado que o sistema produziu com o resultado esperado (vide **Listagem 3**).

Background

Muitas vezes é necessária a repetição do mesmo *step Given* em todos os cenários de um arquivo *.feature*. Nesses casos, essas repetições são genéricas para vários cenários e não são essenciais para descrever um cenário de forma individual.

A melhor abordagem para isso é mover todos os steps repetidos entre diferentes cenários para uma seção *background*. Dessa forma, os steps no *background* serão executados para todos os cenários. A **Listagem 3** mostra um exemplo de uso de *background* para agrupar repetições de steps em um único bloco.

Listagem 3. Exemplo de uso de *background* para agrupamento de repetições.

Feature: Line activation / deactivation

Background:

Given the country code is 55

Scenario:

Activate line

Given I have an order to activate a line with the 1198767653 number

When the order is received

Then the service invoked must be SVC_ACTIVATE_LINE

And the input to the legacy system should contains /data/

service = 'ACTIVATE_LINE'

Scenario Outline

Um scenario outline pode ser definido como um template para cenários. Quando se tem muitas variáveis e a estrutura lógica de processamento permanece a mesma, pode-se implementar um template de execução. Assim, é possível ter várias instâncias de cenários que diferem entre si apenas pelos valores das variáveis.

A **Listagem 4** demonstra o uso de dois cenários iguais, sendo que a única diferença são os valores passados como parâmetros. Já a **Listagem 5** apresenta uma abordagem mais concisa para esses mesmos cenários, utilizando scenario outline. Note que as variáveis em um scenario outline são definidas da seguinte forma: `<nomeDaVariavel>`.

As variáveis são substituídas de acordo com a ordem em que aparecem no cenário e trocadas pelos valores da tabela de exemplos definida no final da declaração do scenario outline. Essas tabelas agrupam os dados a serem utilizados nos steps de scenarios outlines, dados esses que devem ter o título de cada coluna

e o nome da variável correspondente no cenário. Com isso, cada uma das linhas dessa tabela (tirando a linha de cabeçalho) gera um novo cenário.

Listagem 4. Exemplo com cenários onde apenas os parâmetros são diferentes.

Feature: Line activation / deactivation

Background:

Given the country code is 55

Scenario: Activate line

Given I have an order to activate a line with the 1198767653 number

When the order is received

Then the service invoked must be SVC_ACTIVATE_LINE

And the input to the legacy system should contains /data/service = 'ACTIVATE_LINE'

Scenario: Deactivate line

Given I have an order to deactivate a line with the 1198767653 number

When the order is received

Then the service invoked must be SVC_CANCEL_LINE

And the input to the legacy system should contains /data/service = 'DEACTIVATE_LINE'

Listagem 5. Exemplo de uso de scenario outline.

Feature: Line activation

Background:

Given the country code is 55

Scenario Outline: Activate line

Given I have an order to <operation> a line with the 1198767653 number

When the order is received

Then the service invoked must be <service>

And the input to the legacy system should contains /data/service = <service>

Examples:

| | | |
|------------------------------|--|--|
| operation service | | |
| activate SVC_ACTIVATE_LINE | | |
| cancel SVC_CANCEL_LINE | | |

Implementação de testes com Cucumber

Os tópicos anteriores forneceram subsídios para o entendimento de conceitos relativos ao Cucumber, testes e BDD. Os tópicos a seguir irão mostrar esses conceitos na prática utilizando o Cucumber para implementar testes em um cenário onde a principal forma de troca de dados é por meio de mensagens em XML.

Contexto

Como exemplo para a implementação de testes automatizados envolvendo mensagens XML, consideremos uma empresa de telecomunicações. A complexidade de empresas de telecomunicações, onde o ambiente tecnológico é formado por diversos sistemas construídos sob plataformas completamente diferentes e que precisam se comunicar, provê um ambiente propício para a adoção de web services e XML. Por isso, esse tópico irá criar uma empresa fictícia de telecom (a TelecoLeco) a fim de fornecer insumos para os testes envolvendo mensagens XML em seu devido contexto.

A TelecoLeco foi fundada há mais de 40 anos e, até hoje, muitos sistemas desenvolvidos no primeiro ano da empresa permanecem

em operação. Esses sistemas foram implementados seguindo outros paradigmas de desenvolvimento, além de utilizarem tecnologias que deixaram de ser populares há bastante tempo. Essas mantêm-se estáveis e os sistemas construídos com elas funcionam bem, porém não recebem mais suporte por parte dos seus fornecedores, isto é, elas não sofrem mais atualizações. Sendo assim, a TelecoLeco preferiu manter esses sistemas do jeito que estão, seguindo a velha máxima: "se está funcionando, é melhor não mexer". Esse tipo de sistema é chamado de sistema legado.

A **Figura 2** ilustra a arquitetura de alto nível da TelecoLeco atual e o relacionamento dos seus principais sistemas. Essa imagem mostra o relacionamento lógico em termos de quais sistemas enviam e/ou recebem mensagens para outros sistemas. Enquanto isso, a **Figura 3** mostra como esses relacionamentos são implementados na TelecoLeco, fazendo uso de um orquestrador de serviços. Sucintamente falando, o orquestrador recebe mensagens XML de um determinado sistema e as envia para outro sistema, realizando, se necessário, transformações nessas mensagens por meio de XSL para que os formatos se adequem ao solicitado pelo sistema que irá receber.

Esse cenário é bastante comum. Basicamente, são princípios de SOA, os quais podem ser encontrados facilmente nos mais diversos segmentos de mercado. Esse tipo de arquitetura faz uso em larga escala de XSLT, o que torna os testes mais complexos do que testar dados em POJOs. Partindo disso, os próximos tópicos mostram, na prática, formas de testar os resultados das transformações feitas por meio de XSLT em ambientes onde é impraticável a conversão XML-POJO para testes.

Modelo de dados

O modelo de dados do contexto de testes apresentado na **Listagem 6** é baseado no processamento de ordens de serviço sobre linhas telefônicas. Nesse cenário, uma ordem de serviço é composta por um ou mais itens, sendo que cada item contém informações a respeito do serviço que deve ser realizado. É interessante notar que, no exemplo apresentado, em nenhum momento são utilizados POJOs para representar a ordem de serviço, eliminando a necessidade de mapeamento de XML para objetos e vice-versa.

Essa listagem mostra um exemplo de ordem de serviço para o cenário hipotético da TelecoLeco. Essa ordem contém três itens, cada um descrevendo um componente da ordem e a ação que deve ser tomada para o mesmo. Por exemplo, o item LINHA possui o *ActionCode default*, representado por um hífen. Isso quer dizer que nenhuma ação deve ser realizada sobre esse item e que ele existe apenas para complementar as demais informações da ordem. Já os itens PLANO possuem, respectivamente, *ActionCode Incluir* e *Excluir*, o que quer dizer que as ações de Incluir e Excluir devem ser executadas sobre eles.

O orquestrador da **Figura 3** recebe essa ordem e a decompõe em diversas requisições para os sistemas legados envolvidos nas ações de incluir e excluir planos. Cada um dos sistemas legados requer que a mensagem enviada para ele seja de um determinado formato. Sendo assim, a ordem enviada será decomposta em

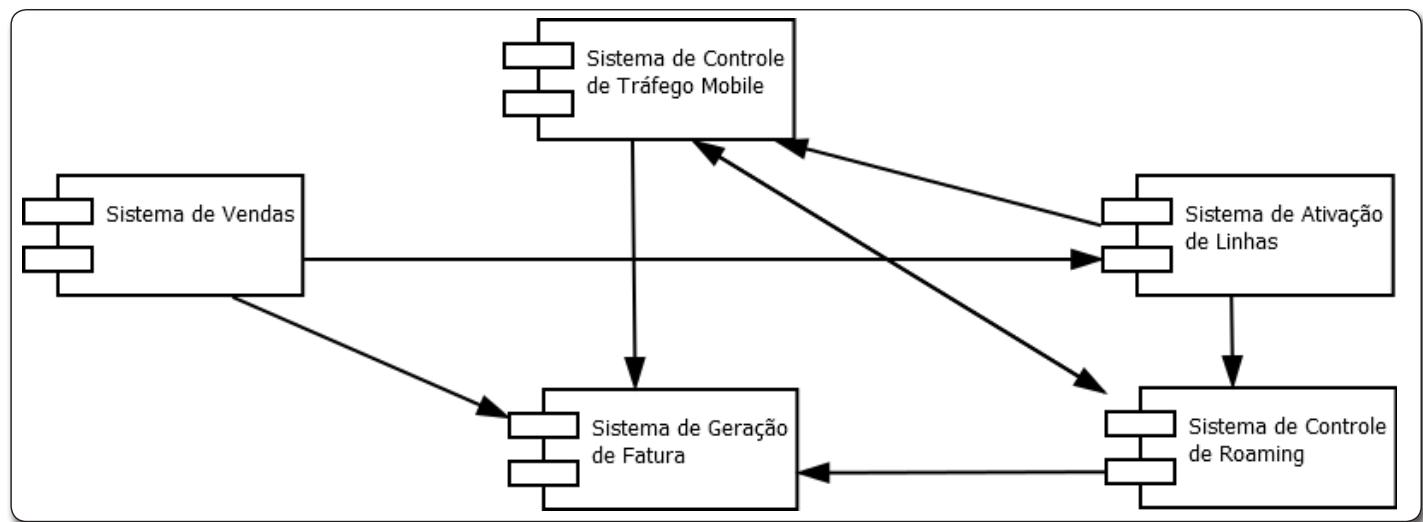


Figura 2. Relacionamento entre os sistemas da TelecoLeco

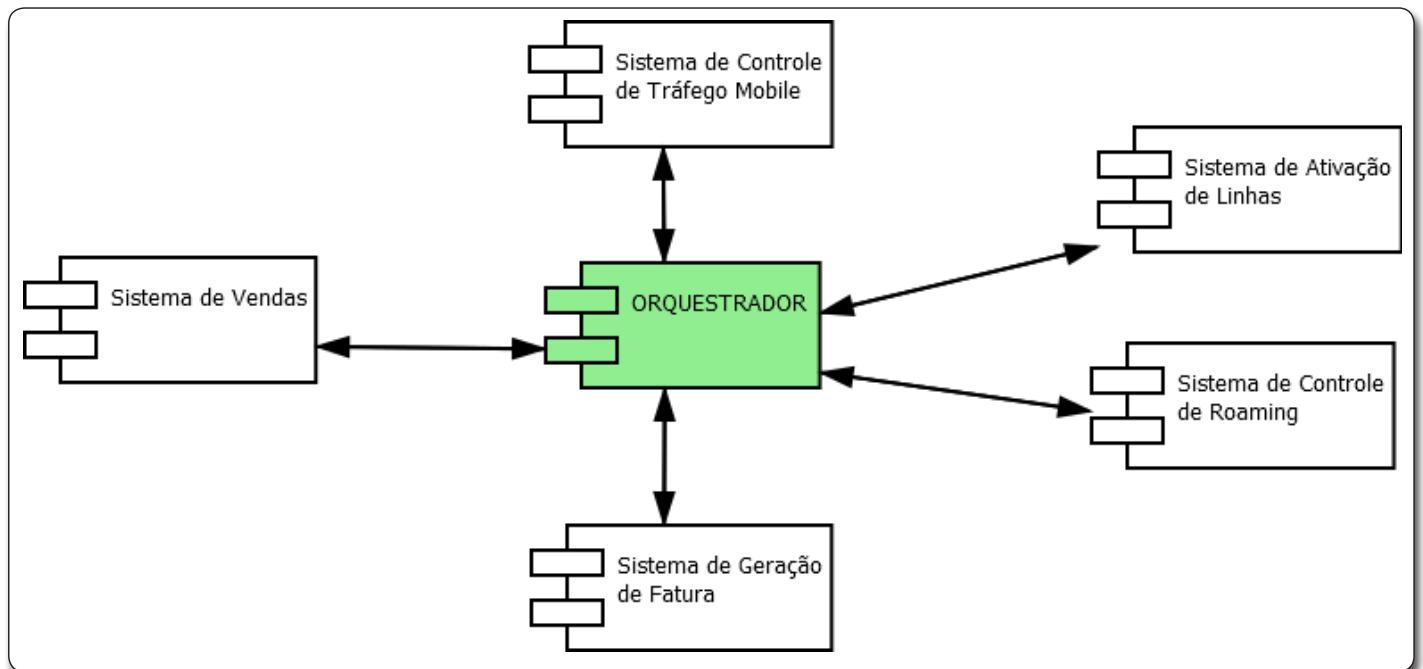


Figura 3. Organização da troca de mensagens entre os sistemas com um orquestrador

Listagem 6. Exemplo de uma ordem para o cenário de teste proposto.

```

<Order>
  <OrderNumber>8783764731</OrderNumber>
  <OrderDate>2016-05-06T16:53:19</OrderDate>
  <OrderType>Ordem de troca de plano</OrderType>
  <ListOfOrderLineItems>
    <OrderLineItem>
      <PhoneNumber>11977845210</PhoneNumber>
      <ActionCode>-</ActionCode>
      <ProductCategory>LINHA</ProductCategory>
      <ProductType>DADOS_MOVEL</ProductType>
    </OrderLineItem>
    <OrderLineItem>
      <ActionCode>Excluir</ActionCode>
      <Product>PLANO201</Product>
      <ProductName>Navegue Mais Web 10GB</ProductName>
    </OrderLineItem>
  </ListOfOrderLineItems>
</Order>
  
```

```

<ProductCategory>PLANO</ProductCategory>
<ProductType>POS_PAGO</ProductType>
<ProductSubType>DADOS_MOVEL</ProductSubType>
</OrderLineItem>
<OrderLineItem>
  <ActionCode>Incluir</ActionCode>
  <ProductPartNumber>PLANO301</ProductPartNumber>
  <ProductName>Navegue por semana 200MB</ProductName>
  <ProductCategory>PLANO</ProductCategory>
  <ProductType>PRE_PAGO</ProductType>
  <ProductSubType>DADOS_MOVEL</ProductSubType>
</OrderLineItem>
</ListOfOrderLineItems>
</Order>
  
```

diferentes mensagens XML. O problema é que cada uma dessas mensagens deve ter seus dados validados. Nos casos em que se tem N sistemas legados exigindo N formatos diferentes de XML, qual seria a melhor forma para fazer essas validações? Por causa da complexidade e trabalho extra com o mapeamento de diversas classes Java para XML e vice-versa, seria inviável criar vários POJOs, um para cada formato de mensagem de sistema legado, para validar com estruturas condicionais em Java.

Sendo assim, a melhor alternativa é realizar consultas XPath de testes diretamente sobre as mensagens XML geradas pelo orquestrador. Neste caso, ao invés de fazer o mapeamento das mensagens XML para POJOs e realizar os testes, aplica-se consultas XPath diretamente nas mensagens XML para sua validação. Essa alternativa pode ser implementada facilmente e, ainda ser melhorada, com a abordagem BDD do Cucumber. Os próximos tópicos mostram como implementar esse cenário de testes para mensagens XML.

Criando o teste com Cucumber

O primeiro passo para a implementação dos testes com Cucumber é a definição de features e cenários de teste e cada um dos seus passos. Como abordado anteriormente, um cenário de testes precisa estar em um arquivo de features, que pode conter um ou mais cenários, os passos dos cenários e demais informações para os testes.

A **Listagem 7** mostra a definição de um cenário hipotético de migração de linha para a TelecoLeco. Neste cenário, chamado de *Migration Pre to Pos*, dentro da feature *Line migrations* são descritos os elementos que uma ordem deve ter para ser considerada uma ordem de migração válida.

Listagem 7. Exemplo de cenário de teste com Cucumber.

Feature: Line migrations

The operators' system have to be able to change the customer's linetype between a set of options and plans.

Rules:

- Clients can't to change to the same type of plan he was before

Scenario: Migration Pre to Pos

Given I have an order with a ProductCategory LINHA and a ProductType DADOS_MOVEL and an ActionCode -
Given I have an order with a ProductCategory PLANO and a ProductType POS_PAGO and an ActionCode Excluir
Given I have an order with a ProductCategory PLANO and a ProductType PRE_PAGO and an ActionCode Incluir
Then the entire order Order.xml has the above definitions

No exemplo, os steps *Given* definem os dados que a ordem de migração deve ter, enquanto que o step *Then* faz a validação de que a ordem recebida condiz com a mensagem XML apresentada na **Listagem 6**.

Após ter o cenário definido, o próximo passo é implementar a sua execução em alguma linguagem de programação. As **Listagens 8** e **9** mostram as classes que implementam o teste desse cenário. A classe OrderTestDefs define os steps do cenário de testes, enquanto OrderItem atua como classe auxiliar para manter os dados da ordem durante a execução dos testes.

Listagem 8. Implementação da classe auxiliar OrderTestDefs.

```
package order;

import java.io.*;
import java.util.*;

import javax.xml.parsers.*;
import javax.xml.xpath.*;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

import cucumber.api.java.Before;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.core.EqualTo.equalTo;

public class OrderTestDefs {

    List<OrderItem> orderItemList = new ArrayList<OrderItem>();

    DocumentBuilderFactory factory;
    DocumentBuilder builder;
    XPathFactory xFactory;
    XPath xpath;

    @Before
    public void inicializa() throws ParserConfigurationException {
        factory = DocumentBuilderFactory.newInstance();
        builder = factory.newDocumentBuilder();
        xFactory = XPathFactory.newInstance();
        xpath = xFactory.newXPath();
    }

    @Given("I have an order with a ProductCategory (.+) and a ProductType (.+)
and an ActionCode (.+)")
    public void addItem(String productCategory, String productType,
String actionCode) {
        orderItemList.add(new OrderItem(productCategory, productType,
actionCode));
    }

    @Then("the entire order (.+) has the above definitions")
    public void loadOrderTemplate(String orderPayload)
throws ParserConfigurationException, SAXException, IOException,
XPathExpressionException {

        String filePath = "./resources/" + orderPayload;

        File orderPayloadFile = new File(filePath);
        assertThat((orderPayloadFile.exists() && orderPayloadFile.isFile()), equalTo(true));

        Document doc = builder.parse(filePath);

        for(OrderItem item : orderItemList) {
            XPathExpression expr = xpath.compile("//Order/ListOfOrderLineItems/
OrderLineItem[" +
"ProductCategory=" + item.productCategory + "" +
"and ProductType=" + item.productType + "" +
"and ActionCode=" + item.actionCode + "']");
            Object result = expr.evaluate(doc, XPathConstants.NODESET);
            NodeList nodes = (NodeList) result;
            assertThat(nodes.getLength() > 0, equalTo(true));
        }
    }
}
```

Como podemos verificar, **OrderTestDefs** possui três métodos com anotações do Cucumber. O método **inicializa()**, anotado com **@Before**, é um método chamado pelo Cucumber antes da execução dos demais testes do cenário em execução.

Listagem 9. Implementação da classe auxiliar OrderItem.

```
class OrderItem {  
  
    public OrderItem(String productCategory, String productType, String actionCode) {  
        this.productCategory = productCategory;  
        this.productType = productType;  
        this.actionCode = actionCode;  
    }  
  
    String productCategory;  
    String productType;  
    String actionCode;  
}
```

Métodos anotados com `@Before` podem ser apropriados para inicializar recursos que serão utilizados por outros métodos. Dessa forma, ao invés de inicializar os recursos todas as vezes que forem necessários, pode-se configurá-los uma única vez. No exemplo da **Listagem 8**, o método instancia **DocumentBuilderFactory**, **DocumentBuilder**, **XPathFactory** e **XPath**. Todos esses objetos serão utilizados por todos os outros métodos da classe.

Os demais métodos da classe **OrderTestDefs** são anotados com `@Given` ou `@Then`. Essas anotações identificam steps dos cenários de testes de acordo com seu tipo semântico (*Given, Then, When, And, But*) e aceitam como parâmetro uma **String**, que deve ser igual ao step definido no arquivo de feature do cenário (lembre-se que, funcionalmente falando, a distinção entre as anotações não tem efeito prático).

Note que ambos os métodos de step possuem uma expressão como “(+)”.

Essas construções em expressões regulares são um dos pilares da flexibilidade do Cucumber, pois proporcionam o reuso de definições de steps, evitam duplicação e tornam os testes fáceis de manter. Cada step na feature deve corresponder exatamente à **String** (expressão regular) adicionada como parâmetro na anotação do determinado método para aquele caso de teste. Além disso, métodos podem aceitar parâmetros definidos no step. Para isso, faz-se uso das expressões para capturar os parâmetros de acordo com seu formato. Como exemplo, considere os métodos implementados **addItem()** e **loadOrderItemTemplate()**, que recebem, respectivamente, três e um parâmetros. Nesse caso, a anotação do step do primeiro método possui três expressões: a primeira para o primeiro parâmetro, a segunda para o segundo parâmetro e assim sucessivamente. Essa mesma lógica se aplica ao segundo método.

Em termos de funcionalidade, o método **addItem()** é chamado nos steps com o seguinte template:

I have an order with a ProductCategory (+) and a ProductType (+) and an ActionCode (+)

Para cada step desse tipo no cenário, o método **addItem()** irá criar um objeto do tipo **OrderItem** com os parâmetros passados por meio do step e adicioná-lo em uma lista de **OrderItem** previamente inicializada. Ao término dos steps que chamam **addItem()**, a lista de **OrderItem** terá todos os itens que a suposta ordem deveria ter.

O segundo método, **loadOrderTemplate()**, é chamado nos steps com o template a seguir, pois esse método é anotado com uma

expressão regular que corresponde à forma como ele foi definido no arquivo de features:

the entire order (+) has the above definitions

Dessa vez, para cada step desse tipo, o método **loadOrderTemplate()** será chamado e receberá como parâmetro uma **String**, que deve ser o nome do arquivo XML contendo a ordem. O método faz a validação da existência do arquivo e, para cada item adicionado na lista de **OrdemItem** nos steps anteriores, faz uma consulta em XPath na ordem para identificar se os dados do item realmente existem.

Com essa abordagem, evita-se o parse das mensagens XML para objetos Java. O teste para validação de conteúdo é realizado diretamente sobre os payloads por meio de consultas em XPath. O teste torna-se, portanto, muito mais simples e fácil de ser implementado, tendo em vista que todas as complexidades de mapeamento XML-objeto são deixadas de lado. Ademais, a validação dos dados no momento dos testes é feita com XPath, uma tecnologia intimamente relacionada com payloads XML, fazendo muito mais sentido esse tipo de teste do que a transformação de payloads em objetos somente para validação.

A abordagem apresentada neste artigo para a criação de testes de payloads XML foca na implementação da infraestrutura inicial de testes com o Cucumber e nas tecnologias para XML relacionadas, como o XPath. A solução apresentada ao longo do artigo abre muitos caminhos para os testes de payloads XML, como, por exemplo, a possibilidade de testes automatizados com uma grande quantidade de cenários. Com essa possibilidade, todas as vantagens e facilidades proporcionadas pelo Cucumber ficam disponíveis para o desenvolvedor utilizar nesses cenários de testes sem a necessidade de mapeamento entre XML e objetos, tornando a codificação dos testes mais rápida e simplificada, de forma a melhorar a cobertura dos mesmos e aumentar a produtividade.

Autor



Gabriel Novais Amorim

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

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

Seus interesses incluem arquitetura de software e metodologias ágeis. Tecnólogo em Análise e Desenvolvimento de Sistemas (Unifieo) e especialista (MBA) em Engenharia de Software (FIAP). Escreve também em: blog.gabrielamorim.com



Links:

Site do Cucumber

<https://cucumber.io/>

Guia de instalação do Cucumber para a plataforma Java

<https://cucumber.io/docs/reference/jvm#java>

Apache Camel: Um guia completo – Parte 4

Veja neste artigo a conclusão da implementação da solução de integração para um sistema de pagamento bancário

ESTE ARTIGO FAZ PARTE DE UM CURSO

Na terceira parte desta série de artigos foram implementados na solução para pagamento bancário os fluxos de comunicação com o sistema bancário e parte da comunicação com o sistema de segurança. Com isso, demos continuidade à construção e elaboração da arquitetura da solução baseada em componentes com comunicação interna utilizando mensageria no formato JSON, permitindo assim baixo acoplamento, processamento assíncrono, performance, flexibilidade, tolerância a falhas, escalabilidade e baixo custo de manutenção. Ademais, foram apresentados conceitos e funcionalidades básicas do Apache Camel através da implementação de fluxos de negócios.

Nessa quarta parte da série, será finalizada a construção da solução de pagamento. Para isso, novos recursos do framework serão introduzidos, com destaque para a implementação do EIP Control Bus, visando a monitoração de transações. Também serão apresentados códigos que fazem a utilização de componentes de mensageria e o roteamento e manipulação de mensagens buscando sempre atender as regras de negócio.

Para exercitar e aprender esses recursos do Camel, os requisitos definidos no artigo anterior continuarão a ser implementados, em sua sequência natural, para que as integrações sejam finalizadas - em especial a realizada com o aplicativo mobile - ao passo que a construção da arquitetura da solução também é evoluída e terminada.

Implementando o requisito RF-05-SISSEG

A implementação do requisito RF-05-SISSEG consiste em permitir que a Solução de Pagamento Bancário

Fique por dentro

Este artigo apresenta recursos do Apache Camel através da implementação de uma solução de pagamento bancário. Por meio dessa implementação, será possível constatar as facilidades e rapidez que o framework oferece para encapsular código complexo capaz de integrar sistemas heterogêneos. Portanto, aqueles que desejam aprender na prática os principais recursos para a construção de um sistema integrado com o auxílio do Apache Camel, encontrarão neste artigo o conteúdo necessário para o alcance desse objetivo.

receba a resposta do Sistema de Segurança com o resultado da validação do número e código de segurança do cartão através de uma mensagem com a flag **isValido**. A ideia da implementação desse requisito segue o que foi proposto em RF-03-SISBAN, onde 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.

Para isso, inclua no método **configure()** da classe **SegurancaIntegracaoRouteBuilder**, presente no componente **SegurancaIntegracao**, uma nova rota para consumir mensagens da fila de resposta do Sistema de Segurança. Esse novo código deve ser equivalente ao da **Listagem 1**.

Note que a linha 1 possui o código para consumir mensagens da fila **SegurancaIntegracaoResposta**. Já a linha 3 inclui um processador do tipo **PreparaRespostaValidacaoCodigoSegurancaCartaoProcessor** para tratar a mensagem de resposta do sistema externo. A linha 4, por sua vez, define uma constante no header do Exchange com a chave **FLUXO** e o valor **RESPOSTA_VALIDACAO_CODIGO_SEGURANCA_CARTAO** para que o fluxo atual possa ser identificado pelo próximo componente. Por fim, a linha 6 faz o envio da mensagem para o **PagamentoBancario**. Deve-se ainda incluir a variável **preparaRespostaValidacaoCodigoSegurancaCartaoProcessor**, responsável por criar o processador,

juntamente com a implementação de seu respectivo método setter para a injeção de dependência.

Na sequência, é preciso codificar o processador **PreparaRespostaValidacaoCodigoSegurancaCartaoProcessor**. Para isso, crie uma classe com esse mesmo nome no pacote **br.com.devmedia.seguranca.integracao.processor** do projeto **SegurancaIntegracao**, conforme a **Listagem 2**.

Listagem 1. Código para consumir mensagens de resposta do Sistema de Segurança.

```
01 from("activemq:queue:SegurancaIntegracaoResposta"
02   concurrentConsumers=5").id("rotaRespostaSegurancaIntegracao")
03   .log(LoggingLevel.INFO, "[SegurancaIntegracao]
04   Recebendo resposta da validação código de segurança do cartão")
05   .process(preparaRespostaValidacaoCodigoSegurancaCartaoProcessor)
06   .setHeader("FLUXO", constant("RESPOSTA_VALIDACAO_CODIGO
07   _SEGURANCA_CARTAO"))
08   .log(LoggingLevel.INFO, "[SegurancaIntegracao] para
09   [PagamentoBancario] Enviando resposta da validação do código de
10  segurança do cartão")
11   .to("activemq:queue:PagamentoBancario?deliveryPersistent=false")
12 .end();
```

Listagem 2. Código da classe PreparaRespostaValidacaoCodigoSegurancaCartaoProcessor.

```
01 package br.com.devmedia.seguranca.integracao.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.slf4j.Logger;
06 import org.slf4j.LoggerFactory;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
10 import br.com.devmedia.seguranca.integracao.comum.RespostaSistema
11 SegurancaCartao;
12
13 @Component
14 public class PreparaRespostaValidacaoCodigoSegurancaCartaoProcessor
15 implements Processor {
16
17   private static final Logger LOGGER = LoggerFactory.getLogger(
18     (PreparaRespostaValidacaoCodigoSegurancaCartaoProcessor.class));
19
20   @Override
21   public void process(Exchange exchange) throws Exception {
22     LOGGER.info("Json recebido: " + exchange.getIn().getBody());
23     RespostaSistemaSegurancaCartao respostaSistemaSegurancaCartao =
24       ConversorJson.converteJsonParaObjeto(exchange.getIn().getBody())
25       .toString(), RespostaSistemaSegurancaCartao.class);
26     String json = ConversorJson.converteObjetoParaJson(
27       (respostaSistemaSegurancaCartao);
28     exchange.getIn().setBody(json);
29   }
30 }
```

Esse código simplesmente recupera a mensagem em formato JSON e a converte para o POJO **RespostaSistemaSegurancaCartao**. Logo após, converte esse mesmo POJO para JSON e o inclui no body da mensagem, para que ela seja enviada posteriormente ao próximo componente do fluxo. Vale ressaltar que para fins didáticos o código apenas faz conversões, mas em um cenário real é possível, por exemplo, gravar dados no banco de dados e executar validações.

Para finalizar a implementação do requisito, é necessário criar um POJO denominado **RespostaSistemaSegurancaCartao** para representar os dados de resposta do Sistema de Segurança. Deste modo, crie o pacote **br.com.devmedia.seguranca.integracao.comum** e implemente a classe **RespostaSistemaSegurancaCartao** na biblioteca **SegurancaIntegracaoComum** conforme o código da **Listagem 3**.

Listagem 3. Código da classe RespostaSistemaSegurancaCartao.

```
01 package br.com.devmedia.seguranca.integracao.comum;
02
03 import java.io.Serializable;
04
05 public class RespostaSistemaSegurancaCartao implements Serializable {
06
07   private static final long serialVersionUID = -663540445183845906L;
08
09   private String numeroCartao;
10   private String codigoSegurancaCartao;
11   private Boolean valido;
12
13   public RespostaSistemaSegurancaCartao() {}
14
15   // Getters e Setters dos atributos omitidos.
16
17   // Método toString() omitido.
18
19 }
```

Concluída essa implementação, mais uma etapa para a construção da solução foi completada. Dessa forma, a situação atual do fluxo de negócio já implementado e da arquitetura da Solução de Pagamento Bancário podem ser conferidas nas **Figuras 1** e **2**, respectivamente.

Implementando os requisitos RF-06-SISAUT e RNF-04-SISAUT

Começaremos este tópico apresentando a implementação do requisito RF-06-SISAUT, que corresponde a receber a resposta do sistema de segurança e 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. Além desse requisito, o RNF-04-SISAUT também será atendido; seu objetivo basicamente é ditar que a comunicação com o sistema de autorização seja realizada via mensageria com o formato JSON.

Assim sendo, será necessário complementar o código da classe **PagamentoBancarioRouteBuilder**, do componente **PagamentoBancario**, para implementar os requisitos anteriores. As alterações são apresentadas na **Listagem 4**.

Verificando o código apresentado, observa-se nas linhas 9 a 11 a inclusão de mais um predicado, com o nome **respostaValidacaoCodigoSegurancaCartao**, na rota de entrada, para verificar se a mensagem que está sendo recebida pertence ao fluxo de resposta de validação do Sistema de Segurança. Se atender a essa condição, a mensagem será direcionada para a rota interna **respostaValidacaoCodigoSegurancaCartao**.

Na sequência, tem-se a codificação da nova rota interna, **respostaValidacaoCodigoSegurancaCartao**, conforme as linhas 15 a 21. Essa rota deve incluir dois processadores.

Apache Camel: Um guia completo – Parte 4

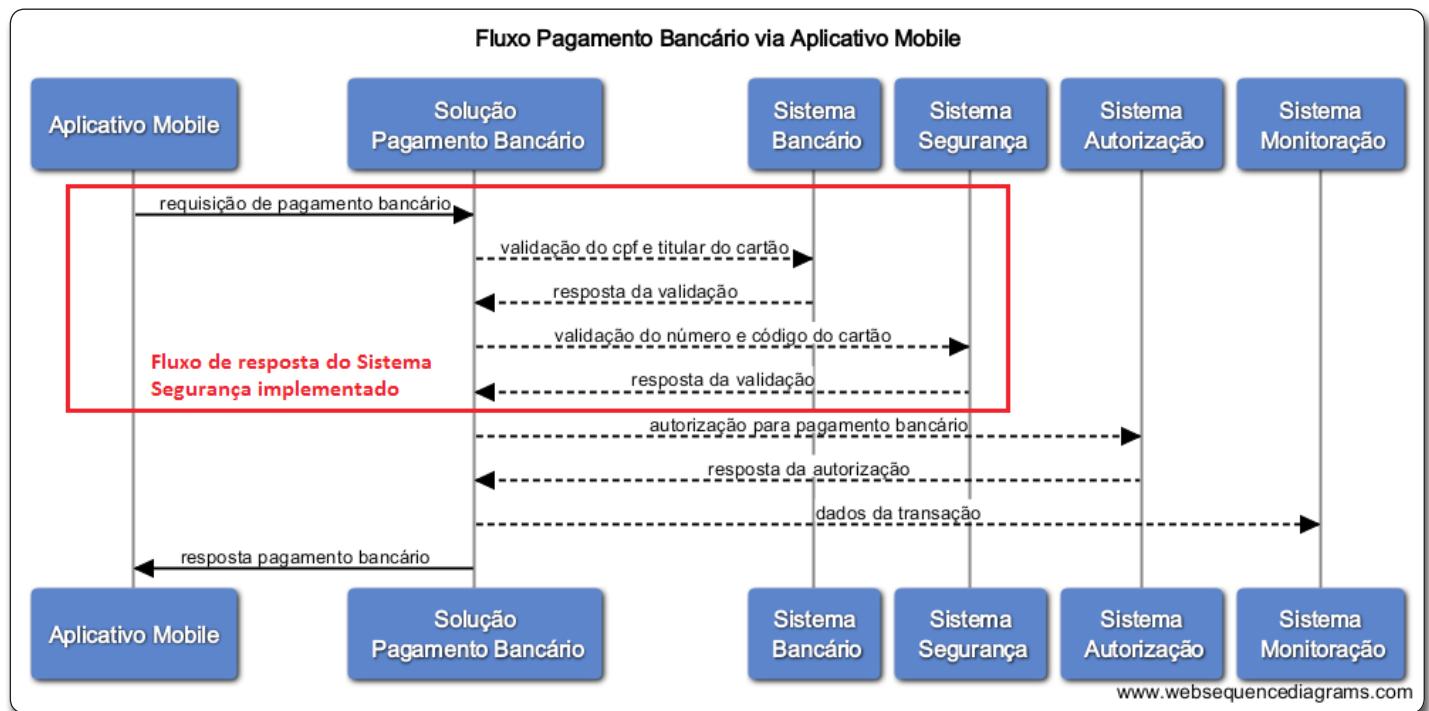


Figura 1. Terceiro fluxo de negócio implementado

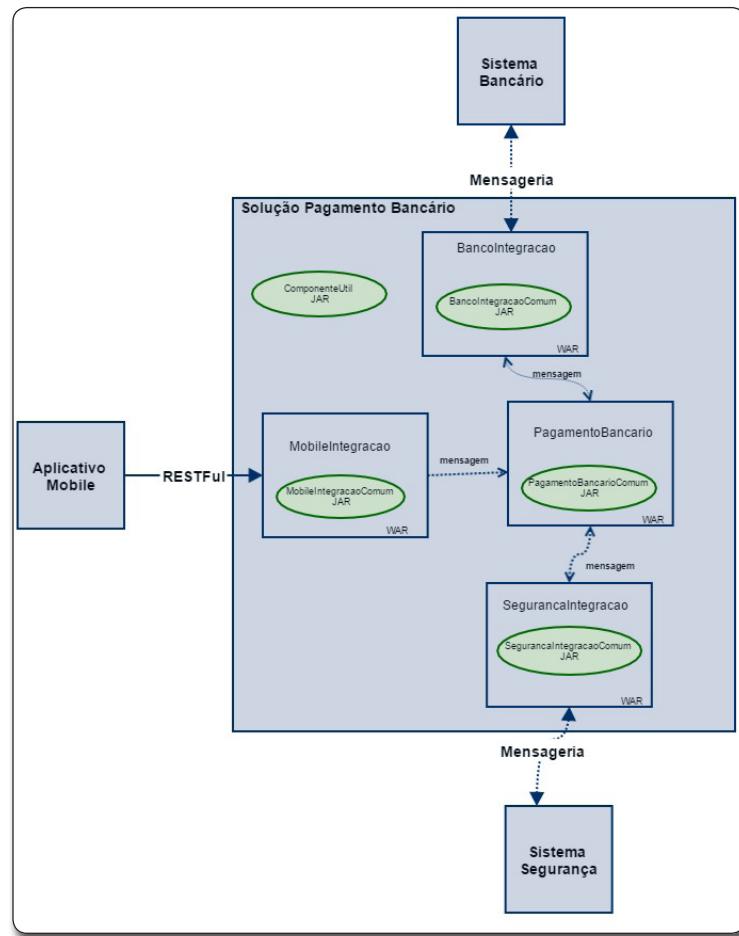


Figura 2. Solução apta a receber mensagens do Sistema Segurança e repassar a resposta

Listagem 4. Novas alterações na classe PagamentoBancarioRouteBuilder.

```

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

```

A função do primeiro é verificar a resposta do sistema de segurança e a função do segundo é preparar a mensagem para o próximo componente. Por fim, na linha 21 é realizado o envio da mensagem a ser consumida pelo próximo componente do fluxo através da fila **AutorizacaoIntegracao**.

Feito isso, crie no pacote **br.com.devmedia.pagamento.bancario.processor** um novo processor, chamado **VerificaRespostaValidacaoCodigoSegurancaCartaoProcessor** (vide [Listagem 5](#)), ainda no componente **PagamentoBancario**.

Listagem 5. Código da classe VerificaRespostaValidacaoCodigoSegurancaCartaoProcessor.

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

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

Continuando, devemos implementar o processor **PopulaAutorizacaoComumProcessor**, a ser criado também no pacote **br.com.devmedia.pagamento.bancario.processor**. Seu código é apresentado na [Listagem 6](#).

Examinando essa listagem, nota-se na linha 16 que o objeto incluído no header da mensagem durante a execução do fluxo no componente **MobileIntegracao** é recuperado para que seja utilizado.

Listagem 6. Código da classe PopulaAutorizacaoComumProcessor.

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

Na próxima linha, 17, um objeto do tipo **AutorizacaoIntegracaoComum** é instanciado para que o código da linha 18 popule, com base justamente nesse objeto recuperado no header da mensagem, o **autorizacaoIntegracao**.

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

Implementando os componentes AutorizacaoIntegracaoComum e AutorizacaoIntegracao

A implementação da biblioteca **AutorizacaoIntegracaoComum**, cuja finalidade é representar os dados que serão trafegados entre os componentes internos da solução e também com o sistema

externo de autorização de pagamento, deve-se iniciar com a criação de um novo projeto/módulo do tipo Java Standalone (JAR) dentro do diretório do projeto agregador. Feito isso, crie nesse novo projeto um *pom.xml* com o código da **Listagem 7** para definir as configurações básicas da biblioteca.

Na sequência, deve-se criar o pacote **br.com.devmedia.autorizacao.integracao.comum** e adicionar o POJO **AutorizacaoIntegracaoComum** para representar os dados a serem enviados para o sistema externo de autorização. Essa classe deve ser implementada em conformidade com a **Listagem 8**.

Listagem 7. Arquivo pom.xml da biblioteca AutorizacaoIntegracaoComum.

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

Listagem 8. Código da classe AutorizacaoIntegracaoComum.

```
01 package br.com.devmedia.autorizacao.integracao.comum;
02
03 import java.io.Serializable;
04 import java.math.BigDecimal;
05
06 public class AutorizacaoIntegracaoComum implements Serializable {
07
08   private static final long serialVersionUID = -1365158473475488427L;
09
10   private String numeroCartao;
11   private BigDecimal valorCompra;
12
13   public AutorizacaoIntegracaoComum() {}
14
15   // Getters e Setters dos atributos omitidos.
16
17   // Método toString() omitido.
18
19 }
```

Verifica-se nesse código que a classe possui os dois atributos (número do cartão e valor da compra) que serão enviados ao sistema externo de autorização. Após a codificação dessa classe, é necessário ainda adicionar a biblioteca **AutorizacaoIntegracaoComum** nas dependências do projeto **PagamentoBancario** e também na declaração de módulos do projeto agregador.

Com a biblioteca **AutorizacaoIntegracaoComum** implementada, é a vez da criação do novo componente, chamado **AutorizacaoIntegracao**, responsável por fazer a comunicação com o Sistema de

Autorização e manter concentrada todas as regras e necessidades voltadas a essa integração.

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

Prosseguindo, deve-se ajustar o arquivo *web.xml*, localizado no diretório *src/main/webapp/WEB-INF*, conforme a **Listagem 9**, para que a aplicação carregue (ao iniciar) as configurações do Spring, Camel e ActiveMQ. Além disso, é preciso incluir um novo arquivo, denominado *application-context.xml* - semelhante ao da **Listagem 10** - no diretório *src/main/resources*, pois são através das declarações dos beans (gerenciados pelo Spring) contidas nesse arquivo que a conexão com o ActiveMQ é efetuada e o carregamento inicial das rotas do Camel é feito.

Listagem 9. web.xml do componente AutorizacaoIntegracao.

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

Examinando a **Listagem 10**, é possível reparar que a lógica e recursos declarados no arquivo *application-context.xml* seguem o mesmo padrão empregado no respectivo arquivo do componente **PagamentoBancario**. O diferencial neste caso está nas linhas 14 a 16, que informam ao Camel que a classe que contém as rotas a serem iniciadas quando a aplicação carregar é a **AutorizacaoIntegracaoRouteBuilder**, referenciada pela tag **camel:routeBuilder** na linha 15.

Finalizada a criação dos arquivos *web.xml* e *application-context.xml*, deve-se adicionar o pacote **br.com.devmedia.autorizacao.integracao.rota** e codificar a classe de rotas **AutorizacaoIntegracaoRouteBuilder** conforme a **Listagem 11**.

Listagem 10. application-context.xml do componente AutorizacaoIntegracao.

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

Analisando essa listagem, observa-se que o código entre as linhas 18 e 23 define uma rota para receber mensagens da fila **AutorizacaoIntegracao**. Na linha 20, um processador do tipo **PreparaPagamentoBancarioParaAutorizacaoProcessor** é incluído para criar uma mensagem compatível com a esperada pelo Sistema de Autorização. Por fim, na linha 22 é feito o envio da mensagem para a fila **SistemaExternoAutorizacao**, para que o Sistema de Autorização receba a mensagem.

Para concluir, ainda é necessário desenvolver o processador **PreparaPagamentoBancarioParaAutorizacaoProcessor**. Portanto, crie o pacote **br.com.devmedia.autorizacao.integracao.processor** e adicione o processador **PreparaPagamentoBancarioParaAutorizacaoProcessor** conforme o código da Listagem 12.

O código dessa listagem converte um objeto **AutorizacaoIntegracaoComum**, que contém os dados para autorização de uma transação de pagamento, para uma **String** em formato JSON e na sequência inclui essa mesma **String** no body da mensagem para que ela seja enviada ao sistema externo posteriormente.

Com essa codificação, os requisitos RF-06-SISAUT e RNF-04-SISAUT foram implementados. Nas Figuras 3 e 4 é possível verificar quais os fluxos de negócios a solução já atende e o estado atual da arquitetura.

Listagem 11. Código inicial da classe AutorizacaoIntegracaoRouteBuilder.

```
01 package br.com.devmedia.autorizacao.integracao.rota;
02
03 import org.apache.camel.LoggingLevel;
04 import org.apache.camel.builder.RouteBuilder;
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.stereotype.Component;
07
08 import br.com.devmedia.autorizacao.integracao.processor.PreparaPagamento
09 BancarioParaAutorizacaoProcessor;
10
11 @Component
12 public class AutorizacaoIntegracaoRouteBuilder extends RouteBuilder {
13
14  private PreparaPagamentoBancarioParaAutorizacaoProcessor
15  preparaPagamentoBancarioParaAutorizacaoProcessor;
16
17  @Override
18  public void configure() throws Exception {
19
20    from("activemq:queue:AutorizacaoIntegracao?concurrentConsumers=5")
21      .id("rotaEntradaAutorizacaoIntegracao")
22      .log(LoggingLevel.INFO, "[AutorizacaoIntegracao]
23        Preparando pagamento bancário para autorização")
24      .process(preparaPagamentoBancarioParaAutorizacaoProcessor)
25      .log(LoggingLevel.INFO, "[AutorizacaoIntegracao] para [SistemaExterno
26        Autorizacao] Enviando pagamento bancário para autorização")
27      .to("activemq:queue:SistemaExternoAutorizacao?deliveryPersistent=fal
```

Listagem 12. Código da classe PreparaPagamentoBancarioParaAutorizacaoProcessor.

```
01 package br.com.devmedia.autorizacao.integracao.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.slf4j.Logger;
06 import org.slf4j.LoggerFactory;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.autorizacao.integracao.comum
10 .AutorizacaoIntegracaoComum;
11 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
12
13 @Component
14 public class PreparaPagamentoBancarioParaAutorizacaoProcessor implements
15 Processor {
16
17  private static final Logger LOGGER = LoggerFactory.getLogger(
18    PreparaPagamentoBancarioParaAutorizacaoProcessor.class);
19
20  @Override
21  public void process(Exchange exchange) throws Exception {
22    AutorizacaoIntegracaoComum autorizacaoIntegracao = ConversorJson
23      .converteJsonParaObjeto(exchange.getIn().getBody().toString(),
24        AutorizacaoIntegracaoComum.class);
25    String json = ConversorJson.converteObjetoParaJson
26      (autorizacaoIntegracao);
27    LOGGER.info("Json enviado: " + json );
28    exchange.getIn().setBody(json);
29  }
30}
```

Implementando o requisito RF-07-SISAUT

A implementação do requisito RF-07-SISAUT consiste em permitir que a Solução de Pagamento Bancário receba a resposta do Sistema de Autorização com o resultado da transação de pagamento bancário contendo: número do cartão, valor da compra, status do pagamento e código de autorização do pagamento, mensagem complementar do status e data de autorização.

Apache Camel: Um guia completo – Parte 4

Vale ressaltar que em um cenário alternativo, se a transação não for aprovada, os campos código de autorização e data de autorização não serão preenchidos.

Portanto, precisamos incluir no método `configure()` da classe `AutorizacaoIntegracaoRouteBuilder`, do componente `AutorizacaoIntegracao`, uma nova rota para consumir mensagens da fila de resposta do Sistema Bancário.

Esse novo código deve ser equivalente ao da [Listagem 13](#).

Verificando essa listagem, observa-se na linha 1 o código para consumir mensagens da fila `AutorizacaoIntegracaoResposta`. Em seguida, a linha 3 inclui um processador do tipo `PreparaResposta-AutorizacaoPagamentoBancarioProcessor` para tratar a mensagem de resposta do sistema externo. Logo após, a linha 4 define uma constante no header da mensagem com a chave `FLUXO` e o

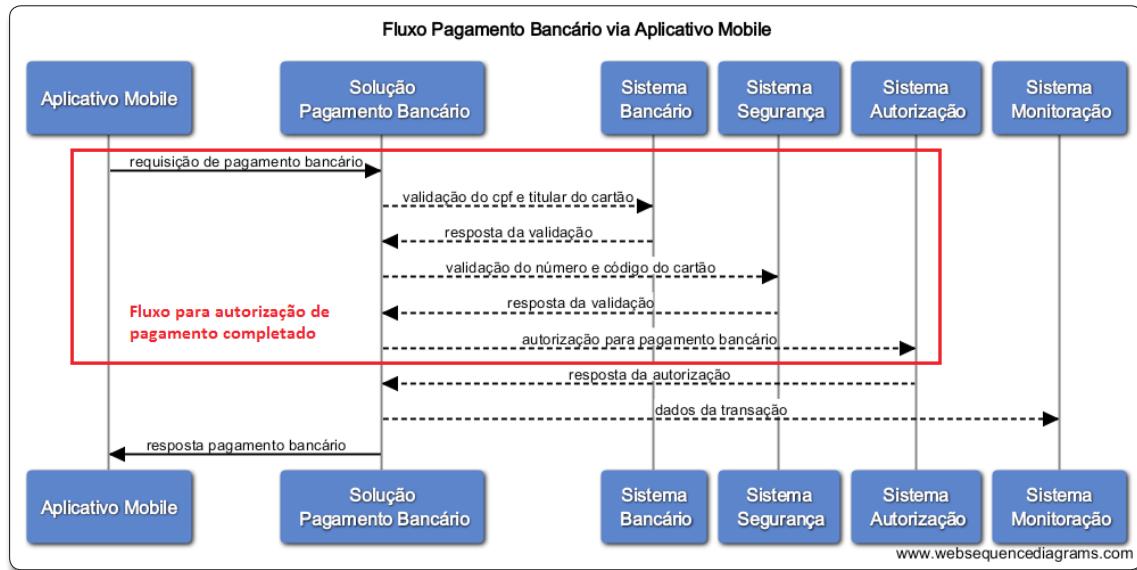


Figura 3. Fluxo para autorização de pagamento implementado

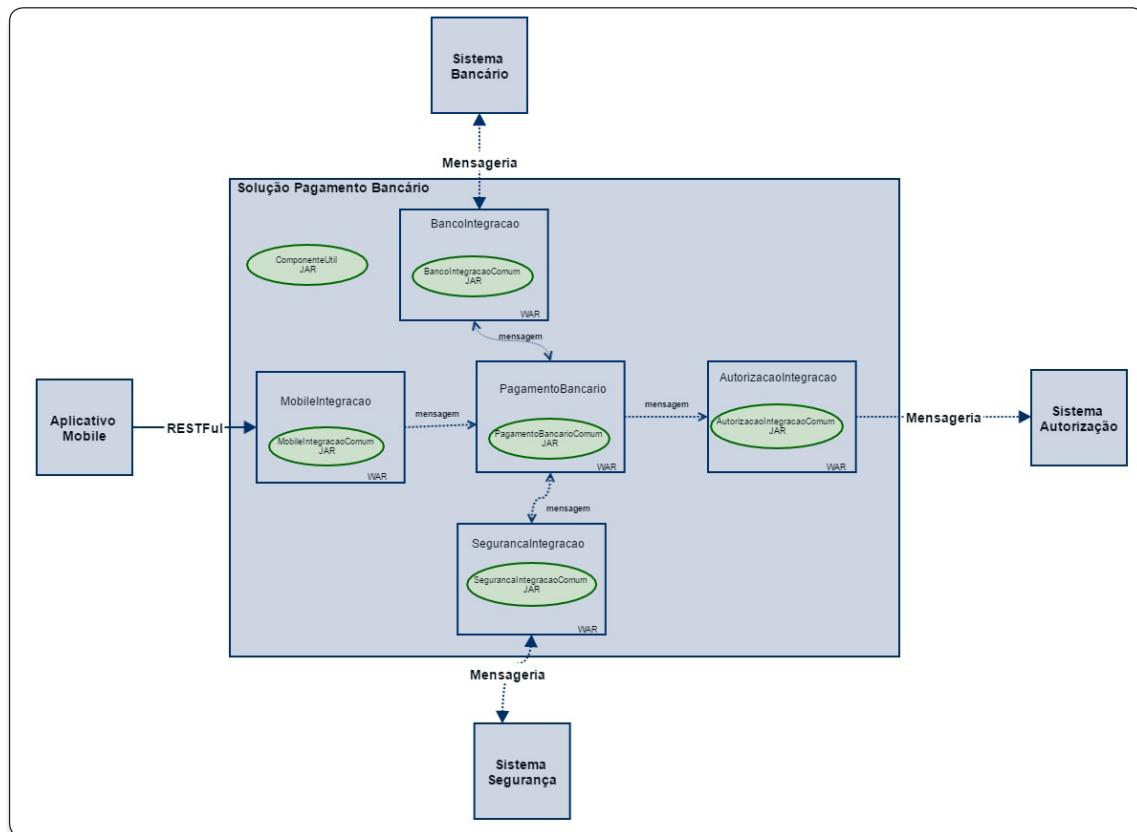


Figura 4. Diagrama da arquitetura da Solução de Pagamento Bancário até o momento.

valor **RESPOSTA_AUTORIZACAO_PAGAMENTO_BANCARIO**, para que o fluxo de negócio que está em execução possa ser identificado pelo próximo componente e assim rotear a mensagem ao destinatário correto. Por fim, a linha 6 faz o envio da mensagem para o **PagamentoBancario**.

Listagem 13. Código para consumir mensagens de resposta do sistema de autorização.

```
01 from("activemq:queue:AutORIZACAOIntegracaoResposta?concurrent  
Consumers=5").id("rotaRespostaAutORIZACAOIntegracao")  
02 .log(LoggingLevel.INFO, "[AutORIZACAOIntegracao] Recebendo resposta da  
autorização de pagamento bancário")  
03 .process(preparaRespostaAutORIZACAOIntegracaoProcessor)  
04 .setHeader("FLUXO", constant("RESPOSTA_AUTORIZACAO_PAGAMENTO_  
BANCARIO"))  
05 .log(LoggingLevel.INFO, "[AutORIZACAOIntegracao] para [PagamentoBancario]  
Enviando resposta da autorização de pagamento bancário")  
06 .to("activemq:queue:PagamentoBancario?deliveryPersistent=false")  
07 .end();
```

Na sequência, é preciso codificar o processor **PreparaRespostaAutORIZACAOIntegracaoProcessor**. Sendo assim, crie uma classe com esse mesmo nome no pacote **br.com.devmedia.autORIZACAO.integracao.processor** do projeto **AutorizaçaoIntegracao**. Seu código é apresentado na **Listagem 14**.

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

Para finalizar a implementação do requisito, é necessário criar um POJO denominado **RespostaSistemaAutORIZACAO** para representar os dados de resposta do Sistema Bancário. Deste modo, no pacote **br.com.devmedia.autORIZACAO.integracao.comum**, implemente a classe **RespostaSistemaAutORIZACAO** na biblioteca **AutORIZACAOIntegracaoComum** conforme o código da **Listagem 15**.

Finalizada a criação dessa classe, o requisito foi atendido e mais um fluxo de negócio implementado. Além disso, mais um pequeno passo foi dado para a construção da solução, que agora está apta a receber as respostas do Sistema de Autorização e enviá-las para o componente seguinte. Para um melhor entendimento da situação atual, observe as **Figuras 5 e 6**. Na sequência, será iniciada a implementação dos requisitos para resposta ao Aplicativo Mobile, incluindo a mensagem de monitoração para o respectivo sistema.

Implementando os requisitos de resposta ao aplicativo mobile e de monitoração

A partir de agora será iniciada a implementação dos requisitos: RF-09-APPMOB, RNF-06-APPMOB, RF-08-SISMON, RF-11-SISMON e RNF-05-SISMON.

O requisito RF-09-APPMOB diz que a Solução de Pagamento Bancário 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. Já o requisito RNF-06-APPMOB indica que essa resposta deve ocorrer em até 20 segundos a partir do início da transação.

Listagem 14. Código da classe **PreparaRespostaAutORIZACAOIntegracaoProcessor**.

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

Listagem 15. Código da classe **RespostaSistemaAutORIZACAO**.

```
01 package br.com.devmedia.autORIZACAO.integracao.comum;  
02  
03 import java.io.Serializable;  
04 import java.math.BigDecimal;  
05 import java.util.Date;  
06  
07 public class RespostaSistemaAutORIZACAO implements Serializable {  
08  
09 private static final long serialVersionUID = 7743525865955397647L;  
10  
11 private String numeroCartao;  
12 private BigDecimal valorCompra;  
13 private String status;  
14 private String codigoAutORIZACAO;  
15 private String mensagem;  
16 private Date dataAutORIZACAO;  
17  
18 public RespostaSistemaAutORIZACAO() {}  
19 // Getters e Setters dos atributos omitidos.  
20  
21 // Método toString() omitido.  
22 }
```

Vale ressaltar que a arquitetura da solução é capaz de suportar tal necessidade, porém, em um cenário real, é importante realizar testes para garantir que um requisito como esse seja atendido.

Já o requisito de monitoração RF-08-SISMON indica que após a conclusão de uma transação de pagamento a Solução de Pagamento Bancário deve enviar uma mensagem, baseada em um layout pré-definido no requisito RF-11-SISMON, com os dados da transação para o sistema de monitoração. Por fim, o RNF-05-SISMON tem como objetivo basicamente ditar que a comunicação com o sistema de monitoração seja realizada via mensageria com o formato texto.

Assim sendo, será necessário complementar o código da classe **PagamentoBancarioRouteBuilder**, do componente **PagamentoBancario**, para implementar os requisitos anteriores.

Apache Camel: Um guia completo – Parte 4

As alterações são apresentadas na **Listagem 16**.

Analizando o código da listagem anterior, nota-se nas linhas 12 a 14 a inclusão de mais um predicado, com o nome **respostaAutorizacaoPagamentoBancarioCartao**, na rota de entrada, para verificar se a mensagem que está sendo recebida pertence ao fluxo de resposta da autorização de pagamento. Se atender essa condição, a mensagem será direcionada para a rota interna **respostaAutorizacaoPagamentoBancario**.

Na sequência, tem-se a codificação da nova rota interna, **respostaAutorizacaoPagamentoBancario**, conforme as linhas 18 a 23. Essa rota inclui um processador chamado **verificaRespostaAutorizacaoPagamentoBancarioProcessor** para verificar a resposta da autorização e na sequência redireciona o fluxo para as rotas internas **notificaTransacaoPagamentoBancario** e **respostaPagamentoBancario**, através da DSL multicast, na linha 21.

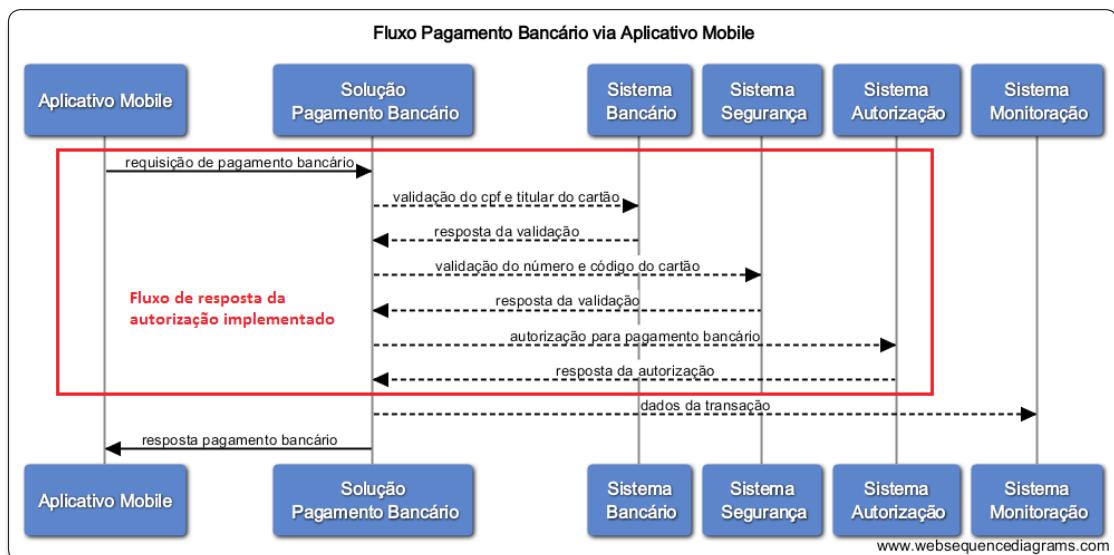


Figura 5. Cenário atual dos fluxos de negócio já atendidos

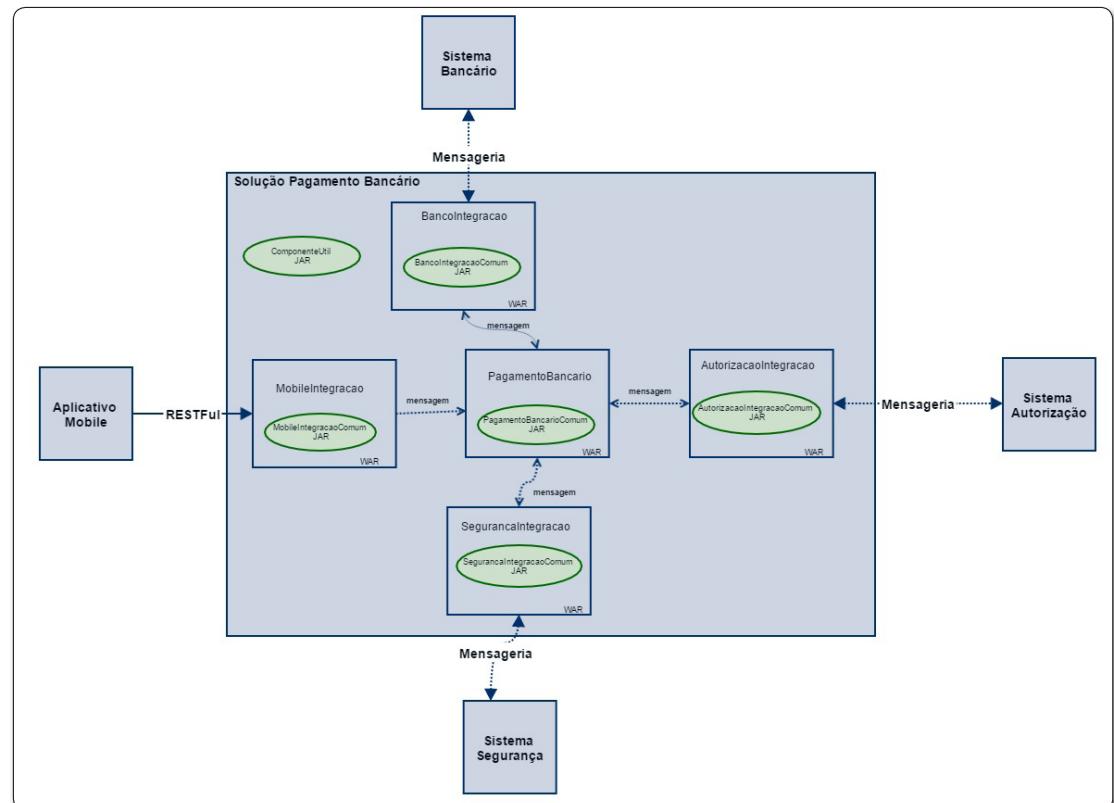


Figura 6. Solução apta a receber mensagens do Sistema Autorização e repassar a resposta

Listagem 16. Modificações na classe PagamentoBancarioRouteBuilder.

```
01 from("activemq:queue:PagamentoBancario?concurrentConsumers=5")
02   .id("rotaEntradaPagamentoBancario")
03   .choice()
04     .when(novaRequisicaoPagamentoBancario)
05       .log(LoggingLevel.INFO, "[PagamentoBancario] Nova requisição de
06         pagamento bancário")
07       .to("sed:a:enviaDadosCartaoValidacao")
08       .when(respostaValidacaoDadosCartao)
09         .log(LoggingLevel.INFO, "[PagamentoBancario]
10           Resposta de validação dos dados do cartão")
11         .to("sed:a:respostaValidacaoDadosCartao")
12       .when(respostaValidacaoCodigoSegurancaCartao)
13         .log(LoggingLevel.INFO, "[PagamentoBancario]
14           Resposta de validação do código de segurança do cartão")
15         .to("sed:a:respostaValidacaoCodigoSegurancaCartao")
16       .when(respostaAutorizacaoPagamentoBancarioCartao)
17         .log(LoggingLevel.INFO, "[PagamentoBancario] Resposta da
18           autorização de pagamento bancário")
19         .to("sed:a:respostaAutorizacaoPagamentoBancario")
20       .endChoice()
21     .end();
22
23   from("sed:a:respostaAutorizacaoPagamentoBancario")
24     .log(LoggingLevel.INFO, "[PagamentoBancario] Verificando resposta da
25       autorização de pagamento bancário")
26     .process(verificaRespostaAutorizacaoPagamentoBancarioProcessor)
27     .multicast()
28     .to("sed:a:notificaTransacaoPagamentoBancario","sed:a:respostaPagamento
29       Bancario")
30   .end();
31
32   from("sed:a:notificaTransacaoPagamentoBancario")
33     .log(LoggingLevel.INFO, "[PagamentoBancario]
34       Preparando dados da
35         transação de pagamento bancário para monitoração")
36     .process(populaMonitoracaoComumDadosTransacaoProcessor)
37     .log(LoggingLevel.INFO, "[PagamentoBancario] para
38       [MonitoracaoIntegracao] Notificando transação de pagamento bancário")
39     .to("activemq:queue:MonitoracaoIntegracao?deliveryPersistent=false")
40   .end();
41
42   from("sed:a:respostaPagamentoBancario")
43     .log(LoggingLevel.INFO, "[PagamentoBancario]
44       Preparando resposta de pagamento bancário")
45     .process(populaRespostaPagamentoBancarioProcessor)
46     .setHeader("FLUXO", constant("RESPOSTA_AUTORIZACAO_
47       PAGAMENTO_BANCARIO"))
48     .log(LoggingLevel.INFO, "[PagamentoBancario] para [MobileIntegracao]
49       Enviando resposta de pagamento bancário")
50     .to("activemq:queue:MobileIntegracao?deliveryPersistent=false")
51   .end();
```

As linhas 25 a 30 definem a rota **notificaTransacaoPagamentoBancario**, responsável por criar uma mensagem de monitoração e enviá-la ao componente **MonitoracaoIntegracao**, que será desenvolvido logo mais. Na linha 27, por sua vez, é incluído um processador denominado **populaMonitoracaoComumDadosTransacaoProcessor**, e na linha 29 é realizado o envio da mensagem para a fila **MonitoracaoIntegracao**.

Já as linhas 32 a 38 declaram a rota **respostaPagamentoBancario**, que criará uma mensagem de resposta ao componente **MobileIntegracao** com o resultado da transação de pagamento através do processador **populaRespostaPagamentoBancarioProcessor**, na linha 34. Além disso, na linha 35 é definida uma constante no header do Exchange com a chave **FLUXO** e o valor **RESPOSTA_AUTORIZACAO_PAGAMENTO_BANCARIO**, para que o próximo componente consiga identificar o fluxo em questão. Por fim, na linha 37 é feito o envio da mensagem para a fila **MobileIntegracao**.

Alterada a classe de rotas, é a vez da implementação dos três processadores mencionados. Primeiramente, será criado o **VerificaRespostaAutorizacaoPagamentoBancarioProcessor**. Para isso, implemente uma classe de mesmo nome no pacote **br.com.devmedia.pagamento.bancario.processor** com o código da **Listagem 17**.

Listagem 17. Código da classe VerificaRespostaAutorizacaoPagamentoBancarioProcessor.

```
01 package br.com.devmedia.pagamento.bancario.processor;
02
03 import java.util.Date;
04
05 import org.apache.camel.Exchange;
06 import org.apache.camel.Processor;
07 import org.springframework.stereotype.Component;
08
09 import br.com.devmedia.autorizacao.integracao.comum
10   .RespostaSistemaAutorizacao;
11 import br.com.devmedia.pagamento.bancario.comum
12   .PagamentoBancarioComum;
13 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
14
15 @Component
16 public class VerificaRespostaAutorizacaoPagamentoBancarioProcessor
17   implements Processor {
18
19   @Override
20   public void process(Exchange exchange) throws Exception {
21     RespostaSistemaAutorizacao respostaSistemaAutorizacao =
22       ConversorJson.converteJsonParaObjeto(exchange.getIn().getBody()
23         .toString(), RespostaSistemaAutorizacao.class);
24     if ("APROVADO".equalsIgnoreCase(respostaSistemaAutorizacao.getStatus())) {
25       PagamentoBancarioComum pagamentoBancario = ProcessorUtil
26         .getObjetoHeader(exchange, PagamentoBancarioComum.class);
27       pagamentoBancario.setDataFinalTransacao(new Date());
28       pagamentoBancario.setStatus(respostaSistemaAutorizacao.getStatus());
29       ProcessorUtil.setObjetoHeader(exchange, pagamentoBancario);
30     }
31   }
32 }
```

O código dessa classe converte a mensagem enviada em formato JSON para um objeto do tipo **RespostaSistemaAutorizacao** e na sequência verifica se o status é equivalente a **APROVADO**. Caso positivo, recupera o objeto **PagamentoBancarioComum** do header do Exchange, inclui a data final da transação e o status da resposta no mesmo e novamente o coloca no header.

Continuando com a codificação dos processadores, é a vez do **PopulaMonitoracaoComumDadosTransacaoProcessor**. Sendo assim, implemente uma classe de mesmo nome no pacote **br.com.devmedia.pagamento.bancario.processor**, conforme a **Listagem 18**.

Essa classe basicamente recupera o objeto **PagamentoBancarioComum** do header do Exchange, instancia um POJO do tipo **MonitoracaoIntegracaoComum** e o popula com os dados retirados do objeto recuperado do header. Por fim, converte o POJO para o formato JSON para que ele seja enviado posteriormente ao componente **MonitoracaoIntegracao**.

O último processador a ser implementado é o **PopulaRespostaPagamentoBancarioProcessor**. Portanto, crie o mesmo no pacote **br.com.devmedia.pagamento.bancario.processor** baseado no código da **Listagem 19**.

Apache Camel: Um guia completo – Parte 4

Listagem 18. Código da classe PopulaMonitoracaoComumDadosTransacaoProcessor.

```
01 package br.com.devmedia.pagamento.bancario.processor;
02
03 import org.apache.camel.Exchange;
04 import org.apache.camel.Processor;
05 import org.springframework.stereotype.Component;
06
07 import br.com.devmedia.monitoracao.integracao.comum
08 .MonitoracaoIntegracaoComum;
09 import br.com.devmedia.pagamento.bancario.comum.Pagamento
10 BancarioComum;
11 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
12
13 @Component
14 public class PopulaMonitoracaoComumDadosTransacaoProcessor implements
15 Processor {
16
17     @Override
18     public void process(Exchange exchange) throws Exception {
19         PagamentoBancarioComum pagamentoBancario = ProcessorUtil
20 .getObjetoHeader(exchange, PagamentoBancarioComum.class);
21         MonitoracaoIntegracaoComum monitoracaoIntegracao =
22             new MonitoracaoIntegracaoComum();
23         populaMonitoracaoIntegracao(monitoracaoIntegracao,
24             pagamentoBancario);
25         String json = ConversorJson.converteObjetoParaJson(monitoracaoIntegracao);
26         exchange.getIn().setBody(json);
27     }
28
29     private void populaMonitoracaoIntegracao(MonitoracaoIntegracao
30 Comum monitoracaoIntegracao, PagamentoBancarioComum
31 pagamentoBancario) {
32         monitoracaoIntegracao.setIdTransacao(pagamentoBancario
33 .getIdTransacao());
34         monitoracaoIntegracao.setCpfTitularCartao(pagamentoBancario
35 .getCpfTitularCartao());
36         monitoracaoIntegracao.setNumeroCartao(pagamentoBancario
37 .getNumeroCartao());
38         monitoracaoIntegracao.setDataInicialTransacao(pagamentoBancario
39 .getDataInicialTransacao());
40         monitoracaoIntegracao.setDataFinalTransacao(pagamentoBancario
41 .getDataFinalTransacao());
42         monitoracaoIntegracao.setStatus(pagamentoBancario.getStatus());
43     }
44 }
```

Esse processador inicialmente recupera o objeto **respostaSistemaAutorizacao** do body da mensagem e define no objeto da classe **PagamentoBancarioComum**, recuperado do header, alguns dados finais da transação. A partir desses objetos, popula um POJO, **PagamentoResposta**, para ser enviado ao componente **MobileIntegracao** e consequentemente ao Aplicativo Mobile.

Mesmo com o término da codificação dos processadores, ainda é necessário implementar os POJOS **MonitoracaoIntegracaoComum**, da biblioteca **MonitoracaoIntegracaoComum** (a ser criada em breve), e **PagamentoResposta**, da biblioteca **MobileIntegracaoComum**, além do componente **MonitoracaoIntegracao**. Será priorizada, nessa etapa, a continuação do código que completará o fluxo da transação de pagamento, isto é, que enviará a resposta da transação ao aplicativo mobile, uma vez que há necessidade de implementações em componentes distintos. Portanto, os parágrafos seguintes detalharão a continuação dessa implementação através da biblioteca **MobileIntegracaoComum**.

Dito isso, crie, no pacote **br.com.devmedia.mobile.integracao.comum**, da biblioteca **MobileIntegracaoComum**, a classe **PagamentoResposta** (veja a **Listagem 20**). Essa classe possui os atributos que serão enviados na resposta ao aplicativo mobile.

Listagem 19. Código da classe PopulaRespostaPagamentoBancarioProcessor.

```
01 package br.com.devmedia.pagamento.bancario.processor;
02
03 import java.text.SimpleDateFormat;
04 import java.util.Date;
05
06 import org.apache.camel.Exchange;
07 import org.apache.camel.Processor;
08 import org.springframework.stereotype.Component;
09
10 import br.com.devmedia.autorizacao.integracao.comum.RespostaSistema
11 Autorizacao;
12 import br.com.devmedia.mobile.integracao.comum.PagamentoResposta;
13 import br.com.devmedia.pagamento.bancario.comum
14 .PagamentoBancarioComum;
15 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
16
17 @Component
18 public class PopulaRespostaPagamentoBancarioProcessor implements Processor {
19
20     @Override
21     public void process(Exchange exchange) throws Exception {
22         RespostaSistemaAutorizacao respostaSistemaAutorizacao =
23             ConversorJson.converteJsonParaObjeto(exchange.getIn()
24 .getBody().toString(), RespostaSistemaAutorizacao.class);
25         PagamentoBancarioComum pagamentoBancario =
26             ProcessorUtil.getObjetoHeader(exchange, PagamentoBancarioComum
27 .class);
28         pagamentoBancario.setDataFinalTransacao(new Date());
29         pagamentoBancario.setStatus(respostaSistemaAutorizacao.getStatus());
30         ProcessorUtil.setObjetoHeader(exchange, pagamentoBancario);
31         PagamentoResposta pagamentoResposta = new PagamentoResposta();
32         populaPagamentoBancarioResposta(pagamentoResposta,
33             respostaSistemaAutorizacao, pagamentoBancario.getIdTransacao());
34         String json = ConversorJson.converteObjetoParaJson(pagamentoResposta);
35         exchange.getIn().setBody(json);
36     }
37
38     private void populaPagamentoBancarioResposta(PagamentoResposta
39 pagamentoBancarioResposta, RespostaSistemaAutorizacao
40 respostaSistemaAutorizacao, String idTransacao) {
41         pagamentoBancarioResposta.setIdTransacao(idTransacao);
42         pagamentoBancarioResposta.setStatus(respostaSistemaAutorizacao
43 .getStatus());
44         pagamentoBancarioResposta.setCodigoAutorizacao(respostaSistema
45 Autorizacao.getCodigoAutorizacao());
46         String dataAutorizacao = new SimpleDateFormat("yyyyMMddHHmmss")
47 format(respostaSistemaAutorizacao.getDataAutorizacao());
48         pagamentoBancarioResposta.setDataAutorizacao(dataAutorizacao);
49     }
50 }
```

Listagem 20. Código da classe PagamentoResposta.

```
01 package br.com.devmedia.mobile.integracao.comum;
02
03 import java.io.Serializable;
04
05 public class PagamentoResposta implements Serializable {
06
07     private static final long serialVersionUID = 2812416304574534589L;
08
09     private String idTransacao;
10     private String status;
11     private String codigoAutorizacao;
12     private String dataAutorizacao;
13
14     public PagamentoResposta() {}
15
16     // Getters e Setters dos atributos omitidos.
17
18     // Método toString() omitido.
19
20 }
```

Com o POJO pronto, é necessário incluir uma nova rota na classe **MobileIntegracaoRouteBuilder**, do componente **MobileIntegracao**, para receber a resposta da transação de pagamento enviada pelo componente **PagamentoBancario**, conforme expõe a **Listagem 21**.

Esse código define uma nova rota para consumir mensagens da fila **MobileIntegracao**. Baseado no conteúdo da mensagem, isto é, se a mensagem possuir em seu header o valor **RESPOSTA_AUTORIZACAO_PAGAMENTO_BANCARIO** na constante **FLUXO**, o processador **PreparaRespostaAutorizacaoPagamentoBancarioProcessor** será executado. Para isso, um predicado denominado **respostaAutorizacaoPagamentoBancario** é utilizado.

Após a mensagem ser manipulada pelo processador anterior, a resposta será enviada automaticamente ao Aplicativo Mobile, sem a necessidade da inclusão de uma DSL **to**, pois o cliente da requisição, ou seja, o Aplicativo Mobile, ainda está aguardando pela resposta, uma vez que a requisição efetuada é síncrona.

Na sequência, a classe para o processador **PreparaRespostaAutorizacaoPagamentoBancarioProcessor** deve ser criada no pacote **br.com.devmedia.mobile.integracao.processor** (vide **Listagem 22**). Basicamente, esse processador recupera o POJO **PagamentoResposta** enviado pelo componente anterior e o converte em formato JSON para que ele seja enviado no body da resposta do web service.

Listagem 21. Modificações na classe MobileIntegracaoRouteBuilder.

```
01 from("activemq:queue:MobileIntegracao?concurrentConsumers=5")
02   .id("rotaEntradaMobileIntegracao")
03   .choice()
04     .when(respostaAutorizacaoPagamentoBancario)
05       .log(LoggingLevel.INFO, "[MobileIntegracao]
          Enviando resposta de pagamento bancário")
06       .process(preparaRespostaAutorizacaoPagamentoBancarioProcessor)
07     .end();
```

Listagem 22. Código da classe PreparaRespostaAutorizacaoPagamentoBancarioProcessor.

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

Finalizada a codificação, o fluxo de resposta ao Aplicativo Mobile foi concluído com sucesso. Resta, no entanto, a implementação da biblioteca **MonitoracaoIntegracaoComum** e do componente **MonitoracaoIntegracao**.

Implementando os componentes MonitoracaoIntegracaoComum e MonitoracaoIntegracao

A implementação da biblioteca **MonitoracaoIntegracaoComum**, cuja finalidade é representar os dados que serão enviados ao sistema externo de monitoração, deve iniciar com a criação de um novo projeto/módulo do tipo Java Standalone (JAR) dentro do diretório do projeto agregador. Feito isso, crie nesse novo projeto um **pom.xml** com o código da **Listagem 23** para definir as configurações básicas da biblioteca.

Na sequência, deve-se adicionar o pacote **br.com.devmedia.monitoracao.integracao.comum** e o POJO **MonitoracaoIntegracaoComum** para representar os dados a serem enviados para o sistema externo de monitoração. A classe deve ser implementada em conformidade com a **Listagem 24**.

Listagem 23. Arquivo pom.xml da biblioteca MonitoracaoIntegracaoComum.

```
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>MonitoracaoIntegracaoComum</name>
15   <artifactId>monitoracao-integracao-comum</artifactId>
16   <packaging>jar</packaging>
17
18 </project>
```

Listagem 24. Código da classe MonitoracaoIntegracaoComum.

```
01 package br.com.devmedia.monitoracao.integracao.comum;
02
03 import java.io.Serializable;
04 import java.util.Date;
05
06 public class MonitoracaoIntegracaoComum implements Serializable {
07
08   private static final long serialVersionUID = 6095051667680770343L;
09
10   private String idTransacao;
11   private String cpfTitularCartao;
12   private String numeroCartao;
13   private Date dataInicialTransacao;
14   private Date dataFinalTransacao;
15   private String status;
16
17   public MonitoracaoIntegracaoComum() {}
18
19   // Getters e Setters dos atributos omitidos.
20
21   // Método toString() omitido.
22
23 }
```

Logo após, inclua essa nova biblioteca nas dependências do projeto **PagamentoBancario** e também na declaração de módulos do projeto agregador.

Agora, é a vez da criação do novo componente, chamado **MonitoracaoIntegracao**, responsável por enviar as mensagens com os dados das transações ao Sistema de Monitoração. Com o envio dessas mensagens, o EIP **Control Bus** será incluído na solução.

Para a construção do componente **MonitoracaoIntegracao**, é necessário criar uma estrutura de projeto Java Web (WAR) semelhante à do **PagamentoBancario**. Além disso, é preciso adicionar esse novo componente nos módulos do projeto agregador, ajustar o arquivo *pom.xml* para incluir em suas dependências as bibliotecas **ComponenteUtil** e **MonitoracaoIntegracaoComum**, assim como ajustar os arquivos *web.xml* e *application-context.xml*. O código completo está disponível para download na página da edição.

Finalizadas as configurações, deve-se criar o pacote **br.com.devmedia.monitoracao.integracao.rota** e incluir a classe de rotas **MonitoracaoIntegracaoRouteBuilder** conforme a **Listagem 25**.

Nessa classe, o código entre as linhas 18 e 23 define uma rota para receber mensagens da fila **MonitoracaoIntegracao**. Especificamente na linha 20, um processador do tipo **CriaPosicionalDadosTransacaoProcessor** é inserido para criar uma mensagem baseada no layout do requisito RF-11-SISMON, mensagem essa esperada pelo Sistema de Monitoração. Na sequência, na linha 22, é feito o envio da mensagem para a fila **SistemaExternoMonitoracao**, para que o sistema de monitoração receba a mensagem.

Para concluir, ainda é necessário desenvolver o processador **CriaPosicionalDadosTransacaoProcessor** (vide **Listagem 26**), responsável por gerar a mensagem em formato de texto (baseada no layout pré-definido nos requisitos) a ser enviada ao sistema de monitoração.

Esse código cria uma mensagem em formato **String** a partir dos dados contidos no POJO **MonitoracaoIntegracaoComum**.

Dessa forma, na linha 21 o POJO é recuperado através da mensagem recebida pelo processador. Na linha 22, é realizada a invocação do método auxiliar **criaPosicional(MonitoracaoIntegracaoComum)**, justamente para criar a mensagem em formato **String** (o código do método auxiliar é apresentado nas linhas 27 a 38). Por fim, o código da linha 24 faz a inclusão dessa nova mensagem no body da mensagem que será enviada ao sistema de monitoração.

Terminada essa codificação, os principais fluxos de negócio foram implementados e a Solução de Pagamento Bancário, construída com sucesso.

Listagem 25. Código inicial da classe **MonitoracaoIntegracaoRouteBuilder**.

```
01 package br.com.devmedia.monitoracao.integracao.rota;
02
03 import org.apache.camel.LoggingLevel;
04 import org.apache.camel.builder.RouteBuilder;
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.stereotype.Component;
07
08 import br.com.devmedia.monitoracao.integracao.processor.CriaPosicionalDadosTransacaoProcessor;
09
10 @Component
11 public class MonitoracaoIntegracaoRouteBuilder extends RouteBuilder {
12
13     private CriaPosicionalDadosTransacaoProcessor criaPosicionalDadosTransacaoProcessor;
14
15     @Override
16     public void configure() throws Exception {
17
18         from("activemq:queue:MonitoracaoIntegracao?concurrentConsumers=5")
19             .id("rotaEntradaMonitoracao")
20             .log(LoggingLevel.INFO, "[MonitoracaoIntegracao] Notifica transação de pagamento bancário")
21             .process(criaPosicionalDadosTransacaoProcessor)
22             .log(LoggingLevel.INFO, "[MonitoracaoIntegracao] para [SistemaExternoMonitoracao]")
23             .enviando dados da transação de pagamento bancário"
24             .to("activemq:queue:SistemaExternoMonitoracao?deliveryPersistent=false")
25             .end();
26     }
27
28     @Autowired
29     public void setCriaPosicionalDadosTransacaoProcessor(
30         CriaPosicionalDadosTransacaoProcessor criaPosicionalDadosTransacaoProcessor) {
31         this.criaPosicionalDadosTransacaoProcessor = criaPosicionalDadosTransacaoProcessor;
32     }
33 }
```

Listagem 26. Código da classe **CriaPosicionalDadosTransacaoProcessor**.

```
01 package br.com.devmedia.monitoracao.integracao.processor;
02
03 import java.text.SimpleDateFormat;
04
05 import org.apache.camel.Exchange;
06 import org.apache.camel.Processor;
07 import org.slf4j.Logger;
08 import org.slf4j.LoggerFactory;
09 import org.springframework.stereotype.Component;
10
11 import br.com.devmedia.monitoracao.integracao.comum.MonitoracaoIntegracaoComum;
12 import br.com.devmedia.pagamento.bancario.conversor.ConversorJson;
13
14 @Component
15 public class CriaPosicionalDadosTransacaoProcessor implements Processor {
16
17     private static final Logger LOGGER = LoggerFactory.getLogger(CriaPosicionalDadosTransacaoProcessor.class);
18
19     @Override
20     public void process(Exchange exchange) throws Exception {
21         MonitoracaoIntegracaoComum monitoracaoIntegracao = ConversorJson.converteJson
22             ParaObjeto(exchange.getIn().getBody().toString(), MonitoracaoIntegracaoComum.class);
23         String dadosTransacao = criaPosicional(monitoracaoIntegracao);
24         LOGGER.info("Mensagem enviada: " + dadosTransacao);
25         exchange.getIn().setBody(dadosTransacao);
26     }
27
28     private String criaPosicional(MonitoracaoIntegracaoComum monitoracaoIntegracao) {
29         StringBuilder dadosTransacao = new StringBuilder();
30         dadosTransacao.append(monitoracaoIntegracao.getLIdTransacao());
31         String dataInicialTransacao = new SimpleDateFormat("yyyyMMddHHmmss").format(monitoracaoIntegracao.getDataInicialTransacao());
32         dadosTransacao.append(dataInicialTransacao);
33         String dataFinalTransacao = new SimpleDateFormat("yyyyMMddHHmmss").format(monitoracaoIntegracao.getDataFinalTransacao());
34         dadosTransacao.append(dataFinalTransacao);
35         dadosTransacao.append(monitoracaoIntegracao.getCpfTitularCartao());
36         dadosTransacao.append(monitoracaoIntegracao.getNumeroCartao());
37         dadosTransacao.append(monitoracaoIntegracao.getStatus());
38     }
39 }
```

O resultado de todo esse processo pode ser conferido através das **Figuras 7 e 8**.

Esta série de artigos abordou em detalhes os conceitos, fundamentos, estrutura e funcionamento do framework para integração entre sistemas Apache Camel. Durante o estudo, um exemplo prático de uma solução de pagamento bancário foi apresentado, para que o entendimento da parte teórica fosse consolidado.

Dessa forma, ficou evidente que esse poderoso framework oferece muitas vantagens, benefícios e facilidades quando o cenário é integração de sistemas. Porém, assim como ocorre com qualquer outra tecnologia e framework, é necessário avaliar se a sua utilização é vantajosa. Para a adoção do Camel, é importante verificar se as tecnologias, plataformas e os formatos de dados que serão utilizados nas integrações são diferentes e, consequentemente,

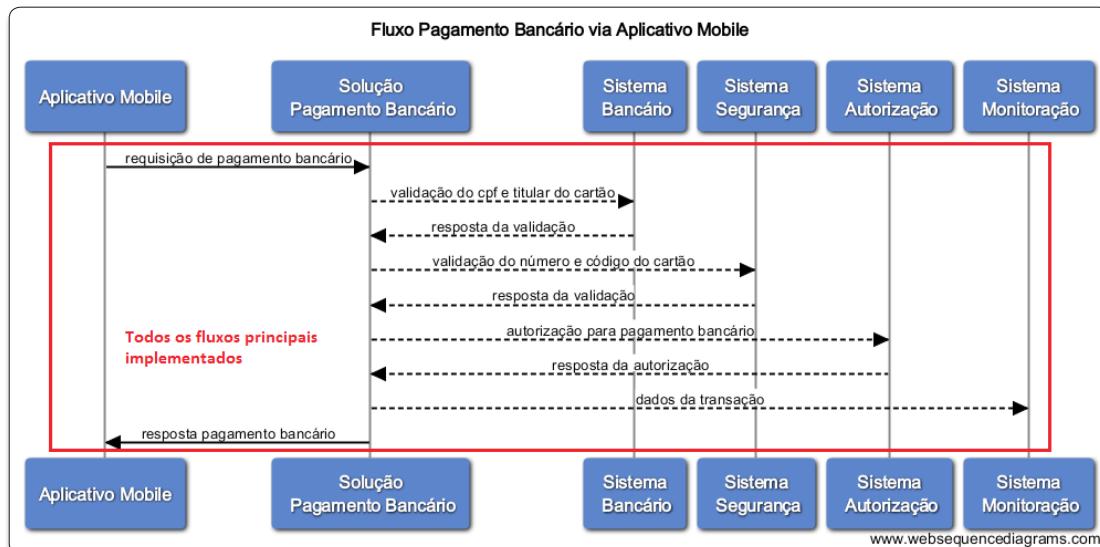


Figura 7. Fluxos de negócio implementados

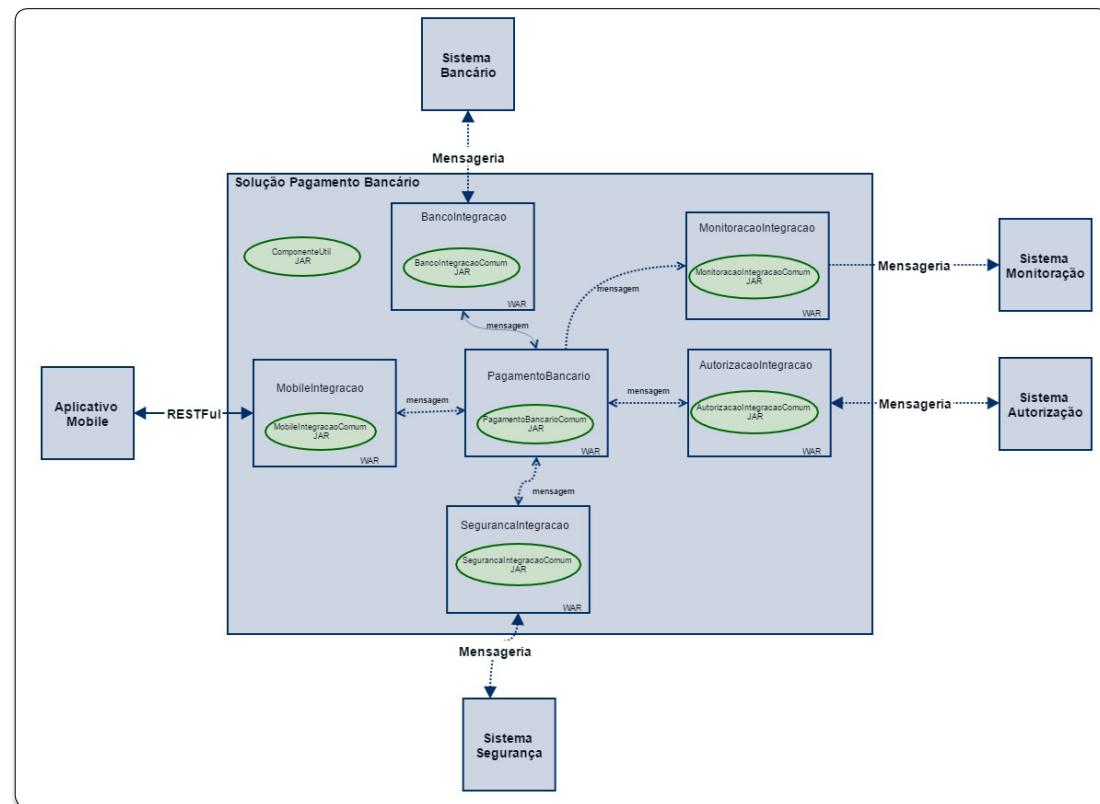


Figura 8. Diagrama final da arquitetura da Solução de Pagamento Bancário

um obstáculo e fonte de riscos para o projeto. Nesses cenários, a utilização do Camel é indicada.

Para cenários em que há pouquíssimas integrações e as tecnologias/plataformas são iguais, o uso de outras bibliotecas externas (frameworks), voltadas exclusivamente para aquele determinado tipo de integração, é mais vantajoso, resultando em um desenvolvimento mais rápido e fácil.

Já para projetos muito grandes, isto é, onde há muitos componentes e também integrações, o uso do Apache Camel pode ser substituído por um ESB, pois esse oferece, de forma nativa, muitas funcionalidades adicionais importantes para um cenário mais complexo.

Enfim, diante de tudo o que foi apresentado, pode-se afirmar que o futuro do Apache Camel é bastante promissor, pois dentro das empresas o número de sistemas com tecnologias distintas que necessitam de integração está aumentando. Ademais, o uso de aplicações, serviços nas nuvens e de aplicativos mobile será cada vez maior no mundo da TI, e nesses cenários, a adoção do Apache Camel para integração pode ser a solução ideal.

Autor



Rodrigo Cunha Santana

rodcunhasantana@gmail.com



Tecnólogo em Processamento de Dados pela FATEC de Americana/SP, com Especialização em Engenharia de Software pela Unicamp, MBA em Arquitetura de Software pelo IGTI e MBA em Gestão e Governança de TI pelo Senac de São Paulo. Trabalha com Java há oito anos, entusiasta do mundo ágil e apaixonado por qualidade de código. Possui as certificações OCJA 5/6, SCJP 6, OCWCD 5, CSD, CSP0 e CSM.

Links e Referências:

Site dos Padrões Empresariais de Integração (EIP).

<http://www.enterpriseintegrationpatterns.com>

Site do Apache Camel.

<http://camel.apache.org>

Site do Apache ActiveMQ.

<http://activemq.apache.org>

Site do Apache Maven.

<https://maven.apache.org/index.html>

Site do WildFly.

<http://wildfly.org>

Livro sobre Padrões Empresariais de Integração (EIP).

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

Livro Camel in Action.

IBSEN Claus; ANSTEY Jonathan. Camel in Action. Manning Publications, 2011.

Livro Apache Camel Developer's Cookbook.

CRANTON Scott; KORAB Jakub. Apache Camel Developer's Cookbook. Packt Publishing, 2013.

Livro Mastering Apache Camel.

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



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.

Supporte diferenciado

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

Serviços

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

1-to-1

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



Porta 80

WEB HOSTING

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

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

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