03

A primeira rota com Camel

Download do projeto inicial

Você pode fazer o DOWNLOAD do projeto inicial aqui (https://s3.amazonaws.com/caelum-online-public/camel/01/projeto-camel-alura.zip).

Pré-requisitos do treinamento

Para assistir ao treinamento com bom aproveitamento, você deve ter bons conhecimentos da linguagem Java e das tecnologias de integração: XML(principalmente), SOAP, HTTP e JMS. Aconselhamos ter assistido aos treinamentos anteriores dessa trilha.

Nós usaremos o Maven, apenas para administrar as dependências.

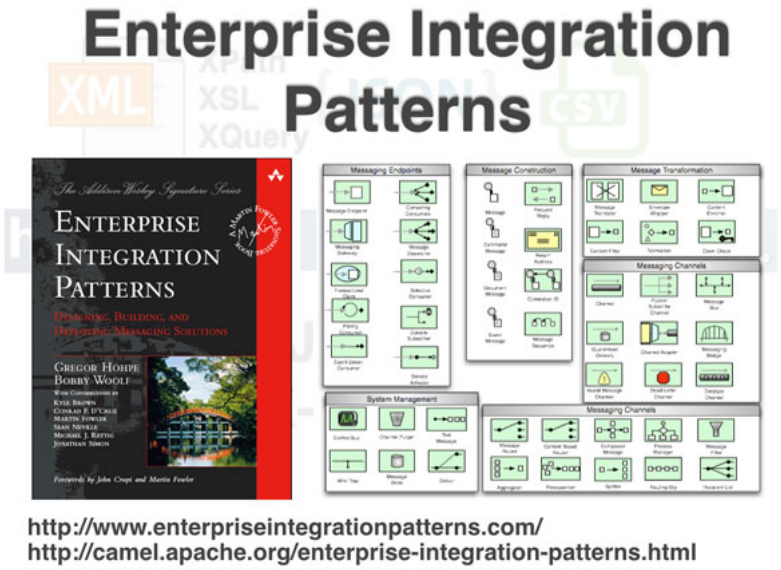
A complexidade da integração

Atualmente é difícil encontrar uma aplicação que funcione de maneira isolada, sem depender de nenhuma outra funcionalidade externa. Queremos reaproveitar funcionalidades já existentes, que é a ideia principal do SOA. Em tempos de Cloud e SOA, a integração entre aplicações se tornou muito comum e faz parte do cotidiano do desenvolvedor.

No entanto, a integração é uma tarefa árdua! Fazer com que as aplicações se comuniquem de maneira robusta e desacoplada é trabalhoso, pois não criamos as aplicações pensando na integração. Além disso, temos que trabalhar com plataformas, linguagens, formatos e protocolos diferentes, manter versões diversas, lidar com problemas da rede e falhas em geral, entre outras preocupações.

Boas práticas: Padrões de integração

Diante da necessidade de integrar aplicações, foram identificados alguns padrões de como resolver os problemas mais comuns na integração. Os **Enterprise Integration Patterns** definem uma série de boas práticas que foram documentadas no livro com o mesmo nome, que descreve as vantagens e desvantagens de cada padrão e define um vocabulário comum a ser seguido.



Segue o link para a página: http://www.enterpriseintegrationpatterns.com/ (http://www.enterpriseintegrationpatterns.com/)

O que é um framework de integração?

O Apache Camel (http://camel.apache.org/), como framework de integração, implementa a maioria dos padrões de integração. Um framework de integração ajuda a diminuir a complexidade e o impacto dessas integrações. Em vez de escrever código de integração na mão, usamos componentes para isso.

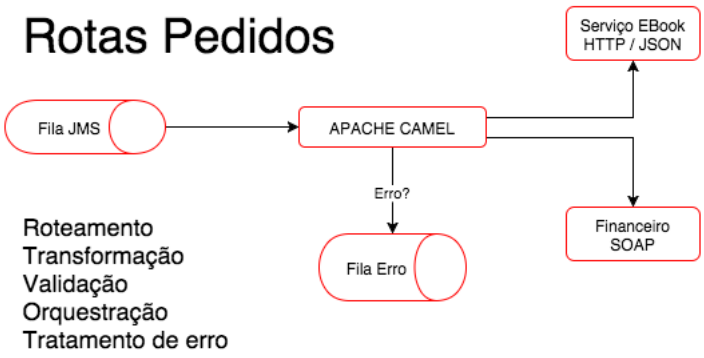
Com um framework de integração seguimos boas práticas que foram identificadas e amadurecidas ao longo do tempo. Apache Camel é o framework de integração mais famoso no mundo Java, mas não é a única opção. O Spring Integration (http://projects.spring.io/spring-integration/) é uma outra alternativa popular.

Roteamento entre endpoints com Apache Camel

Essencialmente, Camel é um roteador (*routing engine*), ou seja o Camel roteia os dados entre dois *endpoints*. Um *endpoint* é um serviço web ou um banco de dados, podendo ser um arquivo ou file JMS. Em geral, é um ponto onde pegamos ou enviamos dados. A tarefa do desenvolvedor é configurar, por meio de um Builder, os endpoints e as regras de roteamento. O desenvolvedor decide de onde vem as mensagens (`from()`), para onde enviar (`to()`) e o que fazer com a mensagem no meio desse processo (*mediation engine*).

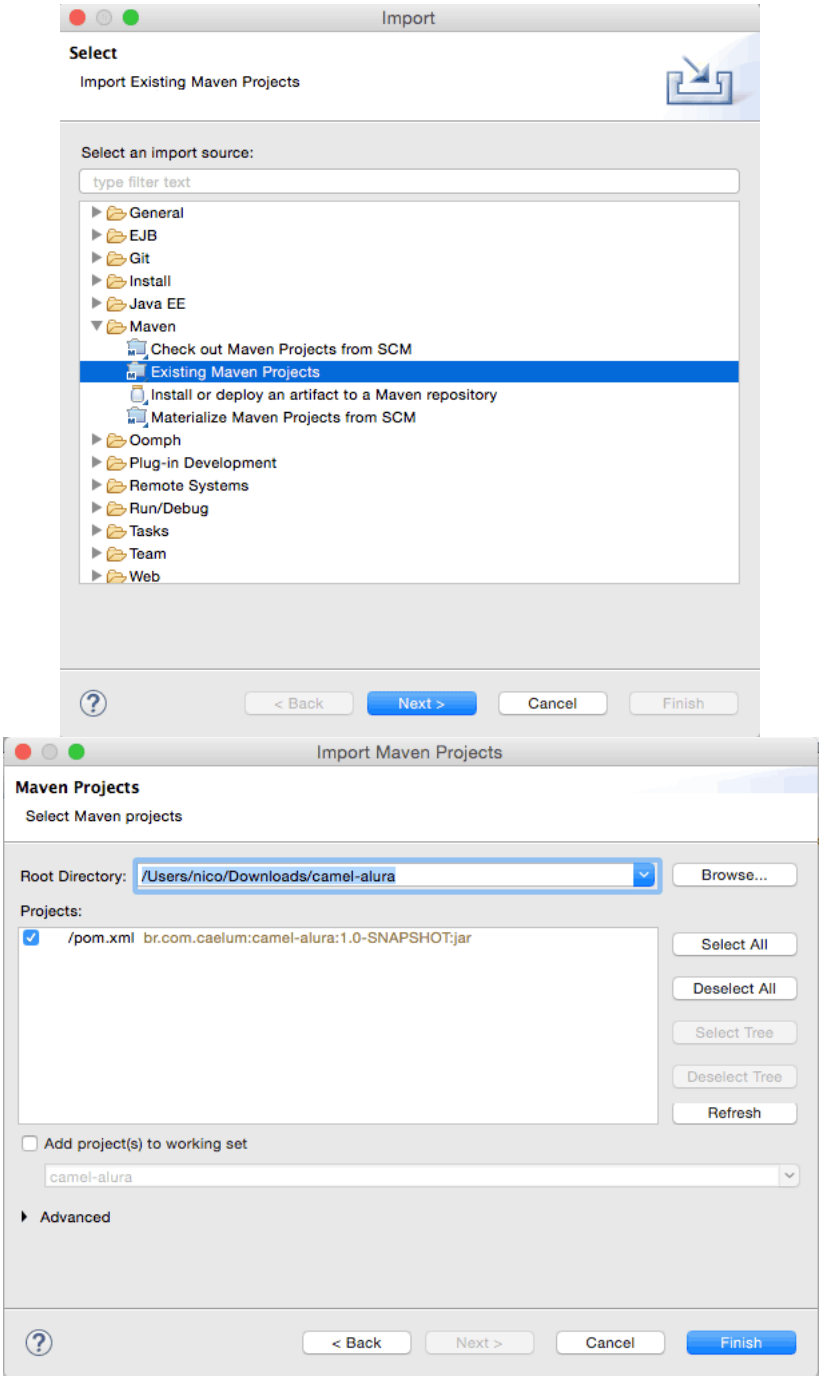
A rota de pedidos

No curso, implementaremos uma rota de pedidos. O objetivo é que uma loja virtual gere um pedido quando uma compra for efetuada, que precisará ser entregue para outros sistemas que recebam os dados em formatos diferentes. Nesta rota, vamos trabalhar com JMS, SOAP, HTTP e várias transformações!

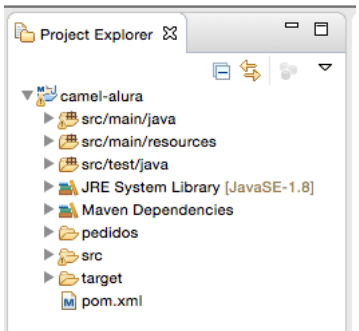


Inicializando Apache Camel

Vamos começar a usar Apache Camel e importar uma projeto inicial disponível aqui (https://s3.amazonaws.com/caelum-online-public/camel/01/projeto-camel-alura.zip). O projeto vem pré-configurado para Maven, ou seja, basta importar o projeto como *Maven Project* no Eclipse:



Atenção: A importação pode demorar pois o Maven precisa inicializar e baixar todas as dependências.



O projeto já possui alguns arquivos XML que usaremos durante as aulas.

Os nossos testes sempre serão executados dentro do método `main`, onde configuraremos a rota. Para tal é preciso criar um `CamelContext`. Abra a classe `RotaPedidos` existente que já inicialize o `CamelContext`:

```
public class RotaPedidos {

    public static void main(String[] args) throws Exception {

        CamelContext context = new DefaultCamelContext();

    }
}
```

Uma vez criado o `CamelContext`, podemos adicionar uma nova rota e configurá-la usando do `RouteBuilder` (atenção, não é `RoutesBuilder`):

```
    CamelContext context = new DefaultCamelContext();
    context.addRoutes(new RouteBuilder() {

            @Override
            public void configure() throws Exception {
                //aqui vem a rota!!
            }
    }
```

Usando a Camel DSL

Essa foi a parte burocrática do Camel, agora vem a parte mais interessante.

Dentro do método `configure` temos acesso à **Camel DSL**, uma linguagem especifica (baseada em Java) para configurar a rota! Para começar da forma mais simples possível, vamos ler alguns pedidos em XML de uma pasta e enviar para outra pasta. Para isso usaremos os métodos da Camel DSL, o método `from(..)` e `to(..)`. Os métodos recebem uma string que definem o *endpoint*, ou seja, o lugar concreto na integração de onde pegamos os dados ou enviamos. Veja como fica o código usando a Camel DSL:

```
public void configure() throws Exception {
    from("file:pedidos"). //aqui tem um ponto para encadear a chamada do próximo método
    to("file:saida");
}
```

Nessa rota, definimos que queremos ler os arquivos da pasta `pedidos` e enviar para a pasta `saida`. Usamos o primeiro componente do Camel, o componente `file`. Ele é bastante simples e sempre começa com `file`, seguido por `:` e o nome da pasta.

Ao executar, perceberemos que ainda não acontece nada. Pois é, é preciso ainda apertar o botão "start()" e esperar um pouco para o Camel fazer o trabalho (`Thread.sleep(..)`).

Veja o código completo:

```
public class RotaPedidos {

    public static void main(String[] args) throws Exception {

        CamelContext context = new DefaultCamelContext();
```

```
        context.addRoutes(new RouteBuilder() {

            @Override
            public void configure() throws Exception {
                from("file:pedidos"). //aqui tem um ponto para encadear a chamada do próximo mét
                to("file:saida");
            }
        }

        context.start(); //aqui camel realmente começa a trabalhar
        Thread.sleep(); //esperando um pouco para dar um tempo para camel
    }
});
```

Pode executar! O resultado ainda não é um dos mais impressionantes pois o Camel apenas moveu os arquivos de uma pasta para outra.

A Camel Expression Language

Vamos mostrar que o Camel realmente está trabalhando e adicionar um passo intermediário entre `from()` e `to()`. Vamos utilizar o componente *log* que o Camel disponibiliza usando um método com o mesmo nome:

```
public void configure() throws Exception {
    from("file:pedidos").
    log("Camel trabalhando!!").
    to("file:saida");
}
```

No componente `log`, podemos aproveitar uma linguagem simples, muito parecido com a *Expression Language* do mundo JSP. Com ela, podemos imprimir informações sobre os dados que o Camel está usando. Quando o Camel lê os dados de algum *endpoint*, ele cria internamente uma **mensagem**. Essa mensagem possui uma *ID* e um *Body* que são trafegados na rota definida. Podemos acessar a `id` e `body` pela *Expression Language* (EL) do Camel:

```
public void configure() throws Exception {
    from("file:pedidos").
    log("${id} - ${body}").//usando EL
    to("file:saida");
}
```

O *id* é automaticamente gerada e o *body*, no caso, é o conteúdo de cada arquivo XML.

Personalizando o trabalho de um componente

A maioria dos componentes podem ser configurados por meio de parâmetros. Com o componente `file`, isso não é diferente. Os parâmetros são usados na Camel DSL com a mesma sintaxe de parâmetros de uma requisição HTTP, por exemplo:

```
public void configure() throws Exception {
    from("file:pedidos?delay=5s").
```

```
        log("${id} - ${body}").
        to("file:saida");
    }
```

Repare que usamos `?delay=5s`. Definimos que o componente `file` deve verificar a pasta `pedidos` a cada 5 segundos. Se quisermos configurar mais um parâmetro, usamos o caracter `&`, por exemplo:

```
public void configure() throws Exception {
    from("file:pedidos?delay=5s&noop=true").
    log("${id} - ${body}").
    to("file:saida");
}
```

O parâmetro `noop=true` significa que os arquivos não serão apagados da pasta `pedidos`, algo útil para nossos testes.

Atenção: Você pode testar o esse código nesse instante mas antes disso você deve copiar os arquivos XML da pasta `saida` para `pedidos`.

Encontrando as configurações

Talvez você se pergunte como descobrir quais são os parâmetros de um componente. A documentação do Camel é excelente e não só mostra todos os parâmetros existentes como também vários exemplos úteis:

http://camel.apache.org/file2.html (http://camel.apache.org/file2.html)

Já vimos bastante conteúdo nesse capítulo e chegou a horas dos exercícios! No próximo capítulo vamos analisar o conteúdo da mensagem para filtrá-la e dividir o conteúdo, sempre seguindo as boas práticas dos padrões de integração.

##O que aprendemos?

- O que é um framework de integração
- Camel é essencialmente uma *routing-engine*
- Camel segue boas práticas por meio do Padrões de Integração
- Devemos criar um *rota* usando o `RouteBuilder`
- A Camel DSL é utilizada para configurar a rota de alto nível
- Os métodos `from(..)` e `to(..)` definem os enpoints
- Por meio da *Camel Expression Language* podemos acessar a mensagem na rota