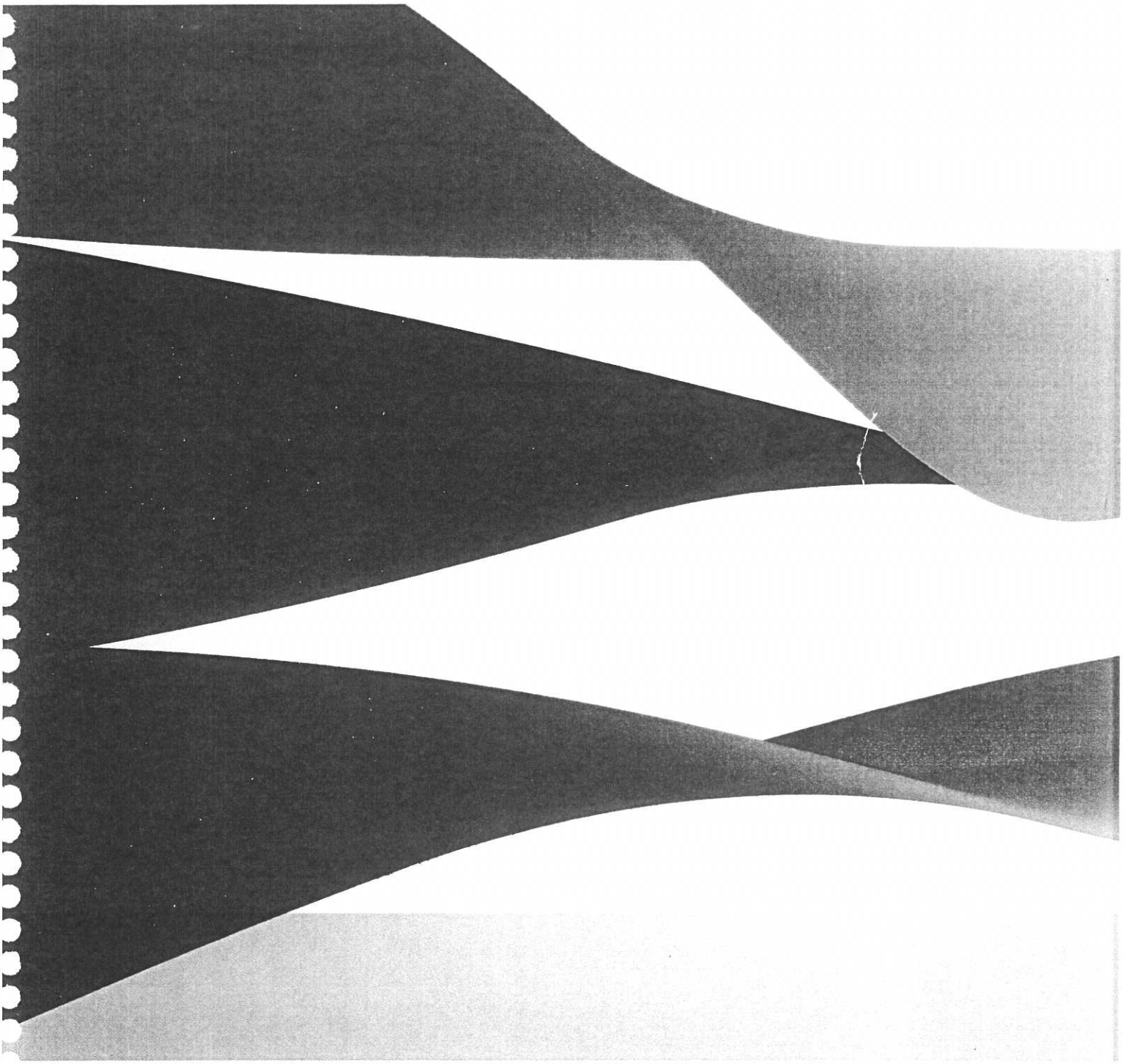


Process Engineer

soa|expert

soa|expert

ÍNDICE PÁGINA

1. Introdução a BPEL 2
2. BPEL no contexto de Serviços Web 3
3. A "DSL" de Máquina de estados 3
4. Processos Abstratos e Processos Executáveis 4
5. Controle de fluxo em BPEL 6
6. A Linguagem 7
7. Sintaxe BPEL 8
8. Manipuladores 14
9. Manipulação de dados em BPEL 15
10. Instâncias de processos: Identificação e ciclo de vida 17
11. Exempo de especificação de processo 17
12. Motores e Aplicações para BPEL 20
12.1 Apache ODE 20
12.2 ActiveVOS 24
12.2.1 ActiveBPEL 25
13. Exemplos de Processos de Negócio 30
14. Ordem de encomenda 36
15. Marcação de Viagem 43
16. Conhecendo o Oracle SOA 48
17. Criando um projeto simples 49
18. Exercício de chamada síncrona 55
19. Exercício de chamada Assíncrona 66
21. Trabalhando com o adaptador de Arquivos 68
22. Trabalhando com WorkFlow Humano 72
23. Correlation ID 76
24. Service Component Architecture 84
25. Assembly Model 85
26. Fases do projeto de aplicações SCA 89
27. Exercícios de SCA 91
28. BAM – Business Analysis Monitoring 95

Confidencial – Todos os Direitos Reservados para SOAExpert.

1. Introdução à BPEL

História do BPEL

Business Process Execution Language (BPEL) é uma linguagem utilizada para definir processos de negócio no contexto de serviços web. Ela é o resultado da convergência das idéias presentes em duas linguagens pré-existentes: **WSFL** (Web Services Flow Language) da IBM e **XLANG** (Web Services for Business Process Design), da Microsoft.

BPEL se propôs a unir a estruturação hierárquica de atividades presente na linguagem XLANG com o controle baseado em grafos da WSFL. Assim, em julho de 2002, a Microsoft e a IBM, conjuntamente com a BEA Systems, lançaram a especificação da linguagem que batizaram com o nome **BPEL4WS 1.0** (Business Process Execution Language for Web Services).

Em abril de 2003, essas empresas, juntamente com a SAP e Siebel Systems, lançaram uma segunda versão pública da especificação da linguagem; a ela deram o nome de **BPEL4WS 1.1**. Essa especificação foi submetida à organização denominada **OASIS** (*Organization for the Advancement for Structured Information Standards)*, um consórcio global que dirige o desenvolvimento, a convergência e a adoção de padrões ligados à área de serviços web e comércio eletrônico.

Com tal atitude, buscava-se obter um padrão aberto e ainda uma aceitação maior na indústria. Com seu amadurecimento, a especificação passou por várias mudanças significativas e, em setembro de 2004, o comitê responsável pela linguagem na OASIS resolveu mudar o nome da especificação para **WS-BPEL 2.0** (Web Services Business Process Execution Language). Em setembro de 2006, essa especificação ficou disponível para revisão pública, ficando neste estado até abril de 2007, data em que foi lançada como um padrão OASIS. Apesar das diversas versões e modificações feitas no decorrer do processo, as diretrizes básicas da linguagem permaneceram as mesmas.

2. BPEL no contexto de serviços Web

BPEL é uma linguagem baseada em XML, desenvolvida para ser utilizada dentro do contexto de serviços web.

Um processo descrito em BPEL é construído baseado no modelo de serviço definido pela especificação WSDL 1.1. Com isso, um processo BPEL se comporta como um serviço web, descrito por meio de uma especificação WSDL. Todas as interações externas realizadas pelo processo ocorrem por meio de interfaces WSDL de serviços web. Um processo de negócio escrito em BPEL define como coordenar as interações entre as diversas instâncias do processo e os serviços web com os quais se comunicam. Assim, uma instância de processo sabe quando deve mandar ou receber uma mensagem. Entretanto, para realizar efetivamente a comunicação, recorre-se às interfaces WSDL disponibilizadas pelos diversos serviços web.

Uma especificação de processo BPEL separa as referências concretas aos serviços reais das representações abstratas das interações entre os serviços. Para isso, utiliza os recursos de abstração providos pelo modelo WSDL (como **port types** e **operations**). A desvinculação entre o comportamento do processo e o ambiente de execução dá ao processo BPEL a característica de ser reutilizável e simples, retirando da especificação as restrições desnecessárias.

"O fato de um processo BPEL ficar disponível como um serviço web traz ainda o caráter modular ao processo, pois um serviço web pode ser chamado de dentro de outro."

3. A "DSL" de Máquina de estados

O BPEL também pode ser enxergado como uma DSL (domain-specific language) para controle de estado entre as chamadas aos serviços.

Esse controle de estado, é conhecido como "Orquestração" o qual armazena "estado" do num processo (stateful), guardando valores e repassando como parâmetro a outros serviços, se necessário.

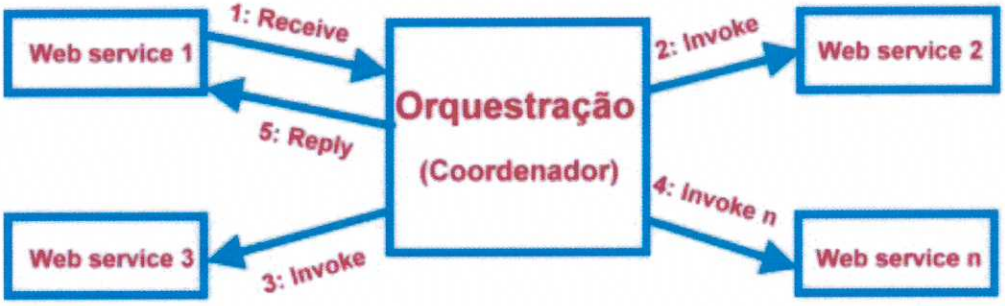


Imagem: Retrata a máquina de estado.

Um erro muito comum é enxergar a Coreografia como "Orqustração" e vice-versa.

Uma Coreografia não precisa controlar o seu estado, logo com utilizar o BPEL para tal, não faria sentido.

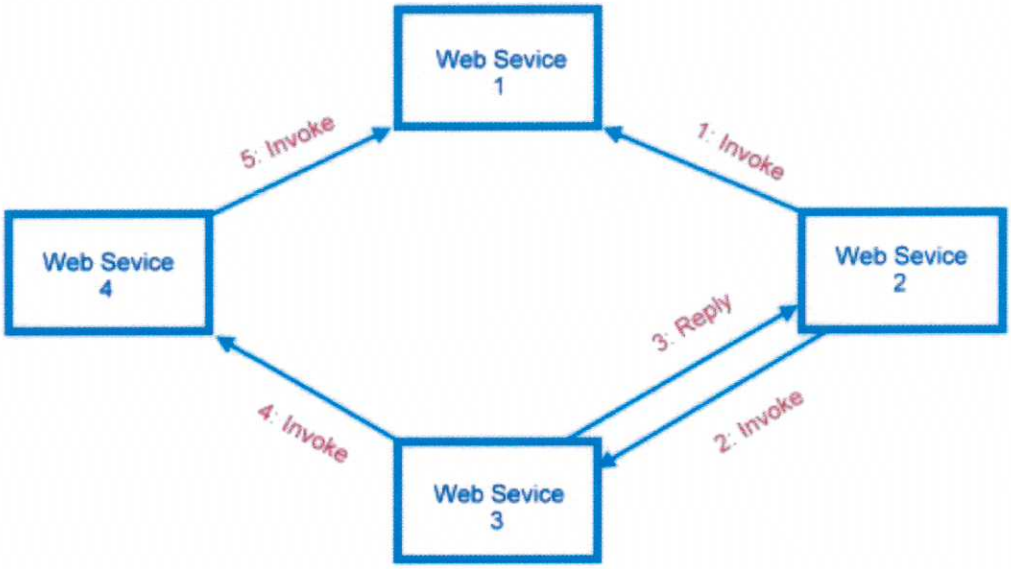


Imagem: Exemplo de coreografia

4. Process abstratos e processos executáveis

Em BPEL, existem duas maneiras de descrever processos de negócio: por meio de *processos executáveis* (executable processes) e por meio de *processos abstratos* (abstract processes).

Um processo executável "define a ordem de execução de um conjunto de atividades, as partes envolvidas no processo, as mensagens trocadas entre elas, bem como os eventos e tratamento de exceções, especificando qual o tratamento quando eventos específicos ou falhas ocorrem".

Um processo executável pode existir independentemente de um processo abstrato. O seu objetivo é representar todos os detalhes de um processo **BPEL (**negócio), possibilitando a sua execução.

Já um processo abstrato não é especificado para execução. "Um processo abstrato é uma especificação do comportamento de uma classe de serviços, que captura restrições quanto à ordenação das mensagens que são enviadas para um serviço ou recebidas de um serviço".

Assim, processos abstratos descrevem as interfaces públicas e as regras associadas a essas interfaces.

Nota-se que, diferentemente do arquivo WSDL, que somente descreve as interfaces utilizadas, no processo abstrato é possível descrever a seqüência das comunicações ou a dependência entre as mensagens.

Existem vários casos de uso em que o processo abstrato se aplica. Entre eles estão a validação de um processo, garantindo que ele obedeça a uma especificação estabelecida; a função de prover um modelo que auxilie no desenvolvimento da implementação de um processo que seja executável; ou ainda a possibilidade de divulgar aspectos públicos de um processo sem que, para isso, seja necessário revelar decisões internas para os utilizadores do processo.

5. Controle de fluxo em BPEL

A união de dois paradigmas

O modo como BPEL lida com o controle de fluxo de seus processos está intimamente relacionado à maneira como suas linguagens predecessoras, XLANG e WSFL, tratam o assunto. Elas utilizam diferentes paradigmas para estabelecer o controle dos fluxos. WSFL utiliza uma estrutura de grafos, aliada a algumas condições, para controlar o fluxo de seus processos. Já XLANG utiliza uma estrutura hierárquica formada por "blocos" que representam conceitos especiais, como **while** e **sequence**, lembrando bastante uma linguagem de programação usual. Dependendo do tipo de processo que se quer modelar, uma das duas abordagens se mostra mais natural que a outra.

BPEL foi desenvolvida de modo a combinar as duas linguagens, tentando extrair o melhor delas, facilitando a naturalidade da representação.

Um processo BPEL relaciona diversas atividades, que podem ser divididas em duas categorias: **atividades básicas** e **estruturadas**.

Atividades básicas são aquelas que descrevem ações atômicas do comportamento do processo. Já as atividades estruturadas são aquelas que envolvem lógica de controle de fluxo, podendo conter outras atividades dentro delas, sejam básicas ou estruturadas, de um modo recursivo.

As atividades estruturadas impõem regras de comportamento e restrições sobre a execução das atividades que elas contêm, ditando a ordem com que essas atividades serão executadas. Juntamente às atividades, BPEL também inclui os conceitos de eventos, falhas, compensações e terminações, que possuem seus respectivos **manipuladores**.

6. A linguagem

A programação com **BPEL** é parecida com a programação das linguagens de programação já existentes, na medida que ela oferece determinados tipos de construções como estruturas de repetição, estruturas condicionais, variáveis e atribuições. Este fato possibilita, que o **Processo de Negócio** seja visto como um algoritmo. Porém, o uso desses recursos é bem simples e limitado, objetivando a facilidade de uso e aprendizagem da mesma.

Uma composição de serviços especificada em BPEL é vista como serviço, ou seja, quem usou um processo de negócio definido com **BPEL** não tem a ciência que na verdade está usando vários serviços para realizar uma determinada operação. Por ser um Serviço Web, a composição necessita ter sua interface baseada no padrão WSDL, troca de mensagens com outros Serviços Web através do padrão SOAP e pode ser cadastrado em um repositório UDDI para fins de consulta.

O padrão BPEL assume que os serviços participantes da composição estejam descritos usando WSDL.

Um processo BPEL, assim como um serviço, pode ser síncrono ou assíncrono. Um processo BPEL síncrono bloqueia o cliente (aquele que está usando o processo) até que o processo termine e retorne um resultado. O processo assíncrono não bloqueia o cliente. Em vez disso, ele usa um retorno de chamada para retornar o resultado (se houver).

Geralmente, os processos assíncronos são usados para processos de longa duração, e os síncronos são usados para processos que retornam um resultado em um tempo relativamente curto. Se um processo BPEL usa serviços Web assíncronos, o próprio processo geralmente também é assíncrono.

A propriedade mais importante de BPEL está relacionada à invocação de Serviços Web. BPEL permite que as operações de um Serviço Web sejam invocadas sincronamente ou assincronamente de maneira mais fácil. Estas operações podem ser invocadas seqüencialmente ou paralelamente.

7. Sintaxe BPEL

O código fonte de BPEL é escrito em XML, com cada função e componente (denominados por "atividades") escrito sequencialmente, sendo cada uma identificada pela expressão "**bpel:**", que a identifica como uma expressão desta linguagem, e não XML normal.

Cada linha de código é delimitada pelos sinais de menor e maior, "<" e ">".

Se a linha de código seja única, isto é, não tenha um domínio associado, possui um carácter que marca o final, "/", antes do sinal de ">" (**<bpel:link name="L1"/>**);

Caso possua um domínio, a linha inicial possui o tipo de atividade a que se refere, limitada pelos sinais normais (**<bpel:sources>**), sendo a linha final do domínio identificada com a mesma expressão da primeira, mas precedida pelo carácter "/" (**</bpel:sources>**). Tudo o que se encontra entre estas duas expressões pertence ao seu domínio.

```
<bpel:sequence>
   <bpel:assign name="IncrementTotal">
      <bpel:copy>
         <bpel:from>number( $orderMessage.order/OrderDetail[ $detailIndex ]/QTY )</bpel:from>
         <bpel:to variable="itemQuantity"/>
      </bpel:copy>
      <bpel:copy>
         <bpel:from>number( $orderMessage.order/OrderDetail[ $detailIndex ]/Cost )</bpel:from>
         <bpel:to variable="itemCost"/>
      </bpel:copy>
      <bpel:copy>
         <bpel:from>($orderTotal + ($itemQuantity * $itemCost))</bpel:from>
         <bpel:to variable="orderTotal"/>
      </bpel:copy>
   </bpel:assign>
   <bpel:assign name="IncrementDetailIndex">
      <bpel:copy>
         <bpel:from>($detailIndex + 1)</bpel:from>
         <bpel:to variable="detailIndex"/>
      </bpel:copy>
   </bpel:assign>
</bpel:sequence>
```

Um processo BPEL é um conjuntode atividades ordenadas que começa por uma instrução de ***receive*** ou ***pick*** (uma vez que, sendo o processo de negócio ele mesmo um WebService, a sua execução só começa quando é invocado).

O BPEL possui uma sintaxe com vários tipos de atividades, cuja conjunção consegue descrever todo o tipo de orquestração se necessite. Na tabela seguinte são mostradas essas atividades, assim como uma pequena descrição.

Instrução	Ação
`<invoke>`	Chama um serviço externo de forma síncrona.
`<empty>`	É uma atividade "sem operação", útil para sincronizar atividades concorrentes.
`<assign>`	Manipular variáveis de dados e referências de destino de *partner links*, criando operações de cópia (*copy to/from*) para estes.
`<suspend>`	Suspende um processo em execução. Útil numa atividade de *catch* ou *catchAll* para "apanhar" erros inesperados.
`<receive>`	Recebe entradas de serviços assíncronos.
`<wait> <until>..`	Pausa o fluxo do processo de negócio durante um determinado período de tempo.
`<exit>`	Pára um processo de negócio.
`<validate>`	Valida os valores de variáveis em relação à concordância com o tipo de dados.
`<continue>`	Termina a iteração do ciclo (*while*, *forEach*, *repeatUntil*) em que foi usada e passa para a próxima.
`<break>`	Faz com que o ciclo actual (*if*, *while*, *forEach*) termine.
`<throw>`	Um meio de lidar com falhas durante o a execução do processo, sinalizando as que ocorrem no seu *scope*.

<rethrow>	assa para o *scope* em que está inserida as falhas que são "apanhadas" pelo *fault handler* local.
<compensate>	Compensa todos os processos internos de um *scope* que já terminaram, quando num deles ocorre um erro, revertendo essas atividades.
<compensateScope>	Faz a compensação dos processos internos de um *scope* alvo.
<opaque>	Apenas utilizada em processos abstratos (não executáveis), substituindo nestes uma atividade que seria utilizada num processo executável.
<sequence>	Permite definir uma série de atividades que serão invocadas numa sequência ordenada.
<flow>	Executa todas as atividades dentro do seu *scope* em paralelo.
<if>	Executa uma atividade baseando-se numa ou mais condições definidas no elemento "*if*" e nos elementos adicionais "*else if*" e "else". As condições são analisadas por ordem, sendo que a primeira que se verificar terá a atividade correspondente executada.
<while>	Executa uma atividade repetidamente, enquanto a condição que lhe está associada for verdadeira.
<pick>	Contém pelo menos uma mensagem ou parte de mensagem e um ou mais alarmes. Quando é executada, esta atividade bloqueia o processo até que a mensagem seja recebida ou, caso contrário, um dos alarmes execute.
<scope>	Providencia um contexto para um conjunto de atividades.
<repeatUntil >	Executa uma atividade repetidamente até que a sua condição seja verdadeira.
<forEach>	As atividades contidas no seu *scope* são executadas um determinado número de vezes, em

	sequência ou em paralelo. O número de vezes é dado por uma expressão, que vai correr N iterações, em que N varia de um valor inicial a um valor final.
<elseif> <else>	Definem condições extra num ciclo *if*. Podem existir vários *elseIf* por ciclo, e um *else*, normalmente usado como a escolha por defeito.
<catch>	Um *fault handler*, que define no seu scope as atividades a executar caso uma falha com um nome e/ou tipo especifico ocorra.
<catchAll>	Um *fault handler* que executa quando uma atividade de *throw* não é apanhada por nenhum *catch.*
<onEvent>	Indica que o evento especificado no seu *scope* está à espera de uma mensagem para ser despertado, quer esta seja de uma atividade *receive/reply* ou operação de sentido único (*one-way*).
<onAlarm>	Define um evento temporal de *timeout*, quer através de uma duração, quer através de um limite (*deadline*), despertando um alarme quando o *timeout* ocorre.
<onAlarm>	Define um conjunto de atividades a executar aquando da recepção de uma mensagem.

Resumo das atividades do BPEL

6. Conceitos Fundamentais

As funcionalidades mais importantes do BPEL :

- Descrever a lógica dos processos de negócio através da composição de serviços;
- Tratar as invocações de operações de modo síncrono ou assíncrono;

- Invocar serviços de maneira seqüencial ou paralela.
- Prover mecanismos para tratar erros durante a invocação de serviços;
- Manter atividades de longa duração (transacionais), assim como interrompe-las;
- Reiniciar uma composição que foi interrompida ou apresentou erros;
- Agendar a execução de atividades e definir a ordem em que elas irão executar.

O BPEL possui elementos fundamentais que podem ser utilizados tanto para especificação de processos de negócio, quanto para **protocolos de integração.**

A estrtuura geral de um Processo de Negócio, é formada por quatro seções:

- ***PartnerLinks***: define os diferentes parceiros (partes envolvidas) que interagem com o processo durante toda sua execução. Eles são usados para identificar a funcionalidade que deve ser oferecida por cada serviço parceiro. As ligações de parceiros (***partner link***) devem estar associadas a um tipo de ligação entre parceiros (***partner link type***) definidos na especificação de serviços - WSDL;

- ***Variables***: define as variáveis de dados usadas pelo processo. As definições são feitas em termos de **tipos de mensagem** WSDL, elementos ou tipos simples de schemas XML. Elas são usadas para manter os dados de estado e o histórico do processo com base nas mensagens trocadas. As variáveis devem estar associadas a tipos de mensagens (***messages***) definidos na especificação WSDL;

- ***FaultHandlers***: seção que contém o controle de falhas e definem

as atividades a serem executadas em resposta às falhas resultantes da invocação de serviços;

- ***Activities***: contém a descrição do comportamento normal para a execução do processo. Existem basicamente dois tipos de atividades:
  - ***Basic Activity***: tipo de atividade usado para executar alguma operação. Algumas dessas atividades básicas envolvem a interação com algum parceiro, as quais são: ***invoke***, ***receive*** e ***reply***. Outras dessas atividades básicas são executas sem a interação com qualquer parceiro, as quais são: ***wait***, ***terminate***, ***assign***, ***empty***, ***throw*** e ***compensate***;

  - ***Structured Activity***: tipo de atividade usado para agrupar atividades básicas (*basic activities*) dentro de algumas estruturas de fluxo. Tais atividades são: ***while***, ***pick***, ***flow***, ***sequence***, ***switch*** e ***scope***.

Tabela 1 – Primitivas básicas usadas em uma composição de serviço

PRIMITIVA	SIGNIFICADO
<invoke>	Invoca um Serviço Web
<receive>	Aguarda a resposta de um cliente
<reply>	Gera resposta síncrona
<assign>	Manipula dados
<throw>	Indica faltas ou exceções
<wait>	Espera por certo tempo
<terminate>	Finaliza um processo
<sequence>	Seqüência de atividades a serem invocadas
<flow>	Usada para definir um conjunto de atividades que podem ser invocadas em paralelo
<if>	Seleção
<while>	Repetição
<pick>	Aguarda por um evento

8. Manipuladores

Em BPEL, existem quatro tipos de manipuladores. São os manipuladores de **eventos**, **falhas**, **compensações** e **terminações**.

Os manipuladores de eventos são formados por pares evento-atividade. Eles são considerados habilitados quando seu **escopo** associado está em execução.

Quando a ocorrência de um evento descrito é registrada, a atividade associada ao evento é, então, disparada e executada concorrentemente à atividade principal. Pode ser considerado evento o término de um período de tempo, funcionando como um alarme, ou ainda uma mensagem externa.

Os manipuladores de falhas são utilizados pela linguagem para tratar as falhas internas e externas que podem acontecer durante a execução de atividades contidas em um escopo.

Existem maneiras de gerar falhas de modo explícito, por meio da utilização de atividades específicas (atividade **throw**). Diferentemente do que ocorre nos manipuladores de eventos, aqui o fluxo principal é interrompido antes que a atividade especificada no interior do manipulador seja executada.

Os manipuladores de compensações são um modo de desfazer as atividades de um escopo que já foi executado. Quando uma atividade compensatória (atividade **compensate** e **compensateScope)** é executada para um determinado escopo, o manipulador de compensações desse escopo será executado (caso exista), disparando a execução da atividade especificada no seu interior. Essa é a maneira existente em BPEL de reverter o que foi feito em partes do processo que necessitam de compensação.

Quando um manipulador de falhas de um escopo é acionado, todas as atividades ativas do fluxo principal desse escopo devem ser finalizadas antes que o manipulador comece a tratar a falha.

Algumas atividades são simplesmente interrompidas, terminando prematuramente, como é o caso de atividades do tipo **wait**, **receive**, **reply** e **invoke**. Já outras têm permissão para acabar o que estão

fazendo (**assign**, por exemplo). No caso de atividades estruturadas, ocorre a interrupção e a propagação da finalização para as atividades aninhadas que estão em execução. Para algumas atividades, existe a possibilidade (ou obrigatoriedade) de se aplicar uma rotina de terminação ao final (forçado) da sua execução. Nesse contexto entra o *manipulador de terminação*.

Os manipuladores de terminação permitem determinar a terminação das atividades. Para isso, é necessário associar, ao escopo devido, uma atividade usada pelo manipulador de terminação. Depois que a atividade primária do escopo é finalizada, a atividade do manipulador é executada. Caso não exista, é executada uma terminação padrão.

9. Manipulação de dados em BPEL

A manipulação de dados em BPEL é feita com a intuito de influenciar o fluxo dos processos e tratar os dados provenientes das mensagens recebidas ou enviadas.

BPEL parte do princípio de que ela não deve ser utilizada como uma linguagem própria para tratamento de dados, pois espera que os dados sejam tratados e armazenados dentro dos serviços web que o processo BPEL utiliza. Assim, ela provê funções limitadas para este fim, com o intuito de servir apenas para manipular os dados relevantes para o controle de fluxo e possibilitar a passagem de dados via mensagens.

Para exercer as suas funções, BPEL utiliza vários recursos. Dentre eles estão as variáveis e as expressões. O uso de variáveis BPEL ocorre da mesma maneira que na maioria das linguagens de programação. Elas po- dem guardar dados relacionados ao estado do processo, guardar valores temporários, ser parte de expressões, ou ainda armazenar mensagens recebidas ou que serão enviadas.

BPEL utiliza três tipos de declaração de variáveis: **WSDL message type**, **XML Schema type** e **XML Schema element**.

Cada variável é declarada dentro de um **escopo**, pertencendo, assim, a ele.

Uma variável declarada dentro do escopo global do processo pode ser vista em todo o processo, sendo chamada de *variável global*.

Uma variável declarada em algum escopo que não seja global, será visível somente no escopo na qual foi definida, ou nos escopos definidos internamente. Nesse caso, ela recebe o nome de *variável local*.

O acesso às variáveis é feito do escopo mais restrito para o mais abrangente, da mesma forma que ocorre na maioria das linguagens de programação.

As variáveis globais não são iniciadas no começo do processo. Da mesma maneira, as variáveis locais não são iniciadas no começo do escopo aos quais elas pertencem. Para sair desse estado, elas devem ser iniciadas utilizando, por exemplo, a atribuição de valores ou mensagens. Se uma leitura de uma variável não-iniciada for feita, uma falha é gerada. As variáveis BPEL também possuem um mecanismo de validação. Por meio da atividade **validate**, pode-se garantir que os valores das variáveis sejam sempre válidos, comparados às definições que foram feitas em algum arquivo XML auxiliar. Se um problema relativo a validação é encontrado, uma falha é gerada.

Somente a utilização de variáveis não é suficiente para controlar o comportamento de um processo. É necessária alguma maneira de extrair e combinar esses dados. Dessa forma, surge a necessidade de algum mecanismo de expressões. BPEL optou por não "reinventar a roda" nesse aspecto, deixando que a linguagem utilizada para descrever as expressões seja escolhida na hora da especificação do processo. BPEL utiliza a linguagem XPath 1.0 para a descrição de expressões, determinando que as implementações de BPEL devem oferecer a interpretação dessa linguagem como funcionalidade obrigatória.

BPEL usa vários tipos de expressões, tais como expressões Booleanas, expressões que devolvem uma data de expiração, expressões que devolvem uma duração, expressões que devolvem um número inteiro e outras de uso geral.

Por meio da atividade **assign**, os dados podem ser copiados de uma

variável para outra. Além disso, ela permite a atribuição de novos dados às variáveis, com o uso de expressões.

10. Instâncias de processos: Identificação e ciclo de vida

Um processo descrito em BPEL é construído baseado em um modelo de serviço definido pela especificação WSDL.

Todas as interações externas realizadas pelo processo ocorrem por meio de interfaces WSDL de serviços web. Porém, o modelo de interação de WSDL não armazena estados, sendo baseado em interações curtas. Entretanto, os processos de negócio em BPEL representam interações longas, que requerem o armazenamento de estados, sendo que cada interação possui um começo, um determinado comportamento durante a execução do processo e um fim.

Tem-se então uma divergência entre os paradigmas de WSDL e dos processos BPEL. Para resolver essa questão, decidiu-se fazer o gerenciamento implícito do ciclo de vida do processo. Dessa maneira, o cliente do serviço web não tem consciência da existência de um processo subjacente ao serviço web acessado por ele. Quando uma mensagem é recebida por meio de determinadas operações que o serviço web disponibiliza, uma nova instância do processo é automaticamente criada. Uma instância é finalizada quando suas atividades finais acabam de executar ou então quando é executada uma atividade que explicitamente termina a execução da instância do processo , **exit** por exemplo.

Existe ainda a questão da identificação das instâncias de um processo. De alguma maneira, BPEL precisa lidar com o roteamento das mensagens que chegam ao serviço web. É necessário saber onde a mensagem será entregue e se o seu destino será uma nova instância do processo ou uma instância já existente.

É essa identificação que permite aos processos participarem de interações longas. Em muitos sistemas existentes, esse problema é resolvido por meio da criação de um identificador para cada instância.

Ao chegar uma mensagem, ela é examinada em busca de um identificador que indique qual é o seu destino. Em BPEL, esta questão é resolvida de uma maneira levemente diferente: por meio do uso de

correlações. Com o mecanismo de correlações, dados específicos da aplicação, que ficam disponíveis nas mensagens trocadas, são utilizados para manter referências para instâncias específicas do processo. Fazendo uma analogia com os Sistemas Gerenciadores de Banco de Dados, esse processo de identificação seria semelhante à identificação unívoca de uma tupla de uma relação a partir de uma chave primária pré-definida.

A localização exata dos dados usados pela correlação variam de mensagem para mensagem dentro de um mesmo processo. Para especificar quais dados devem ser utilizados na correlação, utiliza-se o arquivo WSDL associado ao processo, no qual são definidas as estruturas responsáveis por fazer estas associações, essenciais para o funcionamento do mecanismo.

11. Exempo de especificação de processo

Segue o trecho de código referente à especificação na linguagem BPEL. Várias simplificações foram feitas, deixando apenas a estrutura essencial para o entendimento do fluxo do processo.

Em BPEL, as tarefas do exemplo são desempenhadas por serviços web. Portanto, na especificação, as tarefas são mapeadas para interações com esses serviços, feitas por meio de atividades BPEL.

Nessa estrutura, pode-se verificar a presença de uma atividade **receive**, que recebe uma mensagem inicial, criando uma instância do processo. Em seguida, duas atividades **invoke** são responsáveis pelo cálculo inicial do preço e escolha da transportadora.

A partir daí, a atividade **flow** entra para aninhar duas outras atividades, paralelizando-as. A primeira delas é uma atividade **invoke**, responsável pelo término do cálculo do preço. A outra atividade é uma seqüência que contém outras duas atividades **invoke**, das quais uma notifica a transportadora e a outra termina o agendamento da entrega. Terminada a atividade **flow**, segue, então a resposta ao cliente feita pela atividade **reply**.

```
<process name=''ProcessamentoPedido'' . . .>
<sequence>
 <receive . . . operation=''recebePedidoCompra''
createInstance=''yes''> ...
</receive>
<invoke . . .operation=''iniciaCalculoPreco''>
...
</invoke>
 <invoke . . .operation=''decideTransportadora''>
...
</invoke>
<flow>
<invoke . . . operation=''terminaCalculoPreco''>
...</invoke>
  <sequence>

  <invoke
..operation=''notificaTransportadora''>..</invoke>

  <invoke
operation=''terminaAgendamentoEntrega''>…</invoke>
</sequence>

</flow>
<reply ...operation=''notificaCliente''>
...
</reply> </sequence>
</process>
```

12. Motores e Aplicações para BPEL

Existe no mercado diversas soluções de orquestração WS, baseadas em BPEL, isto é, aplicações que permitem a designers desenvolverem processos sendo possível também fazer o *deploy* destes como serviço, para posterior invocação.

Estas aplicações e "motores" BPEL (BPEL *Engines*) possuem também IDEs e outras ferramentas de edição de código, disponibilizadas gratuitamente, sendo muitas delas também *open source*, o que permite que sofram evoluções regulares por parte da comunidade; assim como versões pagas, mais completas e com melhor suporte, especialmente desenhadas para processos numa escala maior, normalmente empresarial.

12.1 Apache ODE

A filosofia do ODE em relação ao BPEL é a de que esta é a linguagem para descrever como implementar capacidades de comunicação baseada em mensagens entre serviços, não só em relação a processos de negócio, mas a todo o tipo de atividades envolvendo WS.

A idéia é implementar uma orquestração sem recorrer a IDEs (GUI, *Graphic User Interfaces*), ambientes integrados para desenvolvimento do software (IDE, *Integrated Development Environment*) ou outras que não apenas algum uso de XML, encapsulando detalhes de concorrência, continuação durável, fiabilidade e recuperação de falhas. Ao ser disponibilizado como um componente ao invés de um sistema (*framework*), o ODE tenta impulsionar os programadores a introduzir orquestração nos seus sistemas].

O ODE, no seu núcleo (*core*), usa uma implementação baseada numa camada de integração para receber e enviar mensagens para serviços externos e para aceder a recursos, como threads. Essa camada possui implementações AXIS2, e os processos do ODE são expostos como WS, para o *Service Mix (ESB)*, em que essa exposição é feita como um motor de serviço JBI (*Java Business Integration*) .

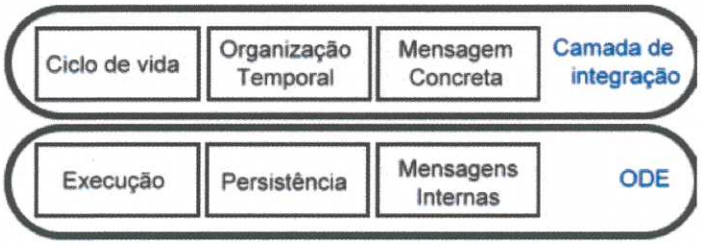


Imagem do Apache ODE.

Dois dos aspectos mais desafiantes para implementar um motor BPEL envolvem a representação do estado de um processo em execução e a gestão da concorrência de processos.

Para suprir estes aspectos, o ODE utiliza o Jacob (*Java Concurrent Objects*), uma *Framework* que permite representar o estado dos processos como uma coleção de objetos, ligados por canais de transmissão de mensagens; não mensagens "pesadas" de WSDL como as enviadas e recebidas externamente, mas simples mensagens internas Java, muito mais leves.

Ao receber uma mensagem, um objeto pode criar novos canais e objetos e enviar mensagens. Assim, a concorrência é gerida através do consumo de mensagens num processo *single-thread* por cada par processo/instância.

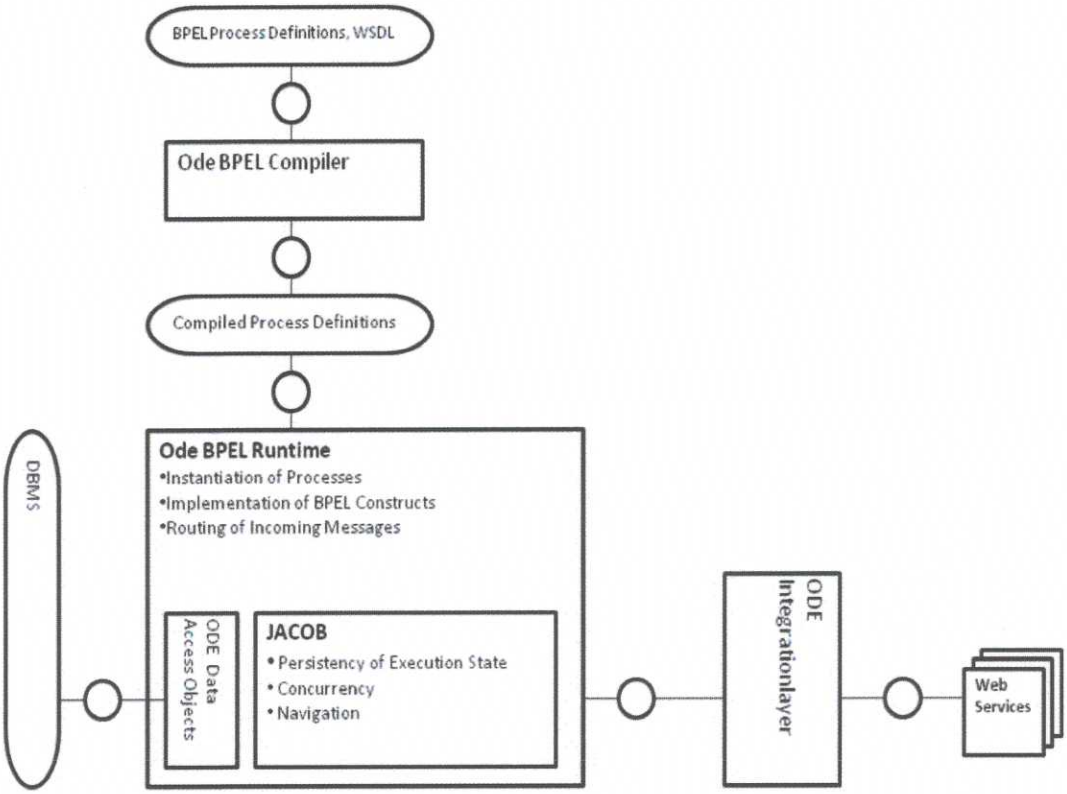
Sendo um motor BPEL, o **ODE** pode ser instalado em dois ambientes distintos, sendo que ambas as instalações possuem uma base de dados própria embutida (Derby):

- Como um WS em AXIS2 (um "motor de núcleo" – *core engine* – de WS da Apache), cuja implementação (*deploy*) é feita usando um servidor de aplicações (sendo o mais comum o Tomcat) e invocado usando SOAP/http.

- Como um conjunto de serviços JBI (*Java Business Integration*), que podem ser implementados através de um repositório JBI (por exemplo, ServiceMix ESB).

Olhando para o esquema da arquitetura do ODE, podem-se distinguir os seus vários componentes principais: o Compilador BPEL (*ODE BPEL Compiler*), o motor de execução BPEL (*ODE BPEL Runtime*

Engine), ODE DAOs (*Data Access Objects*), camadas de integração (*ODE Integration Layers*) e ferramentas dos utilizadores.



Arquitetura do Apache ODE.

O compilador é o responsável, como o nome indica, por compilar o código base de BPEL (Contratos - WSDL e Schemas, XSD), isto é, converter todos os elementos associados ao BPEL em código executável, sendo o seu output um código compilado corretamente (normalmente com a extensão ".cbp") em caso de sucesso ou uma lista de erros encontrados durante a compilação, caso contrário.

O motor de execução, *runtime*, recebe o arquivo resultante da compilação descrita acima, sendo o responsável pela execução do código compilado. O *runtime* fornece as implementações para as várias estruturas do BPEL, assim como implementa toda a lógica necessária para determinar quando uma nova instância é necessária e para que instância se deve encaminhar cada mensagem recebida.

A aplicação de gestão de processos (*Process Management API*) é implementada pelo *runtime*, sendo utilizada pelas ferramentas do utilizador para interagir com o motor.

De modo a garantir viabilidade num ambiente não viável, o *runtime* apoia-se em DAOs para garantir a persistência dos processos, normalmente utilizando uma base de dados relacional.

Ao nível das instâncias, o *runtime* implementa as construções BPEL via Jacob, que proporciona um mecanismo de concorrência em nível das aplicações, sem depender de *threads*, assim como um mecanismo transparente para interromper a execução, enquanto garante a manutenção do estado.

O ODE *Data Access Objects,* DAOs, serve de mediador entre o *runtime* e uma camada inferior de dados armazenados, tipicamente uma base de dados relacional JDBC (*Java Database Connectivity*), sendo os DAOs implementados via uma biblioteca de acesso a dados OpenJPA (*Java Persistence API*). Assim, *runtime engine* requer que os DAOs resolvam os seguintes problemas de persistência:

- Instâncias Ativas: manter um registo atualizado das instâncias criadas.
- *Routing* de mensagens: qual instância está à espera de qual mensagem.
- Variáveis: o valor das variáveis BPEL de cada instância.
-*Partner Links*: os valores dos *partner links* BPEL em cada instância.
- Estado da execução dos processos: o estado serializado da "máquina de persistência virtual" (*virtual persistence machine*).

As camadas de integração resolvem a necessidade do motor de execução tem em interagir com o mundo exterior, uma vez que é incapaz por si mesmo.

Assim, o que estas fazem é envolver o *runtime* num ambiente de execução, fornecendo-lhe canais de comunicação, assim como mecanismos de organização de *threads*, ao mesmo tempo que gere o seu ciclo de vida (isto é, configura e inicializa o *runtime*).

Para criar os processos em BPEL, utilizam-se aplicações, gráficas ou textuais que permitam codificar os processos.

Uma das mais utilizadas para este efeito é a ferramenta "Eclipse", uma IDE (*Integrated Development Environment*, ou ambiente de desenvolvimento integrado) com um interface gráfica que suporta as atividades e componentes do BPEL.

12.2 ActiveVOS

Consistindo num motor BPEL e numa IDE, o **ActiveBPEL** é uma aplicação *standalone*, *open source* e sob a licença da Active Endpoints, que pretende englobar numa só aplicação todos os standards de WS, baseando-se em BPEL.

O ActiveBPEL possui uma versão freeware, de desenvolvimento e não empresarial de um outro produto desta companhia, o ActiveVOS. Este é um produto assumidamente corporativo, possuindo algumas funcionalidades próprias para projetos de grandes dimensões e de maior complexidade. Com uma licença paga e com um maior suporte, assim como um grande cuidado no teste das aplicações e soluções para encontrar falhas e erros e consequente recuperação, o ActiveVOS garante o funcionamento de processos críticos para uma empresa.

Apesar disso, o ActiveBPEL apresenta todas as funcionalidades e ferramentas para desenvolver qualquer projeto em BPEL que um usuário ou pequena empresa pretenda.

Apesar de não ser *open source*, como o motor BPEL, a IDE é disponibilizada pela Activepoints para download gratuitamente no seu site
(http://www.activebpel.org/products/activebpeldes/get_code.html)

O ActiveBPEL designer possui um *servlet* incorporado, um *subset* do conhecido Apache Tomcat, plenamente funcional. Deste modo, não é necessária a instalação de um servidor externo para fazer o *deploy* dos processos criados nesta ferramenta. Além de designer, o ActiveBPEL possui também incorporado um motor BPEL, fornecendo todas as ferramentas numa só aplicação para desenvolver, testar e executar orquestrações em BPEL .

Para fazer o *deploy* dos processos BPEL, o motor ActiveBPEL "empacota" todos os seus componentes num arquivo ".bpr" (*business process archive*), que é automaticamente executado, existindo também a possibilidade de gerar o *script* Ant (um pouco como o *make* de programas em Linux, mas baseado na linguagem Java) para posterior re-execução do arquivo.

Durante a criação do arquivo "bpr", é gerado também um catálogo de recursos automaticamente pelo assistente **bpr**. As informações desse catálogo é mostrada no console de administração do motor, providenciando também um meio deste procurar contratos WSDL, schemas e outros recursos durante a execução.

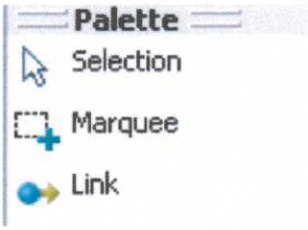
12.2.1 ActiveBPEL

Neste capítulo mostraremos as "Atividades" que fazem parte da linguagem BPEL, assim como a sua representação no editor ActiveBPEL. Estas "atividades" constituem a base do BPEL, sendo necessária a familiarização com tais afim de criar uma orquestração.

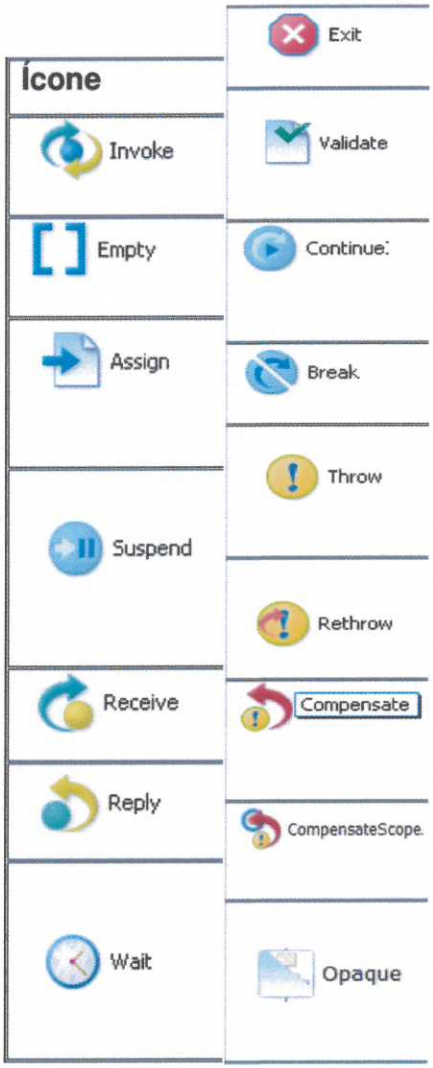
Na paleta do ActiveBPEL estão representadas as atividades que constituem a sintaxe do BPEL.

Paleta de componentes.

Na primeira parte da paleta, temos o separador "*Links and Selections*", que, apesar de não ser constituído por atividades BPEL, permite escolher as ferramentas de selecção de atividades, assim como a ferramenta utilizada para criar *links* entre atividades.



Na paleta, no separador "*Activity*", encontram-se as principais atividades, as atividades básicas, de um processo BPEL, inclusive as usadas para criar *Fault Handlers*.



Atividades básicas usadas em BPEL.

Das atividades descritas, algumas apenas podem ser usadas em certos contextos num processo. Assim, as atividades de *compensate* e *compensateScope* apenas podem ser usadas dentro de *compensation handlers* (que têm um *scope* – domínio - onde estão presentes as

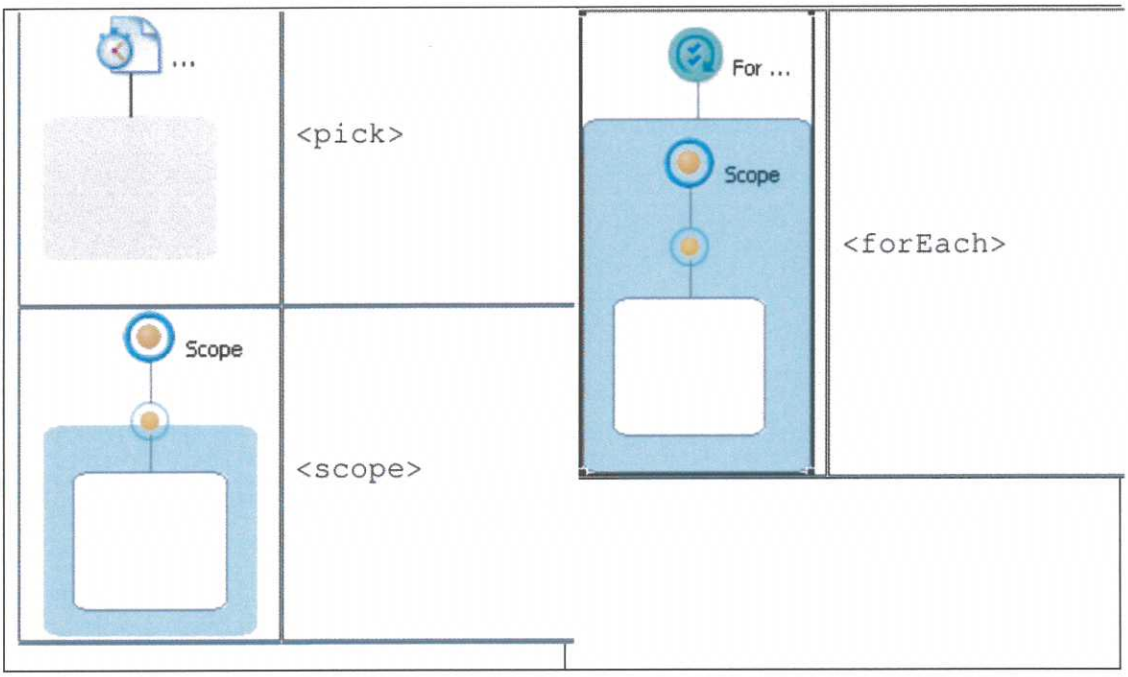
atividades que têm que compensar), *fault handlers* (responsáveis pelas ações em caso de falha) ou *termination handlers* (responsável pelas ações quando um processo ou parte dele é terminado antes que a sua execução esteja completada).

A atividade de *rethrow* apenas pode ser usada dentro de uma atividade de *catch* e *catchAll*, sendo uma das partes standard destas últimas (ambas as atividades são elementos pertencentes aos *fault handlers*).

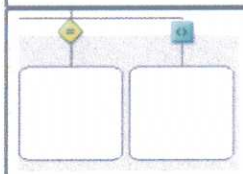
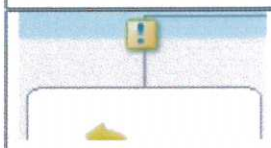
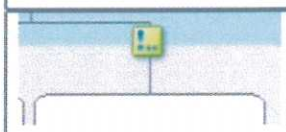
Quer a atividade *continue* ou não - *break*, estão confinadas a ciclos (*if, while* ou *forEach*), uma vez que a sua ação destina-se a alterar a normal execução destes, terminando uma iteração do ciclo ou o ciclo completo, respectivamente.

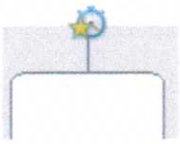
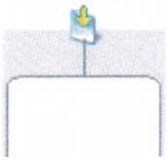
No terceiro *container*, estão as atividades que têm associados *scopes* à sua estrutura (denominadas de estruturais), isto é, ciclos, fluxos, sequências, etc, que servem de suporte para um conjunto de atividades dentro dos seus "domínios".

Ícone	Instrução		
...	<sequence>	If	<if>
...	<flow>	...	<while>



Depois das atividades estruturais, aparece na *paleta* um separador (*other*) com atividades complementares.

Ícone	Instrução
	<elseif> <else>
	<catch>
	<catchAll>
	<onEvent>

	<onAlarm>
	<onMessage>

Os elementos deste separador são de suporte para outras atividades, permitindo aumentar a complexidade e profundidade das orquestrações.

Com os elementos presentes nestas "*paletas*" e com WS externos, podem-se criar processos de negócio dos mais simples aos mais complexos, invocando, gerindo, isto é, orquestrando vários WS para que sejam atingidos os fins a que um projeto BPEL se propõe – a criação de um processo de negócio – também ele disponível sob a forma de um WebService.

13. Exemplos de Processos de Negócio

Qual é a aparência de um BPEL? Qual é o código XML que lhe serve de base? Como são construídos os elementos, as invocações e as ligações entre estes?

Mostraremos às seguir exemplos de processos de negócio usando representações criadas no ActiveBPEL *Designer* e explicando o código gerado.

Os processos de negócio analisados mostram o uso da maior parte das atividades BPEL, de modo a exemplificar a sua importância dentro de um processo, a sua utilidade e a sua utilização.

Aprovação de um empréstimo

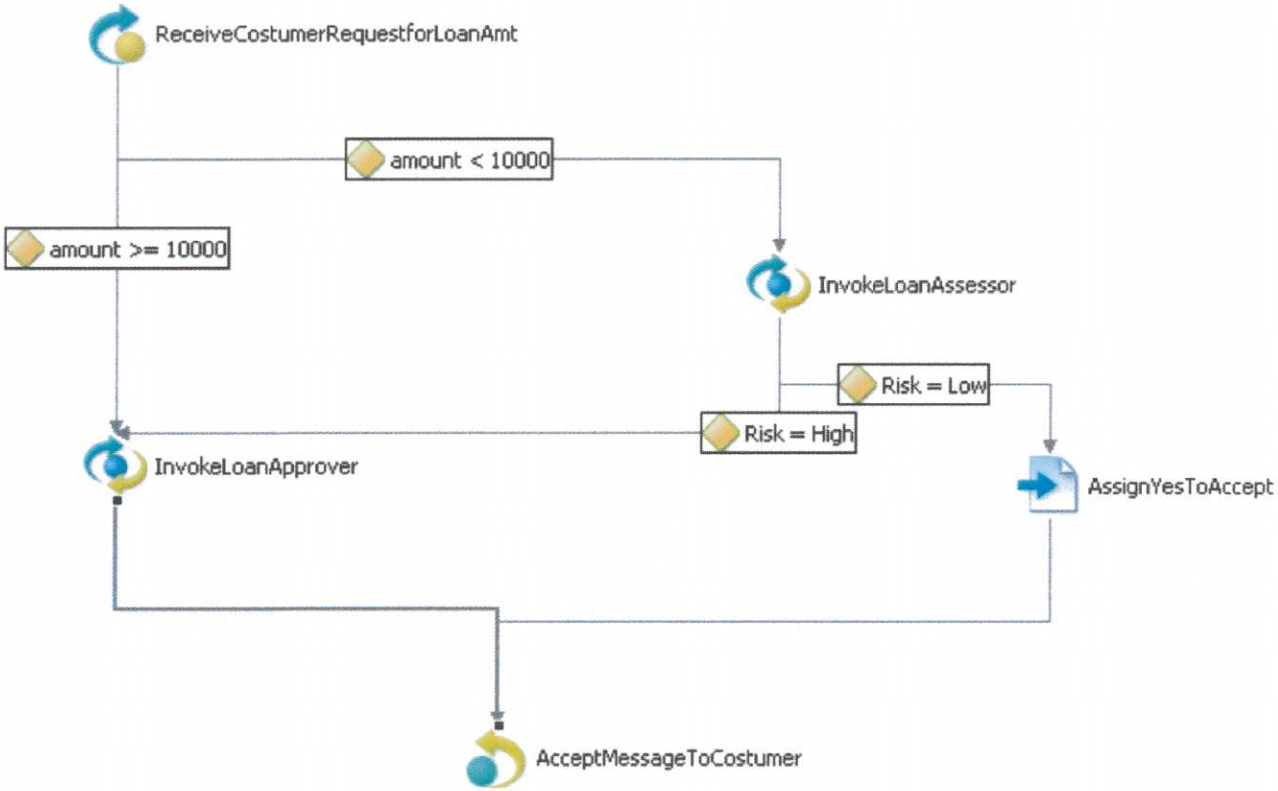


Imagem: Orquestração de um pedido de empréstimo.

Este é um processo de negócio simples, que consiste numa aprovação de empréstimo, tendo como critérios de aprovação a quantia pretendida e o risco que um cliente representa.

Este processo é iniciado com a recepção de um pedido do cliente com o valor do empréstimo pretendido. Se esse valor é inferior a 10000, é invocado um WS do empréstimo, e que vai verificar se o cliente é de alto ou baixo risco. Se o risco é baixo, o empréstimo é feito com sucesso, caso contrário é enviado para um novo WS, o *LoanApprover*, que vai aprovar ou não o empréstimo. Caso o valor do pedido seja superior ou igual a 10000, o pedido é imediatamente enviado para o *LoanApprover*. O cliente receberá uma aceitação por parte do assessor ou um sim/não por parte do *Approver*.

O processo começa com uma instrução de *receive*, que vai escolher o caminho tomado pelo pedido baseando-se no montante:

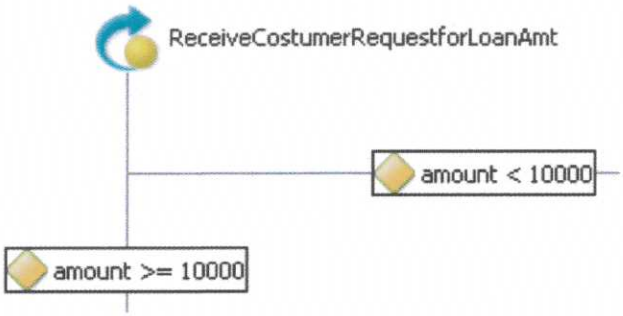


Imagem: Parte do BPEL correspondente à etapa.

```
<bpel:receive createInstance="yes"
name="ReceiveCostumerRequestforLoanAmt" operation="request"
partnerLink="costumer" portType="lns:loanServicePT" variable="request">
        <bpel:sources>
           <bpel:source linkName="receive-to-assess">
              <bpel:transitionCondition>$request.amount &lt;
10000</bpel:transitionCondition>
           </bpel:source>
           <bpel:source linkName="receive-to-aprove">
              <bpel:transitionCondition>$request.amount &gt;=
10000</bpel:transitionCondition>
           </bpel:source>
        </bpel:sources>
     </bpel:receive>
```

Imagem: Código referente à mesma.

No código pode ler-se a instrução inicial *receive*, delimitada pelas expressões "<bpel:receive" e "</bpel:receive>", seguida dos argumentos necessários à sua criação, como o seu nome ("ReceiveCostumerRequestforLoanAmt"), o nome da operação ("request"), do *Partner Link* a que esta associada ("costumer"), o *portType* utilizado ("lns:loanServicePT") e o nome da variável (que vai ter neste caso o mesmo nome que a operação - "request").

Dentro do corpo da instrução, situam-se as condições que vão definir o caminho a seguir pelo pedido.

```
<bpel:transitionCondition>$request.amount &lt;
10000</bpel:transitionCondition>
</bpel:source>

<bpel:source linkName="receive-to-aprove">
<bpel:transitionCondition>$request.amount &gt;=
10000</bpel:transitionCondition>
```

Estas condições são demarcadas pela primitiva **BPEL** "bpel:transitionCondition", em que o campo "*amount*" do pedido é comparado com um valor (10000), sendo que o resultado dessa comparação leva à escolha do caminho deste.

Como se pode ver pelo código, todas as primitivas e construções BPEL estão claramente assinaladas com a expressão "<bpel:...", tornando-as assim distintas de código XML normal.

Em seguida, se o pedido for inferior a 10000, é invocado um WS, com o nome "*InvokeLoanAssessor*", que vai fazer a avaliação do pedido dependendo do risco do cliente.

Caso o risco seja "*Low*", é feito um ***assign*** com a resposta positiva para o cliente, caso contrário é necessário invocar um novo WS, *InvokeLoanApprover*, que também é invocado quando o valor do empréstimo pretendido excede ou iguala o valor de 10000, que irá ser responsável pela decisão final em relação à aceitação ou não do pedido de empréstimo

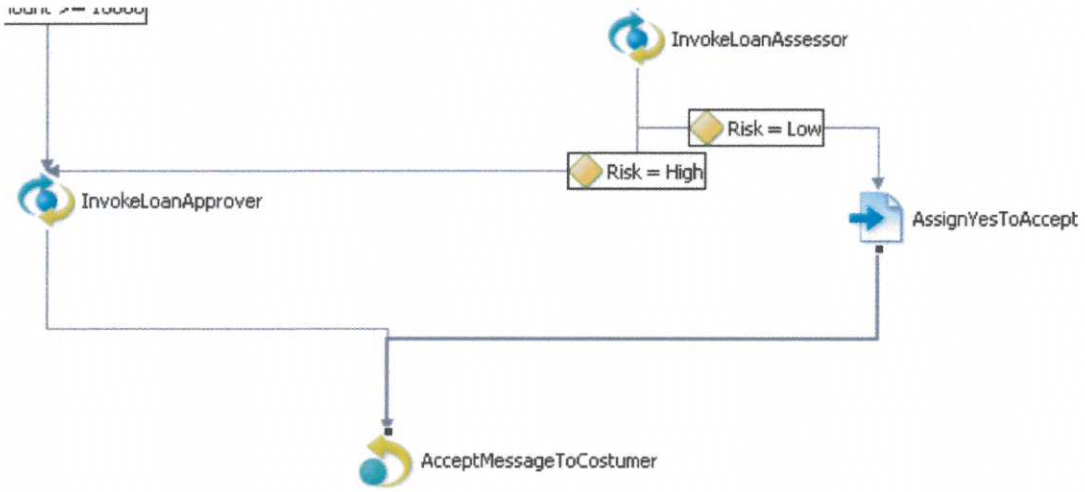


Imagem: O Web Service invocado, *invokeLoanAssessor*, decide o destino do pedido, de acordo com o risco associado ao cliente.

```
<bpel:invoke inputVariable="request" name="InvokeLoanAssessor"
operation="check" outputVariable="risk" partnerLink="assessor"
portType="lns:riskAssessmentPT">
        <bpel:targets>
           <bpel:target linkName="receive-to-assess"/>
        </bpel:targets>
        <bpel:sources>
           <bpel:source linkName="assess-to-aprove">
              <bpel:transitionCondition>$risk.level !=
'low'</bpel:transitionCondition>
           </bpel:source>
           <bpel:source linkName="assess-to-setMessage">
              <bpel:transitionCondition>$risk.level =
'low'</bpel:transitionCondition>
           </bpel:source>
        </bpel:sources>
</bpel:invoke>
```

Neste caso temos uma instrução de ***invoke***, delimitada pelas primitivas "**<bpel:invoke...**" e "**</bpel:invoke>**"; no cabeçalho da instrução, estão presentes as variáveis de entrada, o nome do WS ("InvokeLoanAssessor"), assim como a operação, a variável de saída, o ***Partner Link*** e o tipo utilizado. Vale notar que os campos presentes no cabeçalho não diferem dos relativos a instruções de *receive/reply*.

```
<bpel:targets>
        <bpel:target linkName="receive-to-aprove"/>
        <bpel:target linkName="assess-to-aprove"/>
</bpel:targets>
```

Os *links* de entrada são designados por *targets*, aparecendo logo a seguir ao cabeçalho da atividade, sendo todos definidos dentro do mesmo contexto (delimitados pelas instruções **<bpel:targets>** e **</bpel:targets>**).

```
<bpel:sources>
            <bpel:source linkName="assess-to-aprove">
               <bpel:transitionCondition>$risk.level !=
'low'</bpel:transitionCondition>
            </bpel:source>
            <bpel:source linkName="assess-to-setMessage">
               <bpel:transitionCondition>$risk.level =
'low'</bpel:transitionCondition>
            </bpel:source>
</bpel:sources>
```

O código que representa os *links* de saída da instrução (são delimitados pelas instruções **<bpel:sources>** e **</bpel:sources>**. Para cada *link* temos associadas as transições/condições que representam, dentro de um contexto próprio para cada *fonte*.

Neste caso, tem-se que, se o risco é "Low" ($risk.level = 'low') o processo continua pelo *link* "assess-to-setMessage", que irá fazer um *assign* da resposta para ser transmitida à saída do processo; caso contrário ($risk.level != 'low') o *link* seguido será "assess-to-aprove", que irá encaminhar o pedido para o serviço "*InvokeLoanApprover*".

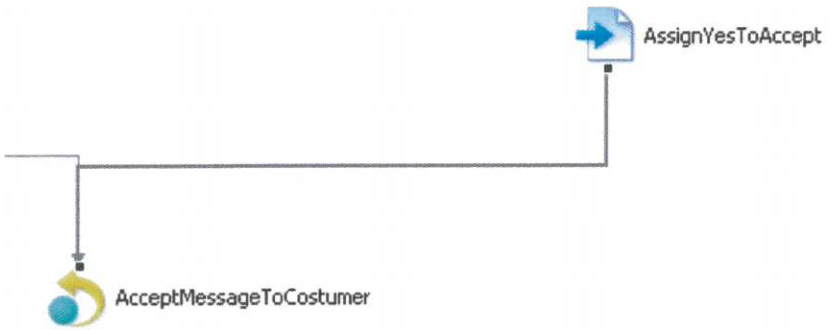


Imagem: Parte final do processo. A resposta é copiada para a variável de saída através de um *assign*, sendo enviada para o utilizador através de um *reply* ("*AcceptMessageToCostumer*")

```
<bpel:assign name="AssignYesToAccept">
        <bpel:targets>
           <bpel:target linkName="assess-to-setMessage"/>
        </bpel:targets>
        <bpel:sources>
           <bpel:source linkName="L2"/>
        </bpel:sources>
        <bpel:copy>
           <bpel:from>
              <bpel:literal>Yes</bpel:literal>
           </bpel:from>
           <bpel:to part="accept" variable="approval"/>
        </bpel:copy>
</bpel:assign>
```

O *assign* apenas possui no seu cabeçalho o seu nome ("**AssignYesToAccept**"). Novamente se encontram definidos os *links* de entrada e saída, *target* e *source* respectivamente, seguidos da atividade a que este se destina, a cópia de um valor para a variável de saída.

```
<bpel:copy>
            <bpel:from>
               <bpel:literal>Yes</bpel:literal>
            </bpel:from>
            <bpel:to part="accept" variable="approval"/>
</bpel:copy>
```

A cópia (delimitadas pelas instruções **<bpel:copy>** e **</bpel:copy>**) é feita definindo a origem do valor (**<bpel:from>**), neste caso o literal "Yes", e a variável de destino, assim como o campo da variável em específico (**<bpel:to** part="accept" **variable**="approval"**/>**).

```
<bpel:reply name="AcceptMessageToCostumer" operation="request"
partnerLink="costumer" portType="lns:loanServicePT" variable="approval">
        <bpel:targets>
           <bpel:target linkName="L1"/>
           <bpel:target linkName="L2"/>
        </bpel:targets>
</bpel:reply>
```

Para finalizar o processo, tem uma atividade de *reply*, que vai devolver os valores de resposta do processo. Possui o cabeçalho com os mesmos campos que as atividades de *request* e *invoke* e, uma vez que não possui *links* de saída (é a atividade final do processo), apenas tem os *targets* que correspondem aos links de entrada.

14. Ordem de encomenda

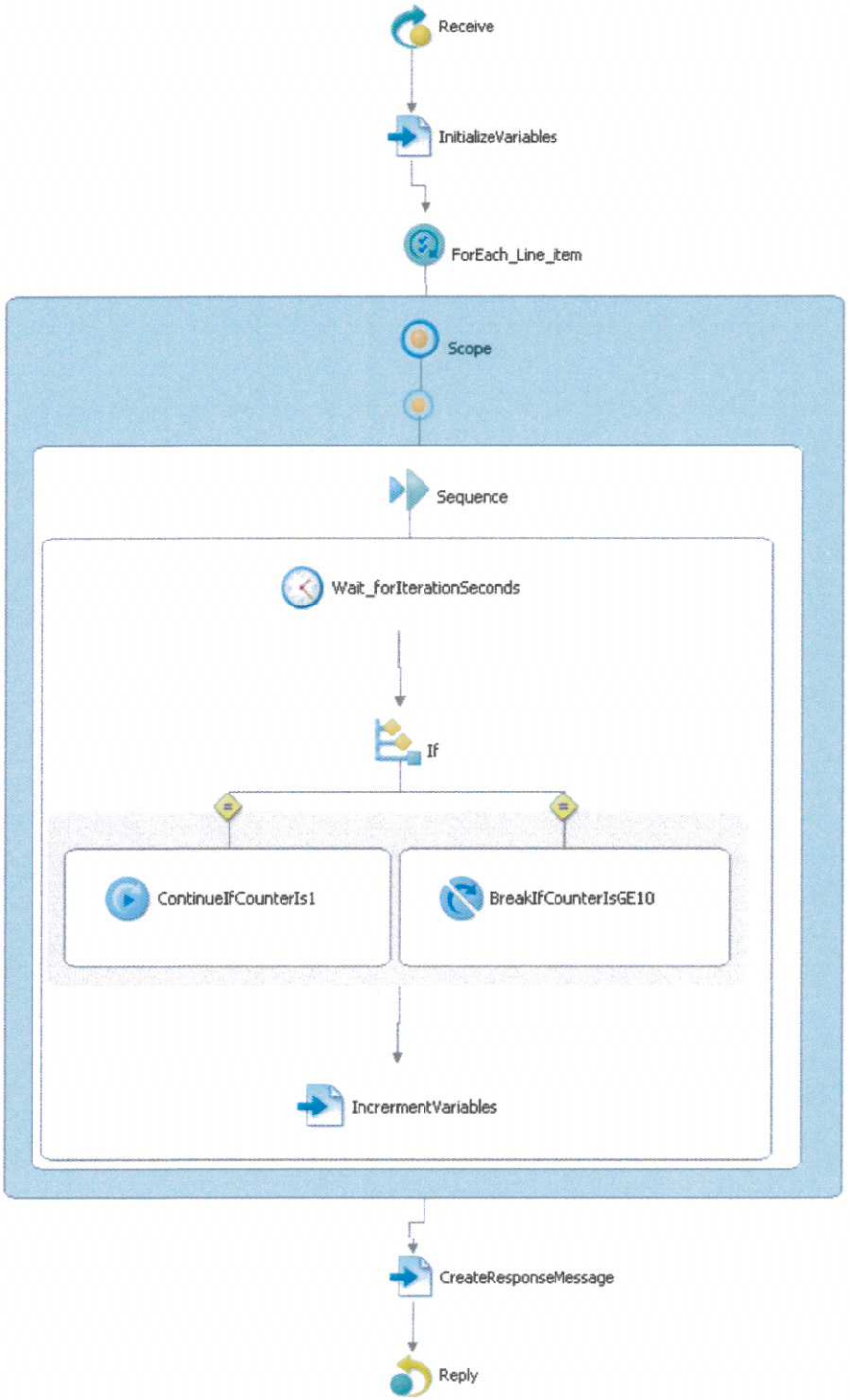


Imagem: Mapa do processo de negócio de processamento de encomendas.

Este processo destina-se a mostrar o funcionamento de vários tipos de atividades presentes na linguagem BPEL, algumas delas, como o if, *forEach*, *continue* e *break* foram implementadas na versão 2.0 do BPEL, que neste momento é o padrão da linguagem.

O processo consiste em percorrer uma série de elementos de uma ordem de encomenda, contando-os à medida que estes são atendidos, e dependendo desse número, continuar a percorrer elementos ou terminá-lo.

O processo começa com um *receive*, que aceita a ordem de encomenda, sendo a resposta devolvida por um *reply* no final do processo, tal como na atividade mostrada anteriormente.

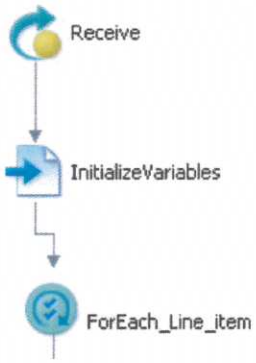


Imagem: O processo é iniciado com a recepção da ordem. Em seguida são inicializadas as variáveis que irão ser utilizadas, seguindo-se a atividade *forEach.*

De seguida são inicializadas as variáveis que vão ser utilizadas no processo utilizando uma instrução de *assign* denominada por "*InitializeVariables*". Este processo é feito através da cópia de valores para as variáveis.

```
<bpel:assign name="InitializeVariables">
        <bpel:targets>
           <bpel:target linkName="L1"/>
        </bpel:targets>
        <bpel:sources>
           <bpel:source linkName="L3"/>
        </bpel:sources>
        <bpel:copy>
           <bpel:from>count($orderMessage.order/OrderDetail
)</bpel:from>
           <bpel:to variable="numberOfDetails"/>
        </bpel:copy>
        <bpel:copy>
```

```
        <bpel:from>0</bpel:from>
        <bpel:to variable="numberOfCountedDetails"/>
    </bpel:copy>
    <bpel:copy>
        <bpel:from>0</bpel:from>
        <bpel:to variable="totalOrderValue"/>
    </bpel:copy>
        </bpel:assign>
```

São três as variáveis inicializadas: "numberOfDetails", que representa o número de elementos iniciais da ordem de encomenda (o número inicial é obtido pela expressão

count($orderMessage.order/OrderDetail);

"numberOfCountedDetails", que vai contar o número de detalhes/elementos da ordem que foram processados (inicializado a 0); "totalOrderValue", que vai guardar o valor total dos elementos da encomenda processados (que também é inicializado a 0).

Depois das variáveis inicializadas, segue-se a instrução que dá o nome ao processo, *forEach*, que vai percorrer cada elemento da ordem da encomenda e aplicar em cada um as instruções que estão no corpo da instrução.

```
        <bpel:forEach counterName="counter" name="ForEach_Line_item"
parallel="yes">
            <bpel:targets>
                <bpel:target linkName="L3"/>
            </bpel:targets>
            <bpel:sources>
                <bpel:source linkName="L2"/>
            </bpel:sources>
            <bpel:startCounterValue>1</bpel:startCounterValue>
           <bpel:finalCounterValue>$numberOfDetails</bpel:finalCounterValue>
```

O cabeçalho desta atividade contém o seu nome (name="ForEach_Line_item"), o nome do contador (counterName="counter") e a escolha da opção se cada "instância" irá ser corrida em paralelo com as restantes, que neste caso é afirmativo (parallel="yes").

As entradas para a instrução, *targets*, e as saídas, *sources*, encontram-se definidas logo de seguida. Também o contador fica definido com o número inicial (<bpel:startCounterValue>1</bpel:startCounterValue>) e final (que corresponde ao número de detalhes/elementos da ordem,

<bpel:finalCounterValue>$numberOfDetails</bpel:finalCounterValue>
).

Todo o corpo da instrução se encontra dentro do seu "*scope*", isto é, o conjunto de instruções a ser aplicado a cada elemento funciona como uma "caixa negra" independente, em que cada um é tratado independentemente, percorrendo todo o fluxo de instruções até ser devolvida uma resposta.

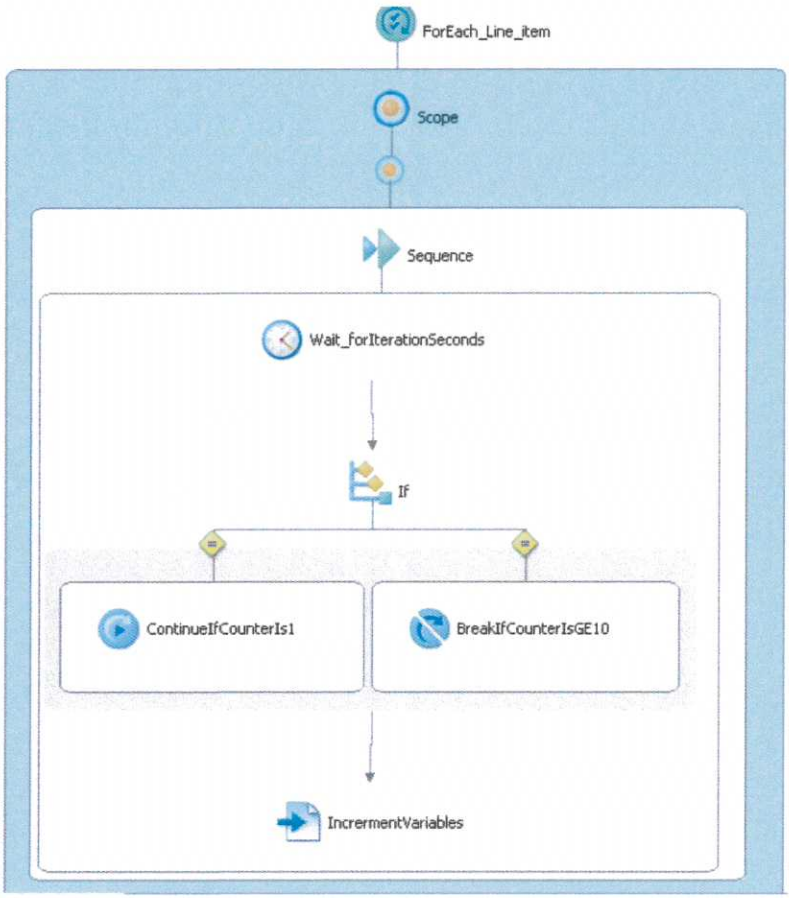


Imagem: No corpo, *scope*, da instrução *forEach,* está definida a sequência pela qual vão passar os elementos da ordem de encomenda.

Neste caso, o corpo da instrução consiste numa sequência (definida pelo cabeçalho <bpel:sequence>).

```
<bpel:wait name="Wait_forIterationSeconds">
      <bpel:for>concat('PT', string($counter), 'S')</bpel:for>
</bpel:wait>
```

Esta começa com uma atividade de *wait*, que usa uma expressão baseada no contador, para definir qual o tempo a esperar, de modo a simular o efeito de um *invoke* a ocorrer em cada uma das iterações da atividade do *forEach*.

```
<bpel:if>
      <bpel:condition>($counter = 1)</bpel:condition>
      <bpel:extensionActivity>
           <ext1:continue name="ContinueIfCounterIs1"/>
      </bpel:extensionActivity>
      <bpel:elseif>
           <bpel:condition>($counter &gt;= 10)</bpel:condition>
           <bpel:extensionActivity>
                <ext1:break name="BreakIfCounterIsGE10"/>
           </bpel:extensionActivity>
      </bpel:elseif>
</bpel:if>
```

Uma instrução condicional, um *if*, vai analisar o valor do contador ($counter). Se o contador é igual a 1 (**<bpel:condition>**($counter = 1)**</bpel:condition>**) é executada uma atividade de *continue* ("ContinueIfCounterIs1"), que termina todas as instruções ainda a correr da iteração, completa o *scope* normalmente, e continua com a próxima iteração (execução sequencial) ou permite que todas as outras iterações continuem (execução paralela, como é o caso neste processo).

Quando o contador chega a 10 (**<bpel:condition>**($counter >= 10)**</bpel:condition>**), é executada uma atividade de *break* ("BreakIfCounterIsGE10"), tal como a instrução de *continue*, termina as instruções ainda a correr na iteração e completa o *scope* normalmente, mas que completa a atividade *forEach* sem terminar quaisquer instruções ainda em espera (execução sequencial) ou, neste caso (execução paralela), força o término de todas as instruções ainda a correr.

```
<bpel:assign name="IncrermentVariables">
                  <bpel:copy>
                     <bpel:from>($numberOfCountedDetails + 1)</bpel:from>
                     <bpel:to variable="numberOfCountedDetails"/>
                  </bpel:copy>
                  <bpel:copy>
```

```
<bpel:from>($orderMessage.order/OrderDetail[$counter]/TotalCost +
$totalOrderValue)</bpel:from>
                    <bpel:to variable="totalOrderValue"/>
                 </bpel:copy>
</bpel:assign>
```

Ao fim da sequência, aparece uma atividade de *assign* ("IncrermentVariables"), que é responsável por incrementar o número de elementos processados ("numberOfCountedDetails"), assim como de atualizar o valor total da encomenda ("totalOrderValue"), à medida que os elementos desta vão sendo processados, incrementando o valor de cada um destes ao total **(($orderMessage.order/OrderDetail[$counter]/TotalCost +$totalOrderValue)).**

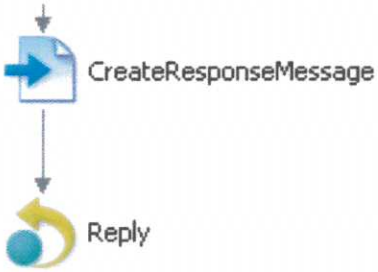


Imagem: Para finalizar o processo, um *assign* cria a resposta para o cliente, que é de seguida devolvida através de um *reply*.

```
<bpel:assign name="CreateResponseMessage">
         <bpel:targets>
            <bpel:target linkName="L2"/>
         </bpel:targets>
         <bpel:sources>
            <bpel:source linkName="L4"/>
         </bpel:sources>
         <bpel:copy>
            <bpel:from>concat('The order contains ', $numberOfDetails, '
detail elements and the proceess traversed ', $numberOfCountedDetails, '
detail elements.  The total calculated order value =  $',
$totalOrderValue)</bpel:from>
            <bpel:to part="response" variable="orderProcessResponse"/>
         </bpel:copy>
         <bpel:copy>
            <bpel:from part="order" variable="orderMessage">
               <bpel:query>OrderHeader/PONo</bpel:query>
            </bpel:from>
            <bpel:to part="PONum" variable="orderProcessResponse"/>
         </bpel:copy>
         <bpel:copy>
            <bpel:from part="order" variable="orderMessage">
               <bpel:query>OrderHeader/CustId</bpel:query>
            </bpel:from>
            <bpel:to part="CustID" variable="orderProcessResponse"/>
```

```
        </bpel:copy>
</bpel:assign>
```

No fim do processo, depois da atividade *forEach* estar concluída, existe um *assign* (CreateResponseMessage) que vai construir a mensagem que irá ser devolvida ao cliente, que terá a forma "*The order contains N detail elements and the proceess traversed M detail elements. The total calculated order value = $Y*", que é obtido usando uma instrução de concatenação, concat(...), para juntar as partes da mensagem.

Depois da mensagem, todas as outras variáveis de retorno são prenchidas com valores calculados (part="PONum" variable="orderProcessResponse"; part="CustID" variable="orderProcessResponse"), usando em todos os casos instruções de cópia.

As variáveis são por fim retornadas ao cliente através da atividade de *reply*, a parte final da atividade *receive/reply* que serviu de base ao processo de negócio.

15. Marcação de Viagem

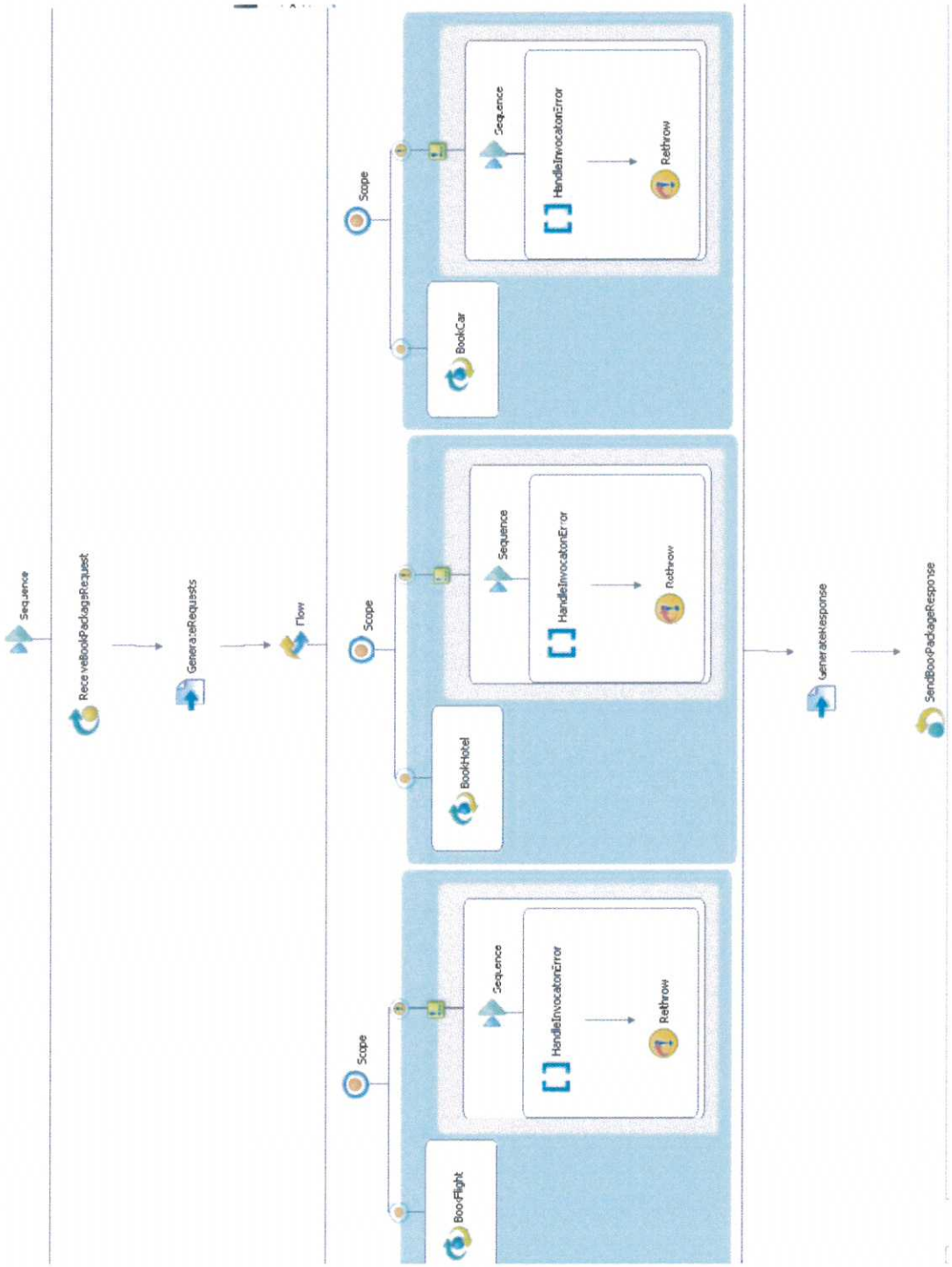


Imagem: Esquema do processo de negócio que permite marcar uma viagem, reservando o voo, o hotel e o carro.

Este processo de negócio permite a reserva de três elementos de uma viagem: os bilhetes de avião, o quarto de hotel e o carro de aluguél, devolvendo ao cliente o resultado das reservas.

O processo invoca vários serviços que vão, cada um, tratar de uma das partes da reserva, possuindo também mecanismos de tratamento de falhas, ou *fault handlers*. Todo o processo está inserido numa atividade de *sequence*, o que quer dizer que todas as suas atividades são executadas de forma sequencial, isto é, apenas quando a atividade anterior terminar é que a seguinte é iniciada.

O cabeçalho da sequência, em XML, apenas possui a indicação de inicio da atividade, sem nome (apesar deste poder ser definido) nem parâmetros de entrada ou saída.

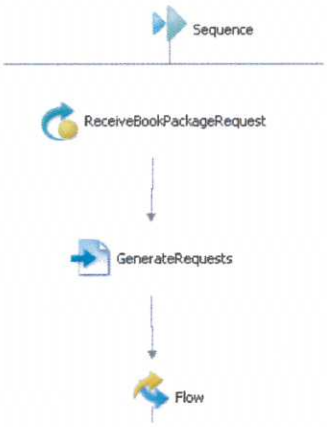


Imagem: Todo o processo se encontra dentro de uma atividade *sequence*. A execução é iniciada com um *receive* seguido de um *assign* que vai copiar os valores do pedido para os vários Web Services que irão ser invocados.

```
<bpel:sequence>
      <bpel:receive createInstance="yes" name="ReceiveBookPackageRequest"
operation="BookPackage" partnerLink="BookPackageLink"
portType="ts:TravelServicePT" variable="BookPackageRequest"/>
      <bpel:assign name="GenerateRequests">
         <bpel:copy>
            <bpel:from part="Document" variable="BookPackageRequest"/>
            <bpel:to part="Document" variable="BookFlightRequest"/>
         </bpel:copy>
         <bpel:copy>
            <bpel:from part="Document" variable="BookPackageRequest"/>
```

Analisando o código XML pode-se ver a definição dos *Fault Handlers*, que utiliza uma atividade *catchAll* para receber qualquer falha que ocorra durante o processo de invocação do WS. Dentro do *scope* do *catchAll* encontra-se um processo vazio, onde a lógica específica de tratamento da falha, caso esta acontecesse numa aplicação real, seria escrita; seguida de uma atividade de *rethrow*, que vai lançar a falha para o *scope* em que está inserido, de modo a ser também tratada pelo *Fault Handler* do processo.

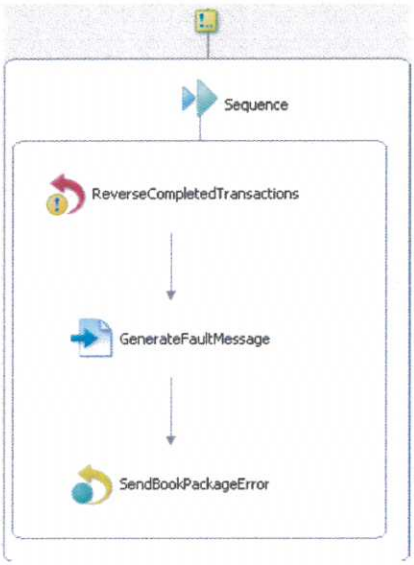


Imagem: Gestor de falhas (*Fault handler)* do processo.

```
<bpel:faultHandlers>
      <bpel:catchAll>
         <bpel:sequence>
            <bpel:compensate name="ReverseCompletedTransactions"/>
            <bpel:assign name="GenerateFaultMessage">
               <bpel:copy>
                  <bpel:from>
                     <bpel:literal>An error occurred while submitting the
order. All transactions have been canceled.
</bpel:literal>
                  </bpel:from>
                  <bpel:to part="Document"
variable="BookPackageFaultVar"/>
               </bpel:copy>
            </bpel:assign>
            <bpel:reply faultName="ts:BookPackageFault"
name="SendBookPackageError" operation="BookPackage"
partnerLink="BookPackageLink" portType="ts:TravelServicePT"
variable="BookPackageFaultVar"/>
         </bpel:sequence>
      </bpel:catchAll>
         </bpel:faultHandlers>
```

Este *Fault Handler* vai receber todas as falhas ocorridas durante a execução do processo e gerar uma mensagem de erro, ao mesmo tempo que cancela todas as atividades deste mesmo já concluídas, ficando estas sem validade (algo que é feito com uma atividade de *compensate*,"ReverseCompletedTransactions").

Este processo mostra assim muitos dos mecanismos do BPEL existentes para lidar com falhas nos processos, mecanismos estes que podem ser locais, numa determinada atividade (como neste caso nos *invokes* dos WS) ou trecho do processo; como globais, gerindo todas as falhas do processo de um modo geral, usando *Fault Handlers* ligados ao processo por inteiro.

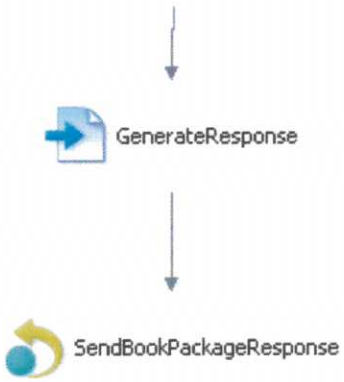


Imagem: No final do processo é escrita uma mensagem com os dados das reservas para o cliente ou com um *log* dos erros que ocorreram, em caso de falha

```
<bpel:assign name="GenerateResponse">
        <bpel:copy>
           <bpel:from>concat( 'Flight: ', $BookFlightResponse.Document ,
'- Hotel: ', $BookHotelResponse.Document , '- Rental Car: ',
$BookRentalCarResponse.Document )</bpel:from>
           <bpel:to part="Document" variable="BookPackageResponse"/>
        </bpel:copy>
     </bpel:assign>
     <bpel:reply name="SendBookPackageResponse" operation="BookPackage"
partnerLink="BookPackageLink" portType="ts:TravelServicePT"
variable="BookPackageResponse"/>
        </bpel:sequence>
```

Para finalizar, uma atividade de *reply* vai devolver a resposta gerada

no *assign*, "GenerateResponse", com os dados das reservas efetuadas (concat('Flight: ', $BookFlightResponse.Document , '- Hotel: ', $BookHotelResponse.Document , '- Rental Car: ', $BookRentalCarResponse.Document)) ou, caso tenha ocorrido uma falha no processo, um *log* do erro.

16. Conhecendo o Oracle SOA Suite

Primeiramente, acesse o console do WebLogic Server (Admin), vá à pasta Serviores e inicialize o servidor : **soa_server1.**

Servers (Filtered - More Columns Exist)

Start | Resume | Suspend | Shutdown | Restart SSL — Showing 1 to

Server	Machine	State	Status of Last Action
AdminServer(admin)		RUNNING	None
bam_server1	LocalMachine	SHUTDOWN	None
soa_server1	LocalMachine	RUNNING	TASK COMPLETED

Start | Resume | Suspend | Shutdown | Restart SSL — Showing 1 to

passo1

Apos o start, clique em cima do Server e vá à aba de Deployments. Verifique se todos estão feitos corretamente.

Applications and Modules Deployed to this Server

Showing 1 to 10 of 24 Previous | Next

Name	State	Type	Deployment Order
AqAdapter	Active	Resource Adapter	324
⊞ b2bui	Active	Enterprise Application	313
⊞ BPMComposer	Active	Enterprise Application	384
⊞ composer	Active	Enterprise Application	315
DbAdapter	Active	Resource Adapter	322
⊞ DefaultToDoTaskFlow	Active	Enterprise Application	314
⊞ DMS Application (11.1.1.1.0)	Active	Web Application	5
FileAdapter	Active	Resource Adapter	321
FtpAdapter	Active	Resource Adapter	325
JmsAdapter	Active	Resource Adapter	323

Showing 1 to 10 of 24 Previous | Next

passo2

JDeveloper

A IDE padrão do Oracle SOA Suíte é o JDeveloper, entretanto, é necessário a instalação de uma extensão, SOA-Composite, para que o mesmo habilite as funcionalidades relativas à SOA.

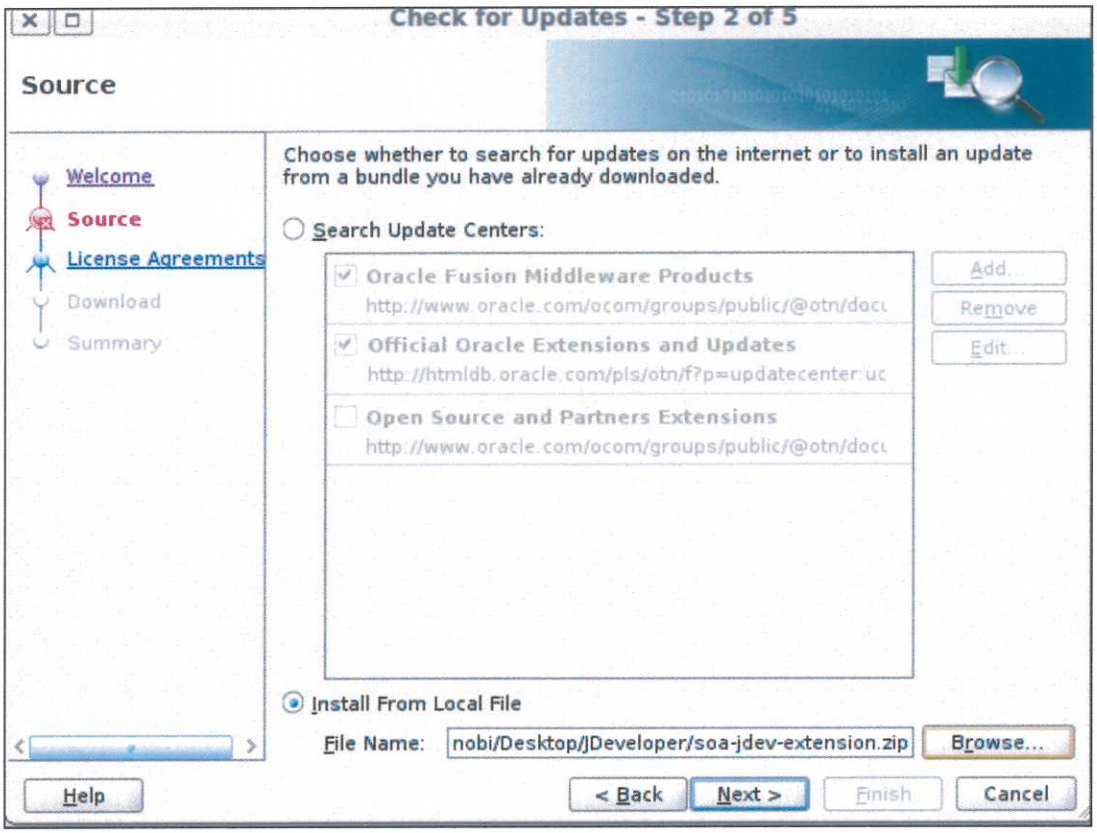
Primeiramente instale o arquivo do JDeveloper, rodando o comando:

java -jar jdedev.xxxxx.jar

Após essa etapa concluída, instale o plugin.

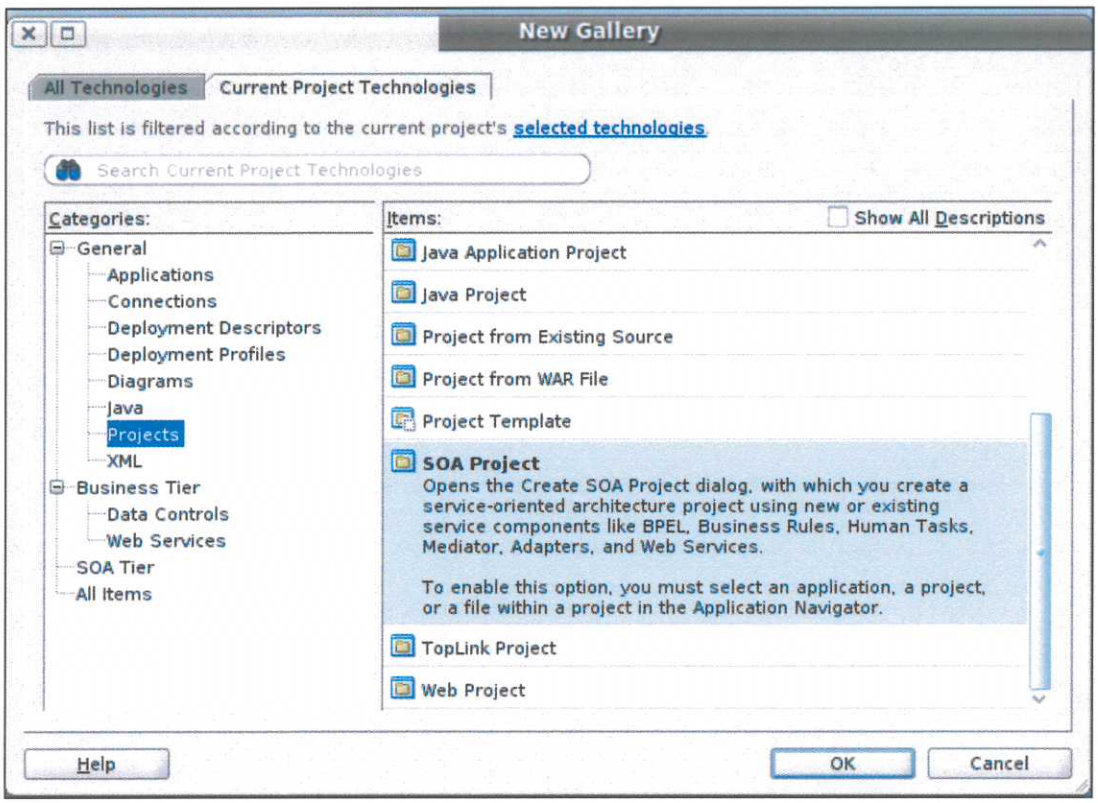
Vá até o menu Help, clique em Check Updates, e escolha a opção para instalar à partir de um arquivo.

Localize o plugin contido no mesmo diretório (.zip) e instale.



17. Criando um projeto simples

1. Crie um novo projeto do tipo SOA Project.

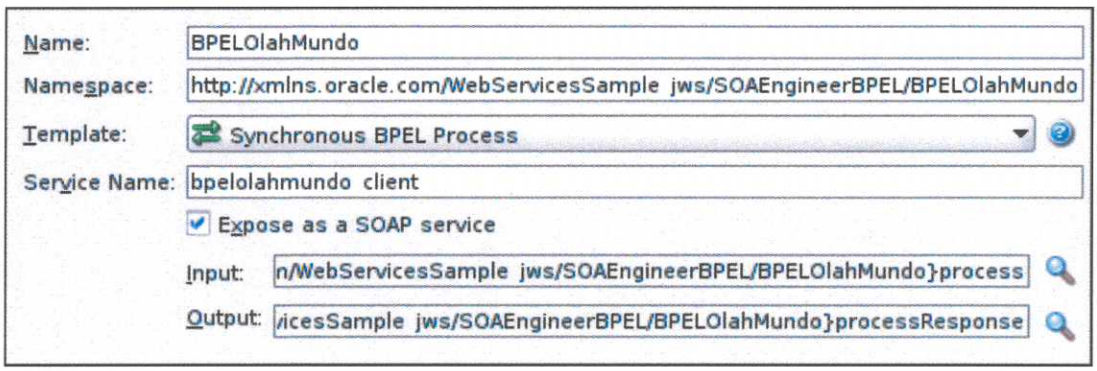


passo1

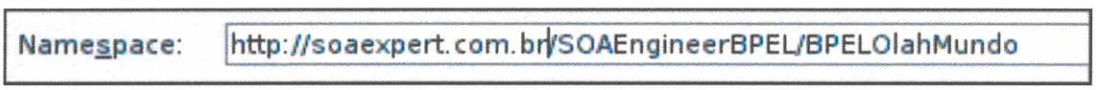
2. Defina o package default da aplicação: **br.com.soaexpert**

3.Crie um Composite com um BPEL Process

4. Crie um BPEL Síncrono com o nome de Olah Mundo: BPELOlahMundo.

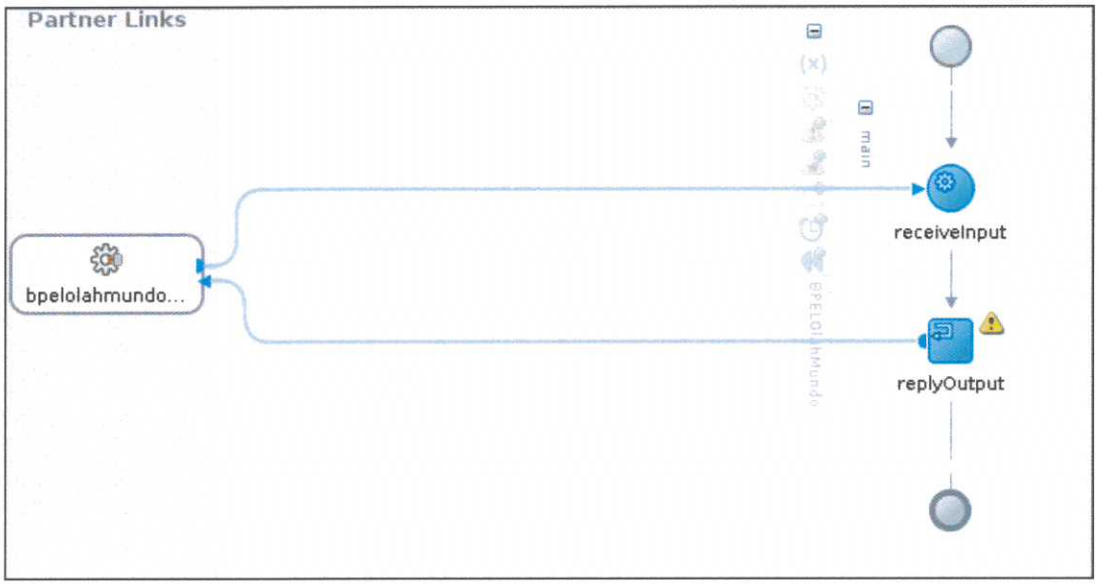


5. Defina o Namespace : soaexpert.com.br



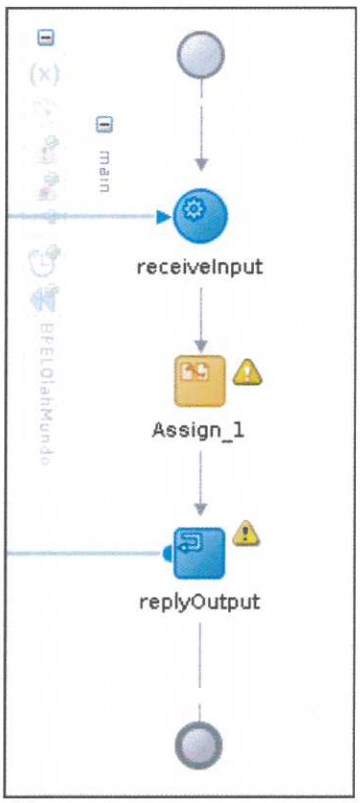
passo5

6. Dê ok e Crie o processo.



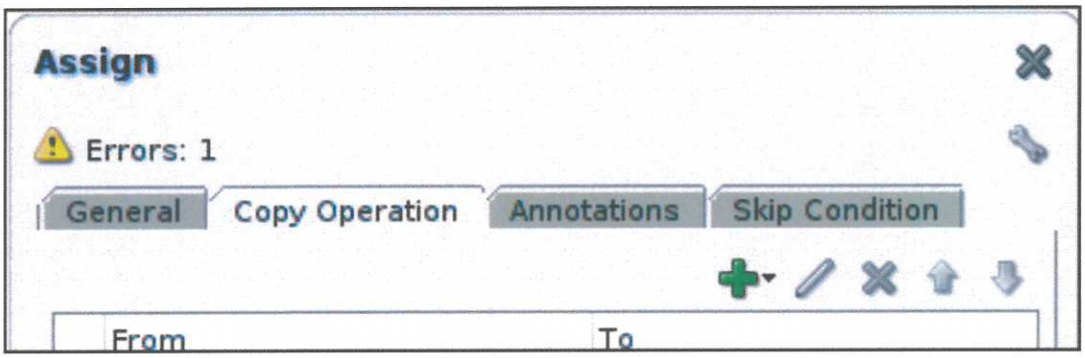
NOTA: A variável replyOutPut não foi inicializada, por isso o alerta !

7. Inclua uma nova atividade entre o fluxo receive e reply : **Assign**:

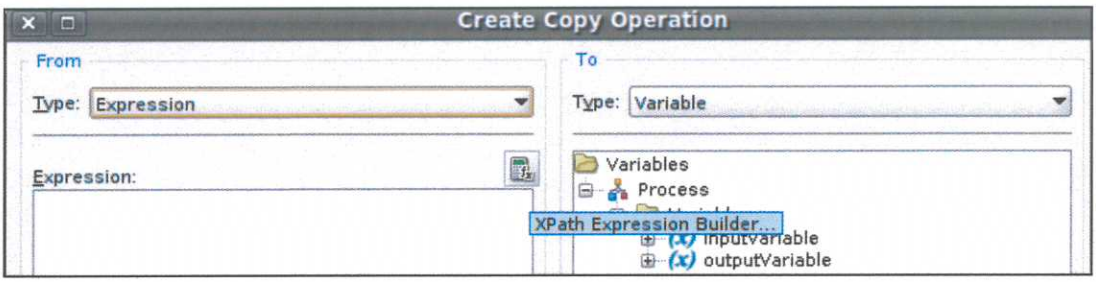


8. Inclua uma nova atividade entre o fluxo receive e reply : **Assign**:

9. No Assign, clique na aba Copy e no mais à direita.

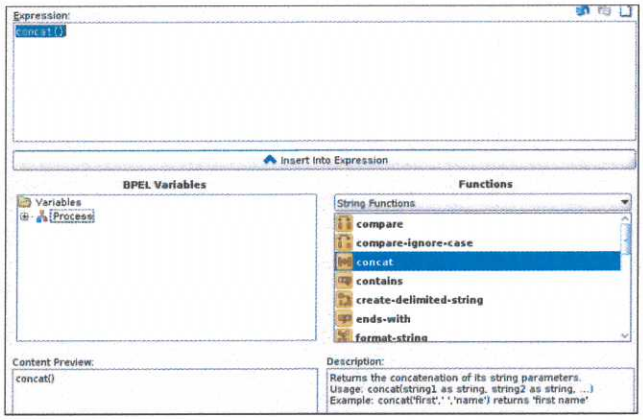


10. No tipo (FROM) escolha o Type Expression e clique no ícone do XPath Expression Builder.



passo10.

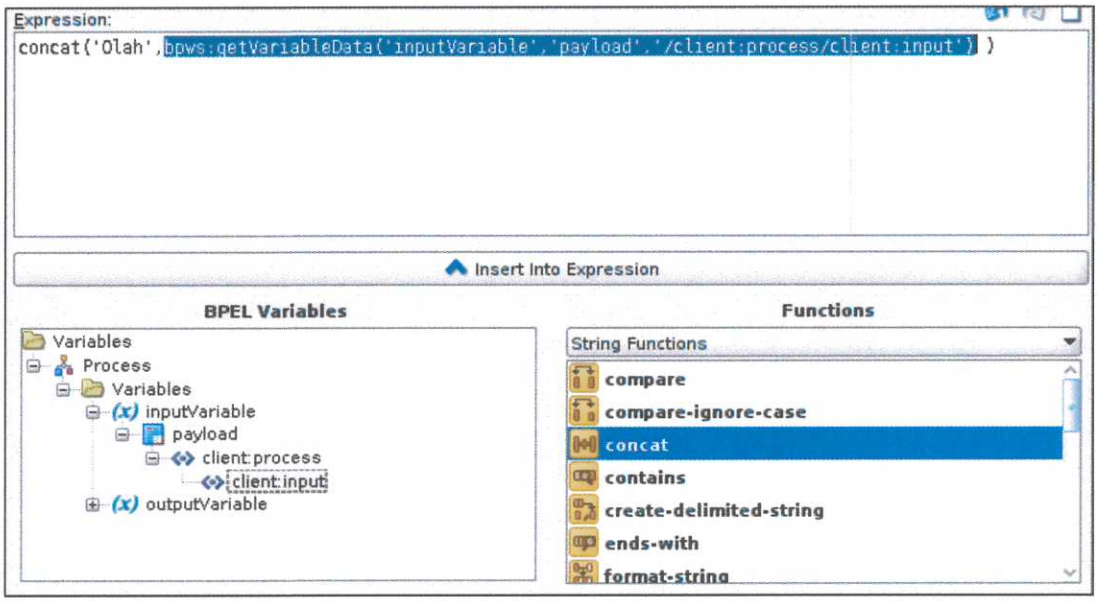
11. No combox à esquerda, escolha functions do tipo String:



passo11

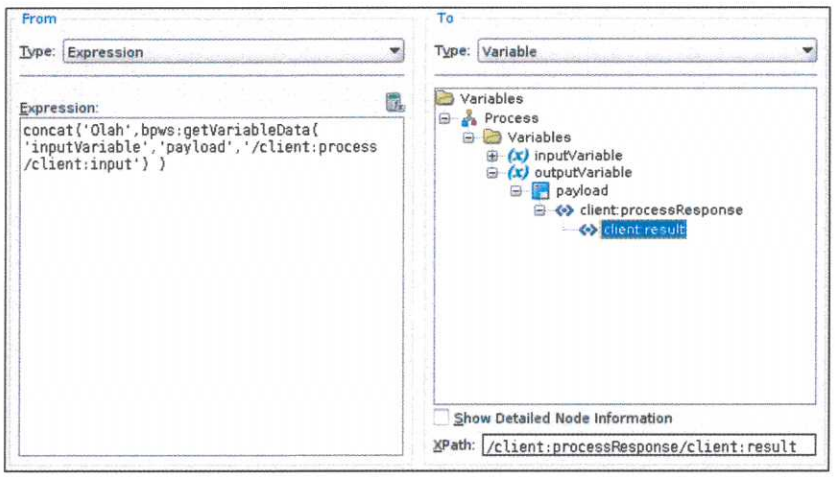
O concat é formado pela soma de 2 Strings. Coloque a expressão: concat('Olah',)

12. Navegue no nó da varievel input até o elemento client:input e adicione o mesmo apos a virgula, para formar a 2 string.



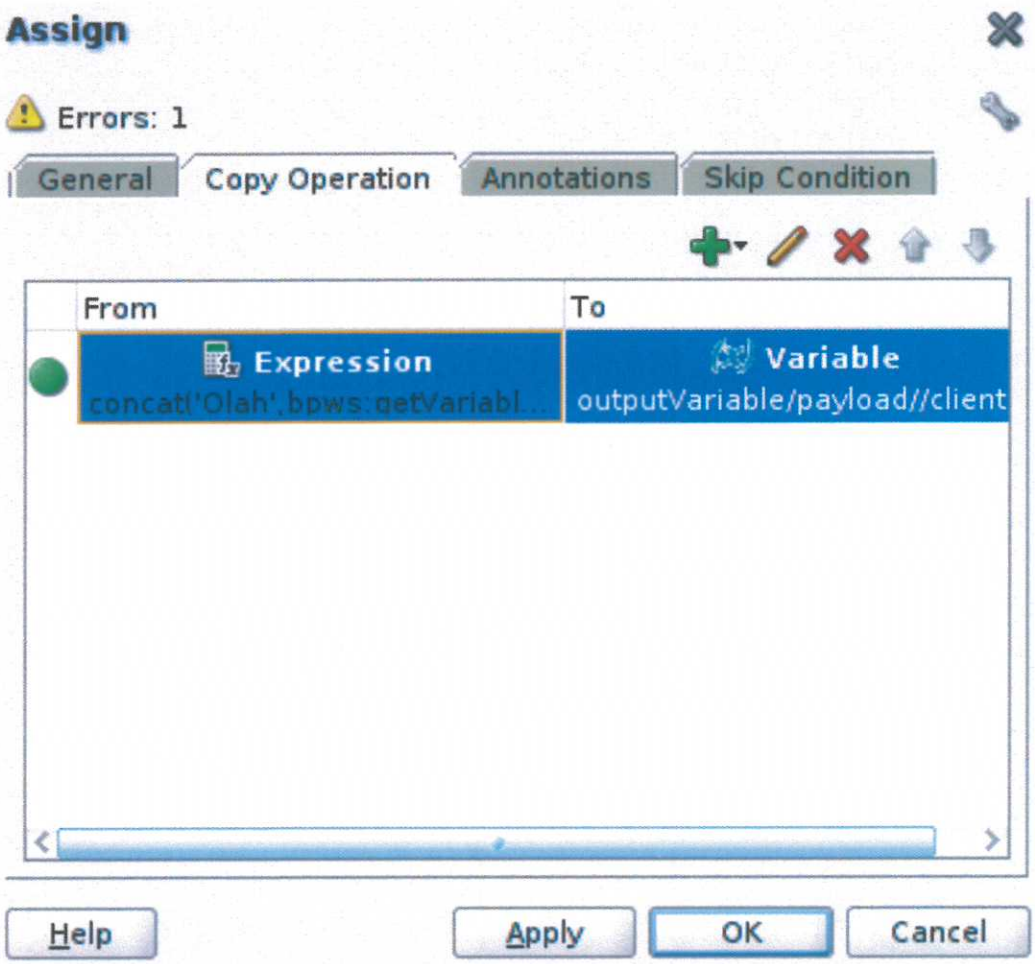
passo12.

13. No lado direito, escolha o Output (Target) e navegue até o elemento client:result.



passo13.

14. IMPORTANTE !! Não se esqueça de aplicar (Apply) a modificação, antes do botão OK !!



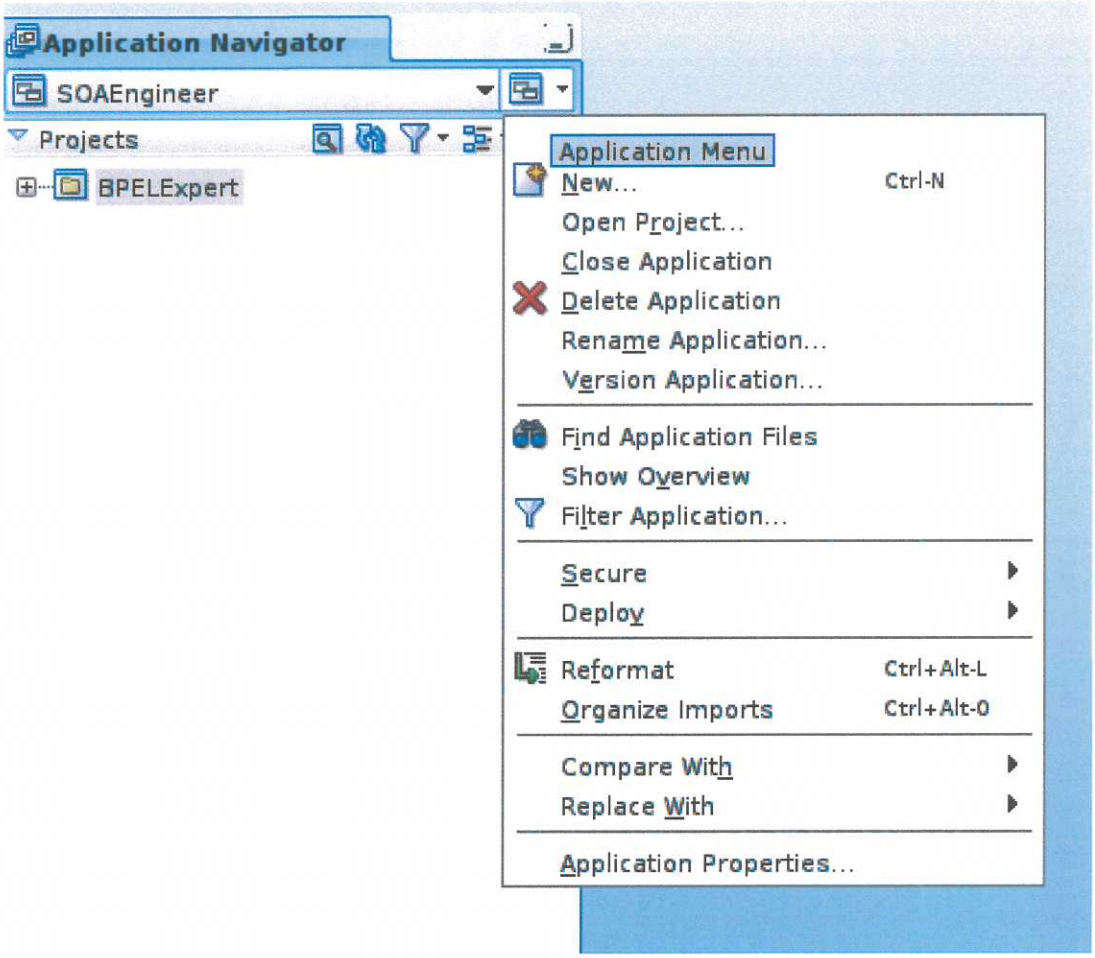
Clique no Apply !! antes !!

18. Exercício de chamada síncrona

Exercício: Entendendo o funcionamento de **PartnerLink** e Transformações em cima do BPEL

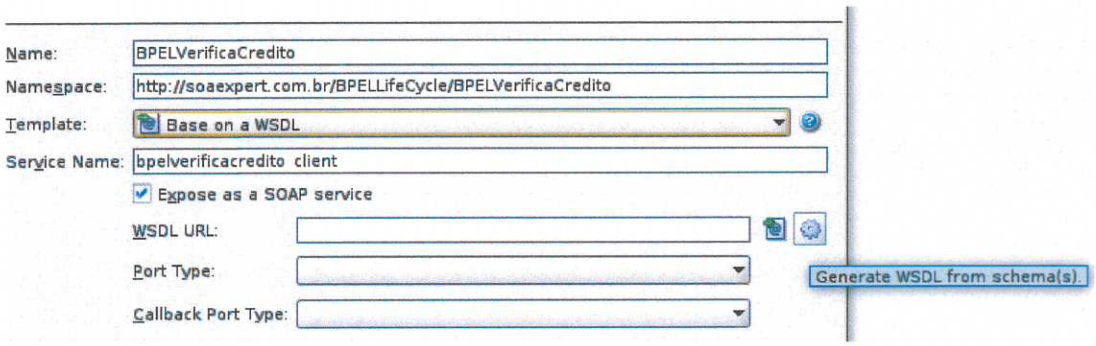
O objetivo do exercício a seguir é criar uma verificação de crédito no serviço exposto no barramento (OSB), utilizando a linguagem BPEL para consulta e manipulação das variáveis.

1. Na aplicação existente, crie um novo projeto, utilizando o botão o menu :



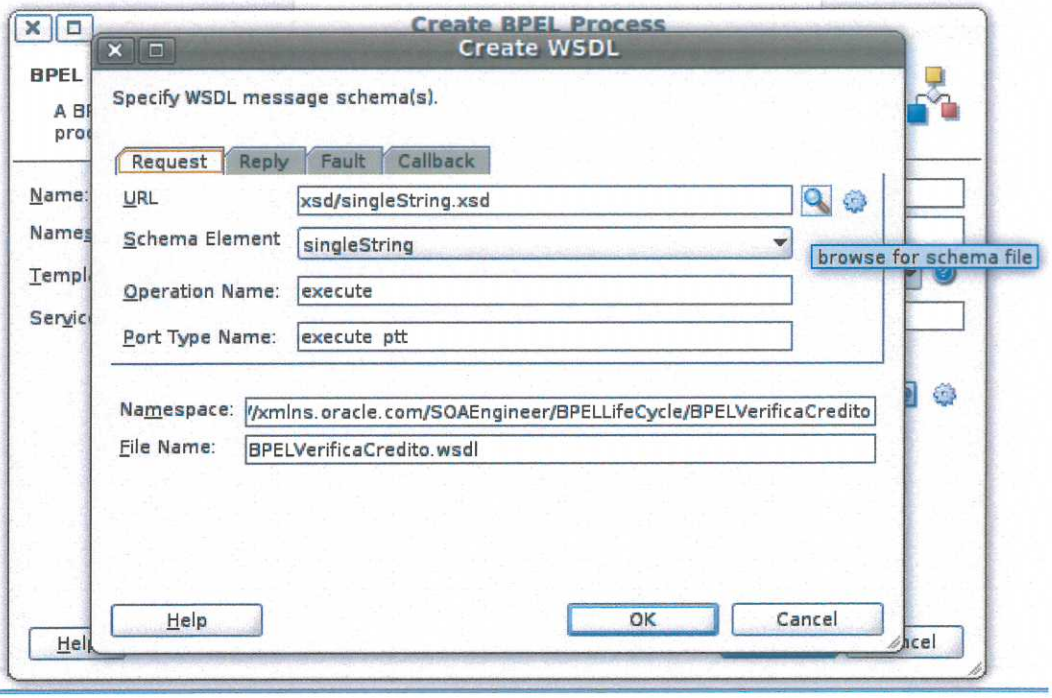
2. Arraste um componente BPEL, para o centro do Components no "composite.xml" – SCA.

3. Nomeie o mesmo como BPELVerificaCredito, defina o namespace: **soaexpert.com.br** para o mesmo e escolha ser baseado em WSDL.



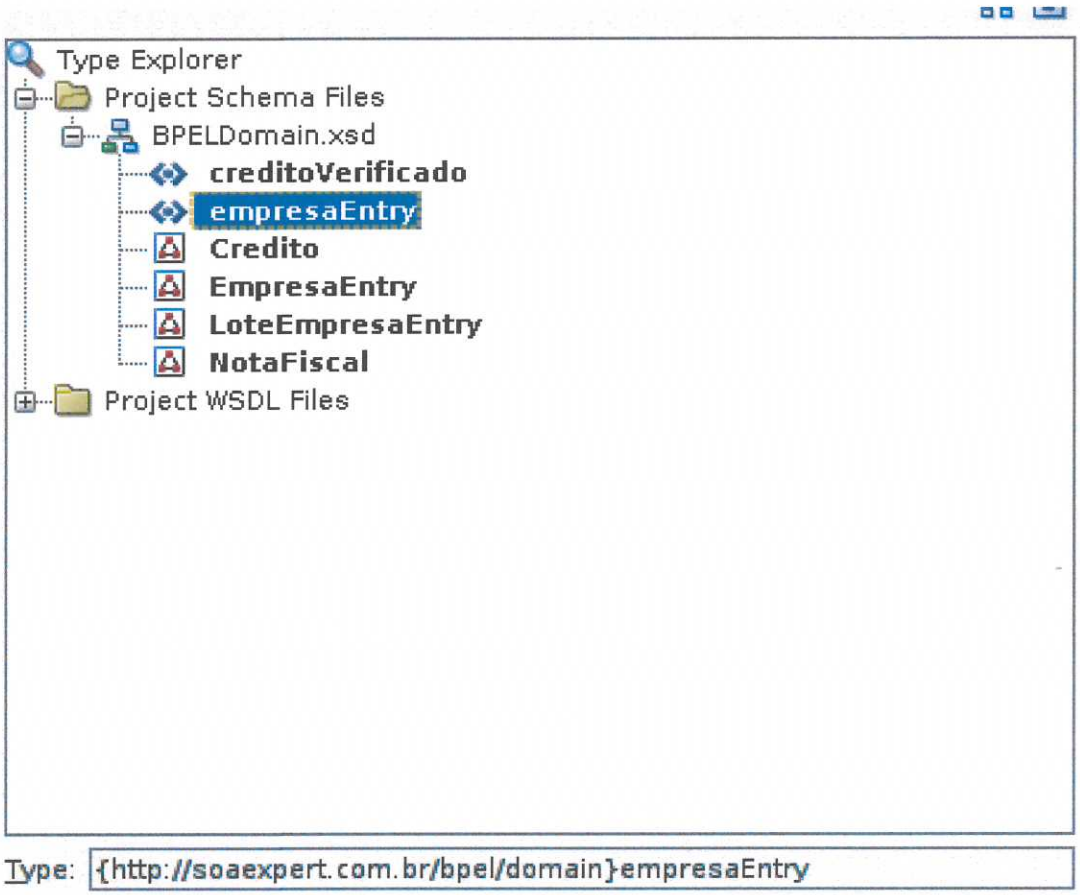
4. clique na engrenagem para gerar um contrato para o processo. (WSDL).

5. Localize o Schema (XSD) a ser utilizado para tipos de mensagem do contrato.

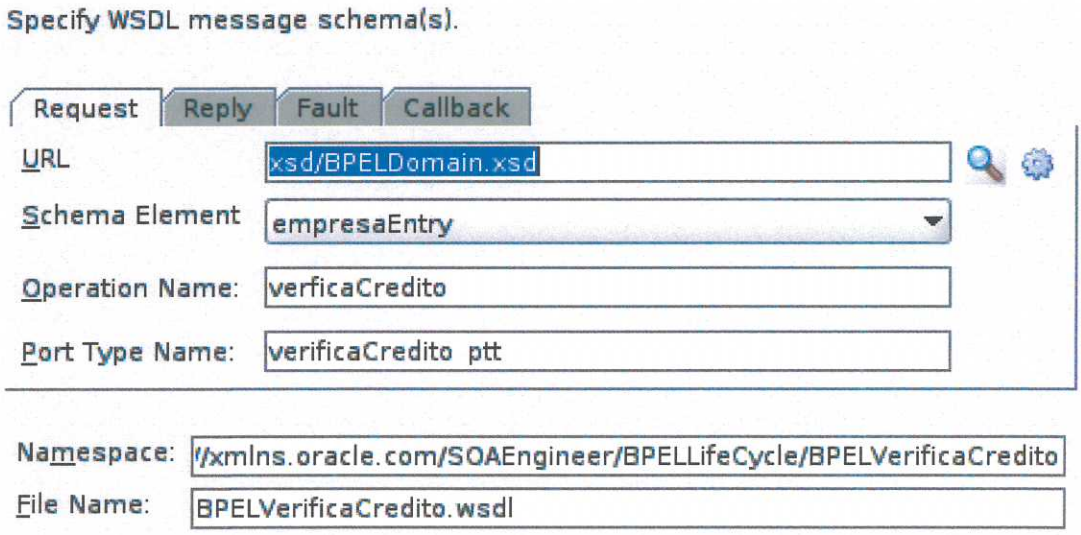


6. Importe um Schema do arquivo, localize o arquivo BPELDomain.xsd.

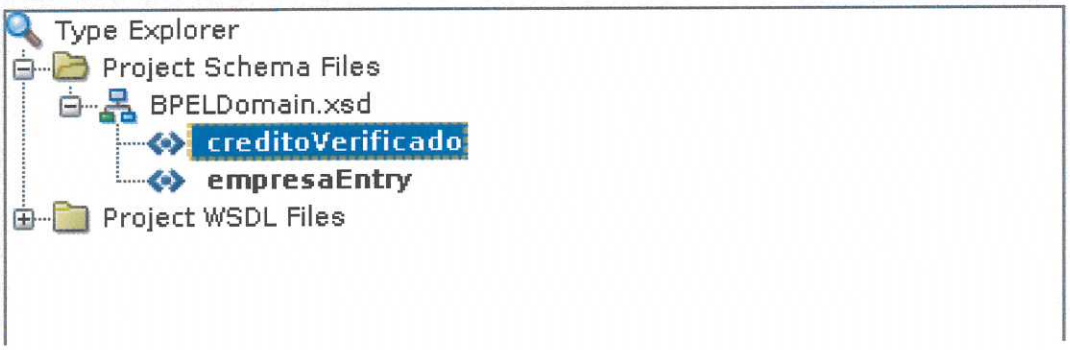
7. Escolha o elemento de entrada: **EmpresaEntry**



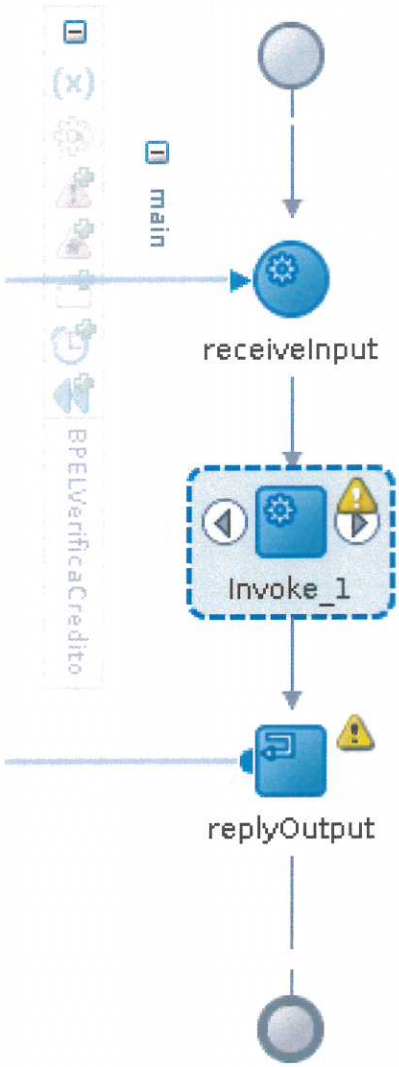
8. Na aba Request, prencha ainda o nome da operação: verificaCredito e o PortType Name: verificaCredito_ppt



9. Na aba Reply, realize o mesmo procedimento e localize o elemento **creditoVerificado**.



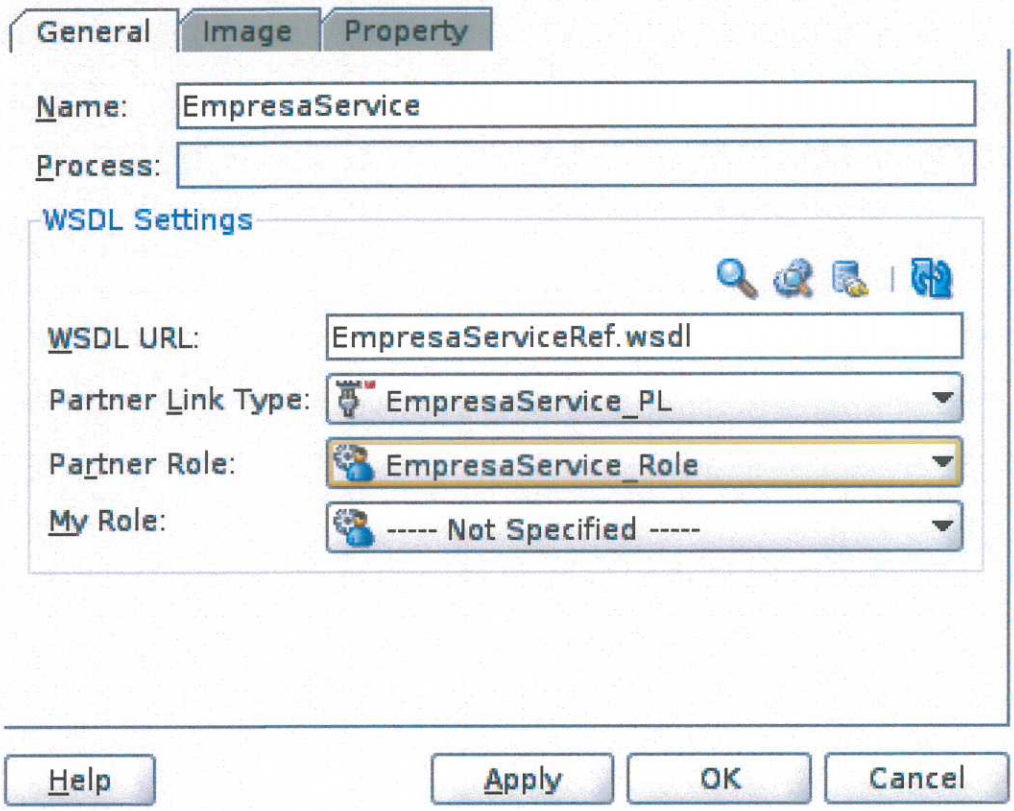
10. Insira uma atividade Invoke entre as atividades **receive** e **reply**.



11.Copie os arquivos que serão carregados.

12. Crie um partner Link, arrastando o componente para a raia cinza do lado direito.

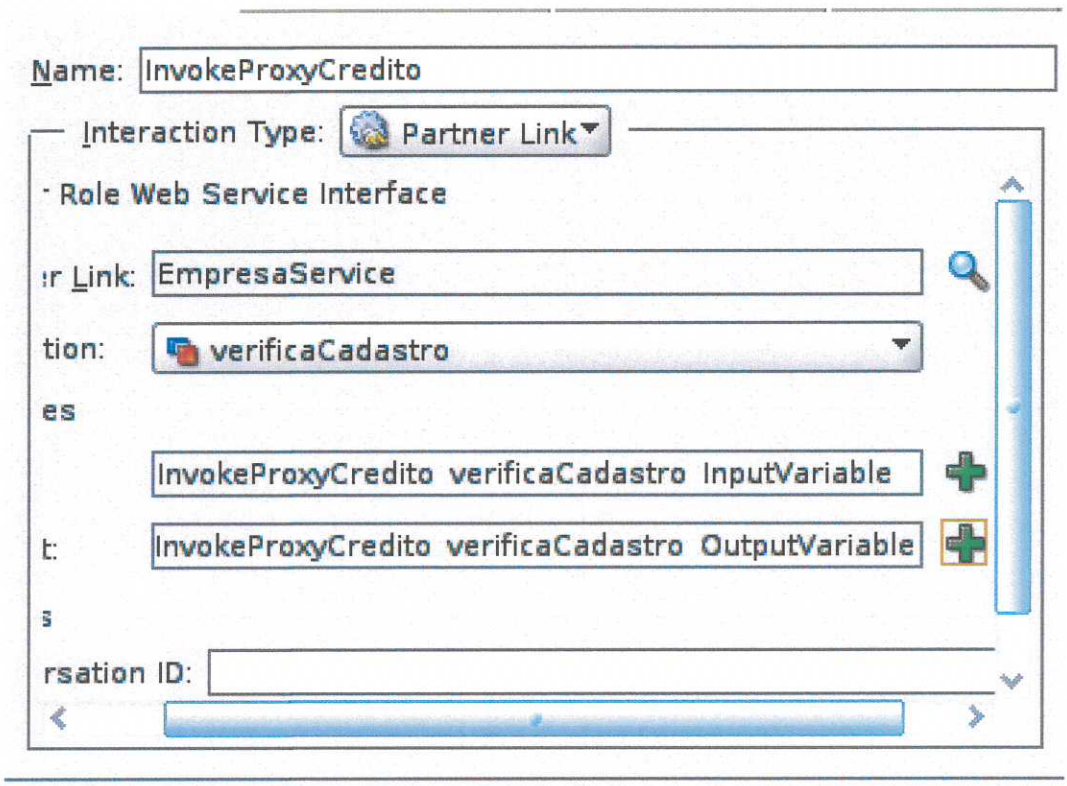
13. Localize o contrato do Proxy de consulta, disponibilizado em aula.



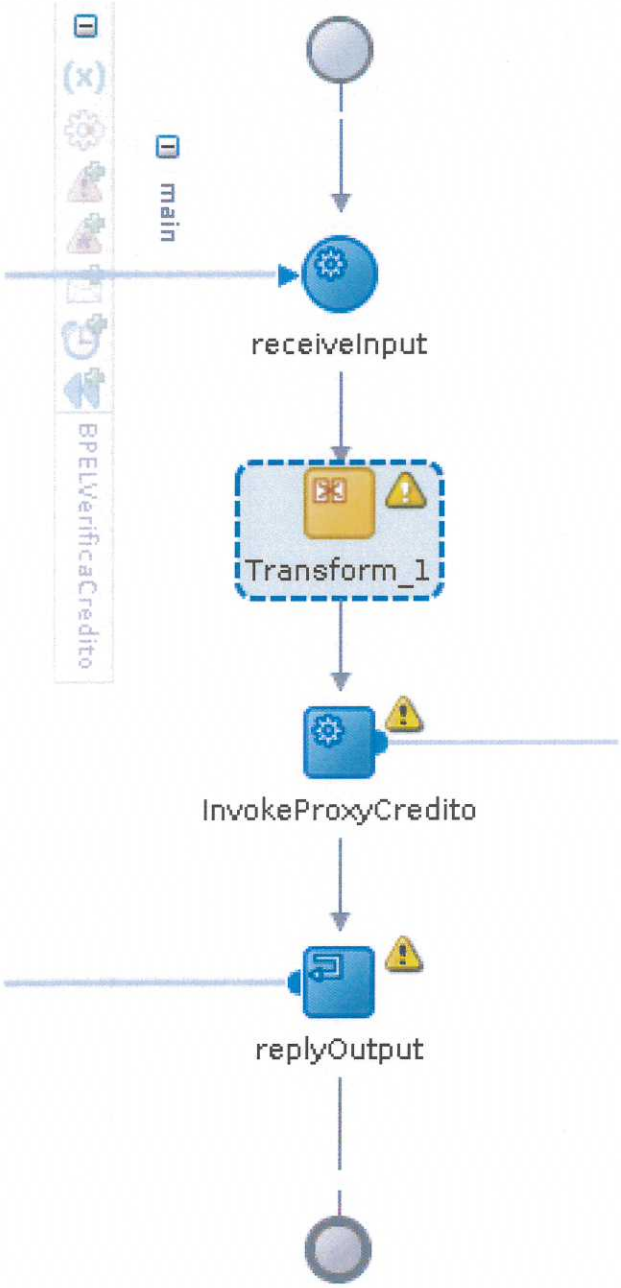
PS: Esse procedimento irá gerar automaticamente um partner link para o serivço, já que o mesmo não possui um.

14. Localize o PartnerLink gerado, **EmpresaService**

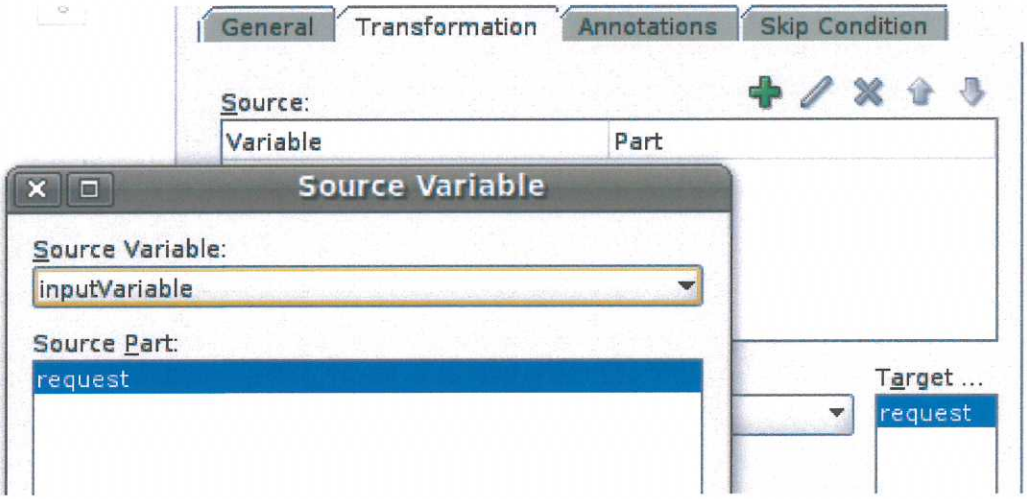
15. Escolha a operação verificaCadastro, e crie as variáveis de entrada e saída, utilizando o botão de + em verde, não se esqueça de aplicar (Apply) antes do ok.



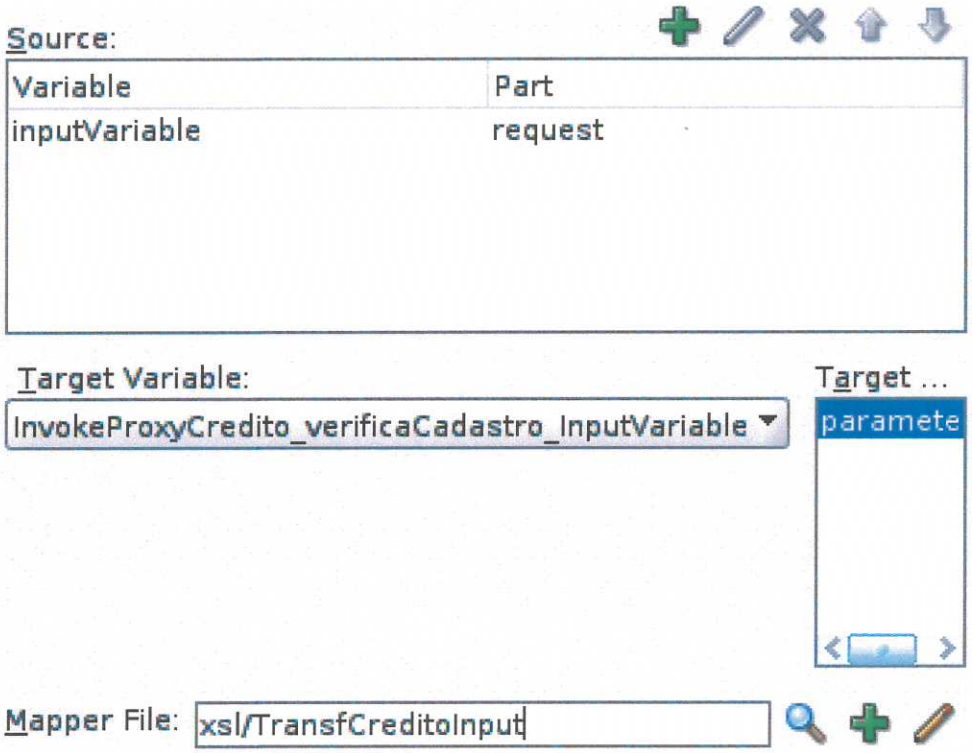
16. Crie uma transformação antes do fluxo de entrada,pois os tipos são diferentes entre os serviços.



17. Na aba de transformação, na variável de source, localize a inputVaribale:



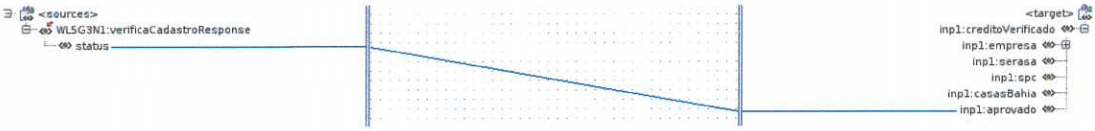
18. Atribua à variável criada para invocar o PartnerLink



19. Preencha o nome da transformação e clique no + na foto acima, para abrir o console de edição da XSLT

20. Ligue os tipos da origem para o destino, como demonstrado:

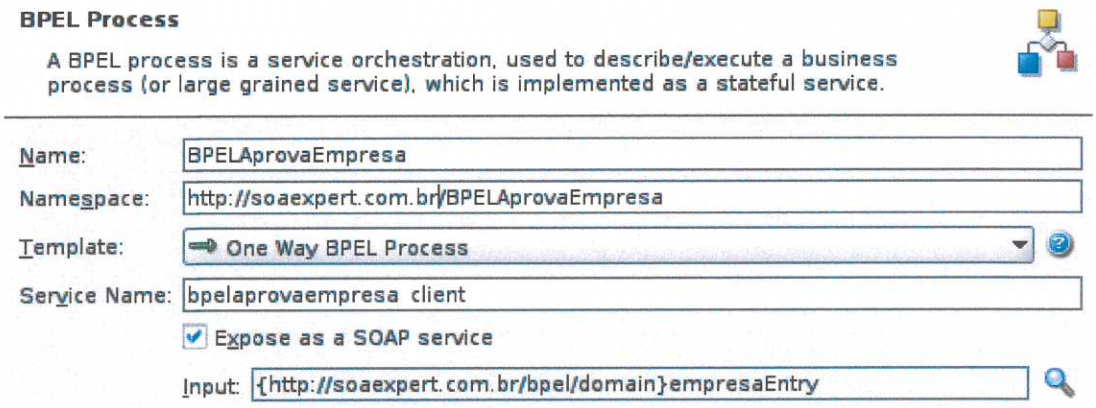
21. Repita o processo de transformação para a variável de Output definida no contrato do BPEL, com o dado de retorno do serviço:



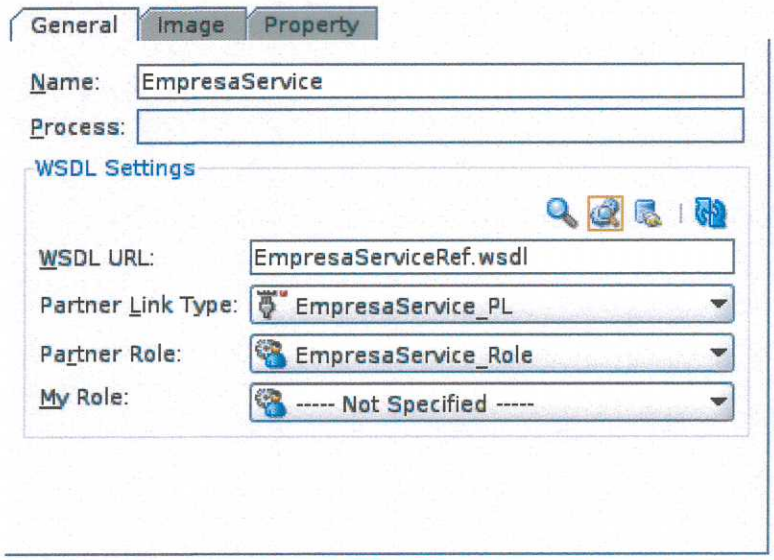
22. Deploy o composite e teste no Enterprise Manager.

19. Exercício de chamada Assíncrona

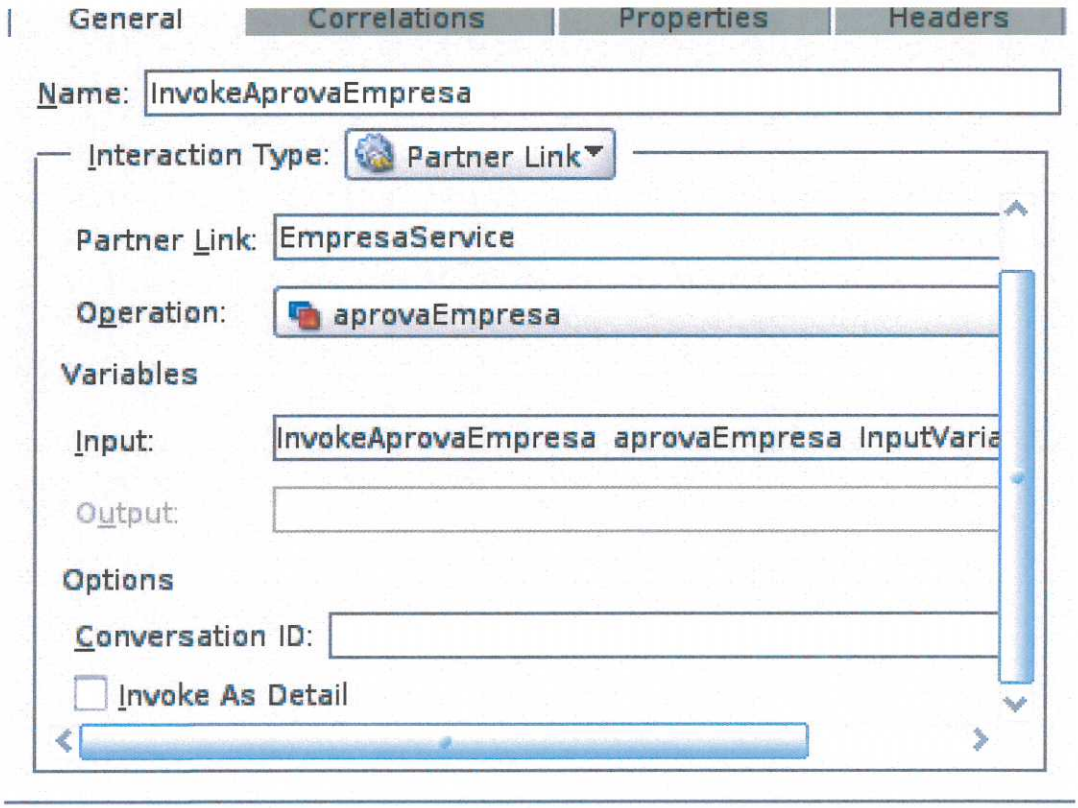
1. Crie um novo processo BPEL, como no exercício anterior, arrastando o componente BPEL para o centro do composite.
2. Nomeie o BPEL como **BPELAprovaEmpresa.**
3. Defina o namespace padrão, e escolha o template assíncrono de OneWay.
4. Localize através da lupa, o elemento empresaEntry definido no BPELDomain.xsd



5. Clique e arraste o componente de PartnerLink e localize o mesmo criado no exercício anterior.



6. Crie uma atividade Invoke para chamada do **PartneLink**, localize o mesmo e escolha a operação **aprovaEmpresa.**

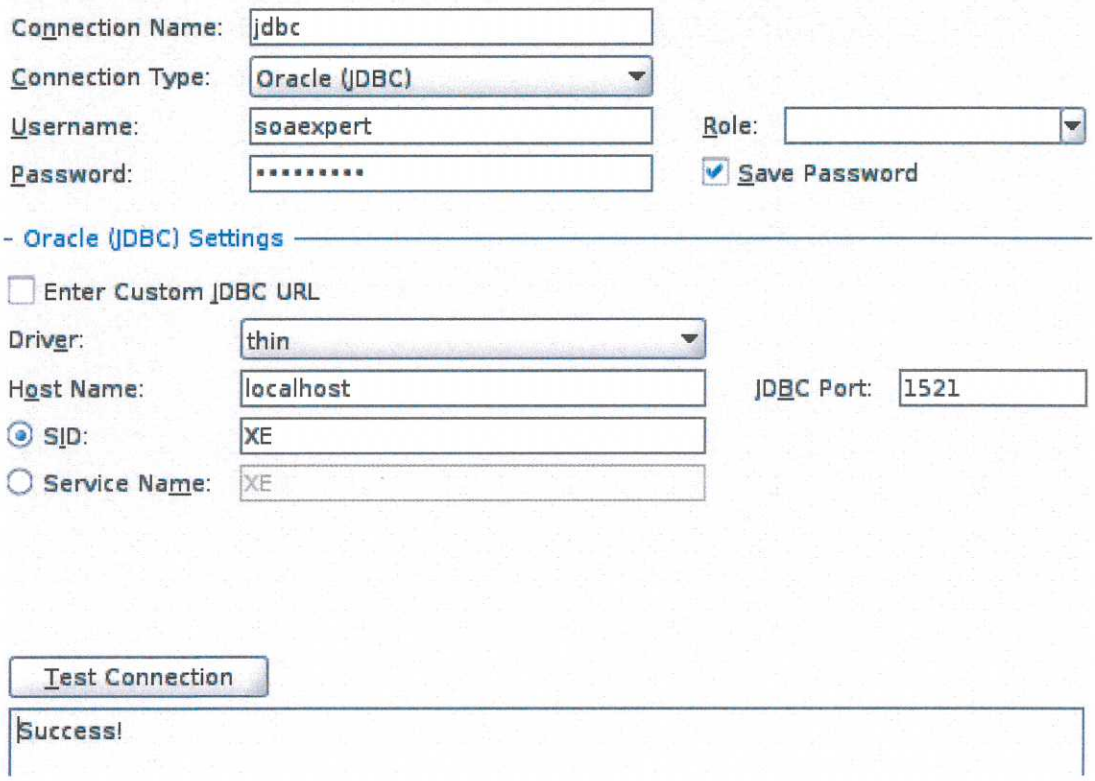


7. Crie uma transformação antes da invocação, a fim de realizar corretamente o payload de entrada.

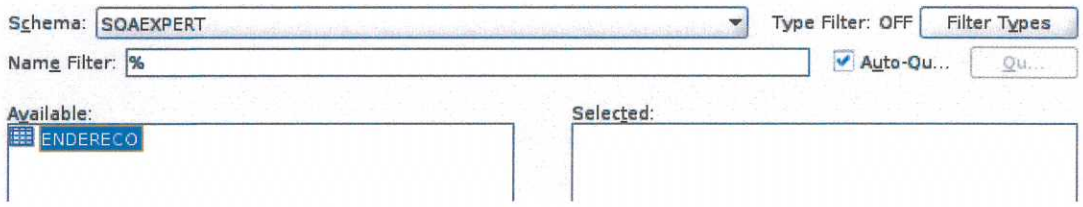
8. Valide o processo e faça o deploy. Teste a aplicação no Enterprise Manager.

20. Utilizando um processo Assíncrono com Adaptador

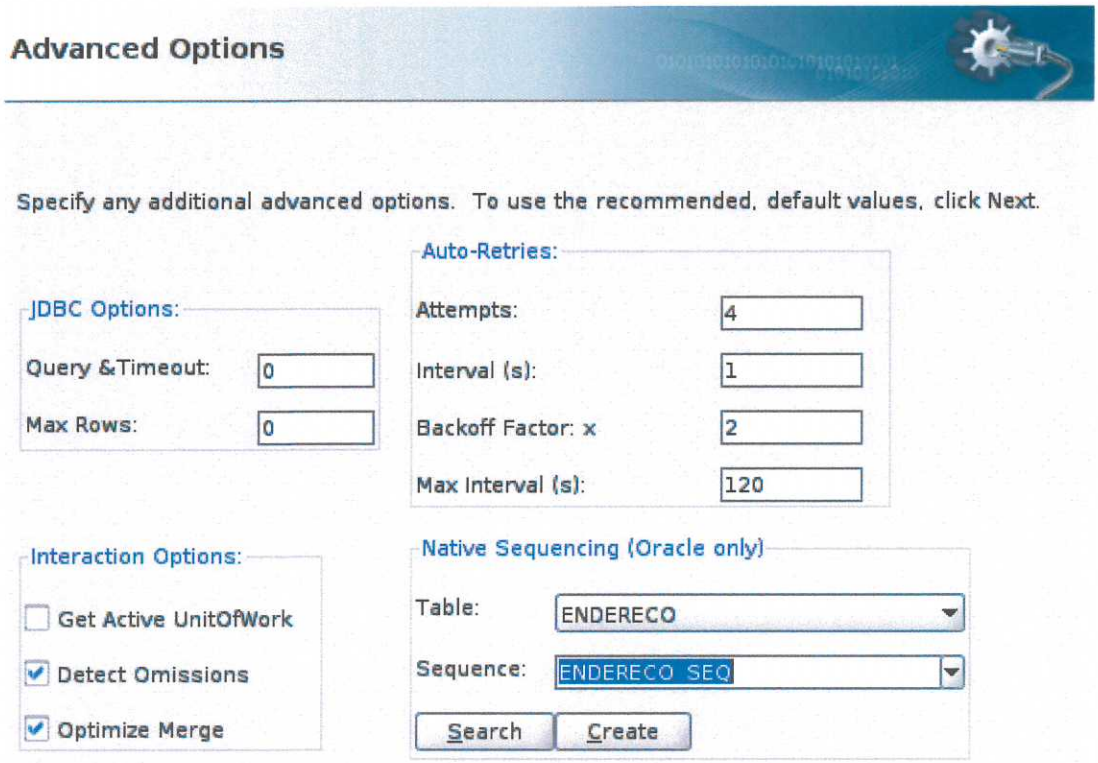
1. Crie um processo chamado BPELCadastraEmpresa.
2. Arraste o Adaptador de banco, para a raia direita e nomeie como EmpresaData.
3. Insira seus dados de configuração local e teste a conexão:



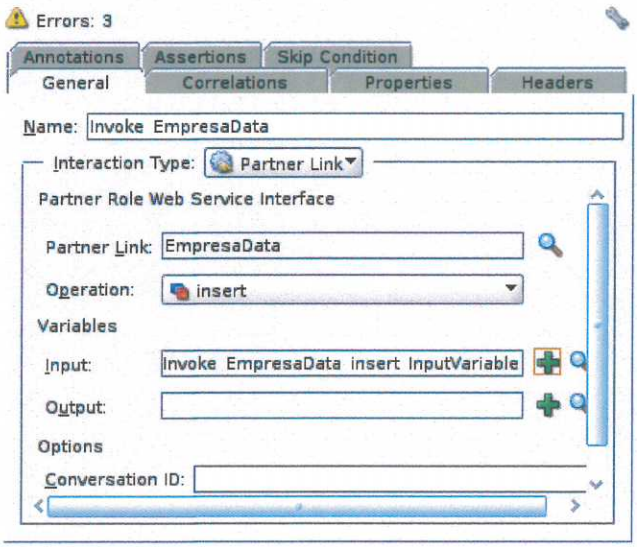
4. Clique no botão Import Tables.
5. Clique no Auto-Query, para carregar automaticamente as tabelas do usuário:



6. Marque a tabela Empresa e na próxima opção, localize a Sequence.



7. Adicione uma atividade Invoke, escolha o partner Link criado (EmpresaData) e a operação insert. Adicione também uma variável de input à chamada.



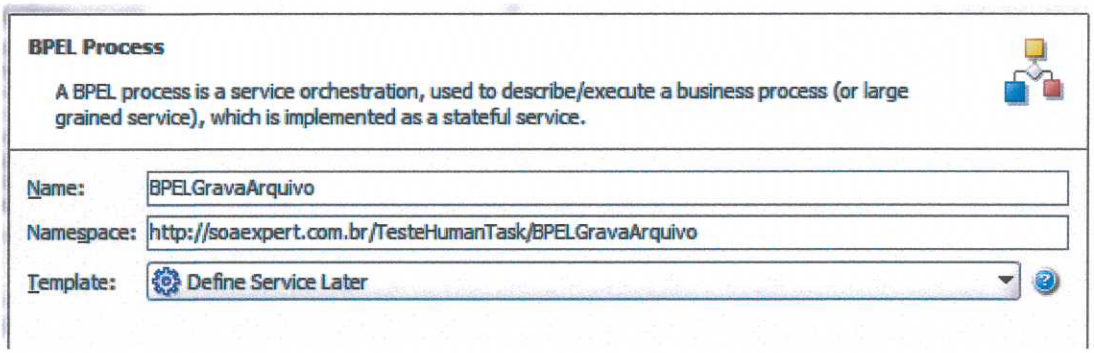
8. Adicione uma transformação de tipos, antes da atividade de invocação.
9. Deploy e teste no Enterprise Manager.

21. Trabalhando com o adaptador de Arquivos

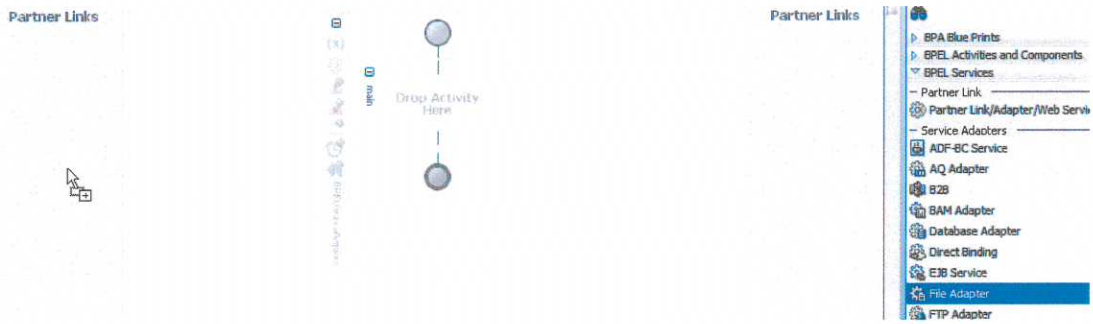
O file Adaptor, é um componente de manipulação de estruturas de arquivos e transporta automaticamente para o processo de negócio, um streaming de bytes, estrutura textual e incorpora em si, transformações de tipos.

Objetivo do exercício: Construir um processo BPEL que lê um diretório contendo um arquivo XML de empresa, e grava o mesmo de forma automática, numa base de dados.

1. Crie um novo processo chamado: BPELGravaArquivo e escolha definir o serviço posteriormente.



2. Procure o componente File Adapter e arraste para a raia esquerda de Partner Link, do designer.



3. Nomeie o serviço como FileEmpresaService

Service Name: FilleEmpresaService

4. Defina os tipos de entrada posteriormente:

Interface: (•) Define from operation and schema (specified later)

( ) Import an existing WSDL

WSDL URL:

Port Type:

5. Escolha a operação READ do componente, pois a idéia do exerciio é ler um arquivo.

Operation Type: (•) Read File

( ) Write File

( ) Synchronous Read File

( ) List Files

Operation Name: Read

6. Defina os diretórios de leitura do arquivo e de arquivamento, após processamento:

Directory for Incoming Files (physical path):
/soaexpert/filebpel/input

☑ Process files recursively

☑ Archive processed files
Archive Directory for Processed Files (physical path):
/soaexpert/archive [Browse]

☑ Delete files after successful retrieval

7. Ecolha o tipo de entrada do arquivo, para XML -

Name Patterns are specified with: ◉ File Wildcards (po*.txt) ○ Regular Expressions (po.*\.txt)

Include Files with Name Pattern: *.xml

Exclude Files with Name Pattern:

8. Ajuste a freqüência de captura de arquivos, para 30 segundos e para não manter arquivos por mais de 2 minutos.

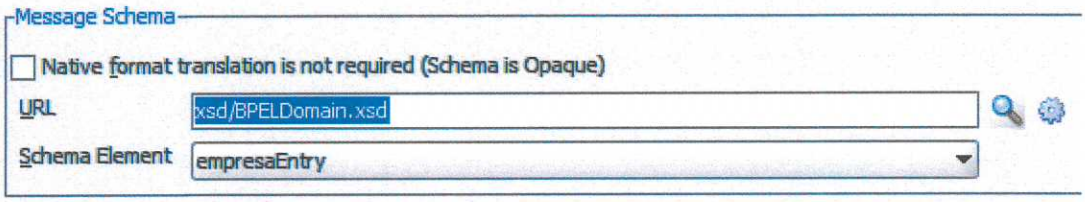
Enter the file polling parameters for the Read File operati

Polling Frequency: 30 seconds

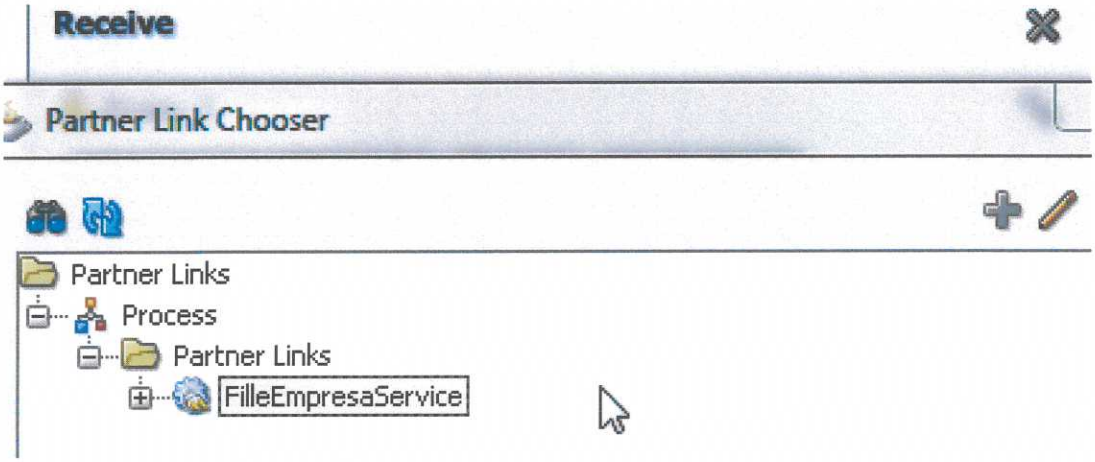
Minimum File Age: 2 minutes

☐ Use trigger file

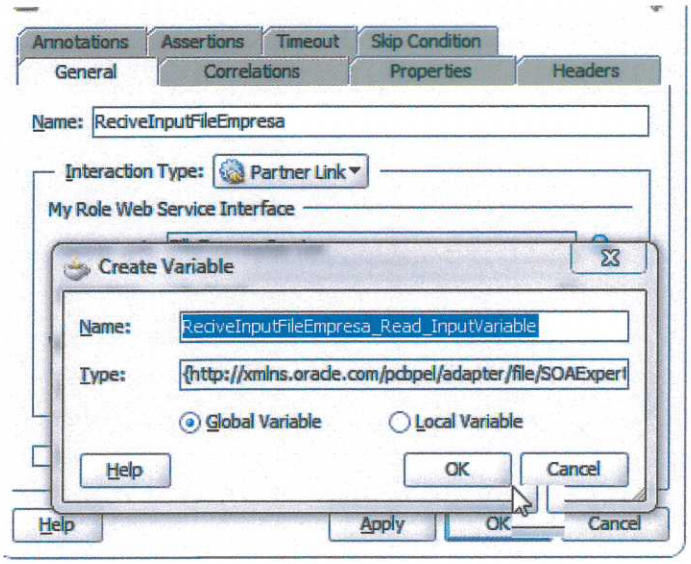
9. Escolha a estrutura de leitura do arquivo, o elemento **empresaEntry,** definida no BPELDomain.xsd



10. Crie uma atividade Recieve, e localize o PartnerLink, definido para captura de arquivos.



11. Defina a variável de entrada do Processo BPEL



12. Crie um Database Adapter do lado direito, seguindo os passos do exercício anterior, entretanto, localize o adaptador já criado.

13. Crie uma nova atividade Invoke e atribua o payload de entrada do processo BPEL (Input) ao input da invocação do serviço de dados.

14. Faça o Deploy e carregue o arquivo empresaEntry.xml para a estrutura de leitura de diretório definida no exercício.

22.Trabalhando com WorkFlow Humano

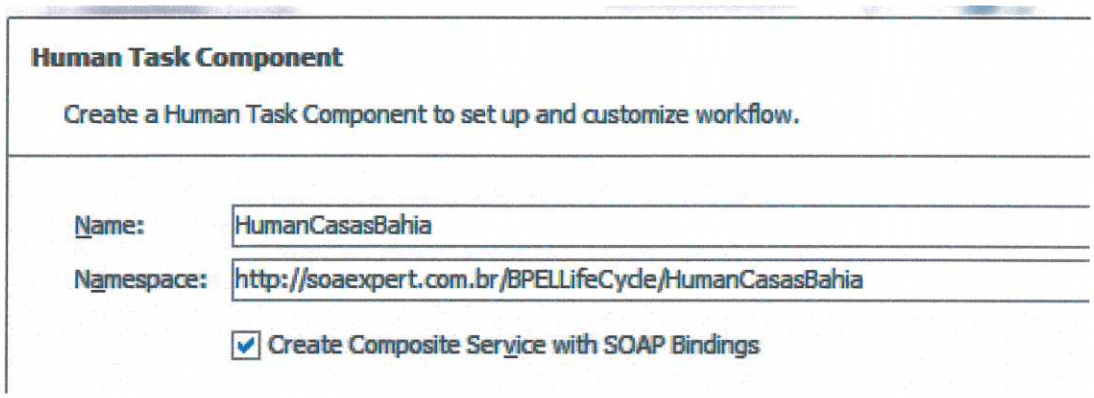
O BPEL possui uma especificação, BPEL4WS que serve ao propósito de disparar processos humanos, de negócio.

O exercício a seguir, demonstra o conceito de forma simples.

Objetivo: Criar uma verificação de crédito Manual.

1. Arraste um componente Human Task para o centro dos componentes:

2. Defina o nome da Human Task como Casas Bahia e escolha criar o composite com SOAP Bindings.



3. Na aba General, preencha as propriedades como o exemplo:

Task Title: Text and XPath | Aprova Casas Bahia
Description: Aprovacao de Credito Casas Bahia
Outcomes: APPROVE,REJECT
Priority: 3 (Normal)
Category: By expression
Owner: User | Static
Application Context:

4. Clique em Owner, e localize o usuário do weblogic default:

Owner: User | Static

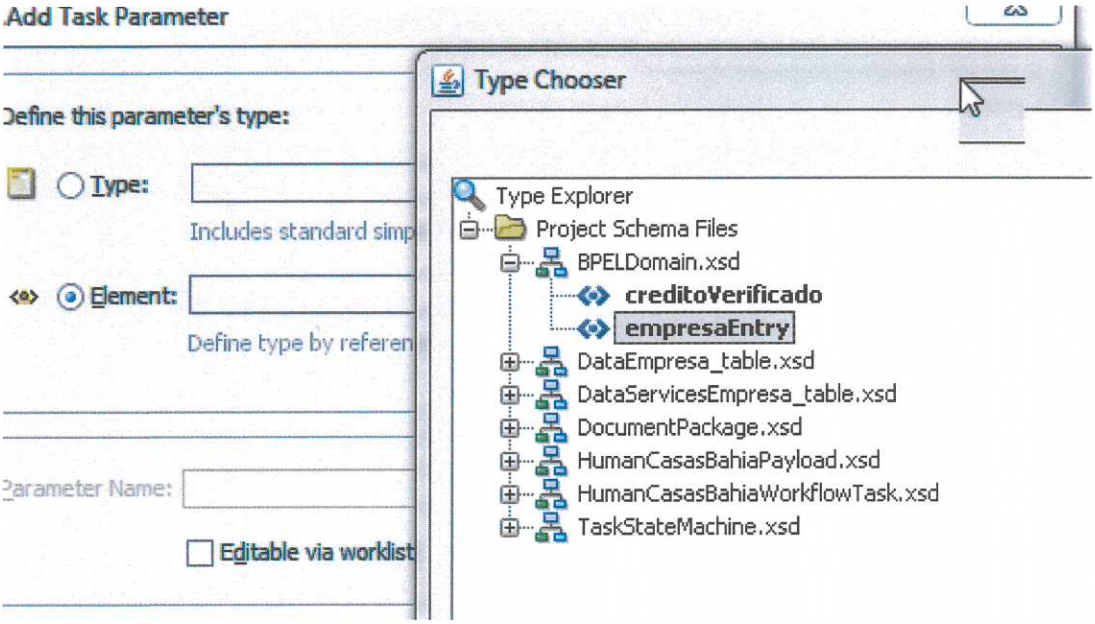
Application Server: felipeLocal (Resource Palette Connection)
Realm: jazn.com
Search Pattern: * | User Name

Search User
oracleesystemuser
weblogic
Select | Hierarchy | Reportees | Detail | << | >>

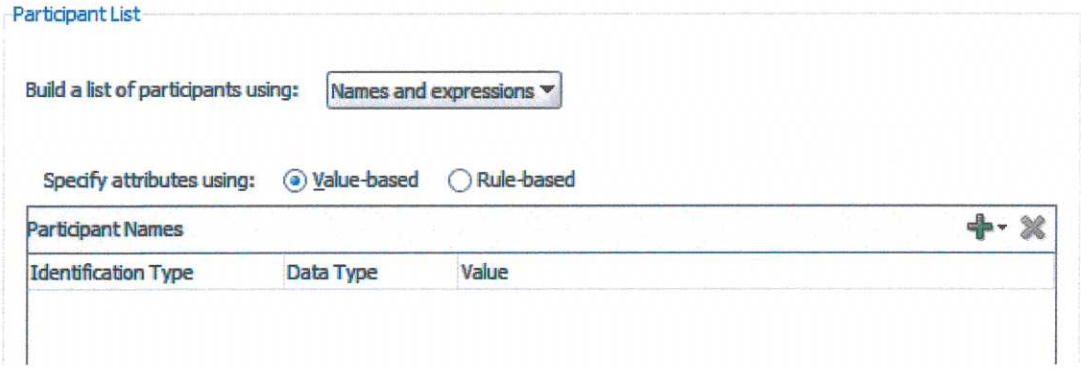
Selected User
weblogic

5.

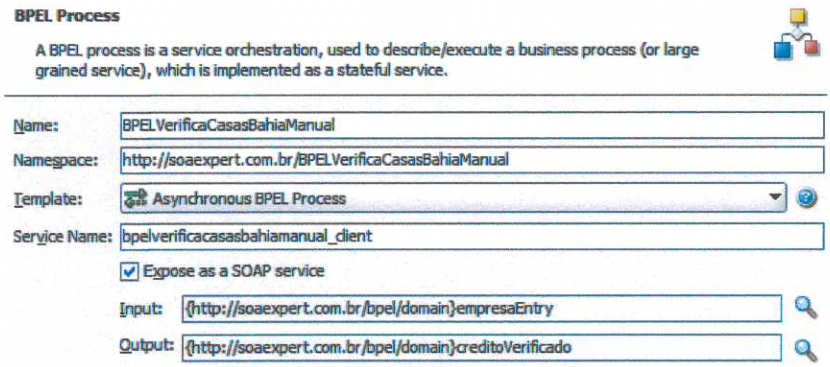
6. Em data, adicione um Elemento de entrada:



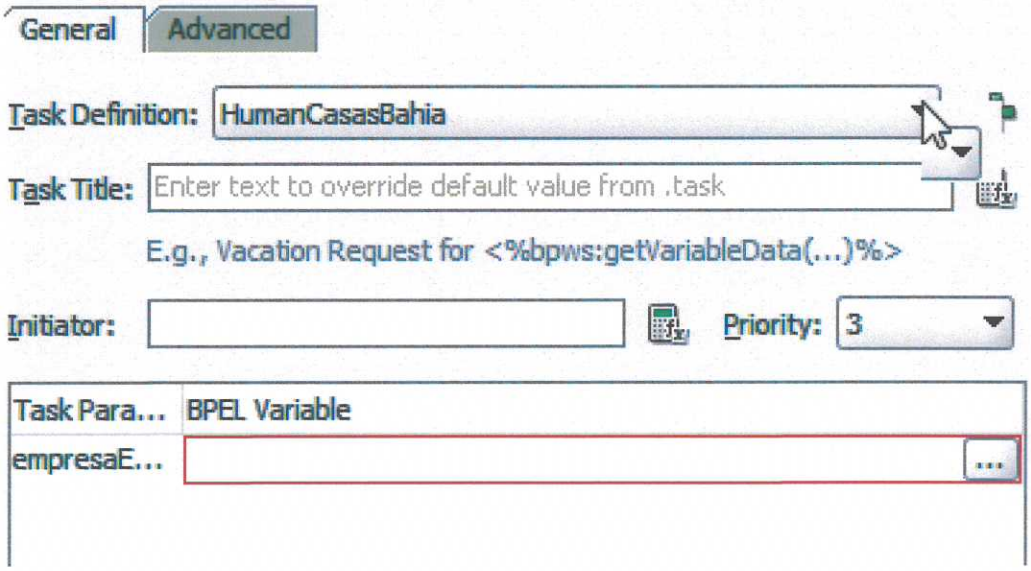
7.Adicione um participante ao WorkFlow e finalize a geração da Task.



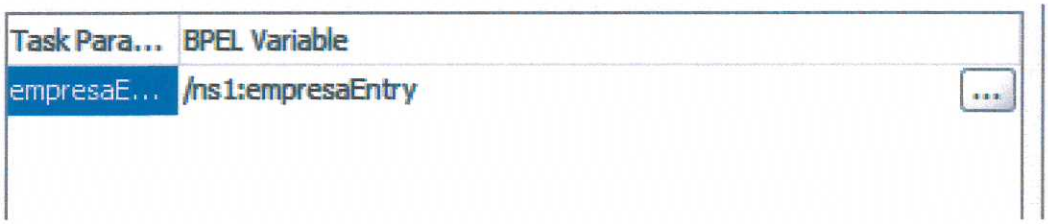
7. No composite, crie um novo processo, chamado **BPELVerificaCasasBahia,** assíncrono com "callback", e crie as variáveis de entrada como os demais processos criados até aqui.



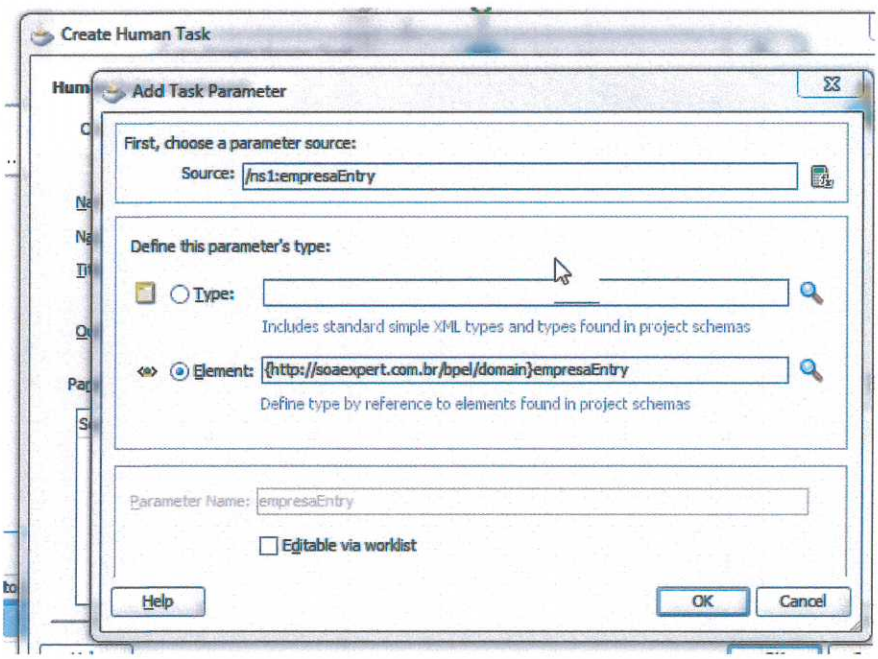
8. No processo BPEL, arraste a tarefa humana para o centro, entre receive e reply, localize a definição de Task Humana Casas Bahia.



9. Crie a variável de entrada para o WorkFlow Humano:



10. Adicione uma tarefa e crie o elemento de entrada



11. Faça o Asign embaixo de cada fluxo de Aprovado ou Rejeitado, com a atribuição da variável respectiva "true" ou "false", pelo expression builder ao nó de retorno do elemento do processo.

12. Faça o Deploy do processo e teste o mesmo (Junto com o instrutor).

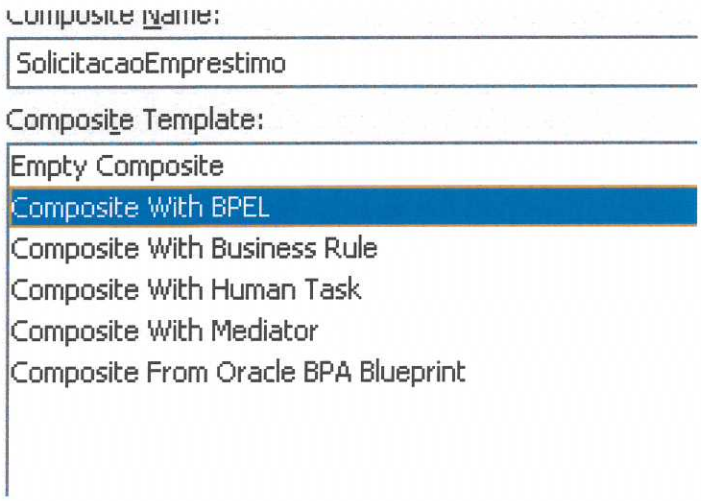
23.Correlation ID

O Oracle BPEL dispõe de uma funcionalidade chamada de **Correlation Sets.** Essa funcionalidade permite localizar um processo em execução, aguardando o recebimento de informações, por exemplo, e continuar seu processamento. Essa localização do processo se dá através da correlação entre um valor já armazenado pelo processo e um valor recebido de uma fonte de dados externa.

Um cenário interessante para se pensar é a solicitação de um empréstimo onde no meio de sua lógica o processo necessita enviar um arquivo para uma verificação de crédito do solicitante e, após isso, receber a resposta pertencente àquela solicitação em específico, dando continuidade no processo a partir deste ponto.

Para o exemplo apresentado, dentro do JDeveloper 11g, crie uma nova aplicação conforme apresentado a seguir:

- Crie uma nova aplicação chamada **BPELCalcula** e um processo com BPEL:

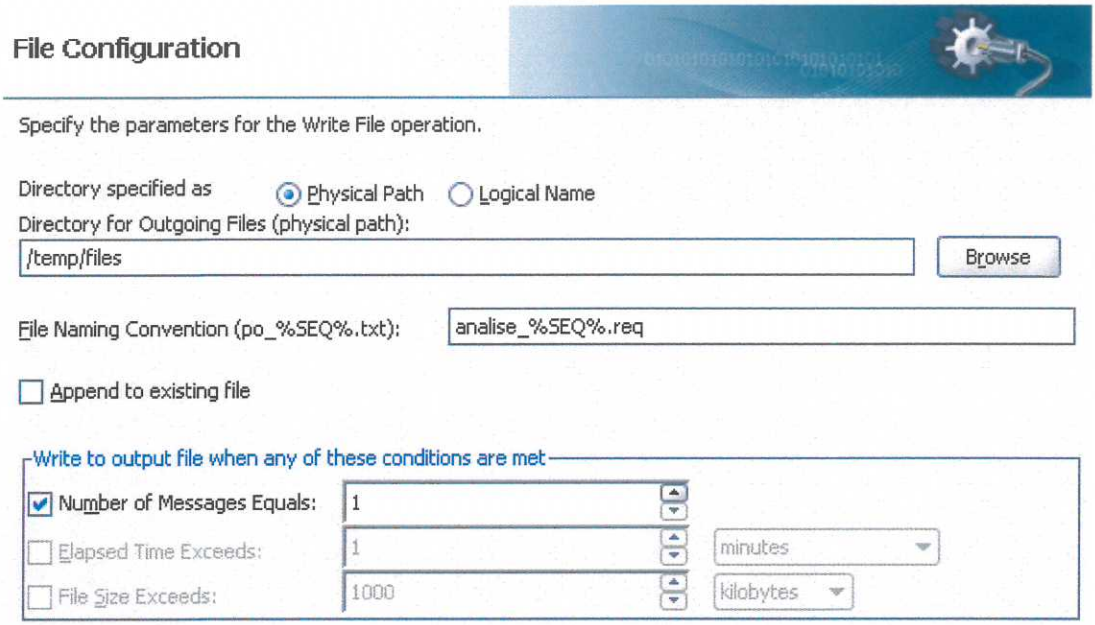


– Dê o nome do processo de **SolicitacaoEmprestimoProcess**

– Carregue o arquivo Emprestimo.xsd e para a variável e defina os tipos das variáveis de input e output.

– Para a manipulação, escrita e leitura de arquivos dentro de um processo BPEL utilizamos uma instância de um adaptador com esse propósito. Para adicionar o adaptador a seu processo, na janela Component Palette selecione BPEL Services, e arraste um File Adapter para a área Partner Links de seu processo BPEL.

Caso seja exibida a tela de boas vindas, clique em Next. Em service name entre com EnvioArquivo. Em adapter interface, mantenha selecionado "Define from operation and schema" e clique em next. A próxima tela exibida é "**Operation**". Selecione "**Write File**" clique em next. Será exibida uma tela similar à apresentada a seguir, preencha as informações conforme apresentado e clique em Next.

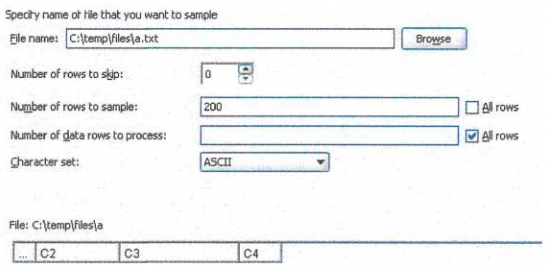


– Na próxima tela, clique em "Define Schema for Native Format" pois gravaremos o arquivo no formato separado por vírgulas (CSV). Na tela de configuração do formato do arquivo, clique em Next na tela de boas vindas. Na próxima tela mantenha "Delimited" selecionado e clique em Next.

Crie um arquivo a.txt na pasta /temp/files com o conteúdo similar ao apresentado a seguir:

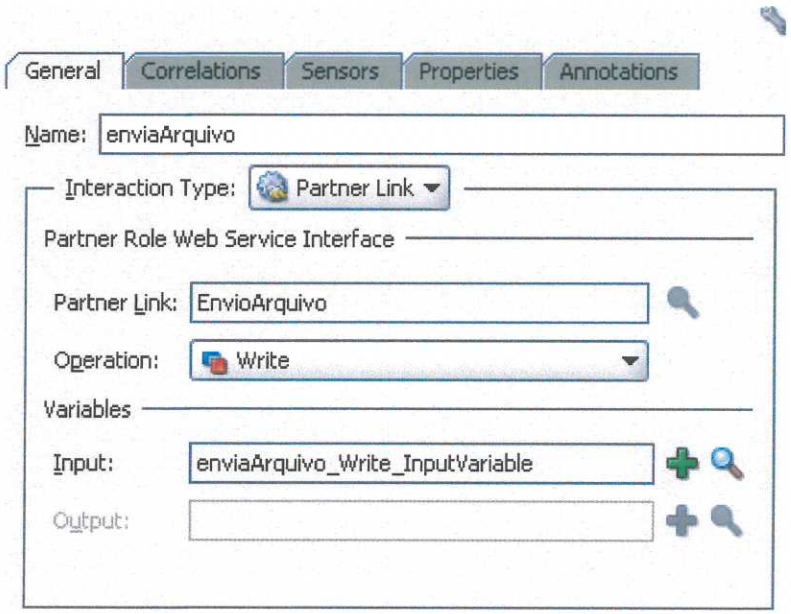
```
1,99999999999,SOAExpert Education,10000
```

6. Nesse arquivo temos os campos na sequência como id, cpf, nome e total requisitado. Na tela File Description utilize esse arquivo como exemplo para criação do schema. A configuração ficará similar à apresentada a seguir:



– Clique em **Next** e **Next** novamente. No campo "Enter a name for element that will represent record" entre com registro. Clique em Next duas vezes e na tela exibida configure os nomes dos campos para que não fiquem como C1, C2, etc.

– Adicione um nó do tipo **Invoke** após o nó **receiveInput**, nomeie o novo nó como **enviaArquivo** e ligue-o à instância do adaptador. Clique no botão + para criação da variável de entrada.



– Precisamos preencher os dados para escrita do arquivo. Adicione um nó Transform antes do nó enviaArquivo. Nomeie-o transformEnviaArquivo.

Edite o nó, selecione como entrada a variável **inputVariable** e como destino a variável enviaArquivo_Write_InputVariable e clique no botão + para criação do arquivo de transformação.

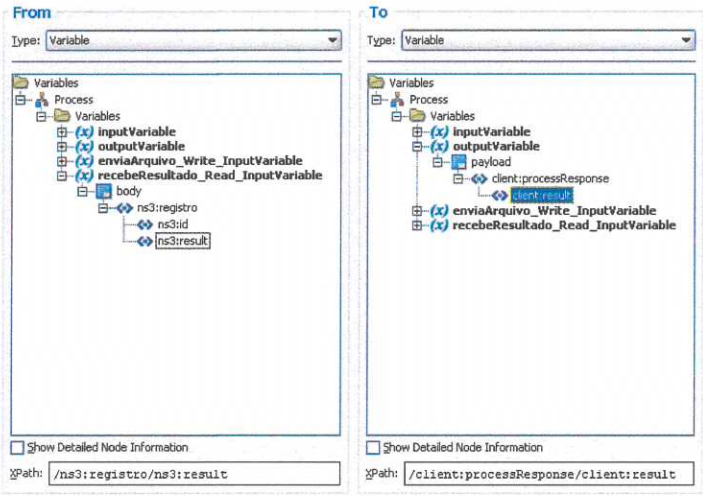
Realize a transformação conforme apresentada a seguir:

– Adicione mais uma instância de file adapter. Nomeie a instância como **RecebimentoResultado**. Selecione a operação como **Read File**. Na seleção para diretórios entre com /temp/files. Na tela File Filtering mantenha File Wildcards e no campo "Include Files with Name Pattern" entre com *.resp. Clique em Next, altere a frequência de polling para 10 segundos. Crie um arquivo b.txt na pasta /temp/files com o seguinte conteúdo:

```
1,APROVADO
```

– Para recebermos as informações lidas pelo arquivo crie um nó do tipo Receive nomeado recebeResultado após o nó enviaArquivo. Ligue-o com o adaptador RecebimentoResultado, criando uma variável para receber as informações lidas.

– Crie uma atividade **assign** após o nó recebeResultado e nomeie-a valorizaRetorno. Crie uma instrução de cópia entre os nós /ns3:registro/ns3:result e /client: processResponse/client:result. A cópia deve ser idêntica à apresentada a seguir:



Entendendo o que aconteceu até aqui:

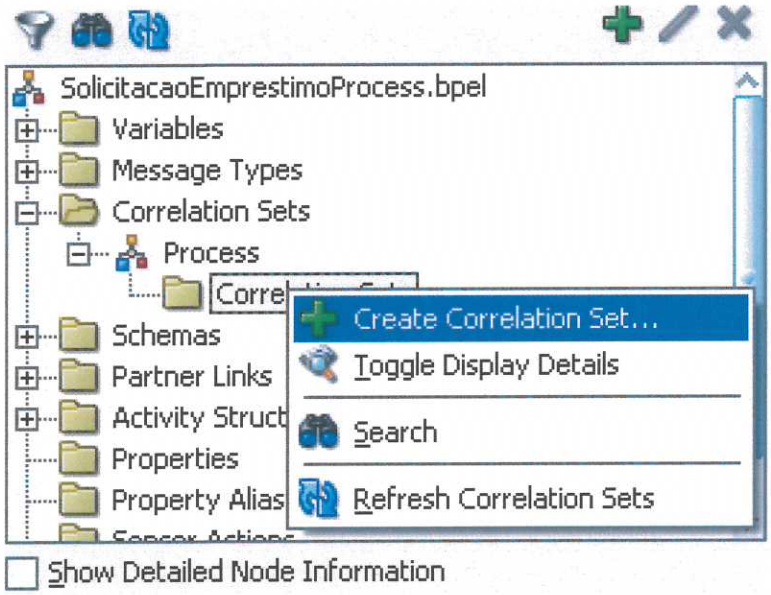
Enviamos a consulta de crédito na forma de um arquivo

Recebemos a resposta também na forma de um arquivo.
Respondemos para o cliente de forma assíncrona se o empréstimo dele foi aprovado ou não. Porém dentro deste fluxo existe uma falha bastante grave. Como sabemos que o arquivo que estamos recebendo é referente àquele determinado processo de solicitação de empréstimo?

No cenário atual simplesmente não sabemos. Com isso, precisamos de um mecanismo para correlacionar o processo que foi iniciado com o arquivo que foi recebido e é aí que os correlation sets (id).

Usaremos um correlation set, para correlacionar a propriedade ID recebida pelo processo, com o ID retornado como resposta à análise de crédito, dando continuidade ao fluxo do processo.

12. Para estabelecer a correlação, precisamos primeiramente criar o correlation set que é uma abstração das propriedades que se deseja correlacionar e depois mapeamos propriedades reais ao correlation set. Para isso, na janela de estrutura do projeto vá até Correlation Sets > Process > Correlation Sets, clique com o botão direito e "Create Correlation Set...".

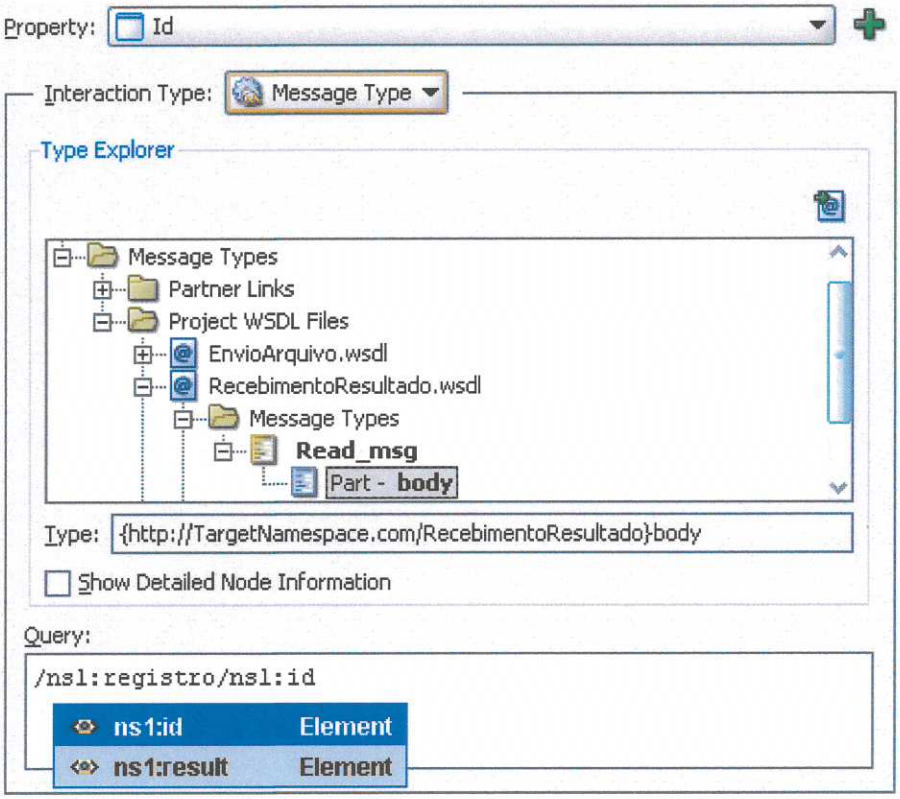


- Ao surgir a tela para criação das propriedades, clique em Add, e crie a propriedade Id do tipo xsd:int

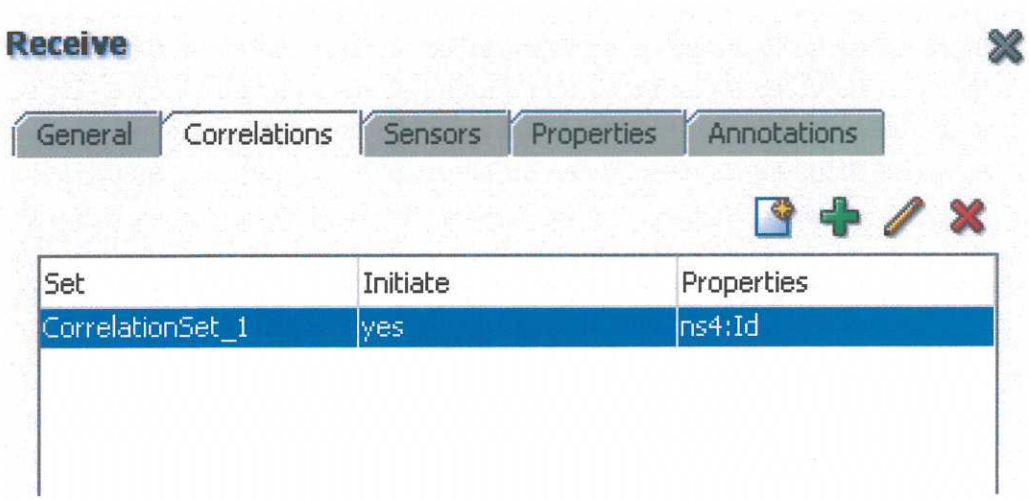
- Agora que criamos o correlation set, precisamos mapear as propriedades. Para isso, criamos o que é chamado de Property Alias. Na janela de estrutura, clique com o botão direito sobre Property Aliases > Create Property Alias.

- Na tela de mapeamento, selecione Type Explorer > Message Types > Partner Links > solicitacaoemprestimoprocess_client > SolicitacaoEmprestimoProcess.wsdl > Message Types > SolicitacaoEmprestimoProcessRequestMessage > Part – payload. Em Query pressione ctrl+barra espaço e entre com /client: process/client:id.

- Repita a operação e selecione Type Explorer > Message Types > Project WSDL Files > RecebimentoResultado.wsdl > Message Types > Read_msg > Part – body e em Query selecione /ns1:registro/ns1:id.



- Por fim, precisamos apenas preencher as propriedades. Dê um duplo clique em receiveInput. Acesse Correlations, local onde podemos especificar o correlation set a ser utilizado. Clique em Add e selecione o correlation set que acabamos de criar. Após isso altere a propriedade Initiate para yes para indicar que devemos inicializar o correlation set nesse passo, conforme vemos a seguir:



- Repita a operação para o nó "recebeResultado", deixando a propriedade Initiate valorizada com "no" pois já inicializamos esse correlation set.

- Faça o Deploy e teste com o instrutor.

24. Service Component Architecture

A SCA (*Service Component Architecture*) é uma proposta de padronização para a composição e a implantação de aplicações orientadas a serviços. Atualmente ela está na versão 1.0 e é suportada por um grupo de empresas, denominado OSOA (*Open Service Oriented Architecture*). O objetivo desse grupo é a manutenção e o desenvolvimento de tecnologias que convirjam para os interesses mútuos no desenvolvimento de aplicações baseadas em SOA. Porém, OSOA não é um orgão de padronização.

A versão 1.0 da SCA foi enviada para padronização no consórcio internacional OASIS (*Organization for Advancement of Structured Information Standards*) que dirige o desenvolvimento, convergência e adoção de padrões para *SOA*.

Elementos SCA

Assembly Model – Consiste num conjunto de modelos e especificações para a composição de serviços.

Especificações de Implementação de Componentes – Define como implementar os serviços em uma gama de linguagens como Java, C++, BPEL, PHP, entre outras

Especificação de *Binding* - Define como acessar os componentes através de tecnologias como Web Services, JMS (Java *Message Service*), RMI-IIOP, REST, entre outras.

Policy Framework – Consiste na adição de detalhes de infra-estrutura na aplicação, como segurança, controle de transação, qualidade de serviço *(Quality of Service)*, entre outros.

25.Assembly Model

Os *Services* consistem nas operações que o componente fornece para o ambiente externo. Eles podem ser especificados de diversas maneiras, dependendo da linguagem que a implementação foi desenvolvida. Por exemplo, um componente implementado em Java, provavelmente utilizará uma Interface Java para descrever os seus *services*, enquanto um componente implementado em BPEL, utilizará o WSDL (*Web Services Description Language*).

A especificação de *service* abaixo utiliza uma Interface Java com *annotations.* Neste exemplo o componente provê o método getInfoEmpresa para ser invocado. A anotação @Remotable indica que o componente pode ser invocado por clientes remoto.

```
@Remotable public interface BovespaInfo {
public String getInfoEmpresa();
}
```

Reference:

A *reference* especifica as operações que o componente necessita invocar para realizar determinada tarefa. Uma *reference* pode ser usada para acessar um serviço como por exemplo: um Web Service ou um EJB (Enterprise Java Bean). A *reference* é composta basicamente por uma interface que possui todos os métodos que o componente irá invocar.

No exemplo abaixo **SetorService** que está destacado em negrito, é uma *reference* do componente EmpresaServiceComponent. A implementação dessa reference é em Java.

```
<component name="EmpresaServiceComponent">
<service name="EmpresaService">
<interface.wsdl
interface="http://bovespainfo#wsdl.interface(Boves
paInfo)" />
<binding.ws
uri="http://localhost:7001/BovespaInfoService"/>
</service>
<implementation.java
class="helloworld.EmpresaServiceImpl" />
<reference name="setorService"
target="SetorService"/>
</component>
<component name="SetorService">
<implementation.java
class="helloworld.SetorServiceImpl"/>
</component>
```

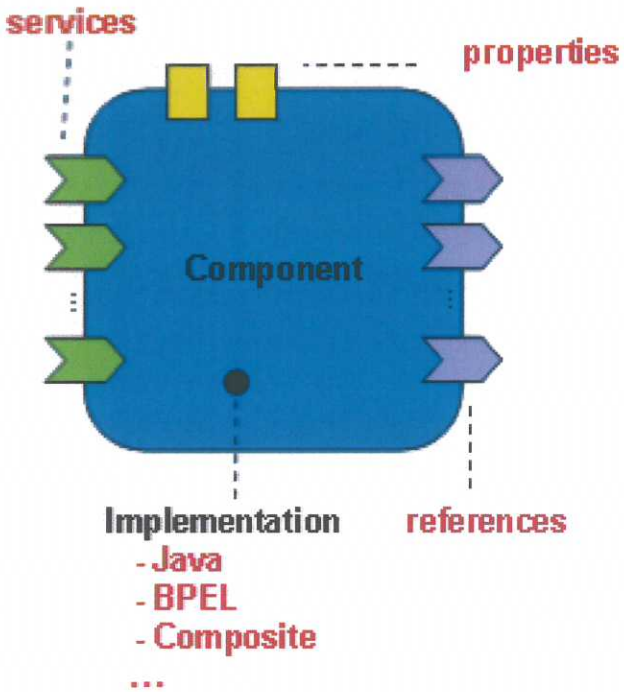
Property

As *property* são atributos dos serviços que correspondem a valores de variáveis ou entidades que são utilizadas pelas implementações dos componentes em sua lógica de negócio. Elas correspondem a dados externos da implementação, dessa forma, são desacopladas de uma implementação específica, permitindo que se troque uma implementação de forma simples, sem alteração na definição do serviço no SCA *Domain*. Um exemplo de *property* pode ser o tipo de moeda que uma implementação deva usar, no caso um componente usa o tipo real, e outro pode usar o tipodólar. Dessa forma, a implementação não está atrelada a um determinado tipo de moeda, o que provê o reuso de outras implementações de maneira fácil.

Component

O *component* é o elemento mais básico de uma composição de serviços, As suas 2 principais funções são consumir e prover serviços. Ele basicamente corresponde à configuração de uma implementação em uma determinada linguagem de programação ou *framework,* tais como Java, EJB, C++ e BPEL.

O simbolo abaixo representa todos os elementos descritos até agora no diagrama da *Assembly Model.*



O código abaixo mostra um exemplo da configuração de um *component para* prover informações de ações negociadas na BOVESPA.

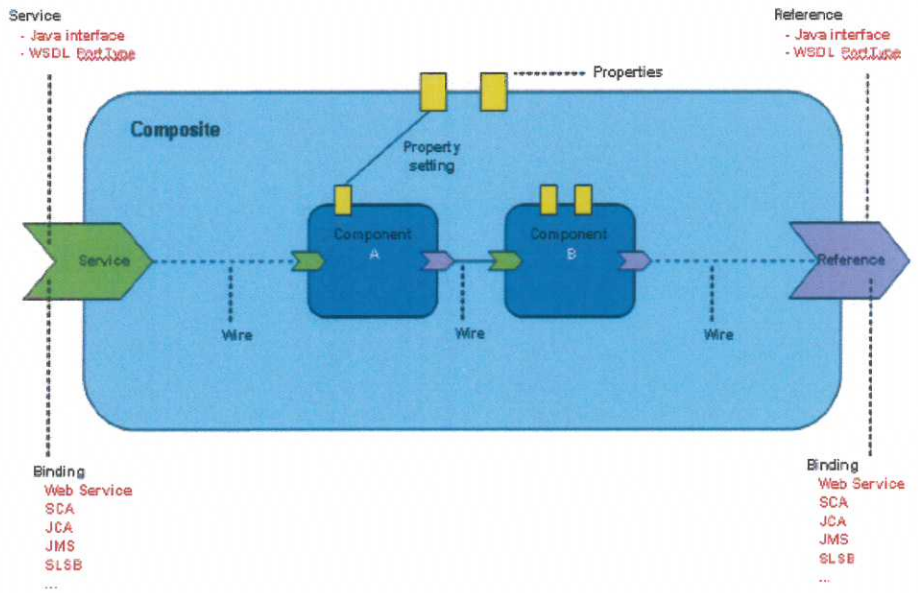
A tag <implementation.java> define que a sua implementação é em linguagem Java.

A tag <service> denota que ele é consumido através da *service* AcoesBovespaService, que possui a interface Java chamada services.bovespa.interface.AcoesBovespaService, espeficicada pela tag <interface.java>. Esse componente é acessado através da tecnologia JMS, definido pela tag <binding.jms>. Ele invoca o serviço intraCorretoraService que é um Serviço Web, definido pela tag <binding.ws>. Além disso, ele possui uma *property currency* que especifica o tipo de moeda que a implementação do componente deve usar.

```
<component name="AcoesBovespaServiceComponent">
<implementation.java
class="services.bovespa.impl.AcoesBovespaServiceImpl">
<service name="AcoesBovespaService"> <interface.java
interface="services.bovespa.interface.AcoesBovespaService" />
<binding.jms />
</service> <reference name="intraCorretoraService"
target="intraCorretoraService">
<binding.ws> </reference>
<property name="currency">real</property> </component>
```

Composição de serviços

A composição de serviços consiste na ligação dos componentes para servir a um determinado fim. Esses componentes podem ser desenvolvidos para atender especificamente à aplicação ou podem ser sistemas existentes implementados em linguagens e plataformas distintas do componente consumidor. A composição desses componentes irá prover um serviço para determinada operação ou lógica de negócio do sistema. Com isso, a composição pode ser vista como um simples serviço. Os serviços compostos trabalhando mutuamente para atender a um determinando negócio constituem o sistema. A figura abaixo representa um diagrama da SCA para uma composição de serviços:



]

26. Fases do projeto de aplicações SCA

O desenvolvimento de aplicações orientadas a serviços utilizando SCA é feito em 3 fases: Desenvolvimento, Montagem e Implantação.

Desenvolvimento

Após o desenvolvimento dos novos componentes ou a identificação dos componentes existentes da aplicação que implementarão as regras de negócio do sistema, é feita a configuração do comportamento de execução. Basicamente, isso é feito configurando arquivos XML, seguindo os modelos da SCA.

Montagem

Nesta etapa, o analista de negócio ou um desenvolvedor compõe um ou mais componentes para atender a uma determinada função de negócio. Uma composição de serviços pode ser executada em um processo de uma máquina ou os componentes da composição podem ser distribuídos em máquinas diferentes, independentemente de plataformas e sistemas operacionais dessas máquinas.

Implantação

Na implantação são feitas as seguintes tarefas:

- Montagem da composição de serviços em unidades funcionais de negócio, ou seja, que atendam a uma determinada função/processo de negócio; definição *das property* que foram estabelecidas durante a montagem;
- Conexão dos pontos de acesso das composições de serviços as *references* de outras composições;atribuição dos valores de qualidade de serviço, tais como, segurança, autenticação e transação, entre outros.

Benefícios da SCA para o desenvolvimento em SOA

Os principais benefícios da SCA são:

1 – Fraco acoplamento: Componentes podem ser integrados com outros componentes sem necessitar saber como eles foram implementados. Fraco acoplamento é um requisito para uma aplicação em SOA.

2 – Flexibilidade: Componentes podem ser facilmente repostos por outros componentes. É um requisito para uma sistema em SOA.

3 – Produtividade: Fácil composição de componentes simples e compostos para formar aplicações orientadas a serviços.

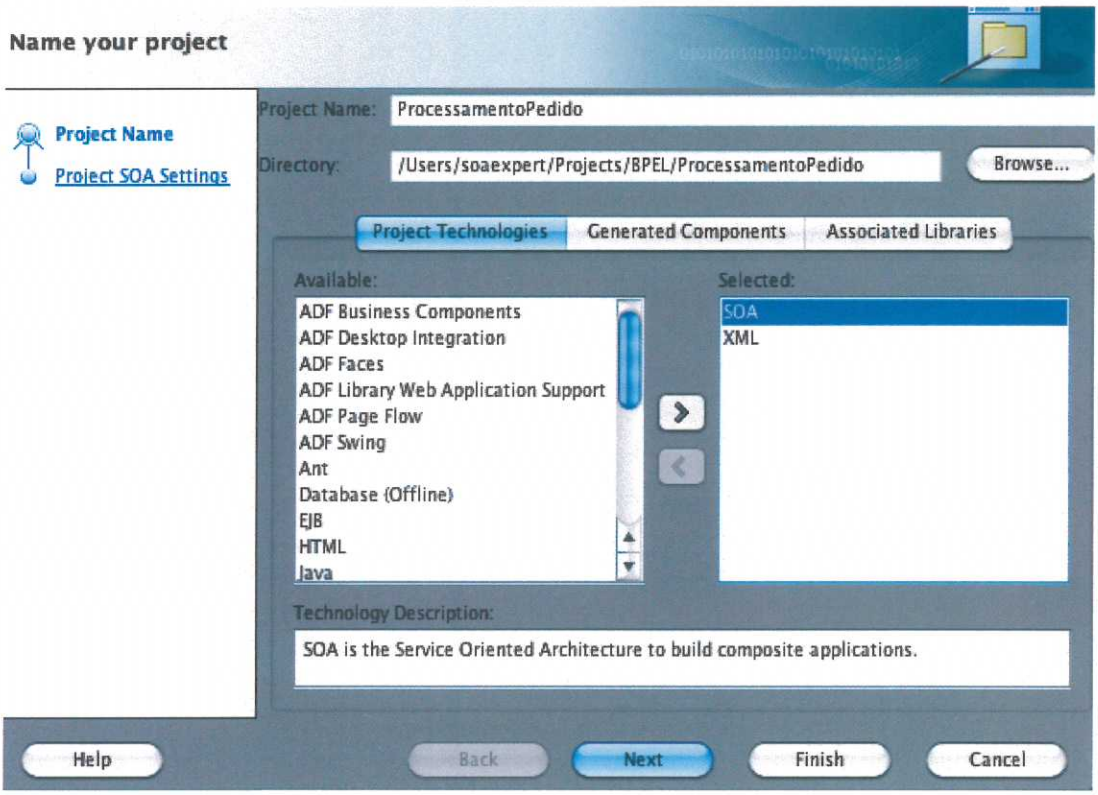
4 – Serviços: Podem ser invocados facilmente de forma síncrona ou assíncrona. **5 – Composição de Serviços:** São descritas de forma clara e precisa através de arquivos
XML

Os benefícios listados acima, se devem fundamentalmente à configuração dos componentes e composições de serviços em arquivos XML. Isso possibilita a abstração dos detalhes de desenvolvimento dos componentes e serviços da invocação e utilização deles. Além disso, o uso de arquivos XML proporciona portabilidade à SCA, uma vez que esse tipo de arquivo é interpretado em qualquer linguagem de programação e sistema operacional.

27.Exercícios de SCA

O objetivo do exercício é explorar o Meditor junto ao Composite SCA, que faz papel de um ESB interno do produto, fazendo transformações e ligando (wiring) as referências para composição de uma solução.

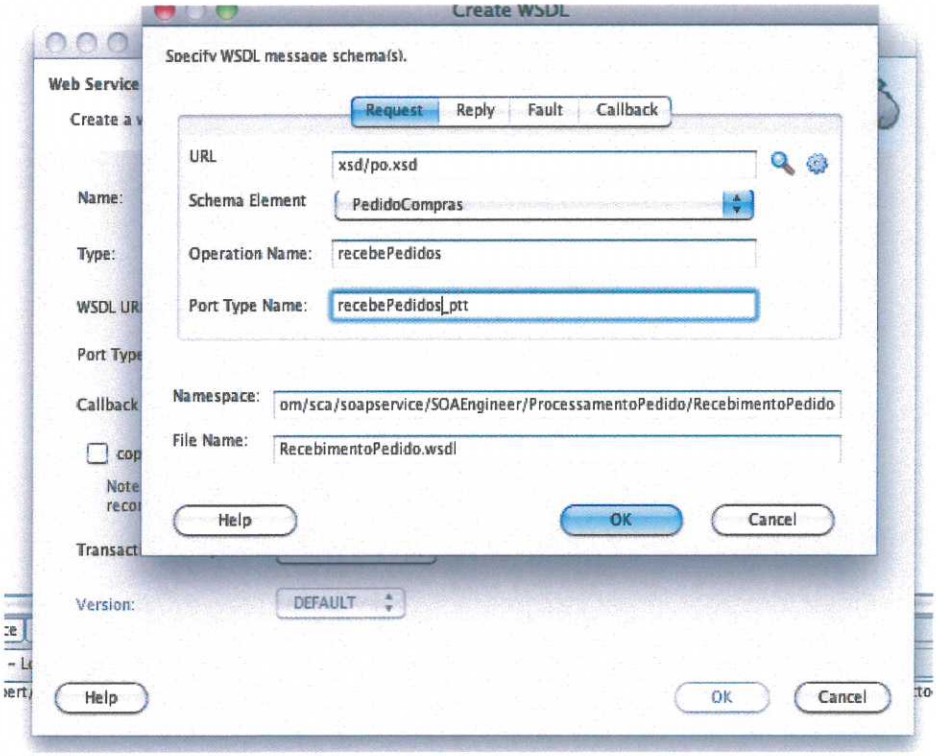
1.Crie um novo projeto chamado ProcessamentoPedido.



2.Crie um composite Empty na sugestão de templates.

3. Clique e arraste o componente WebService, para a raia esquerda de Serviços Expostos:

4. Escolha para gerar um contrato, através da engrenagem e localize o po.xsd, para carregar a estrutura de pedidos como variável de entrada.

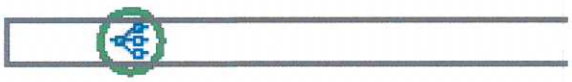


5. Defina na raia direita um Adapter do tipo File, como nos exercícios anteriores e escolha o tipo de escrita pra saída, assim como o elemento de saída.

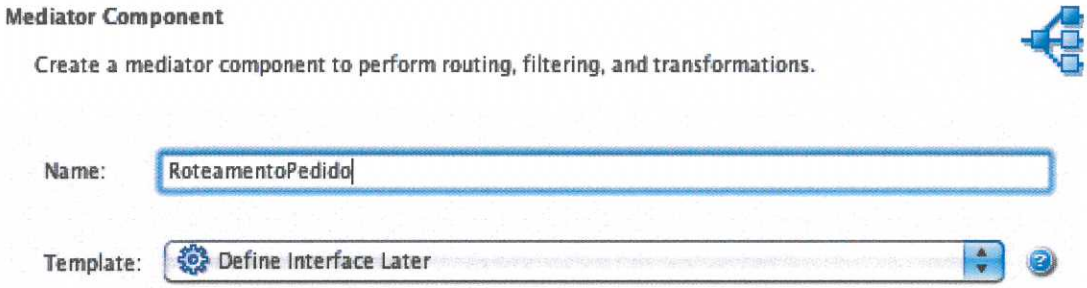
6. Especifique o diretório de gravação de arquivo, e a convenção para gravação do mesmo: ***pedido_%SEQ%.txt.***

7. Localize o Schema **pedidointerno.xsd** e escolha o elemento Pedido.

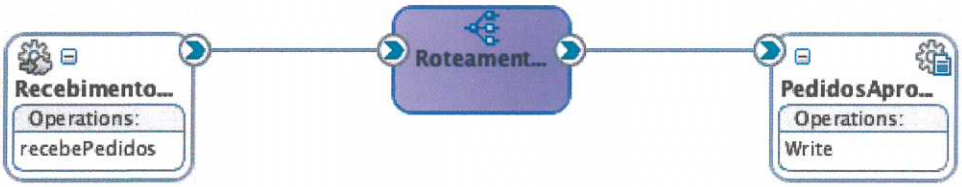
8. Clique e arraste um componente **Mediator**, para o centro do composite.



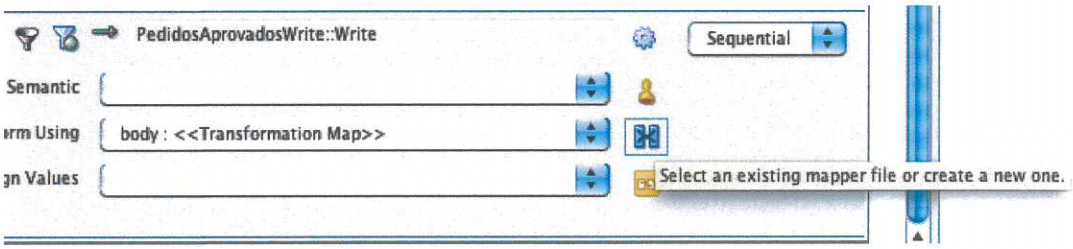
1. Nomeie como **RoteamentoPedido**



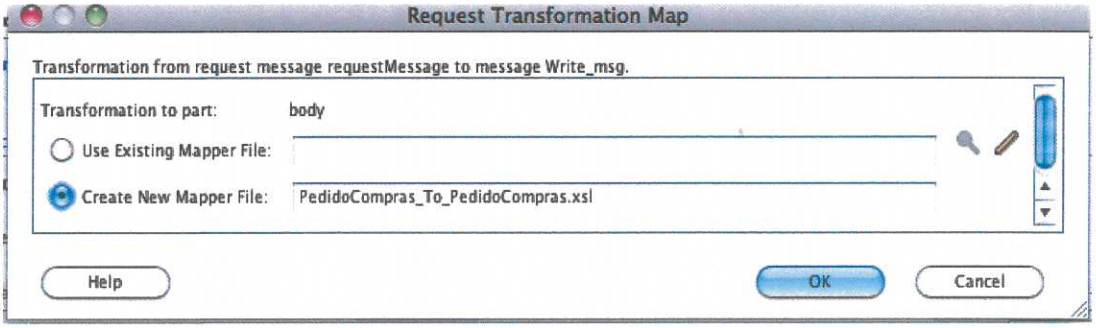
2. Ligue o WebService de recebimento ao Componente e o Roteamento ao Fie Adapter.



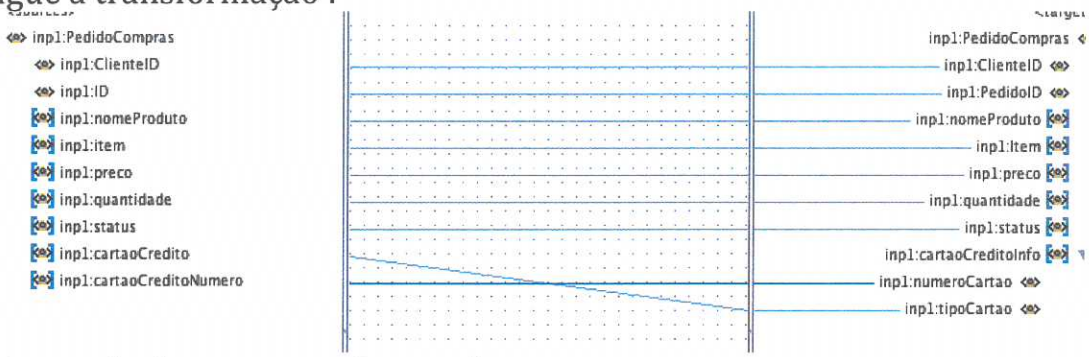
3. Edite a transformação de tipos entre os serviços:



4. Crie uma nova transformação como abaixo:



5. Ligue a transformação :



6. Faça o deploy e teste o Composite.

28. BAM – Business Analysis Monitoring

O BAM é um componente de software que nos permite analisar em tempo real o nosso processo de negócio.

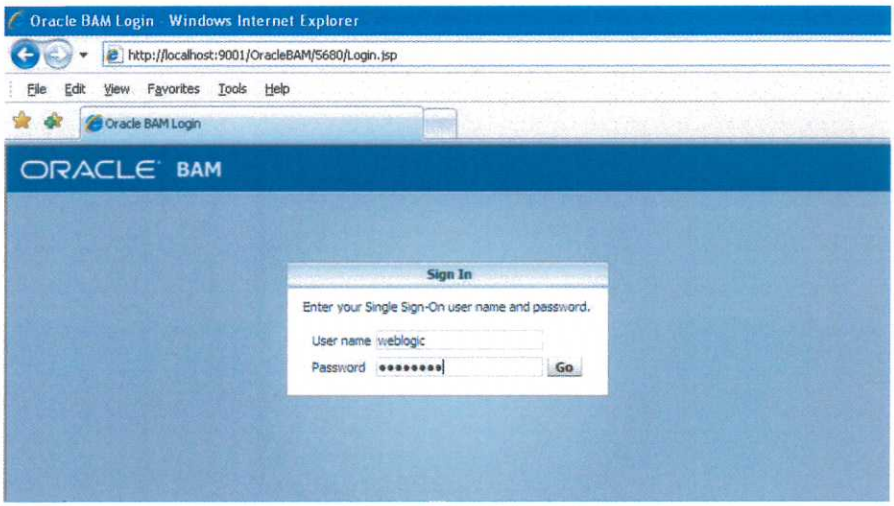
O exercício a seguir, tem como objetivo mostrar um relatório simples no BAM e suas ferramentas para composição (Studio), (Arquitetura) e configuração junto à IDE.

Criando DashBoards no BAM

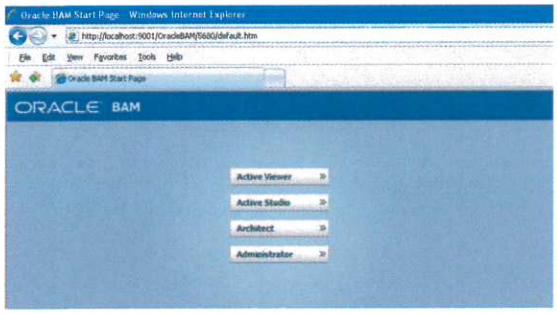
Nesta etapa, será criada um data object no BAM para armazenar os dados relacionados aos itens de compras.

1. Inicie o seu servidor entre com seu login e usuário:

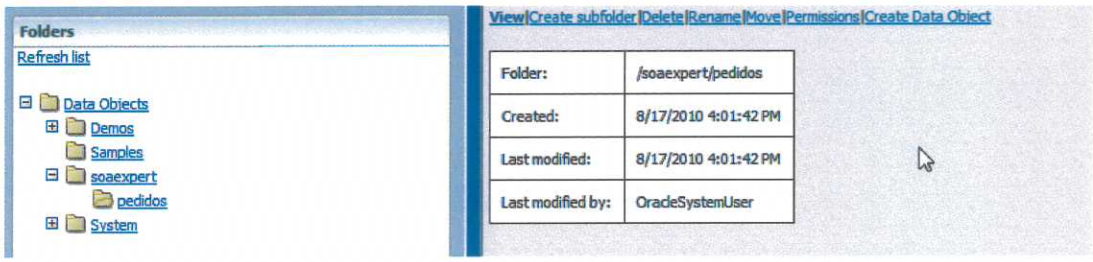
http://localhost:9001/OracleBAM



2. Abra a aplicação "Architect"



3. Clique nos links Data Objects na estrutura de pastas à esquerda e crie uma subpasta chamada soaexpert ee uma outra sub-pasta chamada pedidos.



4. Adicione um novo objeto **data object,** clicando em "Create Data Object" e nomeie como **Pedidos**

5. Adicione campos e atribua os tipos correspondents de acordo com os tipos e tamanhos.

Nome	Tipo	Max Size or Scale
pedidoID	String	30
clienteID	String	30
nomeProduto	String	50
item	String	30
preco	Decimal	10
quantidade	Decimal	10
status	String	50

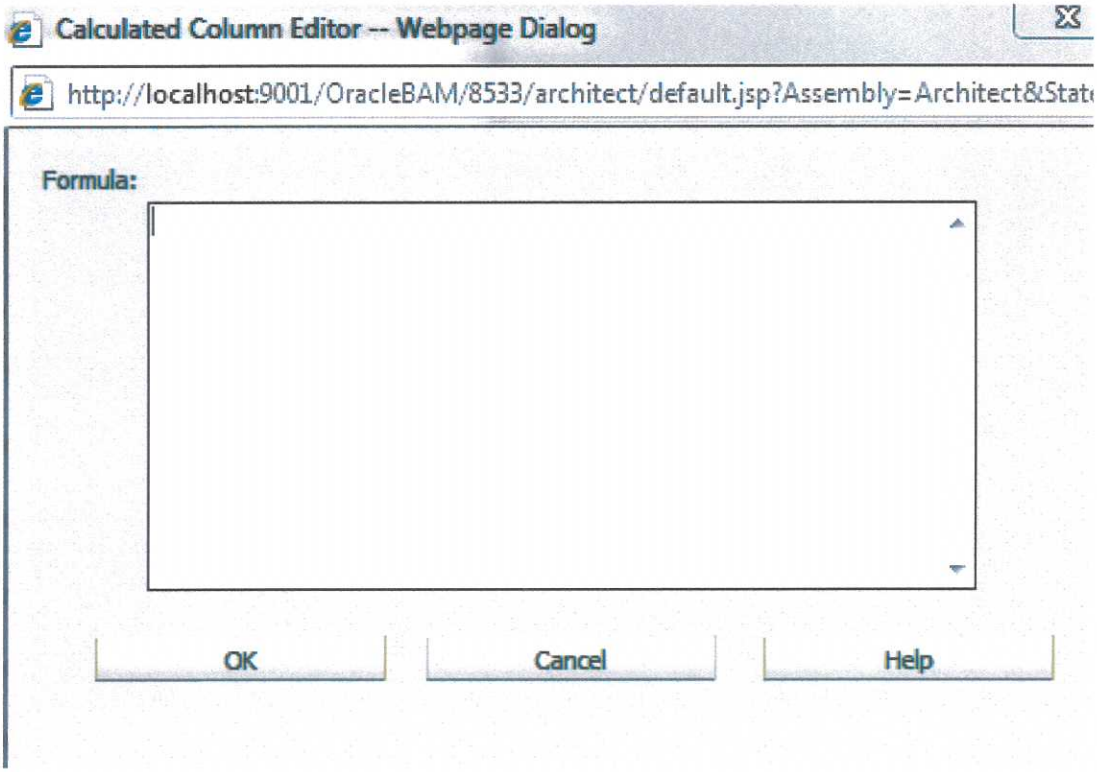
total	Calculated	-
dataRecebimento	Timestamp	-

pedidoID	String	Max size: 30	Nullable	Public	Tip Text:	Remove
clienteID	String	Max size: 30	Nullable	Public	Tip Text:	Remove
nomeProduto	String	Max size: 50	Nullable	Public	Tip Text:	Remove
item	String	Max size: 30	Nullable	Public	Tip Text:	Remove
preco	Decimal	Scale: 10	Nullable	Public	Tip Text:	Remove
quantidade	Decimal	Scale: 10	Nullable	Public	Tip Text:	Remove
status	String	Max size: 50	Nullable	Public	Tip Text:	Remove
Total	Calculated	Edit formula	-	Public	Tip Text:	Remove
recebimento	Timestamp		-	Public	Tip Text:	Remove

Add a field | Add one or more lookup fields

Create Data Object | Cancel

6. Em "calculated" clique em cima e adicoine a seguinte fórmula: preco * quantidade.



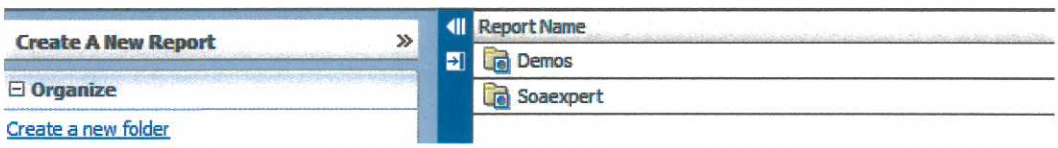
7. Nesta etapa você irá definir o DashBoard, um simples relatório que irá apresentar os dados em tempo real do seu processo.

Volte para a página incicial, e clique em Active Studio.

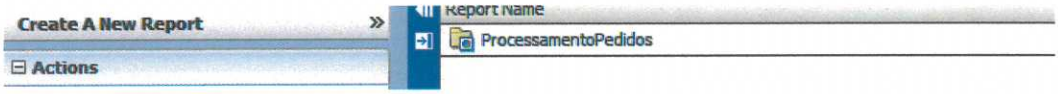
"Active Studio".

- Clique na aba "Shared Reports" para criar um relatório compartilhado.

1. Crie uma nova pasta SoaExpert, a fim de manter seu projeto organizado.

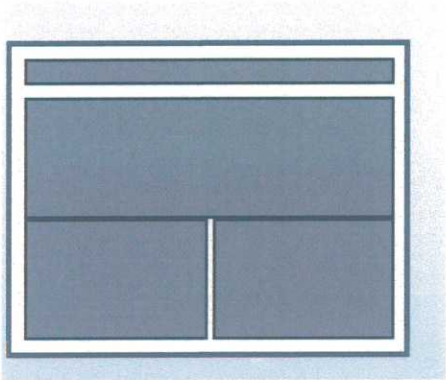


10. Crie uma subpasta **ProcessamentoPedidos**



11. Dê dois cliques em cima da pasta e clique no botão: "Create a New Report.

12. Selecione o seguinte Layout para visualização do relatório:



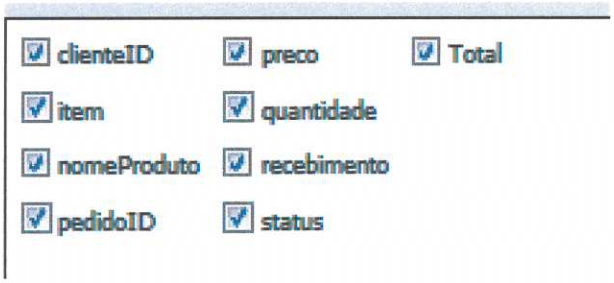
13. Dê o título para o relatório de: **Relatório de processamento de pedidos.**

Relatório de processamento de pedidos

14. Para a área de cima, escolha o elemento gráfico Updating List:

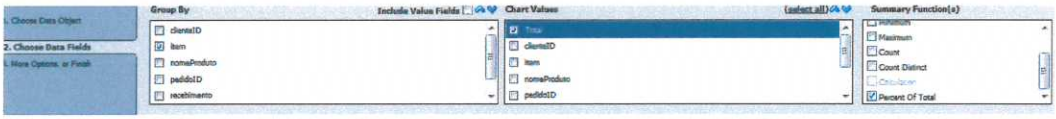
15. Localize o Pedido Object, criado no passo anterior do exercício.

16. Marque todos os atributos:



17. No lado esquerdo, escolha um elemento **3D Pie Chart.**

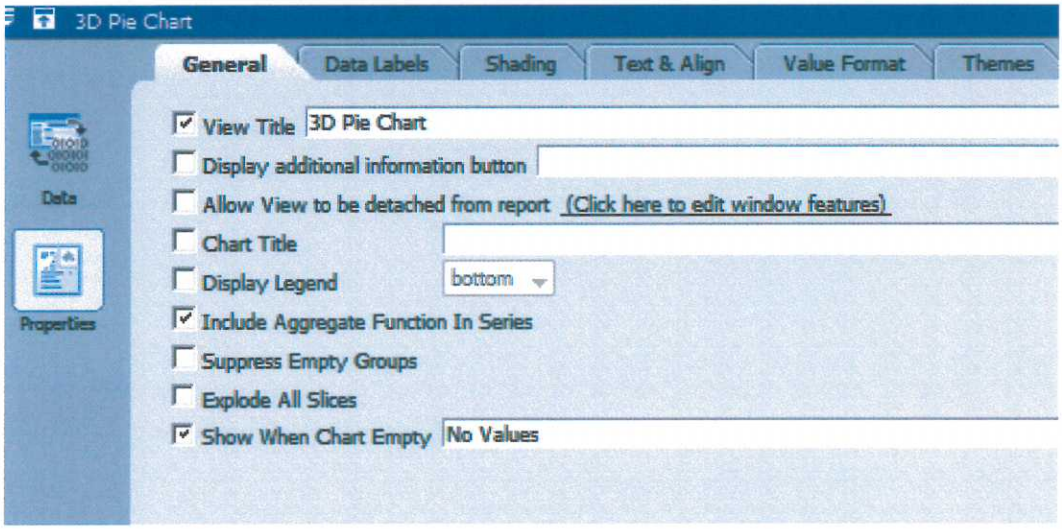
18. Escolha os seguintes campos:



Item – Total e Percenf of Total.

PS: Desmarque a opção Sum.

- Em View Title , modifique o título para: Total de produtos vendidos por item.



- No lado direito, selecione o elemento gráfico : **3D Bar Chart.**
- Assinale do data object: Status, pedidoID e em summary a opção count.

- Salve o Relatório: