



# Business Process Digitalization and Cloud Computing

## 14. Advanced BPEL

---

Andrea Morichetta, Phd

January 12, 2017

Computer Science Division

slides are based on the WS-BPEL 2.0 for SOA Composite Applications with Oracle SOA Suite

# Table of contents

1. Loops
2. Delays
3. Faults
4. Sopes
5. Compensation
6. Events
7. Event Handler
8. Business process lifecycle
9. Correlation and message properties
10. Dynamic Partner Links
11. Building BPEL application Using JDeveloper
12. Deploy

In the previous part we saw:

- Operation `<invoke>`, `<receive>`, `<reply>`
- Declaring variables `<variable>`
- Updating variable contents `<assign>`
- Structured activities `<sequence>`, `<flow>`
- Conditional behaviour `<if>`

# Loops

---

BPEL supports the following three types of loop:

- `<while>`
- `<repeatUntil>`
- `<forEach>` provide the possibilities to start instances in parallel

Loops are useful for using arrays.

**Arrays** can be simulated using XML schema complex type.

- **Repeat the enclosed activity** until the boolean condition no longer holds true.
- The **Boolean condition** is expressed through the condition element, using the selected expression language

```
<while>  
  <condition> boolean-expression </condition>  
  <!-- Perform an activity or a set of activities enclosed by  
    <sequence>,  
    <flow>, or other structured activity -->  
</while>
```

## While example

Lets consider the possibility to **check the flight availability** for more than one person.

```
<while>
  <condition>${Counter} <lt; ${NoOfPassengers}</condition>
  <sequence>
    <!-- Construct the FlightDetails variable with passenger data-->
    ...
    <!-- Invoke the web service -->
    <invoke partnerLink="AmericanAirlines"
      portType="aln:FlightAvailabilityPT"
      operation="FlightAvailability"
      inputVariable="FlightDetails" />
    <receive partnerLink="AmericanAirlines"
      portType="trv:FlightCallbackPT"
      operation="FlightTicketCallback"
      variable="FlightResponseAA" />
    ...
    <!-- Process the results ... -->
    ...
    <!-- Increment the counter -->
    <assign>
      <copy>
        <from>${Counter} + 1</from>
        <to variable="Counter"/>
      </copy>
    </assign>
  </sequence>
</while>
```

# Repeat Until

The `<repeatUntil>` loop repeats the enclosed activities **until the Boolean condition remains true**.

```
< repeatUntil >
  <!-- Perform an activity or a set of activities enclosed by <sequence>, <flow>,
        or other structured activity -->
  <condition> boolean-expression </condition>
</ repeatUntil >
```



## For each loop

- The <forEach> loop is a **for type loop**
- Can execute the loop **branches in parallel or serial** (The first one is very similar to the **for** the latter is similar to **flow**)
- Requires to specify the variable for counter  
(startCounterValue, finalCounterValue)
- Requires that all the activities executed within the branch should be **grouped into a <scope>**
- The loop will complete when **all branches have completed.**

```
<forEach counterName="BPELVariableName" parallel="yes|no">  
  <startCounterValue>unsigned-integer-expression</startCounterValue>  
  <finalCounterValue>unsigned-integer-expression</finalCounterValue>  
  <scope>  
    <!-- The activities that are performed within forEach have to be nested within a scope. -->  
  </scope>  
</forEach>
```

## For each finish condition

- Could be useful if the `<foreach>` loop would **not have to wait for all branches to complete**
- We can specify that the loop will **complete after at least N branches** have completed
- Using the `<completionCondition>` we can specify the number N of `<branches>`
- Using the `successfulBranchesOnly` we can specify if we would like to count **only successful** or **all branches**.

```
<foreach counterName="BPELVariableName" parallel="yes|no">
  <startCounterValue>unsigned-integer-expression</startCounterValue>
  <finalCounterValue>unsigned-integer-expression</finalCounterValue>
  <completionCondition> <!-- Optional -->
    <branches successfulBranchesOnly="yes|no(default)">
      unsigned-integer-expression
    </branches>
  </completionCondition>
  <scope>
    <!-- The activities that are performed within foreach have to be nested within a scope. -->
  </scope>
</foreach>
```

# Delays

---

Delays:

- for a specified **period of time**
- until a certain **deadline** is reached

Typically we could specify delays to invoke an operation at a **specific time** or **wait some time** and invoke the operation

We need to wait:

- **before to pool the results** of a previously initiated operation
- **between the iteration** of the pool

# The Wait Activity into Delays

- **for** : We can specify **duration**; or a period of time.

```
<wait>  
  <for> duration-expression </for>  
</wait>
```

- **until** : We can specify a **deadline**; we specify a certain date and time.

```
<wait>  
  <until> deadline-expression </until>  
</wait>
```

# Deadline Expression

The deadline data types are either **dateTime** or **date**.

The **lexical representation** should be conformed to the XPath

- C represents centuries
- Y represents years
- M represents months
- D represents days
- h represents hours
- m represents minutes
- s represents seconds
- Z is used to designate Coordinated Universal Time (UTC)
- T is used as time designator to indicate the start of the representation of the time.

```
<wait>  
  <until>'2004-03-18T21:00:00+01:00'</until>  
</wait>
```

```
<wait>  
  <until>'18:05:30Z'</until>  
</wait>
```

# Duration Expressions

- P is used as the time duration designator. Duration expressions always start with P .
- Y follows the number of years.
- M follows the number of months or minutes.
- D follows the number of days.
- H follows the number of hours.
- S follows the number of seconds.

4 hours and 10 minutes

```
<wait>  
  <for>'PT4H10M'</for>  
</wait>
```

1 month, 3 days, 4 hours, and 10 minutes

```
<wait>  
  <for>'P1M3DT4H10M'</for>  
</wait>
```

1 year, 11 months, 14 days, 4 hours, 10 minutes, and 30 seconds

```
<wait>  
  <for>'P1Y11M14DT4H10M30S'</for>  
</wait>
```

## Empty Activities

In BPEL process sometimes you need to specify activities for satisfying the XML Schema, but you **do not really want to perform any activities**.

For example `<if>` activity, require **to specify an activity for each branch**.

Not specifying an activity would result in an **error** so the solution is to specify an empty activity:

```
<empty/>
```



The `<exit/>` activity **terminate a business process immediately** before the normal ending.

With the exit **no fault and compensation handling is performed.**

# Faults

---

BPEL **interact** with their partners through **operation invocations** of web services, so the typical faults are:

- The communication is over a **not highly reliable internet connection**.
- **Logical errors**
- **Execution errors** raised from defect on the infrastructure

So **fault handling and signaling** is an important aspect for business process **reliability**

Possible faults:

- In a synchronous invocation, the operation might return a **WSDL fault message**.
- A process **explicitly signal** (throw) a fault
- **Automatic signal** generated by a join failure
- **Standard fault** (error condition in runtime environment, network communications etc)

## WSDL faults

Fault in WSDL are **denoted by <fault> element** within the operation declaration.

In the Travel Process Example we can modify the travel approval operation:

```
<portType name="TravelApprovalPT">
  <operation name="TravelApproval">
    <input message="tns:TravelRequestMessage"/>
    <output message="aln:TravelResponseMessage"/>
  </operation>
</portType>

<message name="TravelFaultMessage">
  <part name="error" type="xs:string" />
</message>

<portType name="TravelApprovalPT">
  <operation name="TravelApproval">
    <input message="tns:TravelRequestMessage" />
    <output message="aln:TravelResponseMessage" />
    <fault name="fault" message="tns:TravelFaultMessage" />
  </operation>
</portType>
```

BPEL process may sometimes need to **explicitly signal a fault**.

```
<throw faultName="name">
```

BPEL **don't require a fault names in advance**, this approach can also be error-prone

Fault can also have **associated a variable** that contain **data related to the fault**.

```
<throw faultName="name" faultVariable="variable-name" />
```

## Signaling Faults in synchronous reply

In synchronous invocation we can use the **reply activity to return a fault** instead of the output message.

In the travel example we can **add `travelFaultMessage`** starting from the definition of a new variable:

```
<variables>  
  <!-- fault to the BPEL client -->  
  <variable name="TravelFault" messageType="trv:TravelFaultMessage"/>  
</variables>
```

# Signaling fault in the synchronous travel example

```
<!-- Check if the ticket is approved -->
<if>
  <condition>
    $TravelResponse.confirmationData/aln:Approved='true'
  </condition>
  <!-- Send a response to the client -->
  <reply partnerLink="client"
    portType="trv:TravelApprovalPT"
    operation="TravelApproval"
    variable="TravelResponse"/>
<else>
  <sequence>
    <!-- Create the TravelFault variable with fault description -->
    <assign>
      <copy>
        <from>string('Ticket not approved')</from>
        <to variable="TravelFault" part="error" />
      </copy>
    </assign>

    <!-- Send a fault to the client -->
    <reply partnerLink="client"
      portType="trv:TravelApprovalPT"
      operation="TravelApproval"
      variable="TravelFault"
      faultName="fault" />
    </sequence>
  </else>
</if>
```

```
<TravelFault>
  <part name="error">
    <error xmlns="http://packtpub.com/bpel/travel/">
      Ticket not approved
    </error>
  </part>
</TravelFault>
```



## Signaling Faults in asynchronous scenario

- In the asynchronous scenario the fault is sent back using the **callback**.
- We define **additional callback** operation in the service WSDL using **the same port type**.

```
<message name="TravelFaultMessage">
  <part name="error" type="xs:string" />
</message>

<portType name="ClientCallbackPT">
  <operation name="ClientCallback">
    <input message="aln:TravelResponseMessage" />
  </operation>

  <operation name="ClientCallbackFault">
    <input message="tns:TravelFaultMessage" />
  </operation>
</portType>
```

# The invoke callback operation

```
<!-- Check if the ticket is approved -->
<if>
  <condition>
    $TravelResponse.confirmationData/aln:Approved='true'
  </condition>
  <!-- Make a callback to the client -->
  <invoke partnerLink="client"
    portType="trv:ClientCallbackPT"
    operation="ClientCallback"
    inputVariable="TravelResponse" />
<else>
  <sequence>
    <!-- Create the TravelFault variable with fault description -->
    <assign>
      <copy>
        <from>string('Ticket not approved')</from>
        <to variable="TravelFault" part="error" />
      </copy>
    </assign>

    <!-- Send a fault to the client -->
    <invoke partnerLink="client"
      portType="trv:ClientCallbackPT"
      operation="ClientCallbackFault"
      inputVariable="TravelFault" />
  </sequence>
</else>
</if>
```

# Handling Faults

- By means of a fault handler, the business process **defines custom activities** that are **used to recover the fault** and **recover the partial (unsuccessful) work** of the activity in which the fault has **occurred**
- The fault handler are specified between activities and variables.

```
<faultHandlers>
  <catch ... >
    <!-- Perform an activity -->
  </catch>
  <catch ... >
    <!-- Perform an activity -->
  </catch>
  ...
  <catchAll>
    <!-- catchAll is optional -->
    <!-- Perform an activity -->
  </catchAll>
</faultHandlers>
```

- <catch> activities to handle **specific faults**
- <catchAll> handle **all other faults**

Attributes:

- **faultName** : Specifies the name of the fault to be handled.
- **faultVariable** : Specifies the variable type used for fault data.

Additional exclusive attributes:

- **faultMessageType** : Specifies the WSDL message type of the fault to be handled.
- **faultElement** : Specifies the XML element type of the fault to be handled.

## Catch example flexibility

The most common permissive variation in the fault handler

```
<faultHandlers>
  <catch faultName="trv:TicketNotApproved" >
    <!-- First fault handler -->
    <!-- Perform an activity -->
  </catch>
  <catch faultName="trv:TicketNotApproved" faultVariable="TravelFault" >
    <!-- Second fault handler -->
    <!-- Perform an activity -->
  </catch>
  <catch faultVariable="TravelFault" >
    <!-- Third fault handler -->
    <!-- Perform an activity -->
  </catch>
  <catchAll>
    <!-- Perform an activity -->
  </catchAll>
</faultHandlers>
```

**NOTE:** fault handlers are very similar to **try/catch** clauses in modern programming languages

# Synchronous example

## Usage of the <reply> to signal the fault to the client

```
<faultHandlers>
  <catch faultName="trv:TicketNotApproved" faultVariable="TravelFault">
    <reply partnerLink="client"
      portType="trv:TravelApprovalPT"
      operation="TravelApproval"
      variable="TravelFault"
      faultName="fault" />
  </catch>
  <catchAll>
    <sequence>
      <assign>
        <copy>
          <from>string('Other fault')</from>
          <to variable="TravelFault" part="error" />
        </copy>
      </assign>
      <reply partnerLink="client"
        portType="trv:TravelApprovalPT"
        operation="TravelApproval"
        variable="TravelFault"
        faultName="fault" />
    </sequence>
  </catchAll>
</faultHandlers>
```

## Synchronous example

In the process instead of **reply** to the client we modify the process simply throwing a fault.

```
<!-- Check if the ticket is approved -->
<if>
  <condition>
    $TravelResponse.confirmationData/aln:Approved='true'
  </condition>
  <!-- Send a response to the client -->
  <reply partnerLink="client"
    portType="trv:TravelApprovalPT"
    operation="TravelApproval"
    variable="TravelResponse"/>
<else>
  <sequence>
    <!-- Create the TravelFault variable with fault description -->
    <assign>
      <copy>
        <from>string('Ticket not approved')</from>
        <to variable="TravelFault" part="error" />
      </copy>
    </assign>
    <!-- Throw fault -->
    <throw faultName="trv:TicketNotApproved"
      faultVariable="TravelFault" />
    </sequence>
  </else>
</if>
```

# Asynchronous Example

We need to define an additional **callback** for the fault.

```
<message name="TravelFaultMessage">
  <part name="error" type="xs:string" />
</message>
<portType name="ClientCallbackPT">
  <operation name="ClientCallback">
    <input message="aln:TravelResponseMessage" />
  </operation>
  <operation name="ClientCallbackFault">
    <input message="tns:TravelFaultMessage" />
  </operation>
</portType>
```



## Asynchronous Example

In the `<faultHandlers>` we send back the fault using the `<invoke>` instead of the `<reply>`

```
<faultHandlers>
  <catch faultName="trv:TicketNotApproved"
    faultVariable="TravelFault">
    <!-- Make a callback to the client -->
    <invoke partnerLink="client"
      portType="trv:ClientCallbackPT"
      operation="ClientCallbackFault"
      inputVariable="TravelFault" />
  </catch>
  <catchAll>
  <sequence>
    <!-- Create the TravelFault variable -->
    <assign>
      <copy>
        <from>string('Other fault')</from>
        <to variable="TravelFault" part="error" />
      </copy>
    </assign>
    <invoke partnerLink="client"
      portType="trv:ClientCallbackPT"
      operation="ClientCallbackFault"
      inputVariable="TravelFault" />
  </sequence>
</catchAll>
</faultHandlers>
```

# Propagation of Faults

BPEL gives the possibility to propagate a fault to a higher-level fault handler.

```
<catchAll>
  <sequence>
    <compensate />
    <rethrow />
  </sequence>
</catchAll>
```

- The `<rethrow>` activity to **rethrow the fault caught by the fault handler**
- Rethrow can be used only within `<catch>` and `<catchAll>`

## Inline Faults handling

- A more efficient way is to handle faults **directly in the activities**.
- The `<invoke>` activity provides a inline fault handlers without rely on a general fault handlers.

```
<invoke ... >
  <catch faultName="fault-name" >
    <!-- Perform an activity -->
  </catch>

  <catch faultName="fault-name"
    faultVariable="fault-variable"
    <!-- Perform an activity -->
  </catch>
  ...
  <catch faultName="fault-name"
    faultVariable="fault-variable"
    faultMessageType="WSDL-message" <!-- Optional one or the other -->
    faultElement="XML-element" >
    <!-- Perform an activity -->
  </catch>
</invoke>
```

# Sopes

---

- **Divide** complex business process into **hierarchically organized parts**.
- Provide **behavioural contexts for activities**.
- Allow to define **different fault handlers for different activities**

# How Scopes can be Defined

The scopes can be define in the following way:

```
<scope>
  <partnerLinks>
    <!-- Partner link definitions local to scope. -->
  </partnerLinks>
  <messageExchanges>
    <!-- Message exchanges local to scope. -->
  </messageExchanges>
  <variables>
    <!-- Variable definitions local to scope. -->
  </variables>
  <correlationSets>
    <!-- Correlation sets local to scope.-->
  </correlationSets>
  <faultHandlers>
    <!-- Fault handlers local to scope. -->
  </faultHandlers>
  <compensationHandler>
    <!-- Compensation handlers local to scope. -->
  </compensationHandler>
  <terminationHandler>
    <!-- Termination handler local to scope. -->
  </terminationHandler>
  <eventHandlers>
    <!-- Event handlers local to scope. -->
  </eventHandlers>
</scope>
```

## Scopes in the Travel Example

The travel example can be divided in three scopes:

- Retrieve the employee travel status (**RetrieveEmployeeTravelStatus**).
- Check the flight availability with both airlines (**CheckFlightAvailability**).
- Call back to the client ( **CallbackClient** ).

The **variables are limited to the scope** in which are defined

# Compensation

---



# Compensation

Consider the travel example enlarged with the following services:

- Reserve flight tickets
- Pay the ticket
- Reserve a hotel room
- Pay the room
  
- Consider the eventuality that the **business travel is not confirmed**
- The reservation and payment activities would have to be **undone - compensated**.
- In BPEL we need to **explicitly compensate** the different activities.
- Compensation is **required** in **long running** and **asynchronous loosely coupled communications**.
- BPEL require that the activity specifies a **reserve activity**, which can be **invoked if it is necessary to undo the effect** of that activity.

# Fault Handling vs Compensation

- **Fault handling**: a business process tries to **recover from an activity that could not finish normally** because an exceptional situation has occurred.
- **Compensation**: **reverse the effects of a previous activity** or a set of activities that have been carried out successfully as part of a business process that is being abandoned.

Compensation handler can be defined for:

- The **scope**
- **Inline** for the `<invoke>` activity

# Compensation handler for the scope

```
<scope>
  <partnerLinks>
    <!-- Partner link definitions local to scope. -->
  </partnerLinks>
  <messageExchanges>
    <!-- Message exchanges local to scope -->
  </messageExchanges>
  <variables>
    <!-- Variable definitions local to scope. -->
  </variables>
  <correlationSets>
    <!-- Correlation sets local to scope-->
  </correlationSets>
  <faultHandlers>
    <!-- Fault handlers local to scope. -->
  </faultHandlers>
  <compensationHandler>
    <!-- Compensation activity
      (or several activities within a <sequence>, <flow>, or other structured activity) -->
  </compensationHandler>
  <terminationHandler>
    <!-- Termination handler local to scope -->
  </terminationHandler>
  <eventHandlers>
    <!-- Event handlers local to scope-->
  </eventHandlers>
  activity
</scope>
```

BPEL provides a shortcut for defining inline compensation handler rather than explicitly using an immediately enclosing scope.

```
<invoke ... >  
  <compensationHandler>  
    <!-- Compensation activity  
      (or several activities within a <sequence>, <flow>,  
      or other structured activity) -->  
  </compensationHandler>  
</invoke>
```

# Events

---

# Managing Events

Business process may have to react on certain events:

- Wait for an **incoming message** <receive> activity
- Wait for a **callback message**
- Wait for the **callback within a certain period of time**

In most of the case the system should **react on two types of events**:

- **Message events**: These are triggered by incoming messages through operation invocation on port types.
- **Alarm events**: These are time related and are triggered either after a certain duration or at a specific time.

# Pick Activity

The `<pick>` activity **awaits the occurrence of one set of events.**

Event can be handled using:

- `<onMessage>` (1 .. n)
- `<onAlarm>` (0 .. n)

Syntax of pick activity:

```
<pick>  
  <onMessage ...>  
    <!-- Perform an activity -->  
  </onMessage>  
  <onMessage ...>  
    <!-- Perform an activity -->  
  </onMessage>  
  ...  
  <onAlarm ...>  
    <!-- Perform an activity -->  
  </onAlarm>  
  ...  
</pick>
```

# Message Events

On message events we have to specify the following **attributes**:

- **PartnerLink** : Specifies which partner link will be used for the invoke
- **PortType** : Specifies the used port type
- **Operation** : Specifies the name of the operation to wait for being invoked
- **Receive or reply variable** : Specifies the name of the variable used to store the incoming message

```
<pick>
  <onMessage partnerLink="name"
    portType="name"
    operation="name"
    variable="name">
    <!-- Perform an activity or a set of activities enclosed by
      <sequence>, <flow>, etc. or throw a fault -->
  </onMessage>
  ...
</pick>
```



# Alarm Events

The `<onAlarm>` element is similar to the `<wait>` element indeed **we use the same literal.**

On alarm events we can specify the following:

- A **duration expression** using a `<for>` duration expression
- A **deadline expression** using an `<until>` deadline expression

Code example with hard-coded times/dates:

```
<pick>
  <onAlarm>
    <for>'PT15M'</for>
    <!-- Perform an activity or a set of activities enclosed by
         <sequence>, <flow>, etc. or throw a fault -->
  </onAlarm>
</pick>
<pick>
  ...
  <onAlarm>
    <until>'2004-03-18T21:00:00+01:00'</until>
    <!-- Perform an activity or a set of activities enclosed by
         <sequence>, <flow>, etc. or throw a fault -->
  </onAlarm>
</pick>
```

# Travel Example using the Pick Activity

We substitute the `<receive>` activity with the `<pick>`.

```
<pick>
  <onMessage partnerLink="AmericanAirlines"
    portType="aln:FlightCallbackPT"
    operation="FlightTicketCallback"
    variable="FlightResponseAA">
    <empty/>
  </onMessage>
  <onMessage partnerLink="AmericanAirlines"
    portType="aln:FlightCallbackPT"
    operation="FlightNotAvaliable"
    variable="FlightFaultAA">
    <throw faultName="trv:FlightNotAvaliable"
      faultVariable="FlightFaultAA"/>
  </onMessage>
  <onMessage partnerLink="AmericanAirlines"
    portType="aln:FlightCallbackPT"
    operation="TicketNotAvaliable"
    variable="FlightFaultAA">
    <throw faultName="trv:TicketNotAvaliable"
      faultVariable="FlightFaultAA"/>
  </onMessage>
  <onAlarm>
    <for>'PT30M'</for>
    <throw faultName="trv:CallbackTimeout" />
  </onAlarm>
</pick>
```

# Event Handler

---

# Event handlers

- With `<pick>` the business process **should wait for events**
- With `Event handler` the system **react on events** that occur while BP executes.

The event handler permit the system to **continue the execution** and still **listen the events** and **handle them** whenever they occur.

- Event Handler is **invoked concurrently** with the business process

**Typical usage** of event handler usage could be the **cancellation message** from the client referring the travel example

# Event handlers

Event handlers can be specified for the whole process as well for each scope.

```
<process ...>
  <partnerLinks>
    ...
  </partnerLinks>
  <variables>
    ...
  </variables>
  <faultHandlers>
    ...
  </faultHandlers>
  <compensationHandler>
    ...
  </compensationHandler>
  <eventHandlers>
    <onEvent ...>
      <!-- Perform activities -->
    </onEvent>
    ...
    <onAlarm ...>
      <!-- Perform activities -->
    </onAlarm>
    ...
  </eventHandlers>
  activity
</process>
```

The syntax of Event Handler is similar to the `<pick>` activity.  
There are two typology of events:

- `<onEvent>` for message event (1 .. n)
- `<onAlarm>` for deadline (1 .. n)

The **event handler** will wait for a message to arrive.

The event syntax is :

```
<eventHandlers>
  <onEvent partnerLink="name"
    portType="name" <!-- Optional -->
    operation="name"
    messageType="name" <!-- Optional -->
    element="name" <!-- Optional -->
    variable="name" <!-- Optional -->
    messageExchange="name"> <!-- Optional -->
    <correlations>
      <correlation set="name" initiate="yes|join|no"? />
    </correlations>
    <fromParts> <!-- Optional -->
      <fromPart part="name" toVariable="name" />
    </fromParts>
    <scope>
      <!-- Perform activities -->
    </scope>
  </onEvent>
  ...
</eventHandlers>
```

- **Partnerlink** partnerlink of the service that can send the message.
- **Operation**: Operation of the partner that cause the event
- **PortType**: contain the operation that will be invoked
- **Variable**: contain the message exchanged by the partners. (It is possible to specify the MessageType or the element)



# Event Handler on Travel Example

```
<process name="Travel" ... >
  ...
  <eventHandlers>
    <onEvent partnerLink="client"
      portType="trv:TravelApprovalPT"
      operation="CancelTravelApproval"
      messageType="trv:TravelRequestMessage"
      variable="TravelRequest" >
      <scope>
        <exit/>
      </scope>
    </onEvent>
  </eventHandlers>
  ...
</process>
```

The `<onAlarm>` element indicates a **time-driven event**

- We can specify a **time duration** after which the event will be signaled. (`<for>`)
- We can specify a **specific point in time (deadline)** when the alarm will be fired. (`<until>`)
- We can also specify a **repeating duration event**. An alarm will be fired repeatedly each time the duration period expires. (`<repeatEvery>`)

# onAlarm syntax

The syntax of an `<onAlarm>` handler is shown as follows:

```
<eventHandlers>
  <onAlarm>
    <for>duration-expression</for> <!-- Optional -->
    <until>deadline-expression</until> <!-- Optional -->
    <repeatEvery>
      duration-expression
    </repeatEvery> <!-- Optional -->
    <scope>
      <!-- Perform activities -->
    </scope>
  </onAlarm>
  ...
</eventHandlers>
```

The system handle an error for 12 hours:

```
<process name="Travel" ... >
  ...
  <eventHandlers>
    <onAlarm>
      <for>'PT12H'</for>
      <scope>
        <exit/>
      </scope>
    </onAlarm>
  </eventHandlers>
  ...
</process>
```

# **Business process lifecycle**

---

The business process lifecycle:

- Start with the `<alert>/<pick>` activity
- Typically invokes several operation on partner web services.
- Wait for partners callback
- Terminate after all activity are performed

In BPEL we **do not create instances explicitly** like in programming languages.

# Business process lifecycle

## Travel Business process lifecycle.

```
...
<sequence>
  <!-- Receive the initial request for business travel from client-->
  <receive partnerLink="client"
    portType="trv:TravelApprovalPT"
    operation="TravelApproval"
    variable="TravelRequest"
    createInstance="yes" />
  ...
  <pick createInstance="yes">
    <onMessage partnerLink="client"
      portType="trv:TravelApprovalPT"
      operation="TravelApproval"
      variable="TravelRequest" >
      <!-- Perform activities -->
    </onMessage>
    <onMessage partnerLink="client"
      portType="trv:TravelCancellationPT"
      operation="TravelCancellation"
      variable="TravelCancel" >
      <!-- Perform activities -->
    </onMessage>
  </pick>

```

## **Correlation and message properties**

---

## Correlation set

BPEL uses the notion of properties to assign global names to relevant data used for correlation messages.

Two roles can be defined:

- **Initiator:** is the partner that sends the first message in an operation invocation
- **Followers:** other partners that interact using the same message.

A correlation set is used to **associate messages with business process instances**.

```
<correlationSets>
  <correlationSet name="correlation-set-name"
    properties="list-of-properties"/>
  <correlationSet name="correlation-set-name"
    properties="list-of-properties"/>
</correlationSets>
```

```
<process ... >
  <partnerLinks>...</partnerLinks>
  <variables>...</variables>
  <correlationSets>
    <correlationSet name="TicketOrder"
      properties="aln:FlightNo"/>
  </correlationSets>
  ...
```



## Using Correlation Set

Correlation can be used in the `<invoke>`, `<receive>`, `<reply>`, `<onMessage>`, `<pick>` and `<onEvent>`

It is necessary to define where the correlation applies

```
<correlations>
  <correlation set="name" initiate="yes|join|no"
    <!-- Optional -->
    pattern="request|response|request-response" />
</correlations>
```

# Correlation Set in the Travel example

```
<sequence>
  <!-- Check the flight availability -->
  <invoke partnerLink="AmericanAirlines"
    portType="aln:FlightAvailabilityPT"
    operation="FlightAvailability"
    inputVariable="FlightDetails" />
  <!-- Wait for the callback -->
  <receive partnerLink="AmericanAirlines"
    portType="aln:FlightCallbackPT"
    operation="FlightTicketCallback"
    variable="TravelResponse" >
  <!-- The callback includes flight no therefore initiate correlation set -->
  <correlations>
    <correlation set="TicketOrder"
      initiate="yes" />
  </correlations>
</receive>
...
<!-- Synchronously confirm the ticket -->
<invoke partnerLink="AmericanAirlines"
  portType="aln:TicketConfirmationPT"
  operation="ConfirmTicket"
  inputVariable="FlightResponseAA"
  outputVariable="Confirmation" >
<!-- Use the correlation set to confirm the ticket -->
<correlations>
  <correlation set="TicketOrder"
    pattern="request-response" />
</correlations>
```

## Correlation Set in the Travel example

```
<!-- Make a callback to the client -->
<invoke partnerLink="client"
  portType="trv:ClientCallbackPT"
  operation="ClientCallback"
  inputVariable="TravelResponse" >
  <!-- Use the correlation set to callback the client -->
  <correlations>
    <correlation set="TicketOrder"
      pattern="request" />
  </correlations>
</invoke>
</sequence>
</process>
```

## Dynamic Partner Links

---

## Dynamic Partner links

Usually:

- Partner links are **defined at design time**
- A **single endpoint is defined for each role**

With dynamic partner links:

- Partner link **endpoint can be referenced at runtime**
- BPEL can **decide which web service can used at runtime**

BPEL should communicate with **web services that have the same WSDL interface** (American and Delta airline in the travel service)

## Endpoint references

- BPEL uses **endpoint references defined by WS-Addressing**
- For each BPEL **instance** and for each **partner role** in a partner link a **unique endpoint** reference exists

We can copy:

- from a partner link to a partner link
- from a partner link to a variable
- from a variable to a partner link

```
<assign>
  <copy>
    <from partnerLink="name"
      endpointReference="myRole|partnerRole"/>
    <to partnerLink="name"/>
  </copy>
</assign>
```

```
<assign>
  <copy>
    <from partnerLink="name"
      endpointReference="myRole|partnerRole"/>
    <to variable="varName"/>
  </copy>
</assign>
```

## Endpoint reference

To dynamically assign an endpoint reference to a partner link, we have to withdraw a service from the **service reference container** `<sref:service-ref>` and copy it to the partner link or copy the XML literal to the partner link.

The endpoint reference are contained in a reference container

```
<EndpointReference xmlns="http://schemas.xmlsoap.org/ws/2004/08/
addressing">
  <Address>ServiceURL</Address>
  <ReferenceProperties>...</ReferenceProperties> <!--optional-->
  <ReferenceParameters>...</ReferenceParameters> <!--optional-->
  <PortType>PortTypeName</PortType> <!--optional-->
  <ServiceName PortName="...">ServiceName</ServiceName> <!--optional-->
</EndpointReference>
```

# Dynamic binding using literal

```
<assign>
  <copy>
    <from>
      <literal>
        <sref:service-ref
          xmlns:sref="http://docs.oasis-
            open.org/wsbpel/2.0/serviceref">
          <EndpointReference
            xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
              <Address>
                http://www.soa.si/default/AmericanAirline
              </Address>
            </EndpointReference>
          </sref:service-ref>
        </literal>
      </from>
      <to partnerLink="Airline"/>
    </copy>
  </assign>
```



# **Building BPEL application Using JDeveloper**

---

# SOA Editor

The screenshot displays the Oracle Developer 11g Release 1 SOA Editor interface. The main workspace shows a composite application diagram for 'TravelApproval'. The diagram is organized into three columns: Exposed Services, Components, and External References.

- Exposed Services:** Contains a 'TravelApproval' component with an operation 'TravelApproval'.
- Components:** Contains a central 'TravelAppro...' component.
- External References:** Contains three external services:
  - 'EmployeeTrav...' with operations 'EmployeeTrav...'.
  - 'AmericanAirlin...' with operations 'FlightAvailability', 'MakeReservation', 'FlightTicketCall...', and 'MakeReservatio...'.
  - 'DeltaAirlines' with operations 'FlightAvailability', 'MakeReservation', 'FlightTicketCall...', and 'MakeReservatio...'.

The diagram shows a flow from the 'TravelApproval' component to the 'TravelAppro...' component, which then references the three external services. The 'Component Palette' on the right lists various SOA components like 'Service Components', 'Business Rule', 'Human Task', 'Mediator', 'Spring Context', 'Service Adapters', 'AOF-BC Service', 'AQ Adapter', 'EJB', 'BAM Adapter', 'Database Adapter', 'Direct Binding', 'EJB Service', 'File Adapter', 'FTP Adapter', 'HTTP Binding', 'JMS Adapter', 'MQ Adapter', 'Oracle Applications', 'Socket Adapter', 'Third Party Adapter', and 'Web Service'. The 'Property Inspector' at the bottom right shows 'Documentation not available'.

# Defining XML schemas

The screenshot displays the Oracle JDeveloper 11g Release 1 interface for editing an XML Schema Definition (XSD) file named `TravelRequestType.xsd`. The main editor shows the following schema structure:

```
<schema>  
  targetNamespace="http://packpub.com/bpel/travel/"  
  
  <import>  
    schemaLocation="FlightRequestType.xsd"  
    namespace="http://packpub.com/service/airline/"  
  
  <import>  
    schemaLocation="EmployeeType.xsd"  
    namespace="http://packpub.com/service/employee/"  
  
  <element base="tns:TravelRequestType" type="tns:TravelRequestType"/>  
  
  <complexType base="TravelRequestType" type="TravelRequestType">  
    <sequence>  
      <element type="bons0:EmployeeType" base="EmployeeType"/>  
      <element type="bons1:FlightRequestType" base="FlightRequestType"/>  
    </sequence>  
  </complexType>  
  
  <complexType base="TravelRequestType" type="TravelRequestType">  
    <sequence>  
      <element type="bons0:EmployeeType" base="EmployeeType"/>  
      <element type="bons1:FlightRequestType" base="FlightRequestType"/>  
    </sequence>  
  </complexType>
```

The graphical structure view on the right shows the `TravelRequestType` element (orange box) containing two child elements: `employee` (type: `bons0:EmployeeType`) and `flightData` (type: `bons1:FlightRequestType`).

The interface includes a Project Explorer on the left showing the project structure, a Schema Components palette on the right, a Property Inspector at the bottom right, and a Design Source tab at the bottom. The status bar at the bottom indicates the file path: `C:\JDeveloper\mywork\TravelApproval\TravelApproval\composite.xml`.

# Defining XML schemas source

The screenshot displays the Oracle JDeveloper IDE interface. The main window shows the XML schema source code for `TravelRequestType.xsd`. The code defines a schema with the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:bons0="http://packtpub.com/service/airline/"
  xmlns:bons1="http://packtpub.com/service/employee/">
  <xsd:import namespace="http://packtpub.com/service/airline/" schemaLocation="http://packtpub.com/service/airline/airline.xsd"/>
  <xsd:import namespace="http://packtpub.com/service/employee/" schemaLocation="http://packtpub.com/service/employee/employee.xsd"/>
  <xsd:element name="TravelApproval" type="tns:TravelRequestType"/>
  <xsd:complexType name="TravelRequestType">
    <xsd:sequence>
      <xsd:element name="employee" type="bons0:EmployeeType" minOccurs="0"/>
      <xsd:element name="flightData" type="bons1:FlightRequestType" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The IDE interface includes a Project Explorer on the left showing the project structure, a Schema Components panel on the right listing various schema constructs, and a Property Inspector at the bottom right. The status bar at the bottom indicates the current file path and editing mode.

# Defining WSDL

Oracle JDeveloper 11g Release 1 - TravelApproval.jes : TravelApproval.gr : C:\Developer\mywork\TravelApproval\TravelApproval\TravelApproval.wsdl

File Edit View Application Refactor Search Navigate Build Run Versioning Tools Window Help

Application EP... TravelApproval

Projects

- Project1
- TravelApproval
  - SOA Content
    - classes
    - testoutlets
    - FileList.xml
    - xsd
      - AirlineTypes.xsd
      - EmployeeType.xsd
      - FlightConfirmationType.xsd
      - FlightRequestType.xsd
      - FlightReservationType.xsd
      - TravelClassType.xsd
      - TravelRequestType.xsd
    - Business Rules
    - Airline.wsdl
    - composite.xml
  - Application Resources
  - Data Controls
  - Recently Opened Files

TravelApproval.wsdl - Structure

- definitions
- Polices
- Imports
- Types
- PartnerLinkTypes
- Services
- bindings
- PortTypes
- ...

Design Source

Design Schema Source History

BPEL - Log Simulations Documentation

BPEL TravelApproval.bpel Warnings: Errors: 1 Warnings: 1

Validation Search

Messages Extensions SOA BPEL Web Services

Last Validated On: 13 Jun 2010 08:00:17 GMT

Search Document

Search

WSDL

- binding
- bound operation
- definitions
- Fault
- input
- input
- message
- operation
- output
- part
- port
- portType
- service

Drag from Component Palette or use Create (c) or Drop a binding (b) to create a new service

Messages

- TravelRequestMessage
  - part - trns:TravelApproval travelRequest

Port Types

- TravelApprovalPT
  - input
  - output

Bindings / Partner Link Types

- travelLT
  - trns:TravelApprovalPT

Services

# SOA Composite Application

Oracle JDeveloper 11g Release 1 - TravelApproval.jws : TravelApproval.jpr : C:\JDeveloper\mywork\TravelApproval\TravelApproval\composite.xml

File Edit View Application Refactor Search Navigate Build Run Versigning Tools Window Help

TravelApproval Overview composite.xml

Composite: TravelApproval

Exposed Services Components External References

To begin creating a SOA composite application, drag-and-drop a Service Component or an Adapter from the Component Palette

Component Palette:

- Service Components
  - BPEL Process
  - Business Rule
  - Human Task
  - Mediator
  - Spring Context
- Service Adapters
  - ADF-BC Service
  - AQ Adapter
  - B2B
  - BAM Adapter
  - Database Adapter
  - Direct Binding
  - FR Service

Project Structure:

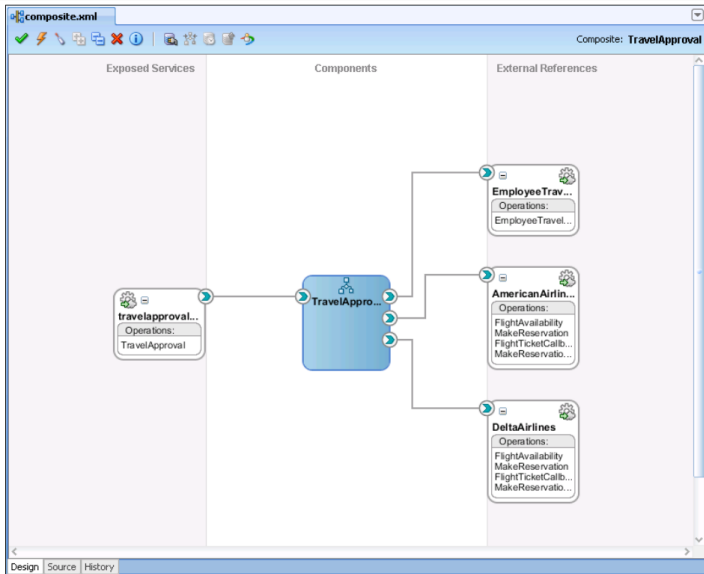
- TravelApproval
  - SOA Content
    - classes
    - testsuites
      - fileList.xml
    - xsd
    - xsl
    - Business Rules
    - composite.xml

Messages - Log

Creating adf-config.xml for workspace TravelApproval.jws  
Adding /soa/shared metadata namespace and store definition in file:/C:/JDeveloper/mywork

Open nodes (16); Saved nodes(4) | Design Editing

# SOA Composite Application



# BPEL Operation Design

The screenshot displays the Oracle JDeveloper 11g Release 1 IDE interface for editing a BPEL process named `TravelApproval.bpel`. The main workspace shows a diagram of the process flow:

- A `client` partner link is connected to the `receiveInput` activity.
- The `receiveInput` activity is followed by a `replyOutput` activity.
- Partner links for `EmployeeTravelSt...`, `AmericanAirlines`, and `DeltaAirlines` are listed on the right.

The interface includes a Project Explorer on the left showing the file structure, a Properties window at the bottom left, and a palette of BPEL activities on the right. The bottom status bar indicates "Opened nodes (24); Saved nodes(4)".



# Partner Link Definition

**Create Partner Link**

General Image Property

Name: AmericanAirlines

Process:

WSDL Settings

WSDL URL: services/default/AmericanAirlines/TicketService?WSDL

Partner Link Type: flightLT

Partner Role: airlineService

My Role: airlineCustomer

Help Apply OK Cancel

# Variables Definition

The screenshot shows a dialog box titled "Edit Variable - inputValue". The "General" tab is selected. The "Name" field contains the text "TravelRequest". Under the "Type" section, the "Message Type" radio button is selected, and the text field next to it contains the URI "{http://packtpub.com/bpel/travel/}TravelReq...". The "Entity Variable" checkbox is unchecked, and the "SDO Capable" checkbox is also unchecked. At the bottom of the dialog, there are four buttons: "Help", "Apply", "OK", and "Cancel".

**Edit Variable - inputValue**

General

Name:

Type

Simple Type

Message Type

Element

Entity Variable

Partner Link:

SDO Capable

Help Apply OK Cancel

# Invoke Operation


**Invoke** ✕

🔗


Headers Annotations Assertions Skip Condition


General Correlations Properties

Name:



Interaction Type:  Partner Link ▼



Partner Role Web Service Interface

Partner Link:  

Operation:  EmployeeTravelStatus ▼

Variables

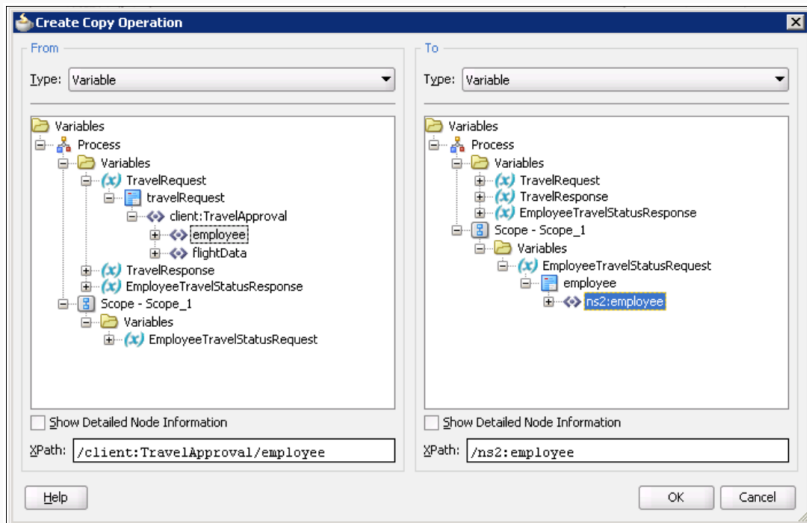
Input:   

Output:   

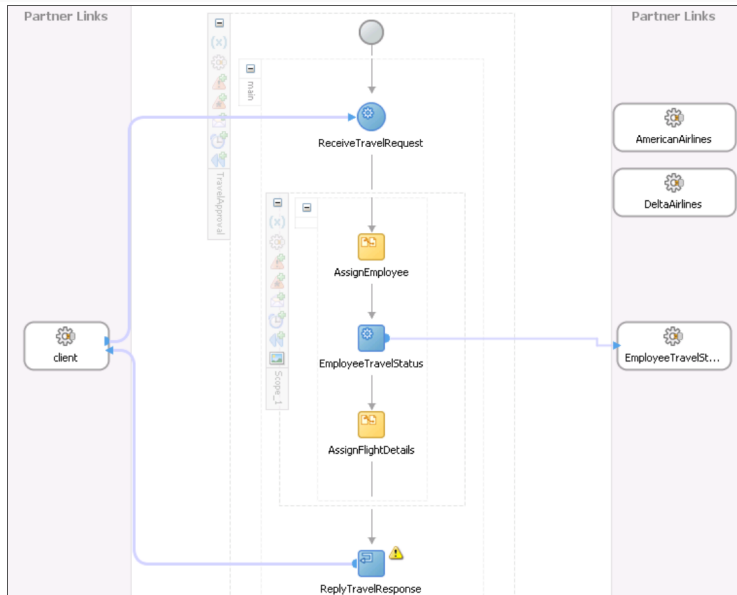
Options

Conversation ID:

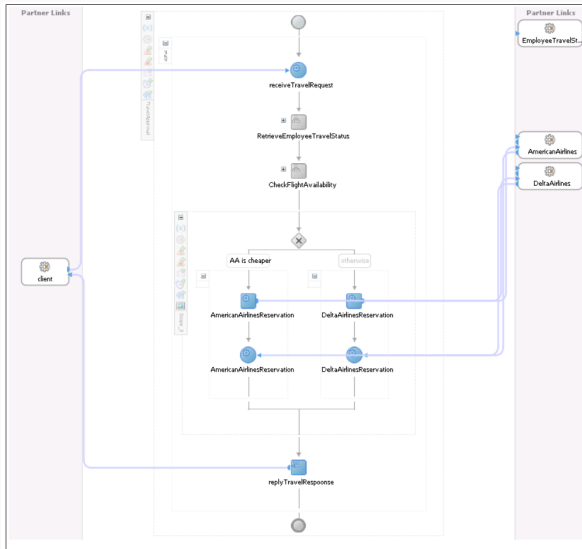
# Assign Operation



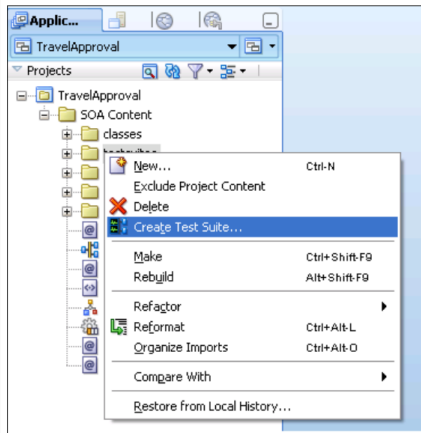
# Definition of variables in the Operations



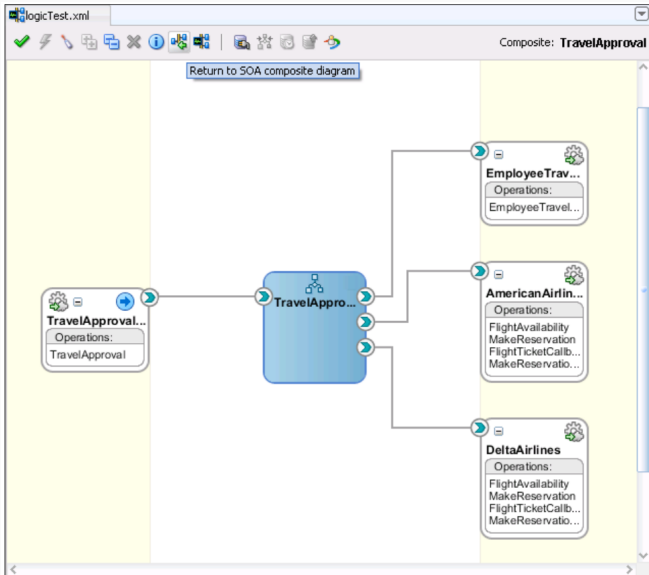
# BPEL Complete Process



# Test Suite Creation



# Composite Diagram





# TravelRequest Operation

Initiate Messages

Operations

TravelApproval

Initiate Message For Operation <TravelApproval>

Message Parts

Part: travelRequest

Value:

Enter Manually  Load From File

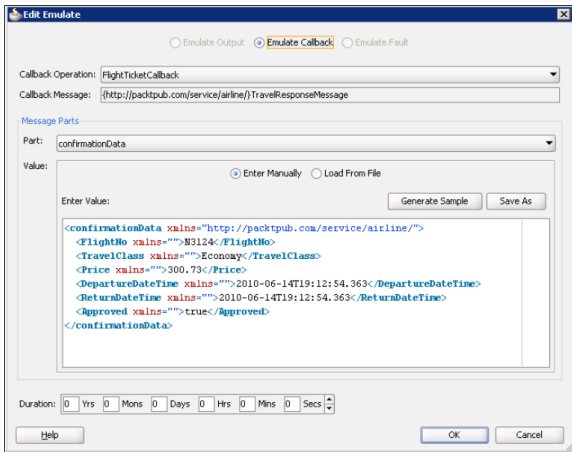
Enter Value:

```
<TravelApproval xmlns="http://packtpub.com/bpel/travel/">
  <employee xmlns="">
    <FirstName>Marcel</FirstName>
    <LastName>Krizevnik</LastName>
    <Department>SIL</Department>
  </employee>
  <flightData xmlns="">
    <RequestNo>RequestNo24</RequestNo>
    <OriginFrom>Ljubljana</OriginFrom>
    <DestinationTo>Paris</DestinationTo>
    <DesiredDepartureDate>2010-06-14</DesiredDepartureDate>
    <DesiredReturnDate>2010-06-14</DesiredReturnDate>
  </flightData>
</TravelApproval>
```

Delay: 0 Yrs 0 Mons 0 Days 0 Hrs 0 Mins 0 Secs

Help

# TicketCallback Operation



# Deploy

---

# Enterprise Manager Console

ORACLE Enterprise Manager Fusion Middleware Control 11g Setup Help Log


Farm ▼ Topology

Farm\_domain1

Application Deployments  
SOA  
WebLogic Domain  
BAM  
Metadata Repositories  
User Messaging Service

Farm\_domain1


Deployments



100%

Name	Status	Target
Application Deployments		
Internal Applications		
Resource Adapters		
/DopustiDopustiService-context-roo	↑	soa_server1
BPMComposer	↑	soa_server1
composer	↑	soa_server1
DefaultToDoTaskFlow	↑	soa_server1
InternalNarocla-InternalNaroclaServi	↑	soa_server1
oer	↑	oer_server1
orade-bam(11.1.1)	↑	bam_server1
OracleBPMComposerRolesApp	↑	soa_server1
OracleBPMProcessRolesApp	↑	soa_server1
OracleBPMWorkspace	↑	soa_server1
PoslovnaPotovanja-PotovanjaRazredi	↑	soa_server1
TFormDopolnilevInternegaNarocila	↑	soa_server1
TFormKonrolaInternegaNarocila	↑	soa_server1

Fusion Middleware



100%

Name	Status	Host	CPU Usage (%)
WebLogic Domain			
domain1			
AdminServer	↑	VMOrasOA11gPS2	2.69
bam_server1	↑	VMOrasOA11gPS2	1.96
oer_server1	↑	VMOrasOA11gPS2	
soa_server1	↑	VMOrasOA11gPS2	2.08
BAM			
OracleBamServer (bam)	↑	VMOrasOA11gPS2	
OracleBamWeb (bam)	↑	VMOrasOA11gPS2	
Metadata Repositories			
mds-owsm		VMOrasOA11gPS2	
mds-soa		VMOrasOA11gPS2	
User Messaging Service			
usermessagingdriver~	↑	VMOrasOA11gPS2	
usermessagingdriver~	↑	VMOrasOA11gPS2	

# Deploy and Undeploy Services



The screenshot shows the SOA Infrastructure Control Center interface. A navigation menu on the left is open, highlighting the 'Control' option. A context menu is displayed over a table of service instances, offering 'Deploy...', 'Undeploy...', and 'Redeploy...' actions. The table shows service instances with their start times.



Start Time
15-Jun-2010 10:03:58
14-Jun-2010 19:45:58
14-Jun-2010 19:45:58
14-Jun-2010 19:41:02
14-Jun-2010 19:41:01
14-Jun-2010 18:13:02
14-Jun-2010 18:13:01
14-Jun-2010 18:10:04
14-Jun-2010 18:09:49

Below the table, there is a section for 'Messages' and a 'Recovery' table.


Recovery
15-Jun-201
15-Jun-201

# Test Instances

TravelApproval [1.0]  Logged in as **weblogic** | Host VMOrSOA11  
SOA Composite  Page Refreshed 15-Jun-2010 11:11:50 CES










Running Instances 0 | Total 27 | Active | Retire ... | Shut Down... | Test | Settings...  


**Dashboard** | Instances | Faults and Rejected Messages | Unit Tests | Policies



### Recent Instances







Show Only Running Instances  Running 0 Total 27

Instance ID	Name	Conversation ID	State	Start Time
240227			 Faulted	15-Jun-2010 10:03:58
 240225			 ---	14-Jun-2010 19:45:58
 240223			 Stale	14-Jun-2010 19:41:01
 240221			 Stale	14-Jun-2010 18:13:01
 240219			 Stale	14-Jun-2010 18:09:49

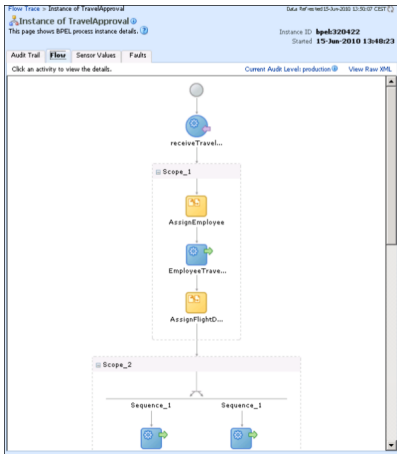
 Show All

### Recent Faults and Rejected Messages

Show only system faults

Error Message	Recovery	Fault Time	Fault Location	Composite Instance ID	Log
 FaultName: {http://scherr		15-Jun-2010 10:03:58	 client_ep	240227	
 <bpelFault><faultType>0		15-Jun-2010 10:03:58	 TravelApproval	240227	

# Runtime information



Activity Audit Trail - Mozilla Firefox

http://localhost:7001/em/ai/sca/share/audit/rfdg/dlgElementDetails.jsp

**receiveTravelRequest**



[2010/06/15 13:48:23]  
Received "TravelRequest" call from partner "client"


```
- <TravelRequest>
- <part name="travelRequest" xmlns:ns1="http://www.w3.org/2001/XMLSchema-instance">
- <ns1:TravelApproval xmlns:ns1="http://packtpub.com/bpel/travel/">
- <employee>
  <FirstName>Hazel</FirstName>
  <LastName>Krisenak</LastName>
  <Department>SIL</Department>
</employee>
- <flightData>
  <RequestNo>45243</RequestNo>
  <OriginFrom>Ljubljana</OriginFrom>
  <DestinationTo>Paris</DestinationTo>
  <DesiredDepartureDate>2010-12-15+01:00</DesiredDepartureDate>
  <DesiredReturnDate>2010-12-17+01:00</DesiredReturnDate>
</flightData>
</ns1:TravelApproval>
</part>
</TravelRequest>
```





[Copy details to clipboard](#)

KonZano

# Test suite

TravelApproval [1.0]  Logged in as **weblogic** | Host VMOrasOA11gP52  
Page Refreshed 15-Jun-2010 14:18:50 CEST 

SOA Composite 






Running Instances: 0 | Total 31 | Active | Retire ... | Shut Down... | Test | Settings...    Related Links 


Dashboard | Instances | Faults and Rejected Messages | **Unit Tests** | Policies


Test Cases | **Test Runs**

**Search**




Click a test run to view its details.

Test Run Name	Test Run ID	Start Time	End Time	Status	Success Rate
Run1	317473e11b	15-Jun-2010 14:28:23	15-Jun-2010 14:28:24	 Passed	100% (of 1 tests, 0 failed, all 1 tests cor
2	317473e11b	14-Jun-2010 19:45:58	14-Jun-2010 19:45:58	 Failed	0% (of 1 tests, 1 failed, all 1 tests comp
1	317473e11b	14-Jun-2010 19:41:01	14-Jun-2010 19:41:02	 Passed	100% (of 1 tests, 0 failed, all 1 tests cor
test2	317473e11b	14-Jun-2010 18:13:01	14-Jun-2010 18:13:01	 Failed	0% (of 1 tests, 1 failed, all 1 tests comp
test1	317473e11b	14-Jun-2010 18:09:50	14-Jun-2010 18:10:06	 Passed	100% (of 1 tests, 0 failed, all 1 tests cor

**Results of Test Run : Run1 (Test Run ID : 317473e11b4e8752:14f80924:128b4f416f2:-7c2b) **

Total 1    Running 0    Passed 1    Failed 0    Unknown 0    Success Rate 100%     Refresh Test Status

Expand a test suite to view the status of each test case. Click a test suite or test case to view assertion details.

Test suites and test cases	Status
 TestSuite1	
 logicTest.xml	 Passed



## **JDeveloepr:**

`http://www.oracle.com/technetwork/  
developer-tools/jdev/downloads/index.html`

**Questions?**