PYMMA

OpenESB Standalone Edition
CASA Editor

PYMMA

Document identifier:

Pymma document: 770-009

Location:

www.pymma.com

Editor:

Pymma Services: contact@pymma.com

Abstract:

This document provides a user guide for OpenESB CASA Editor

Status:

This document is in a beta state.

ABOUT PYMMA CONSULTING

Pymma Services is a technical architect bureau founded in 1999 and headquartered in London, United Kingdom. It provides expertise in service oriented integration systems design and implementation. Leader of OpenESB project, Pymma is recognised as one of the main actors in the integration landscape. It deeply invests in open source projects such as Drools rules engine. Pymma is a European company based in London with regional offices in France, Belgium and Canada. (contact@pymma.com or visit our website at www.pymma.com)

Copyright

Disclaimer

Information in this document is subject to change without notice and does not represent a commitment on the part of Pymma Services LTD. This manual is provided “as is” and Pymma is not responsible for disclaims all warranties of any kind with respect to third-party content, products, and services. Pymma will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

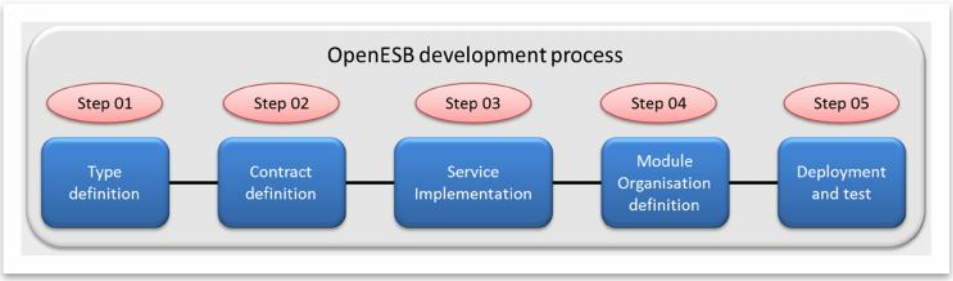
Trademark Notice

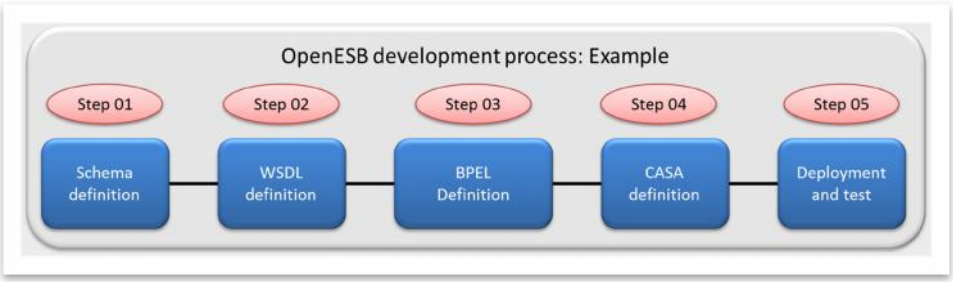
Pymma is a registered of Pymma Engineering LTD. Java is registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

1 Introduction

OpenESB (OE) offers a well defined development process for service oriented projects. This development process is simple, straightforward, rigorous but efficient. It has been used with success for thousands of projects all around the word for small to big companies or governmental organisations such as Airbus, General Electric, HMCR, Leroy-Merlin, Humanis, etc.…





Currently, “Module Organisation Definition” step is supported by the CASA Editor. CASA means: “Composite Application Service Assembly”. In this document we will use CASA, Composite Application or Service Assembly indifferently.

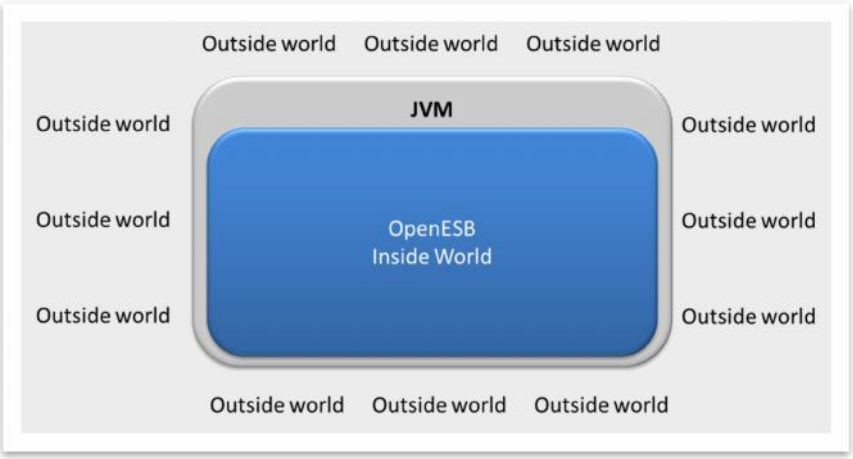
The aim of this document is to explain in detail what composite applications are, how we define them in OpenESB, and how they provide flexibility, agility and promote reusing in OpenESB projects.

OpenESB was the first integration tools to implement Service Assembly principles. Then many other editors such as Oracle, WebMethod or IBM used through SCA specifications, this powerful concept in their own development tools.

This document is not an exhaustive documentation on CASA Editor (“quality of service” does not feature in this paper) but user guidelines. It focuses on the composite application concepts and implementation and provides a simple but persuasive use case which shows CASA ability to provide flexibility for your integration projects.

2 Technical background

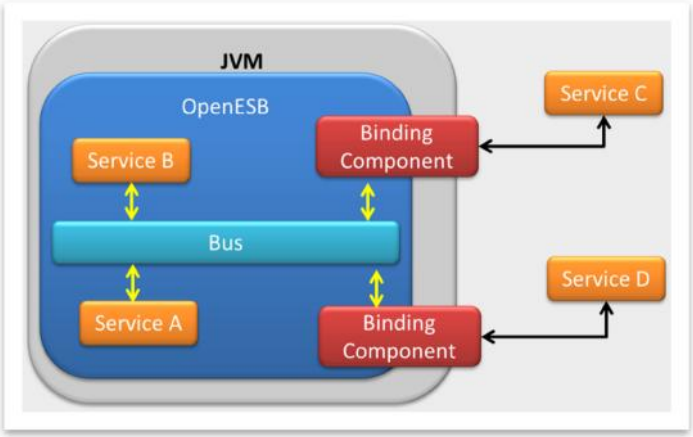
To understand how a composite application works we have to understand some basic concepts on OpenESB (more details on JBI Specifications[1]). OE designers made a difference between partners (or services) running in the same JVM than OpenESB and partners (or services) running outside OpenESB environment. So we can say that a JVM acts as a boundary between the outside and inside world.



OpenESB designers defined the following rule:

Partners (or Services) running in OpenESB environment invoking one another communicate directly through the bus.
Partners (or Services) running in OpenESB environment invoking a service running outside OpenESB environment communicate through a binding component and conversely.

[1] http://open-esb.net/index.php?option=com_content&view=article&id=86&Itemid=480

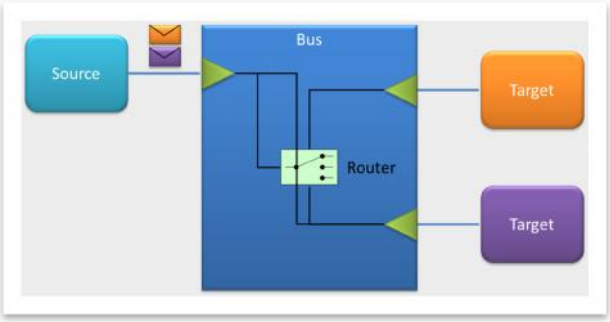


In the example above,

Consumer	Provider	Invocation path
Service A	Service B	Bus only
Service B	Service A	Bus only
Service A or B	Service C or D	Bus + Binding component
Service C or D	Service A or B	Binding Component + bus
Service C	Service D	BC + Bus + BC
Service D	Service C	BC + Bus + BC

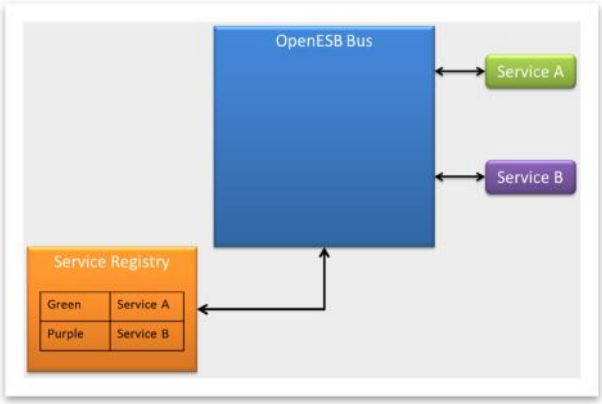
2.1 Service repository (difference between OpenESB and its competitors)

With integration tools such as Mule, Camel or WSO2, users define routes for messages statically or dynamically at the design time. Thanks to components such as a router or a filter, a message can use many routes to be sent to a partner. Whatever the route the message follows, it is not concerned about the partner standing at the end of the route. Here, the main effort is to define routes between partners.



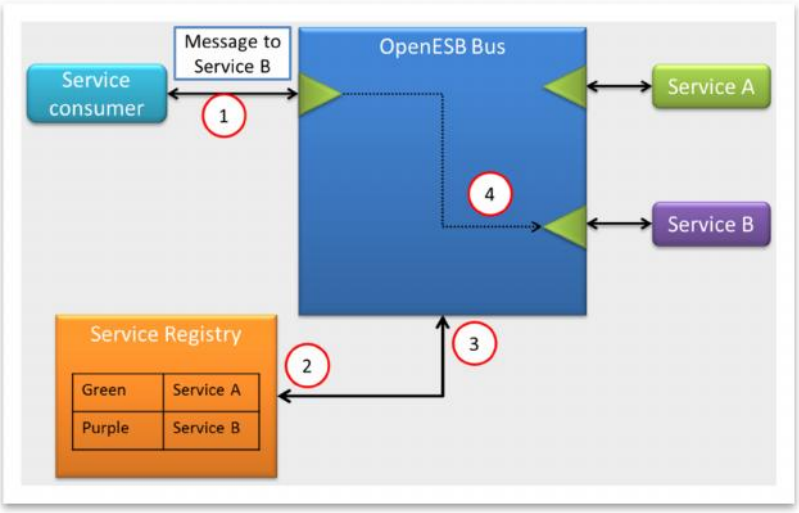
-

OpenESB design is completely different. OpenESB is an intelligent bus which does not rely on routes to address messages to a partner. It relies on the concept of service to send messages to the accurate partners.



How does OpenESB work?

OpenESB bus is associated with an internal service registry. When a component is deployed on the bus, the association between a component and the services it implements is stored in a service registry. In our case, the green component provides the service A and the purple component provides the service B.



At runtime:

1- A service consumer invokes “Service B” through a message.
2- OpenESB bus checks into the service registry which component provides “Service B”
3- The research returns that the purple component provides service B
4- OpenESB sends the message to the purple component

So when we invoke a service with OpenESB, we neither put a message on a predefined route nor indicate the component where we want to send the message but we declare the service we want to invoke.
Now we know enough to learn more on composite applications

3 Composite Application: “What is it?”

3.1 Composite Application definition

A composite application definition is:
A SOA composite is an assembly of services, service components, and references designed and deployed together in a single application. Wiring between the service, service component, and reference enable message communication. The composite processes the information described in the messages. OpenESB specifications describe a service assembly as a collection of deployment artefacts and associated metadata.

What does it mean?

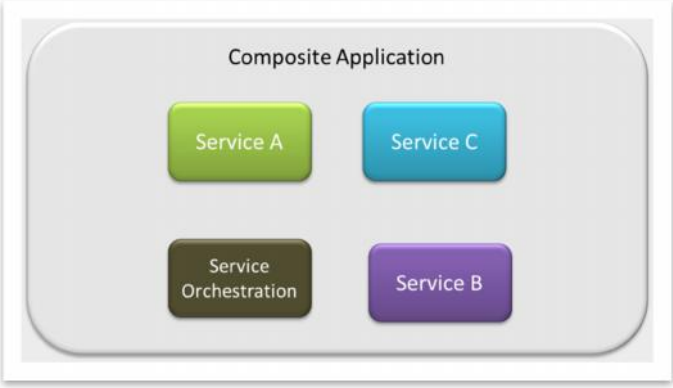
It is a good practice to design component as reusable, autonomous and stateless. Service granularity must be coarse enough to have a business meaning, but also fine enough to be reusable. At the design time, we don’t know in which context our components will run and since they are reusable, they must work in many environments without impacting. In order to increase their reusability, it is a good practice to postpone environment’s parameters setup and not defining any parameters linked to the runtime context during the design time. For this reason OpenESB creates an additional step in its development process and as much as possible, runtime parameters must be defined at the CASA stage.
In this way, a composite application is the place where links between services or services and environments are defined. At this stage, project developers fix the concrete service implementations they want to address and the protocol they want to use.

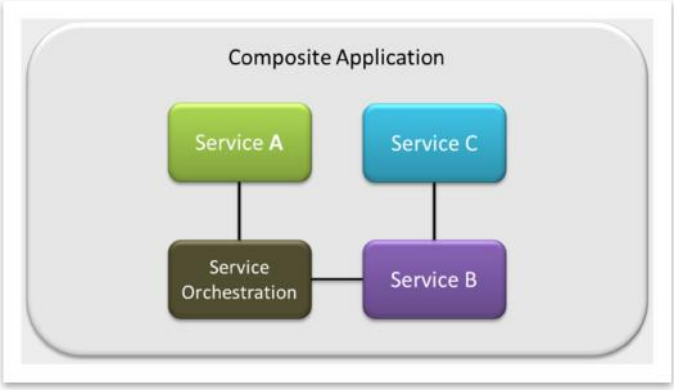
3.2 CASA examples

In this chapter, we list the different types of links between components we can define in a Service Assembly.

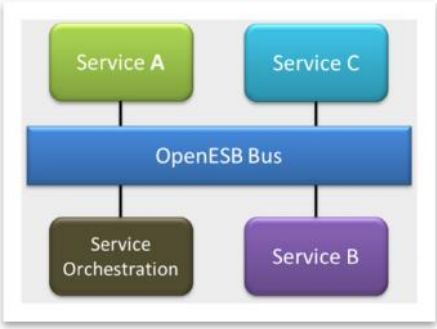
3.2.1 Simple CASA

To go forward with Composite Application, let’s design a composite application with three services A, B, and C plus an orchestrator.



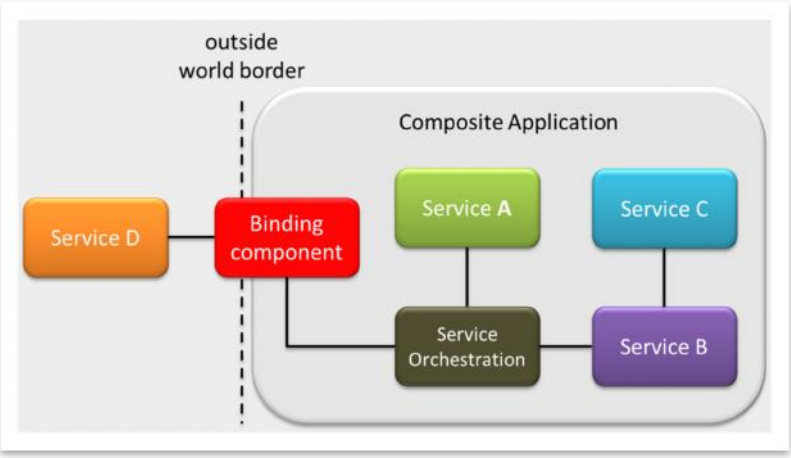


Composite applications embedded components and the links between components. In this case, since the components are deployed in the same OpenESB instance, they belong to the same environment and communicate through the bus.

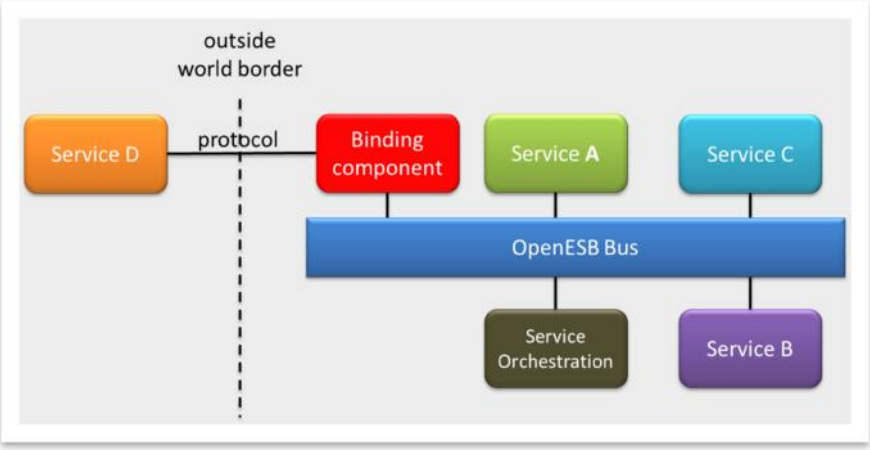


3.2.2 CASA with External Service

Let's suppose the service orchestration needs to access to an external service. As explained above, a binding component will be used to access the outside world.

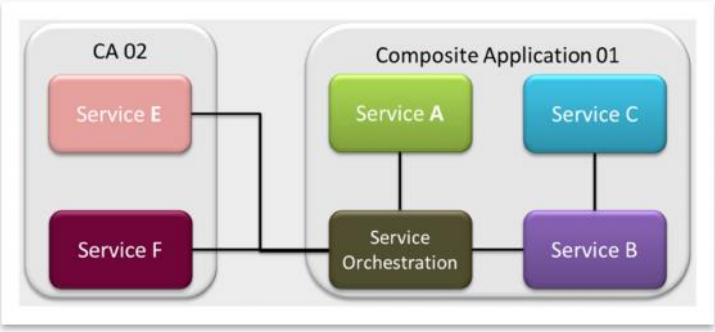


At the bus level, components are connected as follows:

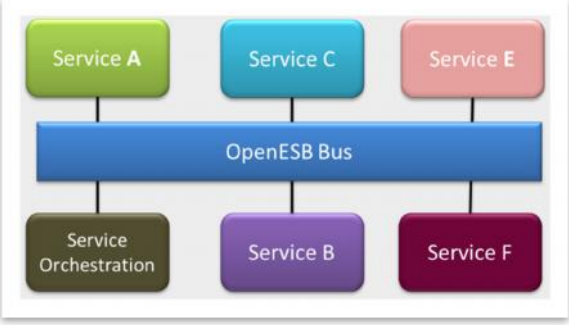


3.2.3 Two CASAs

We define an additional Service Assembly, the service orchestration needs to invoke the services embedded in the second service assembly



At the bus level components are deployed as follows:



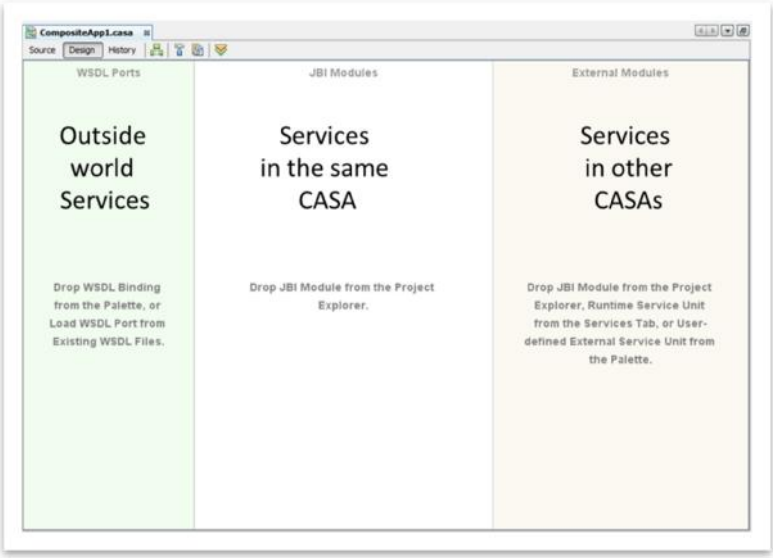
Component in the composite application 01 and 02 are deployed on the same OpenESB instance, so they belong to the same environment. They communicate together through the bus and there is no difference if you invoke a service between invoking services from CA01 or CA02.

3.3 Different invocations between services in CASA Editor

In our example below, we found three invocation types.

1- Invoking a service embedded in the same composite application
2- Invoking a service embedded in another composite application
3- Invoking a service in the outside world

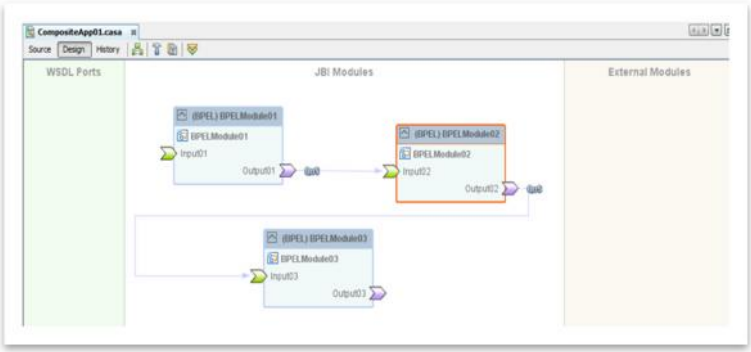
CASA editor design comes from these three types of invocations.



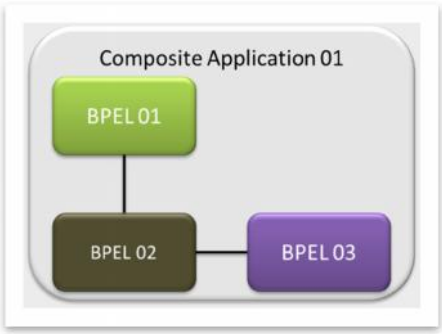
1- WSDL Ports lane defines services from the outside world. We use binding components to invoke them.
2- JBI Modules lane defines service embedded in the current composite application.
3- External Modules lane defines services embedded in other composite applications.

To illustrate the invocation types, let's give some example with the CASA Editor.

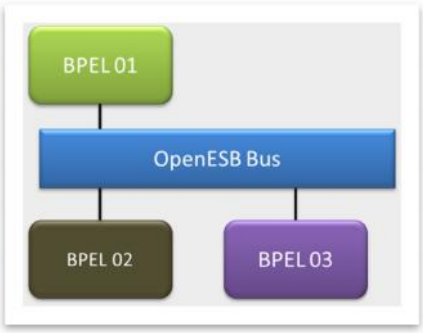
3.3.1 Services in the same Service Assembly



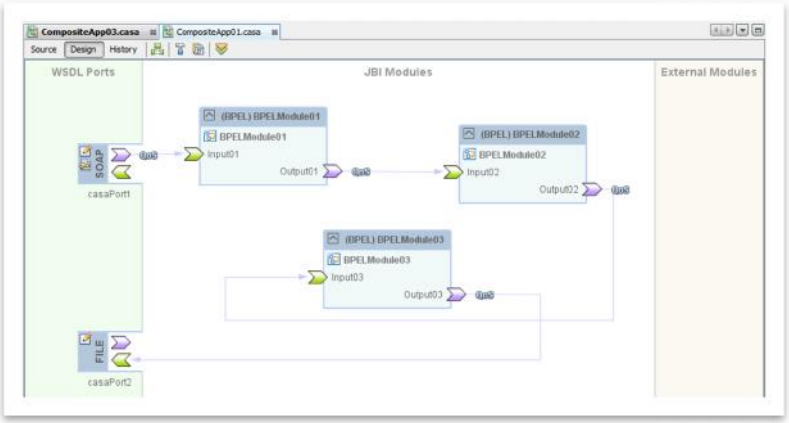
Three modules BPEL01, BPEL02 and BPEL03 are defined in the Editor. They are all defined as JBI Module. So they belong to the same CASA. At component level the CASA is as follows:



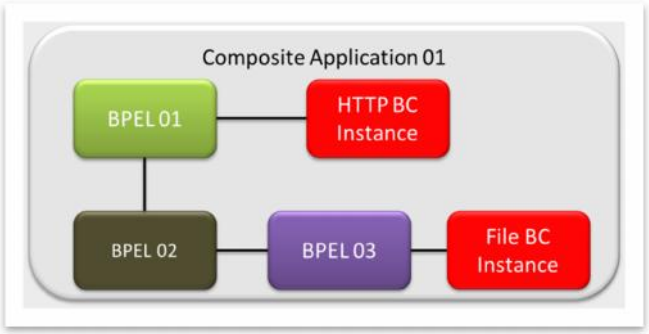
Deployment on the bus can be seen as follows:

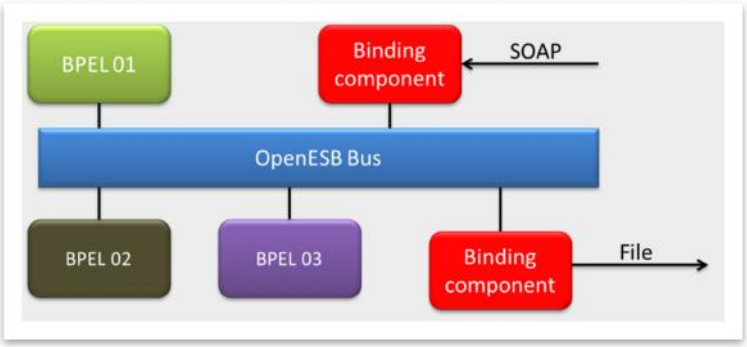


3.3.2 Services in the same Service Assembly + Binging components

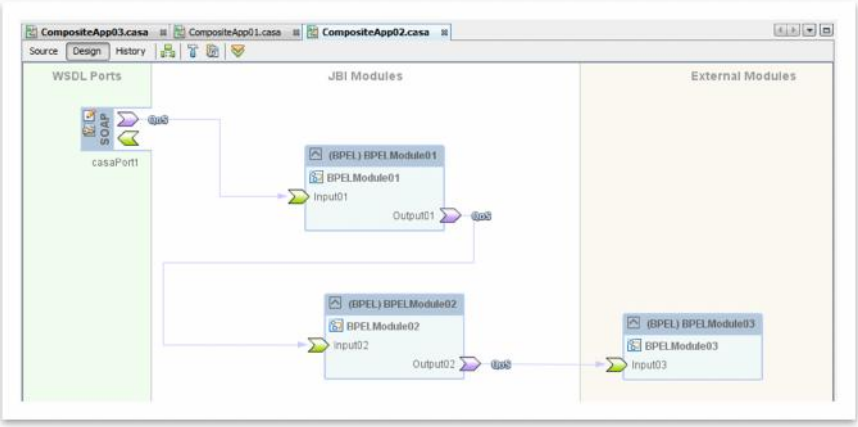


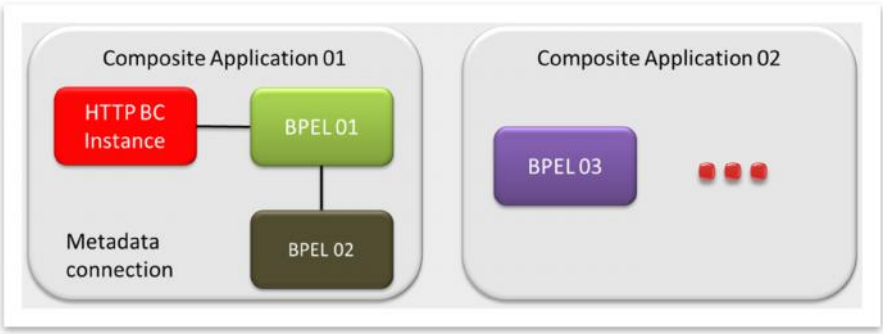
Binding components have been added to communicate with external services. Binding components can be seen as internal components in the service assembly.

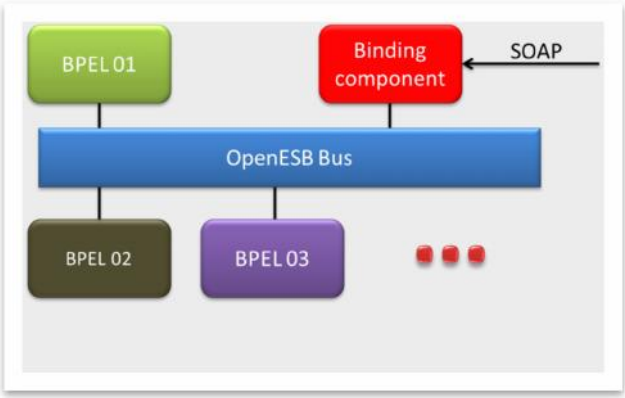




3.3.3 Service in two Service Assemblies + Binding Component



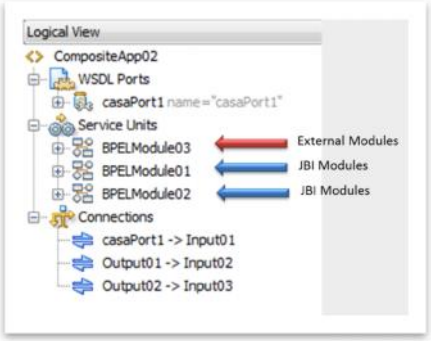




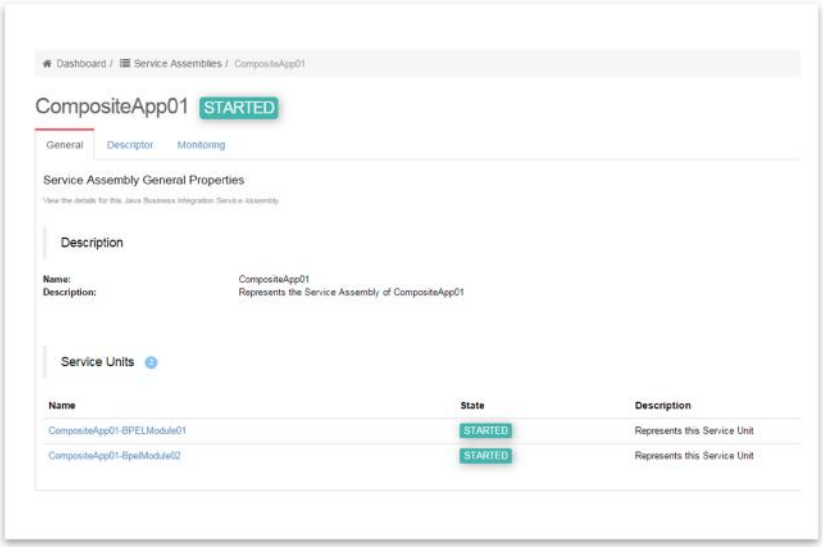
In our example, three modules are displayed in CASA editor: BPEL01, BPEL02 and BPEL03. BPEL01 and BPEL02 modules are defined as JBI Module, so they belong to the same Service Asembly. BPEL03 is seen as an External Module, it does not belong to the current Service Assembly, but to an external CASA (to be defined or an existing one).

In the OpenESB Studio, to set up a module as an external module, just drag it and drop it on the External Module lane.

Then, if you have a look at the composite application project, you can see that BPEL03 project is embedded in CASA.



Deploy the CASA and let's have a look at the OpenESB web console:



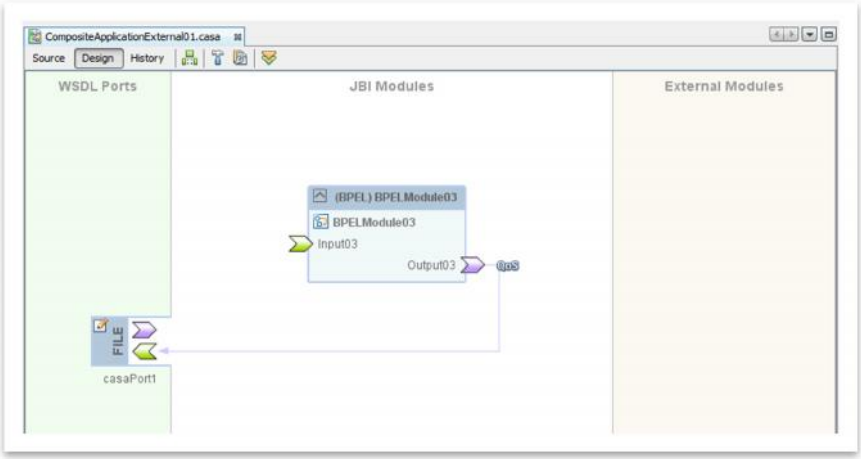
We notice that BPEL03 is not a Service Unit of our Composite Application. So we confirm that BPEL03 is independent of the Service Assembly even if BPEL03.jar is embedded with the service assembly. BPEL03.jar is just here as a reference and not as a service unit. So there is no link between BPEL03 and the composite application. It is important to understand that a link exists between the Service Assembly and the interface (or WSDL) implemented by BPEL03 only.

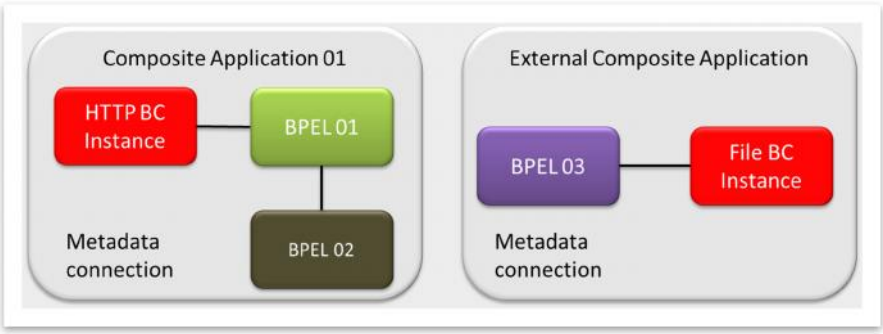
Let's have a look at Service Assembly JBI.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi"
xmlns:ns1="http://enterprise.netbeans.org/bpel/BPELModule02/BPELModule02"
xmlns:ns2="http://enterprise.netbeans.org/bpel/BPELModule03/BPELModule03" xmlns:ns3="CompositeApp02"
xmlns:ns4="http://enterprise.netbeans.org/bpel/BPELModule01/BPELModule01"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/jbi ./jbi.xsd">
  <service-assembly>
    <identification>
      <name>CompositeApp02</name>
      <description>Represents the Service Assembly of CompositeApp02</description>
    </identification>
    <service-unit>
      <identification>
        <name>CompositeApp02-BPELModule01</name>
        <description>Represents this Service Unit</description>
      </identification>
      <target>
        <artifacts-zip>BPELModule01.jar</artifacts-zip>
        <component-name>sun-bpel-engine</component-name>
```

```
        </target>
      </service-unit>
      <service-unit>
        <identification>
          <name>CompositeApp02-BPELModule02</name>
          <description>Represents this Service Unit</description>
        </identification>
        <target>
          <artifacts-zip>BPELModule02.jar</artifacts-zip>
          <component-name>sun-bpel-engine</component-name>
        </target>
      </service-unit>
      <service-unit>
        <identification>
          <name>CompositeApp02-sun-http-binding</name>
          <description>Represents this Service Unit</description>
        </identification>
        <target>
          <artifacts-zip>sun-http-binding.jar</artifacts-zip>
          <component-name>sun-http-binding</component-name>
        </target>
      </service-unit>
      <connections>
        <connection>
          <consumer endpoint-name="Ouput02PortTypeRole_partnerRole" service-name="ns1:Output02"/>
          <provider endpoint-name="Ouput02PortTypeRole_myRole" service-name="ns2:Input03"/>
        </connection>
        <connection>
          <consumer endpoint-name="casaPort1" service-name="ns3:CompositeApp02Service1"/>
          <provider endpoint-name="Input01PortTypeRole_myRole" service-name="ns4:Input01"/>
        </connection>
        <connection>
          <consumer endpoint-name="output01PortTypeRole_partnerRole" service-name="ns4:Output01"/>
          <provider endpoint-name="output01PortTypeRole_myRole" service-name="ns1:Input02"/>
        </connection>
      </connections>
    </service-assembly>
</jbi>
```

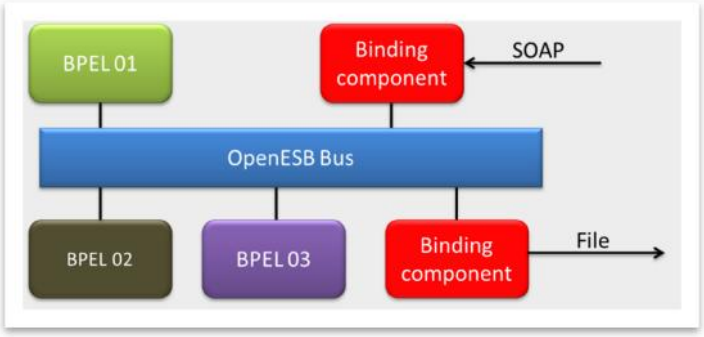
JBI.xml shows that two BPEL modules and one instance of HTTP BC are embedded in the Services Assembly.
There is no reference to BPEL03 but just a reference to the service implemented by BPEL03.
To complete the project, we need to create and deploy a new composite application which embedded BPEL03.





Link between BPEL02 and BPEL03 will be created by the bus based on metadata connection defined in the composite applications.

Components will be deployed on the bus as follows:



4 Conclusion

Service composition is the less known step in the development process. There are few documents written on this topic and few users really understand the power of this OpenESB feature. CASA Editor appeared as an unclear mandatory step to deploy applications. Unfortunately, OpenESB tutorials (except Pymma tutorials) tend to keep project developments simple, straightforward and avoid talking about Service Assembly Design and CASA configuration. I hope this document fills the gap.

Service Assembly is one of the key features of OpenESB technology; it provides an additional degree of flexibility in SOA development. CASA concepts are powerful and provide good isolation between components, allow safe concurrent development and promote reusing and flexibility. CASA concepts defined in JBI specifications are reused by SCA specifications and many other tools.

Today (2015 Q3), OpenESB Studio does not use the entire flexibility offered by OpenESB Bus and some remaining features must be developed to take into account its incredible flexibility. We plan to improve CASA flexibility in the next version of OpenESB and make OpenESB more and more powerful.

5 Appendix A: External module as module Proxy

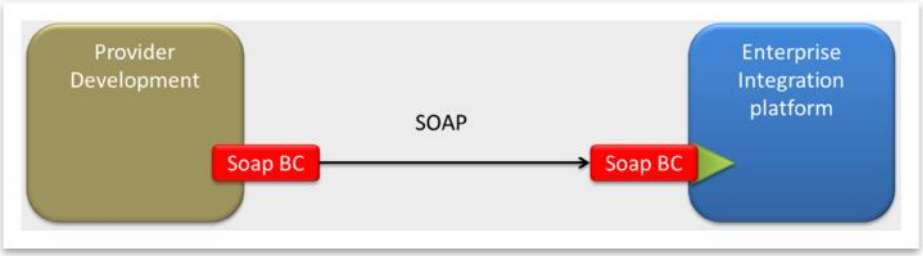
In this appendix, we would like to show how using external service features provide flexibility in your development.

User case:

- On its Enterprise integration platform, a company developed a business service named “Validation Service” and publishes it as a web service for external service consumers.

- After a few months, the business team defined a new business process where “ValidationService” will be used. A provider has been chosen to implement this new business process. Developed off site, the new application will be tested and validated off site and then deployed on site. It will collocate with the “Validation Service” on the Enterprise Integration Platform.

- Validation service is a complex service and requires additional features and confidential data to validate data and for security reasons, cannot be relocated to the partner environment. Then to use it, the provider’s team developed the new application and invokes “Validation Service” through a SOAP call.

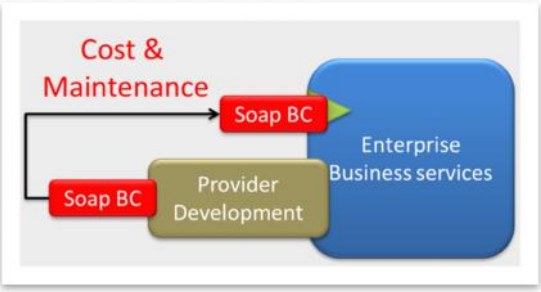


At the beginning of the project, the development team used the WSDL document published by the company and got it a from a browser call (http://myServer:myPort/myService?wsdl). By definition, the WSDL defined a SOAP binding access to “Validation Services”.

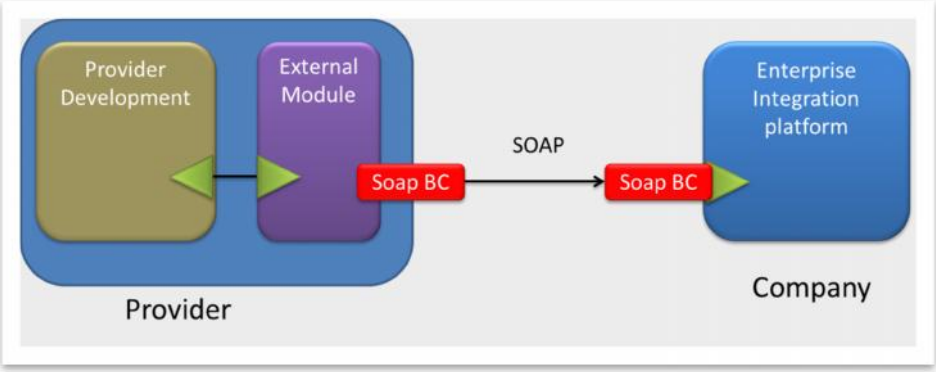
The development team used the WSDL in their OpenESB module and unfortunately defined runtime parameters during the development time. Once these environment parameters defined at the design time, it is not possible to use CASA flexibility and adapt the project to the runtime environment.

In that case, SOAP will remain the communication channel to “Validation Service” whatever the environment since the protocol has been defined at the development time. However, it has been decided that the new project and ‘Validation Service” will collocate in the same environment, consequently as explained above, they don’t require any additional communication channel such as SOAP to communicate together.

So when the new project is integrated into the company’s platform, the SOAP link remains and overloads the platform management and maintenance.

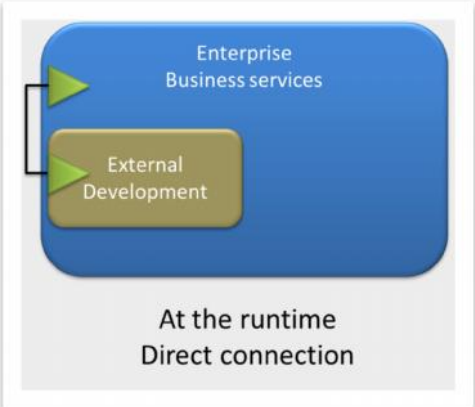


So to avoid this inaccurate design, the development team can use an additional module to solve this issue and give flexibility to the development.



In this new design, provider's development invokes "Validation Service" through the new module in purple. This purple module exposes the same interface "Validation Service" to the project and act as a proxy.

When the development is tested and validated, CASA must be upgraded to define a link between the development and "Validation Service". Then the new module will invoke "Validation Service" directly and not through the proxy module.



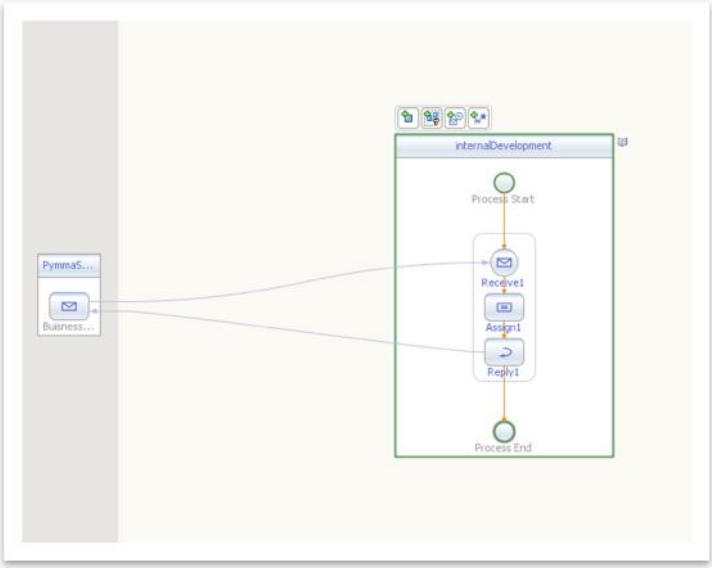
There is not difference for the provider's development to invoke the external module or the Enterprise Service. The only differences between both cases are the metadata defined in the CASA editor and stored in the Service Assembly's jbi.xml. This upgrade is very easy to do and leaves the applications unchanged. It can be made either by the provider or the company. So we solve the issue found and SOAP channels are not required anymore.
In the same way, by using mock services, many teams can work in parallel mode on concurrent projects.

5.1 Implementation

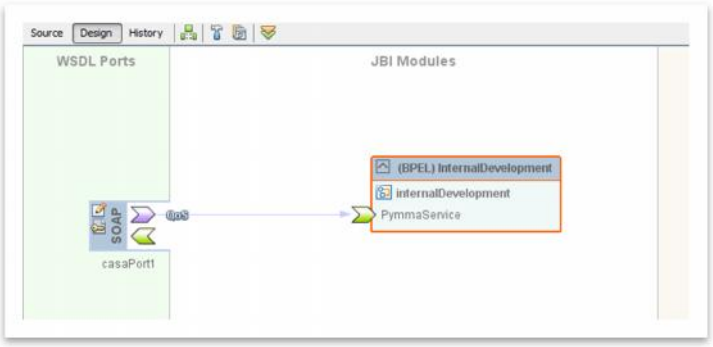
Let see in what way we implement this example with OpenESB Studio.

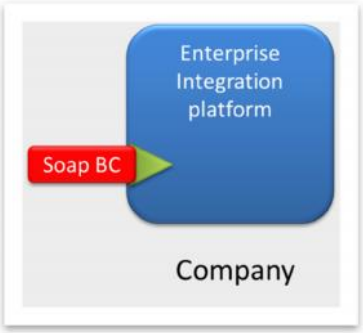
5.1.1 Step 01: Internal Module

The internal module is the "Validation Service" in our example. Here is it a simple "Hello World" BPEL service named internal development and deployed on the Integration platform



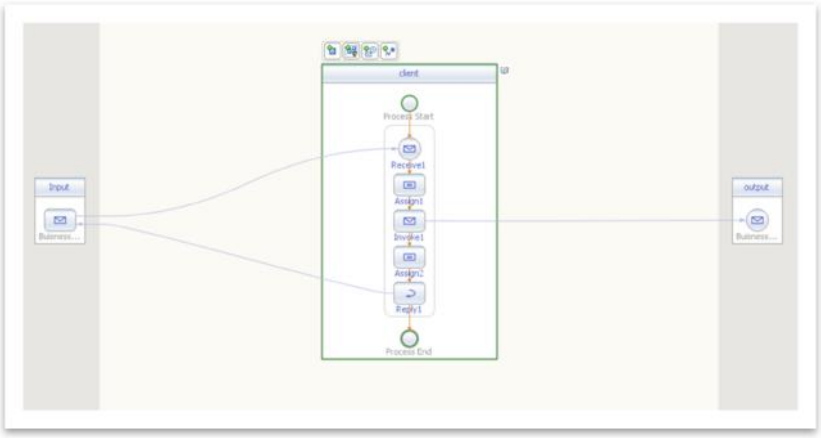
We add a SOAP binding component in the CASA to allow an external access to the service.



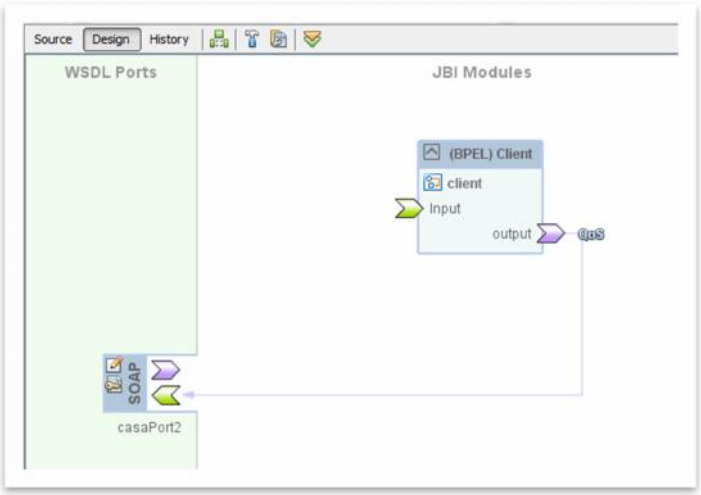


5.1.2 Step 02: Client Module

Client module is the intermediate between the internal service and the new project developed by the partner. It is up to the partner to create a proxy to access to the Enterprise Services.



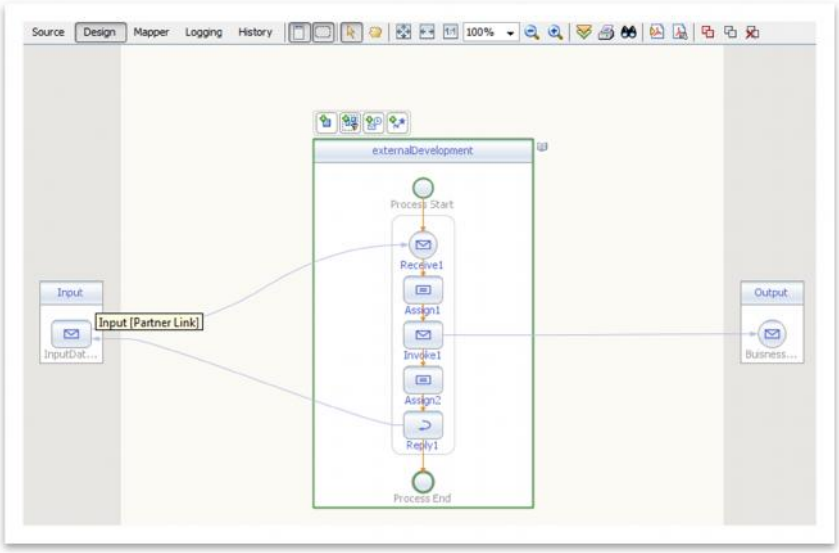
The proxy is a simple BPEL project that invokes the Enterprise Service. Notice that Inbound and outbound WSDLs use the same message type and the same port type.



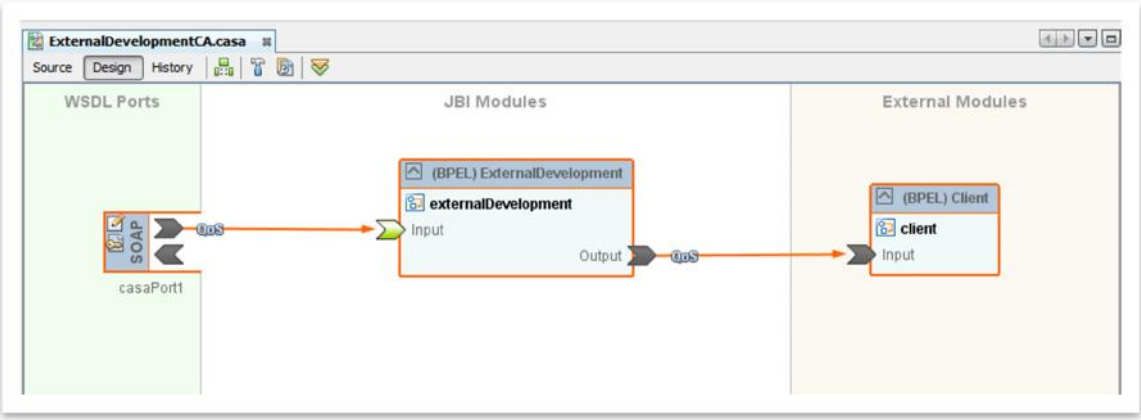
Proxy CASA uses an outbound SOAP Binding Component to invoke the Enterprise Service. There is no need for an inbound Binding Component.

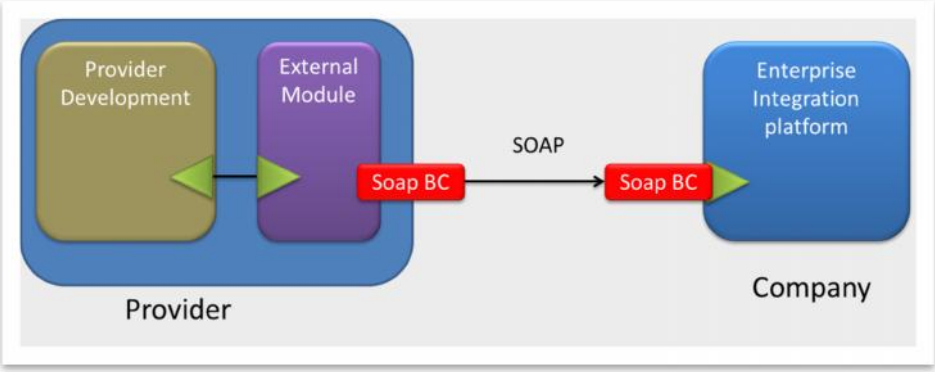
5.1.3 Step 03: New project development and test

The partner develops an application to be deployed on the Enterprise Platform. He uses “Validation Service” WSDL as outbound WSDL.



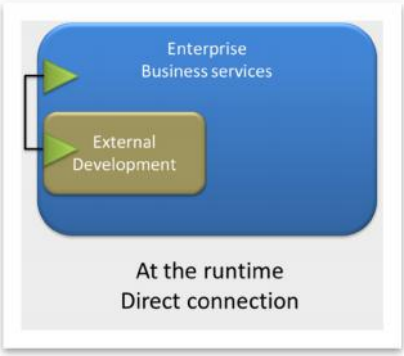
At the CASA level, the partner’s module invokes the Enterprise Service through the Proxy. Please note that Client project (the proxy) is defined as an external module, so it does not belong to the current Service Assembly.





5.1.4 Step 04: deployment on the platform

When the partner module is validated and tested, the client module is replaced by the internal module that is in the Service Assembly. Then the CASA can be deployed on the enterprise platform.



Environment changes between the development and production environment did not impact the new project itself, but requires a minor modification at the CASA level to run perfectly.