Article
Istvan Hahn · May 16, 2016 11m read

# Accelerate Ensemble

*Introducing non-persistent messages. eXpert-to-eXpert*

### Background

InterSystems Ensemble as a tool does a lot for the Developer. One of the nice features is the Message trace utility. It shows a message flow diagram. The diagram shows the progress of the message processing real time. You can get many-many useful information from the production. In any case, someone needs to find a bug in a production implementation, without the Message trace utility it could turn into a real nightmare.

On the other hand, keeping message " traceability " is not for free. A heavy loaded production can very quickly run out of resources just because of the house keeping functions of Ensemble. House keeping functions such as maintaining message header, log entries, message queue generates a significant load on the Caché database used by Ensemble.

This article is about to show how to force Ensemble work more for the everyday life, instead of being prepared for " any-time-debugging " .

This is an eXpert-to-eXpert article. Therefore, I assume the deep understanding of Ensemble.

### Introduction

Ensemble 2016.1 introduced a brand new configuration use case. It is when Ensemble is used as an ESB. It is a distinct scenario. Our documentation gives examples how an ESB can be configured. What makes it so special? ESB type configuration is focusing on high-speed message delivery with minimal (or no) routing and/or transformation.

Technically speaking the key enabling the ESB type high performance messaging is that several components are prepared for " in memory " messaging. The $$$EnsInprocPerist macro is the " public interface " to control the components. This article is all about to show that custom written services and operations can also take advantage of that new feature. As you might expect nothing is for free. Please look at the " Classic vs InMemory " section at the end of the article to see the price.

As an addition, a set of pass-through services and operations are introduced. Pass-through operations are invoked InProc. Since the InProc invocation translates to a procedure call, the request and response structures (messages) are exchanged in memory. To maximize the throughput in such cases there is no need to persist the messages. Besides not persisting messages no queue is maintained, no header is stored.

In the following example, I will show how can you extend the ESBish model to other services /operations. The key new feature demonstrated by this article is squeezed into a single macro $$$EnsInProcPersist. Using this macro in a service will force all InProc Business Operations called by the particular service not to save neither messages nor message headers. The macro itself gets/sets an Ensemble runtime parameter stored in the %Ensemble local array.

Let me reiterate: the only thing this article wants explain is that if you know why you need in-memory messaging, I can teach you how to take it on. Also, there is nothing new in the source code you can not find in the class library in a second or two. If you take your time… Those are higher quality codes right from the dedicated developers. But in the meantime, you have to live with my examples in the context of the article.

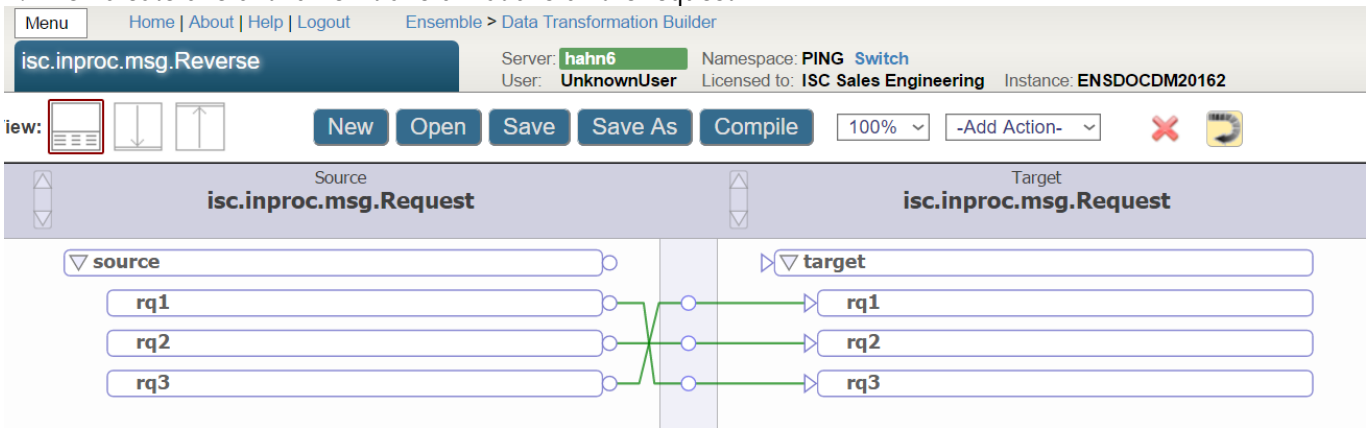
### Preparation

Firstly we need to create a traditional or classic production. By classic I mean production which is developed in the way you learned from your first Ensemble tutor. Please take your time and create the components like I do. You are free to use other naming convention, programming style. It does not matter. The important is to have the components.

1. Create a pair of request /response message classes.

```
Class isc.inproc.msg.Request Extends (%Persistent, %XML.Adaptor)
{
 Property rq1 As %String(MAXLEN = "");
 Property rq2 As %String(MAXLEN = "");
 Property rq3 As %String(MAXLEN = "");
}
Class isc.inproc.msg.Response Extends (%Persistent, %XML.Adaptor)
{
 Property rp1 As %String;
 Property rp2 As %String;
}
```

2. Then create one or two new transformations on the request




3. Here comes the service. It is going to be a SOAP service.

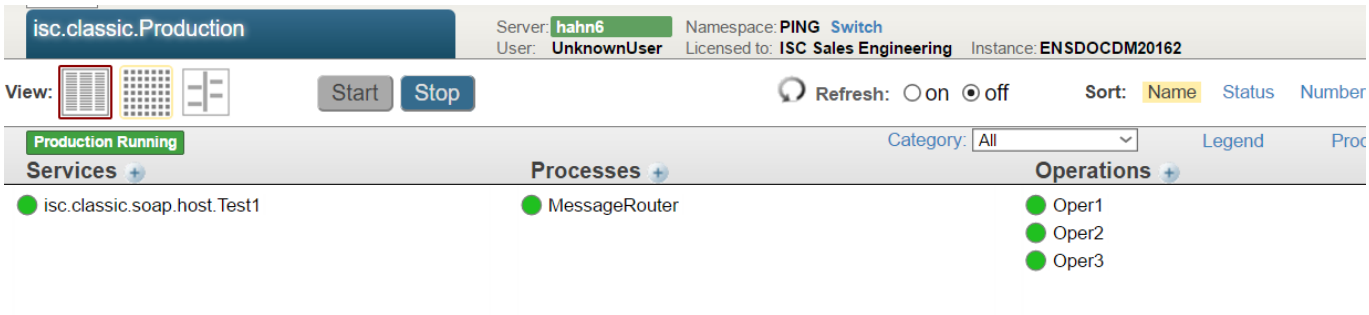
```
/// isc.classic.soap.host.Test1
Class isc.classic.soap.host.Test1 Extends EnsLib.SOAP.Service [ ProcedureBlock ]
{
Parameter ADAPTER;
/// Name of the WebService.
Parameter SERVICENAME = "ClassicSOAPTest1";

/// TODO: change this to actual SOAP namespace.
/// SOAP Namespace for the WebService
Parameter NAMESPACE = "http://tempuri.org";

/// Namespaces of referenced classes will be used in the WSDL.
Parameter USECLASSNAMESPACES = 1;
Property TargetConfigName As Ens.DataType.ConfigName;

Parameter SETTINGS = "TargetConfigName:Additional";

/// TODO: add arguments and implementation.
```

```
/// InProc
Method InProc(docin As isc.inproc.msg.Request) As isc.inproc.msg.Response [ WebMethod
 ]
 {
  set st=..SendRequestSync(..TargetConfigName,docin,.retval)
  Quit retval
 }
}
```

4. Next is the Business Operation which is a file operation. Do not use the message map XData block. It is by purpose

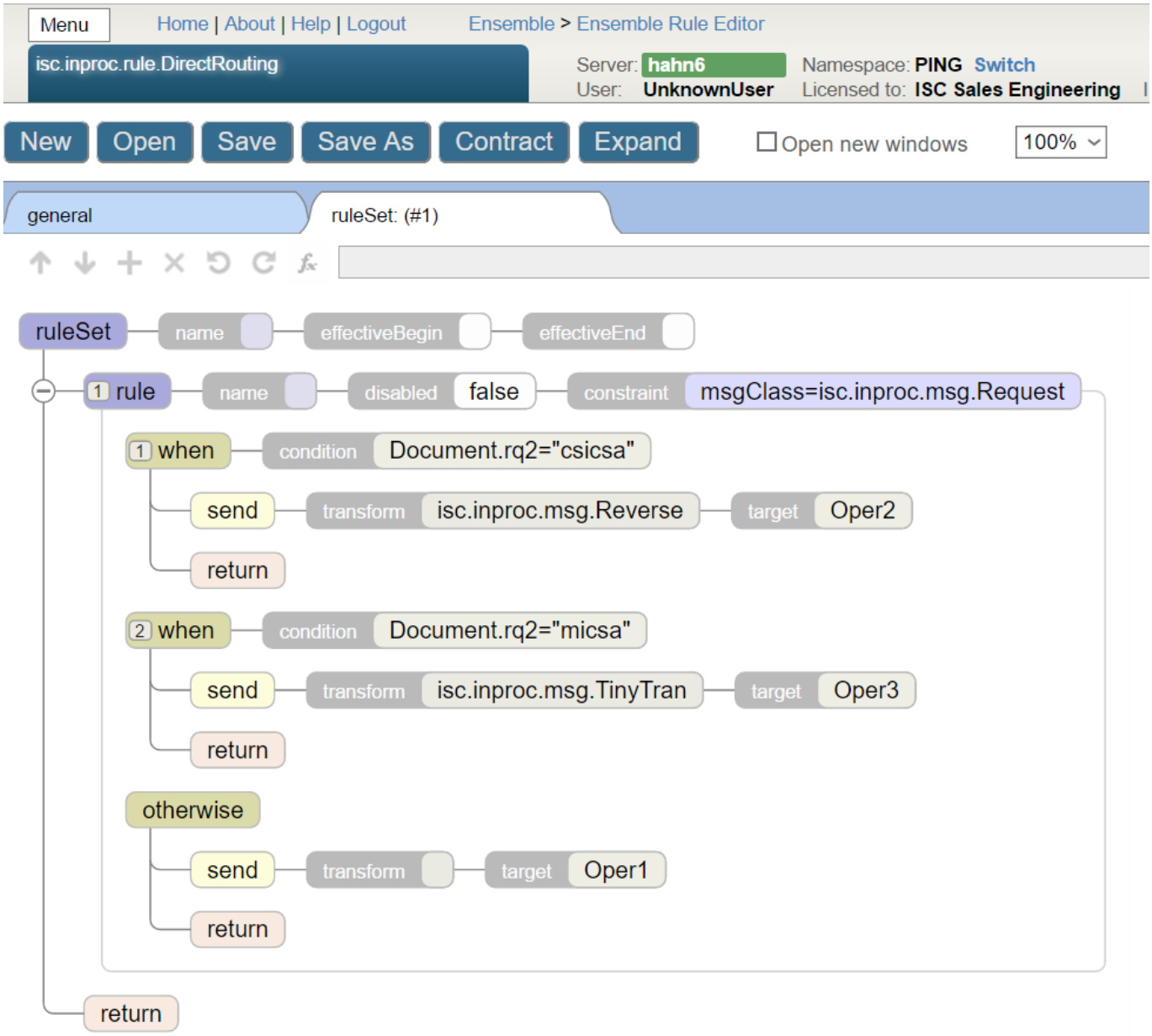
```
Class isc.classic.host.Operation Extends Ens.BusinessOperation
{
Parameter ADAPTER = "EnsLib.File.OutboundAdapter";
Parameter INVOCATION = "Queue";
Property Adapter As EnsLib.File.OutboundAdapter;

Method doit(pRequest As isc.inproc.msg.Request, Output pResponse As isc.inproc.msg.Re
sponse) As %Status
 {
  set filename = ..%ConfigName_".LOG"
  do ..Adapter.PutLine(filename,"============ NEW REQUEST ================")
  do ..Adapter.PutLine(filename,pRequest.rq1)
  do ..Adapter.PutLine(filename,pRequest.rq2)
  do ..Adapter.PutLine(filename,pRequest.rq3)
  set pResponse = ##class(isc.inproc.msg.Response).%New()
  set pResponse.rp1 = ..%ConfigName
  set pResponse.rp2 = $zdt($ztimestamp,3)

  Quit $$$OK
}

/// This is the default message handler. All request types not declared in the messag
e map are delivered here
Method OnMessage(pRequest As %Library.Persistent, Output pResponse As %Library.Persis
tent) As %Status
 {
  If '$G($$$EnsInProcPersist,1) Do ##class(Ens.Util.Statistics).InitStats(..%ConfigNa
me)

  if pRequest.%Extends("isc.inproc.msg.Request") { set st=..doit(pRequest,.pResponse)
 }
  else {set st = $$$ERROR(5001)}

  If '$G($$$EnsInProcPersist,1) && ##class(Ens.Util.Statistics).StatsStarted(..%Confi
gName)
   {
    Do ##class(Ens.Util.Statistics).RecordStats($$$eHostTypeOperation,..%ConfigName)
   }
  quit st
 }
}
```

5. At this moment we are almost done. We need a production and some adjustment on the item settings




Oper1, Oper2, Oper3 are instances of the Business Operation we just created in step 4. MessageRouter is an instance of EnsLib.MsgRouting.RouterEngine. Settings you need to review: for Oper1, Oper2, Oper3 the file path to point to a server file system directory; for MessageRouter the Response From setting must be *.

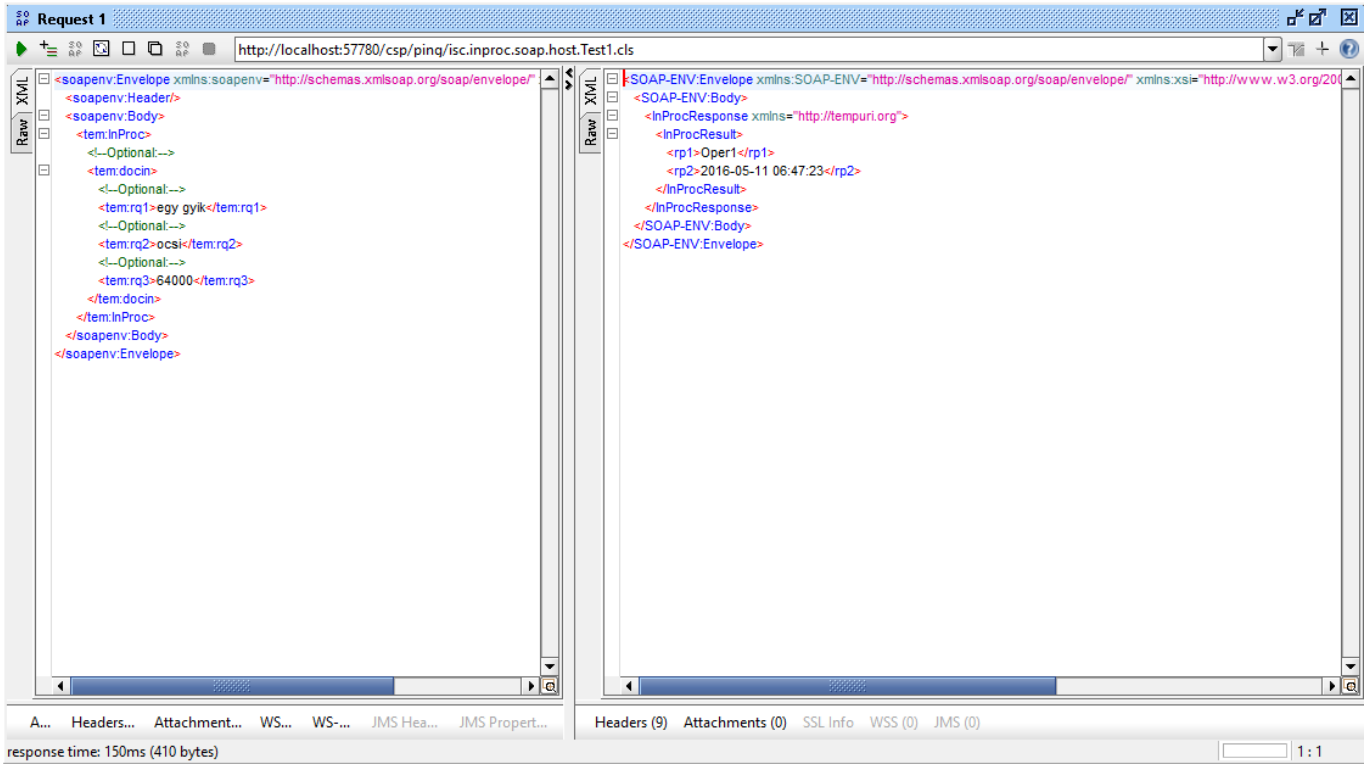
6. Lastly, we need the rule to control the message routing




Once the rule has been created, assign the name of the rule to the MessageRouter Business Rule Name setting.

What the Production does: takes an input of three strings as a SOAP service, does some routing based on the contents and writes the transformation result to a file. The returned response is: which of the three operations were writing the output file, and when. Like a Commit ACK. No response is expected from the application.

Let us test it with SOAP UI. You should see pretty much like this:




### Turn to “ ESBish ”

In fact, there is not much to work on it.

1. Set the invocation parameter of the operation to InProc I made a copy of the original operation, but it is not required:

```
Class isc.classic.host.Operation Extends Ens.BusinessOperation
{
Parameter ADAPTER = "EnsLib.File.OutboundAdapter";
Parameter INVOCATION = "InProc";
Property Adapter As EnsLib.File.OutboundAdapter;

Method doit(pRequest As isc.inproc.msg.Request, Output pResponse As isc.inproc.msg.Re
sponse) As %Status
 {
  set filename = ..%ConfigName_".LOG"
  do ..Adapter.PutLine(filename,"============ NEW REQUEST ================")
  do ..Adapter.PutLine(filename,pRequest.rq1)
  do ..Adapter.PutLine(filename,pRequest.rq2)
  do ..Adapter.PutLine(filename,pRequest.rq3)
```

```
  set pResponse = ##class(isc.inproc.msg.Response).%New()
  set pResponse.rp1 = ..%ConfigName
  set pResponse.rp2 = $zdt($ztimestamp,3)

  Quit $$$OK
 }

/// This is the default message handler. All request types not declared in the messag
e map are delivered here
Method OnMessage(pRequest As %Library.Persistent, Output pResponse As %Library.Persis
tent) As %Status
 {
  If '$G($$$EnsInProcPersist,1) Do ##class(Ens.Util.Statistics).InitStats(..%ConfigNa
me)

  if pRequest.%Extends("isc.inproc.msg.Request") { set st=..doit(pRequest,.pResponse)
 }
  else {set st = $$$ERROR(5001)}

  If '$G($$$EnsInProcPersist,1) && ##class(Ens.Util.Statistics).StatsStarted(..%Confi
gName)
   {
    Do ##class(Ens.Util.Statistics).RecordStats($$$eHostTypeOperation,..%ConfigName)
   }
  quit st
 }
}
```

Should look familiar. Only the INVOCATION= ” InProc ” makes the difference.
2. The service class now sub-class of isc.inproc.host.soap.Service

```
/// isc.inproc.soap.host.Test1
Class isc.inproc.soap.host.Test1 Extends isc.inproc.host.soap.Service [ ProcedureBloc
k ]
{
Parameter ADAPTER;

/// Name of the WebService.
Parameter SERVICENAME = "InProcSOAPTest1";

/// TODO: change this to actual SOAP namespace.
/// SOAP Namespace for the WebService
Parameter NAMESPACE = "http://tempuri.org";

/// Namespaces of referenced classes will be used in the WSDL.
Parameter USECLASSNAMESPACES = 1;

Property TargetConfigName As Ens.DataType.ConfigName;

Parameter SETTINGS = "TargetConfigName:Additional";

/// TODO: add arguments and implementation.
/// InProc
Method InProc(docin As isc.inproc.msg.Request) As isc.inproc.msg.Response [ WebMethod
 ]
```

```
 {
  set st=..SendRequestSync(..TargetConfigName,docin,.retval)
  Quit retval
 }
}
```

3. And where is this new service superclass? Here it is:

```
Class isc.inproc.host.soap.Service Extends EnsLib.SOAP.Service
{
Parameter SETTINGS = "PersistInProcData:Additional";

/// Persist data to operations with invocation InProc that are called Synchronously.<
br/>
/// The default is On. <br/>
/// This setting is only used if calling an operation with invocation InProc. <br/>
/// If this setting is off then no message headers will be created and message bodies
 will not be saved.<br/>
/// If this setting is off there will be no trace in the message viewer. <br/>
/// If this setting is off there will be no retry attempts by the operation - only on
e attempt will be made. <br/>
Property PersistInProcData As %Boolean [ InitialExpression = 0 ];

/// This user callback method is called via initConfig() from %OnNew() or in the case
 of CSP invoked services from OnPreSOAP()
Method OnInit() As %Status
 {
  Set $$$EnsInProcPersist=..PersistInProcData
  Quit ##super()
 }
}
```

The important is this magic macro $$$EnsInProcPersist.

4. The last step is to create a message router for InProc.

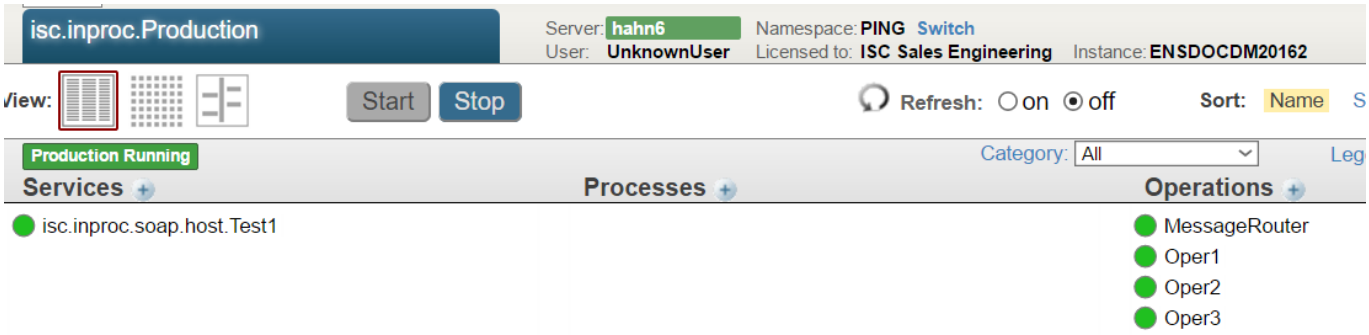
You need that guy only if you are about the rule-based routing of the messages. Unfortunately, no Business Process is prepared for InProc mode, and actually, it does not make any sense. So this example MessageRouter operation is an extract from the original router engine.

```
Class isc.inproc.host.MessageRouter Extends Ens.BusinessOperation
{
Parameter INVOCATION = "InProc";
Property RoutingRule As Ens.DataType.Class;
Parameter SETTINGS = "RoutingRule:Additional:ruleSelector";

/// This is the default message handler. All request types not declared in the messag
e map are
delivered here
Method OnMessage(pRequest As %Library.Persistent, Output pResponse As %Library.Persis
tent) As %Status
 {
```

```
  If '$G($$$EnsInProcPersist,1) Do ##class(Ens.Util.Statistics).InitStats(..%ConfigNa
me)
  set context = ##class(isc.inproc.rule.Context).%New()
  set context.%MessageReceived = pRequest
  set context.MsgClass = pRequest.%ClassName(1)
  set context.Document = pRequest
  set sc= $$$OK
  set sc = $classmethod(..RoutingRule, "evaluateRuleDefinition", context, .ruleSet, .
begin, .end, .retval, .reason, 0)
  if $$$ISOK(sc) {
   for i=1:1:$length(retval,";") {
    set oneaction = $piece(retval,";",i)
    set command = $piece(oneaction,":",1)
    if command = "send" {
     set target = $piece(oneaction,":",2), transform = $piece(oneaction,":",3)
     set tRequest = pRequest.%ConstructClone(1)
     for j=1:1:$length(transform,",") {
      set onetransform=$piece(transform,",",j)
      continue:onetransform=""
      set tRequest2=tRequest,sc=$classmethod(onetransform,"Transform",tRequest2,.tReq
uest)
     }
     for k=1:1:$length(target,",") {
      set onetarget=$piece(target,",",k)
      set sc = ..SendRequestSync(onetarget, tRequest, .pResponse)
     }
    }
   }
  }
  If '$G($$$EnsInProcPersist,1) && ##class(Ens.Util.Statistics).StatsStarted(..%Confi
gName)
   {
    Do ##class(Ens.Util.Statistics).RecordStats($$$eHostTypeOperation,..%ConfigName)
   }
  Quit sc
 }
}
```

This is a little more code than with the previous steps, but still manageable.

5. Build a new production, but use the new components. At the end it should look like this




**Remember:** MessageRouter is not a process anymore. It is an operation. The MessageRouter uses exactly the same rule as before.
Finally, we reached the point we lived for. The service has a setting Persist Messages Sent InProc. Please keep

this switch on until you are testing the production.

6. Create a new test in SOAP UI.

7. Once your production is working, change the Persist Messages Sent InProc to false. And test again using SOAP UI. When you test a production, please look at the left-bottom corner of your SOAP UI window and read the response time. Please remember those numbers…

### Classic vs. InMemory

We are done. Please compare the results.

| | Classic | InMemory |
|---|---|---|
| Working model | Sync/Async and Queue/InProc is supported | Only Sync, InProc is supported |
| Can run a BPEL business process? | Yes | No |
| Can do any parallel processing? | Yes | No |
| Who does the error handling? | Container | Production |
| Minimize Ensemble Overhead | No | Yes |

### Summary

The feature summary shows that there is “ only ” one use case for InMemory configurations. When the performance counts AND there is no significant data processing attached. It is typically the case for ESB kind of productions.

I reiterate the key characteristics of such a production.

- High volume of messages
- Transformations are providing sufficient functionality for message processing
- Synchronous message processing provides sufficient capacity
- When “ Commit ACK ” mode is enough to ensure the “ guaranteed message delivery ” . Or “ from the other corner of the ring ” .

Do not use InMemory if:

- The production uses true BPEL business processes
- High degree of parallelization is used in the production
- When you prepare your first live presentation with Ensemble

The good news is that you can turn a classic Ensemble production into an InMemory one with minimal efforts (as the example proofs).

Any indication of the capacity growth by employing the InMemory model? Not from me. In this particular demo you can experience on your own by running SOAP UI tests. Although on my laptop the capacity doubled without having the “ magic IO peeks ” , I perfectly know that it can not be simply projected to a random production. “ Magic IO peeks ” ? Come on, do not eat the birthday cake at once! MIOP is a topic for another community article.

#Ensemble