Java CAPS 6/JBI and OpenESB
Using JBI, Note 1

Basic "File to BPEL 2.0 Process to File" Project

Michael Czapski, June 2008

1 Introduction

As implemented in OpenESB and Java CAPS 6/JBI, JBI at first put me off completely. I had no idea where to start and what to do. This was particularly annoying as I was reasonably effective in developing solutions using Java CAPS 5.x and did not look forward to trying to figure out how to do in OpenESB the same thing I already new how to do.

This document is intended to save you the anxiety and help you get over the initial hurdles. It walks through the process of creation, deployment and execution of a simple File-to-File integration solution, with detailed step-by-step illustrations. The focus is the practice of using JBI components not the theory of JBI.

This document addresses the integration solution developers, not developers of Service Engines or Binding Components.

The projects use JBI components only, that's why they are just as good for OpenESB exploration as they are for Java CAPS 6/JBI exploration.

JBI (Java Business Integration) is not discussed to any great extent. JBI artifact names are used in discussion but not elaborated upon. Explanations are provided where necessary to foster understanding of the mechanics of developing integration solutions using JBI technologies in OpenESB and Java CAPS 6/JBI.

Java CAPS 6 and OpenESB are two of a number of toolkits that implement the JBI specification (JSR 208). When I use an expression like "In JBI …" I actually mean "In JBI as implemented in Java CAPS 6 and OpenESB …". The same things may well be implemented differently in other JBI toolkits.

Java CAPS 6 "Revenue Release" is used and shown in illustrations. OpenESB can be used instead however the appearance of components shown in illustrations may vary somewhat.

2 WSDLs

What initially put me off the most was the explicit use of WSDL to define message structures and interactions between Binding Components (what in 5.x one would call eWays or Adapters) and Service Units (what one would call Java Collaborations, eInsight Business Processes and suchlike in 5.x). In 5.x WSDL was used for the same things but, unless one wanted to expose an eInsight Business Process as a web service, or consume a web service described by a WSDL, WSDL definitions were effectively invisible to the developer. This made the 5.x toolkit appear simpler to use, and conversely, made OpenESB and Java CAPS 6/JBI appear more complex and

appear to require much deeper technical knowledge to work with as compared to Java CAPS 5.x.

In JBI WSDL is used to provide definition of payload message structures that are exchanged between components and, in case of Binding Components, to provide the means to configure the Binding Component as required by the solution.

3 File In to BPEL BP to File Out

In the spirit of "Let's get to it", here is a very simple Java CAPS 6/JBI (as well as OpenESB) project that picks up a text file, converts its content to upper case using a BPEL 2.0-based business process, and writes the result to a text file.

Let's create a Project Group, "01File2File", to group our File-to-File projects. This is illustrated in Figures 3-1 and 3-2. You don't need to create a project group. Java CAPS 6/JBI and OpenESB use NetBeans 6.1 IDE. Project organization in NetBeans 6.1 IDE is different from what you might be used to in eDesigner (5.x). There is no concept of hierarchical project structure with subprojects. Project structure is flat. Each project lives at the 'root' level. The closest one comes to project organization is Project Groups. See NetBeans 6.1 IDE help and documentation. Java CAPS 6 5.1-style projects are an exception. They are/can be structured hierarchically much as was the case in Java CAPS 5.x.

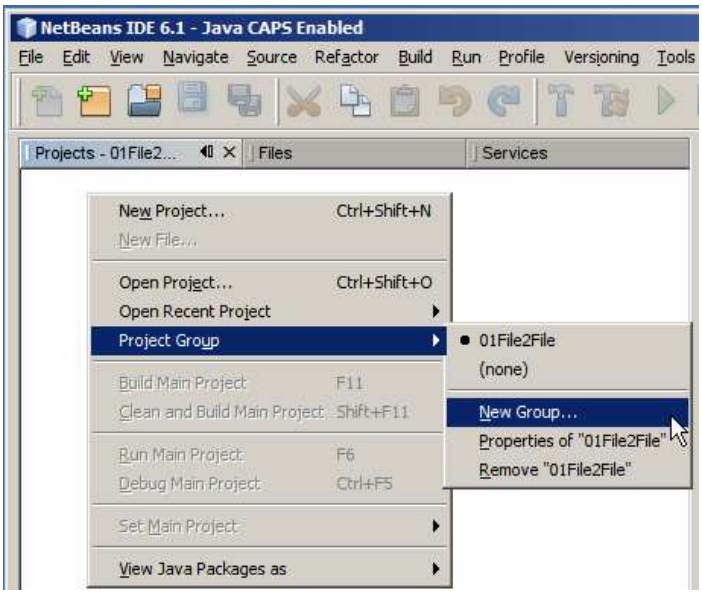

Figure 3-1 Start Project Group creation process

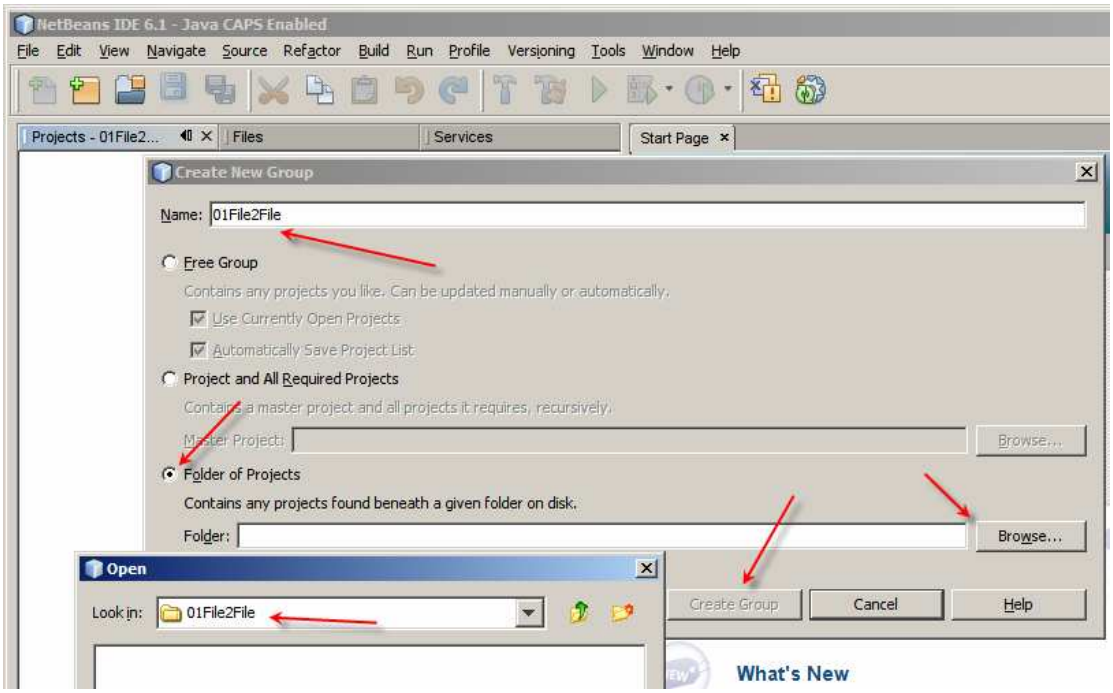

Figure 3-2 Name new project Group and identify file system directory to contain its files

We expect the file content to be read by the File Binding Component (File BC), be processed by a BPEL 2.0 Service Unit hosted in the BPEL Service Engine, the written to an output file using the File BC.

To begin, let's start by creating a new BPEL Module Project. Right-click anywhere in the empty space under the Projects – 01File2File tab, see Figure 3-3, choose SOA from the categories list and BPEL Module from projects list, then click "Next>", as illustrated in figure 3-4, name the project bmFile2File, choose the right Project Group folder to contain its files and click "Finish' as shown in Figure 3-5.

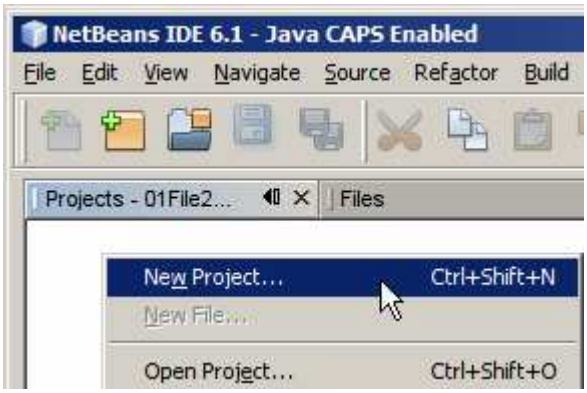

Figure 3-3 Create a New Project

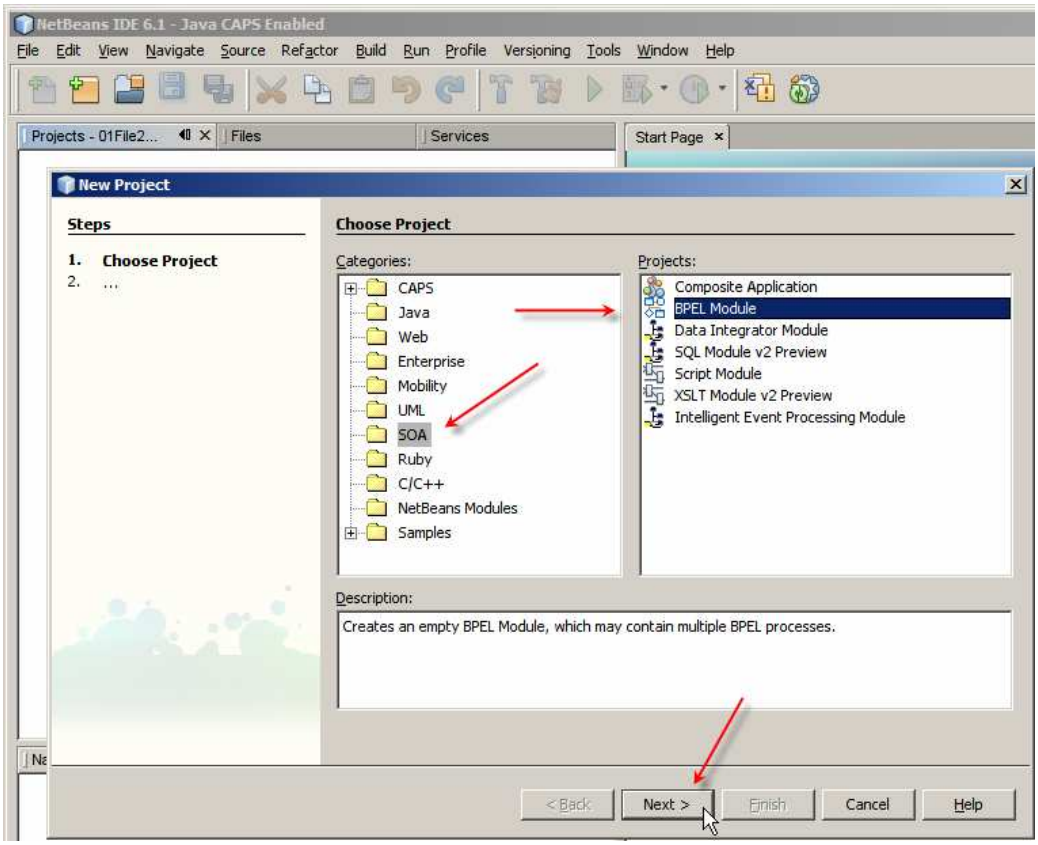

Figure 3-4 Choosing SOA -> BPEL Module Project

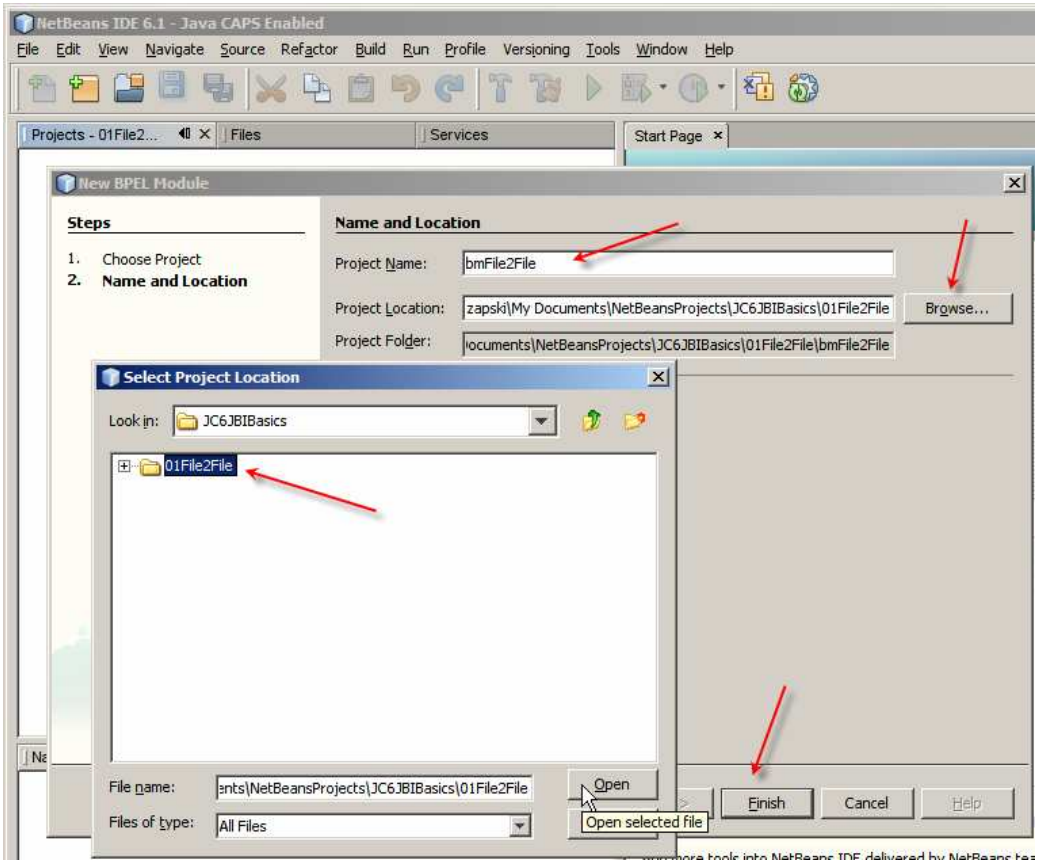

Figure 3-5 Name the project bmFile2File

Before we can design the BPEL 2.0 Business Process we need to design the message structure describing the message to be used as input to the process and the message structure describing the message to be used as output from the process. We also need to design the interface to be used by the initial receive activity and the interface to be used by the final invoke activity. The receive activity will be the first activity in the process and will be the source of the message the process instance will receive when created. The final invoke activity will be the last activity in this process and will be used to send out the message constructed/generated/transformed by the process.

Rather then going to the trouble of designing XML Schema documents describing the input message and the output message we will use simple XSD string data type for each.

The interface to be used by the initial receive activity will be described by a WSDL description. Let's create a new WSDL Document, named wsdlFileIn, describing a service with operation named "opFileIn", of type "One-way Operation", with an input message consisting of one xsd:string part named sStringIn. This process will get us the "interface" part of the WSDL and is illustrated in Figures 3-6 through 3-8.

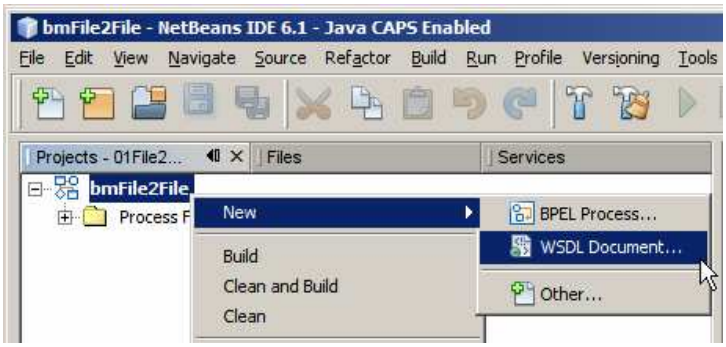

Figure 3-6 Creating a new WSDL Document

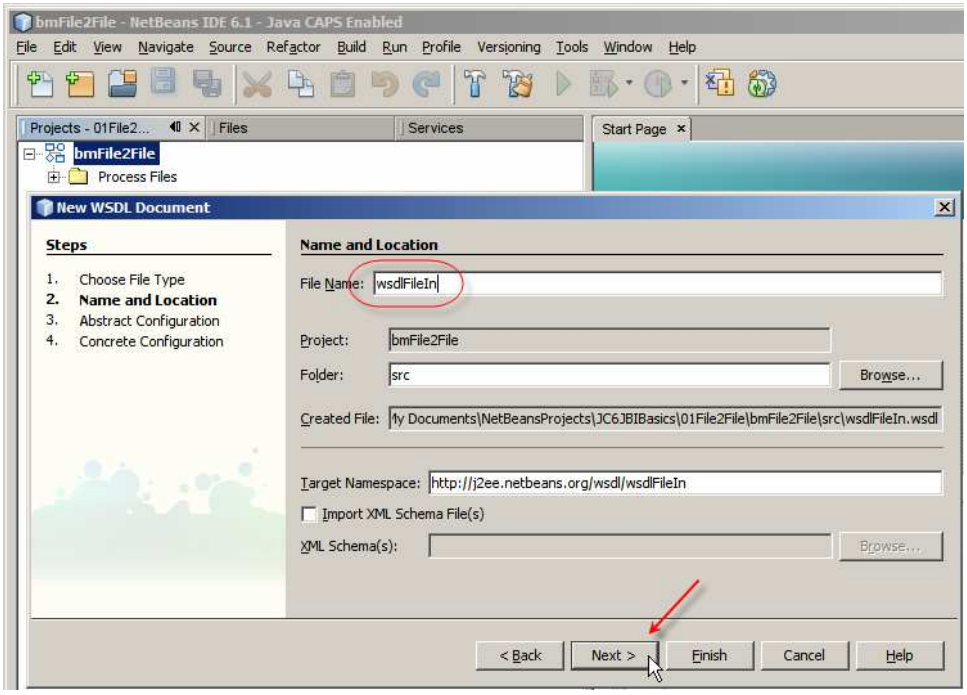

Figure 3-7 WSDL name wsdlFileIn

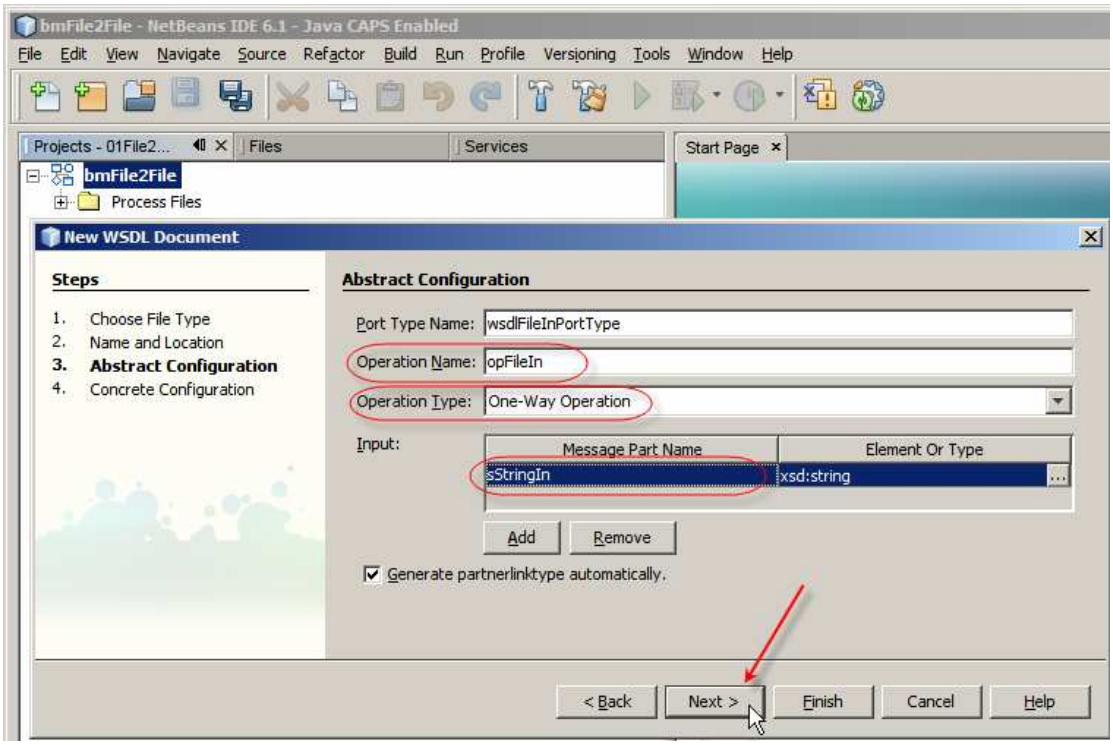

Figure 3-8 One-Way Operation opFileIn with xsd:string part sStringIn

Once the operation name, type and input message are defined we have the "interface" or the "abstract" part of the WSDL. Continuing with the "Next>" we define the "implementation" or "concrete" part of the WSDL by choosing the Binding Type. In this case we wish to use the File Binding Component so we choose the FILE binding as shown in Figure 3-9.

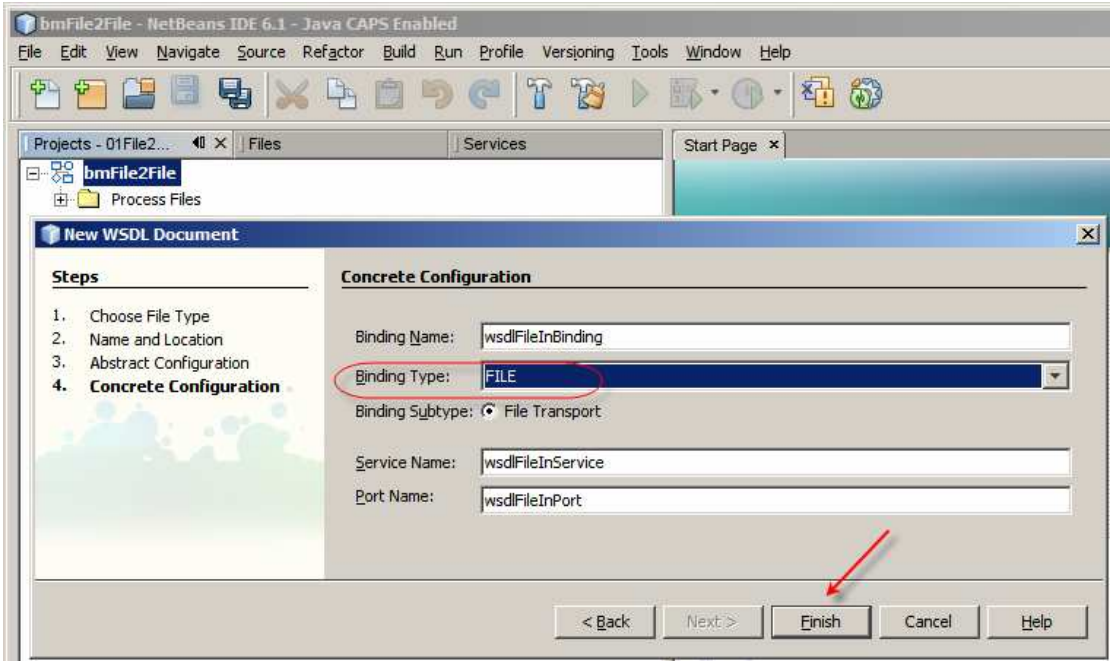

Figure 3-9 Nominate File Binding as the binding to use

Clicking "Finish" completes creation of the WSDL that describes the structure of the message submitted by the Binding Component to the BPEL 2.0 process (simple xsd:string in this case), describes the type of operation used and names that operation.

It also nominates File BC as the source of messages and provides extension nodes used to configure physical characteristics of the File BC, like directory where input files are to be found and names of files this process File BC will be looking for.

Note

> We need to have a file picked up by a File BC and delivered to the process. From the stand point of service definition this is a one-way operation. There is an input message but no output message. There is a request but no response. This is the detail that initially threw me off as the only File-to-File example I have been able to find used the File BC in request-reply mode, that is the same File BC read an input file (receive) and wrote an output file (reply).

The WSDL we created has a standard structure which the tool provided us with. This WSDL, as mentioned, is used to both specify the message structure and to configure the Binding Component's physical properties. What we need to do now is to configure the directory where the BC will look for files and the name of the file it will be looking for. For the File BC the directory is specified as the value of the Services -> wsdlFileInService -> wsdlFileInPort -> file:Address -> fileDirectory property, as illustrated in Figure 3-10. Note that we named the WSDL document wsdlFileIn. This name is used to derive service (**wsdlFileIn**Service) and port (**wsdlFileIn**Port) names shown in the WSDL.

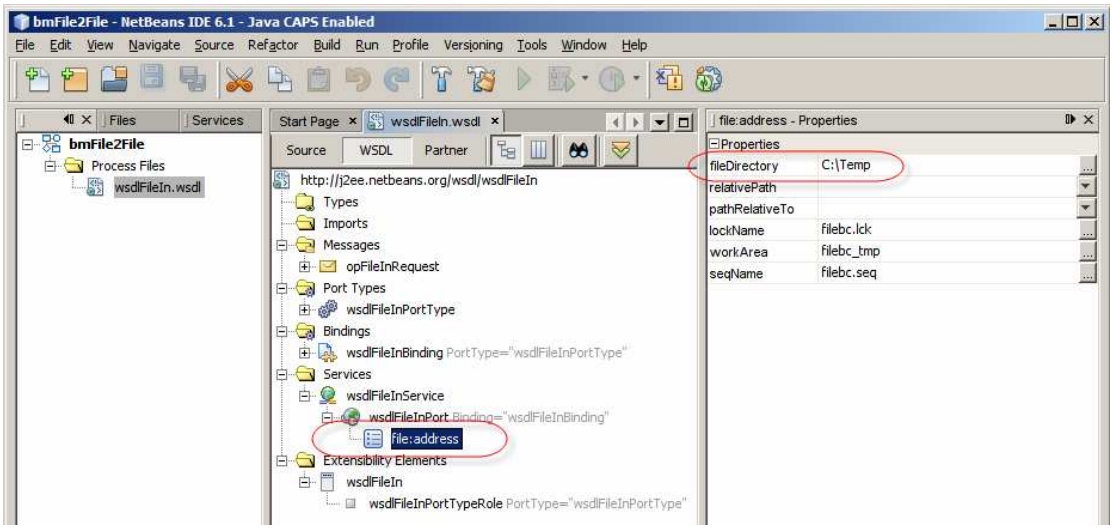

Figure 3-10 Configuring File BC's file directory property

The name of the file which the BC is to look for is specified in Bindings -> wsdlFileInBinding -> opFileIn -> input1 -> file:message -> filename property, see Figure 3-11. Note several properties there. Most notable are whether the file name is a pattern, the polling interval and pre-transfer (staging) and post-transfer (archive) directory and name properties. Note again some names derived from the name of the WSDL, wsdlFileIn.

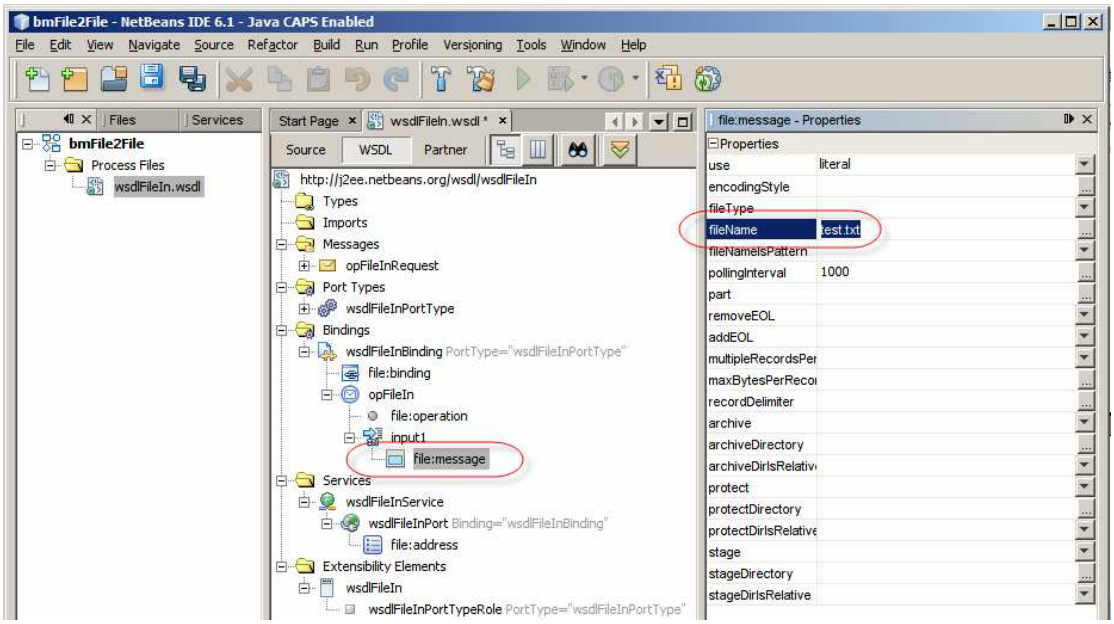

Figure 3-11 Specifying name of the file

Once the WSDL naming the input message and configuring the inbound File BC is ready we need to create a WSDL for the outbound File BC.

Let's create a new WSDL document, wsdlFileOut, as illustrated in Figures 3-12 through 3-15. The steps illustrated in the figures are very much the same as these we went through to create the WSDL for the inbound File BC.

Figure 3-12 New WSDL for the Outbound File BC

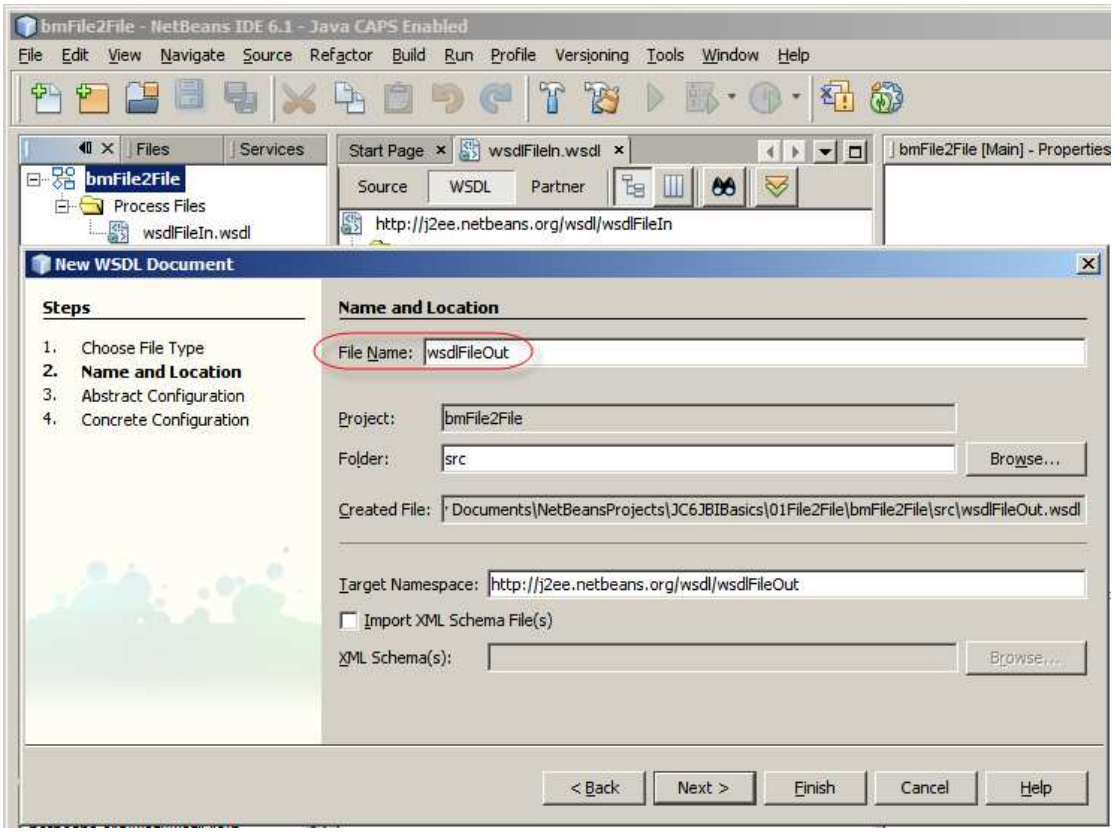

Figure 3-13 Naming the WSDL file

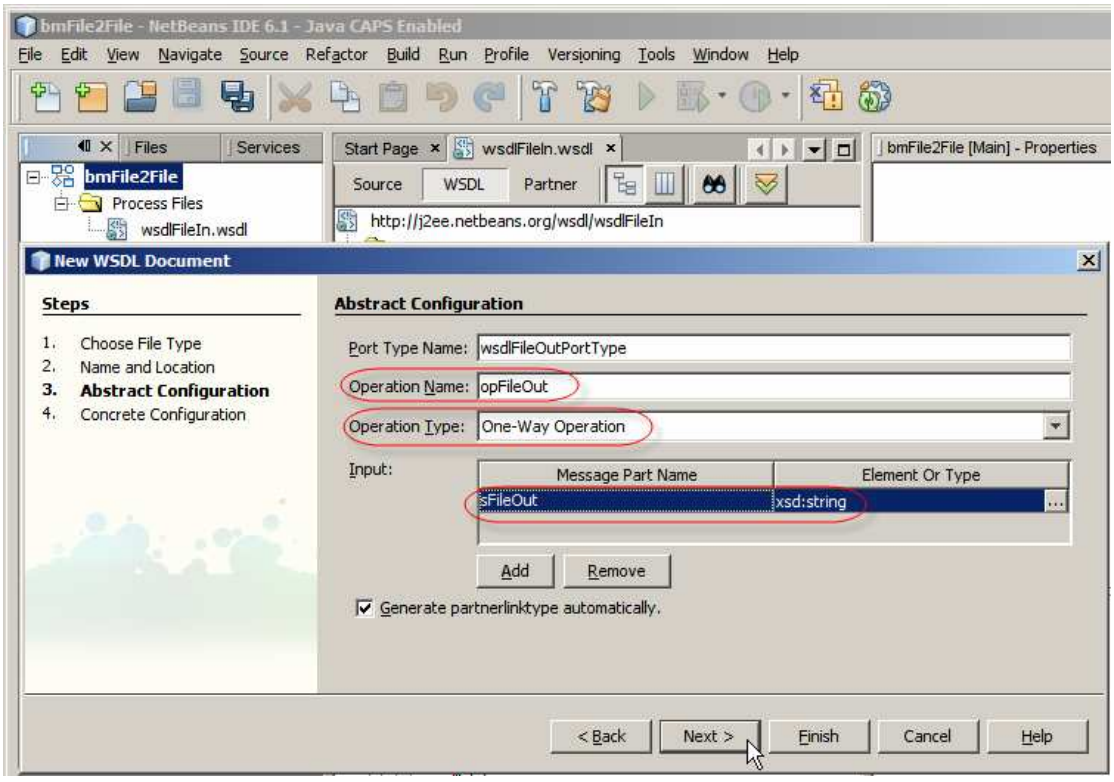

Figure 3-14 Naming the operation, choosing operation type and naming the input message part

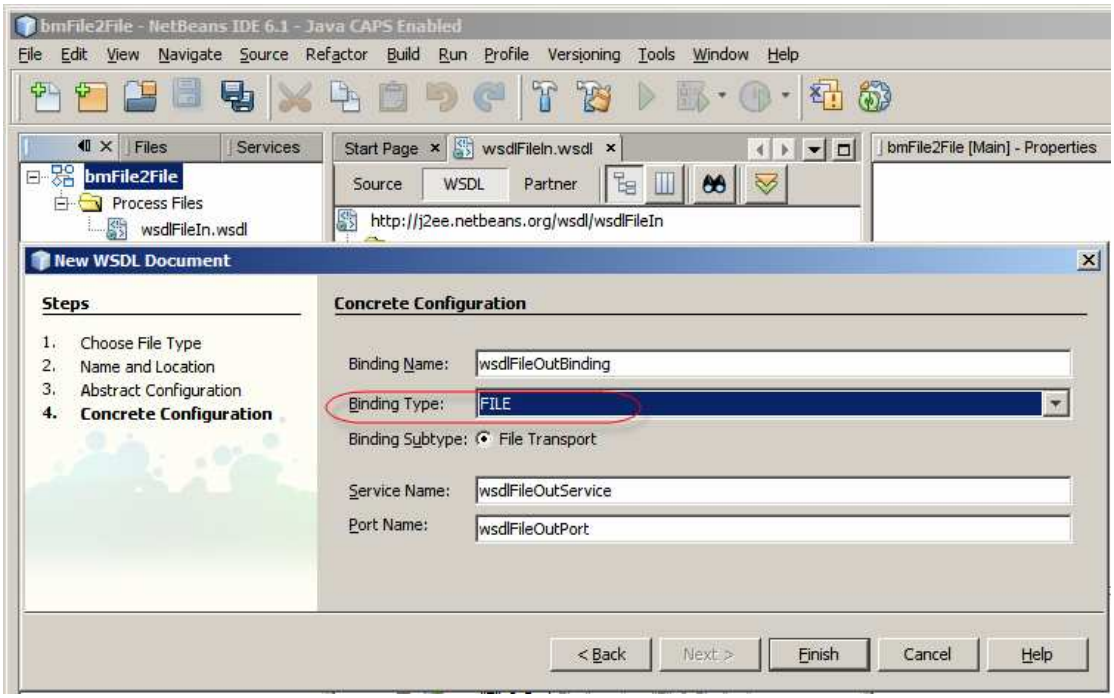

Figure 3-15 Choosing File binding

Once the WSDL is created we need to configure the output directory and the output file name. Much as was done for the inbound File BC Services -> wsdlFileOutService -> wsdlFileOutPort -> file:address -> fileDirectory property names the directory to which the output file will be written, see Figure 3-16.

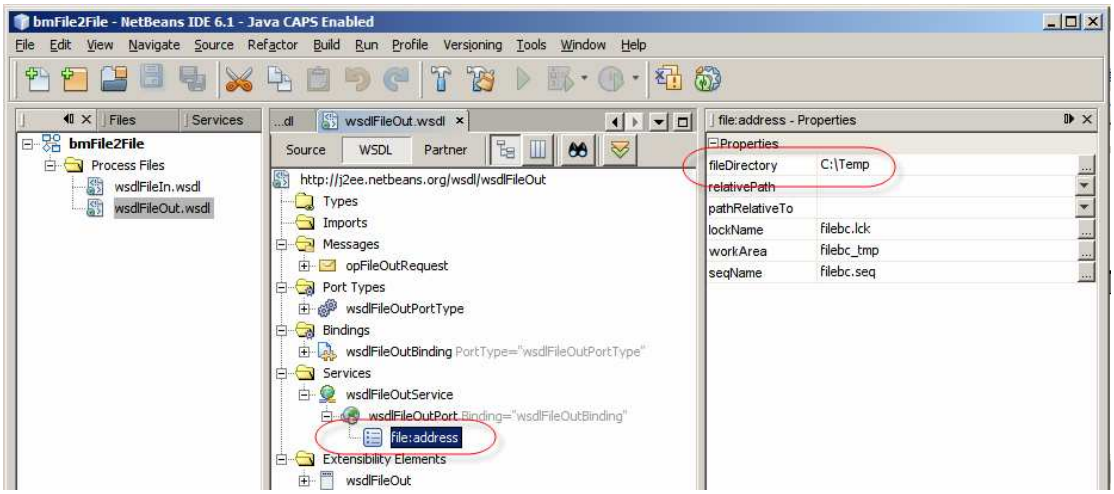

Figure 3-16 Naming output directory

Much as was done for the inbound File BC Bindings -> wsdlFileOutBinding -> opFileOut -> input1 -> file:message -> filename and fileNameIsPattern properties are used to configure the name of the output file (test_%d.out) and the fact that the name is a pattern. Figure 3-17 illustrates this.

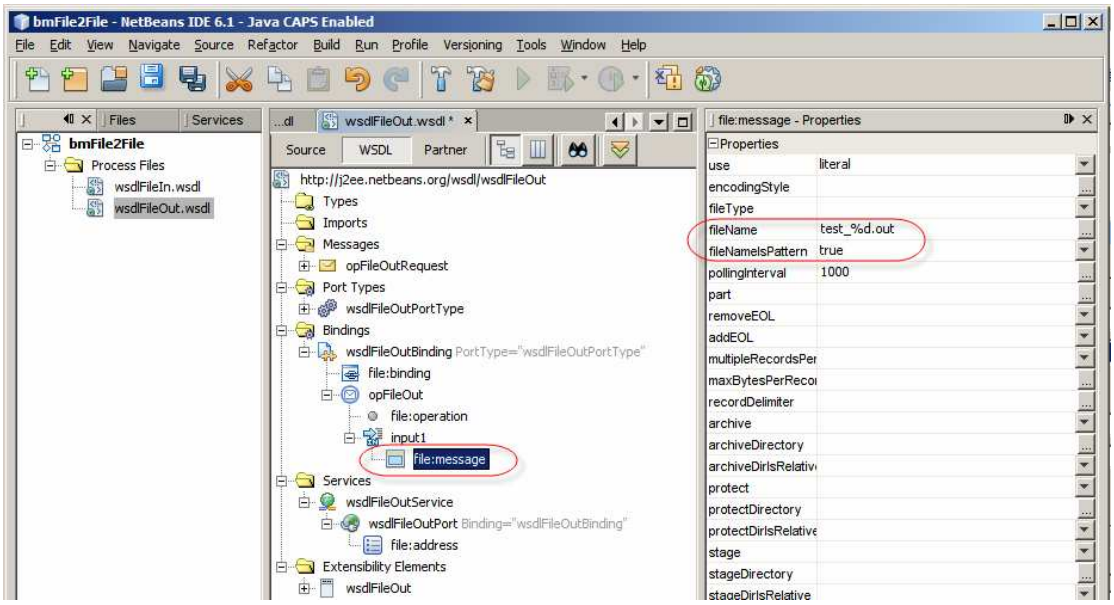

Figure 3-17 Providing output file's file name pattern

Once both WSDLs are ready we can create the BPEL process that will receive the payload from the input file, process it and write the result to the output file. See figures 3-18 and 3-19.

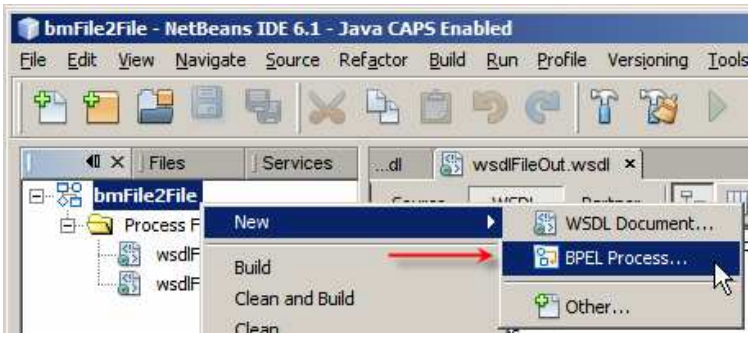

Figure 3-18 Starting BPEL Process creation

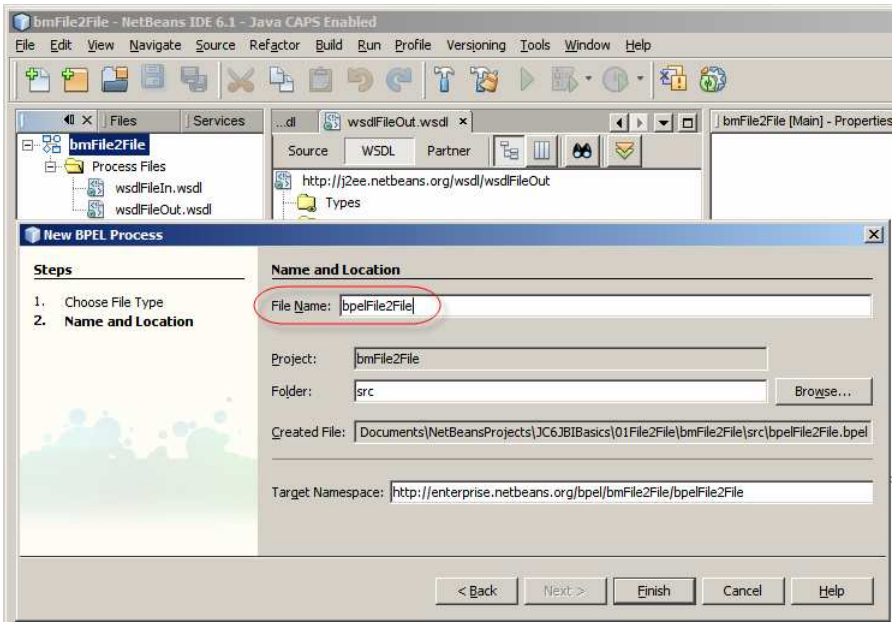

Figure 3-19 Naming BPEL Process – bpelFile2File

To begin, let's drag the wsdlFielIn WSDL to the 'inbound' swim-line of the process model, as illustrated in Figure 10-20. Make sure to drag the WSDL precisely onto the "target marker".

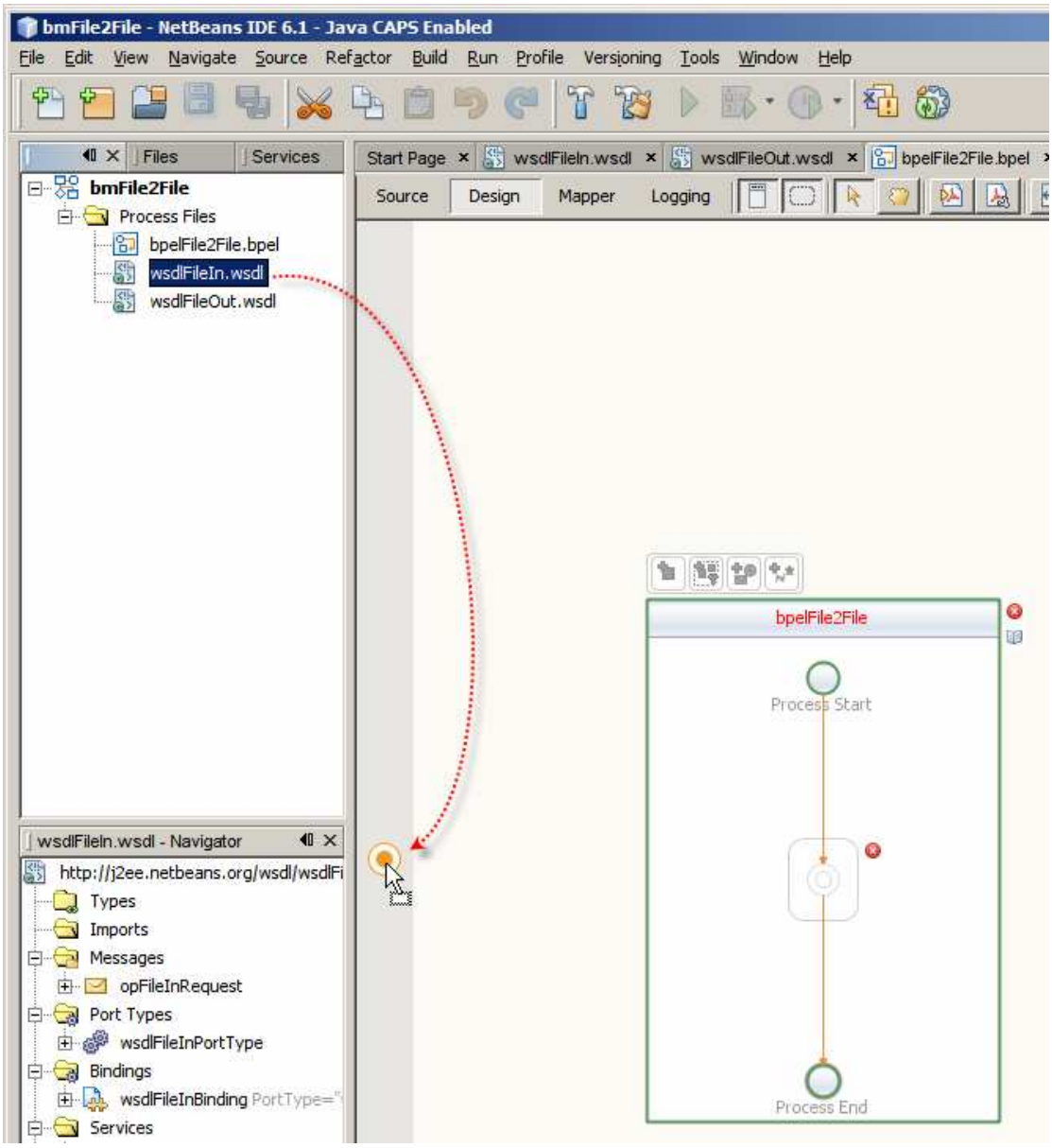

Figure 3-20 Adding contract for the inbound File BC to the process model

We could leave it at that. I am fastidious, somewhat, so I like to rename default partner links so the names reflect the partners. Figures 3-21 and 3-22 illustrate the process of changing partner link names.

We now have the WSDL that describes the input message this process will be dealing with.

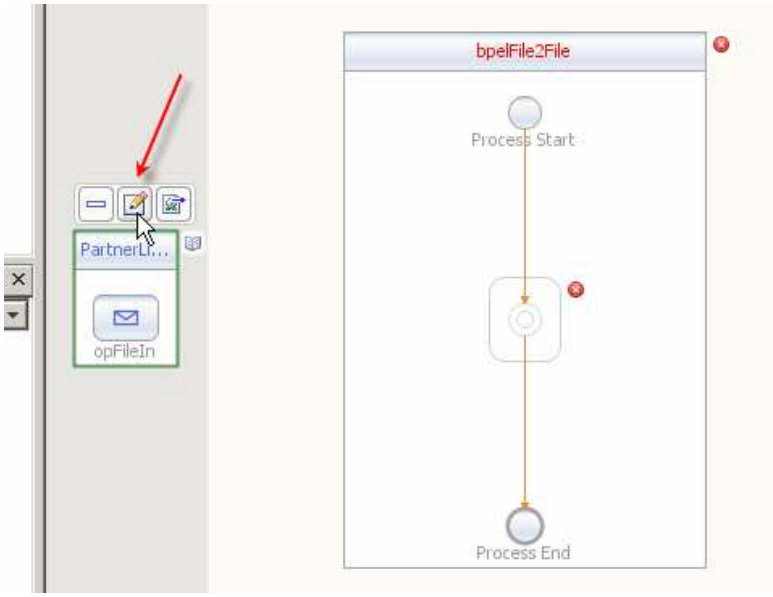

Figure 3-21 Editing Partner Link properties

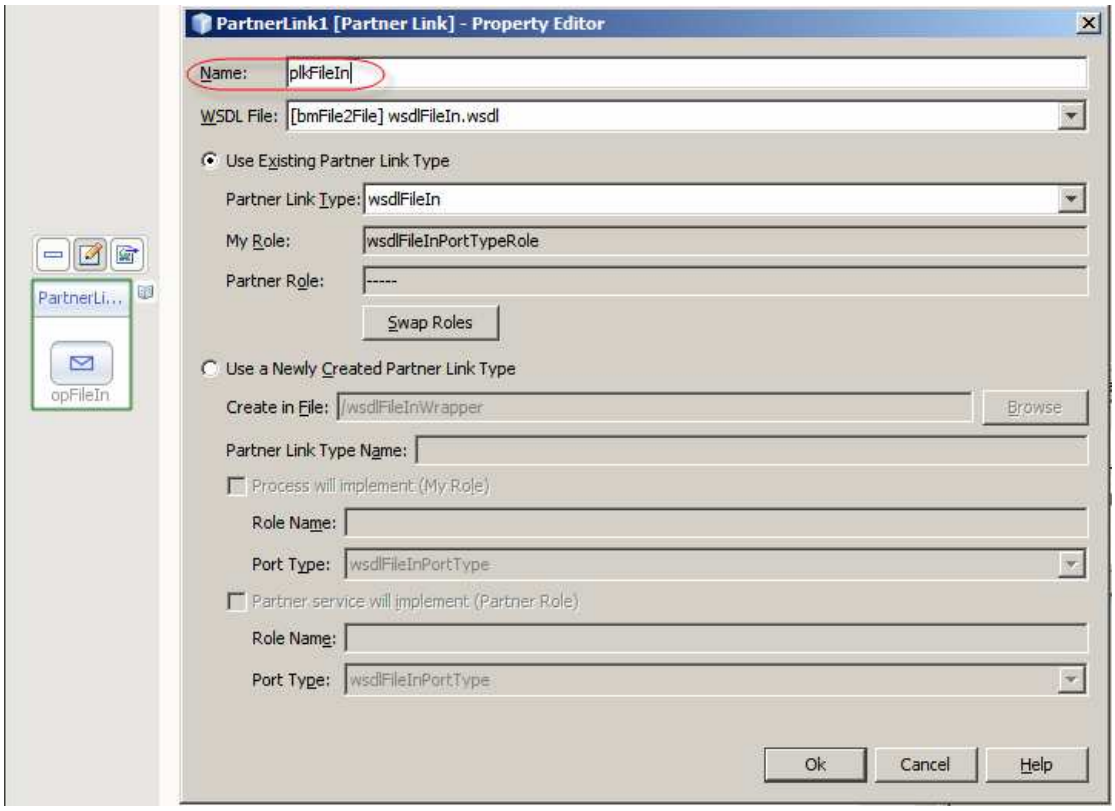

Figure 3-22 Changing default name of the Partner Link

Let's now drag the WSDL corresponding to the output message and BC onto the 'outbound'/'invoke' swim-line. Figures 3-23, 3-24 and 3-25 illustrate the process of adding the new partner link and changing its name to something more appropriate, much as has been done for the 'inbound' side.

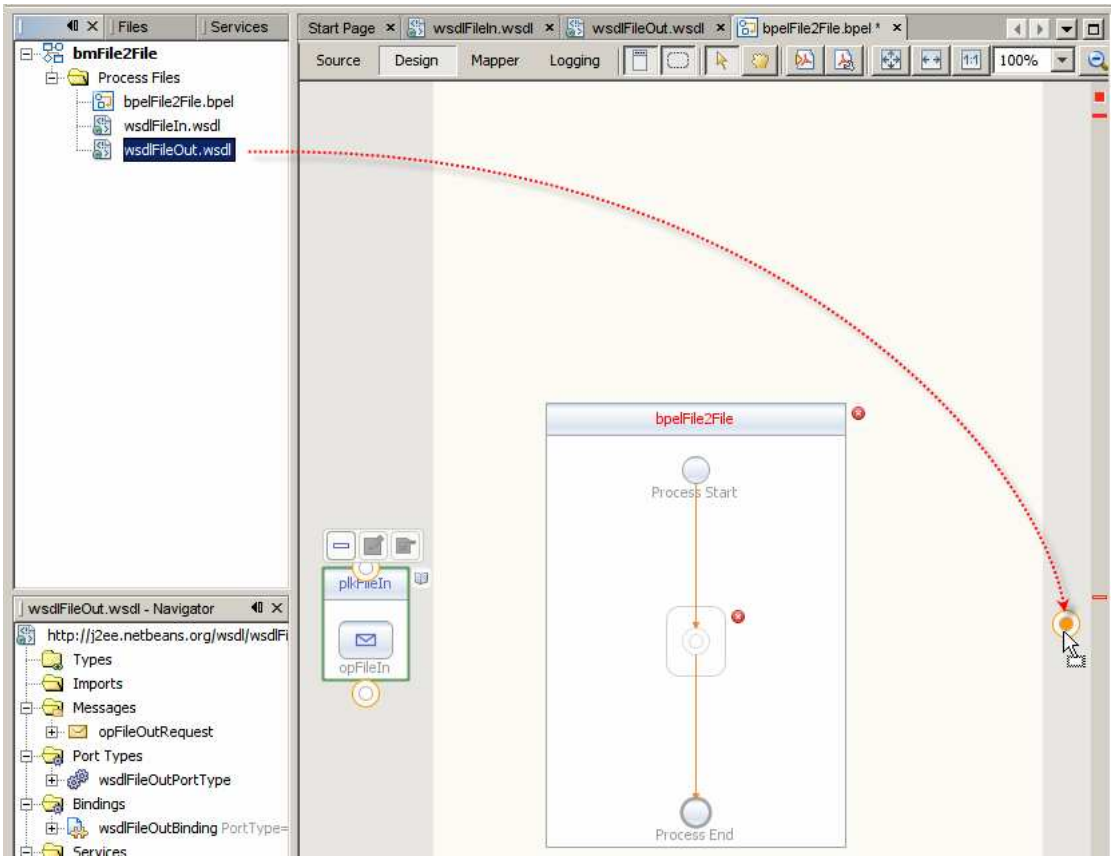

Figure 3-23 Adding outbound Partner Link

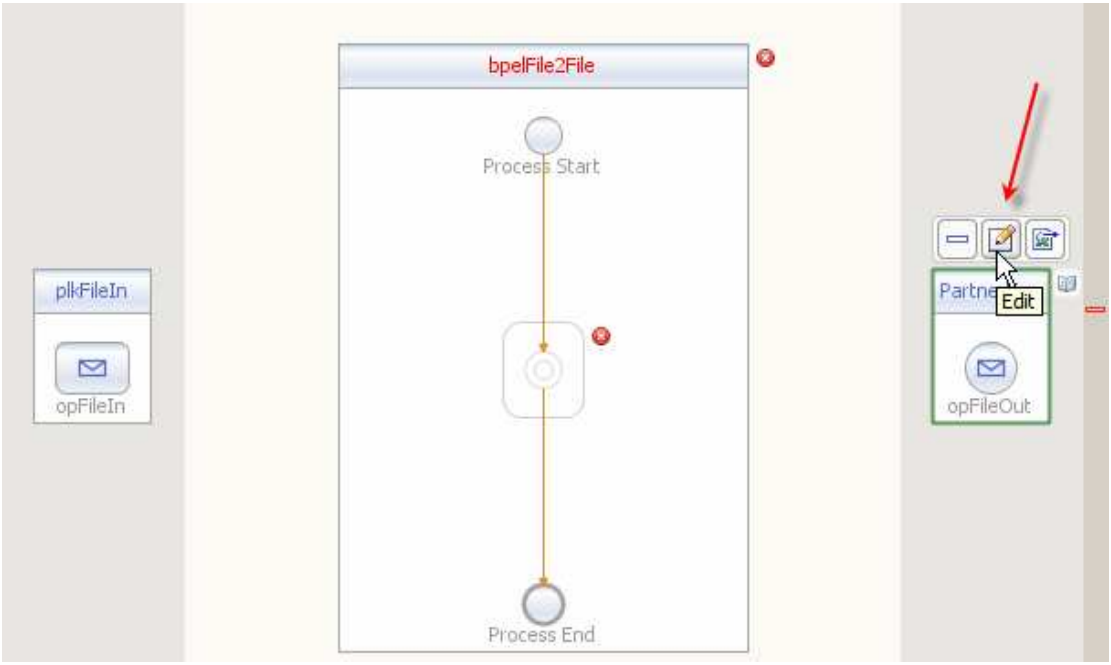

Figure 3-24 Editing Partner Link properties

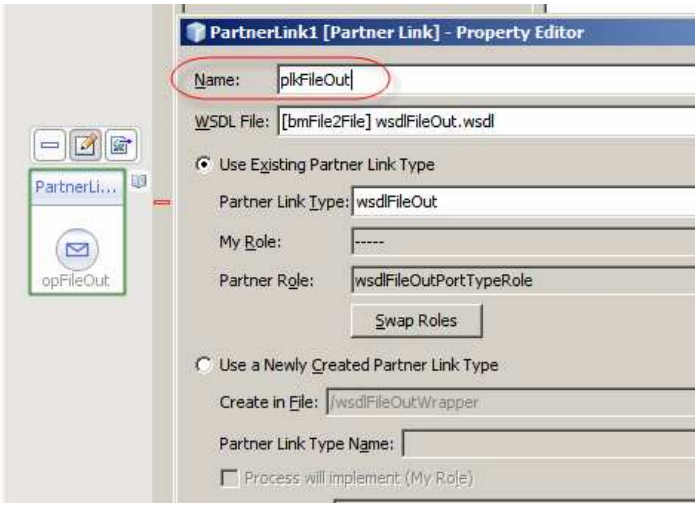

Figure 3-25 Changing Partner Link name

As one would expect, the process will receive the payload from the File BC (receive activity), use BPEL mapping rules to convert the payload string to upper case (assign activity) and pass the uppercase text to the output File BC for writing (invoke).

Let's start by dragging the 'Receive' activity from the activities palette to the process modeling canvas aiming at the 'target marker' inside the scope between Process Start and Process End activities as illustrated in Figure 3-26.

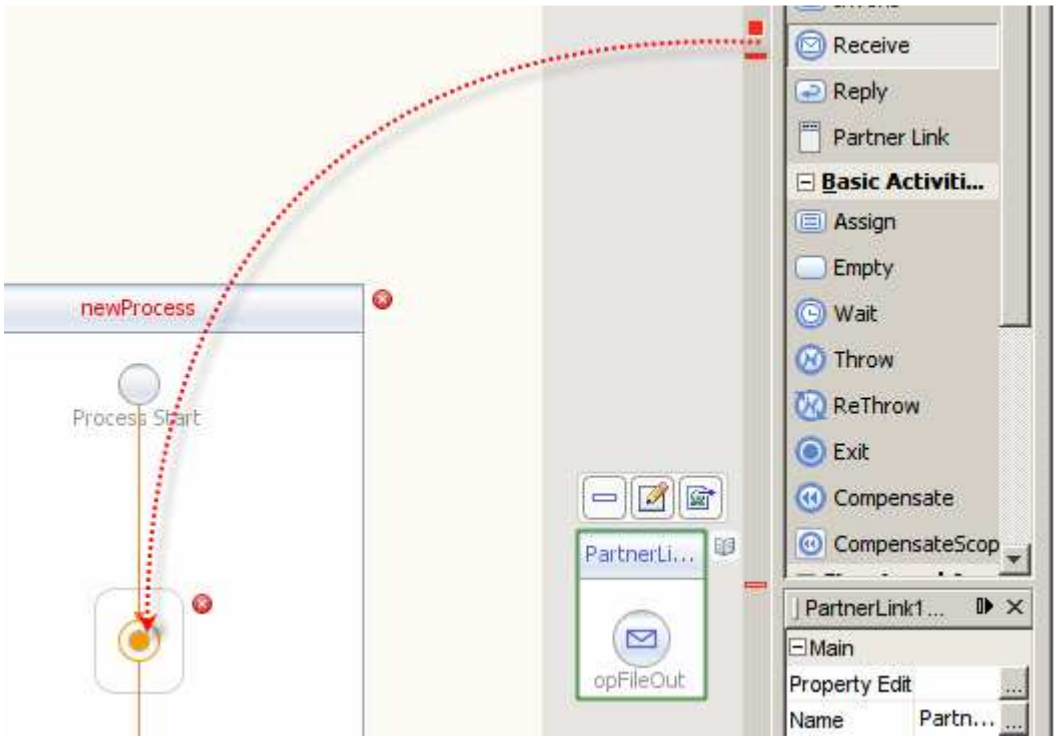

Figure 3-26 Adding Receive activity to the process model

Next, let's drag the Assign activity to the 'target marker' below the receive activity we just added. Figure 3-27 illustrates this.

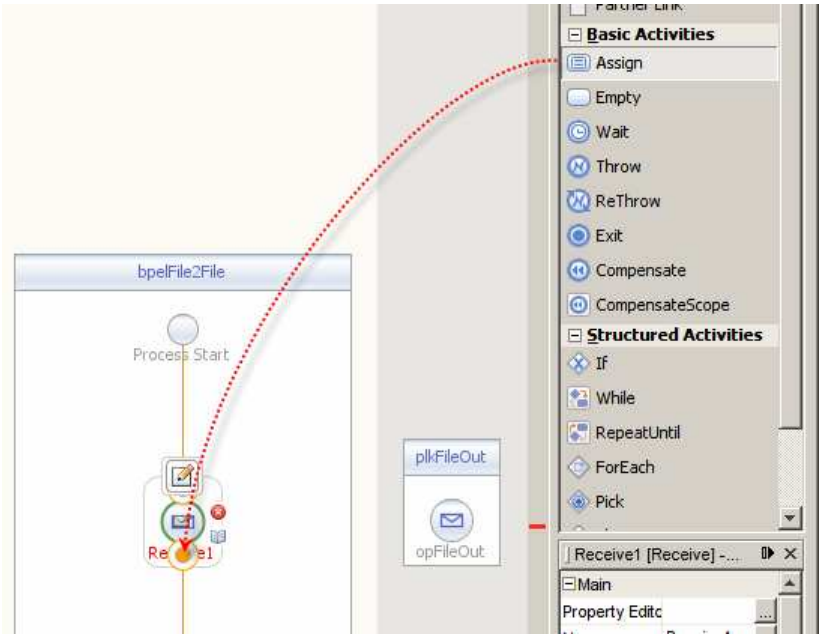

Figure 3-27 Adding Assign activity

Finally, let's drag the Invoke activity onto the 'target marker' following the Assign activity we just added. We are using the Invoke activity because we are invoking the File BC. The process we are building is not a synchronous process, invoked as a request/reply service, therefore it does not have the Reply activity as the final activity.

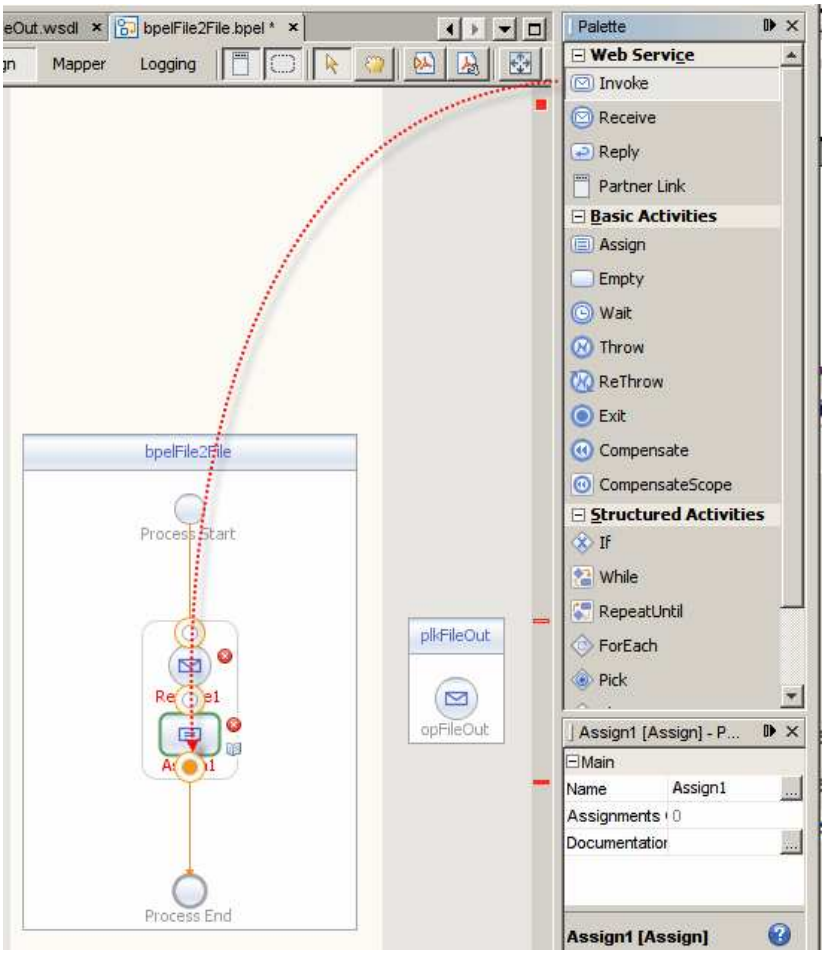

Figure 3-28 Adding the Invoke activity

Let’s rename the activities to rcvFilePayload, mapInput and sndFileOut respectively, as illustrated in Figure 3-29.

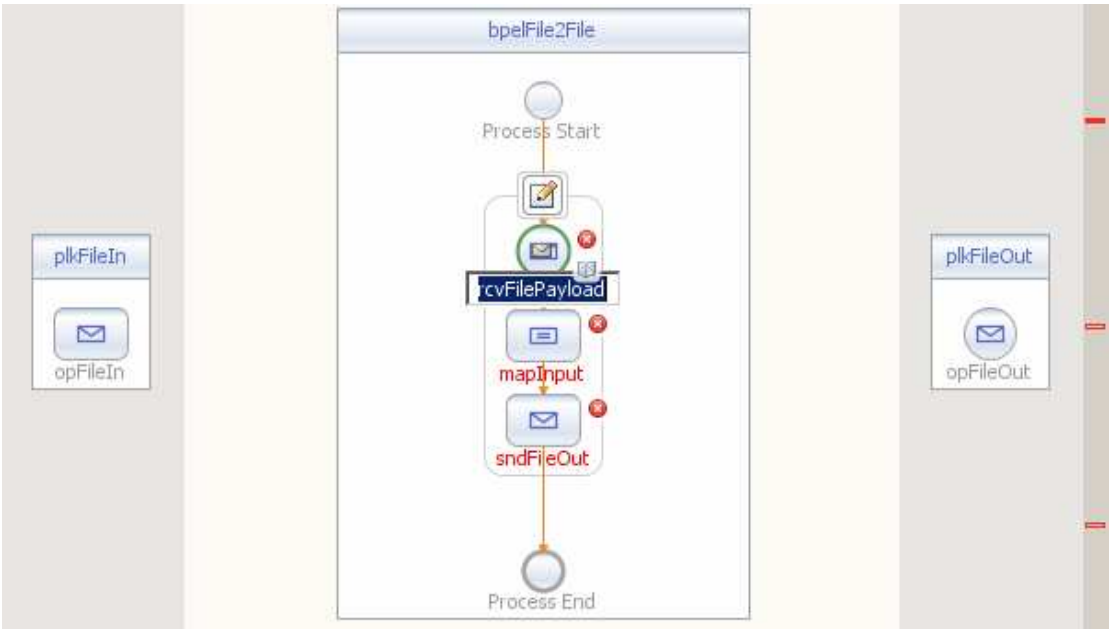

Figure 3-29 Explicitly naming activities

Let’s connect the Receive activity to its corresponding partner link by click-dragging from the activity icon to the partner link icon and releasing. Figure 3-20 illustrates this.

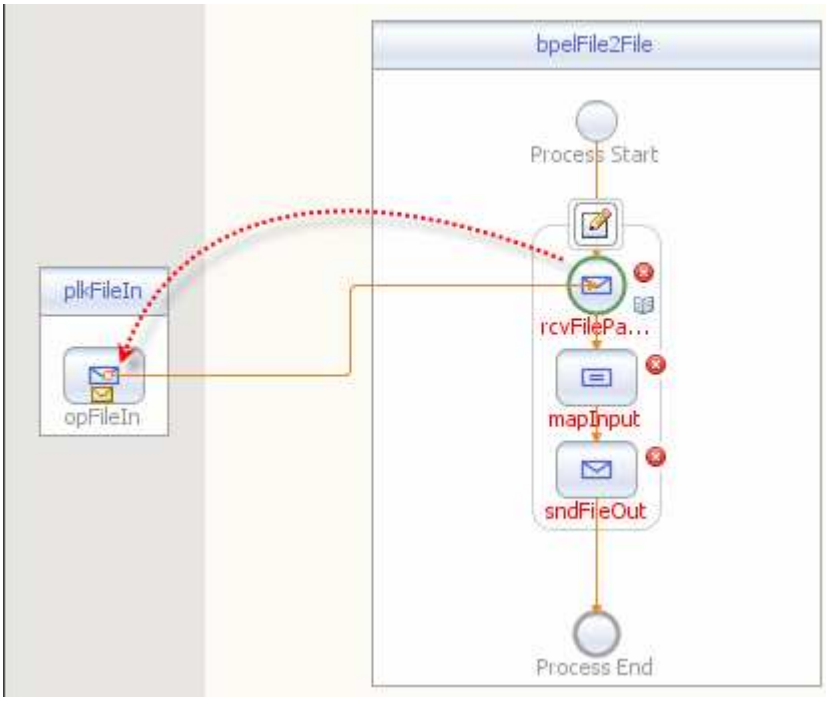

Figure 3-30 Connecting the receive activity and the partner link

Let's now connect the Invoke activity to its corresponding partner link by click-dragging from the activity icon to the partner link icon and releasing. Figure 3-21 illustrates this.

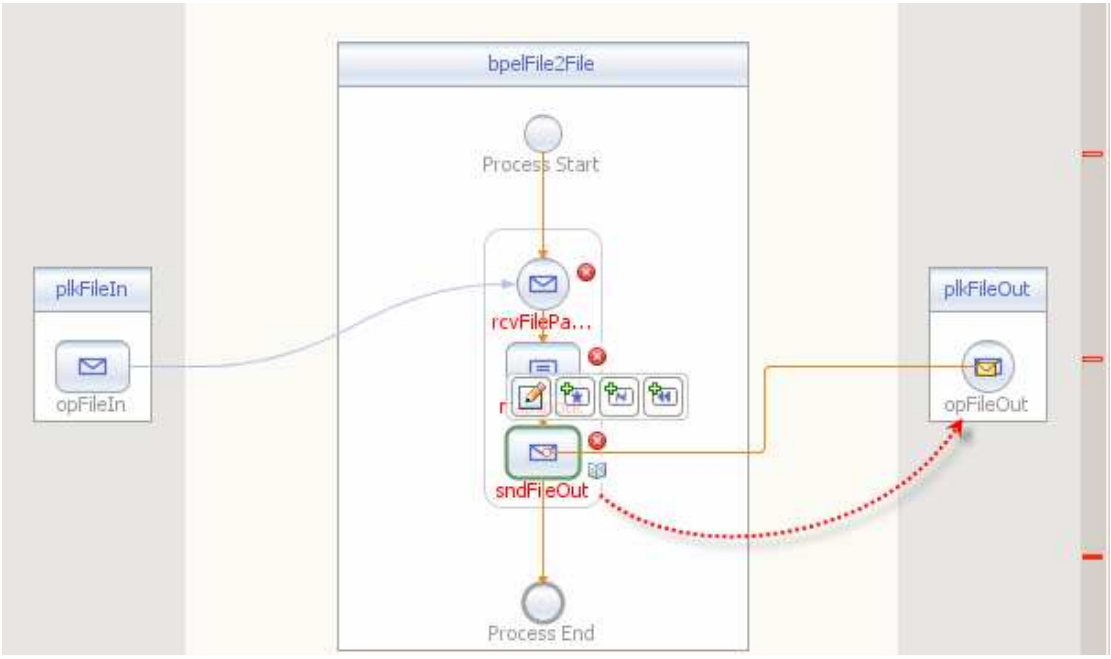

Figure 3-31 Connecting the Invoke activity and its Partner Link

Figure 3-32 shows the process model with both activities connected to their partner links.

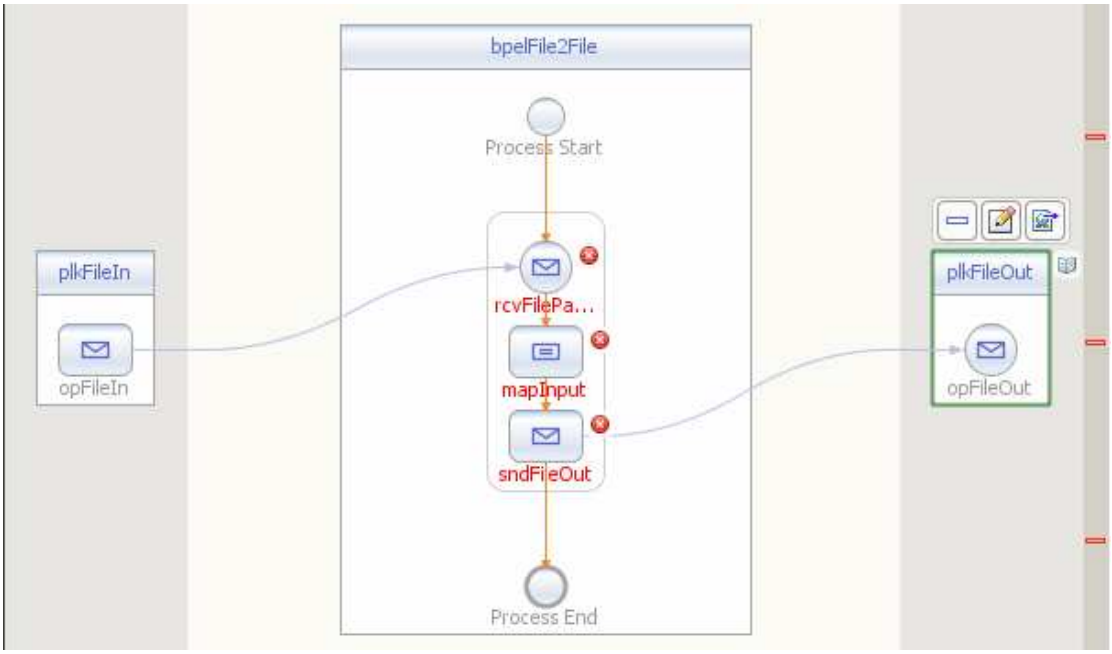

Figure 3-32 Receive and Invoke activities connected to their partner links

Let's configure the Receive activity and add to it the Input Variable. This variable will be the variable representing the inbound message. Figures 3-33 through 3-36 illustrate the process.

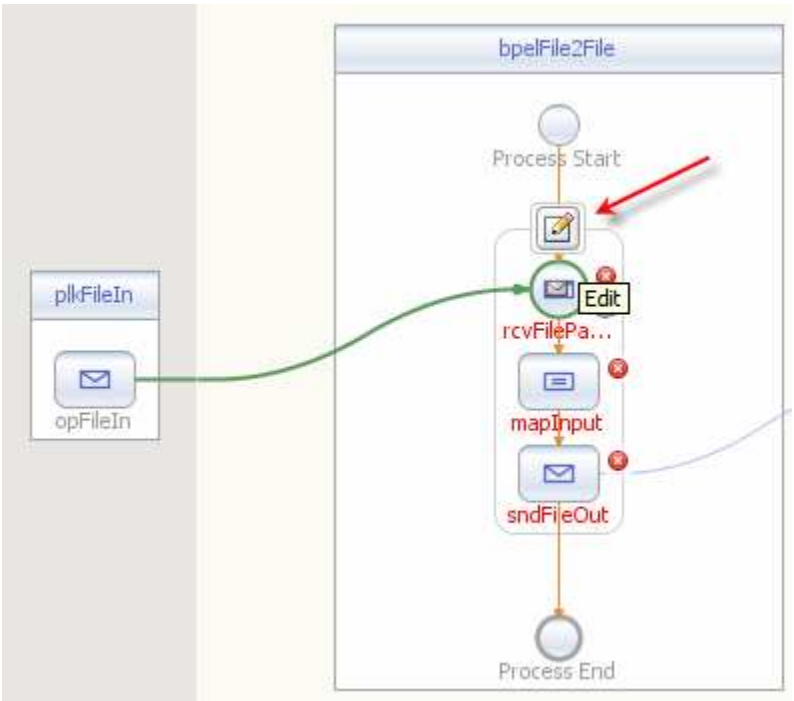

Figure 3-33 Locating the "Edit Properties" "button"

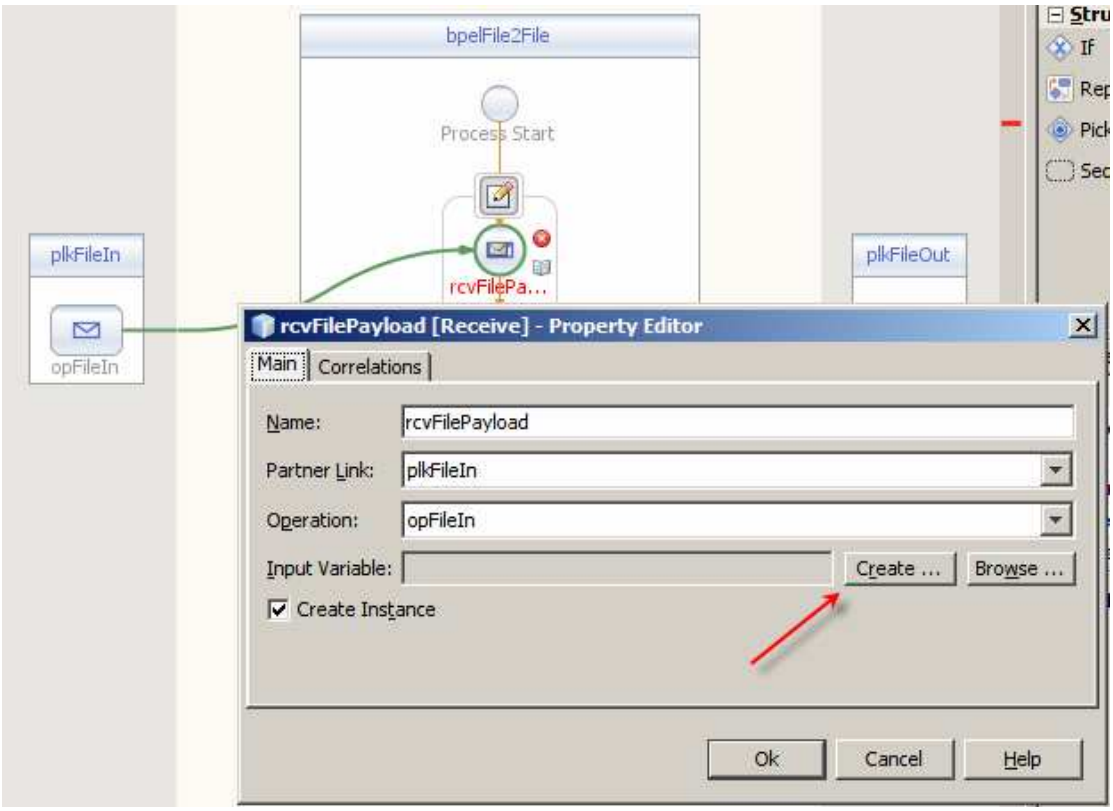

Figure 3-34 Initiating Input Variable creation process

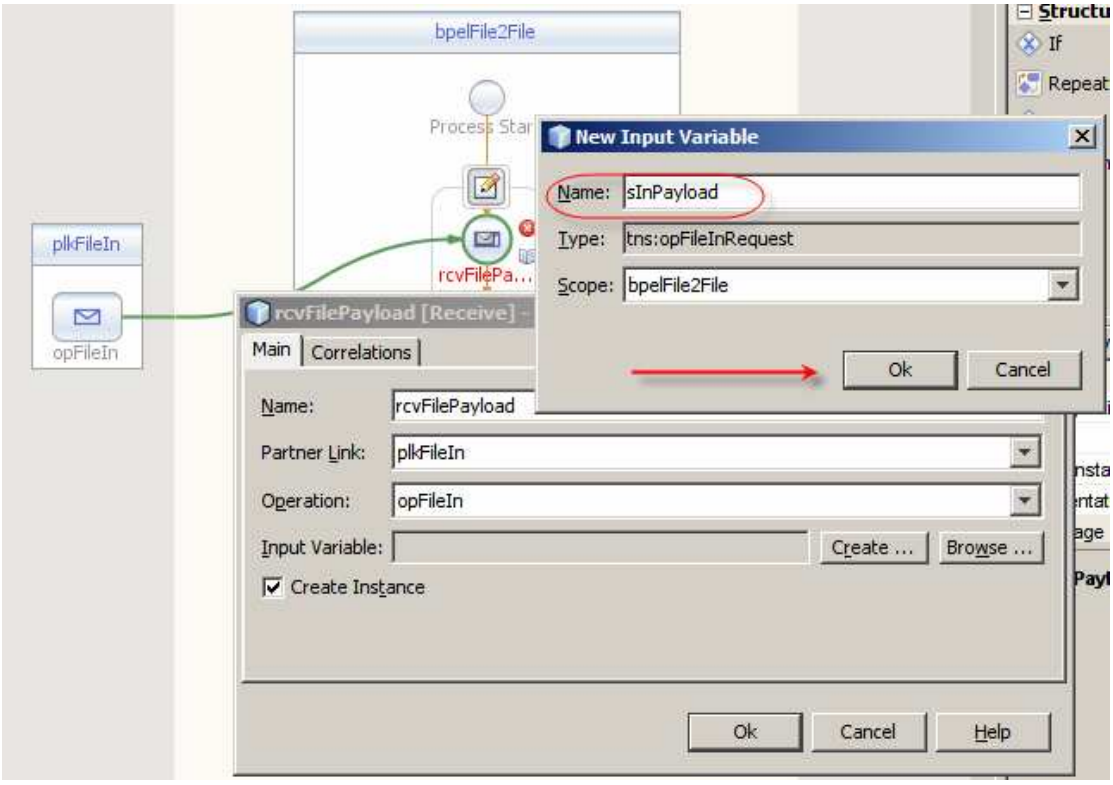

Figure 3-35 Naming the Input Variable

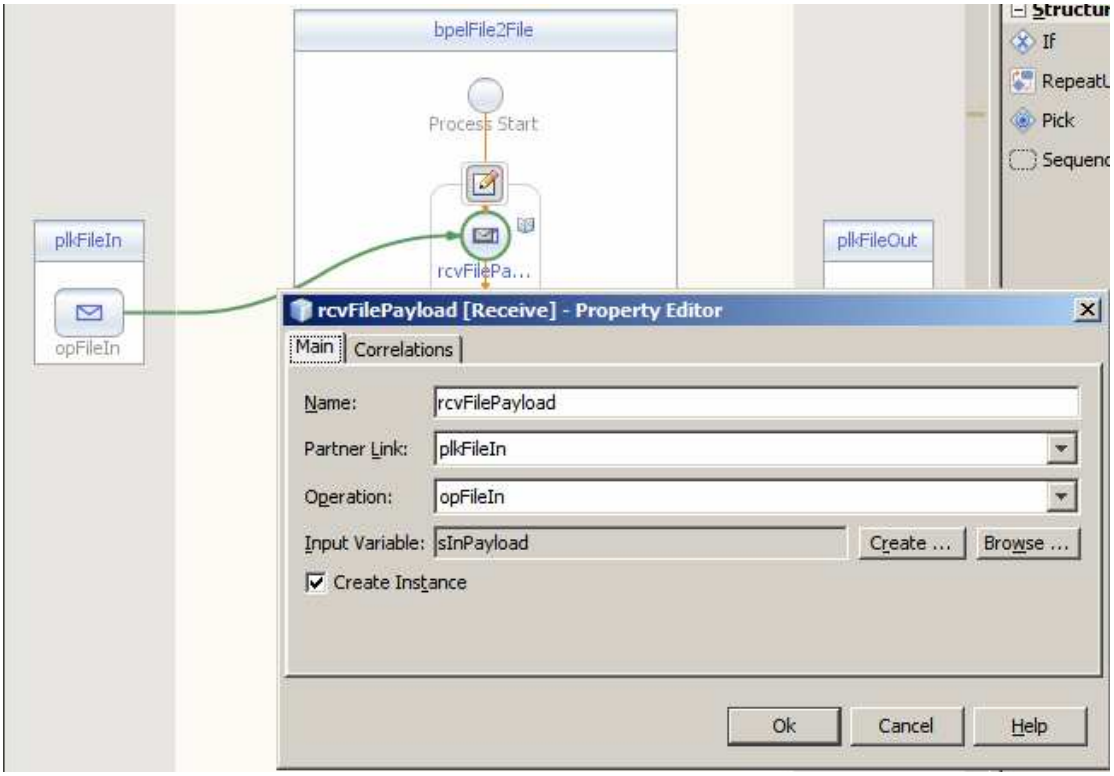

Figure 3-36 Receive activity configured with the Input Variable

Now we need to configure the Invoke activity and add an input variable to it, much as was done for the Receive activity. This Input Variable will represent the outbound message. Figures 3-37 through 3-40 illustrate the steps involved in this process.

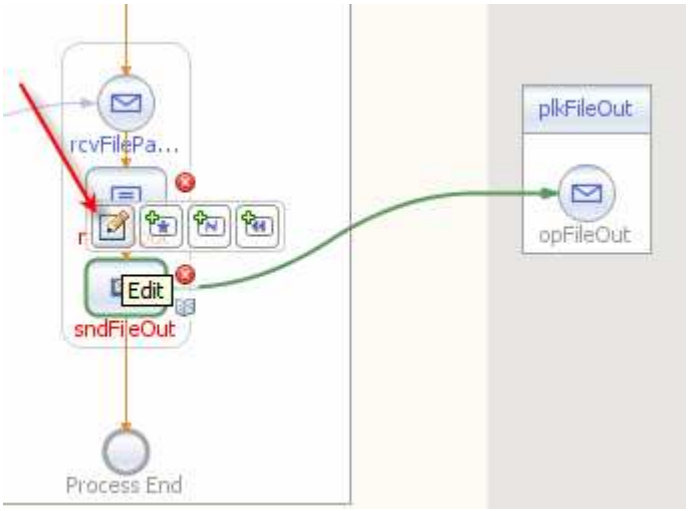

Figure 3-37 Locating “Edit Properties” “button”

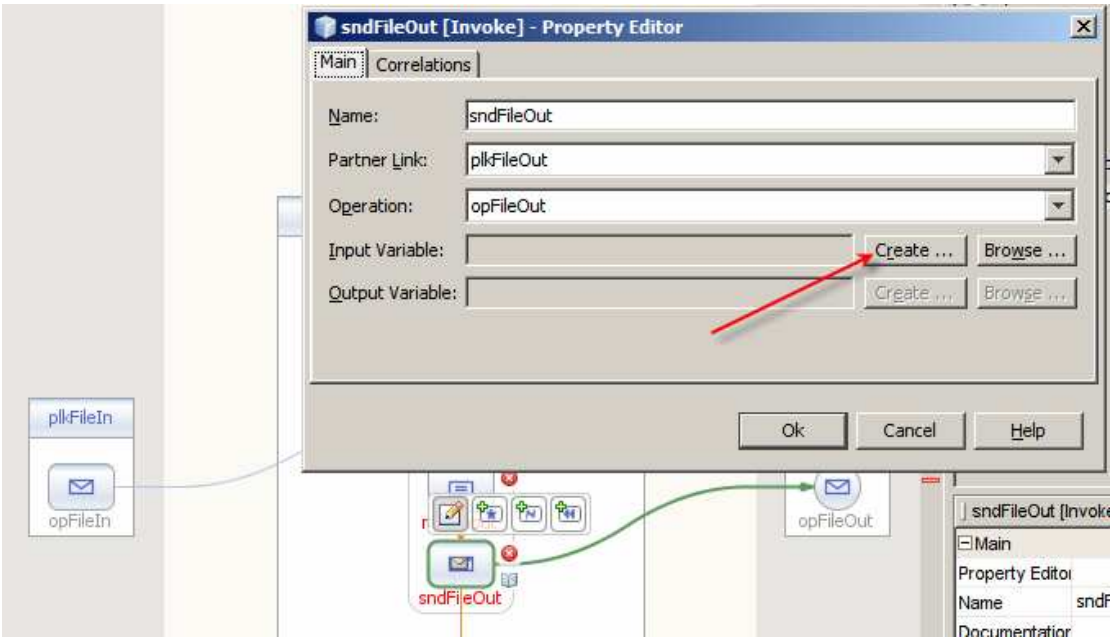

Figure 3-38 Initiating creation of Input Variable for the Invoke activity

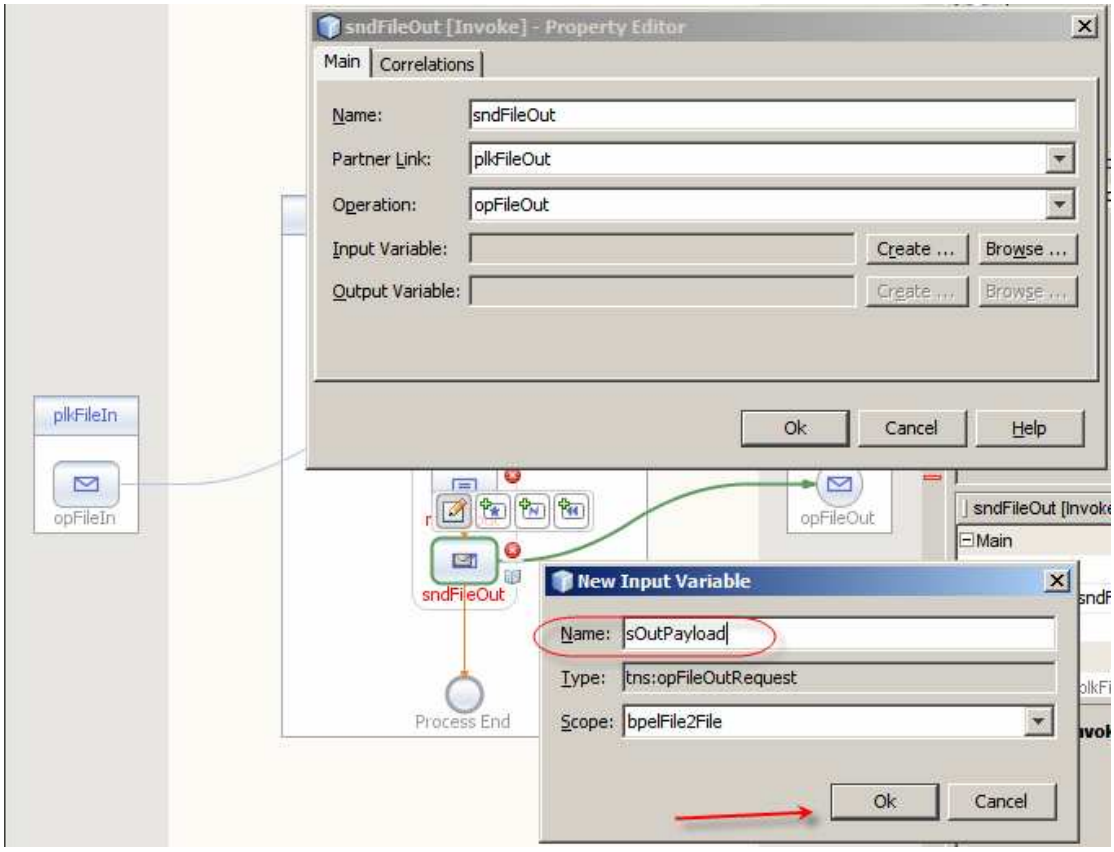

Figure 3-39 Naming the Input Variable

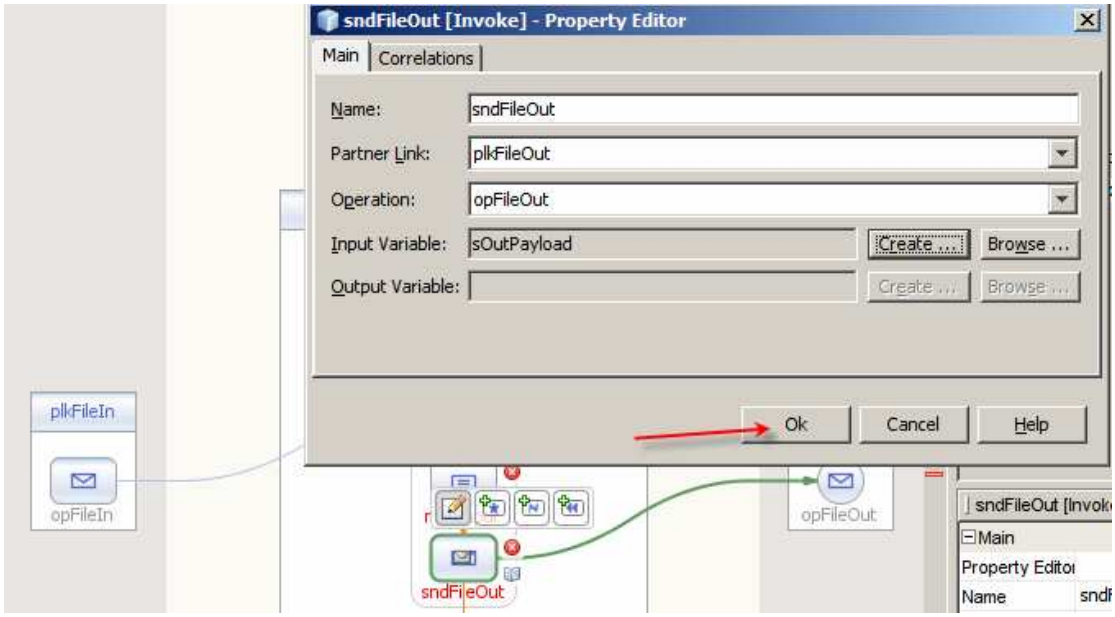

Figure 3-40 Completing creation of the Input Variable for the Invoke activity

With the variable representing the input message and the variable representing the output message we can now perform the mapping/transformation required to convert the input text to upper case.

Right-click on the Assign activity and choose Go To -> BPEL Mapper option as shown in Figure 3-41.

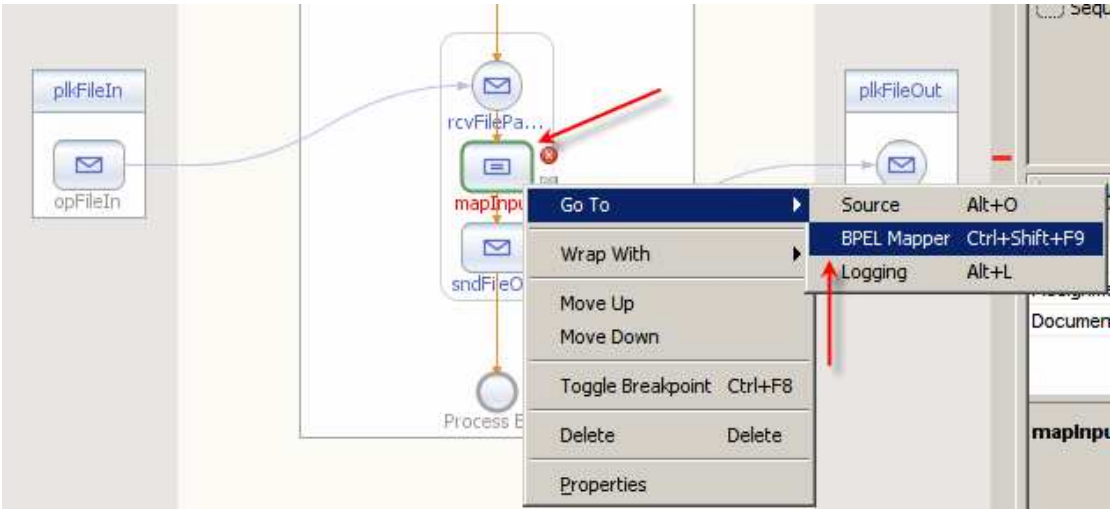

Figure 3-41 Accessing the BPEL Mapper (one way)

Let's select the input variable (left hand side) and the output variable (right hand side) and drag the Translate string function into the "swim line" associated with the output variable as shown in Figure 3-42. BPEL / XPath has no notion of a case conversion function so we need to use the Translate function. This function will be used to translate all lower case latter to their corresponding upper case letters as shown in Figure 3-43.

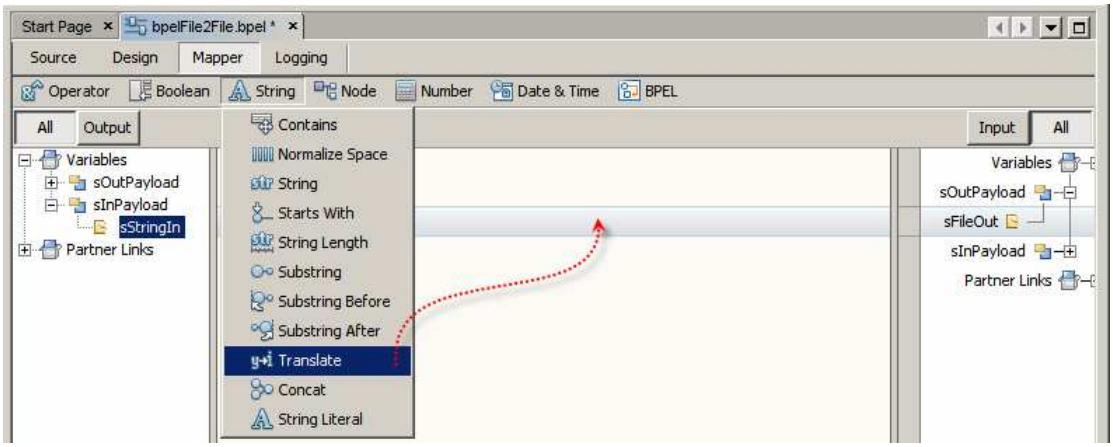

Figure 3-42 Add Translate function to the mapper canvas

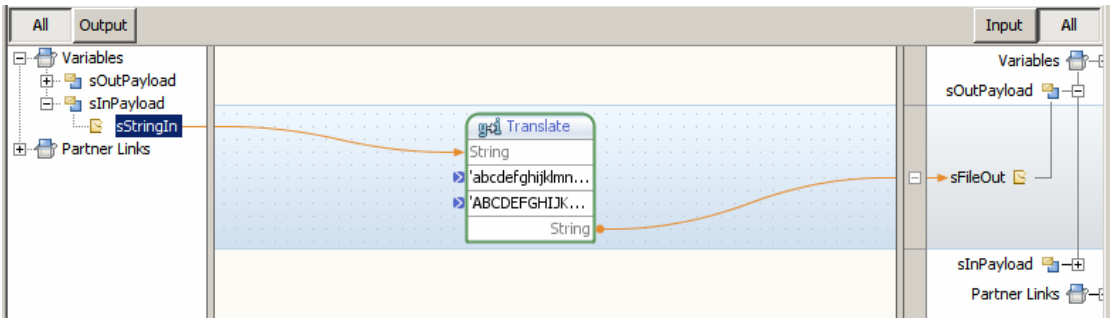

Figure 3-43 Configure the Translate function and connect its input and output

With this simple mapping completed the process is ready. Let's choose the "Clean and Build" menu option, as shown in Figure 3-44.

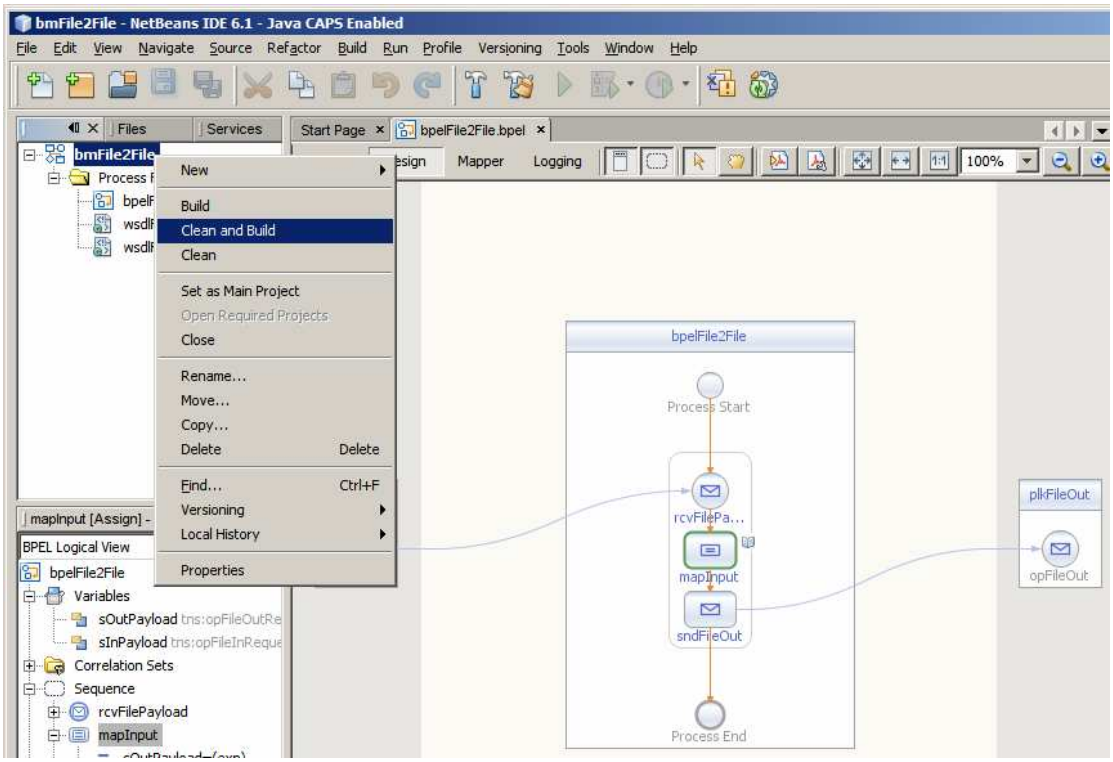

Figure 3-44 Choosing "Clean and Build" menu option to build the process

Figure 3-45 shows the success messages following the build process.

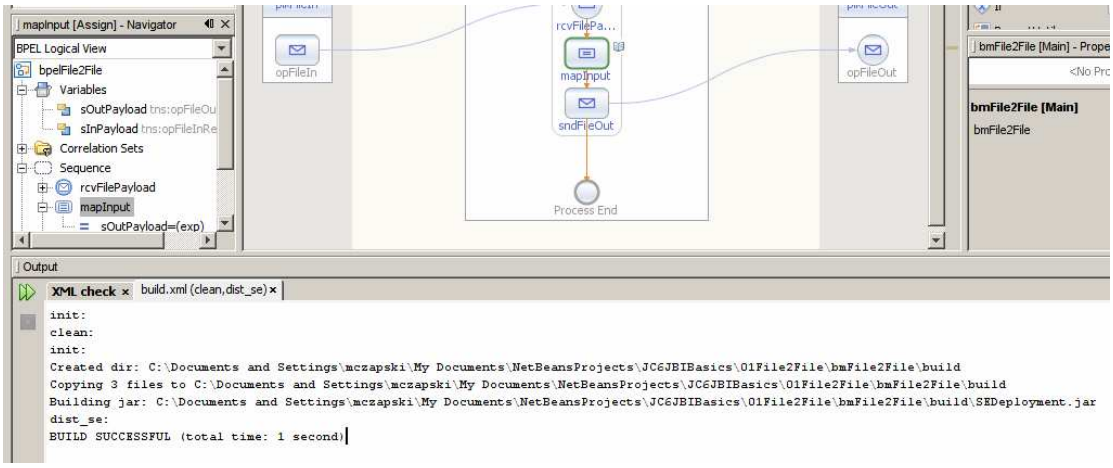

Figure 3-45 Feedback messages following a successful build

JBI requires us to create a Composite Application which will contain the BPEL Module we just created and built. This composite application is the deployment unit in JBI much as an Enterprise Archive is a deployment using in Java EE.

Let's create a new Composite Application project, caFile2File, as illustrated in Figure 3-46 through 3-48.

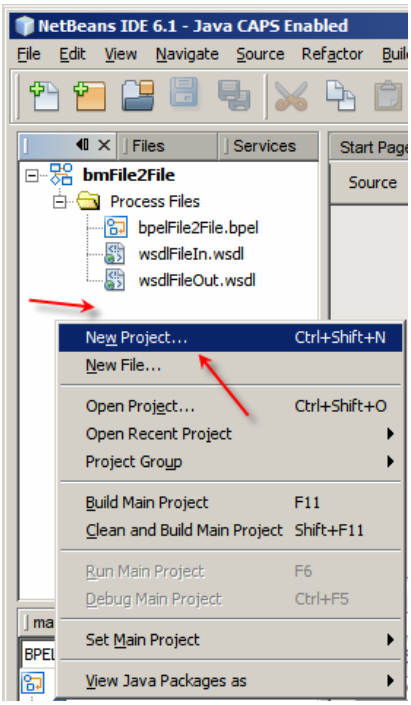

Figure 3-46 Choosing New Project menu option

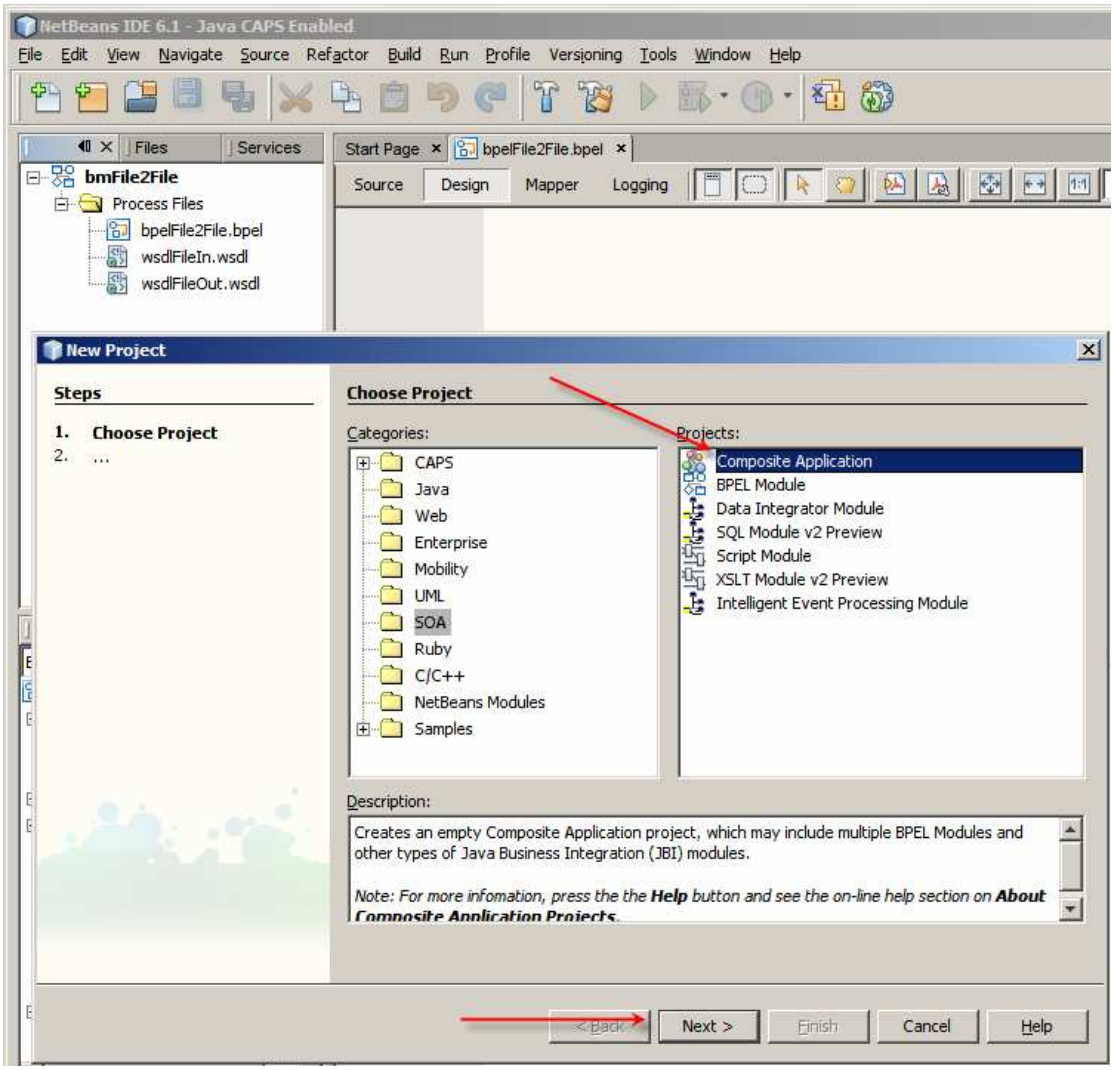

Figure 3-47 Choosing SOA -> Composite Application project

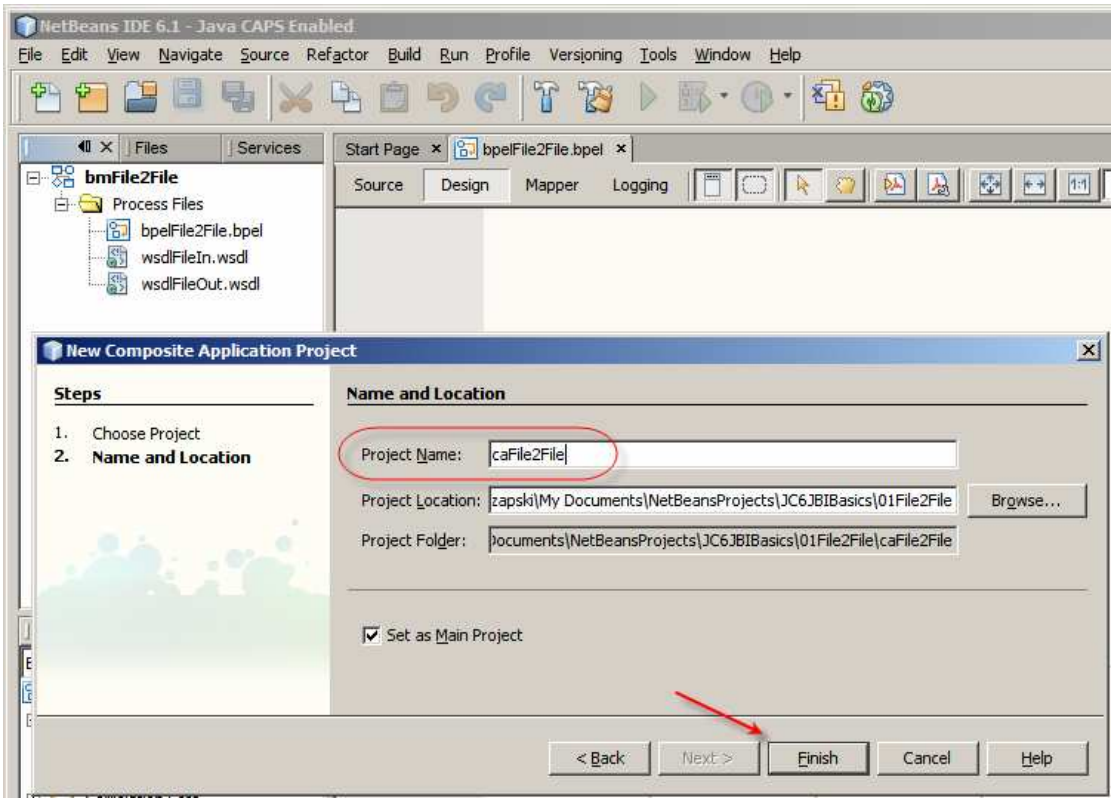

Figure 3-48 Naming the Composite Application project

Once the new Composite Application project with an empty Service Assembly is created we need to add the BPLE module to the JBI Modules part of its canvas. Figure 3-49 illustrates the process of dragging the BPEL Module project onto the Service Assembly canvas.

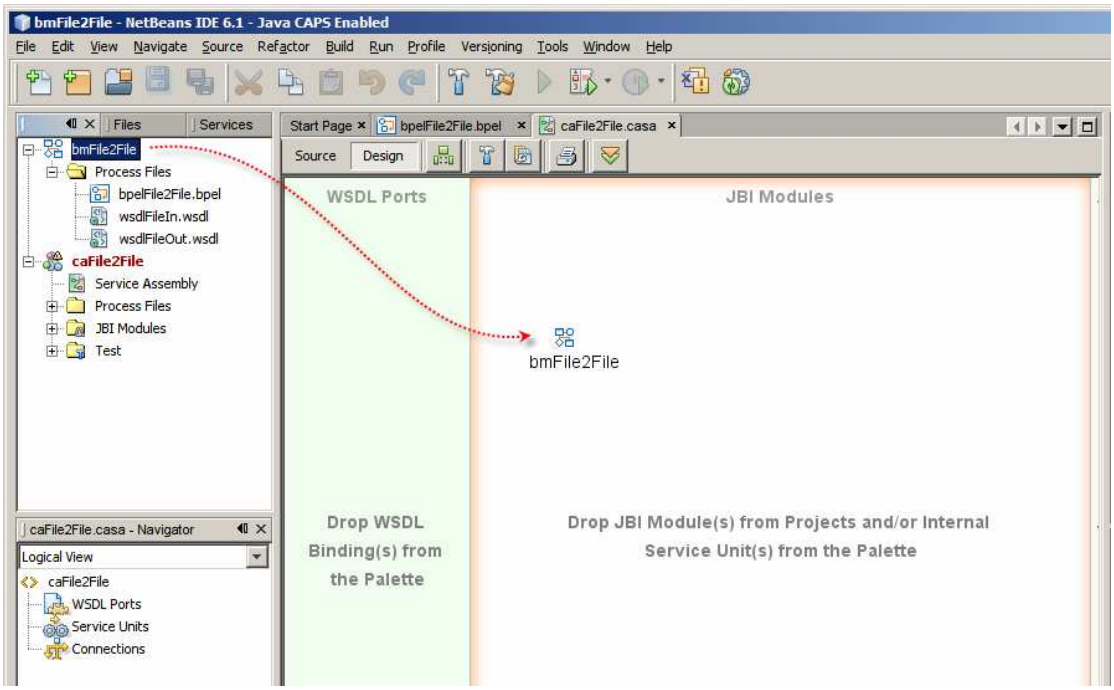

Figure 3-49 Adding a BPEL Module to the Service Assembly

Building the Service Assembly, see Figure 3-50, adds Binding Components and connections to the Service Assembly and produces a deployable service assembly.

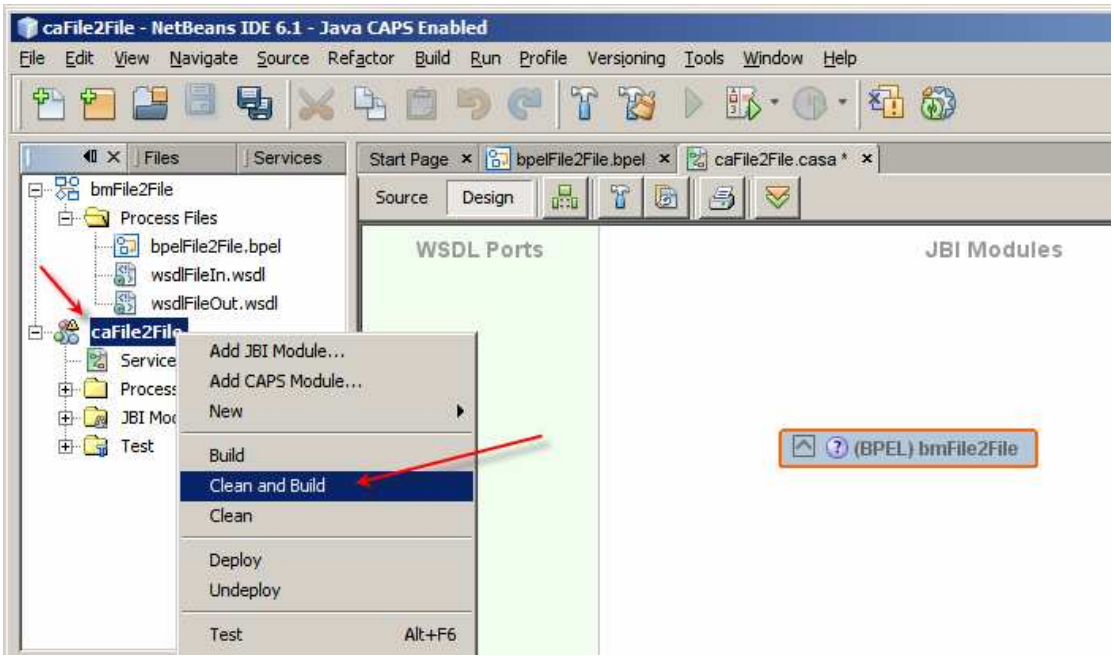

Figure 3-50 Building the Service Assembly

The Service Assembly is now deployed to the Application Server as illustrated in Figure 3-51. This assumes that the Application Server is configured, which by default it is. If the Application Server is not started the deployment process will cause it to be started and the service assembly to be deployed.

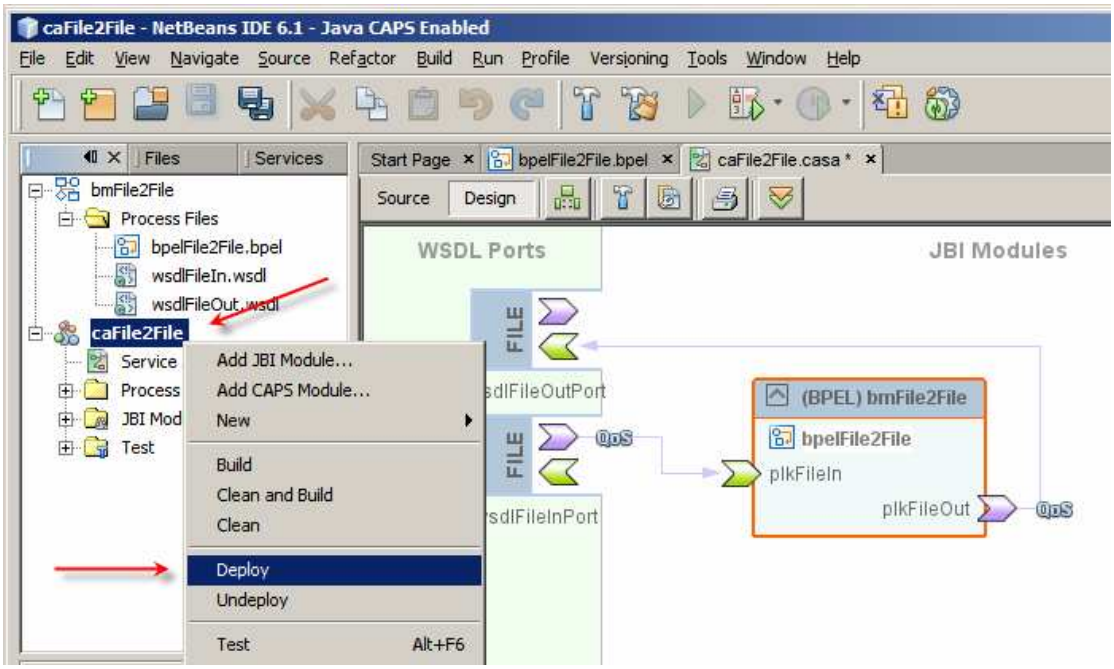

Figure 3-51 Deploying the Service Assembly to the Application Server

Figure 3-52 shows the feedback from a successful deployment of the service assembly we just developed.

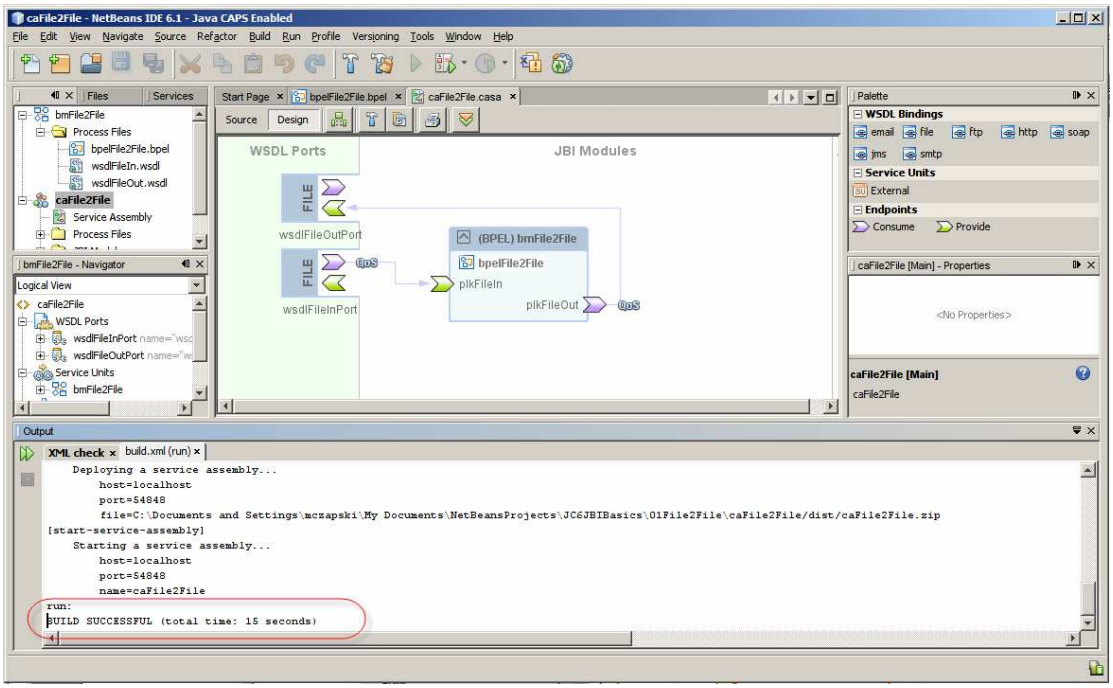

Figure 3-52 Feedback from successful deployment of the service assembly

With the service assembly / composite application deployed, we are now in a position to exercise the solution. Our File BC was configured to look for a file named test.txt in the directory C:\temp. Let's create such a file in that location with the sample content. Instead of creating the file test.txt we create a file with a different name, for example text.txt.~in, edit it to add the content of our choosing, then rename it to the name the BC is looking for. Figures 3-53 and 3-54 illustrate the steps.

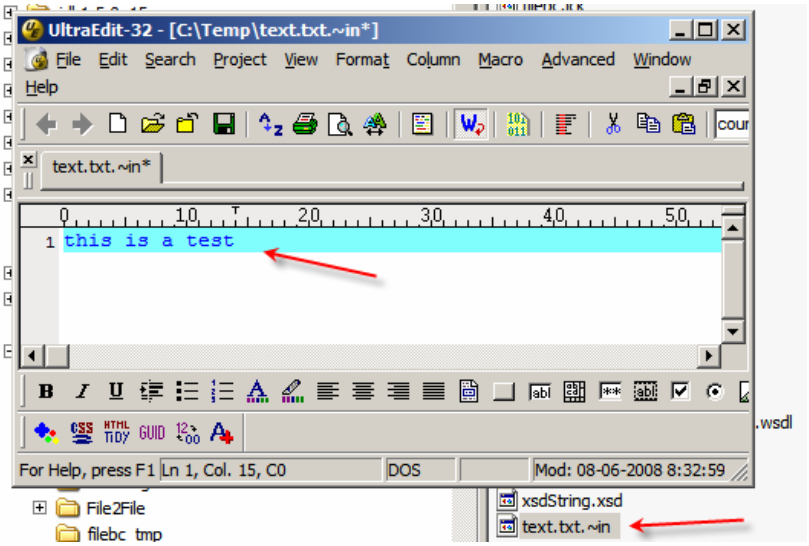

Figure 3-53 Creating a sample input file with a name different from the one the BC is looking for

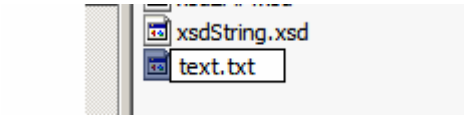

Figure 3-54 Renaming the file so the File BC can find it

Once the File BC pickled up the file the BPEL Process will process its content, convert it to upper case and pass the content to the outbound File BC for writing to

the output directory. The outbound File BC is configured to write a file named text_%d.out to C:\temp. AT runtime %d will get replaced by a number. Figure 3-55 shows the directory containing the output file and Figure 3-56 shows the content converted to the upper case.

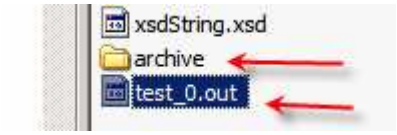

Figure 3-55 Output file

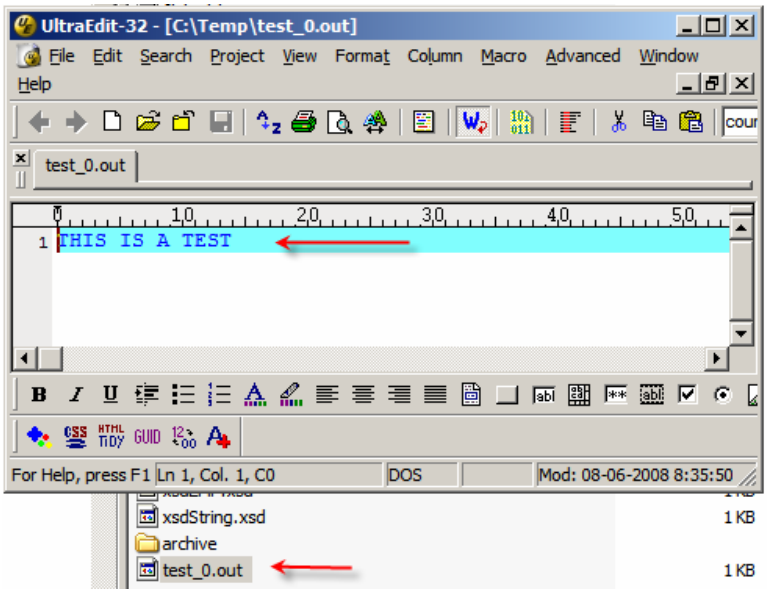

Figure 3-56 Content of the output file

The input file was renamed, by appending e GUID and the literal "_processed" to the original file name and was moved to the archive directory under the original directory in which the file was located. Figure 3-57 shows this.

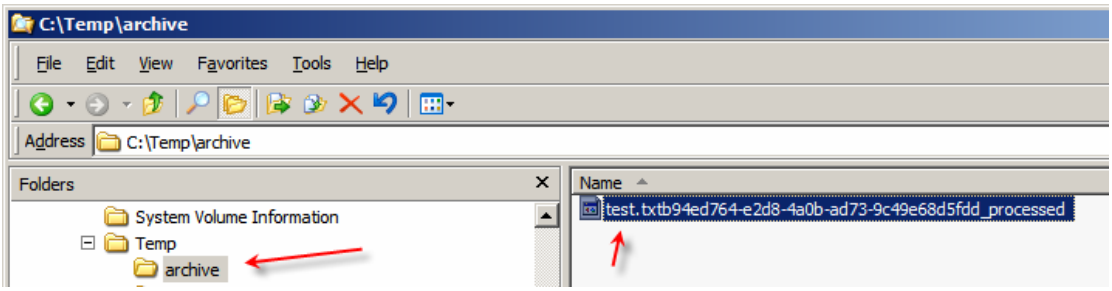

Figure 3-57 Input file archived after processing

Once the service assembly has been deployed its runtime properties can be inspected and it can be managed through the NetBeans IDE. Figures 3-58 and 3-59 illustrate this.

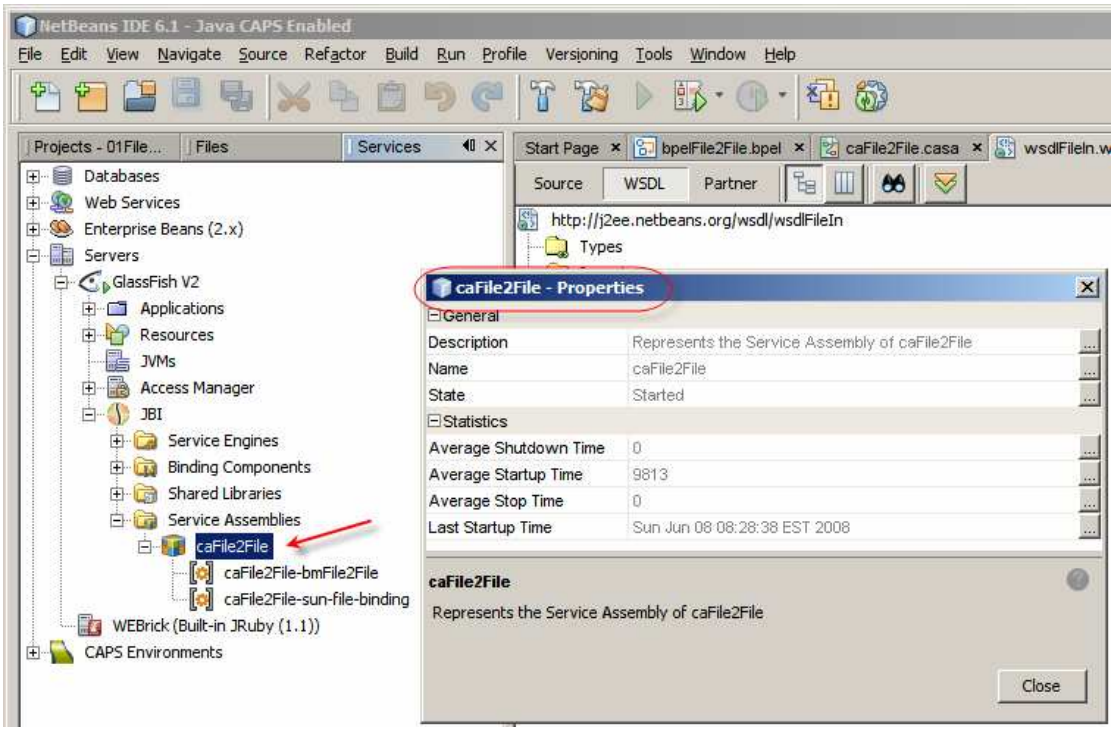

Figure 3-58 Service Assembly properties

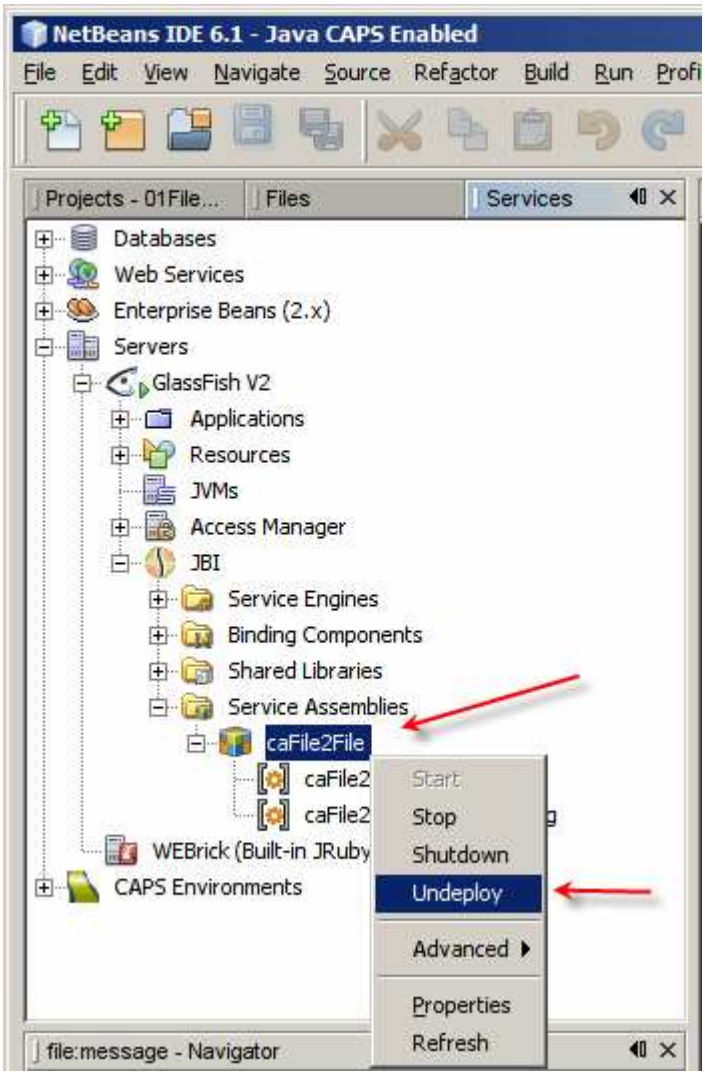

Figure 3-59 Undeploying the Service Assembly

4 Summary

This document walked the reader, step-by-step, through the process of creating, building, deploying and exercising a Java CAPS 6/JBI (or OpenESB) basic File to BPEL to File integration solution.