V 7.6 Tutorial

Remote Information Access

Progress Sonic Workbench Online Help Tutorial Instructions in PDF Format

Progress® Sonic ESB® Product Family V7.6 Tutorial
Remote Information Access

© 2008 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

A (and design), Actional, Actional (and design), Affinities Server, Allegrix, Allegrix (and design), Apama, Business Empowerment, ClientBuilder, ClientSoft, ClientSoft (and Design), Clientsoft.com, DataDirect (and design), DataDirect Connect, DataDirect Connect64, DataDirect Connect OLE DB, DataDirect Technologies, DataDirect XQuery, DataXtend, Dynamic Routing Architecture, EasyAsk, EdgeXtend, Empowerment Center, Fathom, IntelliStream, Neon, Neon New Era of Networks, O (and design), ObjectStore, OpenEdge, PeerDirect, Persistence, POSSENET, Powered by Progress, PowerTier, ProCare, Progress, Progress DataXtend, Progress Dynamics, Progress Business Empowerment, Progress Empowerment Center, Progress Empowerment Program, Progress Fast Track, Progress OpenEdge, Progress Profiles, Progress Results, Progress Software Developers Network, ProVision, PS Select, SequeLink, Shadow, ShadowDirect, Shadow Interface, Shadow Web Interface, ShadowWeb Server, Shadow TLS, SOAPStation, Sonic ESB, SonicMQ, Sonic Orchestration Server, Sonic Software (and design), SonicSynergy, SpeedScript, Stylus Studio, Technical Empowerment, Voice of Experience, WebSpeed, and Your Software, Our Technology-Experience the Connection are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and/or other countries. AccelEvent, Apama Dashboard Studio, Apama Event Manager, Apama Event Modeler, Apama Event Store, AppsAlive, AppServer, ASPen, ASP-in-a-Box, BusinessEdge, Cache-Forward, DataDirect Spy, DataDirect SupportLink, DataDirect XML Converters, Future Proof, Ghost Agents, GVAC, Looking Glass, ObjectCache, ObjectStore Inspector, ObjectStore Performance Expert, Pantero, POSSE, ProDataSet, Progress ESP Event Manager, Progress ESP Event Modeler, Progress Event Engine, Progress RFID, PSE Pro, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Sonic, Sonic Business Integration Suite, Sonic Process Manager, Sonic Collaboration Server, Sonic Continuous Availability Architecture, Sonic Database Service, Sonic Workbench, Sonic XML Server, The Brains Behind BAM, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries. Vermont Views is a registered trademark of Vermont Creative Software in the U.S. and other countries. IBM is a registered trademark of IBM Corporation. JMX and JMX-based marks and Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners.

Third Party Acknowledgements:

SonicMQ and Sonic ESB Product Families include code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

SonicMQ and Sonic ESB Product Families include the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

SonicMQ and Sonic ESB Product Families include software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright 2000-2001 ModelObjects Group. All rights reserved. The name "ModelObjects" must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called "ModelObjects", nor may "ModelObjects" appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

SonicMQ and Sonic ESB Product Families include files that are subject to the Netscape Public License Version 1.1 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright 1998-1999 Netscape Communications Corporation. All Rights Reserved.

SonicMQ and Sonic ESB Product Families include versions 8.3 and 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which is available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a "Contributor". Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a "Contributor". A small number of modules, or enhancements to modules, have been developed by other individuals (either written specially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

SonicMQ and Sonic ESB Product Families include software developed by IBM. Copyright 1995-2002 and 1995-2003 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

SonicMQ and Sonic ESB Product Families include software Copyright © 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

SonicMQ and Sonic ESB Product Families includes software developed by the University Corporation for Advanced Internet Development <http://www.ucaid.edu>Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

Portions of SonicMQ and Sonic ESB Product Families were created using JThreads/C++ by Object Oriented Concepts, Inc.

SonicMQ and Sonic ESB Product Families were developed using ANTLR

SonicMQ and Sonic ESB Product Families include software Copyright © 1991-2007 DataDirect Technologies Corp. All rights reserved. This product includes DataDirect products for the Microsoft SQL Server database which contain a licensed implementation of the Microsoft TDS Protocol.

SonicMQ and Sonic ESB Product Families include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright (c) 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

February 2008

Remote Information Access tutorial

As applications grow in complexity and become widely distributed, it becomes increasingly important to develop business processes that reliably retrieve data from remote data sources. Progress Sonic ESB enables you to develop ESB processes to retrieve and aggregate data from multiple back-end data sources, and provides additional functionality such as content-based routing and data transformation. With Progress Sonic ESB, you can develop processes to:

- Handle multiple requests from a single initiating point, such as a portal.
- Send the same event to multiple back end sources, taking into account issues such as data format changes.
- Cache results near the portal to minimize back-end traffic by reusing data already collected, but without placing a burden on the portal to store data in memory.

The Remote Information Access tutorial demonstrates how ESB processes can be developed and implemented to address these issues. You can go through the entire tutorial step by step, or, if you prefer, you can work through some of the phases of the tutorial yourself, then go directly to running and testing the fully implemented ESB process using the completed sample processes and resources included with the tutorial.

It might take you up to three hours to work through the entire Remote Information Access tutorial. The following times for each part of the tutorial are only estimates; you might complete them in less time:

- Preparing to develop the Remote Information Access sample application — 20 minutes
- Phase 1: Creating the prototype ESB process, processRequest — 15 minutes
- Phase 2: Implementing multiple operations using a content-based router — 30 minutes
- Phase 3: Implementing getAccounts using a Split and Join Parallel service — 25 minutes
- Phase 4: Using stylesheets to format responses — 15 minutes
- Phase 5: Implementing getAccountActivity using content-based routing — 30 minutes
- Testing the fully implemented ESB process, processRequest — 15 minutes

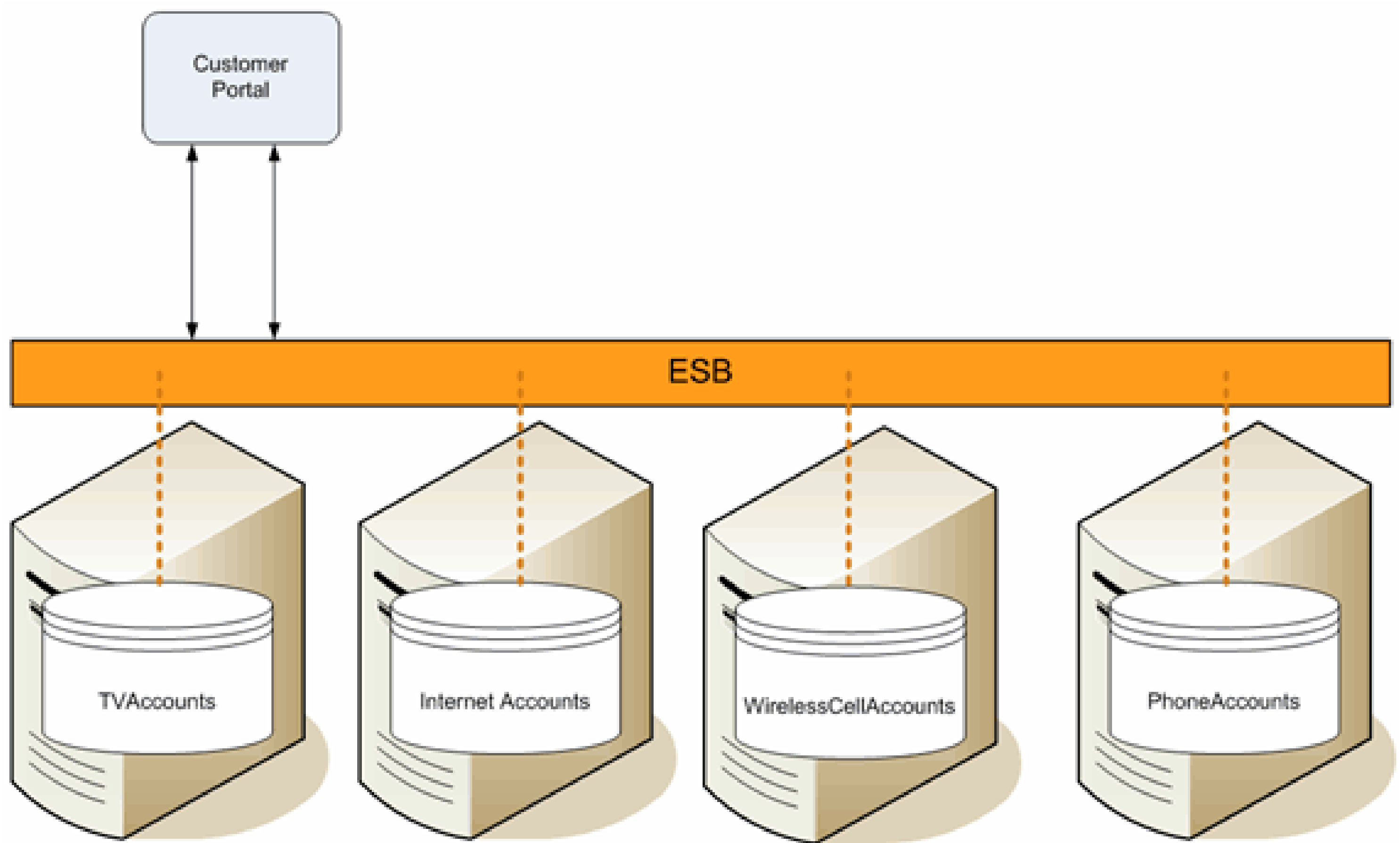
The Remote Information Access tutorial is available in the following formats:

- Online help — Click the Tutorials link on the Sonic Workbench Welcome page or find the tutorial in the *Progress Sonic ESB Product Family: Developer's Guide* (Sonic Workbench online help) under "Progress Sonic ESB Samples and Tutorials."
- PDF — Click the link to the PDF from the Documentation page.
- Demonstration — Click the link to the video from the Documentation page.

Next, look at the Remote Information Access scenario used in this tutorial.

Remote Information Access scenario

The following figure shows a typical Remote Information Access scenario. This scenario demonstrates using distributed queries to aggregate information across multiple back-end data sources:



In the Remote Information Access tutorial, a customer service representative of a simulated telecommunications company is required to get information about customer accounts and activity on those accounts. Customers can have multiple types of accounts, including TV, Wireless Cell, and Phone. Each account application is deployed on a separate server. These accounts can be different applications or databases. The tutorial demonstrates how Progress Sonic ESB is used to create a unified view of the data stored in each database. The tutorial addresses two use cases:

- **Use Case 1: Get Accounts** — A request is made to retrieve a list of all the accounts for a specified customer. To retrieve all the account information, the request is sent simultaneously to all TV, Wireless Cell, and Phone account databases. When all the information has been retrieved, the data is returned in a single message.
- **Use Case 2: Get Account Activity** — A request is made to get the account activity for a specified account. In this case, data is retrieved from one of the databases, based on the account type specified in the request.

The tutorial implements an ESB process that handles these use cases by routing messages through different branches of the process based on the type of request sent to the process.

Next, look at the Remote Information Access process you will develop and implement in this tutorial.

Remote Information Access process

In this tutorial, you take the role of an architect who has to design the ESB component of the Remote Information Access application. As the architect, you will first prototype the interface you want to expose to your customers. You will do this by testing the interface with sample documents you have created with your customer. Next, you will determine how to move these requests to the applications that supply the data, then you will define the interfaces you want these applications to provide. Ultimately, you will use these interfaces to define the actual services you want the applications to provide over the ESB.

In working through the tutorial, you will do the following:

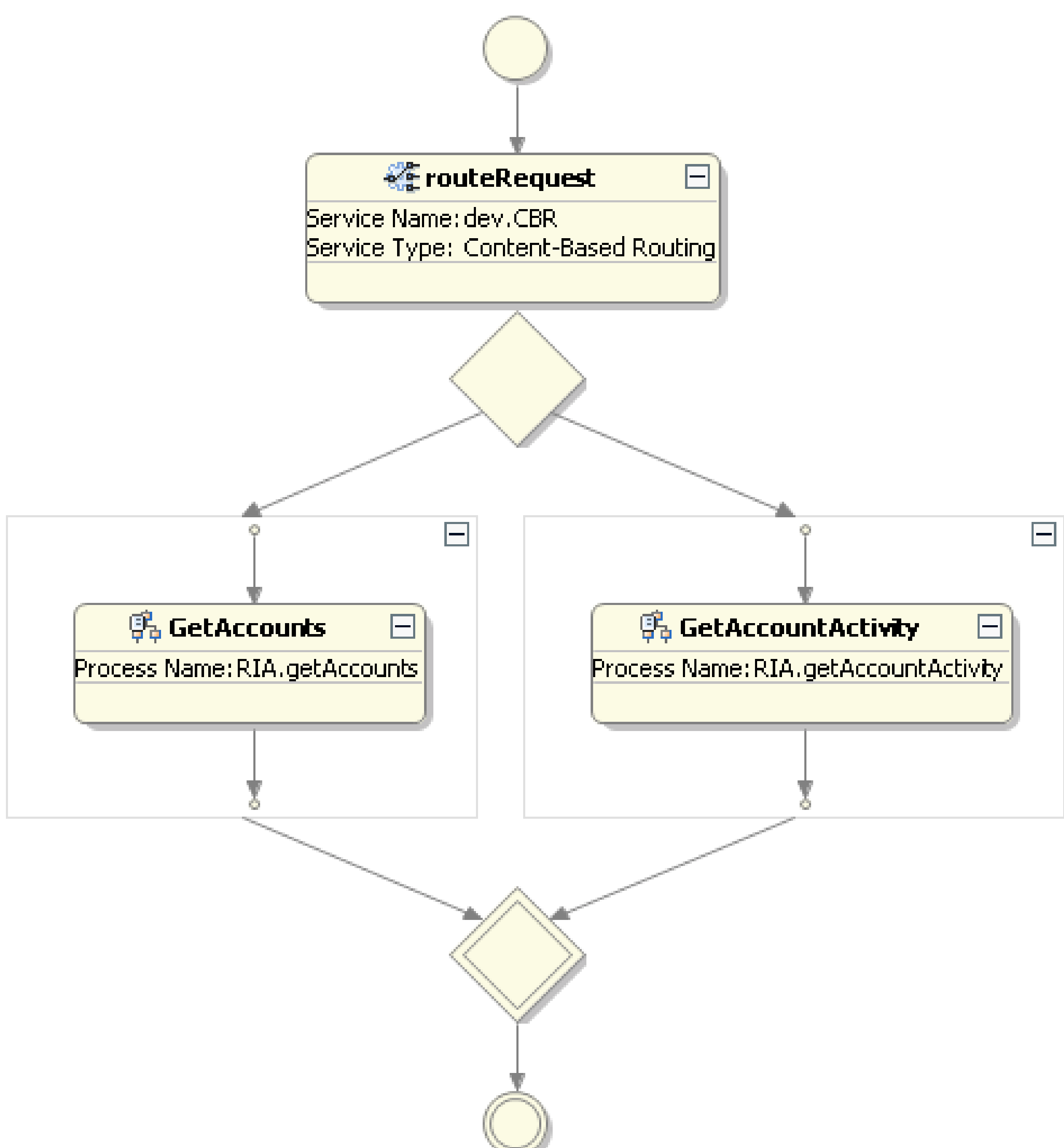
1. Implement the interface you expect to provide over the ESB
2. Design an ESB process to route the specific requests to the remote information data stores

In this tutorial, you develop an ESB process, **processRequest**, to handle both use cases of the Remote Information Access scenario. The processRequest process includes two branches, one for each use case. A content-based router sends incoming requests to the appropriate branch based on the request type in the incoming message. Each branch includes a subprocess that returns the requested information.

By using subprocesses, you allow each subprocess to perform one logical step of the solution. Subprocesses can be used in one or more service invocations, and in the ESB there is no overhead for going from one process to another. In the Remote Information Access application, each subprocess handles a different use case. By placing the steps to handle each use case within a subprocess for that use case, it is possible to a reuse each subprocess in another context. This technique also increases the readability of the main process. In subsequent iterations of the project, the subprocesses can be changed without having to redo the main process.

In the fully implemented processRequest process, shown below, the following routing takes place:

1. Incoming requests to **processRequest** are sent to the content-based router, **routeRequest**, which applies XPath routing rules to evaluate the request type, either **Get Accounts** (use case 1), or **Get Account Activity** (use case 2).
2. Requests to **Get Accounts** are sent to the branch that includes the getAccounts subprocess, which returns a list of accounts for the customer specified in the request.
3. Requests to **Get Account Activity** are sent to the branch that includes the getAccountActivity subprocess, which returns a summary of account activity for the specified account type.



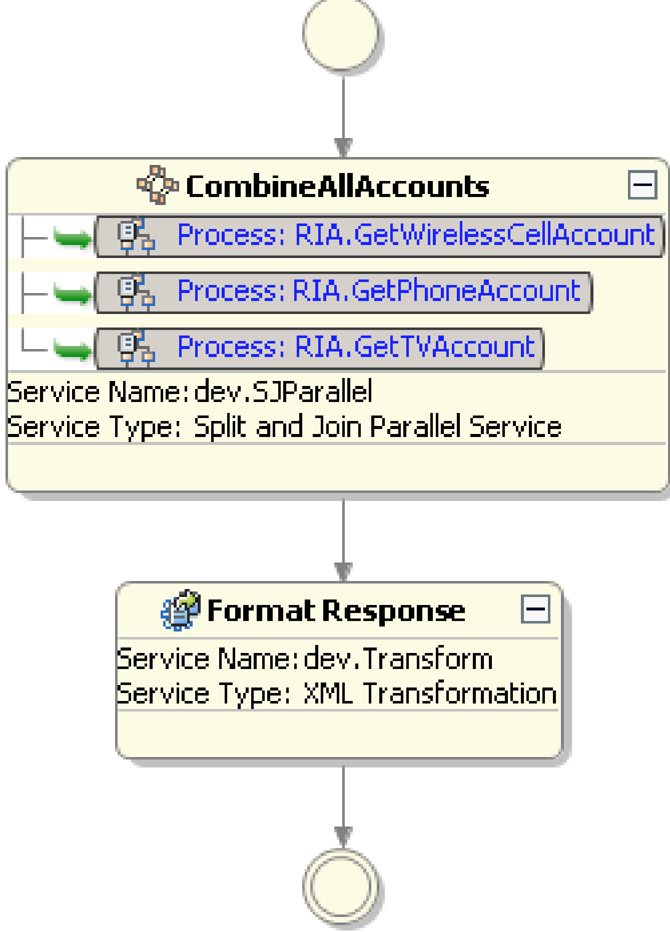
Next, look at the getAccounts subprocess.

getAccounts subprocess

The branch of processRequest that handles requests for customer accounts (use case 1) contains the getAccounts subprocess. Requests routed to this branch of processRequest are handled as follows:

1. The request is sent to a Split and Join Parallel service, **CombineAllAccounts**, that simultaneously calls the databases for each account type to return data from each account. The service aggregates that data into a single response.
2. The response is formatted by an XML Transformation service, **Format Response**, that uses an XSLT stylesheet to reformat the XML into the desired response document.

The fully implemented getAccounts subprocess looks like this:

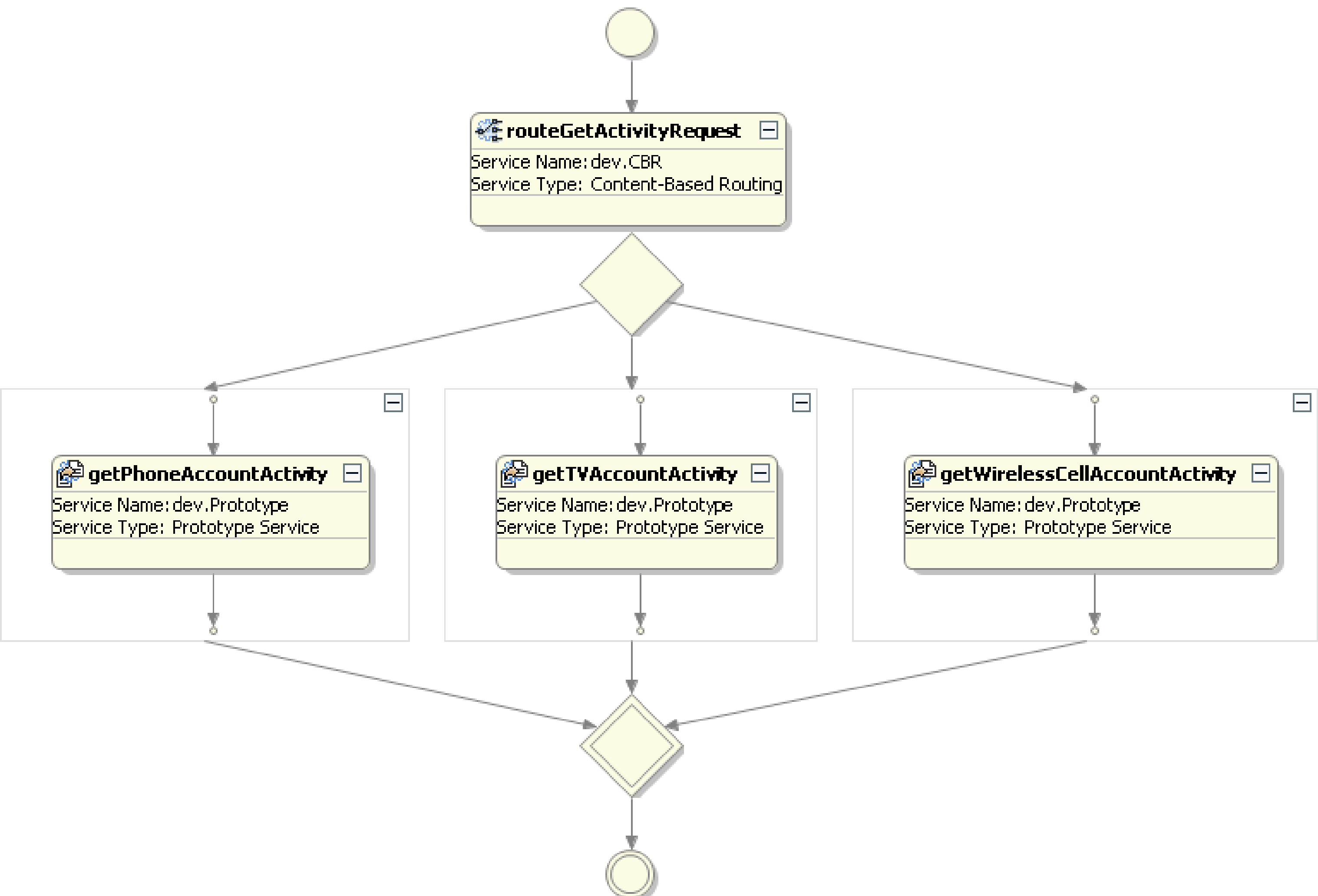


Next, look at the getAccountActivity subprocess.

getAccountActivity subprocess

The branch of processRequest that handles requests for account activity (use case 2) contains the getAccountActivity subprocess. Requests routed to this branch of processRequest are sent to another content-based router, **routeGetActivityRequest**. This router applies Xpath routing rules to evaluate the account type specified in the incoming request, then sends the request to a branch configured to return information for that account type.

The fully implemented getAccountActivity subprocess looks like this:



Next, see how the Remote Information Access tutorial uses a top-down, phased implementation to create the ESB process, processRequest, and its subprocesses.

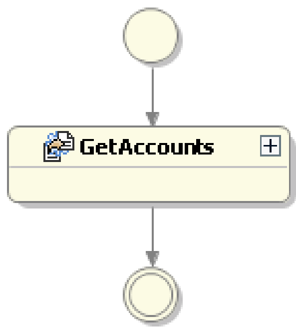
Phased implementation of the Remote Information Access sample application

The Remote Information Access tutorial demonstrates top-down design, using phased implementation through prototype steps. This approach enables you to develop and execute the itinerary for your business process using prototype steps, then gradually replace those prototypes with services or subprocesses that implement the steps. This top-down, phased approach is used to create an ESB process to handle the two use cases in the Remote Information Access scenario.

This tutorial develops the ESB process named processRequest, and its subprocesses, in five phases of implementation:

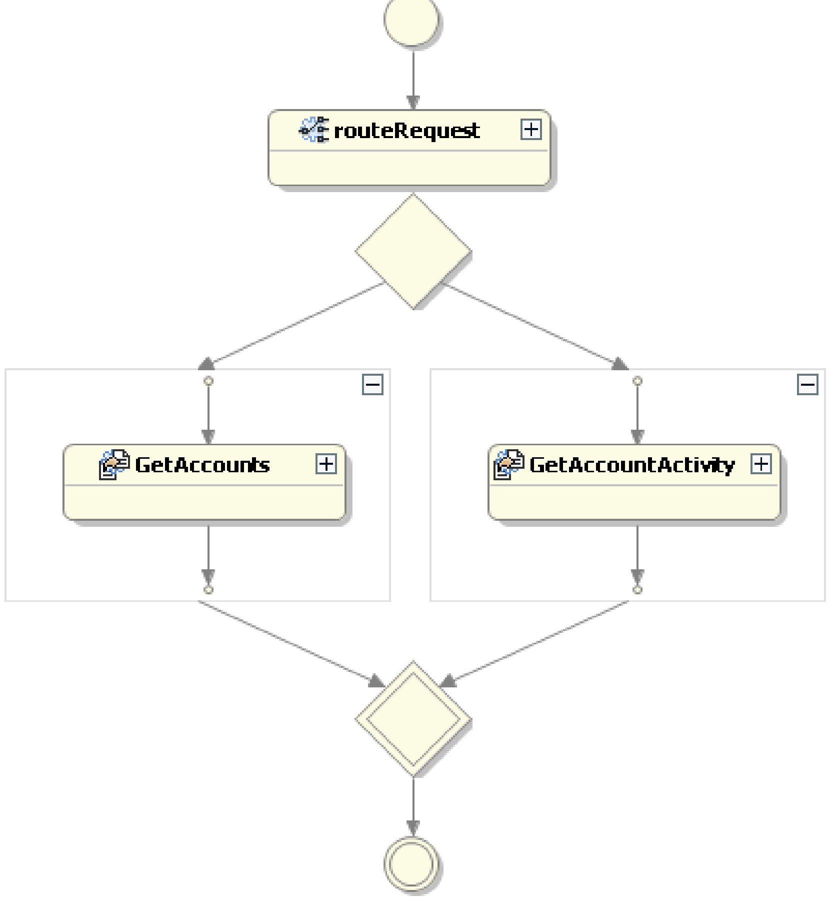
Phase 1. Create the prototype ESB process:

- Create the ESB process, **processRequest**, containing the Prototype service, **GetAccounts**.
- Test the interface with the ESB by sending a request and receiving a response.



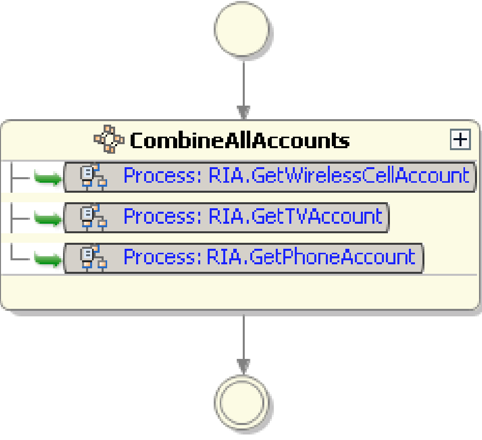
Phase 2. Implementing multiple operations using a content-based router:

- Create an content-based router, **routeRequest**, to route messages based on the operation (either GetAccounts or GetAccountActivity).
- Create a prototype branch for each use case.
- Test the routing with scenarios for each use case.



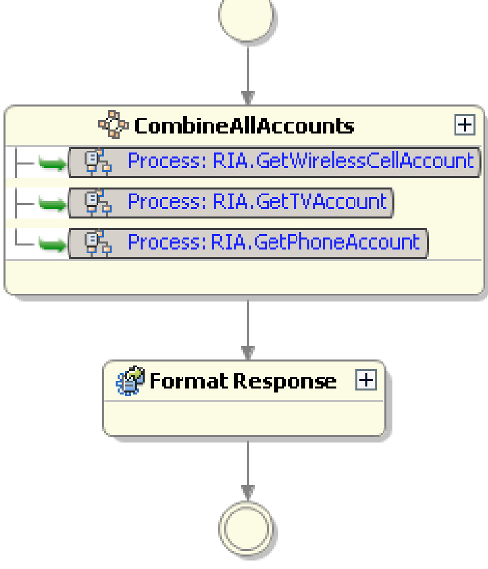
Phase 3. Implement the getAccounts subprocess:

- Create the **getAccounts** subprocess to handle the Get Accounts use case.
- Add a Split and Join Parallel service, **CombineAllAccounts**, to simultaneously retrieve data from different accounts and aggregate the data into a single response.
- Test the subprocess with a scenario.



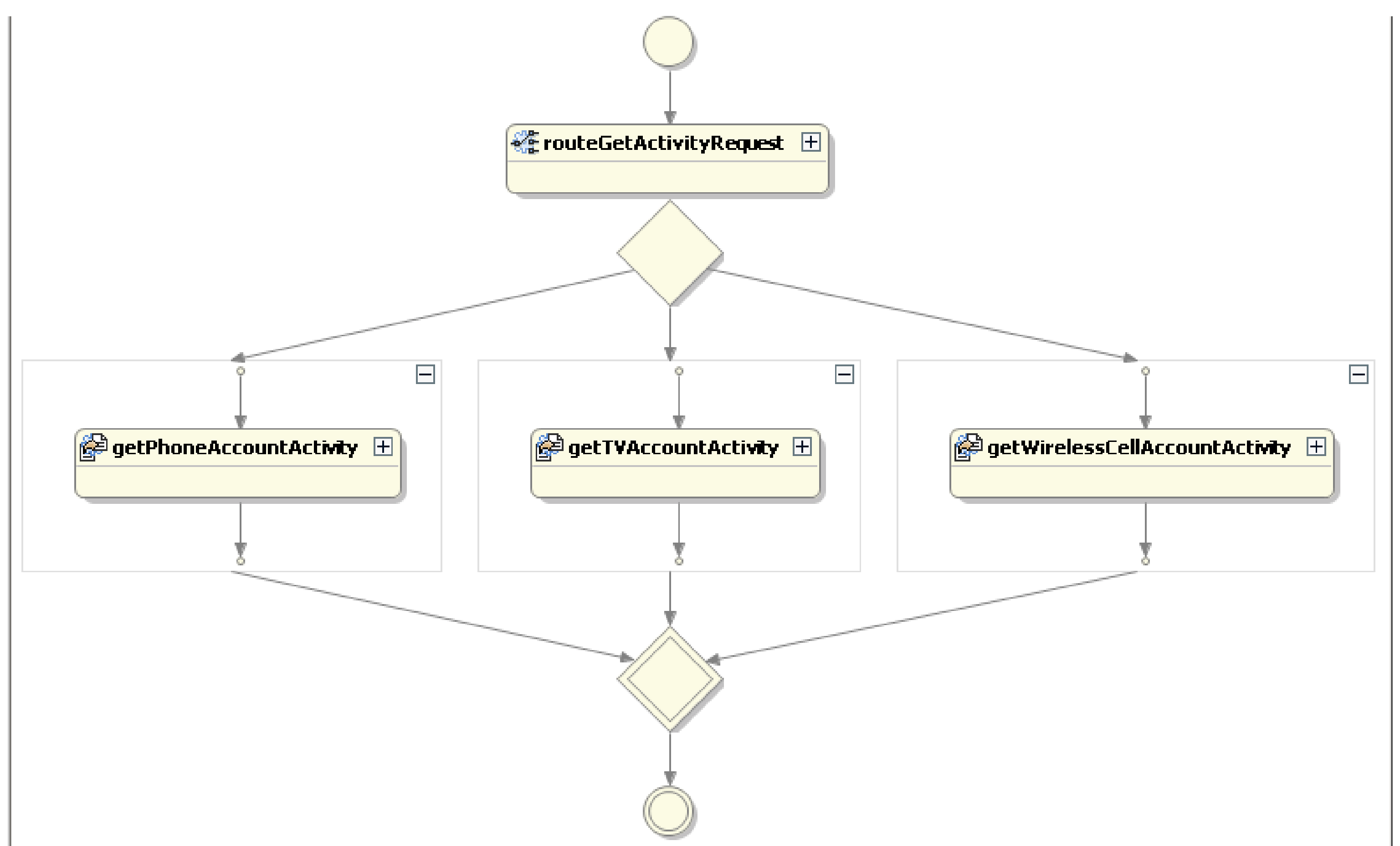
Phase 4. Transform the response in getAccounts:

- Add an XML Transformation service, **Format Response**, to format the response from the Split and Join Parallel service.
- Test the transformation stylesheet with a scenario.
- Test the fully implemented subprocess with a scenario.



Phase 5. Implement the getAccountActivity subprocess:

- Create the **getAccountActivity** subprocess to handle the Get Account Activity use case.
- Add a content-based router, **routeGetActivityRequest**, to route requests based on the specified account type.
- Configure three branches of the content-based router, one for each account type.
- Test the routing with scenarios for each account type.



When you have completed all five implementation phases, you will be ready to test the fully implemented processRequest process.

To get started, prepare to develop the Remote Information Access sample application.

Preparing to develop the Remote Information Access sample application

Before running the Remote Information Access sample application, you must:

1. Start Sonic Workbench.
2. Import the Remote Information Access sample project.
3. Examine the Remote Information Access sample project

Begin by starting Sonic Workbench.

Starting Sonic Workbench

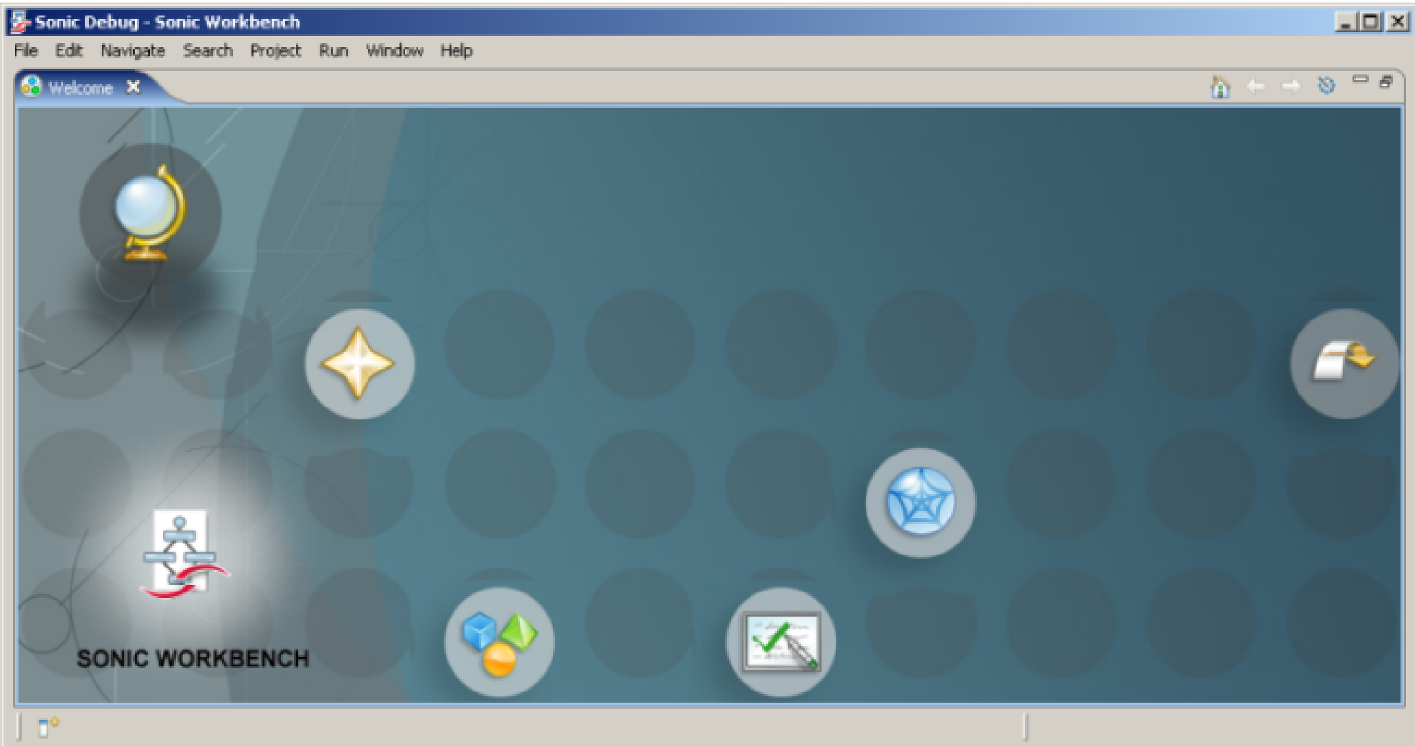
To start Sonic Workbench:

1. Select **Start > Programs > Progress > Sonic 7.6 > Start Domain Manager**:

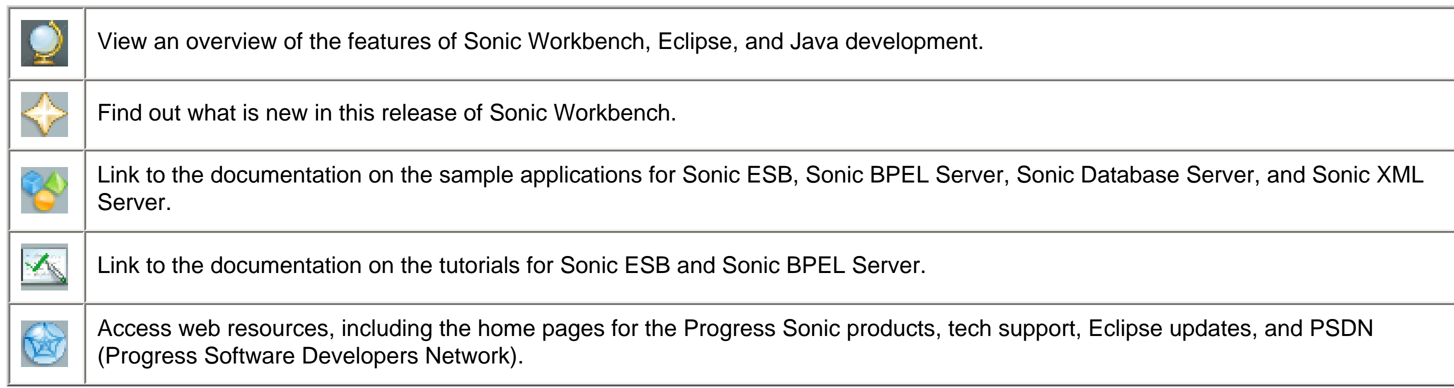
   A console window opens showing that Sonic Workbench is starting the domain manager, configuration repository, and development containers.

2. Select **Start > Programs > Progress > Sonic 7.6 > Workbench**:

   The Sonic Workbench Welcome screen opens:



3. Click the icons on the Welcome screen to see how you can access the following information:



	View an overview of the features of Sonic Workbench, Eclipse, and Java development.
	Find out what is new in this release of Sonic Workbench.
	Link to the documentation on the sample applications for Sonic ESB, Sonic BPEL Server, Sonic Database Server, and Sonic XML Server.
	Link to the documentation on the tutorials for Sonic ESB and Sonic BPEL Server.
	Access web resources, including the home pages for the Progress Sonic products, tech support, Eclipse updates, and PSDN (Progress Software Developers Network).

4. Click to close the Welcome screen and start using Sonic Workbench.

Note: You can reopen the Welcome screen by selecting **Help > Welcome**.

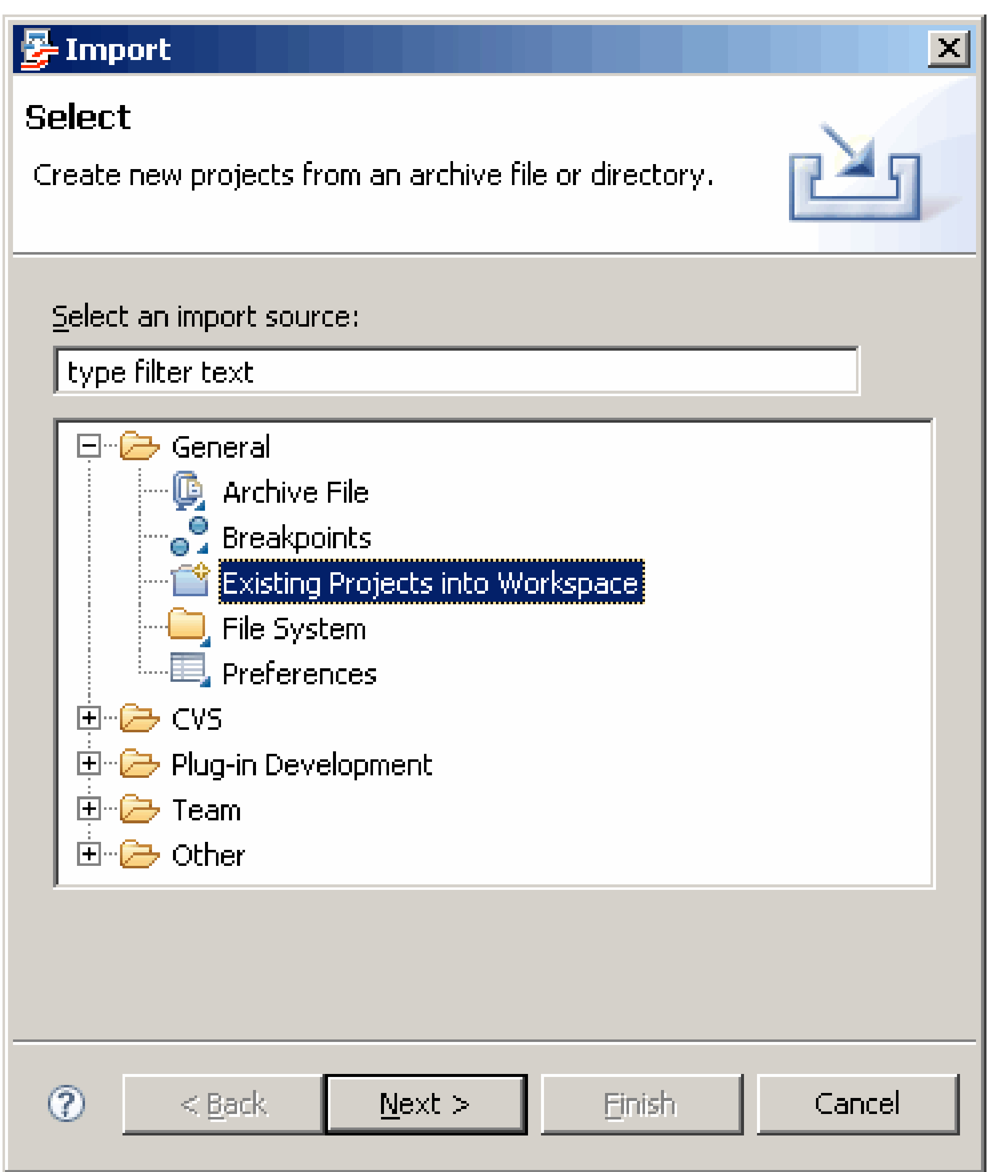
Next, import the Remote Information Access sample project.

Importing the Remote Information Access sample project

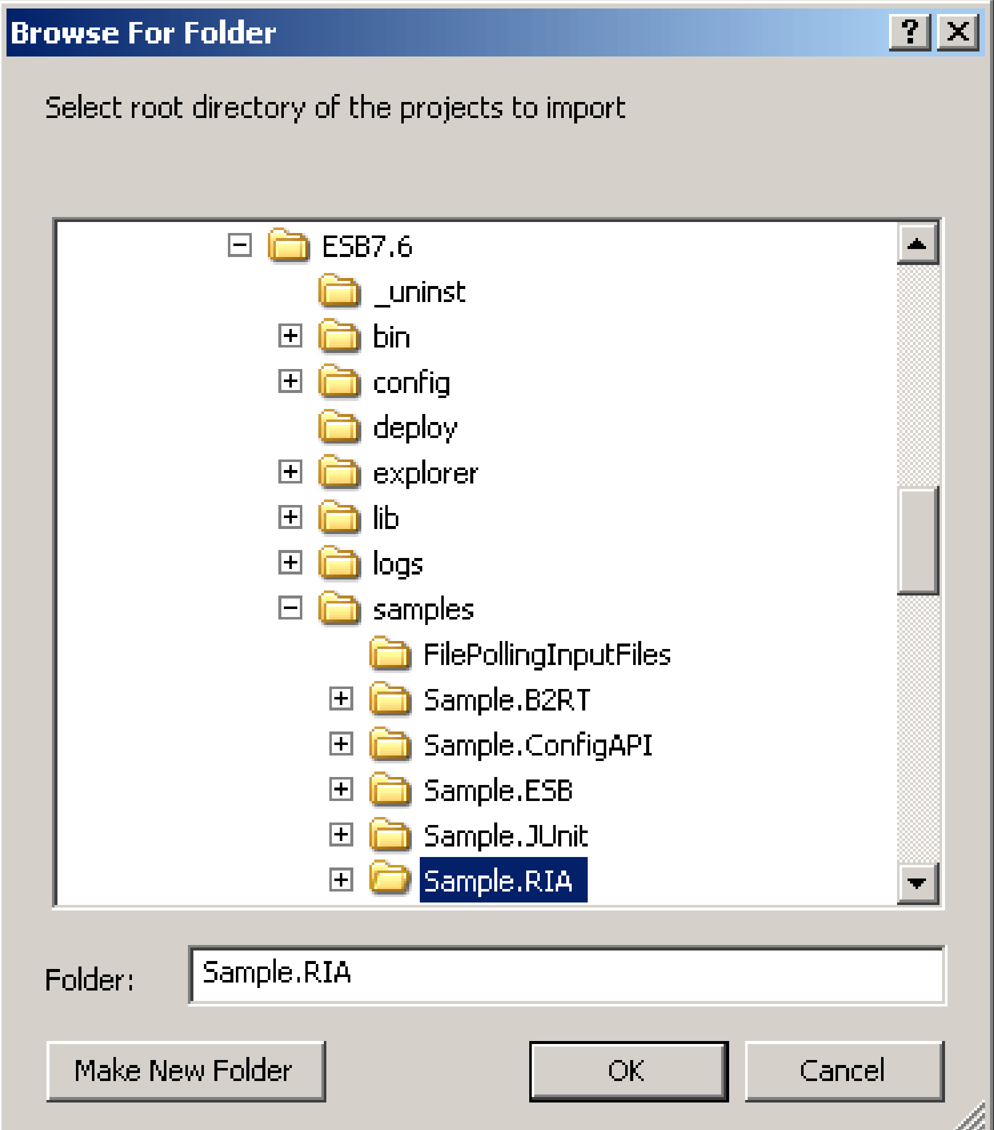
To import the sample project for the Remote Information Access tutorial:

1. After closing the Welcome screen, you are ready to use Sonic Workbench in the Sonic Design perspective:

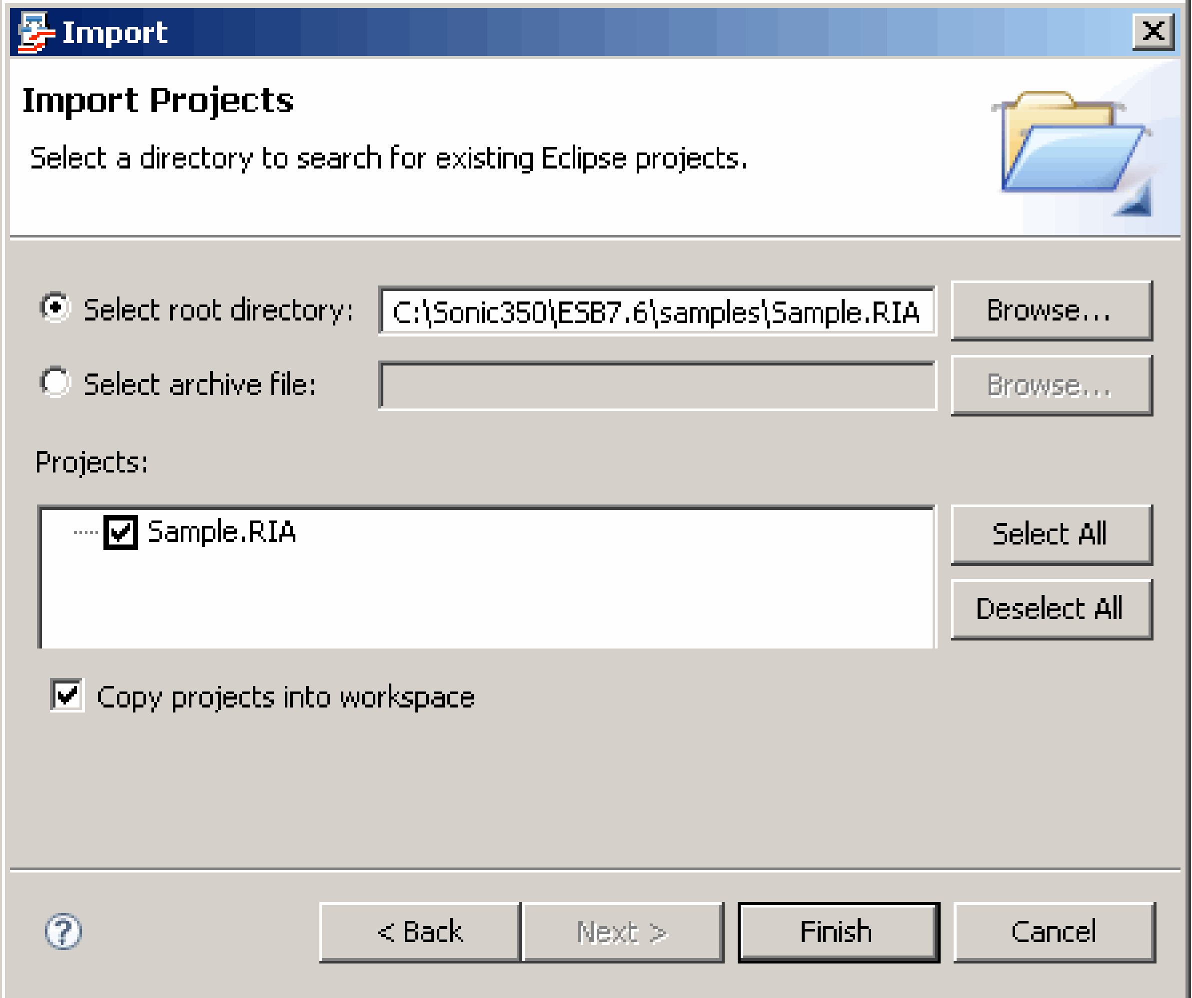
2. Select **File > Import**. The **Select** page of the **Import** wizard opens.
3. Select **General > Existing Projects into Workspace**:



4. Click **Next**. The **Import Projects** page of the **Import** wizard opens. Choose **Select root directory** and click **Browse**. The **Browse for Folder** dialog box opens.
5. Select the **Sample.RIA** folder under **Sonic > ESB7.6 > samples**:



6. Click **OK**. The **Import Projects** page of the **Import** wizard opens.
7. The Sample.RIA project is automatically checked.
   Be sure to check **Copy projects into workspace** (this option prevents you from changing the original project if you modify the imported project):



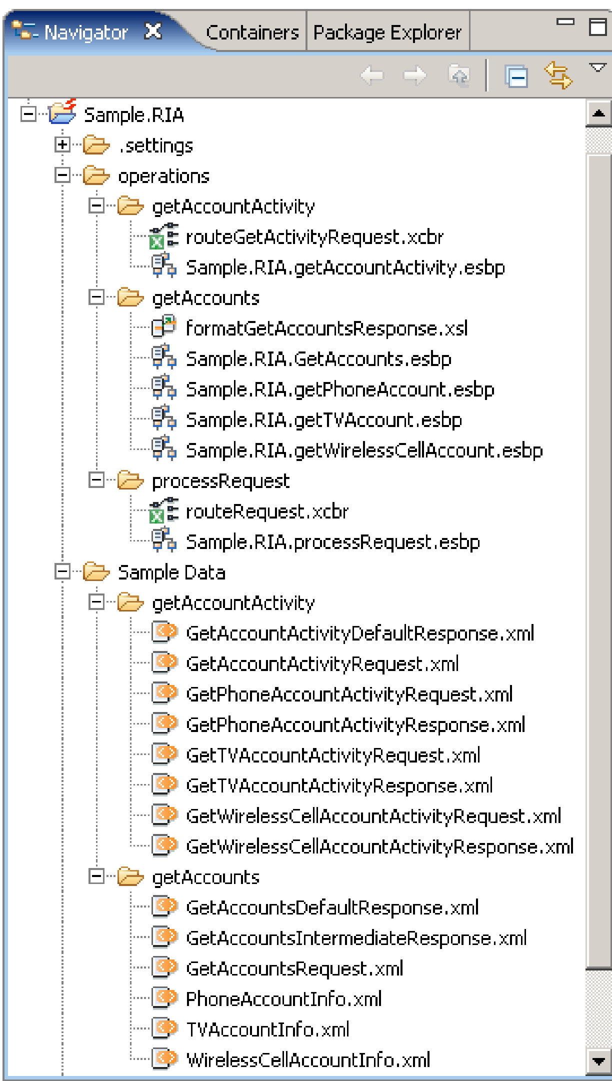
8. Click **Finish**. Sonic Workbench loads the Remote Information Access sample project.

Next, examine the Remote Information Access sample project.

Examining the Remote Information Access sample project

After you import the Sample.RIA project, you upload the project and examine its files:

1. Go to the Navigator view. Select the Sample.RIA project and select **Project > Upload all** from the menu bar to upload the project.
2. Click **OK** to confirm the uploading.
3. Expand the Sample.RIA folder and the \operations and \Sample Data subfolders to view the files in the Sample.RIA project:



4. You can double-click a file to view it in the appropriate Sonic Workbench editor.

You can learn more about the files in the Remote Information Access sample project now, or wait until you work with them in the tutorial.

Now you are ready to develop the Remote Information Access sample application.

Developing the Remote Information Access sample application

The Remote Information Access tutorial is divided into five phases of implementation and a testing and debugging section. In each phase, you start by creating a Prototype service and configuring it to return a simulated response. This enables you to test your design as you develop it. After confirming basic functionality, you replace the prototypes with services or subprocesses that implement the required functionality.

You can stop the tutorial at any time and come back to it. Just be sure to save the files you are working on.

The parts of the tutorial, and the estimated times to complete each part, are:

- **Phase 1.** Create a prototype interface ESB process that the client application will call, then test the interface with the ESB by sending a request and receiving a response. (15 minutes)
- **Phase 2.** Implement multiple operations using a content-based router to route messages based on the operation (either GetAccounts or GetAccountActivity). Create a prototype branch for each use case. Then test the routing with scenarios for each use case. (30 minutes)
- **Phase 3.** Create a subprocess to handle the Get Accounts use case. Add a Split and Join Parallel service to simultaneously retrieve data from different accounts and aggregate the data into a single response. Then test the subprocess with a scenario. (25 minutes)
- **Phase 4.** Add an XML Transformation service to format the response from the Split and Join Parallel service. Then test the transformation stylesheet and the fully implemented subprocess. (15 minutes)
- **Phase 5.** Create a subprocess to handle the Get Account Activity use case. Add a content-based router to route requests based on the specified account type. Configure three branches of the content-based router, one for each account type. Then test the routing with scenarios for each account type. (30 minutes)
- **Test and debug** the fully implemented ESB process, processRequest. (15 minutes)

Start by creating the prototype ESB process, processRequest.

Note: If you do not want to develop and implement all the phases of processRequest yourself, you can run and test the sample files included in Sample.RIA. The sample ESB processes are similar to the processes created and implemented in this tutorial, and include scenarios to run the processes. Simply open the sample ESB process you want to test and proceed directly to the instructions to run and test the process:

- Running and testing the getAccounts subprocess — The sample ESB process `Sample.RIA.getAccounts.esbp` is similar to the subprocess completed in Phases 3 and 4.
- Running and testing the getAccountActivity subprocess — The sample ESB process `Sample.RIA.getAccountActivity.esbp` is similar to the subprocess completed in Phase 5.
- Testing the fully implemented ESB process, processRequest — The sample ESB process `Sample.RIA.processRequest.esbp` is similar to the process completed in this tutorial.

Phase 1: Creating the prototype ESB process, processRequest

To begin the RIA tutorial you create your own project, then create and test the prototype ESB process, processRequest. In later phases of the tutorial you will implement additional functionality in the ESB process, but for now you simply create the prototype and test its interface to the ESB:

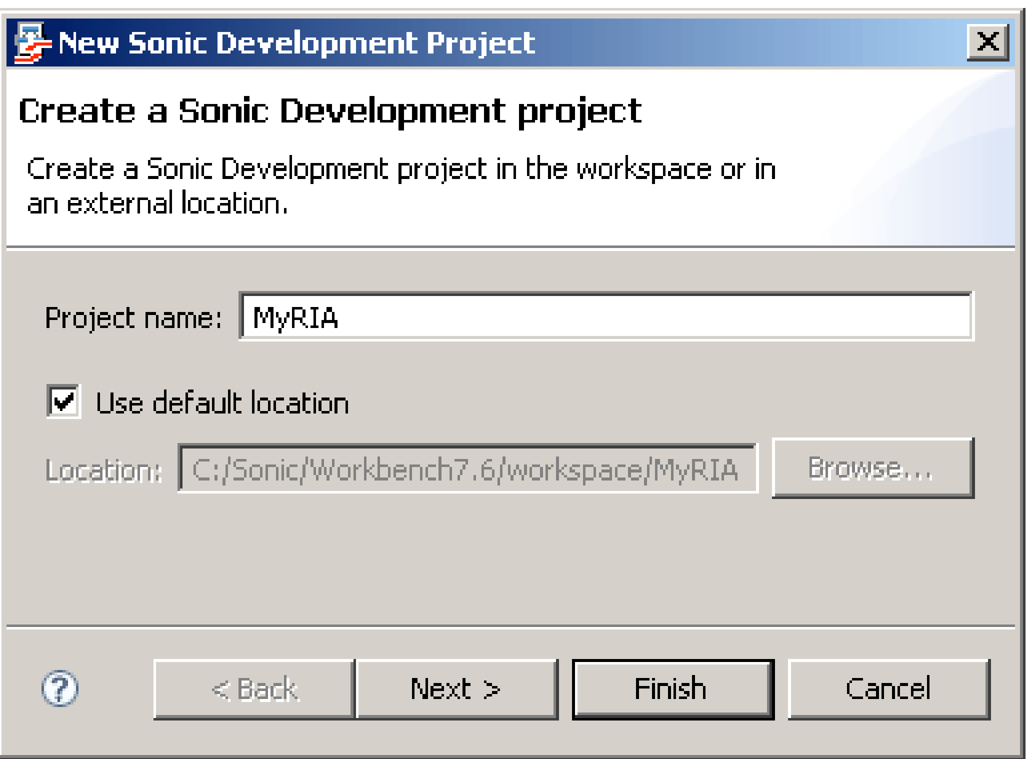
1. Create a new project — Create your own Sonic development project.
2. Copy the sample data — Copy the sample data into your newly created project.
3. Create the prototype ESB process — Create the processRequest ESB process.
4. View processRequest — View processRequest in the ESB Process editor and look at the palette options.
5. Create a scenario — Create a scenario to run processRequest.
6. Run and test processRequest — Use the scenario you created to run processRequest and verify that it returns a response.
7. Modify processRequest to return a response — Configure a default response for processRequest to simulate a response based on an incoming request.
8. Test the modified ESB process — Use the scenario to verify that processRequest returns a response based on the incoming request.

Start by creating a project.

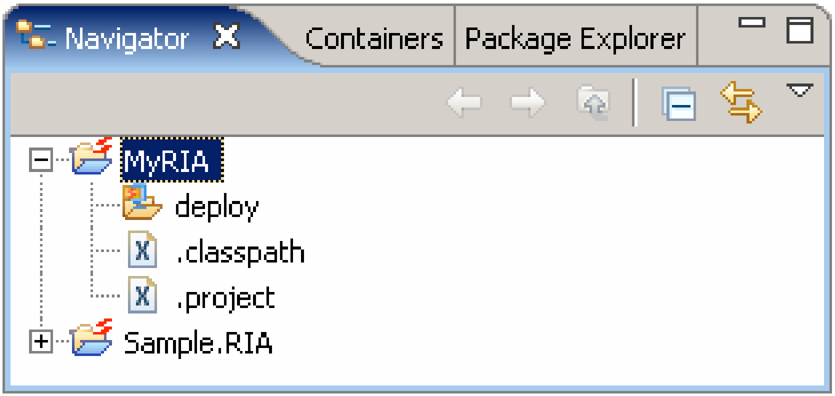
Creating a new project

To create your own Sonic development project for the RIA tutorial:

1. In Sonic Workbench, select **File > New > Sonic Development Project**. The **New Sonic Development Project** wizard opens.
2. Enter *MyRIA* as the name of your project:



3. Accept the default location and click **Finish**. Sonic Workbench creates the new project.
4. Go to the Navigator view to see the new project:



5. The Remote Information Access tutorial requires you to create an ESB process with subprocesses and additional ESB processes. To keep the ESB processes and associated resources separate and organized, it is a good idea to keep them in separate directories. You can create these directories as you go, or you can create them now. To create a new

directory, select **File > New > Folder**. In the **New Folder** dialog box that opens, select a parent directory and enter a folder name. Repeat these steps to create the following folders:

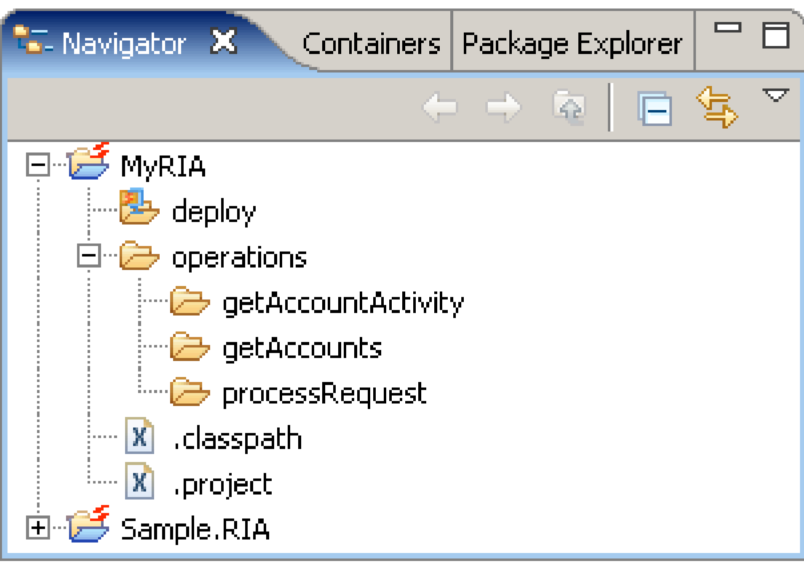
   a. Under the parent folder **MyRIA**, create the folder *operations*.

   b. Under the parent folder **MyRIA/operations**, create the folder *processRequest*. This folder will hold resources associated with the main ESB process, processRequest.

   c. Under the parent folder **MyRIA/operations**, create the folder *getAccounts*. This folder will hold resources associated with the ESB subprocess, getAccounts.

   d. Under the parent folder **MyRIA/operations**, create the new folder *getAccountActivity*. This folder will hold resources associated with the ESB subprocess, getAccountActivity.

6. Go to the Navigator view to see the new folders:



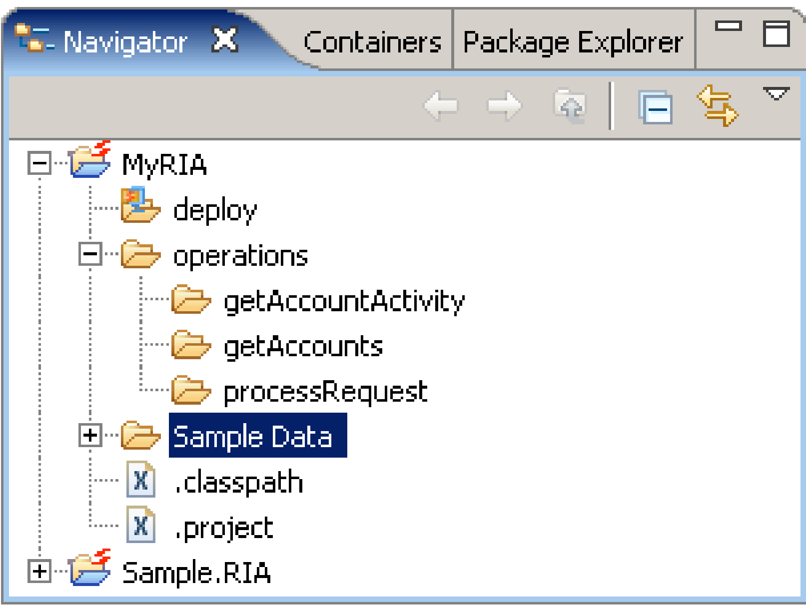
Next, copy the sample data.

RIA tutorial, phase 1

Copying the RIA sample data

Now that you have created the project, you can copy the sample data folder containing the XML documents you will use as example request and response messages for the ESB processes:

1. Select the Sample Data folder under the Sample.RIA folder, right-click, and select **Copy**.
2. Select your **MyRIA** folder, right-click, and select **Paste**. Sonic Workbench adds the Sample Data folder to your project:



Next, create the prototype ESB process, processRequest.

RIA tutorial, phase 1

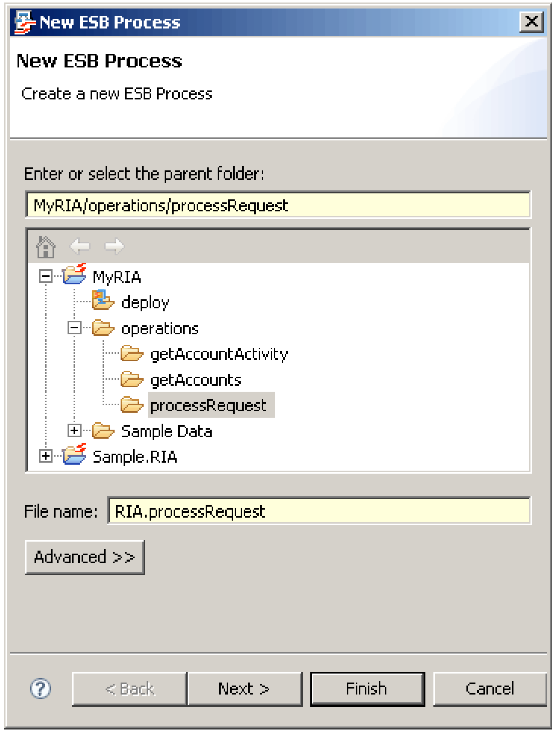
Creating the prototype ESB process

After creating the project and copying the sample data, you are ready to create an ESB process.

Creating this prototype ESB process enables you to establish and test an interface between the customer portal and Sonic ESB.

To create the prototype ESB process, processRequest:

1. Select **File > New > ESB Process**. The **New ESB Process** wizard opens.
2. Select **MyRIA/operations/processRequest** as the parent folder. (In this sample, putting all your processes in the separate directories you created earlier helps organize the files.)
3. Enter *RIA.processRequest* as the name of the ESB process. (You use a different name to avoid over-writing the process in the sample project you imported if they are both uploaded.)



4. Click **Finish**. Sonic Workbench creates the new ESB process.

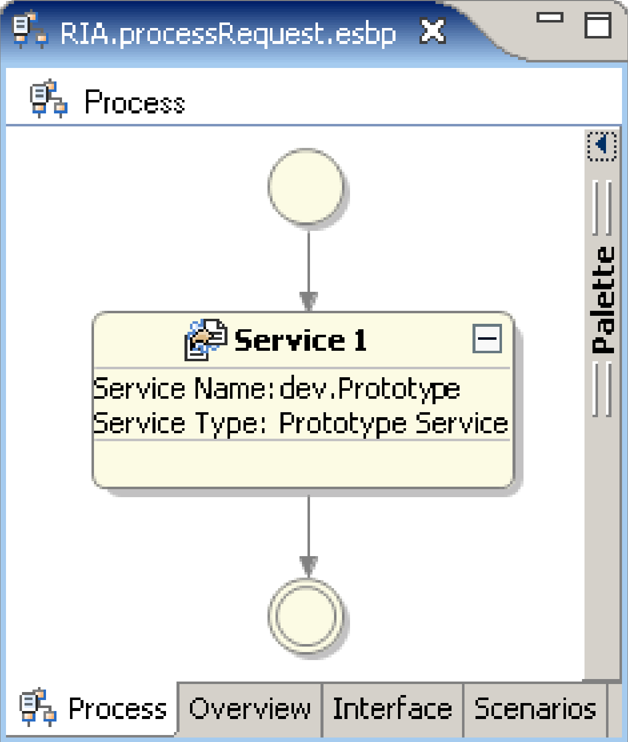
Next, view the prototype ESB process in the ESB Process editor.

[RIA tutorial, phase 1](#)

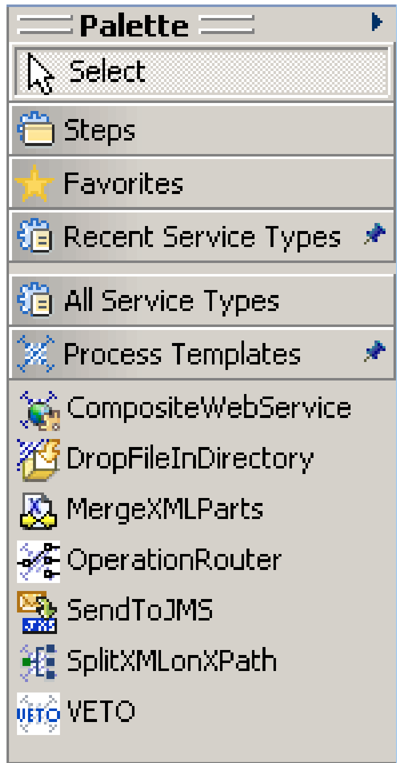
Viewing the prototype ESB process

After creating the prototype ESB process, processRequest, you can view it in the ESB Process editor:

1. The new ESB process opens in the Process page in the ESB Process editor. By default, the ESB process contains the Prototype service, Service 1 (shown expanded):



2. Open the Palette (on the right side of the Process page) and view the sections of the Palette:



Later, you will drag process templates and services from the Palette onto the ESB process as you implement new functionality in the ESB process.

Next, create a scenario to test the interface between processRequest and the ESB.

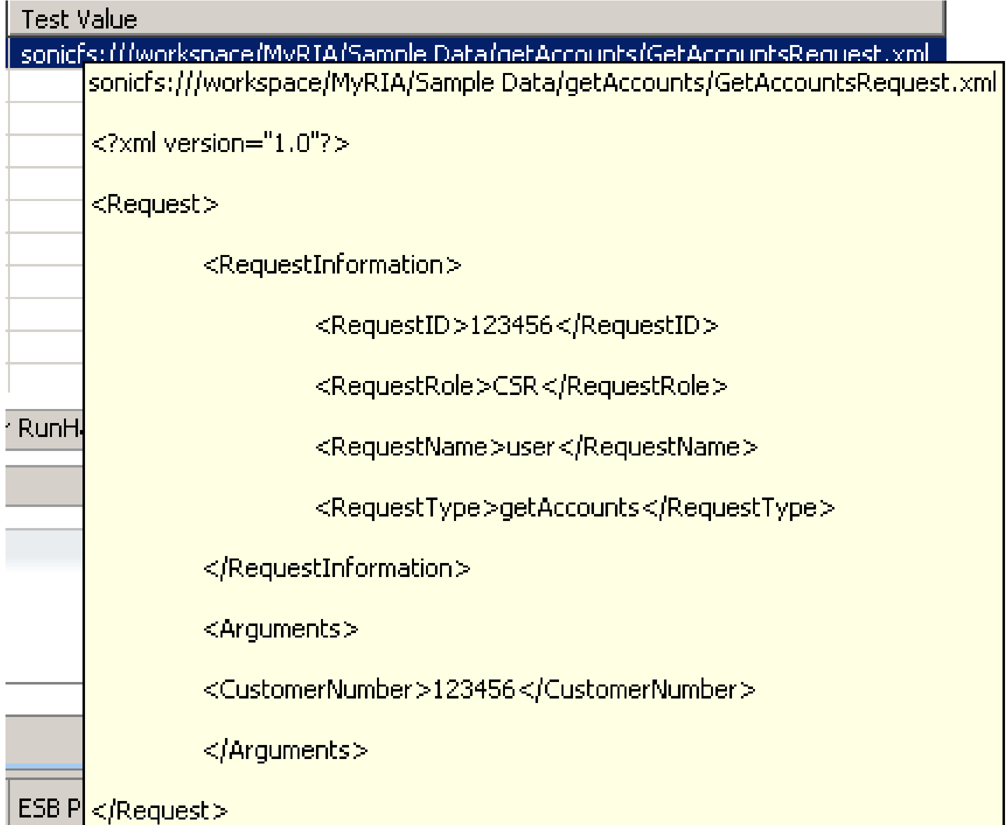
Creating a scenario

You can create scenarios to test ESB processes and other artifacts in Sonic Workbench. Now that you have created the prototype ESB process, processRequest, you can create a scenario to test the interface between the ESB process and Sonic ESB.

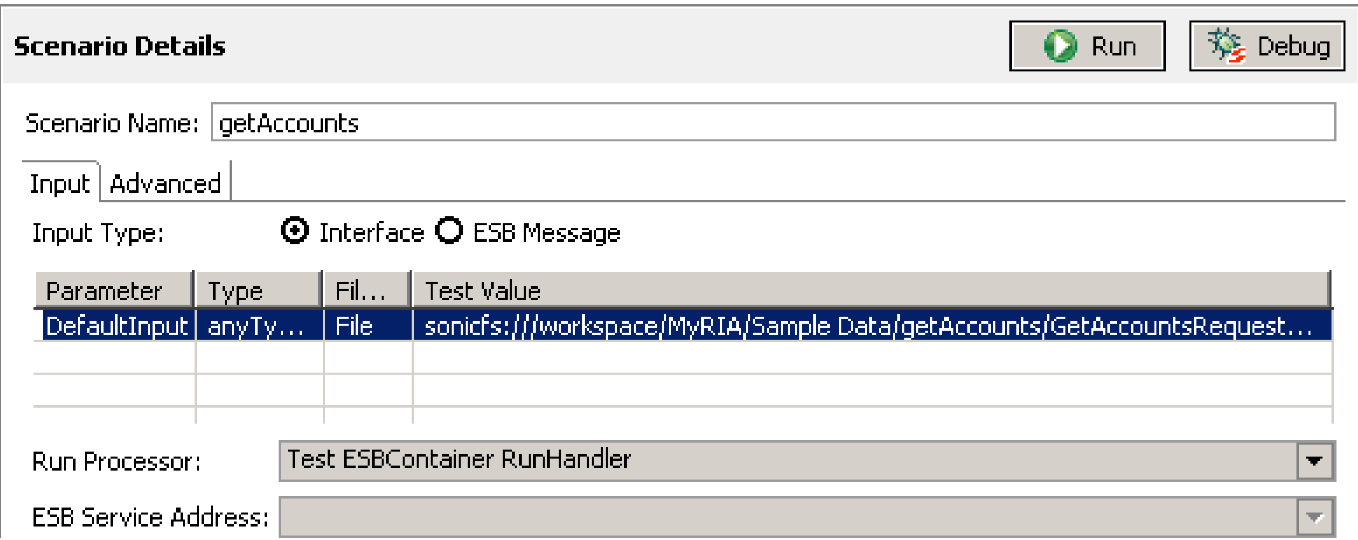
This initial test of the interface simply sends a request to `RIA.processRequest.esbp` and returns data passed through in the request. Create this scenario using the sample XML file, `GetAccountsRequest.xml`, which provides the prototype service with a request for account information. Because you have not yet configured the ESB process to do anything with the request, when run, the scenario will simply return the contents of the request.

To create the scenario:

1. With `RIA.processRequest.esbp` open in the ESB Process editor, click the **Scenarios** tab to open the Scenarios page.
2. In the **Scenarios** section, click **Add Scenario** to create a new scenario. By default, the new scenario is named `RIA.processRequest_default`.
3. In the **Scenario Details** section, enter or select the following:
   - **Scenario Name**: *getAccounts*.
   - In the **Input** section, select the **Input Type** *Interface*.
     This selection specifies that the input will be supplied in interface parameters, rather than in an ESB message.
   - If the **File/Literal** selection in the **Input** table is not already set to **File**, click the entry in the field and select **File** from the pull-down list.
   - To enter a scenario **Test Value**, drag the sample XML file, `GetAccountsRequest.xml`, from the folder MyRIA/Sample Data/getAccounts in the Navigator view.
     Notice that you can position your cursor over the scenario test value URL to view the contents of the file:



The **Scenario Details** section now looks like this:

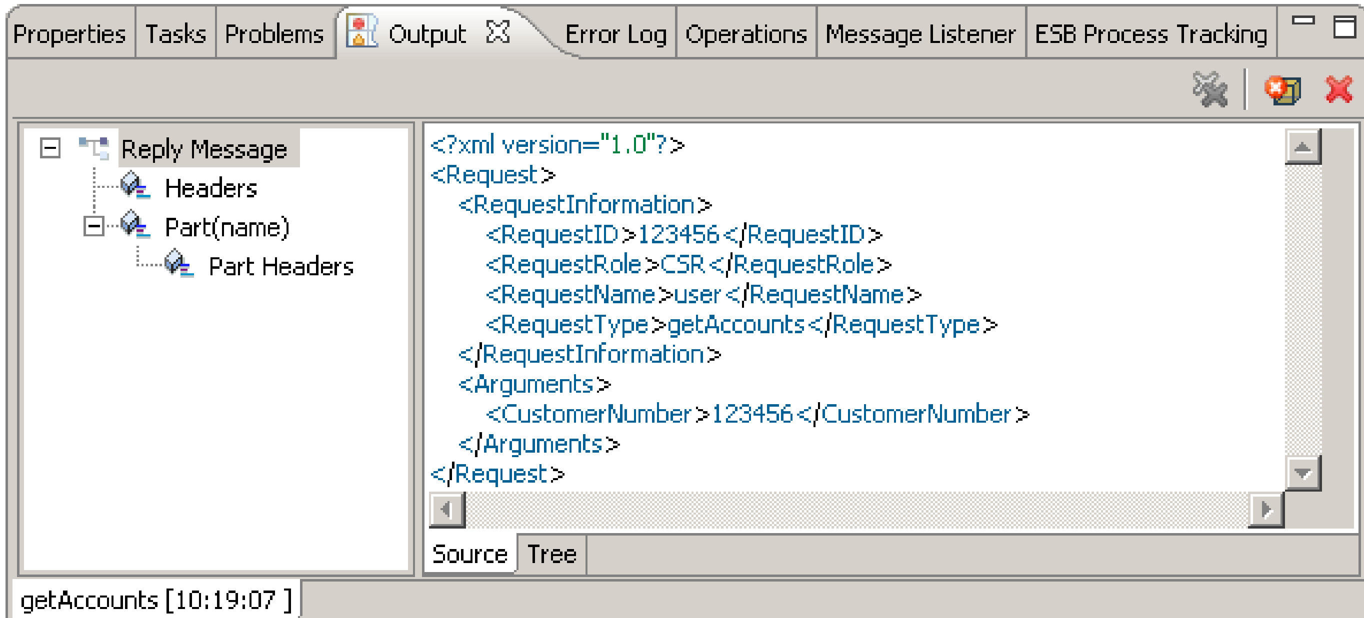


Next, run processRequest using this scenario.

Running and testing the prototype ESB process

Now that you have created the getAccounts scenario, you can run the scenario to test the interface between the prototype ESB process, processRequest, and Sonic ESB. Because you have not yet configured the ESB process to do anything with the request, when run, the scenario will simply return the contents of the request:

1. With `RIA.processRequest.esbp` open in the Scenarios page of the ESB Process editor, select the **getAccounts** scenario.
2. Click **Run** to run the process using this scenario.
3. View the **Reply Message** in the Output view. At this initial stage of development, the ESB process simply returns the request sent by the scenario:



4. Observe that the content of the reply message is the same as the content of the XML file GetAccountsRequest.xml, sent in the scenario.

This tutorial uses iterative development techniques to build on each phase of implementation. The next step, therefore, is to modify the prototype process to implement more functionality.

Next, instead of simply passing through the request, modify the ESB process to return a response based on an incoming request.

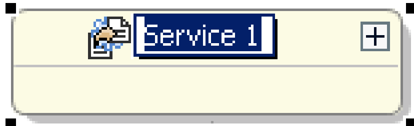
Modifying processRequest to return a response

After creating and testing the prototype ESB process, the next step is to modify processRequest to return a more meaningful response, based on an incoming request. You do this by dragging a sample response XML file from the Navigator view onto the service step. When the incoming request is received, the service will return the information in this response file.

In the following procedure you also rename the initial step, Service 1, to a more meaningful name. Renaming steps to something meaningful helps you keep track of the different parts of an ESB process.

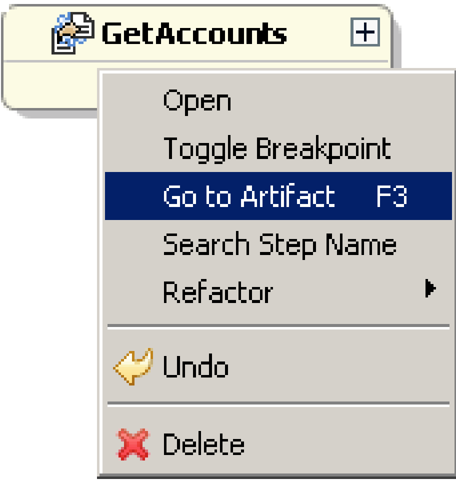
Later, you will implement a subprocess to return information from different accounts, but for now you are just establishing that the prototype service will return a response based on an incoming request:

1. Return to the Process page. Select the prototype service, **Service 1**, that was created automatically when you created processRequest. Click the name on the step, **Service 1**, so you can rename it:

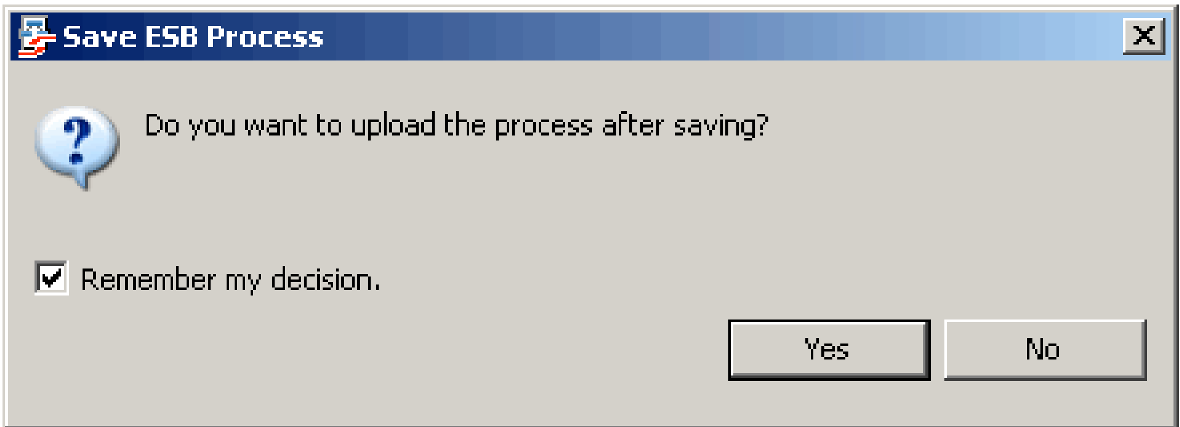
   

2. Change the name to *GetAccounts*.
3. Drag the sample XML response file, `GetAccountsDefaultResponse.xml`, from the folder MyRIA/Sample Data/getAccounts in the Navigator view, onto the GetAccounts service step in the Process page of the ESB editor. This XML file supplies a response containing account information. When you run the ESB process using the getAccounts scenario, the GetAccounts service will now return the data contained in `GetAccountsDefaultResponse.xml`.

   Note: You can view the contents of this XML file by right-clicking **GetAccounts** and selecting **Go to Artifact**:

4. Save the modified ESB process.
5. The **Save ESB Process** dialog box prompts you to upload the process after saving (modified ESB processes must be uploaded before you can run and test them). Check the checkbox next to **Remember my decision** to automatically upload when saving any modified ESB processes or resources:

   Click **Yes** to continue.

Next, run the scenario to test the modified ESB process.

Testing the modified ESB process

Now that you have modified processRequest to return a response based on an incoming request, you can run the ESB process using the getAccounts scenario to confirm that the process now returns a response based on the incoming request:

1. Select the **Scenarios** tab to open the Scenarios page, then select the **getAccounts** scenario and click **Run**. The GetAccounts step in `RIA.processRequest.esbp` has now been implemented, and returns the accounts data based on the incoming request.
2. View the **Reply Message** in the Output view. The response includes the request information, which specifies the customer for whom accounts are returned, along with an entry for each of the customer´s accounts:

```
<?xml version="1.0"?>
<Response>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccounts</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
  </Arguments>
  <Data>
    <Accounts>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>TVAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>InternetAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>WirelessCellAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>PhoneAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
    </Accounts>
  </Data>
</Response>
```

3. Compare this response with the initial response you got before modifying the processRequest. Instead of simply returning the request message, processRequest now returns account information based on the customer specified in the incoming request. As you can see in the request information included in the response, the data in this response is for customer number 123456.

For now, processRequest simulates the account information returned, but later you will implement the GetAccounts step as a subprocess to return actual account data.

You have now successfully created and tested the prototype ESB process, processRequest, and you are ready to continue on with Phase 2. The use cases in this tutorial require the ESB process to handle two different types of requests. In Phase 2 you create a content-based router to route requests to different branches based on the type of incoming request.

Note: If you do not want to develop and implement all the phases of processRequest yourself, you can stop here, and run and test the sample files included in Sample.RIA. The sample ESB processes are similar to the processes created and implemented in this tutorial, and include scenarios to run the processes. Simply open the sample ESB process you want to test and proceed directly to the instructions to run and test the process:

- `Sample.RIA.GetAccounts.esbp` — Follow the instructions in Running and testing the getAccounts subprocess
- `Sample.RIA.getAccountActivity.esbp` — Follow the instructions in Running and testing the getAccountActivity subprocess
- `Sample.RIA.processRequest.esbp` — Follow the instructions in Testing the fully implemented ESB process, processRequest

Phase 2: Implementing multiple operations using a content-based router

The processRequest process includes two use cases, GetAccounts and Get AccountActivity. These use cases require the process to handle requests for both a list of a customer´s accounts (GetAccounts) and account activity on a specified account (GetAccountActivity). In this phase, you create a content-based router with two branches (one for GetAccounts and one for GetAccountActivity) and configure XPath routing rules to route the different request types to different branches. In later phases you will implement each branch of processRequest to handle the request type routed to it. For now, you create a prototype operation router and establish the two branches:

1. Create a content-based router — Create an operation router with two branches.
2. Create branch 1 — Create a branch for the GetAccounts use case that will compile an account list for a given customer. Later, you will implement a subprocess in this branch.
3. Create branch 2 — Create a branch for the GetAccountActivity use case that will retrieve data from different sources. For now you create a prototype step on this branch; later you will implement a subprocess in this branch.
4. Modify the routing rules — Create XPath routing rules that will route messages based on the request type in the message content. Configure two rules:
    - Rule 1 — Send requests for a list of a customer´s accounts to branch 1.
    - Rule 2 — Send requests for all activity on a specified account to branch 2.
5. Create scenarios to test the content-based router — Create two scenarios for the processRequest process. The scenarios test the routing to each branch of the content-based router.
6. Run and test the content-based router — Run the scenarios to confirm that messages are routed correctly based on the request type in the message content.

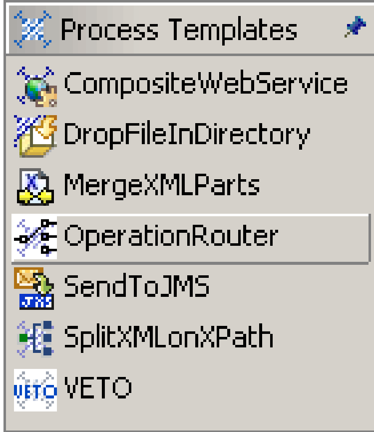
Start by creating a content-based router.

Creating an operation router

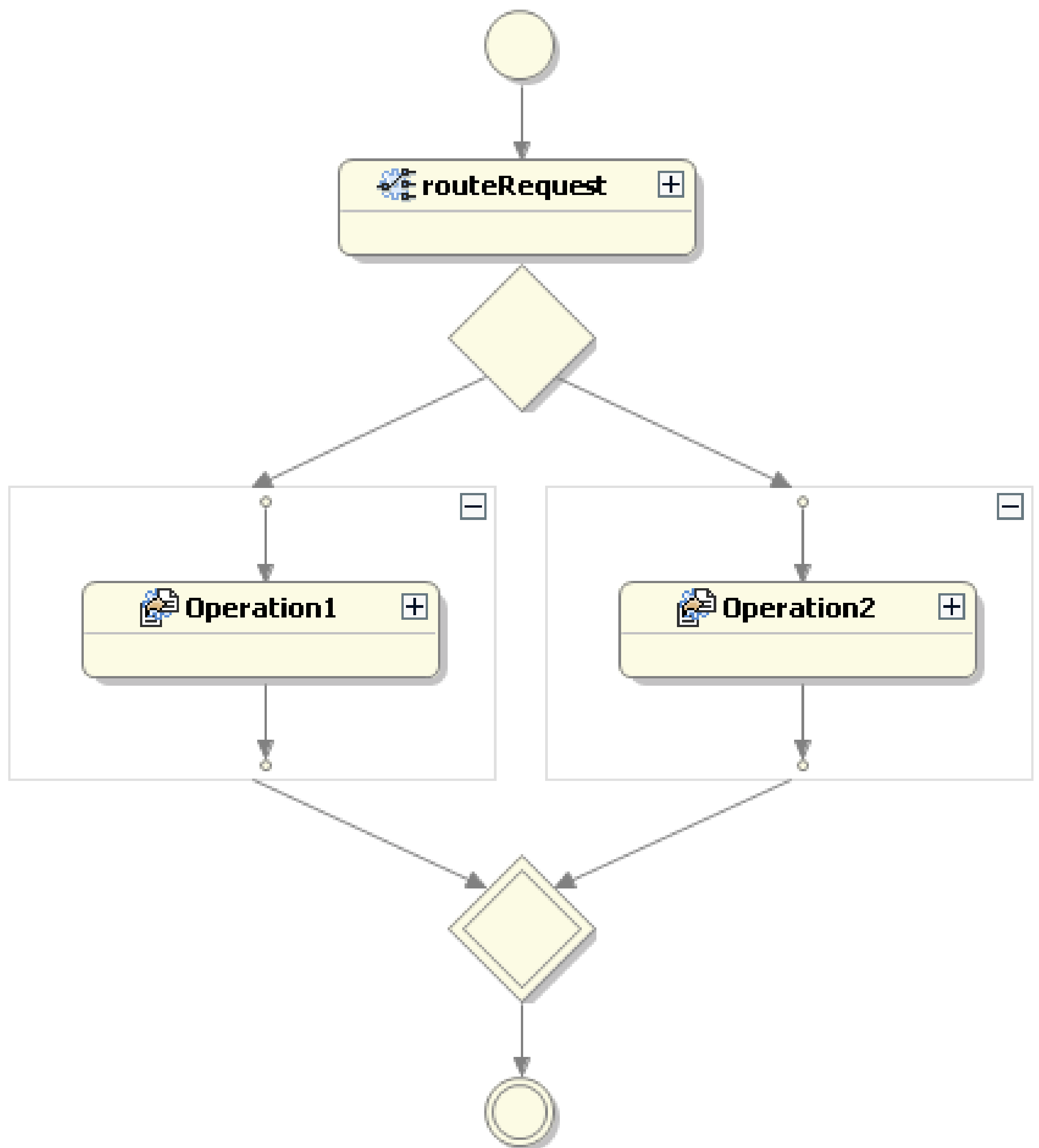
In this tutorial, requests sent to processRequest specify the request type, either getAccounts or getAccountActivity. To handle these requests, you create a content-based router to send requests for accounts and account activity to different branches of the process. One branch of the router handles requests for account information (using the GetAccounts service you previously implemented), and the other branch handles requests for account activity.

The following procedure makes use of the operation router template available on the ESB Process editor Palette. This process template contains a prototype operation router to help get you started:

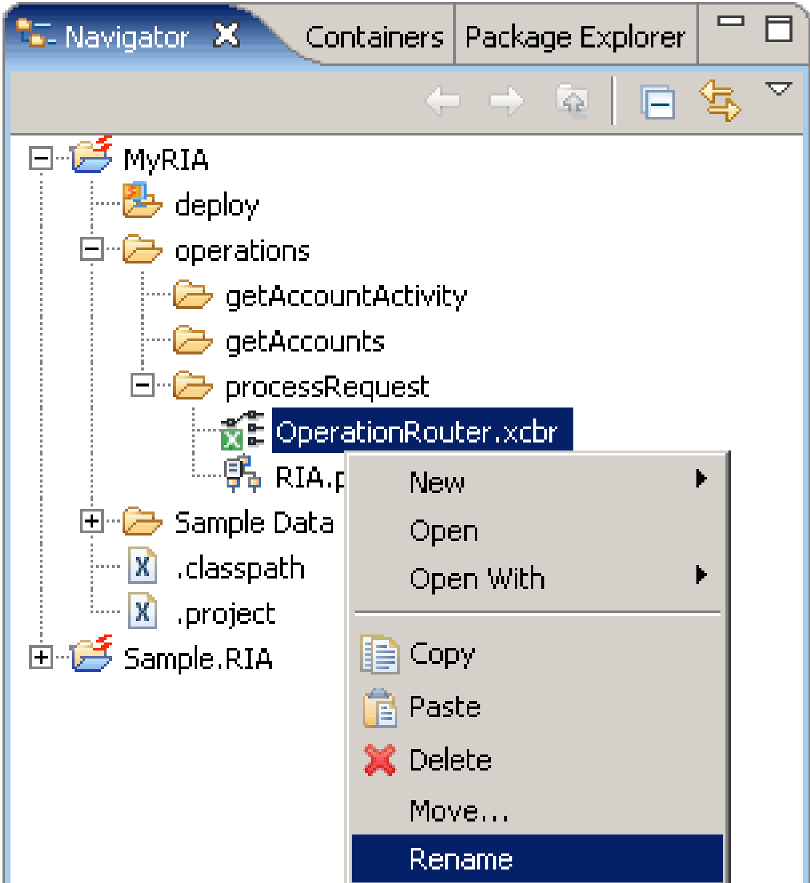
1. Go to the Process page. Select **OperationRouter** from the **Process Templates** section of the Palette:



2. Drag the operation router template onto the process, dropping it above or below the existing **GetAccounts** step. The prototype operation router includes a routing step and two branches by default.
3. Delete the **GetAccounts** step, since you are replacing this step with the operation router.
4. Select the new **Operation Router** step and click the step name so you can rename it. Change the step name to *routeRequest*. The ESB process now has an operation router and two branches:



5. When you create the operation router, an XPath routing rules file is created with the default name **OperationRouter.xcbr** and saved in the same location as the ESB process in which it is created. It is a good idea to rename this file to help you keep track of it. Select the file in the Navigator view, right-click, and select **Rename**:



6. Change the name to *routeRequest.xcbr*, and confirm the name refactoring when prompted. Click **Yes** in the **Save All Modified Resources** dialog box, then click **OK** in the **Sonic Rename Processor** dialog box. Finally, select **MyRIA** in the Navigator view and choose **Project > Upload All** from the menu bar to upload the renaming changes.

   The name of the XPath routing rules file is now descriptive of its functionality, which is helpful in managing your resources when you have multiple ESB process and routing rules files.

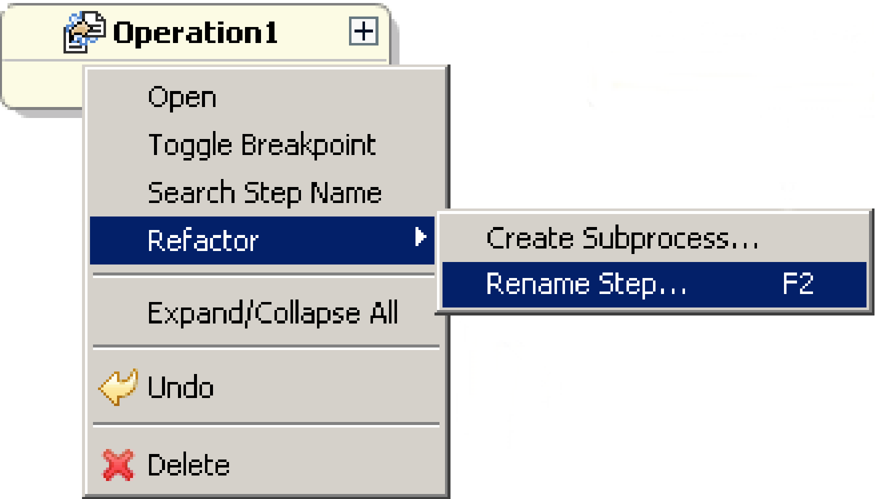
Next, implement one branch of the operation router to compile a list of accounts for a given customer.

[RIA tutorial, phase 2]

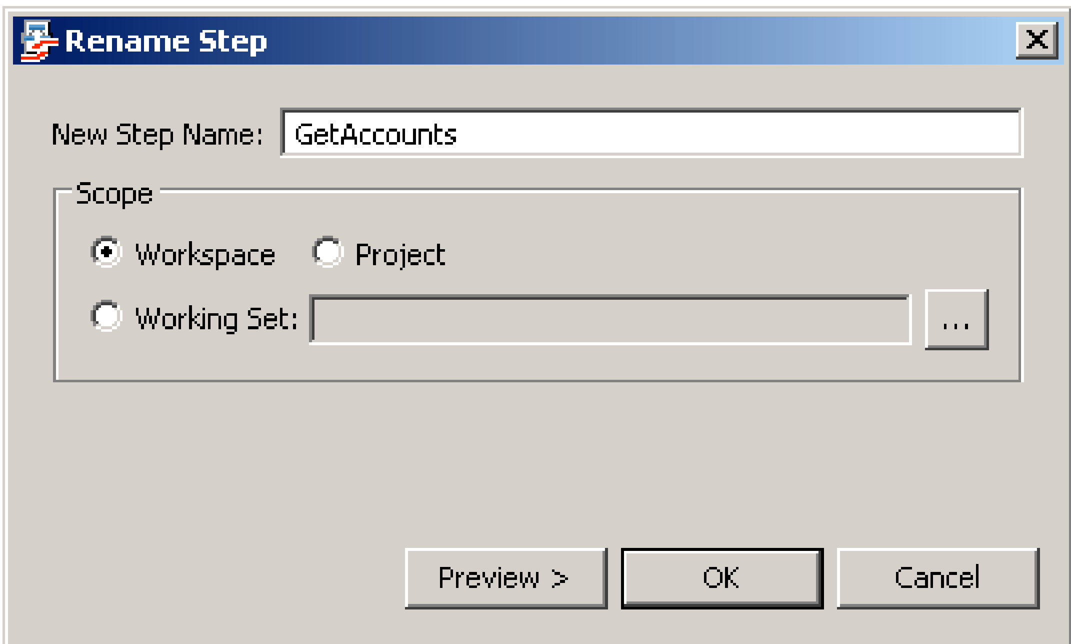
Branch 1: Compiling an account list for a customer

Now that you have [created the prototype operation router], you can implement one branch of the operation router to handle requests for accounts for a given customer. Requests directed to this branch of processRequest will contain the request type **getAccounts**, and will specify a particular customer. In a later phase, you will implement a subprocesss on this branch to aggregate all the accounts held by the customer and return a response containing a list of the customer´s accounts. For now, the prototype GetAccounts step will return a simulated response:

1. Right-click **Operation 1** and select **Refactor > Rename Step**:



2. In the **Rename Step** dialog box that opens, enter the step name **GetAccounts**:

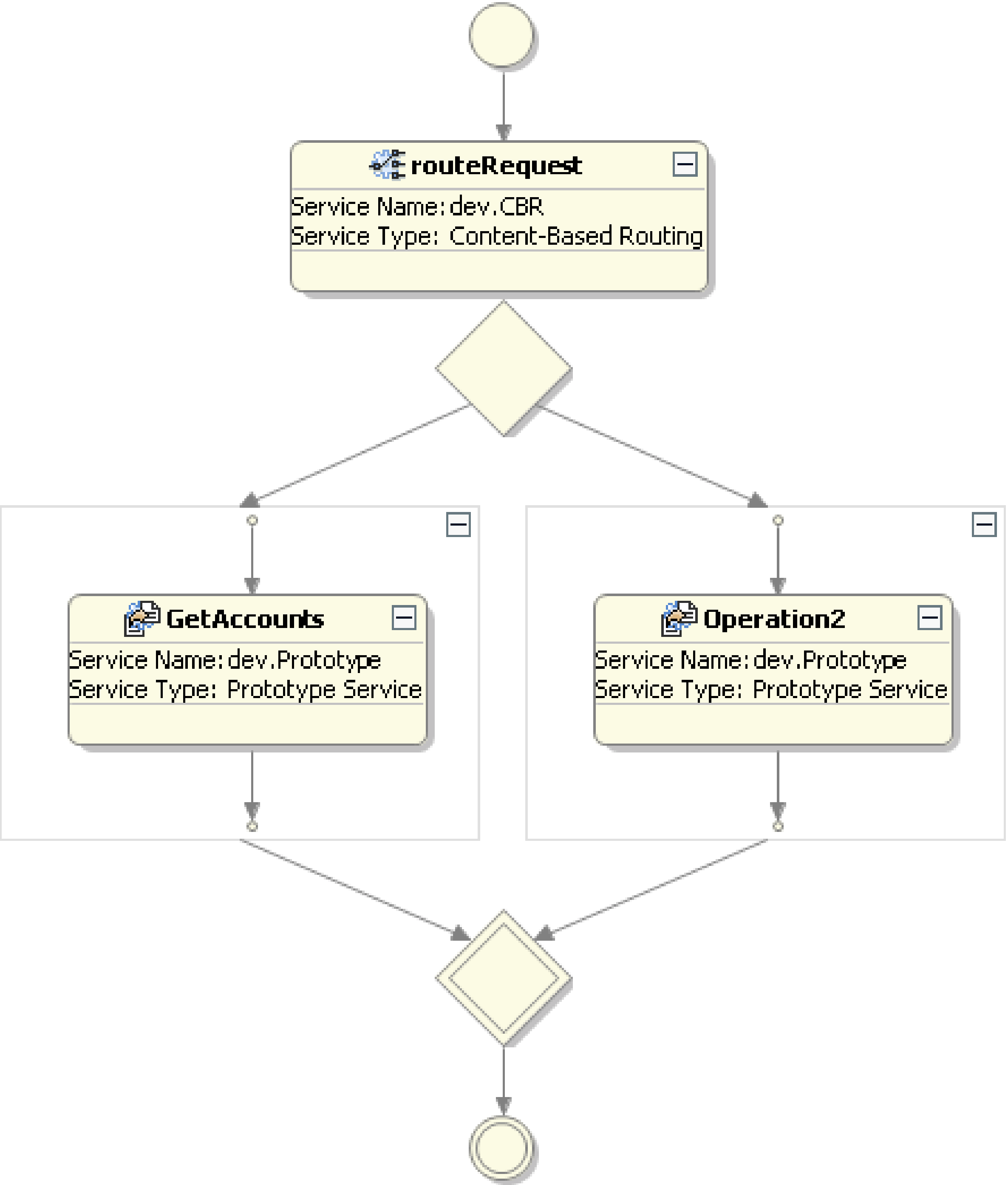


Accept the default scope, Workspace, then click **OK**.

3. Drag the sample XML response file, `GetAccountsDefaultResponse.xml`, from the

folder MyRIA/Sample Data/getAccounts in the Navigator view, onto the **GetAccounts** service step.

4. Observe that the process now has a branch configured with the prototype GetAccounts service (steps shown expanded):



5. Save the modified ESB process.

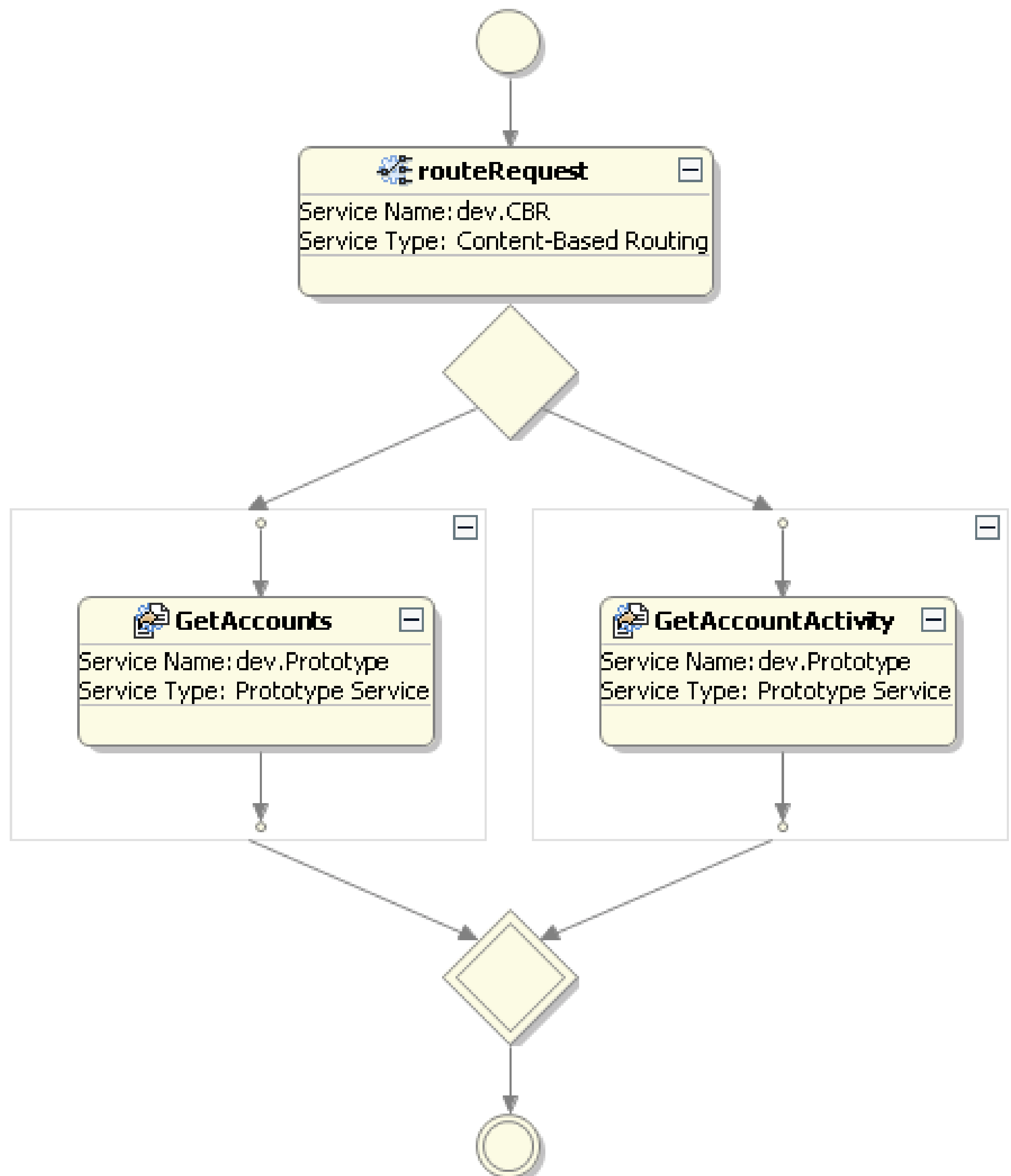
Next, implement the other branch to handle requests for account activity.

Branch 2: Retrieving data from different sources

Now that you have configured the first branch of the operation router with the prototype GetAccounts step, you can configure the other branch to handle requests for account activity. Requests directed to this branch of processRequest will contain the request type **getAccountActivity**, and will specify a particular customer. In this initial phase, this branch will return a simulated response. Later, when fully implemented, this branch will return a response containing all the activity for the customer and account type specified in the request.

For now, create a prototype service for this branch, which will enable you to test the content-based routing based on different incoming request types:

1. Right-click **Operation 2** and select **Refactor > Rename Step**. In the **Rename Step** dialog box that opens, enter the step name **GetAccountActivity**, then click **OK**.
2. Drag the sample XML response file, `GetAccountActivityDefaultResponse.xml`, from the folder MyRIA/Sample Data/getAccountActivity in the Navigator view, onto the **GetAccountActivity** service step.
3. Right-click the **GetAccountActivity** service step and select **Go to Artifact** to view the contents of the default response file `GetAccountActivityDefaultResponse.xml`. This XML file supplies a response containing account activity to simulate the actual data that will be returned later when you fully implement this step.
4. Observe that the process now has two branches containing prototype services:



5. Save the modified ESB process.

Next, modify the routing rules for the content-based router.

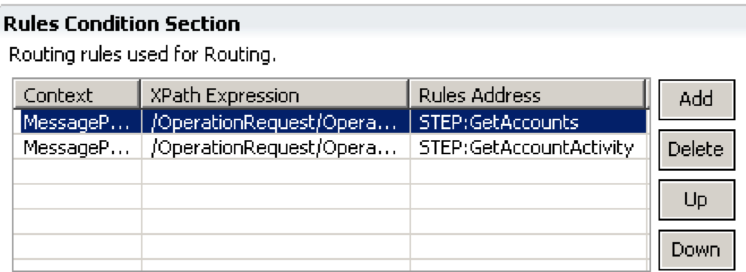
Modifying the routing rules

Having created the GetAccounts and GetAccountActivity branches of the operation router, you are ready to modify the routing rules to route requests based on the request type in the request.

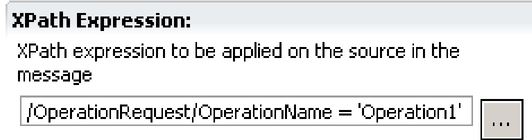
In the following procedure, you configure an XPath expression for each request type. Using the XPath Helper in Sonic Workbench, you select the sample request documents provided for each use case. These documents help you create an XPath expression that checks the request type of the incoming message.

To modify the routing rules:

1. In `RIA.processRequest.esbp`, right-click the **routeRequest** step and select **Go to Artifact** to open `routeRequest.xcbr`. The file opens in the XPath Routing Rules editor, and has two routing rules, one for each account branch.
2. Modify a rule to route to the GetAccounts branch:
    a. In the **Rules Condition Section** of the **XPath Routing Rules editor**, select the rule for the **GetAccounts** step:

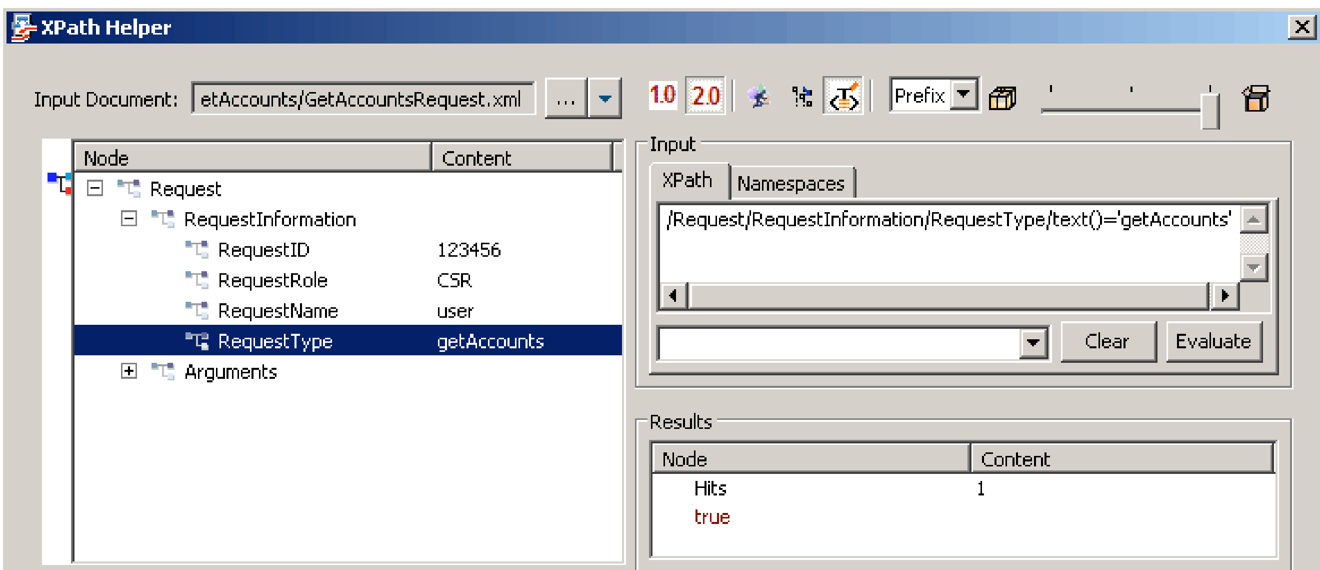
    b. In the **XPath Expression** section, click **...** next to the default XPath expression:

    Browse to the sample input document sonicfs:///workspace/MyRia/Sample Data/getAccounts/GetAccountsRequest.xml.

    c. In the **XPath Helper** that opens, double-click the node **Request/RequestInformation/RequestType**. Notice that the **XPath** field in the **Input** section now contains the expression **/Request/RequestInformation/RequestType/text()**.

    d. In the XPath field, next to the expression you just added, enter: **=´getAccounts´**. Click **Evaluate** to confirm that this XPath expression evaluates to **true**, as shown:

    e. Click **OK** to close the XPath Helper.

    f. In the **Rules Address** section of the XPath Routing Rules editor, confirm that the destination is **STEP:GetAccounts**:

The first XPath routing rule will now route requests having request type **getAccounts** through the GetAccounts branch of the process.

3. To modify a rule to route requests having request type **getAccountActivity** through the GetAccountActivity branch, you can copy and modify the XPath expression from the getAccounts rule:
   a. In the **Rules Condition Section**, select the rule for the **GetAccounts** step and copy the **XPath Expression**.
   b. Select the rule for the **GetAccountActivity** step and paste the copied expression into the **XPath Expression** section.
   c. Modify the XPath expression as follows: */Request/RequestInformation/RequestType/text()**=´getAccountActivity´**.*
   d. In the **Rules Address** section of the XPath Routing Rules editor, confirm that the destination is **STEP:GetAccountActivity**.
4. Save the modified routing rules file.

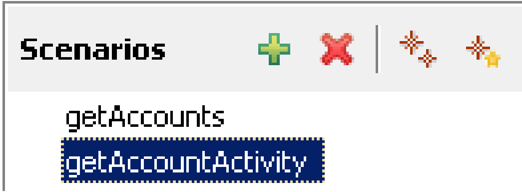
Next, create scenarios to test the content-based router.

RIA tutorial, phase 2

Creating scenarios to test the content-based router

Now that you have created the content-based router and routing rules, you can use scenarios to test how processRequest handles different requests. You can use the scenario you created previously to test the routing through the GetAccounts branch. In the following procedure, you create an additional scenario to test the GetAccountActivity branch:

1. With `RIA.processRequest.esbp` open in the ESB Process editor, select the **Scenarios** tab to open the Scenarios page.
2. Click **Add Scenario** to add a new scenario. In the **Scenario Details** section of the Scenarios page, enter or select the following:
   - **Scenario name**: *getAccountActivity*.
   - In the **Input** section, select the **Input Type** *Interface*.
   - If the **File/Literal** selection in the **Input** table is not already set to **File**, click the entry in the field and select **File** from the pull-down list.
   - To enter a scenario **Test Value**, drag the sample XML file, `GetAccountActivityRequest.xml`, from the folder MyRIA/Sample Data/getAccountActivity in the Navigator view.
3. The ESB process, processRequest, now has a scenario to test each branch of the content-based router:



Next, run the prototype content-based router to test how it handles different requests.

Running and testing the prototype content-based router

Now that you have created routing rules and scenarios for each use case, you can run them to test how the content-based router handles different requests. In the following procedure, you test processRequest using the getAccounts scenario to test the GetAccounts branch, and the getAccountActivity scenario to test the GetAccountActivity branch:

1. Select the getAccounts scenario.
2. Click **Run** to run the process using the getAccounts scenario, which supplies a request for accounts.
3. View the **Reply Message** in the Output view:

```
<?xml version="1.0"?>
<Response>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccounts</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
  </Arguments>
  <Data>
    <Accounts>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>TVAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>InternetAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>WirelessCellAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>PhoneAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
    </Accounts>
  </Data>
</Response>
```

Notice that the **RequestType** is getAccounts, and the **Data** element contains data about multiple accounts for the specified customer. This output demonstrates that the request was routed through the GetAccounts branch of the content-based router.

4. Select the getAccountActivity scenario and click **Run**. This scenario supplies a request for account activity.
5. View the **Reply Message** in the Output view:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
      </Account>
   </Arguments>
   <Data>
      <AccountActivity>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
         <OutstandingBalance>44.99</OutstandingBalance>
      </AccountActivity>
   </Data>
</Response>
```

Notice that the result contains activity for a particular account, the TV account. This output demonstrates that the request was correctly routed through the GetAccountActivity branch of the content-based router.

In this prototype phase of the tutorial, the results returned from running these scenarios are based on the default response XML files you configured for each service, GetAccounts and GetAccountActivity. Using iterative development techniques, the remaining phases of the tutorial build on the work you have already done, expanding the functionality of the ESB process and services you have created. In the next phases of this tutorial, you will implement subprocesses on each branch to return the account or account activity data.

You have now successfully created and tested the prototype content-based router for processRequest, and you are ready to continue on with Phase 3. In Phase 3 you implement the GetAccounts branch of processRequest by refactoring GetAccounts as a subprocess that will retrieve and combine data from multiple sources into a single response.

Note: If you do not want to develop and implement all the phases of processRequest yourself, you can stop here, and run and test the sample files included in Sample.RIA. The sample ESB processes are similar to the processes created and implemented in this tutorial, and include scenarios to run the processes. Simply open the sample ESB process you want to test and proceed directly to the instructions to run and test the process:

- `Sample.RIA.GetAccounts.esbp` — Follow the instructions in Running and testing the getAccounts subprocess
- `Sample.RIA.getAccountActivity.esbp` — Follow the instructions in Running and testing the getAccountActivity subprocess
- `Sample.RIA.processRequest.esbp` — Follow the instructions in Testing the fully implemented ESB process, processRequest

Phase 3: Implementing getAccounts using a Split and Join Parallel service

In Phase 2, you implemented and tested a prototype content-based router with two branches in processRequest. Now you are ready to refactor the GetAccounts step in processRequest as a Split and Join Parallel service, enabling your process to combine data from multiple sources into a single response:

1. Refactor GetAccounts — Refactor the GetAccounts step in processRequest as a subprocess containing a Split and Join Parallel service, CombineAllAccounts. This service simultaneously calls out to different data sources and combines the data returned from the data sources.
2. Create ESB processes for each account type — Create an ESB process to return data for each account type (Phone, TV, and Wireless Cell). In a later phase, you can implement each of these ESB processes to connect to the data source for a particular account and return the actual account data. For now, you configure each ESB process with a simulated response.
3. Configure a list of called addresses — Configure a called address for each of the three ESB processes you just created. These ESB processes will return simulated data for each account type (Phone, TV, and Wireless Cell)
4. Configure the service runtime parameters — Configure service parameters for the Split and Join Parallel service.
5. Run and test the Split and Join Parallel service — Create and run a scenario to retrieve account information for a specified customer, and verify the response.

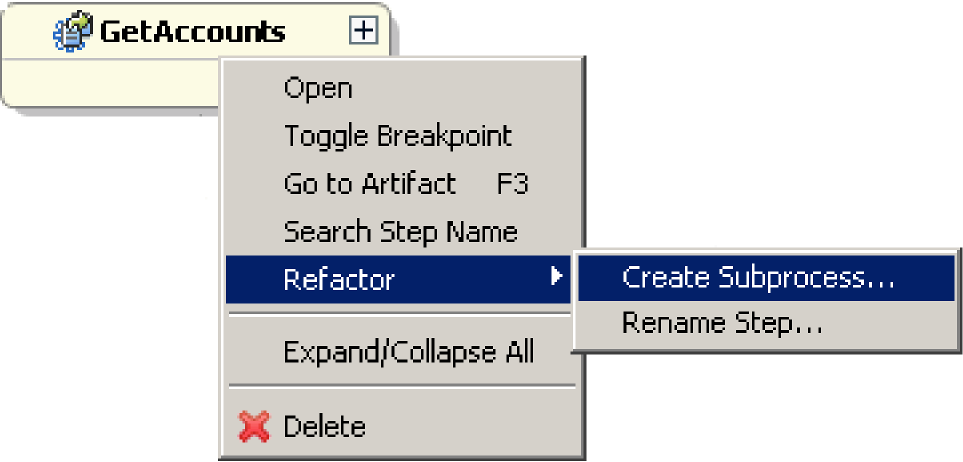
Start by refactoring GetAccounts as a subprocess.

Refactoring GetAccounts as a subprocess

In Phase 2, you created a content-based router having two branches. Now you can refactor the service on one of those branches, GetAccounts, as a subprocess to aggregate data for separate accounts. In the subprocess, you create a Split and Join Parallel service, which you later configure to combine data from the three account types used in this tutorial.

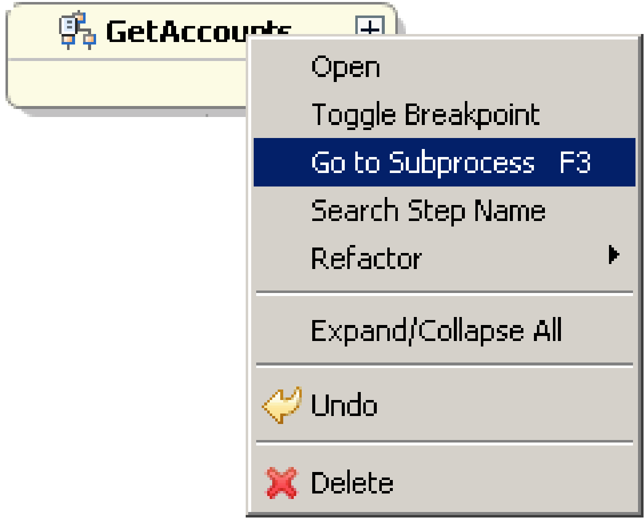
To refactor GetAccounts as a subprocess:

1. Go to the Process page. Right-click the **GetAccounts** step, and select **Refactor > Create Subprocess**:

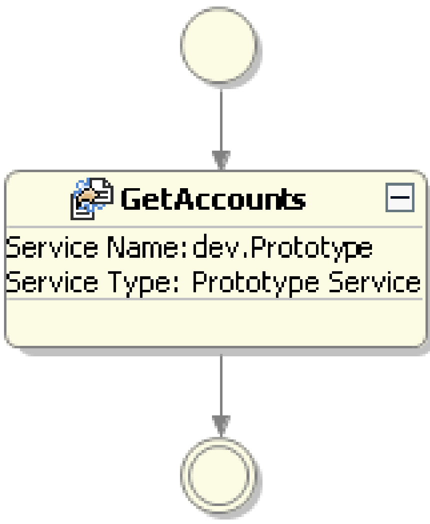


2. In the **New ESB Process** dialog box, select the parent folder *MyRIA/operations/getAccounts* and enter the file name *RIA.getAccounts*. Click **Finish**, then observe that the icon and information on the GetAccounts step has changed, indicating that the service has been refactored as the subprocess RIA.getAccounts:

3. To open the subprocess, right-click the **GetAccounts** step and select **Go to Subprocess**:



The subprocess opens in the ESB Process editor (the step is shown expanded):

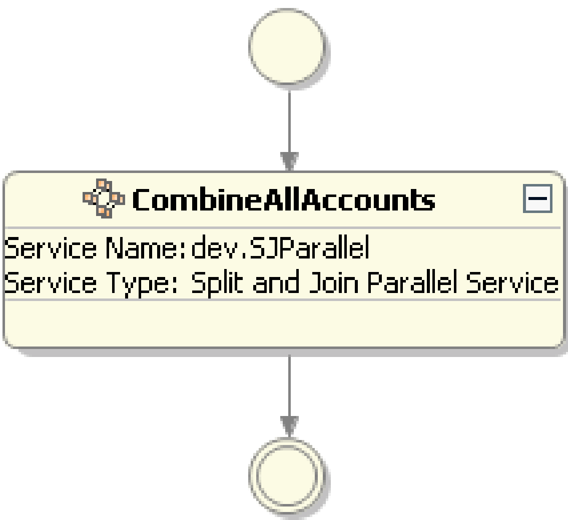


4. Drag the **Split and Join Parallel Service** from the **All Service Types** section of the Palette onto the GetAccounts step. The icon and information on the GetAccounts step changes, indicating the step is now a Split and Join Parallel service:

5. Select the step and click the step name, **GetAccounts** so you can rename it. Change the step name to *CombineAllAccounts*.

   Note: If you have not yet saved the subprocess, the **Save All Modified Resources** dialog box opens, prompting you to save the resources before continuing with renaming the step. Select **OK**, then select **Yes** to upload the process, if prompted.

   The refactored subprocess now looks like this:



Next, create an ESB process for each account type (Phone, TV, and Wireless Cell).

Creating ESB processes for each account type

Now that you have refactored GetAccounts as a subprocess, you can create ESB processes to use as called addresses for the Split and Join Parallel service, CombineAllAcounts. Requests for customer accounts are routed to the GetAccounts branch of processRequest by the content-based router you created in Phase 2. The CombineAllAccounts service will call out to each address simultaneously to retrieve account data, and will add the data from each account to the request message.

Create an ESB process for each account type (Phone, TV, and Wireless Cell), and configure each ESB process with account information by dragging a sample XML file containing this information onto each service. This information simulates information returned from data sources for each account:

1. Select **File > New ESB Process**.
   In the **New ESB Process** dialog box that opens, enter or select the following:
   - **Parent folder**: *MyRIA/operations/getAccounts*
   - **File name**: Enter one of the following:

     RIA.GetPhoneAccount
     RIA.GetTVAccount
     RIA.GetWirelessCellAccount

   Click **Finish**to create each process. (Repeat this step to create all 3 processes.)
   Each process opens in the ESB Process editor.
2. Repeat the following steps for each new ESB Process:
   a. Select and rename each service step in each ESB process:

      GetPhoneInfo (in `RIA.GetPhoneAccount.esbp`)
      GetTVInfo (in `RIA.GetTVAccount.esbp`)
      GetWirelessCellInfo (in `RIA.GetWirelessCellAccount.esbp`)

   b. Drag the corresponding information XML file from the folder, MyRIA/Sample Data/ getAccounts, in the Navigator view onto each ESB process:

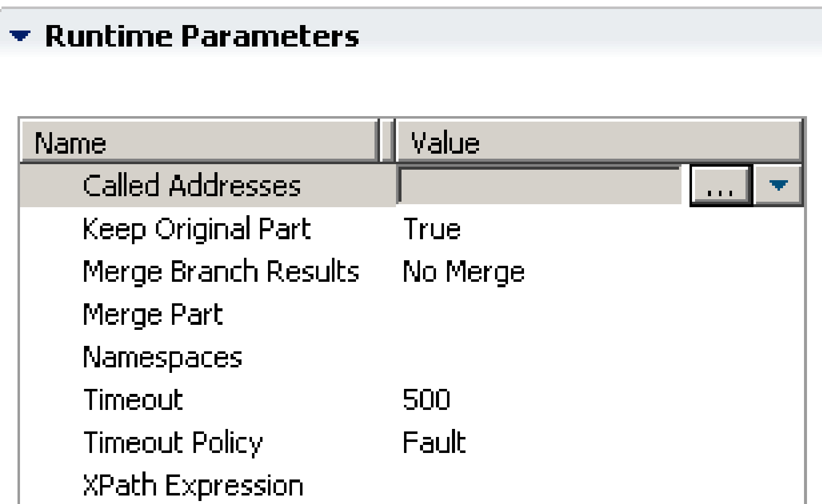
      `PhoneAccountInfo.xml` (onto `RIA.GetPhoneAccount.esbp`)
      `TVAccountInfo.xml` (onto `RIA.GetTVAccount.esbp`)
      `WirelessCellAccountInfo.xml` (onto `RIA.GetWirelessCellAccount.esbp`)
3. Save `RIA.GetPhoneAccount.esbp`, `RIA.GetTVAccount.esbp`, and `RIA.GetWirelessCellAccount.esbp`.
4. Select your project folder, **MyRIA**, in the Navigator view, then select **Project > Upload All** to upload the new ESB processes.

Next, configure a list of called addresses for the Split and Join Parallel service.

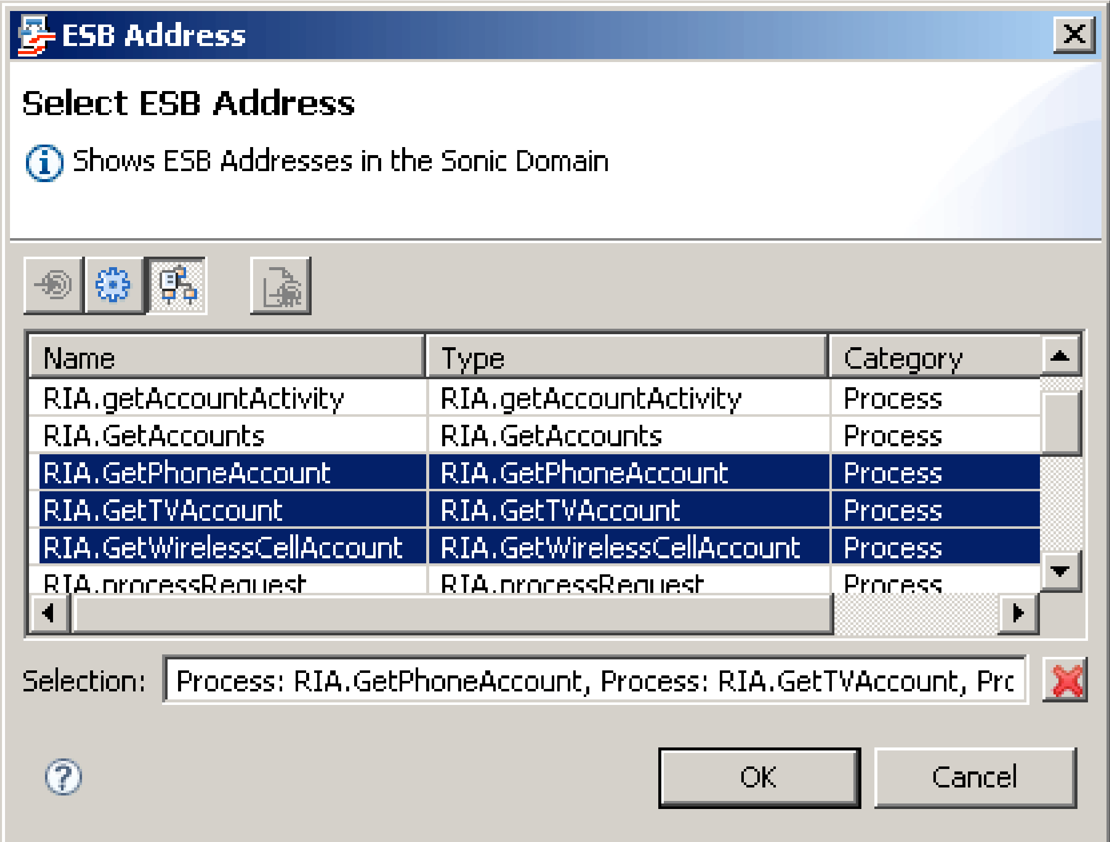
Configuring a list of called addresses

Now that you have created ESB processes to return data for each account type, you can configure a list of called addresses for the Split and Join Parallel service, CombineAllAccounts. This list will contain the addresses of the ESB processes (GetPhoneAccount, GetTVAccount, and GetWirelessCellAccount) you created for each account type:

1. Open `RIA.getAccounts.esbp` and double-click the **CombineAllAccounts** step to open the service.
2. In the **Runtime Parameters** section, click **...** next to **Called addresses**:

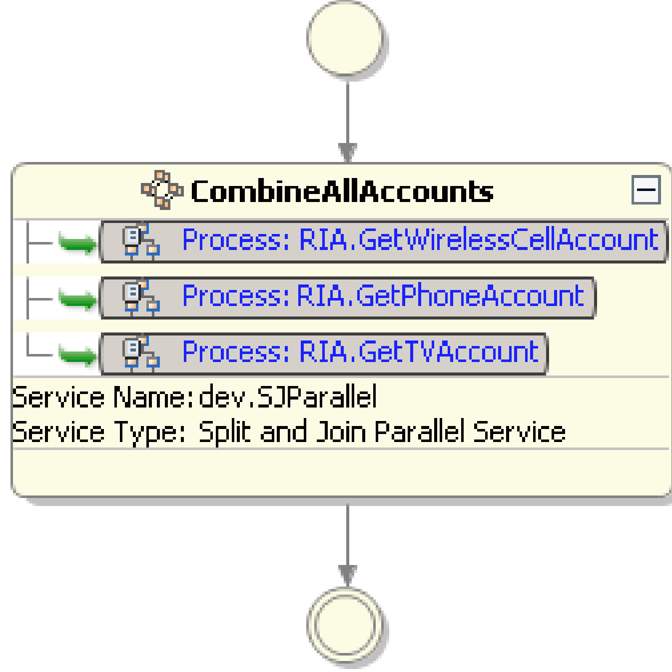


3. In the **Select ESB Address** dialog box that opens, select the three ESB processes you made for the account types: **RIA.GetPhoneAccount**, **RIA.GetTVAccount**, and **RIA. GetWirelessCellAccount**:



Click **OK** to add the ESB processes to the service as called addresses.

4. Save the ESB process.
5. Return to the Process page and expand the CombineAllAccounts step to see that the three ESB processes have been added as called addresses:



Next, configure additional service runtime parameters to specify the timeout behavior and result format.

Configuring the service runtime parameters

Now that you have configured a list of called addresses for the Split and Join Parallel service, CombineAllAccounts, you can configure additional runtime parameters to specify the timeout behavior and how the results format:

1. On the service page for **CombineAllAccounts**, set the following parameters in the **Runtime Parameters** section:

Runtime Parameter	Setting
Keep Original Part	**True** — Include the original message part(s) in the response.
Merge Branch Results	**Append Child Nodes** — The results returned from the three ESB processes are to be merged and appended as child nodes in the response.
Merge Part	**0** — The index of the part to merge the results into.
Timeout	**500** — The global timeout for all branches, in milliseconds.
Timeout Policy	**Continue** — The service continues even if there are no replies from timed out branches.
XPath Expression	**/*[1]** — The response will be appended to the first child node.

2. Save the ESB process.
3. Confirm that your runtime parameters are now configured like this:

Runtime Parameters

Name	Value
Called Addresses	Process: RIA.GetTVAccount, Process: RIA.GetPhoneAccount, Process: RIA.GetWirelessCellAccount
Keep Original Part	True
Merge Branch Results	Append_child_nodes
Merge Part	0
Namespaces	
Timeout	500
Timeout Policy	Continue
XPath Expression	/*[1]

The service has three called addresses, one for each account type subprocess, and is configured to merge the results from the three subprocesses.

Next, run a scenario to test the Split and Join Parallel service.

Running and testing the CombineAllAccounts service

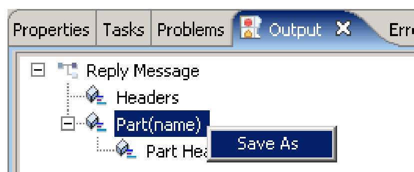
Now that you have finished configuring the Split and Join Parallel service, CombineAllAccounts, you can test the getAccounts subprocess to see the response this service returns. You can create and run a scenario to send a request for accounts for a specified customer to getAccounts. When it receives the request, the service will call in parallel to each of the three ESB processes representing the account types, returning data from all three accounts in a single message. You can view the returned data in the Output view after running the scenario.

In the next phase of the tutorial, you will create a stylesheet to map this output to a different format. You can use the result you obtain in the procedure as input to your stylesheet mapping, so the last step in the following procedure is to save the output for later use:

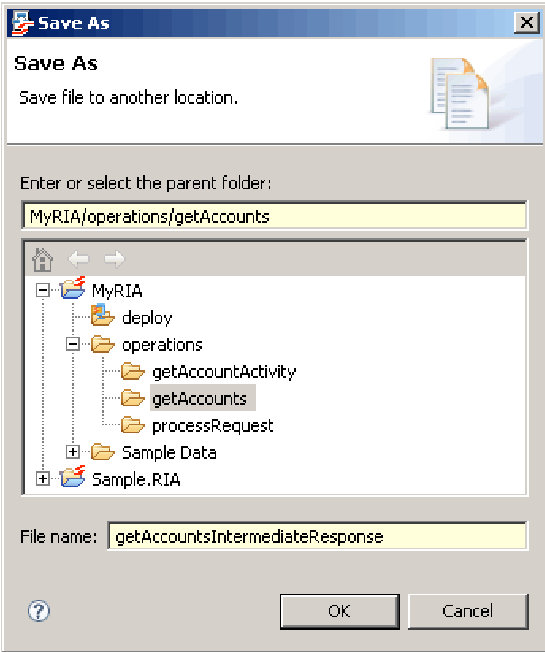
1. Open `RIA.GetAccounts.esbp` and click the **Scenarios** tab to open the Scenarios page.
2. In the **Scenarios** section, click **Add Scenario** ✚, then enter or select the following:
   - **Scenario Name**: *getAccounts*
   - In the **Input** section, select the **Input Type**. *Interface*
   - If the **File/Literal** selection in the **Input** table is not already set to **File**, click the entry in the field and select **File** from the pull-down list.
   - **Test Value**: `GetAccountsRequest.xml` (drag the sample XML file from the folder MyRIA/Sample Data/getAccounts in the Navigator view)
3. Click **Run**. The process runs, calling the three account subprocesses and returning data from each account.
4. Observe the **Reply Message** in the Output view, which contains the original request information and returns data for each account (Phone, TV, and Wireless Cell):

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccounts</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
  </Arguments>
  <Account>
    <AccountNumber>9999999</AccountNumber>
    <AccountType>WirelessCellAccount</AccountType>
    <IsOpen>true</IsOpen>
  </Account>
  <Account>
    <AccountNumber>9999999</AccountNumber>
    <AccountType>TVAccount</AccountType>
    <IsOpen>true</IsOpen>
  </Account>
  <Account>
    <AccountNumber>9999999</AccountNumber>
    <AccountType>PhoneAccount</AccountType>
    <IsOpen>true</IsOpen>
  </Account>
</Request>
```

5. To save this output for use when you create a stylesheet to map the response formats, select the message **Part(name)**, right-click, and select **Save As**:



6. In the **Save As** dialog box that opens, save the file in your folder **MyRIA/operations/getAccounts**. Name the file **getAccountsIntermediateResponse**:



Click **OK** to save the file. The file is saved as XML, and the extension `.xml` is added to the file name.

You have now successfully implemented and tested CombineAllAccounts in the getAccounts subprocess, and you are ready to continue on to Phase 4, where you map from the intermediate response format you just saved into the required output format.

Note: If you do not want to develop and implement all the phases of processRequest yourself, you can stop here, and run and test the sample files included in Sample.RIA. The sample ESB processes are similar to the processes created and implemented in this tutorial, and include scenarios to run the processes. Simply open the sample ESB process you want to test and proceed directly to the instructions to run and test the process:

- `Sample.RIA.GetAccounts.esbp` — Follow the instructions in Running and testing the getAccounts subprocess
- `Sample.RIA.getAccountActivity.esbp` — Follow the instructions in Running and testing the getAccountActivity subprocess
- `Sample.RIA.processRequest.esbp` — Follow the instructions in Testing the fully implemented ESB process, processRequest

Phase 4: Using stylesheets to format responses

In your applications, it is sometimes necessary to transform the output of an ESB process step into a different response format. In the RIA tutorial, the output returned from the Split and Join Parallel service, CombineAllAccounts, is not in the desired format, so you will add a step to the getAccounts subprocess to transform the output. In this phase of the tutorial, you add an XML Transformation service to the ESB subprocess, getAccounts, to transform the output of CombineAllAccounts. You use the output you saved when running CombineAllAccounts as input to the stylesheet, and map these formats to the response format:

1. Add an XML Transformation service to format responses — Add the XML Transformation service Format Response to the getAccounts subprocess.
2. Create a stylesheet — Create a stylesheet for the XML Transformation service you just added to the subprocess.
3. Select interface parameters — Use the intermediate response you saved previously as the default input to the stylesheet, and use a sample XML provided with the sample as the default output for the stylesheet.
4. Map response parameters — Map the output from the CombineAllAccounts step to the response format provided in the sample XML default response file.
5. Test the stylesheet — Create a scenario and test the XML Transformation service.
6. Test the subprocess, GetAccounts — Use a scenario to test the getAccounts subprocess, which now includes the Split and Join service, CombineAllAccounts, and the XMl Transformation service, Format Response.

Start by adding an XML Transformation service.

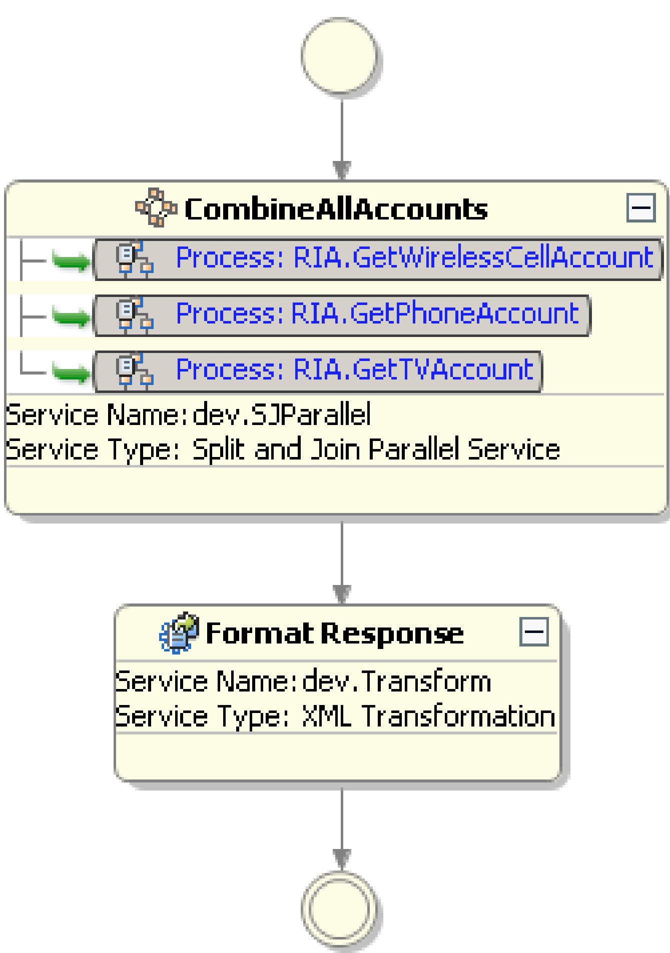
Adding an XML Transformation service to format responses

You can add an XML Transformation service to transform output into a preferred format. In this step, you add an XML Transformation service to the subprocess, getAccounts, to transform the response from the Split and Join Parallel service. Later, you create a stylesheet for the XML Transformation service and map the response formats.

To add an XML Transformation service to the subprocess, getAccounts:

1. Open `RIA.getAccounts.esbp`, which you modified in Phase 3 to contain a Split and Join Parallel service.
2. Under **All Service Types** in the Palette, select the **XML Transformation** service XML Transformation and drag it onto the ESB process, dropping it below the service **CombineAllAccounts**.
3. Selece the XML Transformation service step and click the step name so you can rename it. Change the step name to *Format Response*.
4. Save the ESB process.

The getAccounts subprocess now looks like this (shown with steps expanded):

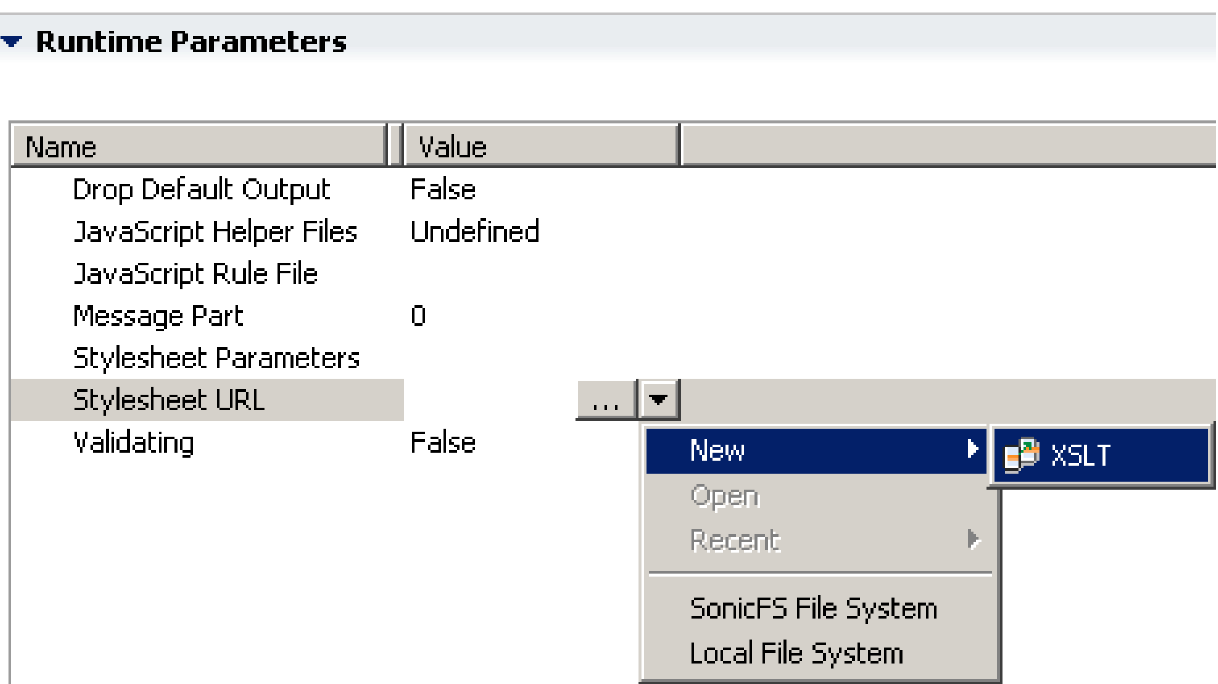


Next, create a stylesheet to map the response formats.

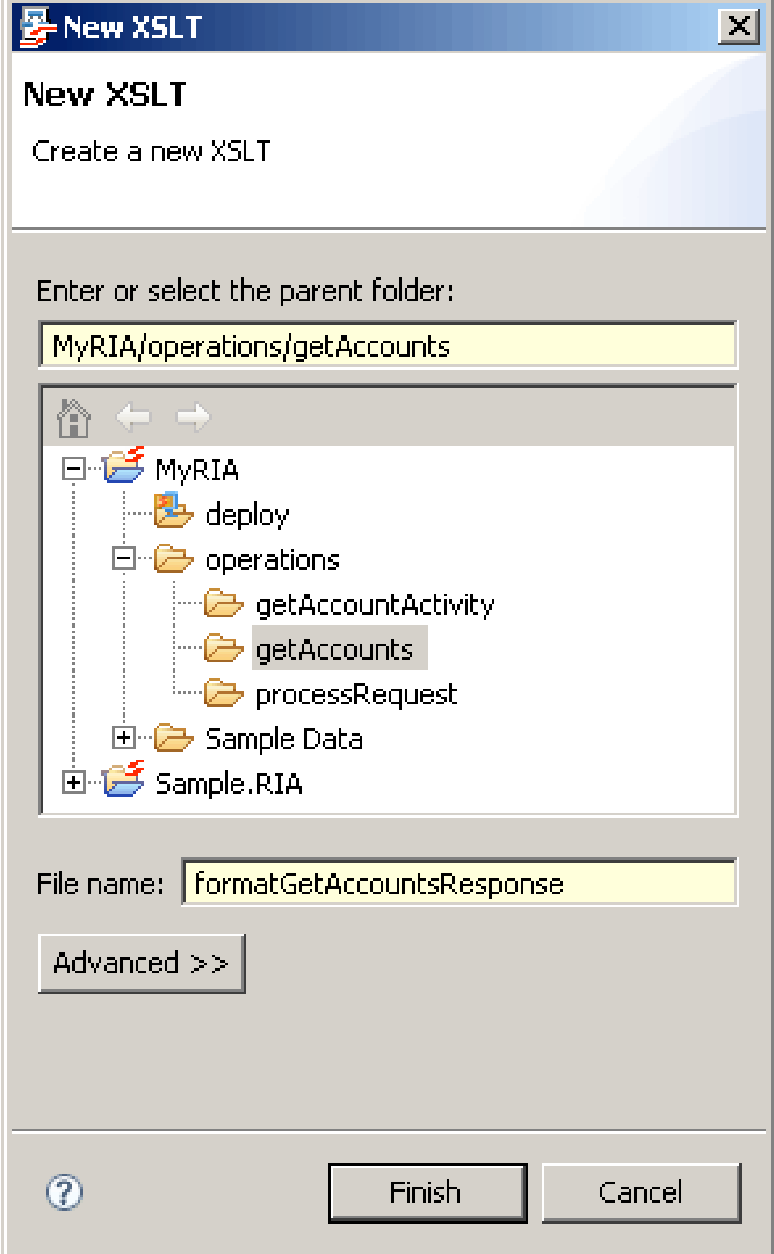
Creating a stylesheet to map response formats

Now that you have added an XML Transformation service, Format Response, to the getAccounts subprocess, you can create a stylesheet to map response formats. The sample XML document, `GetAccountsDefaultResponse.xml`, contains the formats required in the response from getAccounts. In the following procedure, you map the output of the Split and Join Parallel service, CombineAllAccounts, to these formats. You use the output that you saved previously when running CombineAllAccounts as input to the stylesheet in the XML Transformation service. First, create the stylesheet:

1. Double-click the **Format Response** step (this step is the XML Transformation service you added previously).
2. In the **Runtime Parameters** section of the Service page that opens, click in the **Stylesheet URL** field and select **New > XSLT** to create a new stylesheet for the XML Transformation service:



3. In the **New XSLT** dialog box, select the location **MyRIA/operations/getAccounts**, and enter the file name **formatGetAccountsResponse**:



Click **Finish** to create the stylesheet. The stylesheet opens in the XSLT editor:

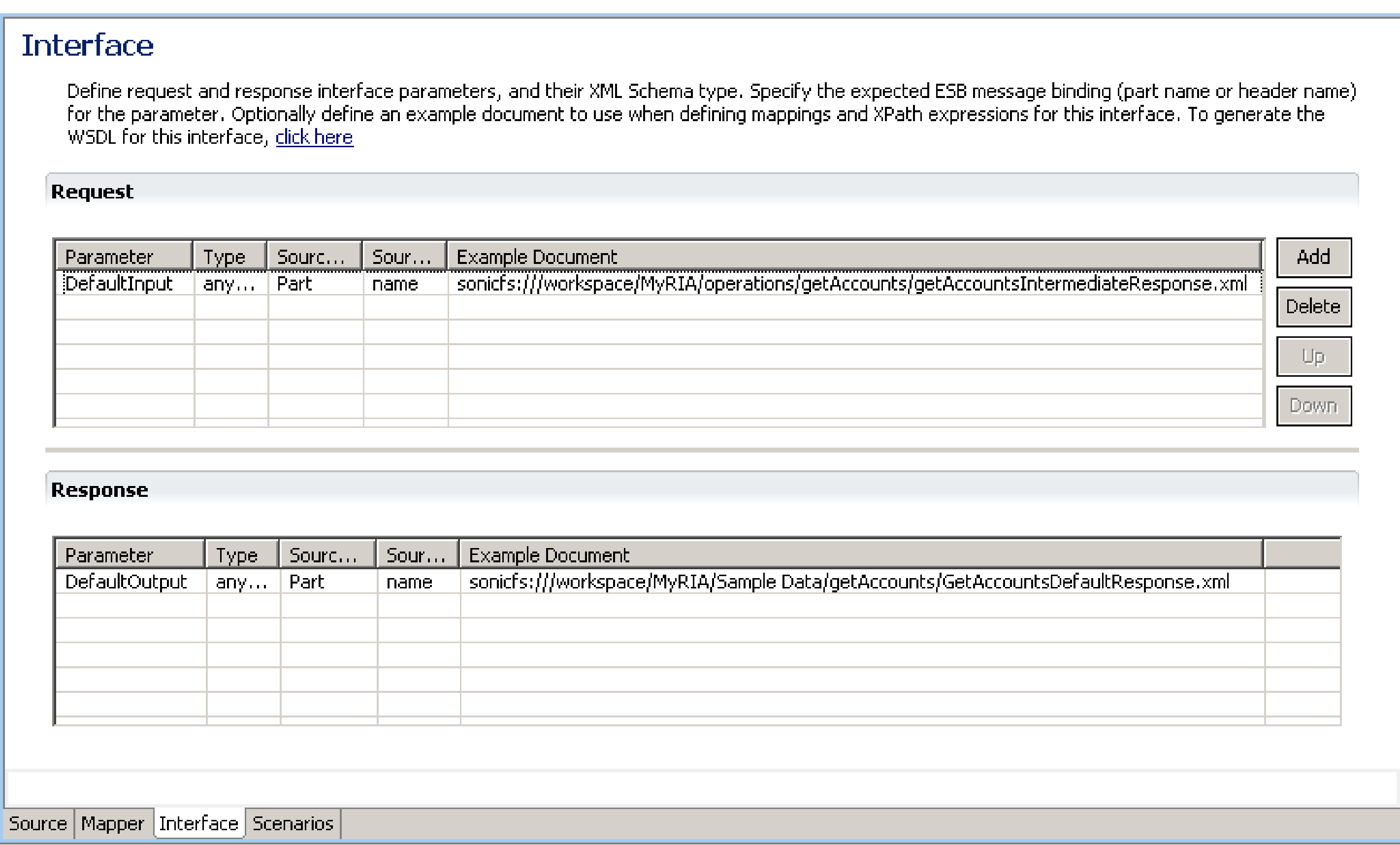
Next, select interface parameters that provide request and response formats in example documents for the XSL stylesheet.

Selecting interface parameters

Now that you have created a stylesheet for the XML Transformation service, Format Response, you can select example XML documents to use as request and response interface parameters for your stylesheet. For this tutorial, use the output that you saved previously when running the Split and Join Parallel service as input to the stylesheet transformation. For the output, use the sample XML document, `GetAccountsDefaultResponse.xml`, which contains the formats required in the response from the subprocess, getAccounts:

1. With the stylesheet `formatGetAccountsResponse.xsl` open, select the **Interface** tab to open the Interface page.
2. Enter the **Request** and **Response** interface parameters by dragging the following documents from the Navigator view into the **Example Document** fields:
   - **Default Input**: `getAccountsIntermediateResponse.xml` (located in MyRIA/operations/getAccounts)
   - **Default Output**: `GetAccountsDefaultResponse.xml` (located in MyRIA/Sample Data/getAccounts)

   The Interface page now looks like this:



Next, map the response parameters in the stylesheet.

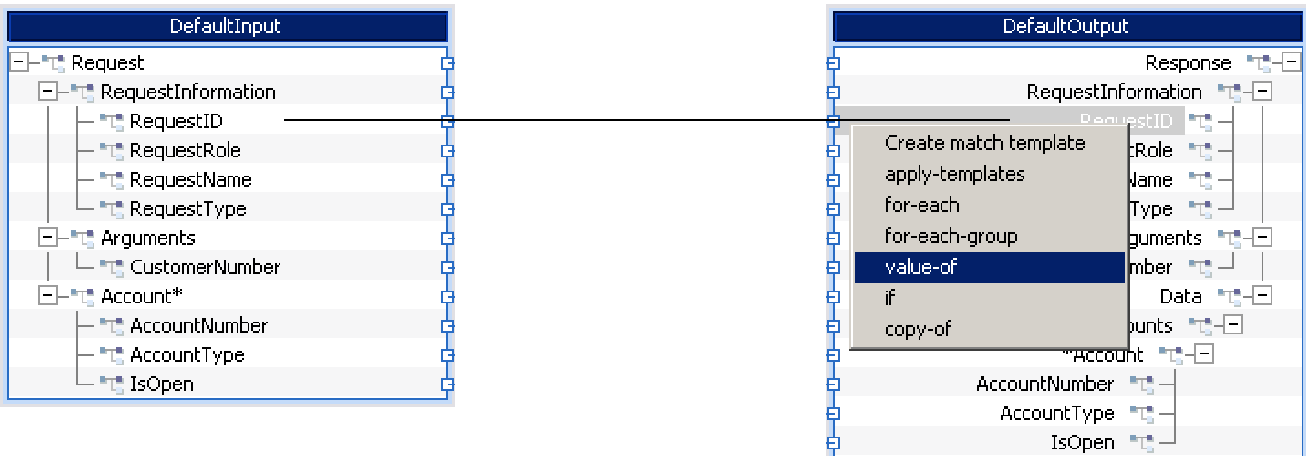
Mapping response parameters

Now that you have selected interface parameters for the stylesheet, you can map the request formats to the response formats using the Mapper tool in Sonic Workbench. The stylesheet is generated as you map from request to response parameters:

1. With the stylesheet `formatGetAccountsResponse.xsl` open, select the **Mapper** tab to open the Mapper page. Initially, the Mapper displays the input and output formats and a default stylesheet:

2. Create the mapping by clicking on an input node and dragging your cursor to an output node. Release the cursor and select the type of mapping. In this stylesheet, you map the values from input to output parameters, so you choose **value-of** mappings for most of the parameters. In the case of the Account node, you choose a **for-each** mapping because this is a repeating block, and you want to map the values of the parameters in this repeating block for each account for which data is returned.

   For example, map RequestID to RequestID, and select a **value-of** mapping:



Map the following parameters:

Input Node	Output Node	Mapping
RequestID	RequestID	value-of
RequestRole	RequestRole	value-of
RequestName	RequestName	value-of
RequestType	RequestType	value-of
CustomerNumber	CustomerNumber	value-of
Account*	*Account	for-each
AccountNumber	AccountNumber	value-of
AccountType	AccountType	value-of
IsOpen	IsOpen	value-of

Note that nodes having an asterisk, such as the Account node, are repeating blocks. In this case, you select the mapping **for-each** to repeat the mapping for each account.

3. View the completed stylesheet in the Source page:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <Response>
            <RequestInformation>
                <RequestID>
                    <xsl:value-of select="Request/RequestInformation/RequestID"></xsl:value-of>
                </RequestID>
                <RequestRole>
                    <xsl:value-of select="Request/RequestInformation/RequestRole"></xsl:value-of>
                </RequestRole>
                <RequestName>
                    <xsl:value-of select="Request/RequestInformation/RequestName"></xsl:value-of>
                </RequestName>
                <RequestType>
                    <xsl:value-of select="Request/RequestInformation/RequestType"></xsl:value-of>
                </RequestType>
            </RequestInformation>
            <Arguments>
                <CustomerNumber>
                    <xsl:value-of select="Request/Arguments/CustomerNumber"></xsl:value-of>
                </CustomerNumber>
            </Arguments>
            <Data>
                <Accounts>
                    <xsl:for-each select="Request/Account">
                        <Account>
                            <AccountNumber>
                                <xsl:value-of select="AccountNumber"></xsl:value-of>
                            </AccountNumber>
                            <AccountType>
                                <xsl:value-of select="AccountType"></xsl:value-of>
                            </AccountType>
                            <IsOpen>
                                <xsl:value-of select="IsOpen"></xsl:value-of>
                            </IsOpen>
                        </Account>
                    </xsl:for-each>
                </Accounts>
            </Data>
        </Response>
    </xsl:template>
</xsl:stylesheet>
```

4. Save the completed stylesheet.

Next, test the stylesheet by running a scenario and confirming that the output has the correct formats.

Testing the stylesheet

Now that you have mapped the response parameters, you can test the stylesheet using a scenario to confirm that the response formats are correct. By default, the stylesheet transforms the example document you selected as the interface parameter, DefaultInput. In this case, that document is the XML file, `getAccountsIntermediateResponse.xml`, which supplies the output of the Split and Join Parallel service, CombineAllAccounts.

To test the stylesheet using a scenario:

1. Open the stylesheet, `formatGetAccountsResponse.xsl`, and click the **Scenarios** tab to open the Scenarios page.
2. Click **Add Scenario** . Observe that the new scenario has the **Test Value** from the example document, `getAccountsIntermediateResponse.xml`.
3. Click **Run**. The stylesheet transforms the default input document, which provides the output from CombineAllAccounts and contains data for each account (Phone, TV, and Wireless Cell). The stylesheet maps the account data to the required response format.
4. View the **Reply Message** in the Output view:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccounts</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
   </Arguments>
   <Data>
      <Accounts>
         <Account>
            <AccountNumber>9999999</AccountNumber>
            <AccountType>TVAccount</AccountType>
            <IsOpen>true</IsOpen>
         </Account>
         <Account>
            <AccountNumber>9999999</AccountNumber>
            <AccountType>PhoneAccount</AccountType>
            <IsOpen>true</IsOpen>
         </Account>
         <Account>
            <AccountNumber>9999999</AccountNumber>
            <AccountType>WirelessCellAccount</AccountType>
            <IsOpen>true</IsOpen>
         </Account>
      </Accounts>
   </Data>
</Response>
```

Observe that the output is now in a Response element, and all account data is included in a single message part.

Next, run the subprocess, getAccounts, to confirm that the subprocess response is in the required format.

Running and testing the getAccounts subprocess

Now that you have implemented the CombineAllAccounts and Format Response services in the subprocess, getAccounts, you can run getAccounts using a scenario to confirm that a properly formatted response is returned containing data for all three accounts (Phone, TV, and Wireless Cell):

1. Open `RIA.getAccounts.esbp` and click the **Scenarios** tab to open the Scenarios page.
2. Select the **GetAccounts** scenario and click **Run**. The process runs, with the Split and Join Parallel service, CombineAllAccounts, calling the three ESB processes for the Phone, TV, and Wireless Cell account, and returning data for each. The XML Transformation service, Format Response, maps the account data to the required response format.
3. View the **Reply Message** in the Output view:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccounts</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
  </Arguments>
  <Data>
    <Accounts>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>TVAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>PhoneAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>WirelessCellAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
    </Accounts>
  </Data>
</Response>
```

   Observe that the output is now in a Response element, and all account data is included in a single message part.

Now that you have successfully implemented and tested the GetAccounts branch of processRequest, you are ready to continue on with Phase 5. In Phase 5 you implement the GetAccountActivity branch of processRequest by refactoring GetAccountActivity as a subprocess using content-based routing to retrieve data from specified account types.

Note: If you do not want to develop and implement all the phases of processRequest yourself, you can stop here, and run and test the sample files included in Sample.RIA. The sample ESB processes are similar to the processes created and implemented in this tutorial, and include scenarios to run the processes. Simply open the sample ESB process you want to test and proceed directly to the instructions to run and test the process:

- `Sample.RIA.GetAccounts.esbp` — Follow the instructions in Running and testing the getAccounts subprocess
- `Sample.RIA.getAccountActivity.esbp` — Follow the instructions in Running and testing the getAccountActivity subprocess
- `Sample.RIA.processRequest.esbp` — Follow the instructions in Testing the fully implemented ESB process, processRequest

Phase 5: Implementing getAccountActivity using content-based routing

In this phase of the RIA tutorial, you refactor the GetAccountActivity step in processRequest as a subprocess to retrieve account data for each account type used in this tutorial, and to aggregate that data into a single response:

1. Refactor GetAccountActivity — Refactor this step as a subprocesss and add an operation router that will aggregate data for separate accounts.
2. Configure three branches of the operation router — Configure three branches, one to get account activity for each of the three account types (Phone, TV, and Wireless Cell accounts).
3. Modify the subprocess routing rules — Create XPath routing rules that will route messages based on the account type in the message content. Configure three rules:
    - Rule 1 — Send requests for Phone account activity to the Phone account branch.
    - Rule 2 — Send requests for TV account activity to the TV account branch.
    - Rule 3 — Send requests for Wireless Cell account activity to the Wireless Cell account branch.
4. Create scenarios to test the subprocess routing — Create three scenarios for the subprocess. The scenarios test each branch of the content-based router.
5. Run and test the subprocess with request routing — Run the scenarios to confirm that messages are routed correctly based on the account type in the message content.

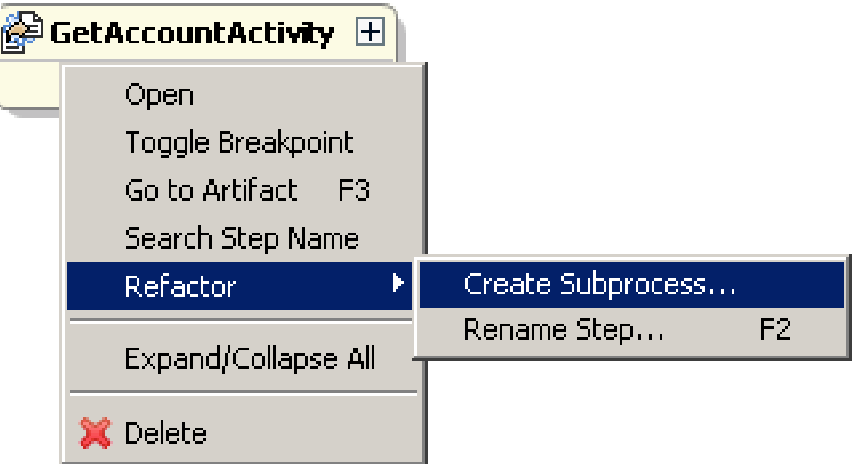
Start by refactoring GetAccountActivity as a subprocess.

RIA tutorial, phase 5

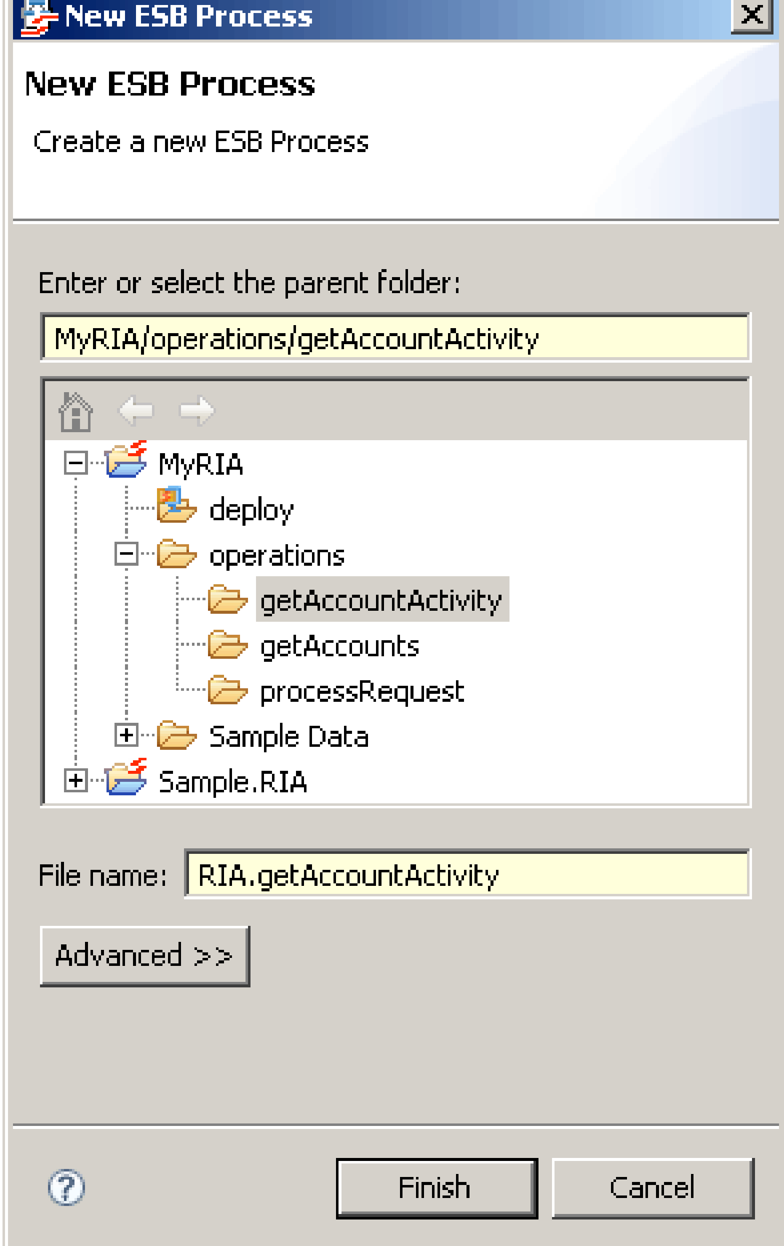
Refactoring GetAccountActivity as a subprocess

In Phase 2, you created a content-based router having two branches. Now you can refactor the service on one of those branches, GetAccountActivity, as a subprocess to retrieve data from specified a account type. In the subprocess, you create a content-based router that routes requests for account activity based on the account type specified in the request. To begin, refactor GetAccountActivity as a subprocess and add an operation router:

1. Open `RIA.processRequest.esbp`, right-click the **GetAccountActivity** step, and select **Refactor >Create Subprocess**:

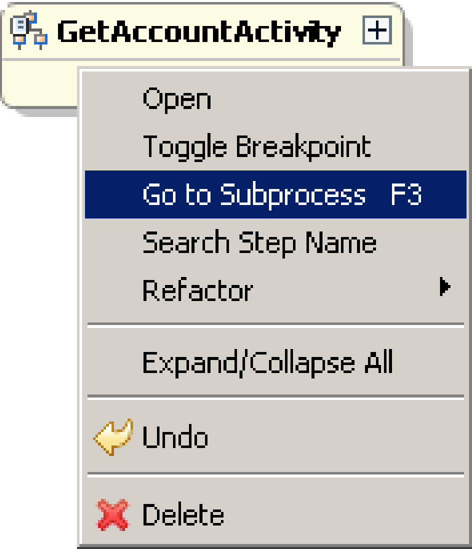


2. In the **New ESB Process** dialog box that opens, select the parent folder *MyRIA/operations/getAccountActivity* and enter the file name *RIA.getAccountActivity*:

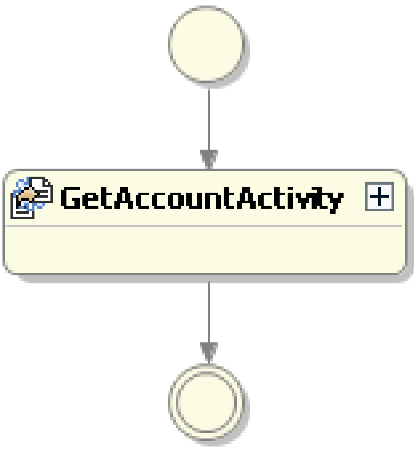


Click **Finish** to create the subprocess. Notice that the icon on the GetAccountActivity step changes to indicate that the step is now a subprocess.

3. Right-click the **GetAccountActivity** step and select **Go to Subprocess**:



The new subprocess opens, containing a single step, GetAccountActivity:



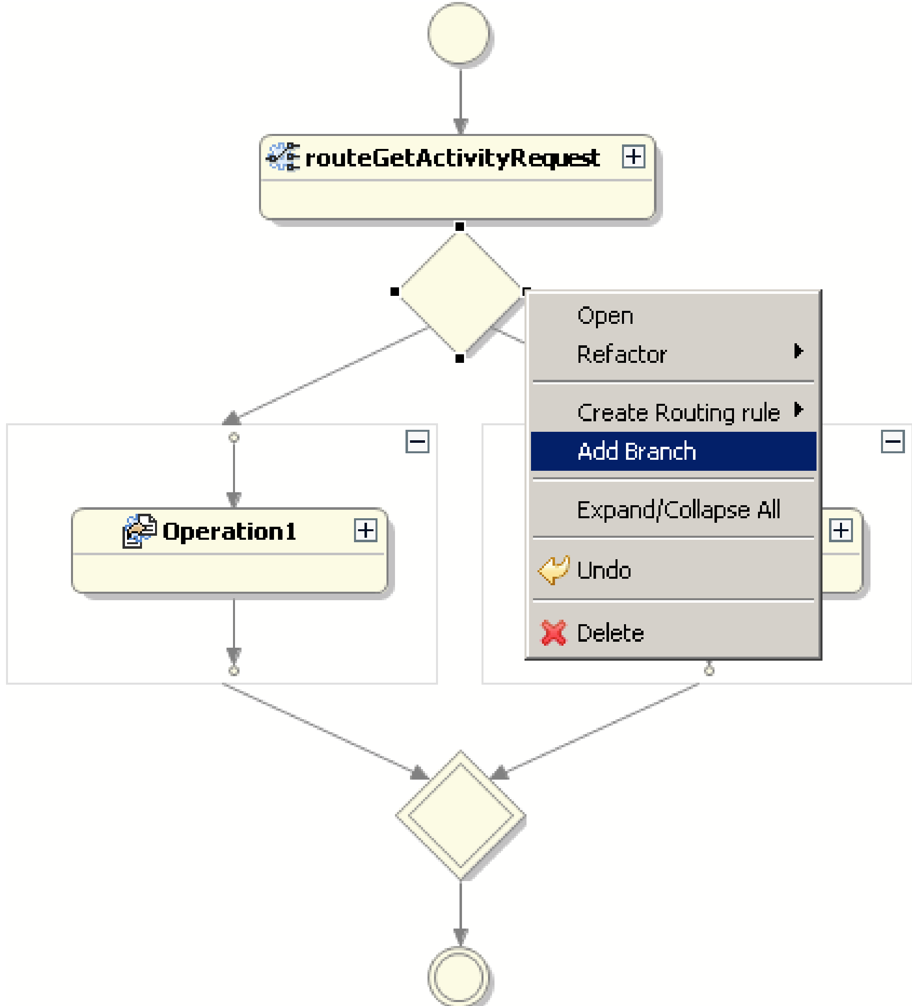
4. From the **Process Templates** section of the Palette, drag an operation router OperationRouter onto the process, and delete the existing GetAccountActivity step (you are replacing this prototype step with the operation router).
5. Save the subprocess.

Next, configure three branches of the operation router to return data for each account type.

Configuring three branches of the operation router

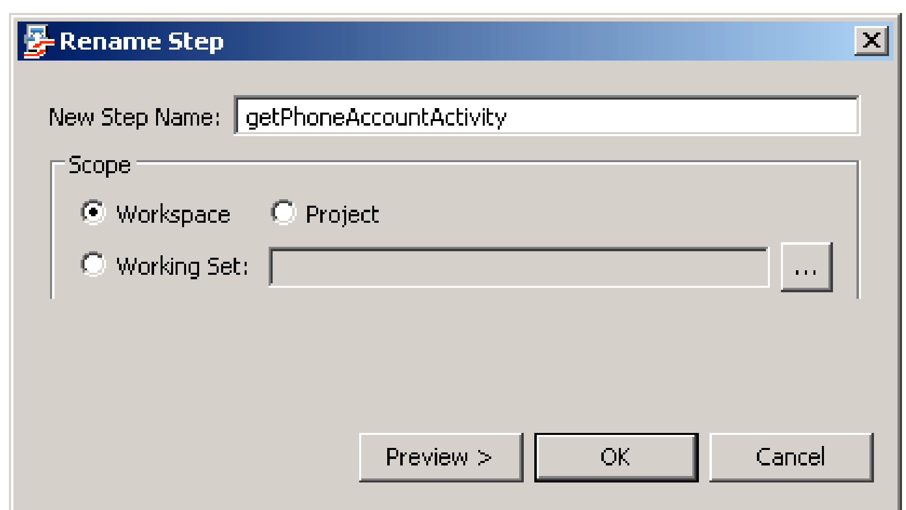
Now that you have refactored GetAccountActivity as a subprocess with an operation router, you can rename the operation router and create three branches to provide data from the different account types (Phone, TV, and Wireless Cell accounts):

1. Open `RIA.getAccountActivity.esbp` and select the new operation router step. Click the step so you can rename it. Change the step name to *routeGetActivityRequest*.
2. When you create the operation router, an XPath routing rules file is created with the default name **OperationRouter.xcbr** and saved in the same location as the ESB process in which it is created. To rename this file (to avoid confusing it with other routing rules files), select the file in the Navigator view, right-click, and select **Rename**.
3. Change the name to *routeGetActivityRequest.xcbr*, and confirm the name refactoring when prompted. Click **Yes** in the **Save All Modified Resources** dialog box, then click **OK** in the **Sonic Rename Processor** dialog box. Finally, select **MyRIA** in the Navigator view and choose **Project > Upload All** from the menu bar to upload the renaming changes.
4. The prototype operation router contains two branches. To create a third branch, right-click the decision step and select **Add Branch**:



The process now looks like this:

5. For each branch of the subprocess (one for each type of account), select the operation or service step, right-click, and select **Refactor > Rename Step**. In the **Rename Step** dialog box that opens, enter the new step name. Rename the step Operation 1 *getPhoneAccountActivity*:



Click **OK** to rename the step. The step name is automatically refactored.

Repeat this step to rename remaining branches:

- *getTVAccountActivity*
- *getWirelessCellAccountActivity*

Note: If you do not save the process after renaming each step, you will be prompted to save all modified resources before refactoring.

6. Next, add XML files containing a default response for each step by dragging an XML file from the folder MyRIA/ Sample Data/GetAccountActivity in the Navigator view:
   - Drag `GetPhoneAccountActivityResponse.xml` onto the **getPhoneAccountActivity** step.
   - Drag `GetTVAccountActivityResponse.xml` onto the **getTVAccountActivity** step.
   - Drag `GetWirelessCellAccountActivityResponse.xml` onto the **getWirelessCellAccountActivity** step.

   Remember that you can right-click any of the service steps and select **Go to Artifact** to view the response XML file that you have configured the step with.

Next, configure the routing rules to route to the three branches.

Modifying the subprocess routing rules

Now that you have configured three branches for the operation router, you can modify the routing rules to configure the getAccountActivity subprocess to route requests based on the account type specified in the incoming request. In this case, there are 3 different types of accounts (Phone, TV, and Wireless Cell), so you must configure a routing rule for each account type:

1. In `RIA.getAccountActivity.esbp`, right-click the **routeGetActivityRequest** step and select **Go to Artifact** to open `routeGetActivityRequest.xcbr`. The file opens in the XPath Routing Rules editor, and has two routing rules, one for the Phone account branch and one for the TV account branch:

   Rules Condition Section

   Routing rules used for Routing.

Context	XPath Expression	Rules Address
MessagePar...	/OperationRequ...	STEP:getPhoneAccountActivity
MessagePar...	/OperationRequ...	STEP:getTVAccountActivity

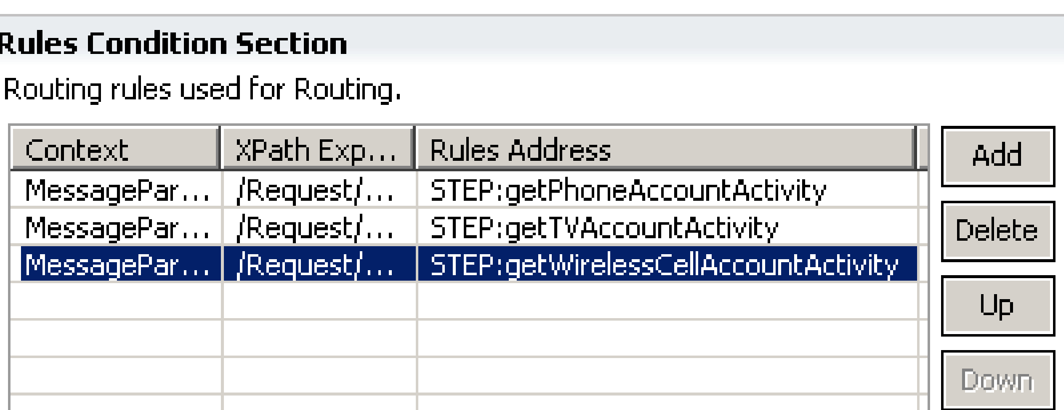
   Add | Delete | Up | Down

2. In the **Rules Condition Section**, select the rule for the **getPhoneAccountActivity** branch.
3. In the **XPath Expression** section, click **...** next to the default XPath expression.
4. In the **XPath Helper**, select a sample **Input Document** by browsing to sonicfs:///workspace/MyRIA/Sample Data/getAccountActivity/GetPhoneAccountActivityRequest.xml.
5. To create a rule that routes to the Phone account branch, double-click the node **Request/Arguments/Account/AccountType**. Notice that the **XPath** field in the **Input** section now contains the expression **Request/Arguments/Account/AccountType/text()**.
6. In the XPath field, next to the expression you just added, enter: **=´PhoneAccount´**. Click **Evaluate** to confirm that this XPath expression evaluates to **true**.
7. In the **Rules Address** section of the XPath Routing Rules editor, confirm that the address for this rule is the step **getPhoneAccountActivity**.
8. You can modify the rules for the remaining branches by copying the XPath expression you just created. Select the routing rule for the TV account branch and paste the copied XPath expression into the **XPath Expression** field for the selected rule. Edit the XPath expression as follows: *Request/Arguments/Account/AccountType/text()**=´TVAccount´***
9. To create a rule for the **getWirelessCellAccountActivity** branch, click **Add** in the **Rules Condition Section**. A new rule is added to the table.
10. Paste the copied XPath expression into the **XPath Expression** field for the new rule. Edit the XPath expression as follows:

*Request/Arguments/Account/AccountType/text()***=´WirelessCellAccount´**

11. In the **Rules Address** section, click **Add**. In the **Add Destination** dialog box, enter the name *getWirelessCellAccountActivity*.

    There are now three routing rules, one for each branch of the routeGetActivityRequest operation router:



12. Save the modified routing rules file.

Next, create scenarios to test the routing in the getAccountActivity subprocess.

Creating scenarios to test the routing in getAccountActivity

Now that you have refactored GetAccountActivity as a subprocess and added a content-based router configured with XPath routing rules to route requests for account activity for specified accounts, you can create three scenarios to test the routing to each branch:

1. Open `RIA.getAccountActivity.esbp` and click the **Scenarios** tab to open the Scenarios page.
2. In the **Scenario** section, click **Add Scenario** to create a new scenario. By default, the new scenario is named RIA.getAccountActivity_default.
3. In the **Scenario Details** section, enter or select the following:
   - **Scenario name**: *getPhoneAccountActivity*.
   - In the **Input** section, select **Interface**.
   - If the **File/Literal** selection in the **Input** table is not already set to **File**, click the entry in the field and select **File** from the pull-down list.
   - Enter a **Scenario Test Value** by dragging the sample XML file, **GetPhoneAccountActivityRequest.xml**, from the folder Sample Data \getAccountActivity in the Navigator window.
4. In the **Scenario** section, select the scenario you created and click **Duplicate Scenario** twice. Modify the two duplicated scenarios with the values required to test the TV and Wireless Cell account branches:

Scenario name	Scenario Test Value
getTVAccountActivity	`GetTVAccountActivityRequest.xml`
getWirelessCellAccountActivity	`GetWirelessCellAccountActivityRequest.xml`

Next, run the process using these scenarios.

Running and testing the getAccountActivity subprocess

Now that you have refactored GetAccountActivity as a subprocess and added a content-based router configured with XPath routing rules to route requests for account activity for specified accounts, you can run the getAccountActivity subprocess to test each branch of the routing. Run getAccountActivity using the three scenarios you created previously:

1. Open `RIA.getAccountActivity.esbp` and click the **Scenarios** tab to open the Scenarios page.
2. Run the **GetPhoneAccountActivity** scenario to see that the subprocess returns Phone account information, demonstrating that the request was routed through the GetPhoneAccountActivity branch:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccountActivity</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
    <Account>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>PhoneAccount</AccountType>
    </Account>
  </Arguments>
  <Data>
    <AccountActivity>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>PhoneAccount</AccountType>
      <OutstandingBalance>33.88</OutstandingBalance>
    </AccountActivity>
  </Data>
</Response>
```

3. Select the **GetTVAccountActivity** scenario and click **Run**. The **Reply Message** in the Output view shows the result returned by the subprocess, which in this case contains TV account information:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
      </Account>
   </Arguments>
   <Data>
      <AccountActivity>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
         <OutstandingBalance>44.99</OutstandingBalance>
      </AccountActivity>
   </Data>
</Response>
```


4. Run the **GetWirelessCellAccountActivity** scenario to see that the subprocess returns Wireless Cell account information, demonstrating that the request was routed through the correct branch:


```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>WirelessCellAccount</AccountType>
      </Account>
   </Arguments>
   <Data>
      <AccountActivity>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>WirelessCellAccount</AccountType>
         <OutstandingBalance>123.56</OutstandingBalance>
      </AccountActivity>
   </Data>
</Response>
```

You have now successfully implemented and tested the GetAccountActivity branch of processRequest. In Phase 3 and Phase 4 you implemented and tested the GetAccounts branch of processRequest. You are now ready to test the fully implemented ESB process, processRequest.

Note: If you do not want to develop and implement all the phases of processRequest yourself, you can run and test the sample files included in Sample.RIA. The sample ESB processes are similar to the processes created and implemented in this tutorial, and include scenarios to run the processes. Simply open the sample ESB process you want to test and proceed directly to the instructions to run and test the process:

- `Sample.RIA.GetAccounts.esbp` — Follow the instructions in Running and testing the getAccounts subprocess
- `Sample.RIA.getAccountActivity.esbp` — Follow the instructions in Running and testing the getAccountActivity subprocess
- `Sample.RIA.processRequest.esbp` — Follow the instructions in Testing the fully implemented ESB process, processRequest

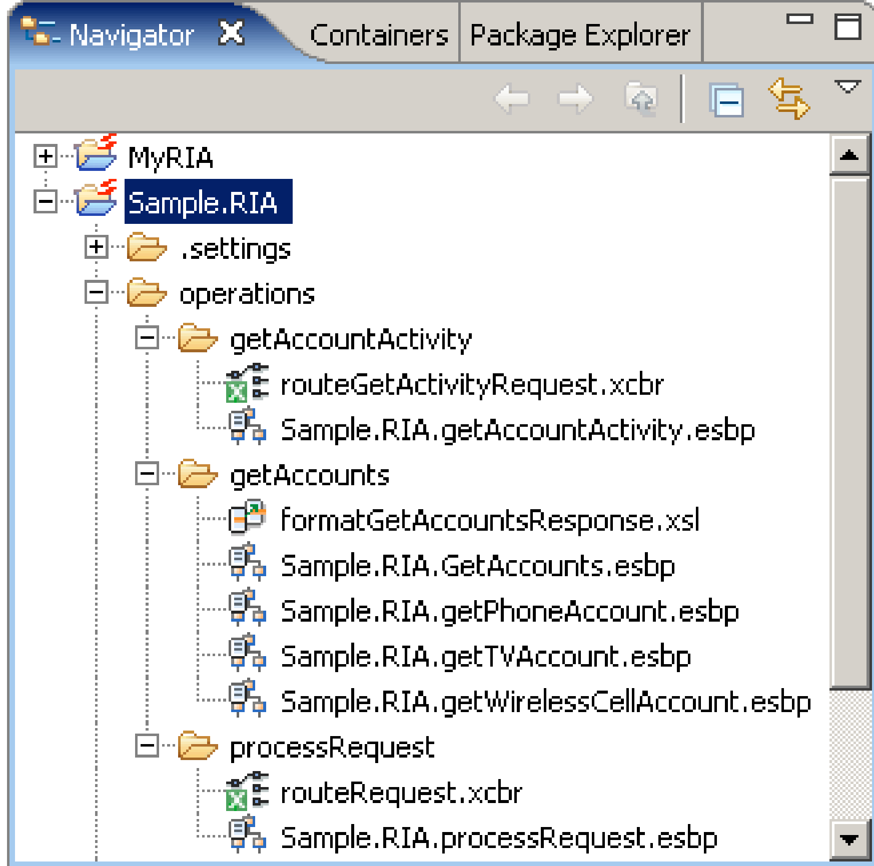
Testing the fully implemented ESB process, processRequest

Now that you have completed all five phases of implementation, you are ready to test and debug processRequest. The ESB process has branches to handle the two use cases in the tutorial: getting account information for a specified customer, and getting account activity on a specified account. Using different scenarios, you can test both branches of processRequest. You can also debug the process to examine both branches, stepping into subprocesses and examining the output of each step:

1. Test processRequest — Test the fully implemented ESB process, processRequest, using scenarios to test each branch of the process.
2. Debug processRequest — Step through the different branches of processRequest using scenarios to examine each branch.

Start by testing processRequest.

Note: If you do not want to develop and implement all the phases of processRequest yourself, you can run and test the sample files included in Sample.RIA. The sample ESB processes are similar to the processes created and implemented in this tutorial, and include scenarios to run the processes. Open the sample ESB processes by double-clicking the files in the Navigator view, in the folder Sample.RIA/operations:



Simply open the sample ESB process you want to run and proceed directly to the instructions to run and test that ESB process:

- `Sample.RIA.GetAccounts.esbp` — Follow the instructions in Running and testing the getAccounts subprocess
- `Sample.RIA.getAccountActivity.esbp` — Follow the instructions in Running and testing the getAccountActivity subprocess
- `Sample.RIA.processRequest.esbp` — Follow the instructions in Testing the fully implemented ESB process, processRequest

Testing processRequest

You can run the ESB process, processRequest, using different scenarios to examine different paths through the process. You can reuse the getAccounts and getAccountActivity scenarios you used when testing the routing of the prototype content-based router:

1. Open `RIA.processRequest.esbp` and click the **Scenarios** tab to open the Scenarios page.
2. Select the **getAccounts** scenario and click **Run**. This scenario supplies a request having request type **getAccounts**. The process runs:
    a. The content-based router, **routeRequest**, evaluates the request type and routes the request to the GetAccounts branch.
    b. The Split and Join Parallel service, **CombineAllAccounts**, calls the three ESB processes for the Phone, TV, and Wireless Cell account, which return data for each account.
    c. The XML Transformation service, **Format Response**, maps the account data to the required response format.
3. View the **Reply Message** in the Output view:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccounts</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
  </Arguments>
  <Data>
    <Accounts>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>TVAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>PhoneAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>WirelessCellAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
    </Accounts>
  </Data>
</Response>
```

Compare this output with the response you got when running the fully implemented subprocess, getAccounts. The outputs are identical.

4. Select the **GetAccountActivity** scenario and click **Run**. This scenario supplies a request having request type **getAccountActivity** and account type **TVAccount**. The process runs:
    a. The content-based router, **routeRequest**, evaluates the request type and routes the request to the GetAccountActivity branch.
    b. The content-based router, **routeGetActivityRequest**, evaluates the account type and routes the request to the GetTVAccountActivity branch.
    c. The **GetTVAccountActivity** step returns account activity data for the TV account.
5. View the **Reply Message** in the Output view:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccountActivity</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
    <Account>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>TVAccount</AccountType>
    </Account>
  </Arguments>
  <Data>
    <AccountActivity>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>TVAccount</AccountType>
      <OutstandingBalance>44.99</OutstandingBalance>
    </AccountActivity>
  </Data>
</Response>
```

Compare this output with the response you got when running the fully implemented subprocess, GetAccountActivity using the getTVAccountActivity scenario. This scenario and the getAccountActivity scenario both supply the account type TVAccount in the request, and, as expected, the outputs are identical.

You have now completed and successfully tested the fully implemented ESB process, processRequest.

You can create new scenarios to test other branches of the getAccountActivity subprocess. Try using GetPhoneAccountActivityRequest.xml or GetWirelessCellAccountActivityRequest.xml as inputs in new scenarios to test the Phone and Wireless Cell account branches.

Optionally, you can continue on to see how to debug processRequest using these two scenarios.

[Testing processRequest](#)

Debugging processRequest

You can run scenarios to debug every path through an ESB process. In this tutorial, you run scenarios to debug one path through each of the branches of processRequest. Debugging an ESB process involves setting breakpoints in the process and any subprocesses you want to examine, then stepping through the breakpoints as the ESB process runs:

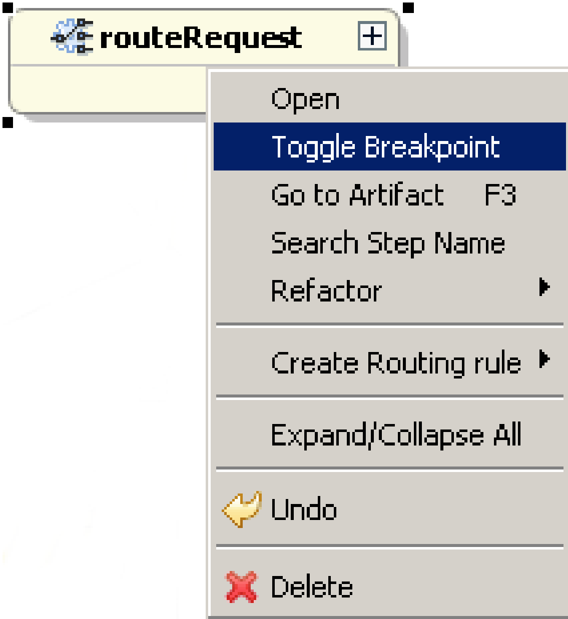
1. Set a breakpoint — Set a breakpoint on the the first step in processRequest so you can visually step through and debug the process as it runs.
2. Debug processRequest and getAccounts — Using the getAccounts scenario, observe the debugging information in the Breakpoints, Debug, and ESB Variables views at the first breakpoint, where the request routing is determined.
3. Step through breakpoints using the getAccounts scenario — Observe the debugging information in the ESB Variables view as you step through processRequest and getAccounts.
4. Debug processRequest and getAccountActivity — Using the getAccountsActivity scenario, observe the debugging information in the Breakpoints, Debug, and ESB Variables views at the first breakpoint, where the request routing is determined.
5. Step through breakpoints using getAccountActivity — Observe the debugging information in the ESB Variables view as you step through processRequest and one branch of getAccountActivity.

Start by setting a breakpoint.

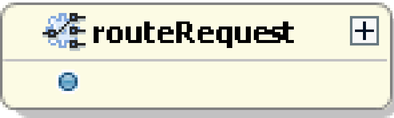
Setting a breakpoint

You can set breakpoints to visually step through and debug processes as they run. By setting a breakpoint on the first step in processRequest, you can step through and debug the process and its subprocesses:

1. Open `RIA.processRequest.esbp`. Select the **routeRequest** step, right-click, and select **Toggle Breakpoint**:



2. Observe the small circle on the step denoting that there is a breakpoint on that step:



Next, start debugging processRequest using the getAccounts scenario.

Testing processRequest

Debugging processRequest using the getAccounts scenario

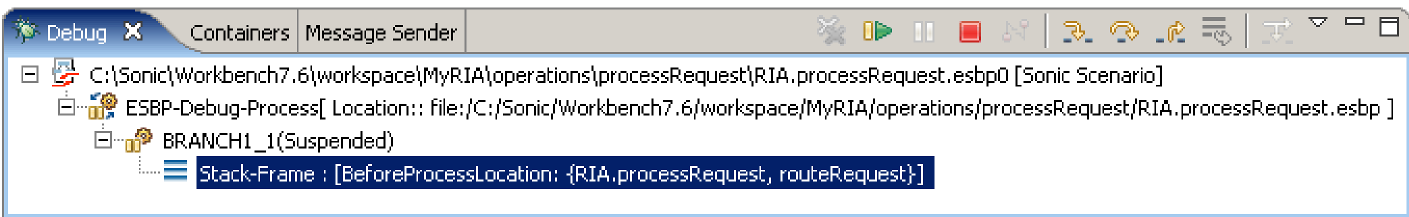
After setting a breakpoint, you can start debugging processRequest. Use the getAccounts scenario to debug the GetAccounts branch of processRequest. This scenario sends a message containing the request type **getAccounts**, which the content-based router **routeRequest** will send to the **GetAccounts** branch of processRequest:

1. Click the **Scenarios** tab to open the Scenarios page.
2. Select the **getAccounts** scenario.
3. Click **Debug** to start debugging the ESB process.
4. Observe that the perspective changes to the Sonic Debug perspective:

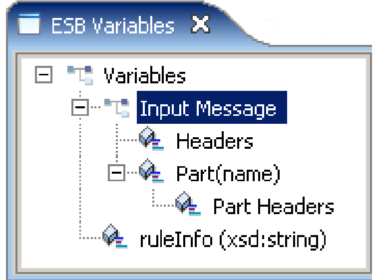
5. Observe that there is now an arrow to the left of the first step with a breakpoint, showing the current step in debugging:

6. Go to the Breakpoints view to see the breakpoint you set before the routeRequest step:

7. Go to the Debug view to view the stack frame location at the first breakpoint:

8. Select the stack frame and go to the ESB Variables view to view the message and variables:

9. Select the Input Message in the left pane and observe that the right pane shows the same data as in `GetAccountsRequest.xml`. This is the content of the request sent in the getAccounts scenario, which contains the request type **getAccounts**.

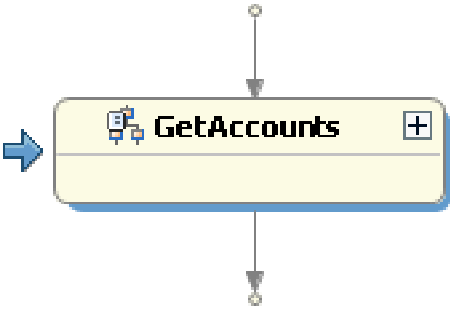
Next, step through the GetAccounts branch of processRequest.

[Testing processRequest](#)

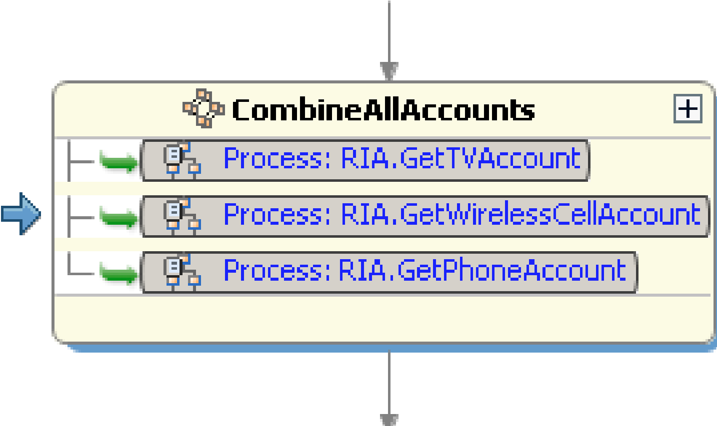
Stepping through processRequest and getAccounts

After starting the debugger, you can go to the next step in processRequest. Because the getAccounts scenario provides a request having request type **getAccounts**, the content-based router, routeRequest, sends the message to the GetAccounts branch:

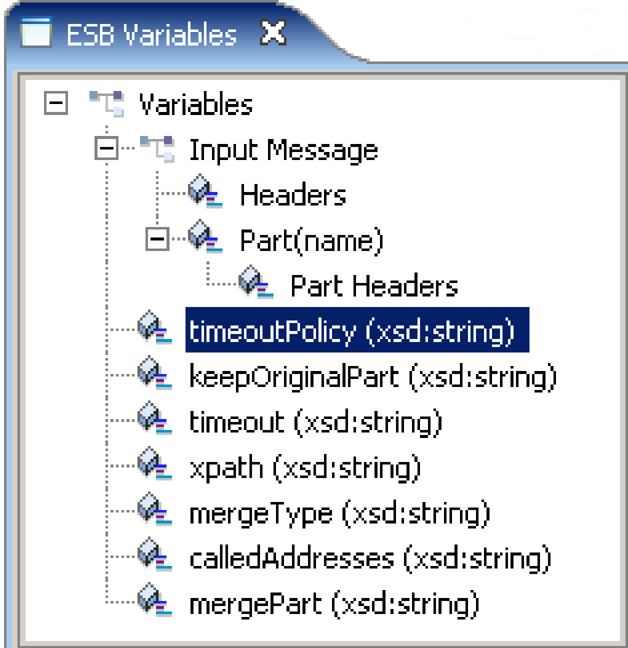
1. In the Debug view, click **Step Into** to go to the next step. Observe that the arrow is now on the **GetAccounts** step:



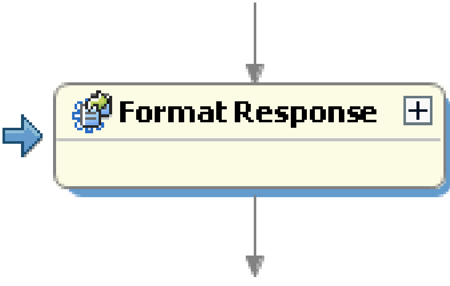
2. Click **Step Into** to go into the getAccounts subprocess. Observe that getAccounts opens in the Process view, and the arrow is now on the CombineAllAccounts step:



3. Go to the ESB Variables view and observe that the runtime parameters for CombineAllAccounts are listed under the Variables node:



4. Click **Step Into** to go to the Format Response step. The ESB Variables view now lists the Format Response service parameters, and the arrow moves to the Format Response step in the Process view:



5. Select the Input Message in the ESB Variables view and observe that the message now contains the information about each account type for the specified customer, which is the same as the output you got from CombineAllAccounts in Phase 3. This message is the input to the Format Response step. Observe that the response is not yet formatted by the Format Response step.
6. Click **Step Into** to complete the process. The Output view opens, containing the Reply Message, which is the output of the Format Response step, and of processRequest.
7. Select the Reply Message and observe the message in the right pane. Notice that the message is now formatted by the Format Response step, and is the same as the response you got when running the fully implemented subprocess getAccounts in Phase 4.

Next, debug processRequest using the getAccountActivity scenario.

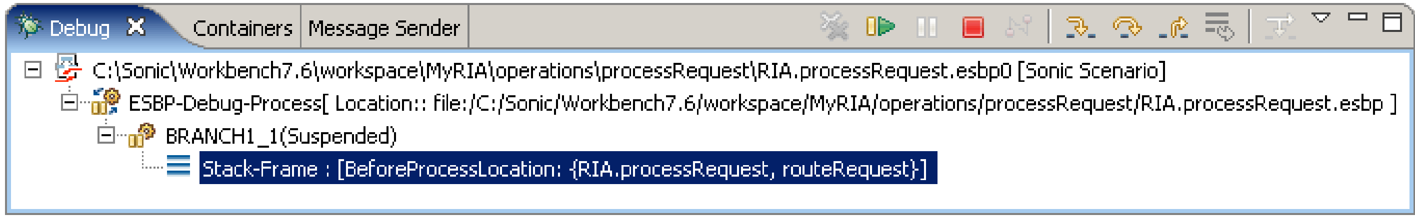
[Testing processRequest]

Debugging processRequest using the getAccountActivity scenario

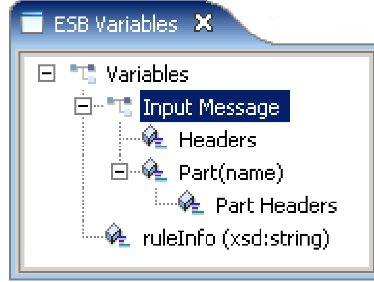
After setting a breakpoint in processRequest, you can use the getAccountActivity scenario to debug the GetAccountActivity branch of the process. This scenario sends a message containing the request type **getAccountActivity**, which the content-based router **routeRequest** will send to the **GetAccountActivity** branch of processRequest. The message also contains the account type **TVAccount**. When the message is sent to the RIA.getAccountActivity subprocess, the content-based router **routeGetActivityRequest** will send the request to the **getTVAccountActivity** branch of processRequest:

1. Select the **getAccountActivity** scenario.
2. Click **Debug** to start debugging the ESB process. The perspective changes to the Sonic Debug perspective and there is now an arrow to the left of the first step with a breakpoint, showing the current step in debugging:

3. Go to the Debug view to view the stack frame location at the first breakpoint:



4. Select the stack frame and go to the ESB Variables view to view the message and variables:



5. Select the Input Message in the left pane and observe that the right pane shows the same data as in `GetAccountActivityRequest.xml`. This is the content of the request sent in the getAccountActivity scenario, which contains the request type **getAccountActivity** and the account type **TVAccount**.

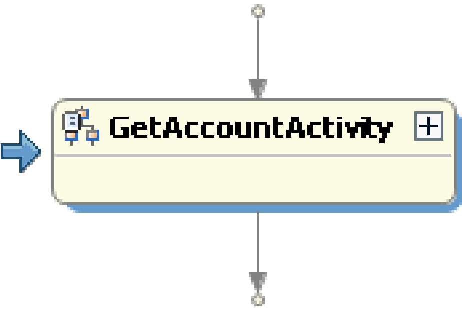
Next, step to the next breakpoint.

[Testing processRequest](#)

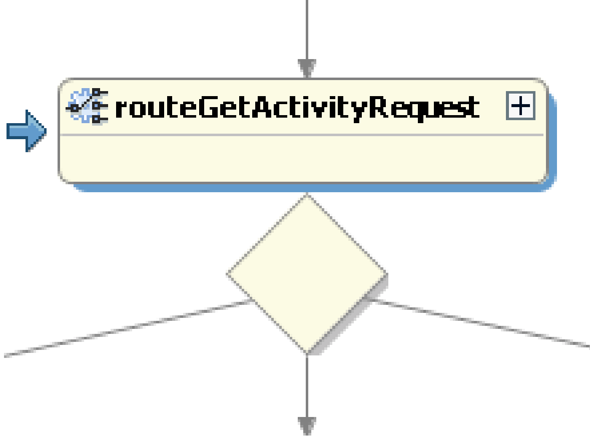
Stepping through processRequest and getAccountActivity

After [starting the debugger](#), you can go to the next step in processRequest. Because the getAccountActivity scenario provides a request having request type **getAccountActivity**, the content-based router, [routeRequest](#), sends the message to the GetAccountActivity branch:

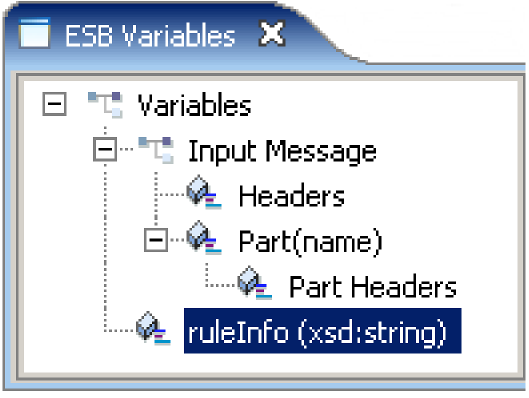
1. In the Debug view, click **Step Into** to go to the next breakpoint. Observe that the arrow is now on the **GetAccountActivity** step:



2. Click **Step Into** to go into the getAccountActivity subprocess. Observe that getAccountActivity opens in the Process view, and the arrow is now on the **routeGetActivityRequest** step:

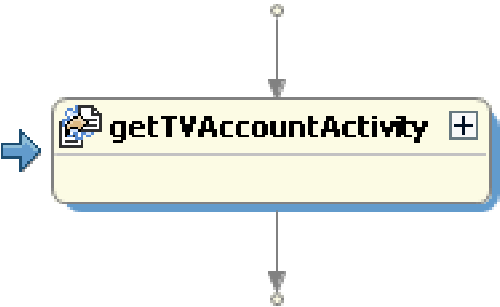


3. Go to the ESB Variables view and observe that **ruleInfo**, which contains the name and location of the XPath routing rules file for routeGetActivityRequest, is listed under the Variables node:



Select ruleInfo to view the XPath routing rules file name and location in the right pane.

4. Click **Step Into** to go to the next breakpoint. The routeGetActivityRequest router evaluates the XPath routing rules to determine that the account type is **TVAccount**, then sends the message to the **getTVAccountActivity** step:



5. In the ESB Variables view, select the Input Message and observe that the input to the getTVAccountActivity step is the request used in the getAccountActivity scenario, `GetAccountActivityRequest.xml`.
6. Click **Step Into** to complete the process. The Output view opens, containing the Reply Message.
7. Select the Reply Message and observe the message in the right pane. Notice that the message is the same as the response you got when running the fully implemented subprocess getAccountActivity in Phase 5, using the getTVAccountActivity scenario. This scenario and the getAccountActivity scenario both supply the account type TVAccount in the request, and, as expected, the outputs are identical.

You can create new scenaros to debug other branches of the getAccountActivity subprocess. Try using GetPhoneAccountActivityRequest.xml or GetWirelessCellAccountActivityRequest.xml as inputs in new scenarios to debug the Phone and Wireless Cell account branches of the process.

You have now completed the Remote Information Access tutorial. See the next steps after running the tutorial.

Next steps after developing the Remote Information Access sample application

If you have not run the Batch to Real-time tutorial, you can do so now. Batch to real time and remote information access are two of the most common enterprise integration scenarios that benefit from a Sonic ESB SOA solution.

Two other scenarios that benefit from a Sonic ESB SOA solution include:

- Remote data distribution
- Respond to real-time business events

You can go to the Progress Software Developers Network (PSDN) at `http://www.psdn.com` to learn more about these and other enterprise integration scenarios.

The Sonic Workbench online help contains information on other sample applications included in your Sonic ESB installation. (Access the sample documentation from **Help > Help Contents > Progress Sonic ESB Product Family: Developer's Guide > Progress Sonic ESB Samples and Tutorials**). These sample applications illustrate different features of Sonic ESB and provide a useful next step in getting acquainted with this product:

- Integration pattern samples
- Split and Join service samples
- File Handling samples
- Audit service sample
- File polling sample
- Resubmit sample

Sonic BPEL Server also provides:

- Sonic BPEL Server tutorial
- Sonic BPEL Server samples

You can also look on the online help to learn more about the concepts in the Remote Information Access tutorial.

Reference: Files in the Remote Information Access sample project

When you double-click the files in the Sample.RIA project in the Navigator view, they open in the appropriate editor in Sonic Workbench. You use the following files in this tutorial:

The **Operations** folder contains the following files:

- **XPath routing rules** — These files specify routing rules for the operation routers implemented in the tutorial:
  - routeGetActivityRequest.xcbr — Provides XPath routing rules for the ESB subprocess, Sample.RIA.getAccountActivity.esbp.
  - routeRequest.xcbr — Provides XPath routing rules for the ESB process, Sample.RIA. processRequest.esbp.
- **XSL stylesheet** — The stylesheet, formatGetAccountsResponse.xsl, formats the response in the ESB subprocess, Sample.RIA.GetAccounts.esbp.
- **ESB processes** — These ESB processes and subprocesses are implemented in phases as you progress through the tutorial:
  - Sample.RIA.processRequest.esbp
  - Sample.RIA.GetAccounts.esbp
  - Sample.RIA.getAccountActivity.esbp
  - Sample.RIA.getPhoneAccount.esbp
  - Sample.RIA.getTVAccount.esbp
  - Sample.RIA.getWirelessCellAccount.esbp

The **Sample Data** folder contains XML files that supply example requests and responses used when prototyping and testing the ESB processes in the tutorial. The files are separated into folders based on which ESB subprocess they relate to:

- **getAccountActivity** — Contains XML files used to prototype and test the ESB subprocess, Sample.RIA.getAccountActivity.esbp:
  - GetAccountActivityDefaultResponse.xml
  - GetAccountActivityRequest.xml
  - GetPhoneAccountActivityRequest.xml
  - GetPhoneAccountActivityResponse.xml
  - GetTVAccountActivityRequest.xml
  - GetTVAccountActivityResponse.xml

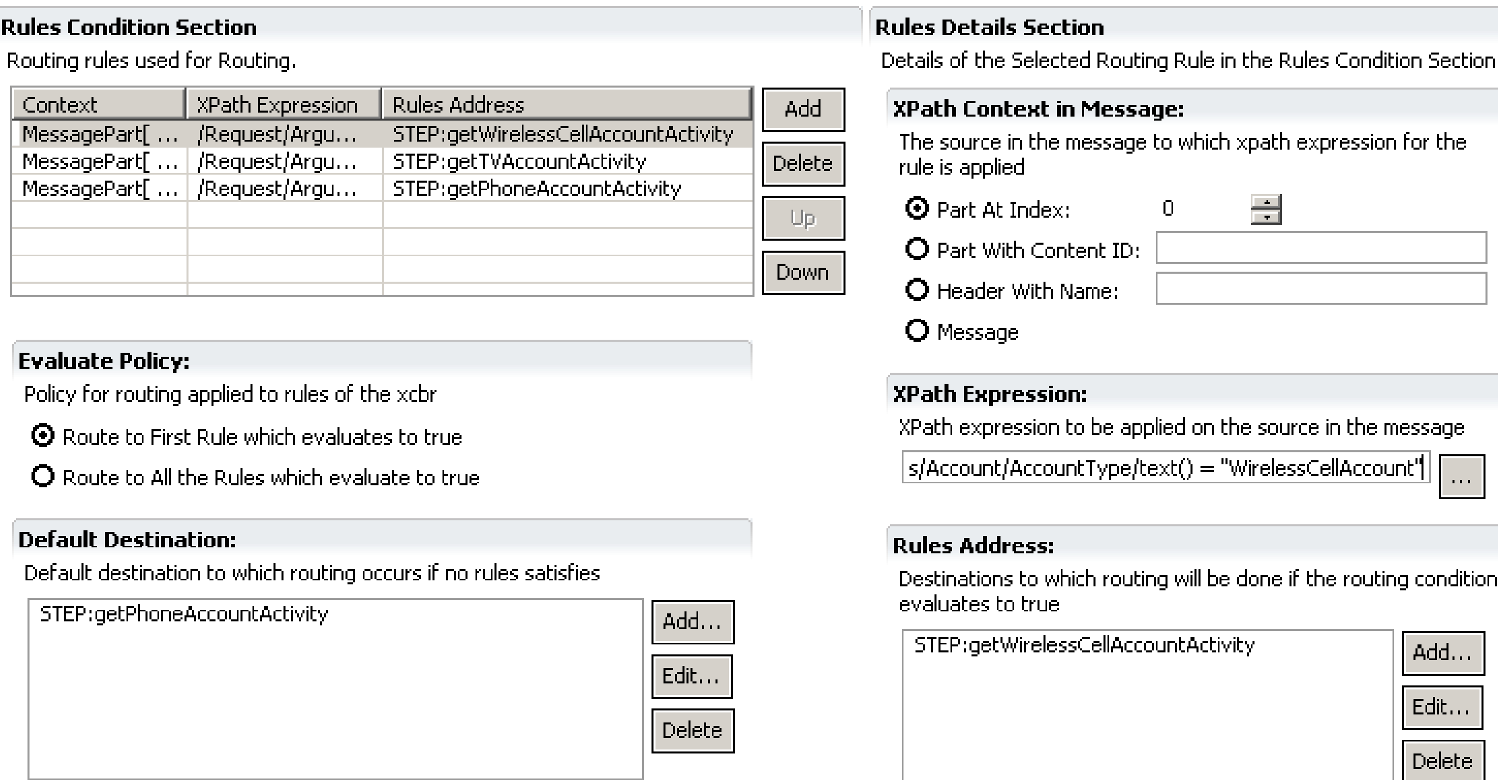
    - GetWirelessCellAccountActivityRequest.xml
    - GetWirelessCellAccountActivityResponse.xml

- **getAccountActivity** — Contains XML files used to prototype and test the ESB subprocess, Sample.RIA.GetAccounts.esbp:
    - GetAccountsDefaultResponse.xml
    - GetAccountsIntermediateResponse.xml
    - GetAccountsRequest.xml
    - PhoneAccountInfo.xml
    - TVAccountInfo.xml
    - WirelessCellAccountInfo.xml

[RIA project components](#)

routeGetActivityRequest.xcbr

The XPath routing rules file `routeGetActivityRequest.xcbr` defines rules that route requests to three branches of the content-based router used in the ESB subprocess GetAccountActivity. The rules evaluate an XPath expression that checks the account type, and routes the request through the appropriate branch for the account type.



RIA project components

routeRequest.xcbr

The XPath routing rules file `routeRequest.xcbr` defines rules for routing requests to two branches of the content-based router used in the ESB process processRequest. The rules evaluate an XPath expression that checks the request type, and routes the request to the appropriate branch for the request type.

Rules Condition Section

Routing rules used for Routing.

Context	XPath Expression	Rules Address
MessagePart[Part Nu...	/Request/RequestI...	STEP:getAccounts
MessagePart[Part Nu...	/Request/RequestI...	STEP:getAccountActivity

Add | Delete | Up | Down

Evaluate Policy:

Policy for routing applied to rules of the xcbr

- (•) Route to First Rule which evaluates to true
- ( ) Route to All the Rules which evaluate to true

Default Destination:

Default destination to which routing occurs if no rules satisfies

STEP:getAccountActivity

Add... | Edit... | Delete

Rules Details Section

Details of the Selected Routing Rule in the Rules Condition Section

XPath Context in Message:

The source in the message to which xpath expression for the rule is applied

- (•) Part At Index: 0
- ( ) Part With Content ID:
- ( ) Header With Name:
- ( ) Message

XPath Expression:

XPath expression to be applied on the source in the message

questInformation/RequestType/text() = "getAccounts" ...

Rules Address:

Destinations to which routing will be done if the routing condition evaluates to true

STEP:getAccounts

Add... | Edit... | Delete

formatGetAccountsResponse.xsl

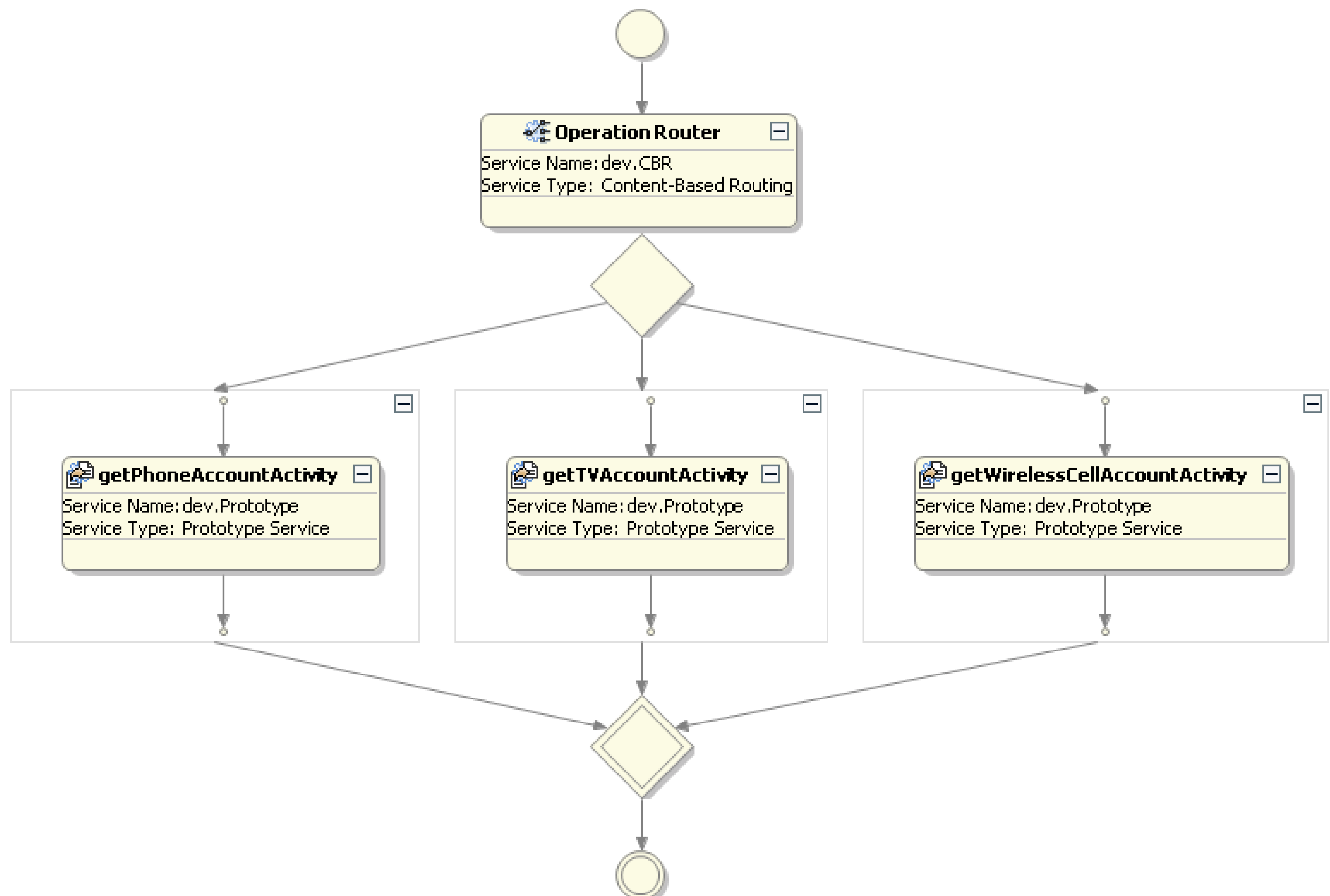
The XSL stylesheet `formatGetAccountsResponse.xsl` is part of the XML Transformation step, Format Response, in the ESB subprocess, GetAccounts. The stylesheet maps intermediate response parameters to the final output format. The tutorial shows you how to generate this stylesheet using the tools in Sonic Workbench.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="/">
      <Response>
         <RequestInformation>
            <RequestID>
               <xsl:value-of select="Request/RequestInformation/RequestID"></xsl:value-of>
            </RequestID>
            <RequestRole>
               <xsl:value-of select="Request/RequestInformation/RequestRole"></xsl:value-of>
            </RequestRole>
            <RequestName>
               <xsl:value-of select="Request/RequestInformation/RequestName"></xsl:value-of>
            </RequestName>
            <RequestType>
               <xsl:value-of select="Request/RequestInformation/RequestType"></xsl:value-of>
            </RequestType>
         </RequestInformation>
         <Arguments>
            <CustomerNumber>
               <xsl:value-of select="Request/Arguments/CustomerNumber"></xsl:value-of>
            </CustomerNumber>
         </Arguments>
         <Data>
            <Accounts>
               <xsl:for-each select="Request/Account">
                  <Account>
                     <AccountNumber>
                        <xsl:value-of select="AccountNumber"></xsl:value-of>
                     </AccountNumber>
                     <AccountType>
                        <xsl:value-of select="AccountType"></xsl:value-of>
                     </AccountType>
                     <IsOpen>
                        <xsl:value-of select="IsOpen"></xsl:value-of>
                     </IsOpen>
                  </Account>
               </xsl:for-each>
            </Accounts>
         </Data>
      </Response>
   </xsl:template>
</xsl:stylesheet>
```

[RIA project components](#)

Sample.RIA.getAccountActivity.esbp

`Sample.RIA.getAccountActivity.esbp` is an ESB process, which opens in the ESB Process editor. The getAccountActivity process contains a content-based router that routes incoming requests through one of three branches, based on XPath routing rules that determine the account type in the request. You can expand each step to show more detail:



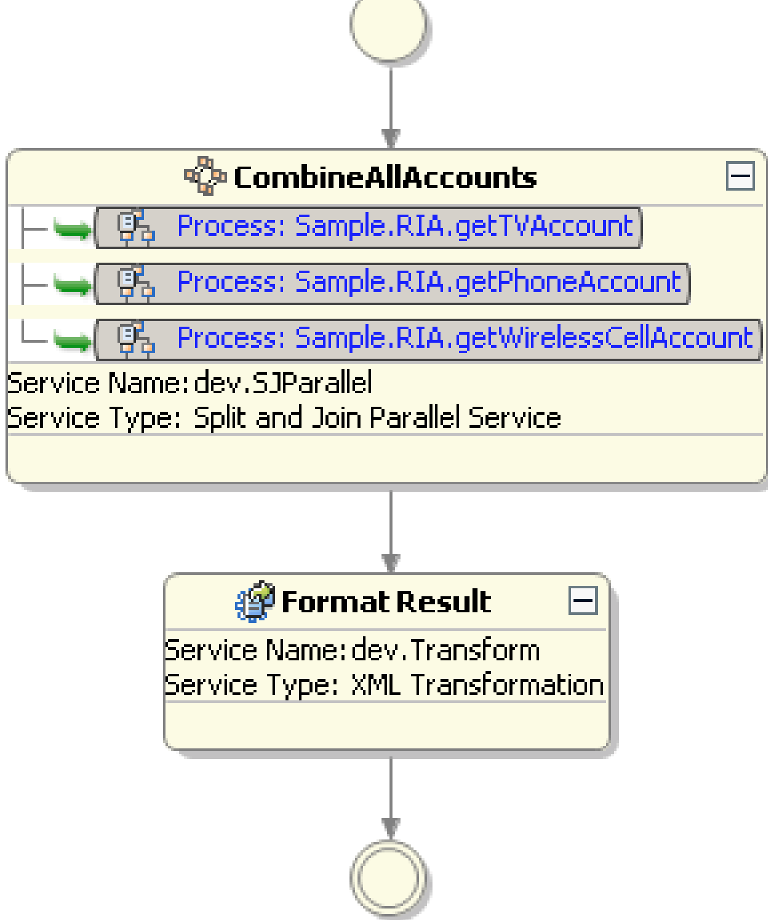
[RIA project components](#)

Sample.RIA.GetAccounts.esbp

`Sample.RIA.GetAccounts.esbp` is an ESB process, which opens in the ESB Process editor. The GetAccounts subprocess returns data from all accounts for a given customer. The data is combined in a single message using the Split and Join Parallel service, CombineAllAccounts, then transformed into a preferred response format in the XML Transformation step, Format Result.

`Sample.RIA.GetAccounts.esbp` is a subprocess of `Sample.RIA.processRequest.esbp`.

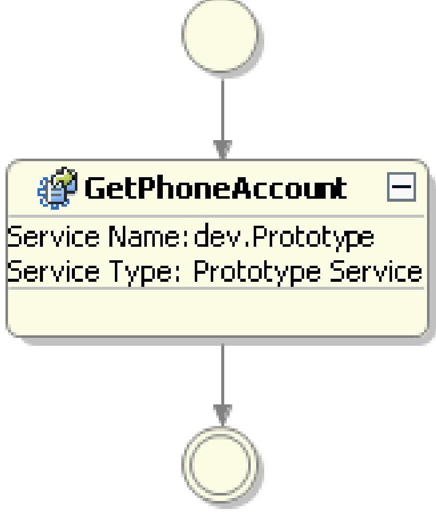
You can expand each step to show more detail:



[RIA project components](#)

Sample.RIA.getPhoneAccount.esbp

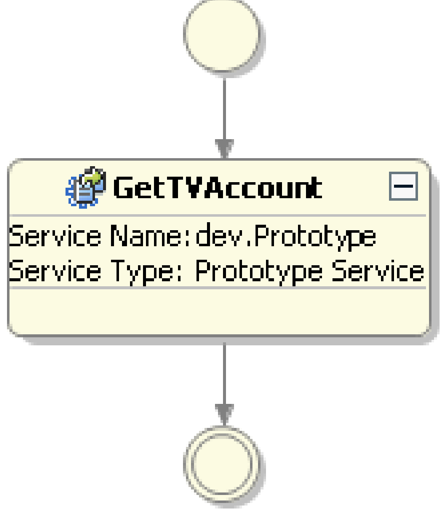
`Sample.RIA.getPhoneAccount.esbp` is an ESB process, which opens in the ESB Process editor. The getPhoneAccount subprocess provides simulated account activity data from the Phone account. In a real application, this process can be implemented to interface with a data source to retrieve actual account activity. You can expand the step to show more detail:



[RIA project components](#)

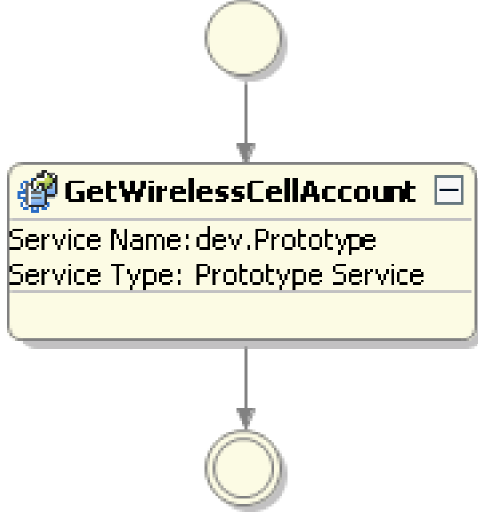
Sample.RIA.getTVAccount.esbp

`Sample.RIA.getTVAccount.esbp` is an ESB process, which opens in the ESB Process editor. The getTVAccount subprocess provides simulated account activity data from the TV account. In a real application, this process can be implemented to interface with a data source to retrieve actual account activity. You can expand the step to show more detail:



Sample.RIA.getWirelessCellAccount.esbp

`Sample.RIA.getWirelessCellAccount.esbp` is an ESB process, which opens in the ESB Process editor. The getWirelessCellAccount subprocess provides simulated account activity data from the Wireless Cell account. In a real application, this process can be implemented to interface with a data source to retrieve actual account activity. You can expand the step to show more detail:

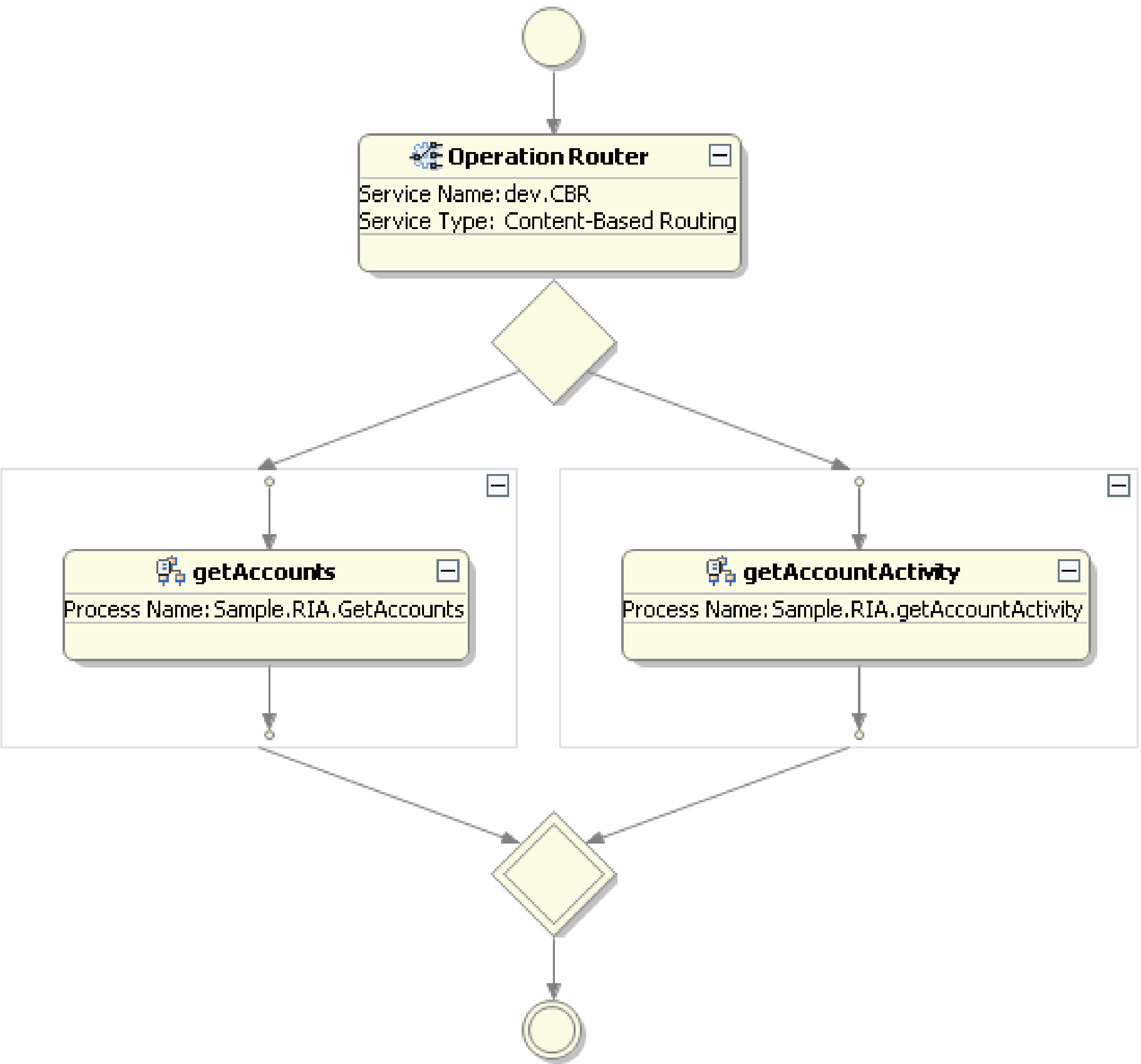


[RIA project components](#)

Sample.RIA.processRequest.esbp

`Sample.RIA.processRequest.esbp` is an ESB process, which opens in the ESB Process editor. This is the main ESB process in the tutorial. Incoming requests are routed by the content-based router based on the request type, as determined by XPath routing rules. The router has two branches, one for each of the use cases in the tutorial. Requests for accounts are routed to the getAccounts branch, which contains the GetAccounts subprocess to compile a list of accounts into a single message and format the response. Requests for account activity are routed to the getAccountActivity branch, which contains the getAccountActivity subprocess to route the request to the appropriate request type and return activity for the specified account.

You can expand each step to show more detail:



GetAccountActivityDefaultResponse.xml

`GetAccountActivityDefaultResponse.xml` is an XML file, which opens in the XML editor. This file provides a default response during the phased implementation of the tutorial project. The response contains data about activity on a customer´s account, and simulates the actual data that would be returned for the fully implemented getAccountActivity subprocess.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
      </Account>
   </Arguments>
   <Data>
      <AccountActivity>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
         <OutstandingBalance>44.99</OutstandingBalance>
      </AccountActivity>
   </Data>
</Response>
```

[RIA project components](#)

GetAccountActivityRequest.xml

`GetAccountActivityRequest.xml` is an XML file, which opens in the XML editor. This file contains a request for the activity on a TV account belonging to customer number 123456. You can use this request as input to the ESB processes in the tutorial project.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Request>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
      </Account>
   </Arguments>
</Request>
```

[RIA project components]

GetPhoneAccountActivityRequest.xml

`GetPhoneAccountActivityRequest.xml` is an XML file, which opens in the XML editor. This file contains a request for the activity on a Phone account belonging to customer number 123456. You can use this request as input to the ESB processes in the tutorial project.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>PhoneAccount</AccountType>
      </Account>
   </Arguments>
</Request>
```

GetPhoneAccountActivityResponse.xml

`GetPhoneAccountActivityResponse.xml` is an XML file, which opens in the XML editor. This file provides a simulated response containing data about activity on a customer´s Phone account, and simulates the actual data that would be returned for the fully implemented GetAccountActivity process.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>PhoneAccount</AccountType>
      </Account>
   </Arguments>
   <Data>
      <AccountActivity>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>PhoneAccount</AccountType>
         <OutstandingBalance>33.88</OutstandingBalance>
      </AccountActivity>
   </Data>
</Response>
```

GetTVAccountActivityRequest.xml

`GetTVAccountActivityRequest.xml` is an XML file, which opens in the XML editor. This file contains a request for the activity on a TV account belonging to customer number 123456. You can use this request as input to the ESB processes in the tutorial project.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccountActivity</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
    <Account>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>TVAccount</AccountType>
    </Account>
  </Arguments>
</Request>
```

GetTVAccountActivityResponse.xml

`GetTVAccountActivityResponse.xml` is an XML file, which opens in the XML editor. This file provides a simulated response containing data about activity on a customer´s TV account, and simulates the actual data that would be returned for the fully implemented GetAccountActivity process.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
      </Account>
   </Arguments>
   <Data>
      <AccountActivity>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>TVAccount</AccountType>
         <OutstandingBalance>44.99</OutstandingBalance>
      </AccountActivity>
   </Data>
</Response>
```

GetWirelessCellAccountActivityRequest.xml

`GetWirelessCellAccountActivityRequest.xml` is an XML file, which opens in the XML editor. This file contains a request for the activity on a Wireless Cell account belonging to customer number 123456. You can use this request as input to the ESB processes in the tutorial project.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccountActivity</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
    <Account>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>WirelessCellAccount</AccountType>
    </Account>
  </Arguments>
</Request>
```

GetWirelessCellAccountActivityResponse.xml

`GetWirelessCellAccountActivityResponse.xml` is an XML file, which opens in the XML editor. This file provides a simulated response containing data about activity on a customer´s wireless cell account, and simulates the actual data that would be returned for the fully implemented GetAccountActivity process.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccountActivity</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
      <Account>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>WirelessCellAccount</AccountType>
      </Account>
   </Arguments>
   <Data>
      <AccountActivity>
         <AccountNumber>9999999</AccountNumber>
         <AccountType>WirelessCellAccount</AccountType>
         <OutstandingBalance>123.56</OutstandingBalance>
      </AccountActivity>
   </Data>
</Response>
```

GetAccountsDefaultResponse.xml

`GetAccountsDefaultResponse.xml` is an XML file, which opens in the XML editor. This file provides a default response during the phased implementation of the tutorial project. The response contains a list of accounts for a customer, and simulates the actual data that would be returned for the fully implemented getAccounts subprocess.

```xml
<?xml version="1.0"?>
<Response>
  <RequestInformation>
    <RequestID>123456</RequestID>
    <RequestRole>CSR</RequestRole>
    <RequestName>user</RequestName>
    <RequestType>getAccounts</RequestType>
  </RequestInformation>
  <Arguments>
    <CustomerNumber>123456</CustomerNumber>
  </Arguments>
  <Data>
    <Accounts>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>TVAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>InternetAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>WirelessCellAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
      <Account>
        <AccountNumber>9999999</AccountNumber>
        <AccountType>PhoneAccount</AccountType>
        <IsOpen>true</IsOpen>
      </Account>
    </Accounts>
  </Data>
</Response>
```

GetAccountsIntermediateResponse.xml

`GetAccountsIntermediateResponse.xml` is an XML file, which opens in the XML editor. This file contains the response obtained when running the getAccounts subprocess before an XML Transformation step is added to the process. You can use this response when creating a stylesheet to transform the response format.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccounts</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
   </Arguments>
   <Account>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>TVAccount</AccountType>
      <IsOpen>true</IsOpen>
   </Account>
   <Account>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>PhoneAccount</AccountType>
      <IsOpen>true</IsOpen>
   </Account>
   <Account>
      <AccountNumber>9999999</AccountNumber>
      <AccountType>WirelessCellAccount</AccountType>
      <IsOpen>true</IsOpen>
   </Account>
</Request>
```

GetAccountsRequest.xml

`GetAccountsRequest.xml` is an XML file, which opens in the XML editor. This file contains a request for a list of accounts belonging to customer number 123456. You can use this request as input to the ESB processes in the tutorial project.

```
<?xml version="1.0"?>
<Request>
   <RequestInformation>
      <RequestID>123456</RequestID>
      <RequestRole>CSR</RequestRole>
      <RequestName>user</RequestName>
      <RequestType>getAccounts</RequestType>
   </RequestInformation>
   <Arguments>
      <CustomerNumber>123456</CustomerNumber>
   </Arguments>
</Request>
```

PhoneAccountInfo.xml

`PhoneAccountInfo.xml` is an XML file, which opens in the XML editor. This file provides information about a Phone account, one of the account types used in the tutorial, and is used during the tutorial to simulate data that would be returned from an actual data source.

```
<?xml version="1.0" encoding="UTF-8"?>
<Account>
   <AccountNumber>9999999</AccountNumber>
   <AccountType>PhoneAccount</AccountType>
   <IsOpen>true</IsOpen>
</Account>
```

TVAccountInfo.xml

`TVAccountInfo.xml` is an XML file, which opens in the XML editor. This file provides information about a TV account, one of the account types used in the tutorial, and is used during the tutorial to simulate data that would be returned from an actual data source.

```
<?xml version="1.0" encoding="UTF-8"?>
<Account>
  <AccountNumber>9999999</AccountNumber>
  <AccountType>TVAccount</AccountType>
  <IsOpen>true</IsOpen>
</Account>
```

WirelessCellAccountInfo.xml

`WirelessCellAccountInfo.xml` is an XML file, which opens in the XML editor. This file provides information about a Wireless Cell account, one of the account types used in the tutorial, and is used during the tutorial to simulate data that would be returned from an actual data source.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Account>
  <AccountNumber>9999999</AccountNumber>
  <AccountType>WirelessCellAccount</AccountType>
  <IsOpen>true</IsOpen>
</Account>
```