ReadSoft Service Bus 2-4

SDK

October 2011 – ReadSoft AB

Questions or comments about this document may be emailed to **documentation@readsoft.com**.

06102011.1013

ReadSoft AB (Head office) | Södra Kyrkogatan 4 | SE-252 23 Helsingborg | Sweden | Phone: +46 42 490 21 00 | Fax: +46 42 490 21 20
ReadSoft Ebydos AG | Walther-von-Cronberg-Platz 2 | 60594 Frankfurt | Germany | Phone: +49 69 90 43 260 | Fax: +49 69 9043 2613
info@readsoft.com | www.readsoft.com

Contents

About the SDK

This help file describes the ReadSoft Service Bus (RSB) Software Development Kit (SDK). The SDK contains:

- Software required develop adapters that interface with RSB to publish and subscribe documents to and from the bus using a small library of helper classes.
- Assemblies required to interface with the bus.
- Example projects.
- API Help.

You can install the SDK as a component which is found on the main RSB installation.

The documentation uses C# as the programming language, but it is possible to use any CLS-compliant .NET language, such as VB.NET.

System requirements

The following is required when building the software that will communicate with RSB.

- Microsoft .Net Framework 4
- Visual Studio 2010 SP1 or later

To be able to test your adapter you also need:

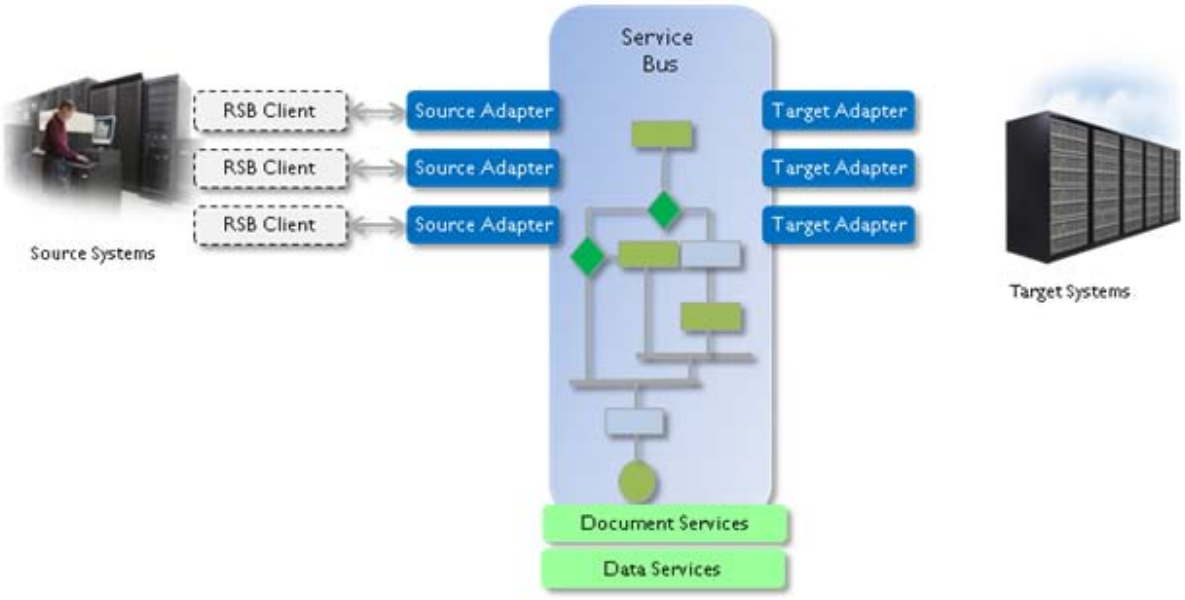
- Microsoft SQL Server (Microsoft SQL Server Express is supported)

Concepts

System overview

RSB utilizes the following concepts:

- **Source system**—a system that sends documents, such as DOCUMENTS, INVOICES etc.
- **Source system**—a system that sends documents, such as DOCUMENTS, INVOICES etc.
- **Target system**—a systems that receives documents, such as an ERP system, INVOICE COCKPIT, INVOICEIT etc.
- **Adapter**—provides additional features/capabilities to the bus and the systems that use it. To create an interface between a source and target system, for example, you need to use at least two adapters: one adapter (source adapter) provides an interface between RSB and the source system, and one (target adapter) that connects RSB to the target system. Adapters typically use the Microsoft Windows Communication Framework (WCF) to communicate with other system components.
- **RSB clients**—provide connectivity between source systems and RSB. Typically, these are lightweight software components that plug into the source system. One example is the RSB INVOICES plug-in.
- **ReadSoft Service Bus**—provides a framework that unites the parts to form a complete system.
- **Document Service**—defines activities between a source system and one or more target systems for transferring documents.
- **Data Service**—defines activities between a provider system (source system) and one or more consumer systems (target systems) to enable the use of data entities.



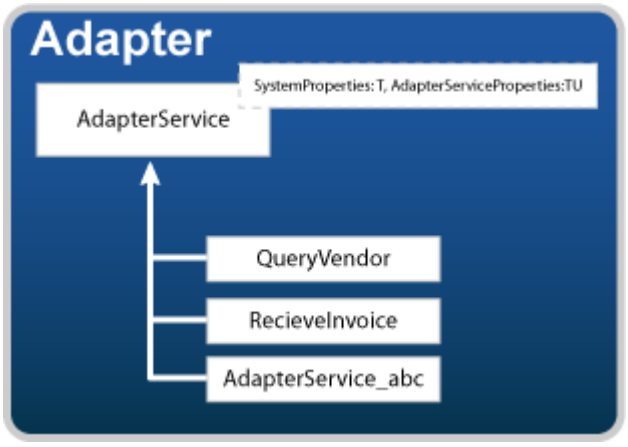
Design goals

The design goal of the SDK is to make it as simple as possible to create adapters for the ReadSoft Service Bus in order to extend the capabilities of ReadSoft Solutions. RSB is designed for component reuse, and should provide basic functionality for the system integration while maintaining scalability.

Extending the bus capabilities

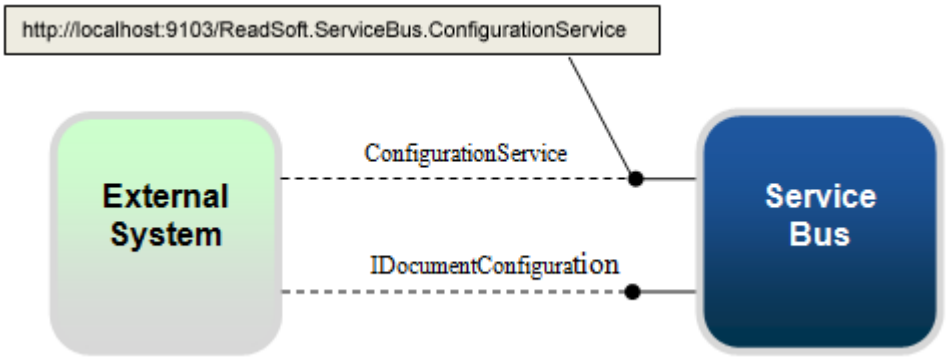
Adapters are used to extend the capabilities the bus can provide. From a deployment perspective, an adapter is a package of functionality that can be installed and registered in the RSB runtime. Through adapters, the user can configure document and data services by creating activities to accomplish specific behavior.

The functionality implemented in an adapter is found as one or more adapter services. Each adapter service defines two sets of properties: one that is related to the system it can connect to (adapter properties) and one that is specific for the adapter service (adapter service properties). Adapter properties are common to all adapter services, whereas adapter service properties are specific for all activities that use the service.



Connecting to the service bus

To be able to make use of the services provided by the bus, you need to have a component (client) that connects the external system to the service bus. The client uses the address of the RSB Configuration Service to make the connection. The configuration service provides a list of all the document and data services that are available in the bus and have been configured. You can select which type of service you want to use from the list. This is done via a document service (service configuration), which contains an ID that you use when you call the service. This can be done programmatically if you know what service to use and it will not change or require user input, which is the case for DOCUMENTS and INVOICES.



Configuration

First, you need to know the computer name where RSB is installed and the web service port. It is specified in the RSB Administration **Settings** view. The default port is 9103.

With SDK helper functions you can create URLs for calling RSB.

With the repository service you can get the available document services from RSB.

The code below demonstrates how to retrieve all the active-service configurations.

```
try

{

   string repositoryUri =   ExternalClientHelper.GetRepositoryUri("RSB_computer_name",
"9103");

   var proxy = ProxyHelper.CreateProxy<IRepository>(repositoryUri);

   ServiceConfigurationList serviceConfigurationList =
proxy.GetActiveRoutesByAdapterID(adapterID);

}

finally

{

   var channel = proxy as ICommunicationObject;

   channel.CloseConnection();

}
```

When you have this information, you are prepared to call the service.

Transfer document

To transfer a document via RSB using .NET, follow these guidelines:

- Create the document you want to send. Normally this is a XML document, but whatever the case, the source adapter should be notified.
- To call the document service on the RSB you need to know:
  - The address to RSB (RSB host name and web port). This is used to generate a proxy.
  - The name of the service you want to call.
- Via the address and the service name you can create the proxy for the service.
- In the call to the service, you pass the document and the ID of the document service.

```
public static Result SendDocumentSynchronous(GenericDocument document, Guid

adapterServiceId,
```

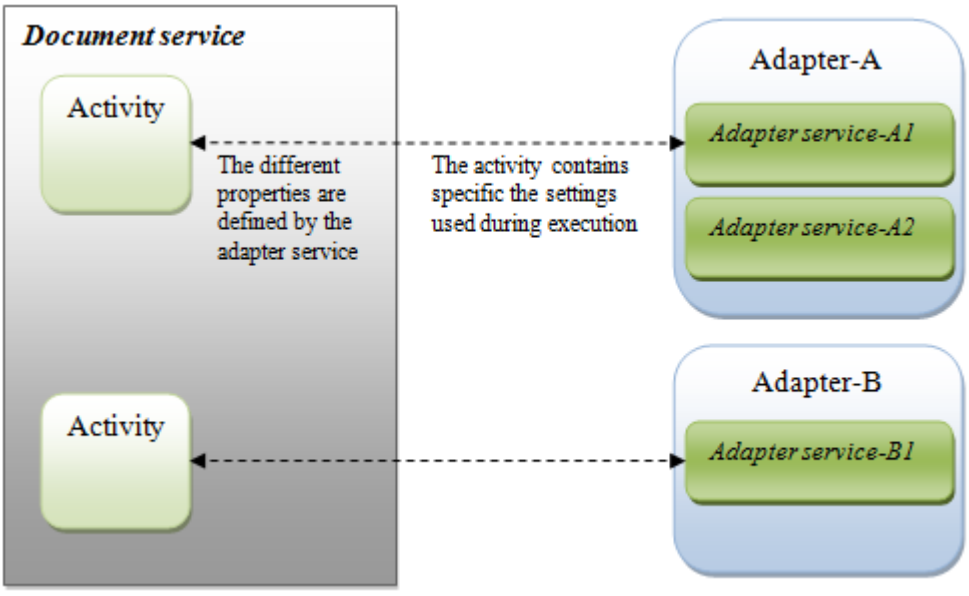
```
Guid serviceConfigurationID)

{
string configurationUri = string repositoryUri =
ExternalClientHelper.GetRepositoryUri("RSB_computer_name", "9103");
   IGenericDocumentSource proxy = GetSourceAdapterProxy(configurationUri, adapterServiceId);

   try

   {
      return proxy.ProcessDocumentSynchronous(document, serviceConfigurationID);
   }
   finally
   {
      var channel = proxy as ICommunicationObject;
      channel.CloseConnection();
   }
}
```

RSB services

An adapter can be seen as a container of specific functionality. To make use of this functionality, you set up different RSB services (for example, a document-routing service). The service settings are built on different activities which the user adds to accomplish a task. Adding new activities is done via the Admin page. During operation, the service executes, based on the defined activities, and calls the adapter service.

For example, if you have an adapter that knows how to connect to an SAP system, and you want to create two document services that support document transfer (one for the production system and one for the test system). The same adapter is used to communicate with both systems, but the activity settings are unique for each document-routing service.

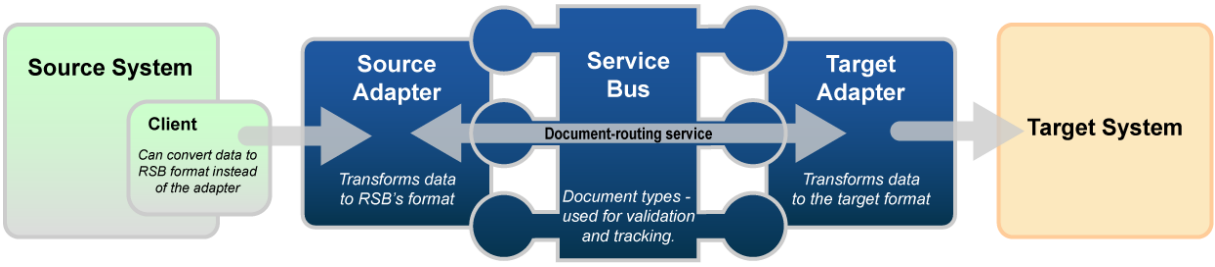
RSB utilizes two main types of services: document services and data services. Together with installed adapters, you can configure specific services, based on these types, to provide a solution for a specific customer problem.



Document services

Document services are services that are normally related to a business process and are used for transferring documents from one system to another. A document-routing service is a standard feature included in RSB. This service is installed automatically and appears as an adapter in the admin application

The service bus can use source adapters to handle specific requirements that are related to the type of system that calls the bus. It can, for example, transform documents into a uniform format or handle specific interfaces or protocols. This could be done within the connecting system, but source adapters provide the ability to solve specific requirements in the context of RSB, and that is one of the reasons that source adapters exist.



The bus contains a document-routing service.

Target system adapters are called by the service that has been configured. This means that the target must implement a specific interface.

Calls to the document-routing service pass the document object, GenericDocument. Apart from the document content, which is in the form of an XML stream, the GenericDocument contains:

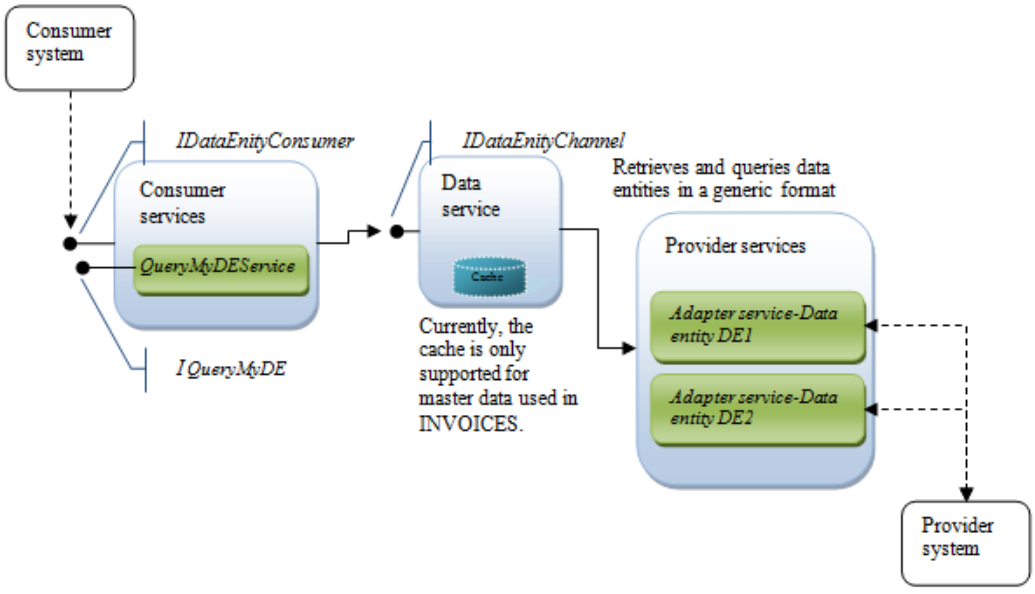
- an ID of the document
- information about the sending system
- basic document properties
- a list of attachments to the document.

For more detailed info about the Document object, see RSB API documentation.

Data services

Unlike document services, which handle documents, data services handle pure data entities, such as master data entities. Data services cannot connect directly to a business process. Instead, the source system (provider), that has the original data entities, is called by a system that wants to use it.

In RSB, the master-data service is included as a standard feature. Naturally, this can be extended with additional adapters.



The consumer can use either a generic query interface provided in the framework or expose its own domain-specific interface. The consumer service translates these services to generic calls exposed by IDataEntityChannel which is implemented by the data service which is responsible for the calls based on the service configuration. Different providers are called to translate the call to fit the provider system.

Some of the data entities can be cached in the bus to prevent performance issues when the provider system has a slow response time. When you create your own adapter, you can implement caching capabilities for your own data entities.

Adapters

There are different types of adapters, but the most common are source and target adapters. Adapters are used to create connectors that integrate two or more systems. When you create a new adapter, you normally start by inheriting a base class that provides base functionality.

To integrate two systems you need at least two adapters: one that provides an interface to the sending system (source) and one connects to the receiving system (target).

Source adapters

- Provide domain-specific services that the external system can use.
- Receive/retrieve documents from the external system.
- Transform documents to RSB's internal format. This can also be done on the client side if necessary.
- Call the generic document service provided by the bus.

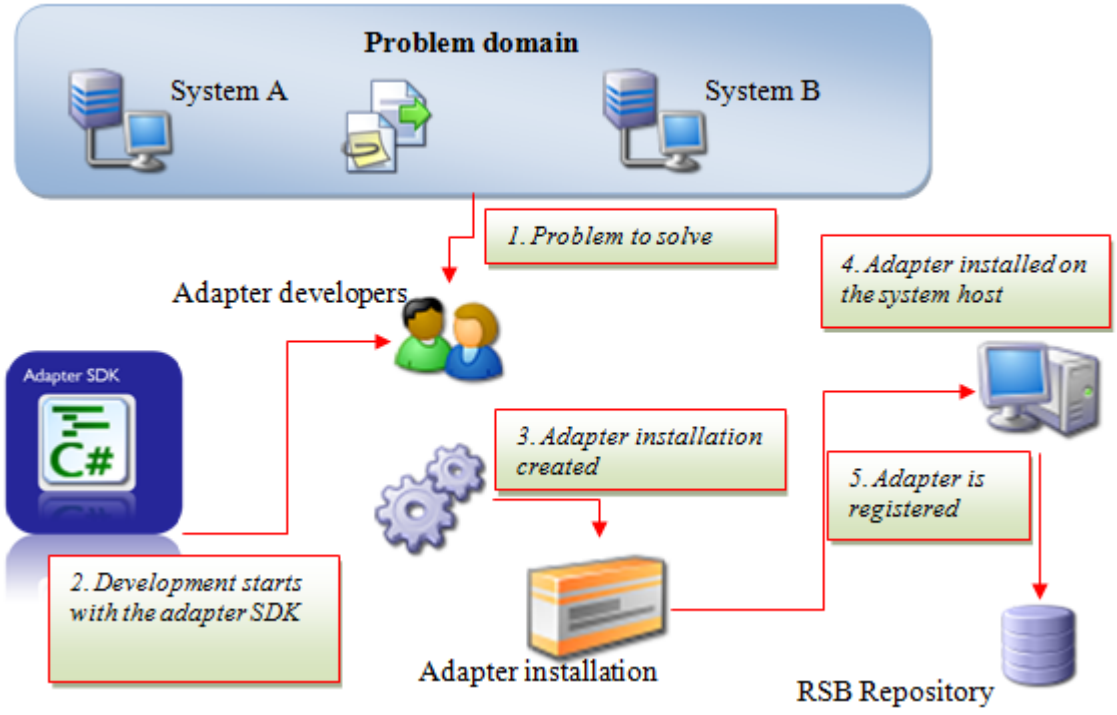
Target adapters

- Transform documents from RSB's internal format to the target-system format.
- Pass documents to the target.
- Return status messages to the document-routing service.

Getting started

Developing adapters

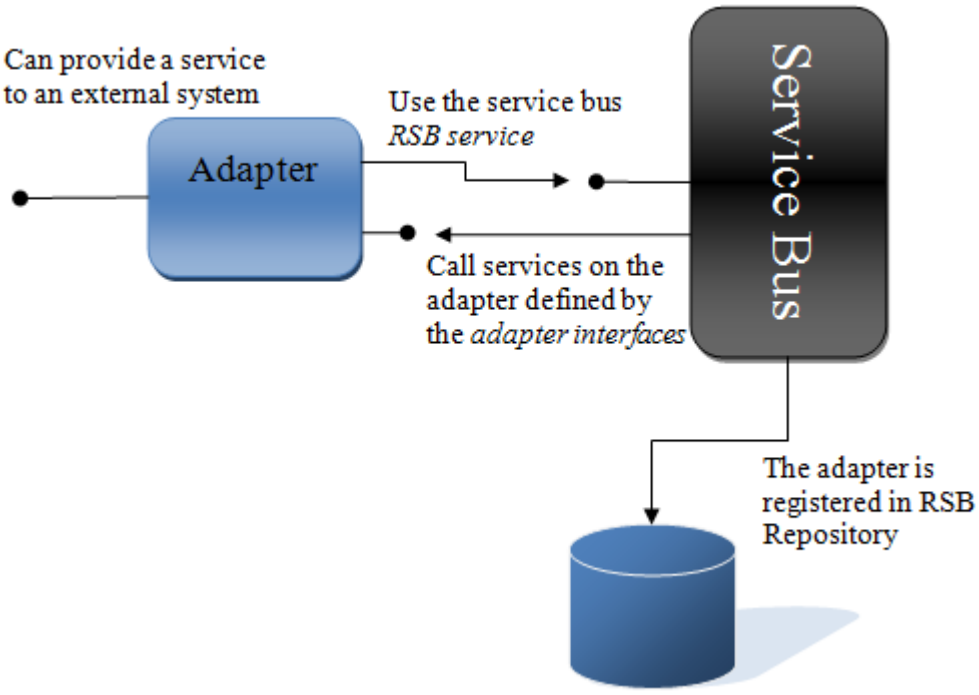
Adapters are an important concept within the ReadSoft Service Bus. Adapters extend the capabilities of RSB. An adapter can be developed by anyone; it is intended for developers in the solution labs, subsidiaries, and partners.



The SDK includes tools to create installation packages for adapters. The adapter installation uses the NSIS installation framework to get a quick and flexible installation process. To be able to use an adapter, the required files must be installed in the host system and the adapter must be registered in the service bus configuration. The installation package generated by the tools in the SDK takes care of both these things.

Adapter communication

An adapter communicates with the bus through services. These services are one part of the RSB API. The other part is classes from which the adapter classes are inherited.



The RSB services that the service bus exposes are something that the adapter developer uses to accomplish the desired tasks. An example of an RSB service is *ProcessDocument*, but there are also services for logging, event handling etc.

RSB also defines a set of interfaces that an adapter can implement. Some of these are mandatory and must exist to be able to work together with the bus while other interfaces are optional that the developer can choose to implement to achieve certain functionality. Each adapter that implements an adapter interface will be called by the bus when certain events occur.

Hosting adapters

RSB supports two ways to host an adapter: either you let the service bus host it, or you implement a Windows service--or any other kind of host--you want to host it for you. It is preferred to let the RSB host the adapter, but there are situations where you want to host it using a Windows service. For example, the ReadSoft solution for SAP uses that adapters are hosted by a separate Windows service in order to run them in a 32-bit environment, since this is a limitation in the toolkit used to access the SAP system).

When an adapter is hosted by the RSB windows service host, it is hosted in a separate application domain (AppDomain), which is isolated from the rest of the service bus. AppDomain has a global mechanism for logging, which is automatically connected to the RSB logging service.

Setting up the environment

Setting up the database

RSB uses a database to store configurations and log data. Microsoft SQL Server or Microsoft SQL Server Express are supported. If you have already installed RSB, the database is already configured.

Setting up IIS

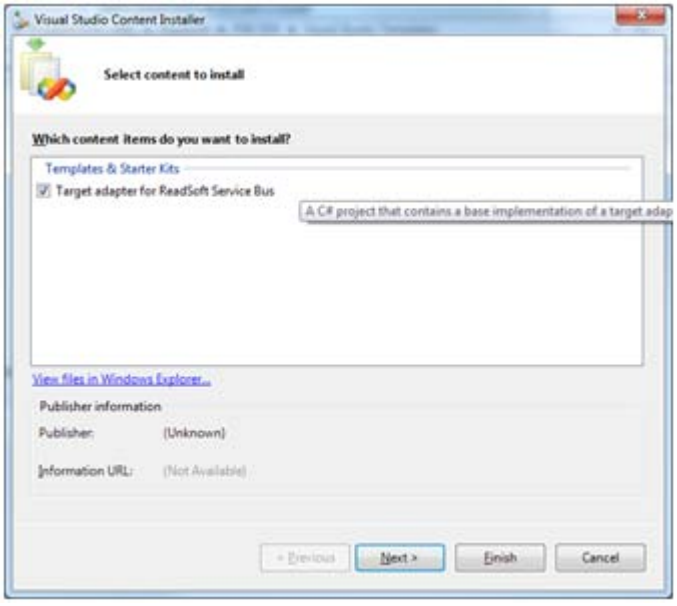
The RSB Administration page can be hosted by IIS. If you intend to develop for an environment in which the RSB Administration page will be hosted by IIS, you must have IIS installed and configured (See the installation guide for more information.). The easiest way to accomplish this is to install RSB with the IIS option, referring to the troubleshooting section of the installation guide, if you have any problems.

Visual Studio project template

The SDK includes a project template for Visual Studio that makes it easier to start developing adapters. To start a project:

1. Navigate to C:\Program Files\ReadSoft\RSB SDK\SDK\Visual Studio Templates.

2. Open RSB Adapter template.vsi and follow the wizard that appears.

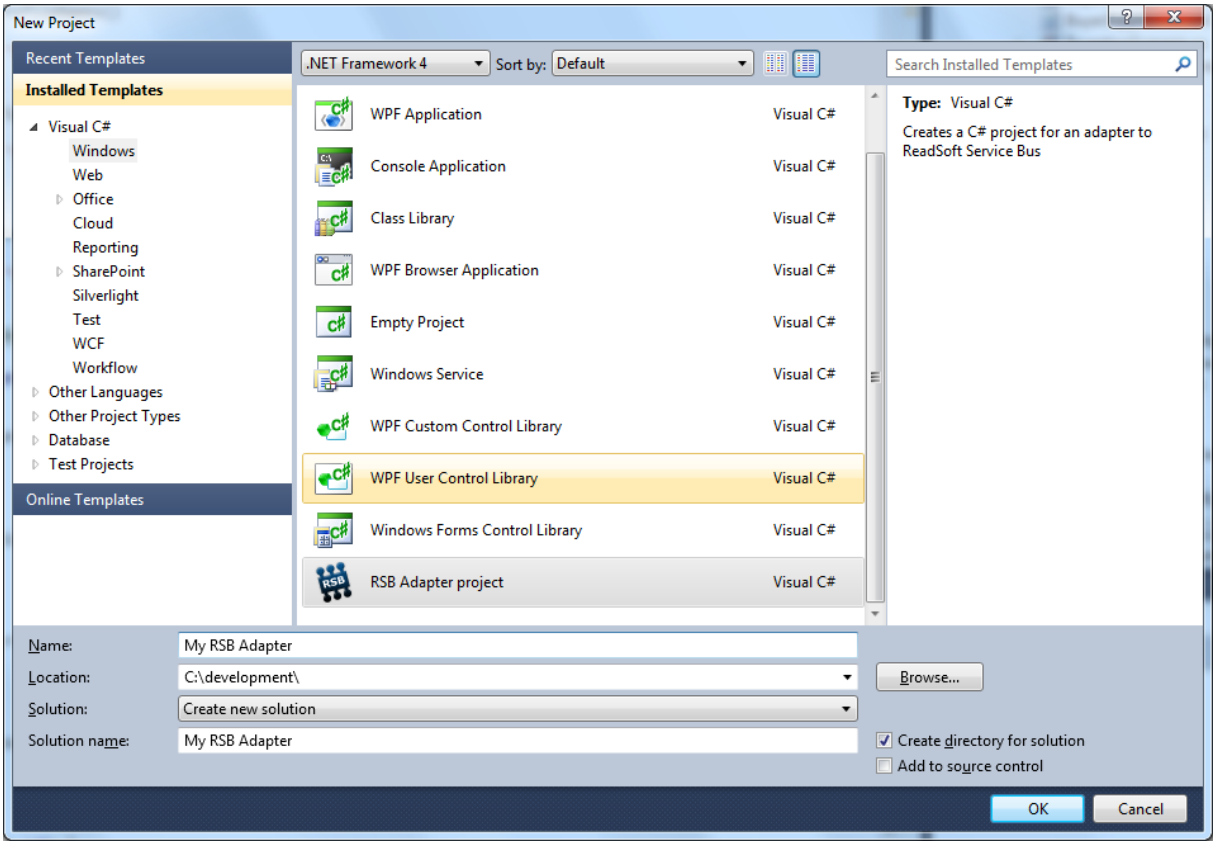


If you have Visual Studio open during the template installation, you must restart it before the template appears in the Templates list. You find the template in Visual C# >> Windows.

Creating an adapter project

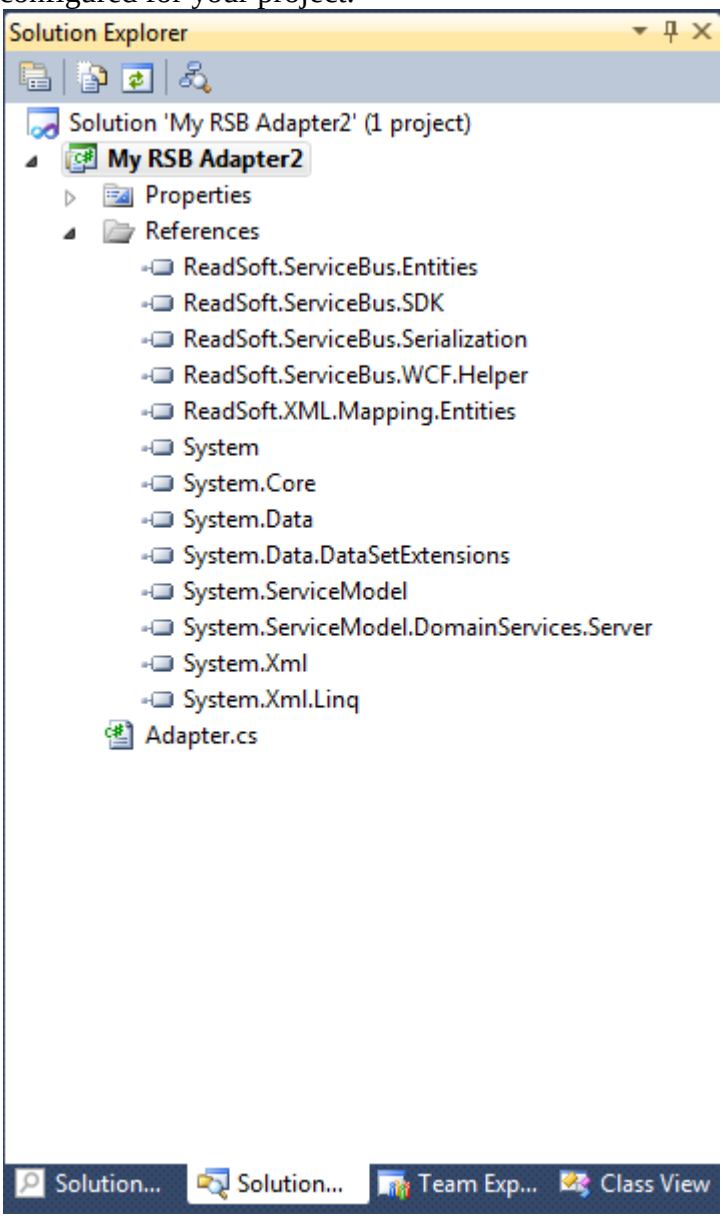
This topic demonstrates how to create an adapter project and create, install, and register an adapter. More details on the different types of adapter are covered later in the document. Regardless of the adapter type, they are basically handled the same way in the administration interface.

1. Start by creating a new project based on the RSB Adapter template.



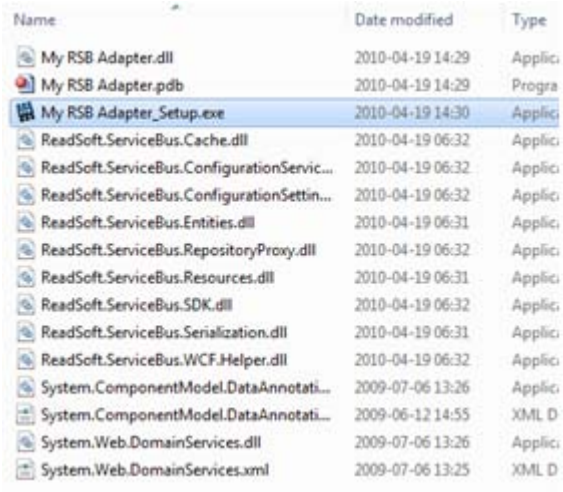
2. This template creates a C# project which includes the basic structure required to develop an adapter for RSB.

3. `Adapter.cs` contains the AdapterProperties class for your adapter. This is used when setting up the system that the adapter will connect to. All basic references are also configured for your project.



4. Build the solution and make sure that all files are compiled correctly.

5. The project template has a post-build step which creates the adapter installation for the adapter. This is done automatically after you build the solution. See Manually registering an adapter for more information.



Installing an adapter

Before activities in an adapter can be used, it must be installed and registered in the RSB repository. After that the user can use it to configure a document or data service. If a source activity in an adapter is map able, it has a RSB schema. The schema is used for:

- Defining the content (fields) of the document which is used for tracking information and routing purposes.
- Defining a map from source fields to target fields.

Translation of incoming documents to a generic format that is used within the bus

You install the adapter using the installation package (described in the previous section). Go the folder where you developed the adapter, navigate to: `\bin\Debug\RSB My Adapter_Setup.exe`, and start the installation.

After the adapter is installed, it is automatically registered within RSB, and you can find it on the RSB Administration page.

- **SDK Version -** The version of the SDK that was used to developed the adapter. This is set automatically and is used to check the compatibility when upgrading.
- **Adapter version -** The version of the adapter which you set yourself.
- **Adapter services -** Lists all the services that are hosted within the selected adapter. These services are used when setting up the document and data services. If you have not created an adapter service yet, this list is empty.

Note that when you change the adapter later, you perform the same procedure outlined above; however, if you have configured a service that is using the adapter, it must be aware that the service is deactivated before the new version is installed.

Manually registering an adapter

A post-build step is included in the project template for the RSB adapter. This step automatically calls a utility that creates an installation package for the adapter. This section describes the process, in case you want to create the installation manually.

AdapterInstallationCreator is found in the bin folder of the SDK and is used to create an installation package that installs and registers the adapter in RSB. You can run this program and at the command prompt, using parameters to control the output.

```
AdapterInstallationCreator.exe [AdapterAssemblyFilePath]
[ResultingSetupExeFilePath] [AdditionalFilePath1] [AdditionalFilePath2]
[AdditionalFilePath..N]
```

AdapterAssemblyFilePath = Full file path to the adapter assembly dll

ResultingSetupExeFilePath = Full file path to the setup exe file that will be created

AdditionalFilePath(s) = Full file path(s) of other files (besides the adapter assembly) that should be included in the installation. May contain wildcard character.

Example1

```
AdapterInstallationCreator.exe "c:\myadapter.dll"
"c:\myadapterSetup.exe"
```

Example2

```
AdapterInstallationCreator.exe "c:\myadapter.dll"
"c:\myadapterSetup.exe" "c:\myfile1.xml" "c:\*.dll"
```

You must run the AdapterInstallationCreator from the SDK bin folder where it has access to the necessary DLLs. You must give the full path to the files you want to use.

If you create your own installation program, you can register your adapter with the following tool:

```
%RsbSdkInstDir%\Bin\ReadSoft.ServiceBus.SDK.AdapterRegTool.exe
```

Usage:

- ReadSoft.ServiceBus.SDK.AdapterRegTool.exe [/silent] /register [adapter file path] [log file path]
- ReadSoft.ServiceBus.SDK.AdapterRegTool.exe [/silent] /unload [adapter GUID] [log file path]

- ReadSoft.ServiceBus.SDK.AdapterRegTool.exe [/silent] /unregister [adapter GUID] [log file path]
- ReadSoft.ServiceBus.SDK.AdapterRegTool.exe [/silent] /check [adapter GUID] [log file path]

Adapter registration tool

When your adapter is ready for serious testing, we recommend installing the adapter before debugging for the best results. However, if you rebuild the adapter frequently and installing the adapter over and over is too time consuming, you can simply register the adapter using the Adapter registration tool (also described above in Manually registering an adapter).

Add the tool to Visual Studio using the following settings:

Command

%RSBSDKInstDir%\Bin\ReadSoft.ServiceBus.SDK.AdapterRegTool.exe

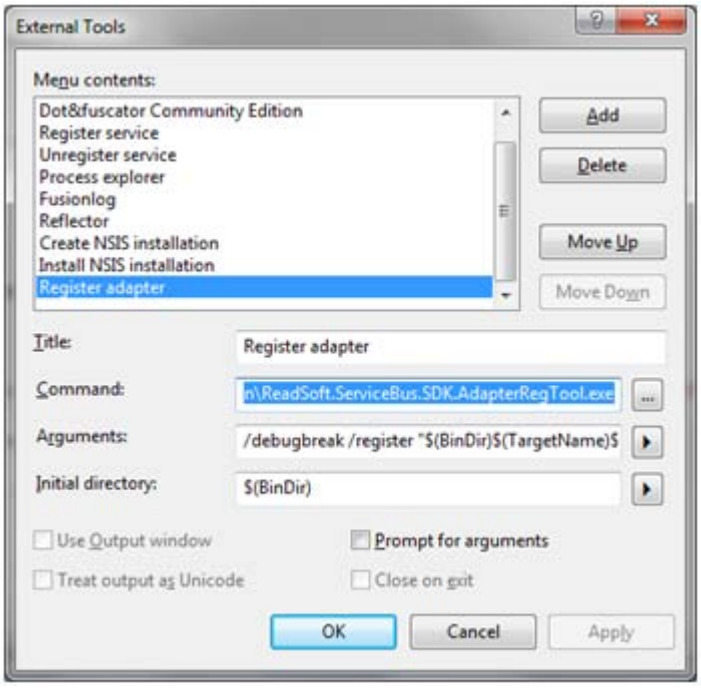
Arguments

/register "$(BinDir)$(TargetName)$(TargetExt)"

Initial directory

$(BinDir)

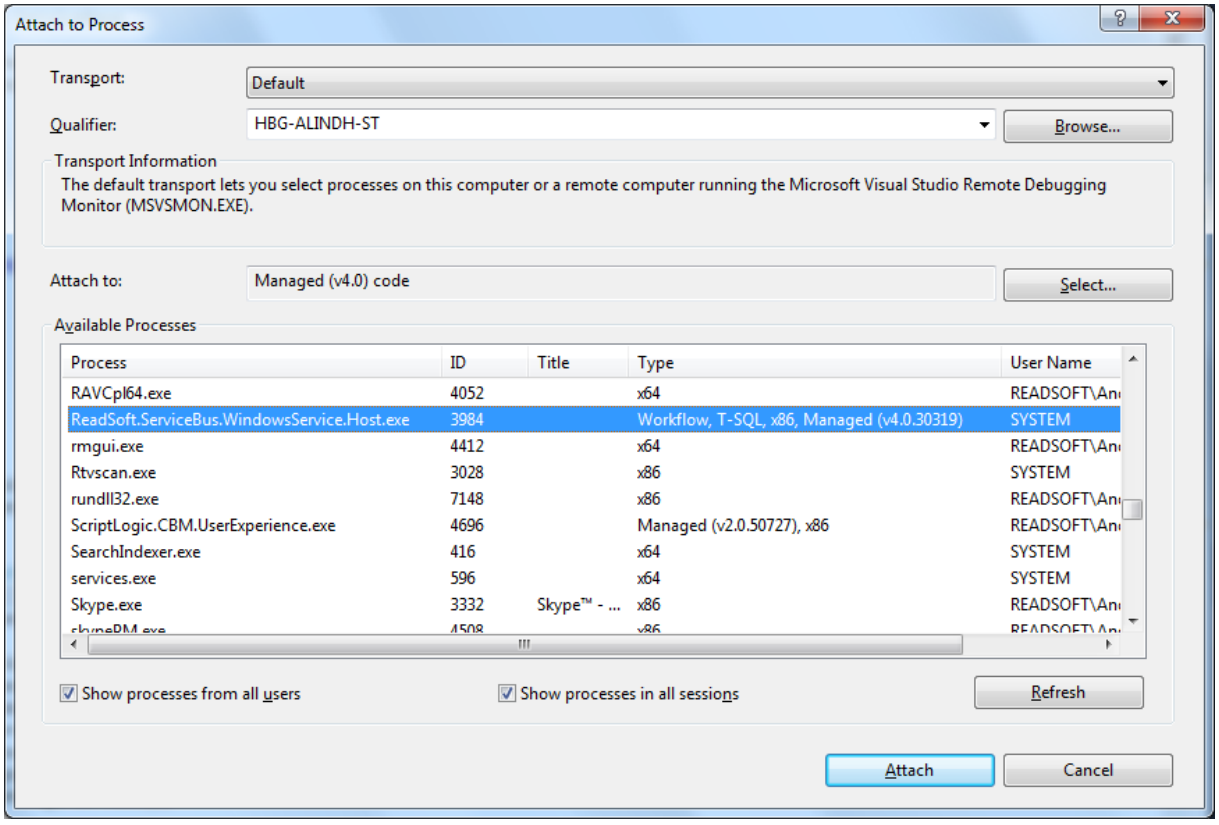
Then to register your adapter, you can simply select your assembly and choose **Register adapter** from the **Tools** menu in Visual Studio.



Debugging an adapter

When you debug an adapter, you generally go through this process:

1. Write the code.
2. Build the solution.
3. Install adapter. You must install the adapter (recommended) or register the adapter using the tool described below.
4. Debug the adapter. Since the adapter is hosted externally and is running in a framework that calls your code, you must attach your debugger (**Tools** > **Attach to process**) to the RSB Windows service.



In order to see the process, ReadSoft.ServiceBus.WindowsService.Host.Exe, you must select **Show processes from all users** and **Show processes in all sessions**. You must also specify **Managed code** in the **Attach to** setting. In Visual Studio 2010, specify **Managed (v4.0) code** in the **Attach to** setting.

Anytime you change your adapter code and rebuild, you must repeat step 3.

Simulators

When developing an adapter, it is good idea to develop a simulator that emulates the system the adapter will connect to. A simulator is a helpful tool for developing and testing the adapter. The RSB installation has a couple simulators which you can use to see what sorts of functions a simulator performs. The SDK also includes an example of a data-services simulator.

SDK examples

The SDK contains examples that describe important design concepts. These examples are used in the coming sections of this document. Each example project automatically creates an installation program for the adapter in a post-build step.

Document-services example

- **DocumentSourceAdapter**—an example of an adapter with a source-adapter service that receives calls from an external client, such as a DOCUMENTS plugin, and sends the data to the bus.
- **FileTargetAdapter**—an example of an adapter with a target-adapter service that writes an incoming document to a file.
- **FileTargetAdapterPropertiesEx**— an example of an adapter with a target-adapter service that writes an incoming document to file. The adapter service contains examples of service properties.
- **FileTargetAdapterWithLog**— an example of an adapter with an target adapter service that writes an incoming document to file. The adapter service contains examples of service properties and shows how to log errors for end-user and debugging purposes.
- **AsynchronousFileTargetAdapter** – an adapter that only works with asynchronous document transfer. The adapter service implements a special function for retrieving the current state of a document.

Data services example

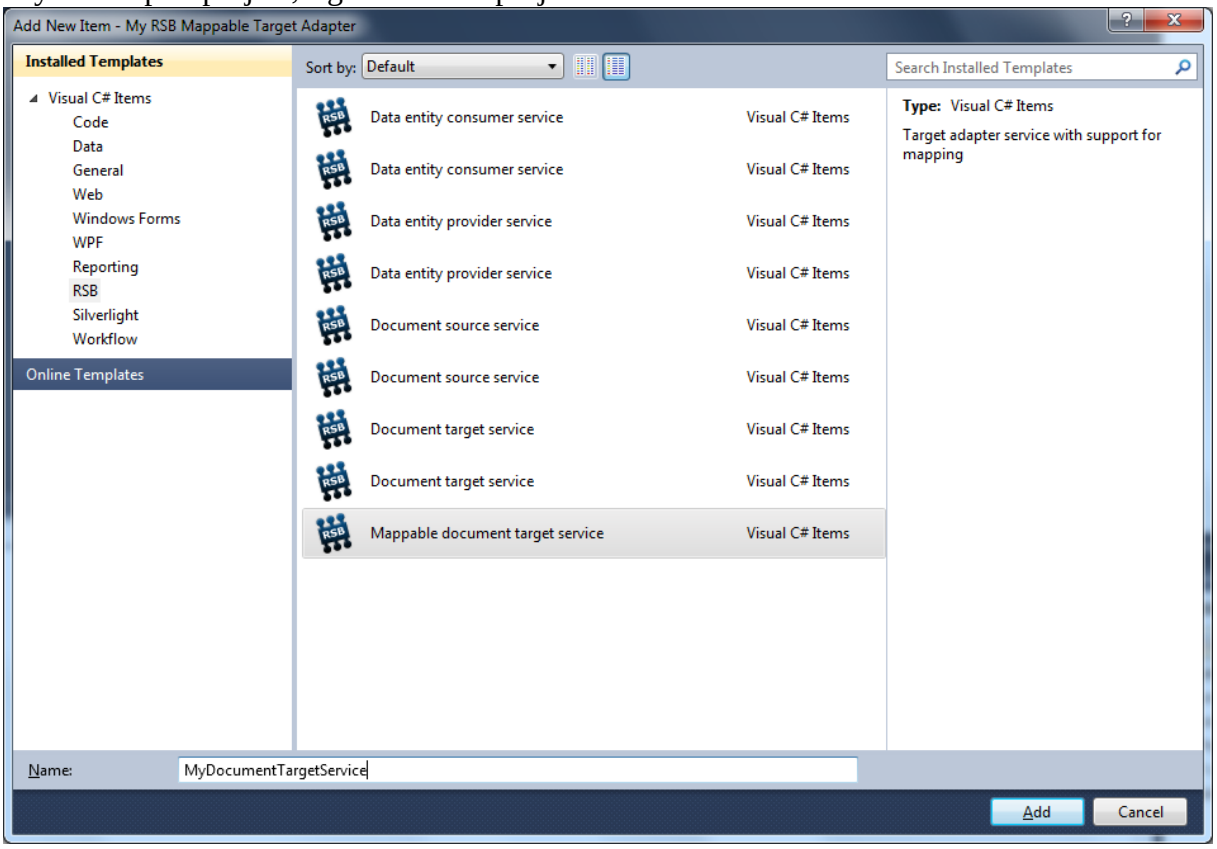
- **ChartOfAccountsConsumer**—an example of a data-entity consumer that uses the ChartOfAccount data entity.
- **ChartOfAccountsProvider**—an example of a data entity provider that uses the ChartOfAccount data entity.
- **DataEntityClientSimulator**—an example of a client application that sends query to ChartOfAccountsConsumer.
- **Entities**— an example that illustrates data entities. To avoid duplicate code when creating a data-entity consumer and data-entity provider for the same data-entity type, we recommend moving the entity to a separate assembly that is shared between consumer and provider.
- **TestChartOfAccountsProvider**—This example contains two tests: a unit test that tests the ChartOfAccountsProvider, and an integration test that calls RSB and the installed ChartOfAccountsConsumer and ChartOfAccountsProvider.

Working with services

Adding an adapter service

An adapter's functionality is implemented as an adapter service. An adapter service provides functionality via the bus or an external system. The SDK aids this task with a number of item templates. The following procedure assumes that you have already created an RSB Adapter project from the included template.

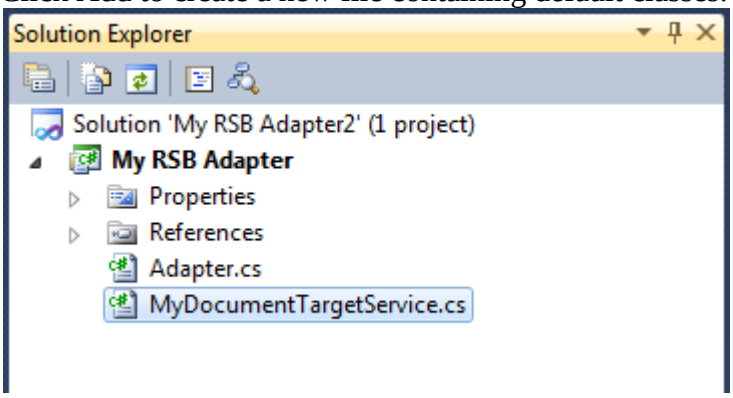
1. In your adapter project, right-click the project and select **Add** > **New item**.



2. Select the RSB in the Categories list to display the templates you can choose. For this first exercise, we will use the Document target service template.

3. Give the adapter service an appropriate name. The name is used to create the default names for the auto-generated classes.

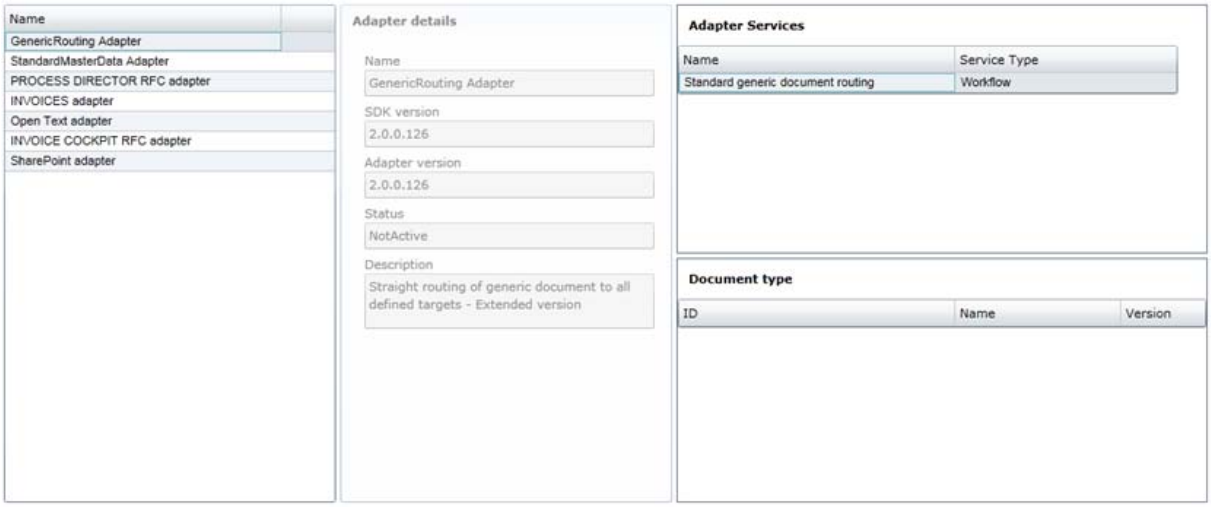
4. Click Add to create a new file containing default classes.



5. The file contains two classes: the class for the service properties and the class that implements the interface required for the target adapter. Both these classes are named using the name given in the template.

```
using ReadSoft.ServiceBus.SDK.Adapter;
using ReadSoft.ServiceBus.SDK.Interface;

namespace ReadSoft.ServiceBus.My_RSB_Adapter
{
    /// <summary>
    /// The service properties for MyDocumentTargetService1
    /// </summary>
    public class MyDocumentTargetServiceServiceProperties : TargetServiceProperties...

    /// <summary>
    /// The adapter service for MyDocumentTargetService1
    /// </summary>
    public class MyDocumentTargetServiceService : GenericDocumentTargetService<MyDocumentTargetServiceServiceProperties, AdapterProperties>...
}
```

Now you can build the solution and try to install the new version of the adapter. You should see the adapter service for you adapter on the Adapters page in the Administration UI.



Adding service properties

PublicProperty

You can use PublicProperty to create an adapter property and display it in the RSB user interface.

Syntax

```
PublicProperty(string name, string description,
AdapterPropertyBase.customType customType, string category, int order)
```

Part	Type	Description
name	String	A descriptive name for the property.
description	String	A description of the property. The description appears in the tooltip of the property control.
customType	Custom	Determines which controls are available to connect to the property. **FilePath** – This type is not used at this time. **Filters** – This is a general property that is automatically added to all target

adapters, so there is no need to implement it.

GenerateValue – Displays a text box and a button.

When this type is used, you should implement the interface, IPropertyConfigService. IPropertyConfigService exposes the function, ConfigureProperty, which is called when the button is clicked. You can use the function to generate a value, for example, or perform other operations.

Normal (default) – Displays a variety of controls depending on the PublicPropertyAttribute type. Use the table below to determine which control is displayed:

Type	Control
String	Text box
Boolean	Check box
Enum	Combo box
TimeDate	Date picker
Int	Spinner control

Password – Displays a password text box.

Scheduler – Displays controls for scheduling master-data activities.

List – Displays list of values that can be populated in the adapter code.

category	string	The name of the category you want the property reside in. When you have many properties, you can use the category to

		organize the properties into different groups (categories). Each category appears as an according control in the user interface.
order	integer	The order in which you want the control to appear in relation to other properties within the same category.

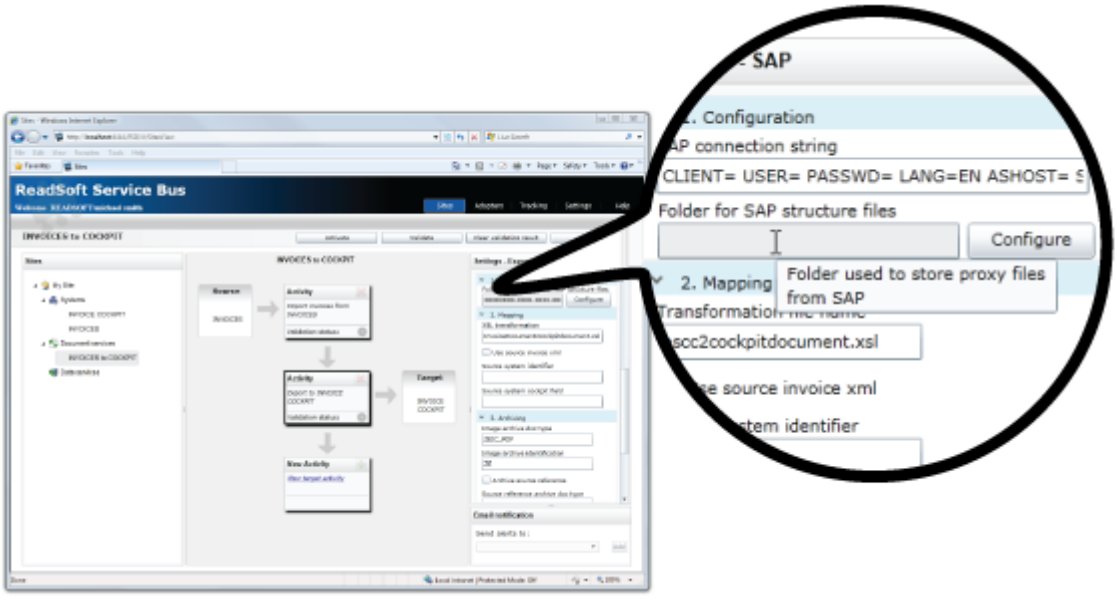
Examples

The following code excerpt creates a property named CarColor. Because the property is a string, a text box will be displayed. The name parameter creates a label above the text box, “Car color”, and when the mouse moves over the text box, "The color of the car" will be displayed in the tooltip.

```
[PublicProperty("Car color", "The color of the car")]
    public string CarColor = string.Empty;
```

The following code produces a property which includes the Folder for SAP structure files box and the Configure button as seen in the picture that follows:

```
[PublicProperty("Folder for SAP structure files", "Folder used to store
proxy files from SAP", AdapterPropertyBase.CustomTypes.GenerateValue,
"1. Configuration",2)]
      public string ProxyFileFolder;
```



See more PublicProperty examples from the example projects.

PublicProperty examples

The code below references two example projects, FileTargetAdapter and FileTargetAdapterEx, and demonstrates how to extend FileTargetAdapter with two new properties: one that defines where to save the attachments files, and one that defines an XSLT file transforms the XML document before writing the file.

The code also demonstrates how to group the properties into groups in the UI. Groups make it easy for users to find properties. To create a group, add an extra string after the description ("1. Folders" and 2. Mapping" below). This string gives the name of the group and will be sorted alphabetically. To specify the order of the groups, start the string with a number as shown in the example.

The order of properties within a group are sorted by adding a sequence number after the group name.

```
public class FileTargetAdapterProperties : TargetServiceProperties

   {

      [PublicProperty("TargetFolder", "Folder where XML files are
written.", "1. Folders", 1)]

      public string TargetFolder = @"C:\FileTargetAdapterTargetFolder";

      [PublicProperty("AttachmentFolder", "Folder where attached files
are written.", "1. Folders", 2)]

      public string AttachmentFolder =
@"C:\FileTargetAdapterTargetFolder\Attachments";

      [PublicProperty("Transformation file name",
"TransformationFileName xslt", "2. Mapping", 2)]

      public string TransformationFileName = "transform.xsl";
```

Validating adapter properties

The IValidation interface is implemented via the target adapter base class. It is used to test adapter properties for errors before they are used. This interface defines three operations:

- The **PerformValidation** function, which you customize to detect errors in the settings of the adapter.
- The **RouteActivated** method, which notifies your adapter of new settings.

- The **RouteDeActivated** method, which notifies your adapter that the service is not active anymore.

Whenever a service is activated, the validation service is triggered and the PerformValidation function is called. If all of the activities that configure the service validate without problems, the RouteActivated method is called. When a route is deactivated, the RouteDeActivated method is called.

The base class provides empty methods. To implement a specific behavior, you must override the following methods:

```
virtual protected bool PerformValidation(T serviceProperty, TU
adapterProperties, Guid activityID, out Dictionary<string, string>
errorMessages)

virtual protected void RouteDeactivated(T properties, TU
systemProperties)

virtual protected void RouteActivated(T properties, TU
systemProperties)
```

- **PerformValidation** – Called by the bus when a service is validated. If false is returned, the validation fails and the service cannot activate.
- **serviceProperties** –Specifies the class which defines the settings of your adapter service, which are used when configuring the activities within a service.
- **adapterProperties** – Specifies the class which defines the settings of your adapter, which are used to configure the system entities.
- **activityID** – The unique ID of the activity to validate.
- **errorMessages** – Return parameter which contains error messages from the validation (if any).

Adding special configuration behavior

The IPropertyConfig interface is implemented via the target adapter base class and is used to get special configurations needed for one or more properties. This interface defines one operation ConfigureProperty.

virtual protected string ConfigureProperty(Activity activity,T serviceProperties, TU adapterProperties)

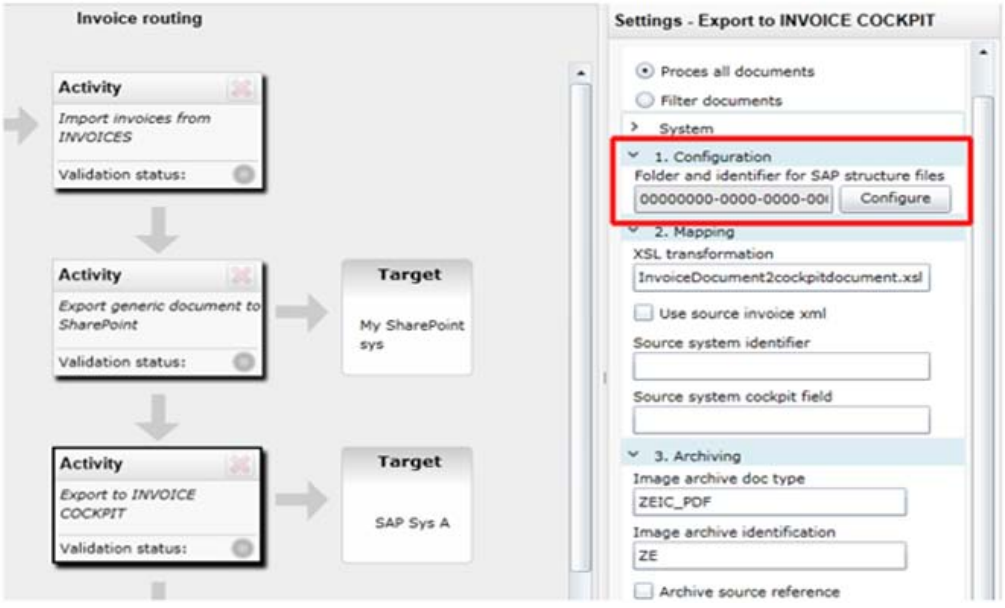
- **ConfigureProperty** – Called by the bus when a service is validated. If false is returned, the validation fails and the service cannot activate.
- **serviceProperties** –Specifies the class which defines the settings of your adapter service, which are used to configure activities within a service.
- **adapterProperties** – Specifies the class which defines the settings of your adapter, which are used to configure the system entities.
- **activity** – The activity that to configure.
- **errorMessages** – Return parameter which contains error messages from the validation (if any).

Example

For the INVOICE COCKPIT connector, we need to retrieve structures that are used in the interface. To accomplish this we request the structures from SAP. The property is tagged with **AdapterPropertyBase.CustomTypes.GenerateValue**.

```
[PublicProperty("Folder for SAP structure files", "Folder used to store
proxy files from SAP", AdapterPropertyBase.CustomTypes.GenerateValue,
"1. Configuration",2)]
public string ProxyFileFolder;
```

The code above displays the following result in the UI.



When the user presses the Configure button, **ConfigureProperty** is called, and the adapter sets and returns the configuration object.

Using the logging service

The ReadSoft Service Bus has a central logging service called LogBookManager. This should be used by the adapters for logging functionality. It is exposed via WCF endpoint.

Connecting to the WCF endpoint and adding log entries is easy, when using the RSB SDK, as long as the adapter is hosted by RSB.

The rest of this section only deals with adapters that are hosted by RSB.

In order to write to the RSB logging service, the adapter should use the static methods in ReadSoft.ServiceBus.LogServiceProxy.Log. The following methods are available, each with several overloads:

- **LogError** – Logs an error message.
- **LogException** – Logs and exception.
- **LogMessage** – Logs and general message.
- **LogNewAlert** – Add an alert to the log.

The service has static help methods that hide the details on the service call when an administrator needs to take action on an error. These methods are found in the ReadSoft.ServiceBus.LogServiceProxy.dll.

The following methods can be used for different logging purposes:

public static void LogError(Guid objectid, string routeName, object callingclass, string message)

public static void LogError(Guid objectid, string routeName, string classname, string message)

public static void LogException(Guid objectid, Exception error)

public static void LogException(Guid objectid, Exception error, string additionaltext)

public static void LogMessage(Guid objectid, object callingclass, string message, LogbookEntry.LogLevel logLevel)

public static void LogMessage(Guid objectid, string classname, string message,LogbookEntry.LogLevel logLevel)

public static bool LogNewAlert(TrackingAlert alert)

public static void LogNewAlert(Guid instanceId, Guid routeId, TrackingInfo.TrackingAlertsType alertType, string description, string source)

Example 1

```
LogMessage(documentInfo, MethodBase.GetCurrentMethod(), " entered");
```

Example 2

```
LogException(documentInfo, ex);

LogMessage(documentInfo, MethodBase.GetCurrentMethod(), " exiting after
exception");
```

Working with adapter properties

Adapter properties enable the configuration of a service within the bus. The properties are stored and managed within the bus. There is usually no need to store external properties (example: configuration files) for adapters, unless they are shared among many instances of an adapter. The properties are controlled by attributes that tell the UI how to display them. In the sample below, we add additional properties to the adapter from the previous section, and show how to control the property via the attributes that are built into RSB.

There are two categories of properties within an adapter:

- Adapter system properties (adapter properties) - common for all activities that are created by the adapter service. You find these settings under the System node on RSB's Administration page.
- Adapter service properties (service properties) - specific for the adapter service (an instance of the adapter) and are used while configuring the service. These settings are displayed when you select an activity in the design area on RSB's Administration page.

Adapter properties

Adapter properties are properties that are common for all activities and are created from the adapter service.

```
   public class AdapterProperties : IAdapterProperties

   {

      public AdapterInformation GetAdapterInfo()

      {
```

```
         return new AdapterInformation

         {

            RootGuid = AdapterHelper.Rootguid(),

            Name = "My RSB Adapter1",

            Description = "The My RSB Adapter1 adapter was generated
for you," +

                          "from a Visual Studio template.",

            AdapterVersion = AdapterHelper.AdapterVersion()

         };

      }

      [PublicProperty("MySystemProperty", "Sample system property for
the My_RSB_Adapter1 adapter.")]

      public string MySystemProperty = @"my connectionstring";

   }
```

Adapter service properties

These properties are specific for the adapter service and are used when configuring the service.

```
   public class SubmitInvoiceServiceProperties :
TargetServiceProperties
   {
      [PublicProperty("StringProperty", "Sample string property for the
SubmitInvoice adapter service.")]
      public string StringProperty = @"StringProperty";
      public override string Name()
      {
         return "SubmitInvoice";
      }
      public override Guid ID()
      {
         return new Guid("d28c6cec-f8ac-4481-8326-4cb1f913036c");
      }
```

```
        public override string Description()
        {
            return "SubmitInvoice does something.";
        }
        public AdapterService.ServiceTypes Type()
        {
            return AdapterService.ServiceTypes.TargetAdapter;
        }
    }
```

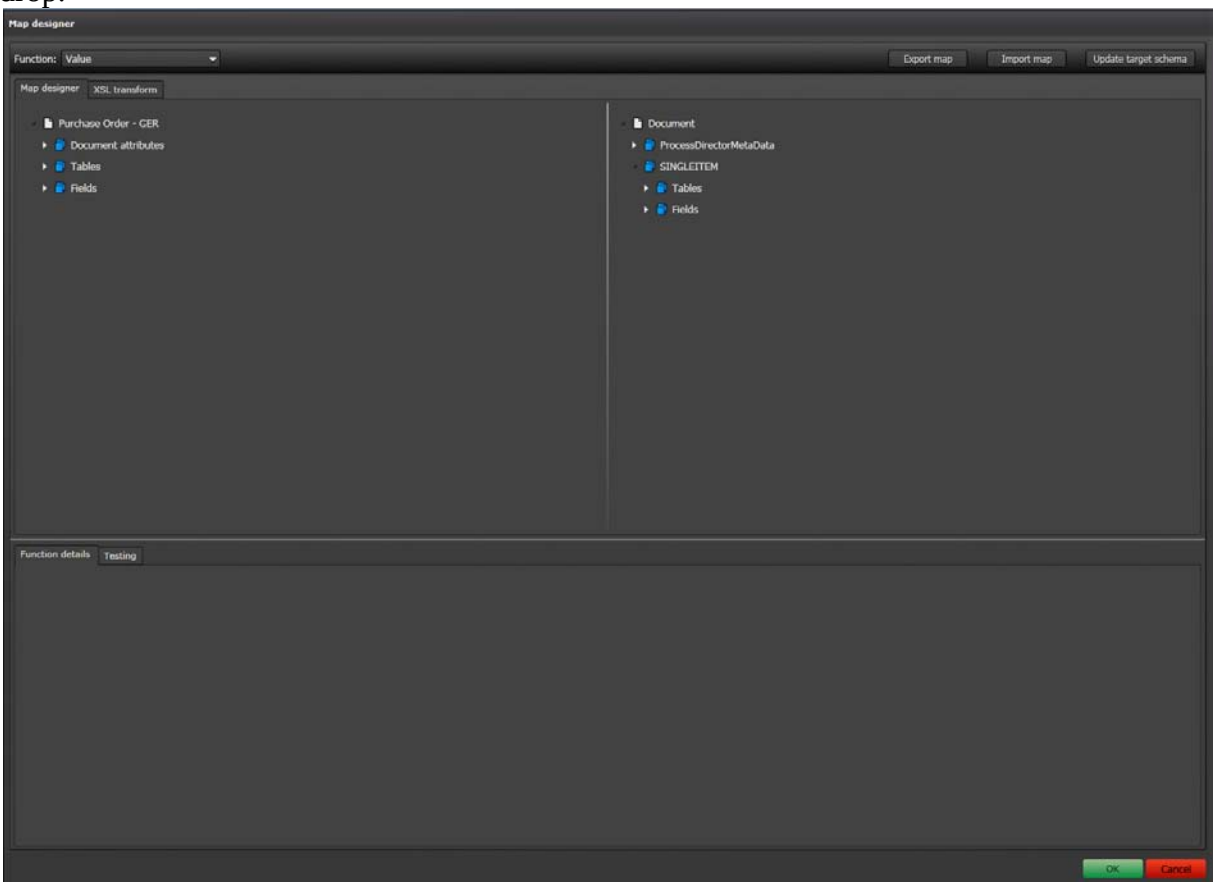
Properties have a PublicProperty that declares how RSB exposes it for the user. In the sample above, StringProperty is exposed in the UI as “StringProperty” with a description “Sample string property for the SubmitInvoice adapter service.". For a full description on how to use the attributes see Attributes and PublicPropertyAttribute.

Creating a mappable target document service

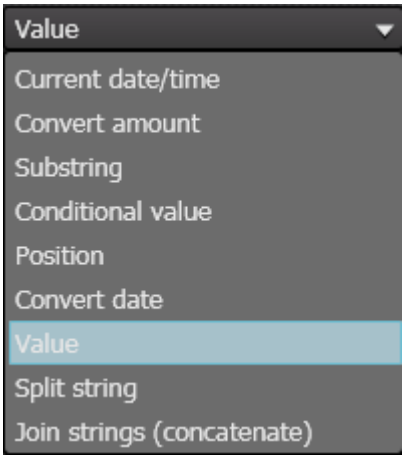
Introduction

When creating a target service it is a good idea to make it mappable, meaning the RSB administration user can use Map designer to select fields from the source and map them to fields for the target. The mapping from a source field to a target field is done by drag-and-

drop.

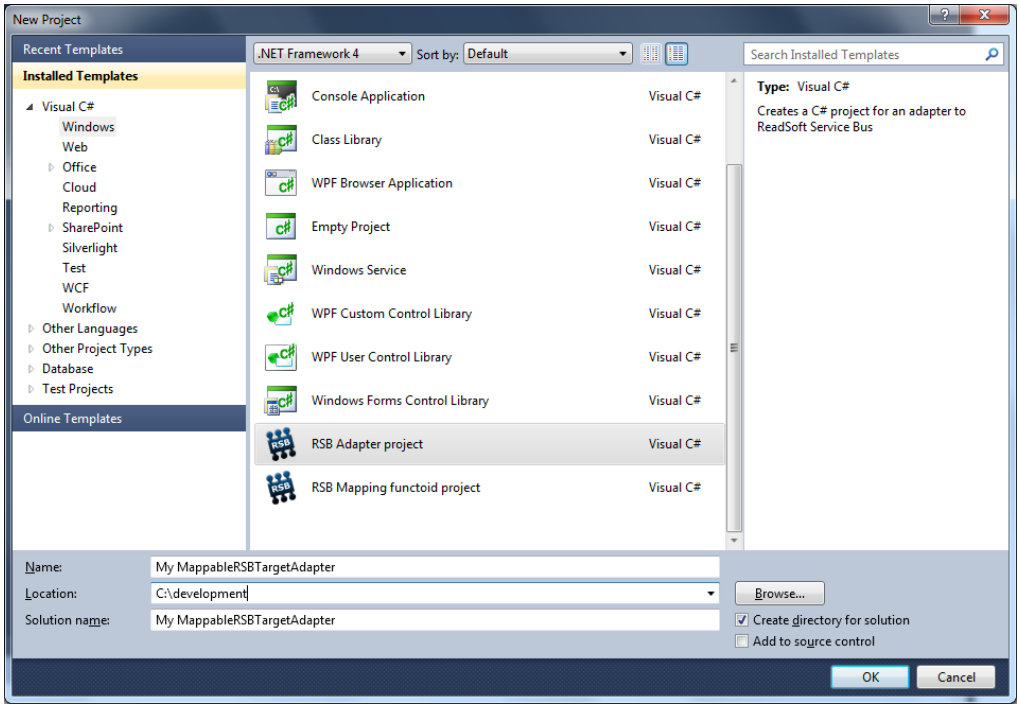


In the upper left corner the Function can be seen. This function is the type of mapping that is applied. In most cases just copy value from source field to target field, but the Function can be either one of the pre-installed Functions, or you can develop your own using a Visual Studio template.

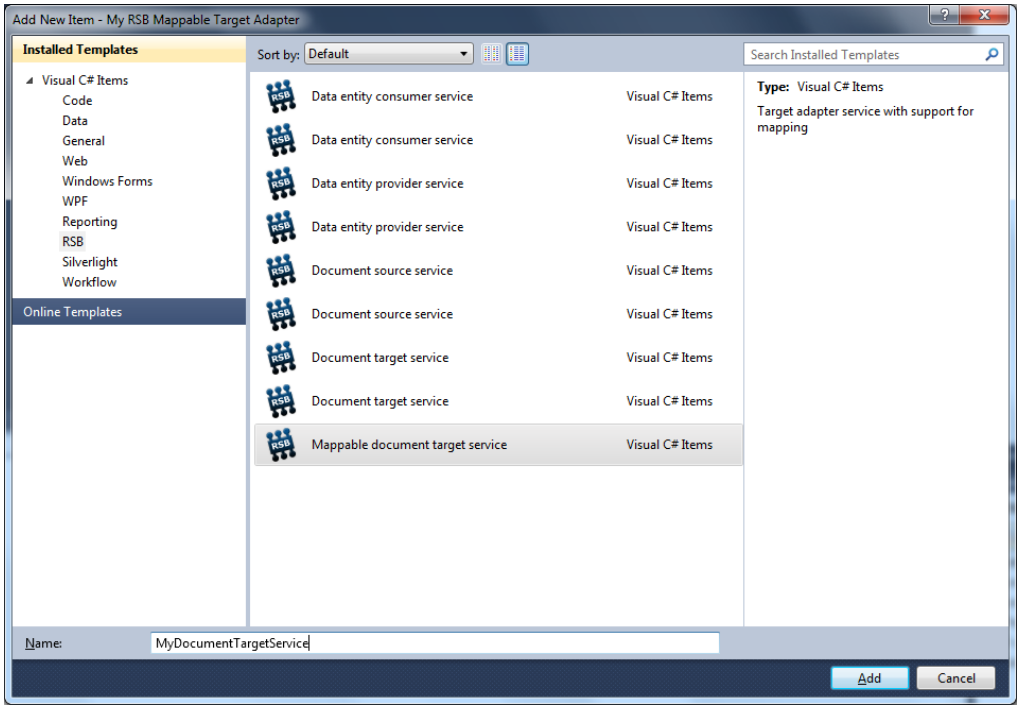


How to create the Adapter

1. Create a new adapter project.



2. And add a Mappable document target service.



After you add the target service via the template, Mappable document target service, a file containing following classes is generated:

```
public class MyDocumentTargetServiceServiceProperties :
MappableTargetServiceProperties
public class MyDocumentTargetServiceService :
GenericMappableDocumentTargetService<MyDocumentTargetServiceServiceProp
erties, AdapterProperties>
```

Classes

- **MyDocumentTargetServiceServiceProperties** – implements settings for the adapter service.
- **MyDocumentTargetServiceService** – implements the actual services that submit the document to the target system and describes the schema.

MyDocumentTargetServiceService implements a stub of a very simple target adapter that processes documents. It inherits the abstract base class, GenericMappableDocumentTargetService, which provides basic functionality and gives directives about what to implement.

Mandatory

The adapter service class inherits all mandatory interfaces required for a mappable target adapter service via the base class, GenericMappableDocumentTargetService.

The methods that are mandatory to implement are:

- SubmitGenericDocument
- CreateMappingSchema

SubmitGenericDocument

This method is called by the document service. Since we are implementing a mappable target adapter the xml in the GenrericDocument is mapped by RSB according the the user mapping set up in Map designer. If you work with the xml this of course possible. Then connect to the target system and send document.

- **serviceProperties** – holds the properties for the adapter service
- **systemProperties** - holds the properties for the adapter system
- **Document** – holds the document
- **serviceConfigurationID** – the service configuration for the document service

CreateMappingSchema

Implement this method to describe the RSB schema for the target. In some cases you might want to connect to your target and dynamically create the schema. In other cases you can statically create the RSB schema or you might want to import an XML schema.

If you want to import a XML schema, you can use the class, XmlSchemaParser, implemented in the SDK assembly ReadSoft.XML.Mapping.Core.dll. This import does not support all types of XML schemas, only very simple schemas can be imported.

An instance of the class, Schema, is returned by the CreateMappingSchema method. Schema can contain groups and fields. Schema is itself also a group, and as such it has a System name and d Display name. A group in Schema can also have groups and fields and has a System name and a Display name.

If you want a group to be dynamic, meaning that a Map designer user can create fields in the group dynamically, set the property AllowsDynamicFields to true.

Here is an example of a target activity schema:

```
var schema =
new Schema
{
   SystemName = "Document",
   DisplayName = "Document",
};
schema.Groups =
new SchemaGroupList
{
   new SchemaGroup
   {
      SystemName = "ProcessDirectorMetaData",
      Fields =
      new SchemaFieldList
      {
         new SchemaField
         {
            SystemName = "DocumentType"
         },
         new SchemaField
         {
            SystemName = "Origin"
         },
         new SchemaField
         {
            SystemName = "MappingId"
```

```
            }
        }
    },
    new SchemaGroup
    {
        SystemName = singleItemNodeId,
        AllowsDynamicFields = true
    }
};
```

RSB schema node attributes

Each node in a schema (such as a field, group, table or even the schema itself) can be given static attributes that are always there.

These attributes have names and values, and are realized as regular XML attributes after mapping.

To add an attribute, just create an instance of SchemaNodeAttribute and add it to the node's Attributes list of attributes. For example, in order always to have a certain query string ready when a document arrives for a certain target system, the target adapter can add the following attributes to the main schema:

```
schema.Attributes = new List<SchemaNodeAttribute> { new
SchemaNodeAttribute { Name = "SqlString", Value = finalQueryString } };
```

Optional

The following methods are optional but highly recommended:

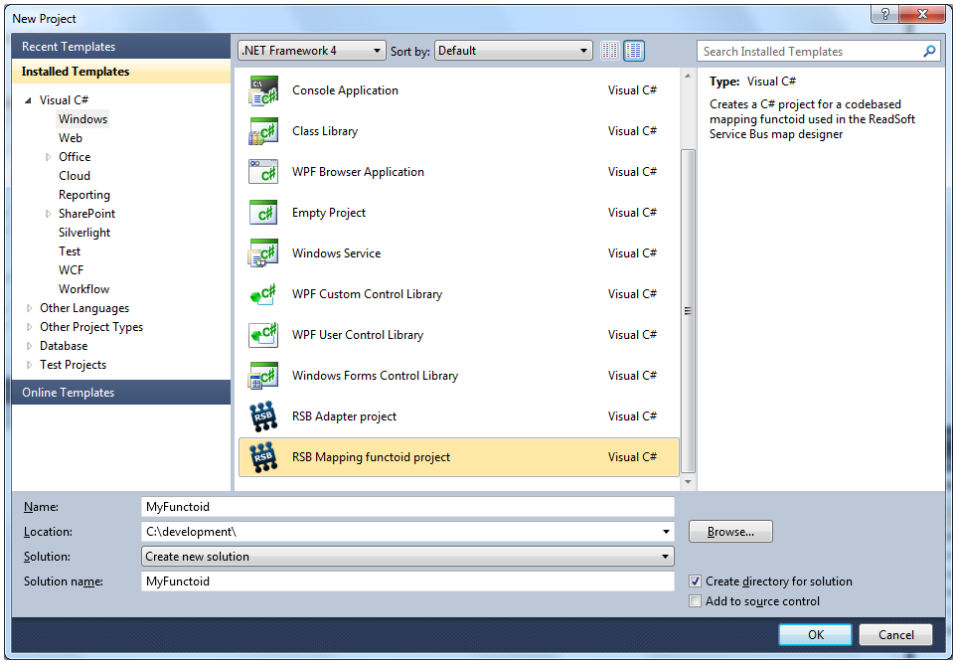
- **Validation** – If you need to perform a custom validation on your adapter, you can override the PerformValidation method with Validation().
- **Testing connection** – If you need to perform a special action when the framework is testing the connection of a service, you can override the TestConnection() method.

Creating a Map designer Function

Introduction

Sometimes the pre-installed functions for mapping in Map Designer is not enough and you need to create your own. In the Visual Studio templates the functions are called functoids.

1. Create a new RSB Mapping functoid project.



Implementation

After creating the project you get the following generated code:

```
/// <summary>

/// Sample functoid that shows how to implement a functoid that
executes code in the runtime transformation process

/// </summary>

[Export(typeof(Functoid))]

public class MyMappingFunctoid : CodeFunctoid

{

   private static readonly Guid Id = new Guid("6d341a06-e870-45b4-b164-
c69ea7bf2395");

  [CodeFunctoidMethod]

    public string DoStuff(string myParam, string myOtherParam)

    public override Guid FunctoidId

    public override FunctoidInfo GetInfo()
```

```
}
```

The Id is just a key and does not need to be changed.

If you want more, or fewer, parameters in your function, change the GetInfo method.

In the method DoStuff, you implement the actual function and all your parameters that you described in GetInfo will be parameters to DoStuff. RSB knows to call DoStuff because of the [CodeFunctoidMethod] attribute.

In SDK installation there are two function samples. One sample of a code-based function of the type described above. There is also a sample for XPath function. In XPath functions, you must overwrite the method, EmitXPath.

Install your function

After you implement your function and test it with unit tests, make sure the assembly name has "functoid" somewhere in the name. Copy the assembly to the `ReadSoft\RSB windows service\WebHost\bin` folder. On Windows 7 the default is `C:\Program Files (x86)\ReadSoft\RSB windows service\WebHost\bin`.

After you copy the assembly, you must restart the Map Designer host.

- If you are using IIS as a host, restart IIS.
- If you are using stand-alone hosting, right-click the RSB icon in the system tray, select **Close RSB** and start RSB again.

Creating a target document service

Introduction

In this section we will dig a little deeper in the target-document service which we already created in *Adding an adapter service*. In that section, you created a target service that transfers invoices to your system. When you added the target service via the template, Document Target Service, a file containing following classes was generated:

```
public class SubmitInvoiceServiceProperties : TargetServiceProperties
public class SubmitInvoiceService :
GenericDocumentTargetService<SubmitInvoiceServiceProperties,
AdapterProperties>
```

Classes

- **SubmitInvoiceServiceProperties** – implements settings for the adapter service.

- **SubmitInvoiceService** – implements the actual services that submit the invoice to the target system.

SubmitInvoiceService implements a stub of a very simple target adapter that processes invoices. It inherits the abstract base class, GenericDocumentTargetService, which provides basic functionality and gives directives about what to implement.

```
public class SubmitInvoiceService :
GenericDocumentTargetService<SubmitInvoiceServiceProperties,
AdapterProperties>
```

Mandatory

The adapter service class inherits all mandatory interfaces required for a target adapter service via the base class, GenericDocumentTargetService.

One method must be implemented: SubmitGenericDocument.

SubmitGenericDocument

This method is called by the document service. Since we are not implementing a mappable target adapter the xml in the GenrericDocument is the xml that is sent by the source adapter. Process the xml and then connect to the target system and send document.

- **serviceProperties** – holds the properties for the adapter service.
- **systemProperties** - holds the properties for the adapter system.
- **Document** – holds the document.
- **serviceConfigurationID** – the service configuration for the document service.

Optional

The following methods are optional, but highly recommended:

- **Validation** – If you need to perform a custom validation on your adapter, you can override the PerformValidation method with Validation().
- **Testing connection** – If you need to perform a special action when the framework is testing the connection of a service, you can override the TestConnection() method.

Asynchronous target services

Asynchronous target services can only be used when the source document service uses asynchronous transfer. What makes a target service asynchronous is that it can return IN_PROGRESS from SubmitGenericDocument and then implements the GetDocumentState

function, which allows a source system repeatedly to query the target service for document state via RSB.

An asynchronous target service normally returns IN_PROGRESS from the SubmitGenericDocument function, but it can also return COMPLETE, ERROR or REJECTED.

Asynchronous transfer goes through the following steps

1. Transfer is initiated through a call to SubmitGenericDocument. This function call typically returns IN_PROGRESS.

2. Transfer is ongoing, but SubmitGenericDocument has exited. An external system is working on the document and will eventually return a result, for example by writing a file to a network share.

3. The source system and RSB can ask the target service any number of times about the state of the document by calling GetDocumentState. These calls return IN_PROGRESS until the external system has returned a result.

4. The external system returns a result.

5. Subsequent calls to GetDocumentState return the result from the external system.

GetDocumentState

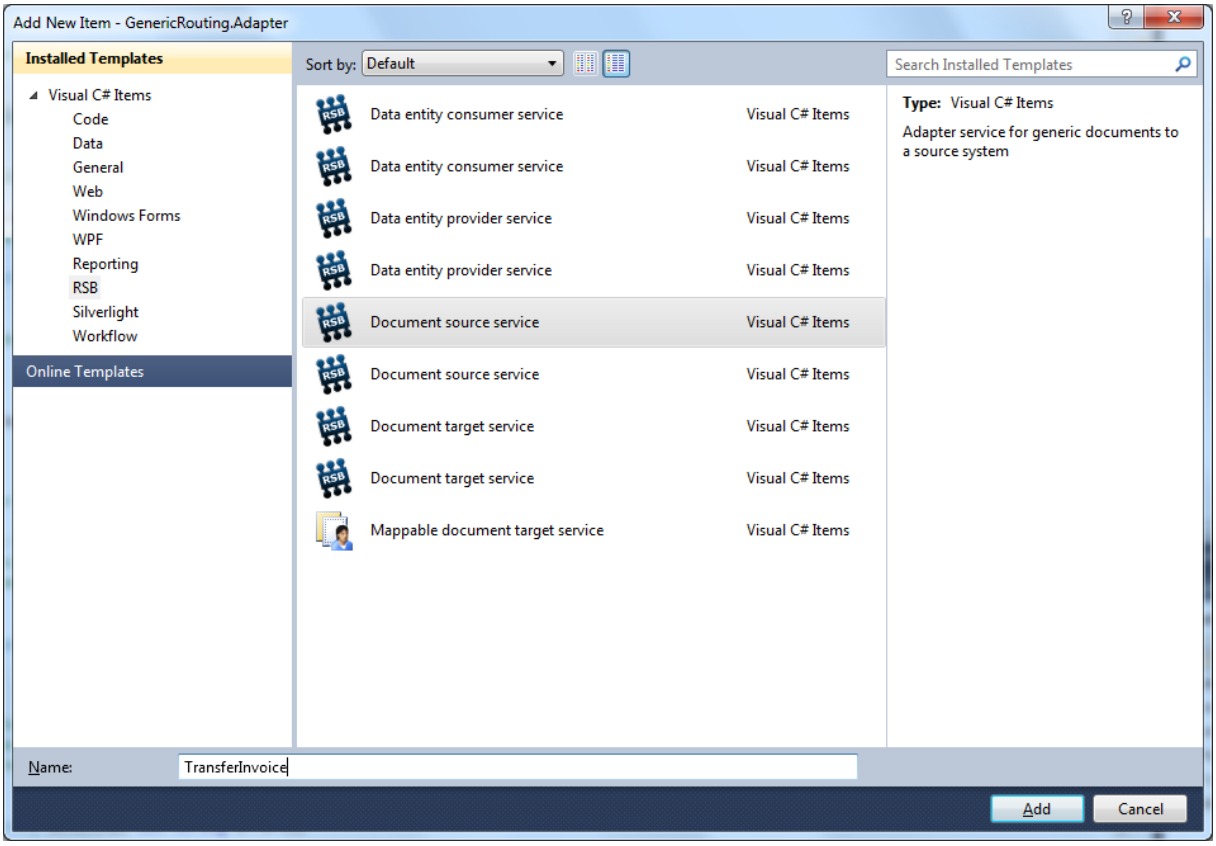
The GetDocumentState function only needs to be implemented by Asynchronous target services. It is capable of retrieving a document state using the GUID of the document and returning this state with any comment inside a Result object.

Creating a source document service

The source-document service is the front end to external systems that access the service bus. The adapter that provides this service exposes an interface to the system that wants to transfer a document to one or more targets. In this section, we show how to create a source-adapter service.

1. Create an adapter project as described in Creating an adapter project.

2. In your adapter project, right-click the project and select **Add** > **New item**.



3. Select the RSB in the Categories list to display the templates you can choose. For this first exercise, we will use the Document source service template.

4. Give the adapter service an appropriate name. The name is used to create the default names for the auto-generated classes.

5. Click Add to create a new file containing default classes.

Classes

The following classes are generated by the template:

```
namespace ReadSoft.ServiceBus.My_RSB_Source
{
   public class TransferInvoiceServiceProperties : IServiceProperties
   public class TransferInvoiceService :
       GenericDocumentSourceService<TransferInvoiceServiceProperties,
AdapterProperties>
}
```

- TransferInvoiceServiceProperties – contains the properties for my adapter service.

- TransferInvoice –implements the adapter service for transfer of invoices. It inherits the source-adapter service base class, GenericDocumentSourceService, which provides the basic behavior for the service.

GenericDocumentSourceService

The adapter-service class inherits all mandatory interfaces via the base class, GenericDocumentSourceService. IGenericDocumentSource has the following methods that must be implemented:

```
override public GenericDocumentState GetDocumentState(Guid docId)
override public Result ProcessDocument(GenericDocument document)
public override Result ProcessDocumentSynchronous(GenericDocument
document)
public override Result ReleaseDocument(Guid docId)
```

- **GetDocumentState** – Returns the state of the document with the ID passed as parameter.
- **ProcessDocument** – Starts an asynchronous transfer process of the document to the target. Control returns after the call to the bus which starts the actual transfer.

  Note: You can choose whether you want to work with asynchronous processes (ProcessDocument) or synchronous processes (ProcessDocumentSynchronous).
- **ProcessDocumentSynchronous** – The synchronous alternative of the ProcessDocument. After the adapter calls the bus and gets control back, the transfer of the document is finished with a positive or negative result.

  Note: You can choose whether you want to work with asynchronous processes (ProcessDocument) or synchronous processes (ProcessDocumentSynchronous).
- **ReleaseDocument** – Used in the asynchronous call to let the bus know that it everything is OK and it is OK to release the document.

Expose your own interface

To see how to expose your own service interface, see the data services example in SDK Examples.

Properties

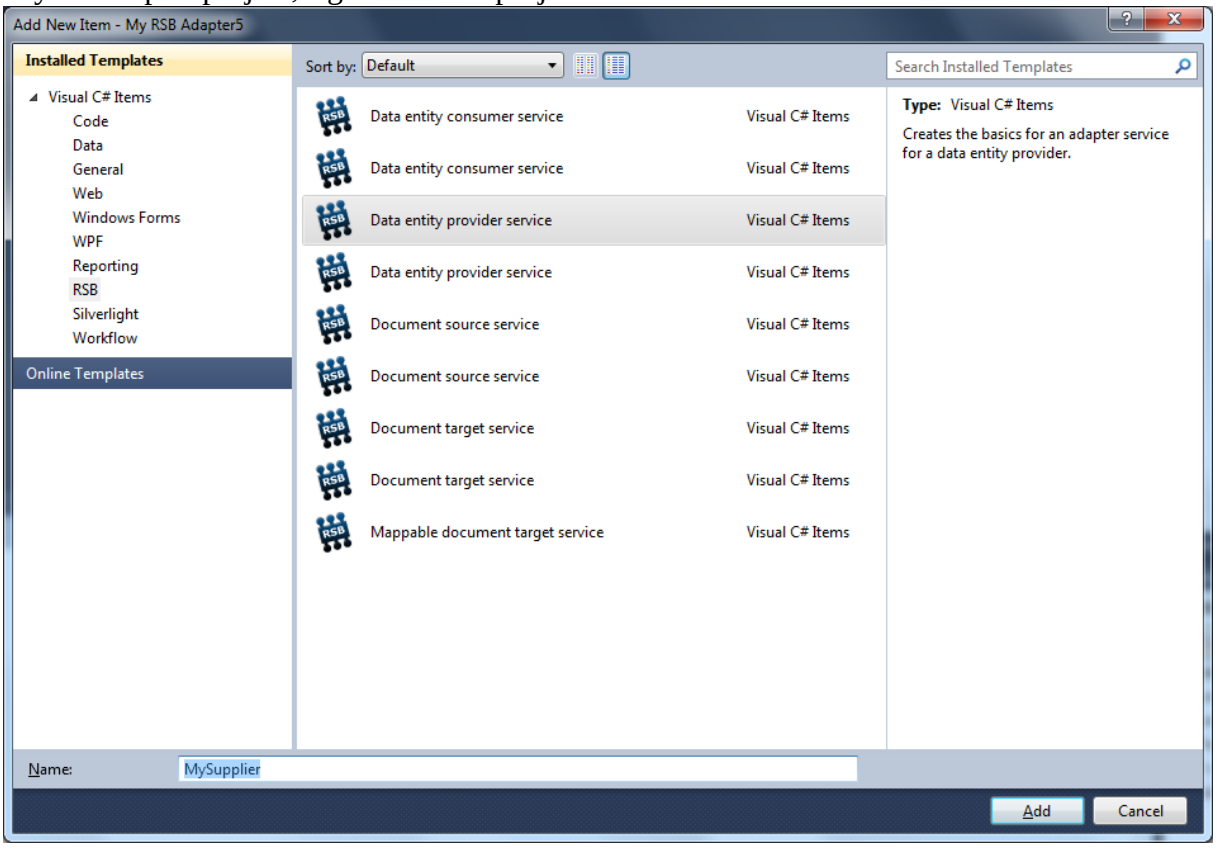
See target adapter service.

Creating a provider data service

A provider adapter supports services that access different data entities (supplier, buyer, purchase order etc.) within the target system. Each data entity is handled in its own adapter service. The SDK provides some tools that make it easy to start developing these services. In this section, we show how to create a provider data service.

1. Create an adapter project as described in Creating an adapter project.

2. In your adapter project, right-click the project and select **Add** > **New item**.



3. Select the RSB in the Categories list to display the templates you can choose. For this first exercise, we will use the Data entity provider service template.

4. Give the adapter service an appropriate name. The name is used to create the default names for the auto-generated classes.

5. Click Add to create a new file containing default classes.

Classes

The following classes are generated by the template:

```
namespace ReadSoft.ServiceBus.MyTest

{

   public class MySupplier

   public class MySupplierQuery

   public class MySupplierServiceProperties :
MasterDataOnlineProperties, IServiceProperties

   public class MySupplierList : List<MySupplier>

   public class MySupplierService :
DataEntityProviderService<MySupplierServiceProperties,

                  AdapterProperties, MySupplier, MySupplierQuery>

}
```

- **MySupplier** – implements the data entity.
- **MySupplierQuery** – implements the query for MySupplier.
- **MySupplierServiceProperties** - the properties for the adapter service (MySupplierService).
- **MySupplierList** – help class for a list of MySupplier. This class converts XML into data which you can work with more easily.
- **MySupplierService** – implements the adapter service for the MySupplier data entity. It inherits the provider service base class which provides the basic behavior.

You implement the actual service for this data entity in MySupplierService. The following method is called by the service bus and that implements the interface to the provider system:

```
public override List<MySupplier> Query(MySupplierQuery queryDataEntity,

MySupplierServiceProperties serviceProperties, AdapterProperties
adapterProperties)

{

    return GetMySuppliers(queryDataEntity, serviceProperties,
adapterProperties);
```

```
}
```

The method above is called by the implementation of an interface in the base class:

```
[ServiceContract]

public interface IDataEntityProvider

{

   [OperationContract]

   EntityList Query(string serializedQuery, Activity activity);

   [OperationContract]

   string SupportedDataEntityType();

   [OperationContract]

   string SerializeEntity();

}
```

- **Query** – returns a list of entities as strings that was found based on the query passed as the parameter. The Activity is the configuration to be used when accessing the target.
- **SupportedDataType** – returns the name of supported data type.
- **SerializeEntity** – used for validating the service configuration. The method is implemented in DataEntityProviderService and returns the serialized data entity.

  The DataEntityProviderService is the base class that implements a general behavior for this interface and that you normally use when developing your own adapter.

```
public abstract class DataEntityProviderService<TActivityProperties,
TSystemProperties, TDataEntity, TQueryDataEntity> :

AdapterService<TActivityProperties, TSystemProperties>,
IDataEntityProvider

where TActivityProperties : MasterDataOnlineProperties,
IServiceProperties, new()
```

```
where TSystemProperties : IAdapterProperties, new()

where TQueryDataEntity : new()
```

- **EntityList Query(…)** – implements the base functionality in the of the query for the specific data entity that the service supports. It calls the abstract method in the class, which perform the actual query. Since it is an abstract method, you need to override it in the adapter service you implement.

- List<TDataEntity> Query(…)

  public abstract List<TDataEntity> Query(TQueryDataEntity queryDataEntity, TActivityProperties properties, TSystemProperties systemProperties);

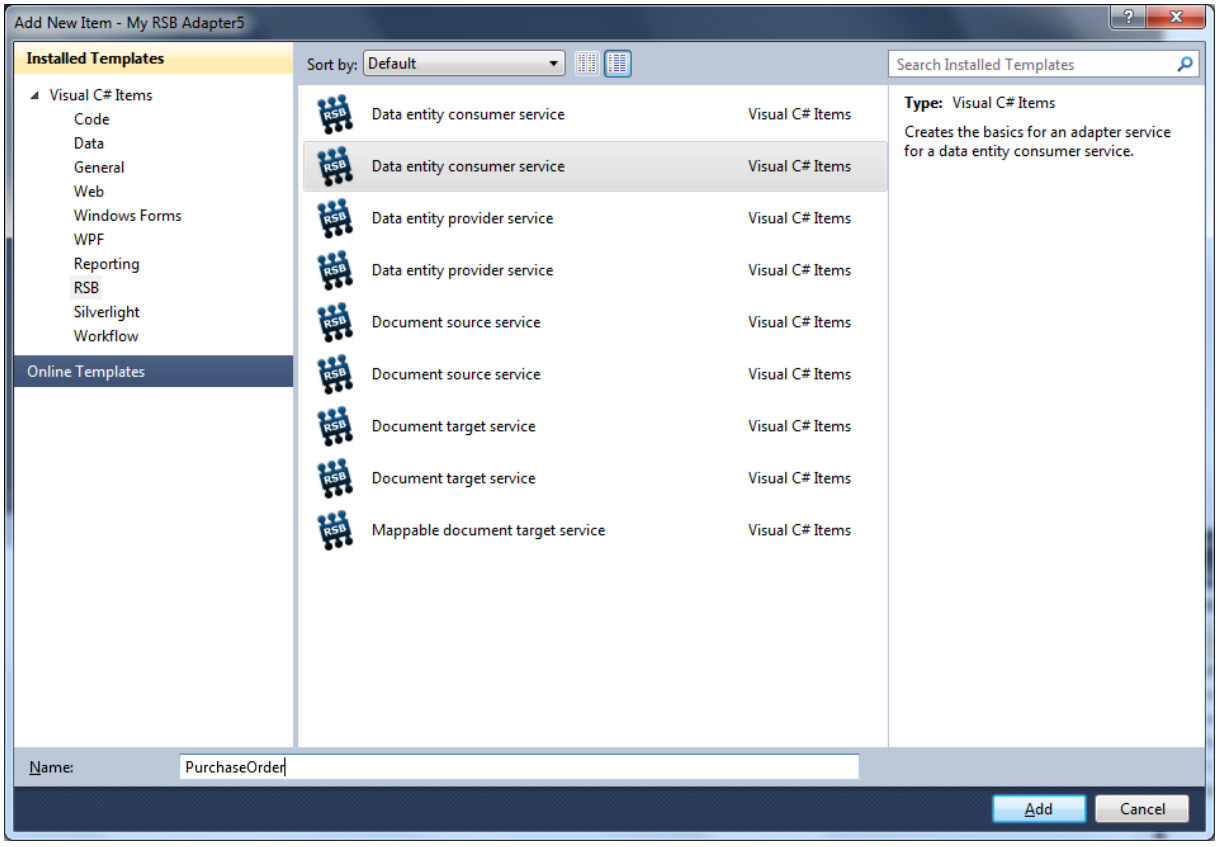
  Since it also inherits from the AdapterService class, you get standard behavior for an adapter service, such as validation and maintenance.

Creating a consumer data service

A consumer adapter provides a front end to an external target system to access different data entities (supplier, buyer, purchase order etc.). Each entity is handled in its own adapter service. To avoid duplicate code when creating a data-entity consumer and data-entity provider for the same data-entity type, we recommend moving the entity to a separate assembly that is shared between consumer and provider. The SDK provides some tools that make it easy to start developing these services. In this section, we show how to create a consumer data service.

1. Create an adapter project as described in Creating an adapter project.

In your adapter project, right-click the project and select **Add->New item**.



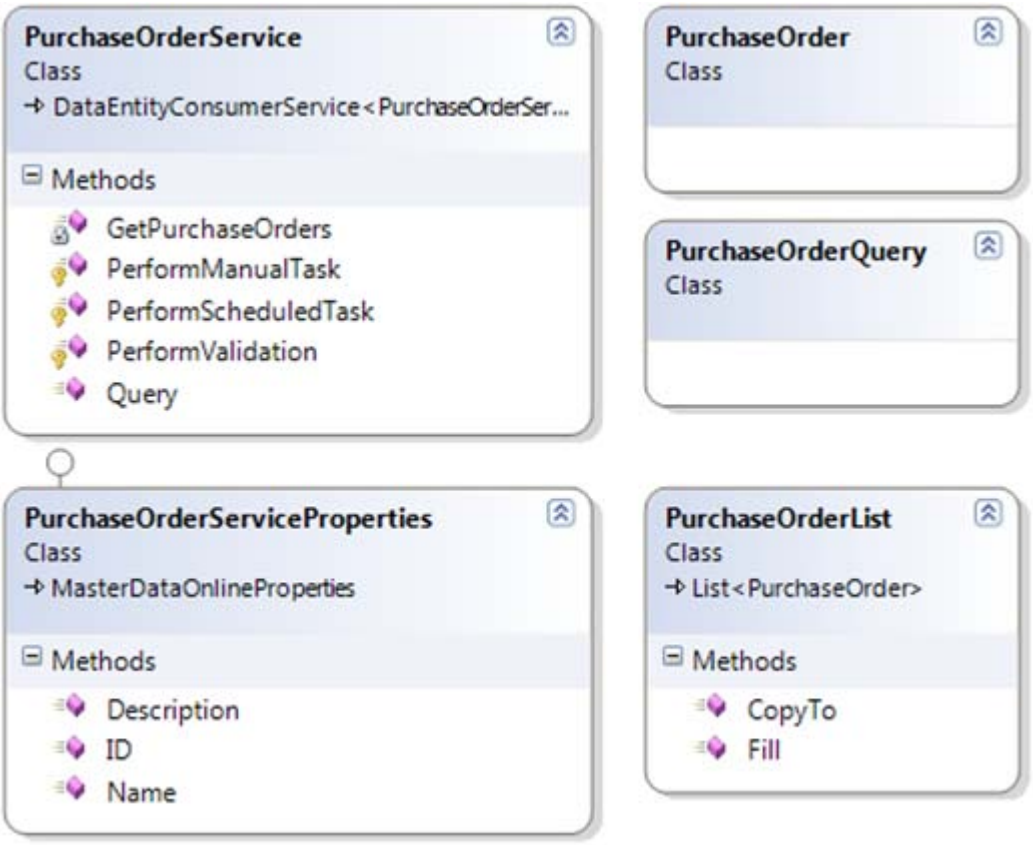
2. Select the RSB in the **Categories** list to display the templates you can choose. For this first exercise, we will use the **Data entity consumer service** template.

3. Give the adapter service an appropriate name. The name is used to create the default names for the auto-generated classes.

4. Click **Add** to create a new file containing default classes.

In this example we will use PurchaseOrder as an example of a data entity.

Classes



- **PurchaseOrder** – implements the data entity.
- **PurchaseOrderQuery** – implements the query for my data entity.
- **PurchaseOrderServiceProperties** - The properties for the adapter service (PurchaseOrderService).
- **PurchaseOrderList** – help class for a list of PurchaseOrders. This class converts XML into data which you can work with more easily.
- **PurchaseOrderService** –implements the adapter service for the PurchaseOrder data entity. It inherits the provider service base class which provides the basic behavior.

You do not actually implement anything in PurchaseOrderService to enable the external system to query the data entity. You can query the target for a specific entity via IDataEntityConsumer, which is implemented by the base class.

```
[ServiceContract]
public interface IDataEntityConsumer
{
```

```
   [OperationContract]
   EntityList Query(string type, string serializedQuery, Guid
ServiceConfigurationID, int
                        ExternalID);
   [OperationContract]
   void DeserializeEntity(string type, string serializedQueryEntity);
}
```

In many cases, however, you might want to expose a more domain-specific service using typed entities. If this is the case, you can implement your own interface and let RSB host this for you. See Expose your own interface for more information.

Expose your own interface

This section will show how you can implement your own service that expose a more domain specific interface. The template to create a consumer service is prepared for this so it should be rather easy to set it up.

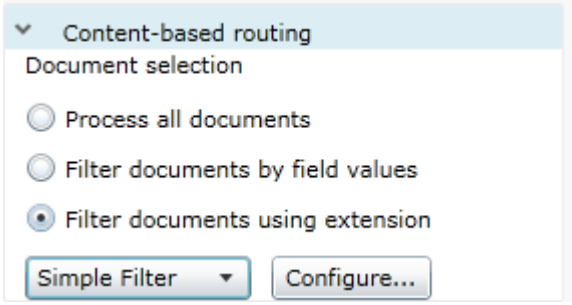
First, you need to uncomment the interfaces. It could also be a good practice to have this in a separate file:

```
[ServiceContract]
public interface IPurchaseOrder
{
   [OperationContract]
   List<PurchaseOrder> Query(PurchaseOrderQuery query, Guid
serviceConfigurationID, int
                               externalID);
}
```

```
public List<PurchaseOrder> Query(PurchaseOrderQuery query, Guid
serviceConfigurationID, int
                                    externalID)
{

    return QueryAny< PurchaseOrder, PurchaseOrderQuery>(query,
serviceConfigurationID,
                              externalID);
}
```

Working with service extensions

Service extensions are used for filtering before target activities are called by RSB. There is no template yet for creating a service extension adapter, but in your RSB SDK directory there is a sample project. If you build it and run the installation, **Simple Filter** will be visible in the user interface for all target activities.



Troubleshooting

Problem "NETWORK SERVICE cannot access ....\ASP.NET Temporary files"

This can happen if you install IIS after you have installed .NET framework.

Solution: Run "aspnet_regiis -i" in a command prompt. The tool is located in the .net framework directory

RSB Admin is not shown

Make sure the ".xap application/x-silverlight-app" MIME type is added to the RSB site in IIS

IIS -> Default web site -> RSB -> Properties -> HTTP Headers -> MIME Map -> File types

RSB log file shows "(405) Method not allowed"

RSB log file shows "(405) Method not allowed" and/or the web interface shows raw file content (such as "<%@ServiceHost language=c# Debug="true" Service="Microsoft.ServiceModel.Samples.CalculatorService" %>") when browsing the service. The solution can be found here:

http://msdn.microsoft.com/en-us/library/ms752252.aspx

Need more help?

If you have a problem that is not covered here, you can post issues on the RSB forum or the SDK forum.

If you have a question or comment about the SDK Help, you can post comments directly in Help.