Service Broker Foundations Workbench

14 August 2007
by Adam Machanic

```
/* Simple-Talk's Workbench series are intended to be loaded into
SQL Servers Management Studio, read and executed. To make the best
use of them we'd encourage you to try things out, to experiment
and generally use them more as a starting-off point with an aspect
of SQL Server. The actual source code is in a downloadable file in
the speech-bubble at the top of the article. As Service broker is
such an important introduction, we will be devoting a series of
three workbenches to it.*/
```

```
/*
```

-- Contents --

-- Introduction to SQL Service Broker --

```
SQL Service Broker (SSB) is a powerful asynchronous queuing and messaging
infrastructure available in all editions of SQL Server 2005. It provides tools to
handle a wide variety of tasks, from simple workload queuing all the way to
advanced message routing between remote servers.

SSB in SQL Server is a paradigm shift; a change for the better and a new way to
solve a whole host of difficult problems. Yet SSB has not been widely adopted yet.
Whether this is due to time constraints placed on developers who have better things
to do than learn Yet Another New Thing or the perceived difficulty of picking up
the skills required, I am not certain. What I do know is that SSB is incredibly
useful, and really not all that hard to pick up.

Although fairly easy to learn, SSB is still relatively big, and a complex
topic--you really can do quite a few different things with it. So I've broken this
workbench into a three-part series. In this, part one of the workbench, we'll cover
the basics: The Service Broker object types, how to create them, and how to set up
the simplest of queues. At the end of this workbench you'll have a solid foundation
for designing basic asynchronous services--not to mention, for continuing in this
workbench series.

So without further ado, let's get started. The Broker awaits!

*/

--To begin with, we'll create a database to work in.

CREATE DATABASE Simple_Talk_SSB
GO

USE Simple_Talk_SSB
GO


/*
```

-- Enabling Service Broker --

```
Depending on how your model database is set up, Service Broker may be disabled by
default when you create a new database. Or, you may want to enable it in a database
that already exists but in which it is not enabled.

Service Broker is turned on or off on a per-database basis, and you can find out
```

```
its current state for your database by querying the sys.databases catalog view.

Turning on Service Broker is easy enough, but there is a slight twist: there must
either be zero users connected to the database, or you must be the only user
connected, when you actually flip the switch. ALTER DATABASE can help us in that
regard, but it doesn't change the fact that somehow, those users have to go.

*/

--Let's find out if Service Broker is turned on.
SELECT is_broker_enabled
FROM sys.databases
WHERE name = 'Simple_Talk_SSB'
GO



--If the preceding query returned a 0, run the following; note
--that the ROLLBACK IMMEDIATE option will disconnect all other
--users from the database--necessary in order to enable SSB.
--That's OK in this test database, but be careful when you do
--this in a production environment.
ALTER DATABASE Simple_Talk_SSB
SET ENABLE_BROKER
WITH ROLLBACK IMMEDIATE
GO

/*
-- Database Master Key --

Next we will create a database master key. This is not necessarily required, but is
recommended since SSB makes use of many of SQL Server 2005's encryption features.
The rest of this workbench will not require a master key, but I want to stress the
importance of this step anyway, so I'm leaving it in.

*/

CREATE MASTER KEY
ENCRYPTION BY PASSWORD = 'onteuhoeu'
GO



/*
-- Message Types --

Service Broker at its core is a messaging infrastructure, and so the first object
to be defined is the type of messages we'd like to send. Message types are the base
construct which allow your services to enforce what data can and cannot be sent,
and by whom--but we'll get to that second part shortly.

A message type itself is nothing more than a named declaration of what type of data
can be sent. The type of data is known as the data type's "validation." Validation
can take any of the following forms:

                            The message sent will be nothing more than a message,
EMPTY                       with no data; this is useful for pingbacks,
                            acknowledgments, and the like.

NONE                        The message sent can be any binary LOB, up to 2 GB in
                            size.

WELL_FORMED_XML             The message sent must be valid XML.

VALID_XML WITH SCHEMA       The message sent must be XML that can be validated
COLLECTION                  against a specific schema -- This is where the idea of
                            validation really gets powerful.

*/


--For now we'll use a message type with simple XML validation,
--to experiment with.
CREATE MESSAGE TYPE Simple_Msg
VALIDATION = WELL_FORMED_XML
GO


/*
```

```
-- Contracts --

Service Broker's messaging capabilities are based on the idea that messages are
sent between two parties, in a two-way exchange known as a "dialog." The
"initiator" is the partner that started the dialog by sending the first message,
and the "target" is the partner to whom the first message was sent.

Contracts determine which party--the initiator or the target-- can send which
message type(s). Note that aside from the first message, which must be sent from
the initiator to the target, no ordering or workflow is enforced by SSB. Your
contract can define as few or as many message types as you'd like, and you can
specify that they can be sent by the INITIATOR, the TARGET, or by ANY (either of
them).

*/

--Our first contract will be quite simple--either party
--can send the message.
CREATE CONTRACT Simple_Contract
(Simple_Msg SENT BY ANY)
GO

/*

-- Queues --

Queues are the physical storage containers in which messages reside. In a later
workbench we'll cover some of the more advanced options that queues provide, but
for now think of them as as merely special tables.

You can query queues just like any other table, using SELECT, but other standard
DDL operations, such as INSERT, UPDATE, and DELETE, will not work--queued data must
be manipulated using Service Broker DDL.

*/

--Generally speaking, you'll want one queue on each end of
--a dialog. So we'll create two queues: One for the initiator,
--and one for the target.
CREATE QUEUE Simple_Queue_Init
CREATE QUEUE Simple_Queue_Target
GO


--At this point you can use SELECT to take a look at the
--queues, but they are not too exciting. There are no rows
--returned, since we haven't yet sent any messages.
SELECT *
FROM Simple_Queue_Init
GO

/*

-- Services --

Service Broker is designed as a tool that can help you build loosely coupled,
service-oriented applications, and its own design keeps these goals in mind. The
physical manifestation of messages--the queues--are decoupled from the logical
endpoints from which and to which messages are sent. These are known as "services."

A service serves two purposes: First of all, as already mentioned, it is a logical
endpoint that sits on top of a queue. Secondly, and equally important, a service's
definition includes which contract(s) it enforces. Later on, you'll see that the
parties involved in dialogs are services. Since services sit on top of the physical
queues, and since they enforce a defined set of contracts, they are the mechanism
by which you can restrict which message types can eventually show up in the queue.

Since the service and the queue are decoupled, a queue can have many services
feeding it. But that's a topic we'll cover in detail in the third workbench!

*/

--We'll create one service for each queue. The services will
--each enforce the Simple_Contract we defined earlier. A given
--service can enforce as many contracts as you'd like--if you
--want to specify more than one, you can comma-delimit their names.
CREATE SERVICE Simple_Service_Init
ON QUEUE Simple_Queue_Init
(Simple_Contract)

CREATE SERVICE Simple_Service_Target
```

```
ON QUEUE Simple_Queue_Target
(Simple_Contract)
GO


/*
```

-- Dialogs (Conversations) --

```
At long last our infrastructure is in place and we are ready to begin actually
sending some messages. Just to quickly recap: we've created an XML-validated
message type, a contract that allows messages of the type to be sent by either the
initiator or the target, two queues, and two services on top of those two queues,
each of which enforce the contract.

To prepare to send messages, we must first start up a dialog. In its most basic
form, a dialog defines the exchange of messages between two services, and specifies
a contract that will be used to enforce what messages can and cannot be sent by the
involved services.

Once the dialog is created, you will receive a "dialog handle," a GUID that
uniquely identifies the dialog for the initiator--this GUID will be used any time
the initiator needs to send messages to the target or receive messages that have to
do with the dialog.

Note that the Service Broker primitive over which messages are sent is actually
referred to as a "conversation." A dialog is a specific type of conversation, a
two-way exchange between two parties. As of today, it is also the only type of
conversation supported by Service Broker, but given the potential for extensibility
built into the design I suspect we will see monologs, trilogs, and other
conversation types in future versions of SQL Server.

*/


--We'll start a dialog between the two services we've created.
--Notice that the TO SERVICE is specified in quotes? SQL Server
--doesn't necessarily need to know about a service in order
--to start sending messages to it; if TO SERVICE didn't yet
--exist, the messages would simply start to queue. More on this
--topic in the next workbench.

DECLARE @h UNIQUEIDENTIFIER

BEGIN DIALOG CONVERSATION @h
   FROM SERVICE Simple_Service_Init
   TO SERVICE 'Simple_Service_Target'
   ON CONTRACT Simple_Contract
   WITH ENCRYPTION=OFF

--Don't lose this GUID!
SELECT @h
GO

/*
```

-- Sending Messages --

```
We've now told Service Broker that we want to start talking, but that's about it.
It knows what kinds of messages we want to send, and it knows who will be involved,
but we haven't actually communicated yet.

To send a message, use the SEND statement. Note that you send on a conversation
rather than a dialog--this will keep things simple in the future, should other
conversation types be added.

To send a message, you need three things: A dialog handle, which identifies your
side of the dialog, a message type, which tells SSB how or how not to validate what
you send, and of course you also need a message to send.

*/

--Now we'll actually send a message. The GUID we generated in
--the last section is used by the initiator to send messages
--to the target.
DECLARE @h UNIQUEIDENTIFIER
--Insert the GUID from the last section
SET @h = '00EF7BC9-D049-DC11-BE14-001641E42AD0'

--Actually do the send
--Note the semicolon. This is necessary to help SQL Server
--correctly parse this statement.
```

```
;SEND ON CONVERSATION @h
MESSAGE TYPE Simple_Msg
('<Hello_Simple_Talk/>')
GO

/*
```

-- Peeking at the Queue --

Now that a message has been sent, we can presume that it's sitting in the target queue. But there is only one way to find out, and that's to take a look.

As mentioned before, queues can be treated just like tables for the sake of SELECT queries. Each message ships with a large amount of information--all available in the queue--and it's important when working with SSB to know and understand what's there.

So let's pause for just a moment and explore...

```
*/

SELECT *
FROM Simple_Queue_Target

/*
```

Assuming that you've done everything right up to this point, you should be looking at a single row of data. A few interesting points to notice:

1. The conversation_handle column contains a GUID, but you might notice that it's not the same one you used to send the message; the GUID you're looking at is the target's handle.
2. The message_sequence_number is an ascending integer, and for the row you're looking at its value should be 0. As more data is inserted into the queue, that number will be incremented, much like an IDENTITY column for a normal table. Service Broker guarantees in-order, exactly once delivery, and the message_sequence_number is a big part of how that's enforced.
3. The service_name is the name of the service to which the message was sent. Since you're now looking at the queue--not the service--and since a queue can have many services, it can sometimes be important to know where a particular message originated.
4. The message_type_name is all-important, as you start creating complex solutions in which many different types of messages are moving back and forth; you'll want to know what you're dealing with!
5. The message_body is binary, but the validation column has a value of "X" so we know that it's actually just binary- encoded XML.

-- Receiving Messages --

We've taken a glance at the contents of the queue, but haven't actually received the message--it's still there, waiting to be picked up, after which it will be removed from the queue-- exactly once delivery means that the message shouldn't be able to be picked up more than once.

When receiving messages, you want at a minimum three things:

1. The message's type, so that you know what to do with it. The type may be one of the ones defined in the contract used to start the dialog, or it may be one of the system message types--we'll get into that shortly.
2. The message's body, obviously the point of this exercise.
3. The dialog handle, so that you can reply to the message.

The T-SQL RECEIVE statement is much like SELECT, supporting a column list. It also optionally supports the TOP modifier, which we'll use here to get only one message at a time.

A WHERE clause can be used too, but only two columns can be predicated: conversation_handle and conversation_group_id. Only one can be used at a time, and the logical operator used must be the equivalence operator--but that's a topic for a future workbench.

```
*/

--Variables used to hold the handle, type, and body, respectively
DECLARE
    @h UNIQUEIDENTIFIER,
```

```
    @t sysname,
    @b varbinary(MAX)

--Note the semicolon..!
;RECEIVE TOP(1)
    @h = conversation_handle,
    @t = message_type_name,
    @b = message_body
FROM Simple_Queue_Target

--Make sure not to lose the handle!
IF @t = 'Simple_Msg'
BEGIN
    SELECT
        CONVERT(XML, @b),
        @h
END
ELSE
BEGIN
    RAISERROR(
        'I don''t know what to do with this type of message!',
        16, 1)
END
GO

/*
```

-- Sending a Message Back --

Now that you've received a message, you've also gotten something else along with it: A dialog handle on which to send something back. The syntax is the same as before, but now we're going the other way.

```
*/

DECLARE @h UNIQUEIDENTIFIER
--Insert the handle from the preceding section
SET @h = 'E1A1BFCF-D049-DC11-BE14-001641E42AD0'

;SEND ON CONVERSATION @h
MESSAGE TYPE Simple_Msg
('<Goodbye_Simple_Talk/>')
GO

/*
```

-- Ending the Conversation --

We've said goodbye to the initiator, but the dialog is still active--and if we let it, it will sit there, ready and waiting for more messages, until the end of time. Or at least until you drop your database.

As I quickly learned on my first real world SSB project, having a bunch of old conversations sitting around in the database is not a good thing. They occupy space, and by leaving them there a busy database can quickly balloon out of control.

Of course, the designers of SSB foresaw this issue and created a way to get rid of them. This is called ending the conversation.

```
*/

DECLARE @h UNIQUEIDENTIFIER
--Insert the handle from the preceding section
SET @h = 'E1A1BFCF-D049-DC11-BE14-001641E42AD0'

END CONVERSATION @h
GO

/*
```

-- Ending the Conversation, Part 2 --

So now the conversation has been ended by the target. But wait a minute--the initiator didn't get the goodbye message. That's not a good way to end a conversation in person, and neither is it great here. But as it turns out, the conversation is still there, albeit in a slightly disabled state. The target is done and can do nothing else with the conversation. The initiator, on the other hand, can still do a couple of things: it can receive any remaining messages, and it can end the conversation itself.

```
Just like before, we can receive the messages one by one from the queue. But this
time I'm going to modify the code just a bit. The first modification is a loop, to
pull back every message on the queue. There are only two, so this shouldn't take
long.

The second modification is a new condition based on the message type, in order to
handle an EndDialog message. This is a key Service Broker principle. As a
conscientious Service Broker user, any time you (or the services you have created)
receive an end dialog message, it is your duty to react in kind, also ending your
side of the dialog.

*/

--Variables used to hold the handle, type, and body, respectively
DECLARE
    @h UNIQUEIDENTIFIER,
    @t sysname,
    @b varbinary(MAX)

--Get all of the messages on the queue
WHILE 1=1
BEGIN
    SET @h = NULL

    --Note the semicolon..!
    ;RECEIVE TOP(1)
        @h = conversation_handle,
        @t = message_type_name,
        @b = message_body
    FROM Simple_Queue_Init

    IF @h IS NULL
    BEGIN
        BREAK
    END
    ELSE IF @t = 'Simple_Msg'
    BEGIN
        SELECT
            CONVERT(XML, @b),
            @h
    END
    --Need more information? See the following BOL topic:
    --ms-help://MS.SQLCC.v9/MS.SQLSVR.v9.en/sqlmsg9/html/4cc704fa-3ebe-42b7-8ddb-22a7b00e2589.htm
    ELSE IF @t = 'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog'
    BEGIN
        SELECT 'EndDialog message received...'

        END CONVERSATION @h
    END
    ELSE
    BEGIN
        RAISERROR('I don''t know what to do with this type of message!', 16, 1)
    END
END
GO


/*
```

-- Waiting for Godot --

```
We've come a long way, sending specific messages between services on dialogs that
we've later ended. But before we close this workbench there is one final
foundational piece that we must investigate: the enhanced WAITFOR statement that
accompanies Service Broker.

WAITFOR, as you might recall, usually goes hand in hand with either the keywords
DELAY or TIME, and is used to make your process sleep for a certain amount of time
or until a certain time, respectively. It has an entirely new purpose in the
Service Broker world: It can be made to wait for a message to appear on a specific
queue or for a specific dialog. This means that you don't have to poll when you're
working with Service Broker. No more messy loops and wasted resources.

The extended version of WAITFOR uses parenthesis, which contain a RECEIVE
statement. You can also optionally specify a timeout, after which your process will
break and stop waiting if no messages have been received.

Run the following in a new window:

WAITFOR
(
    RECEIVE *
```

```
    FROM Simple_Queue_Target
), TIMEOUT 300000

*/

--Once you've started the WAITFOR, send a new message.
--You'll need a new conversation as well.
DECLARE @h UNIQUEIDENTIFIER

BEGIN DIALOG CONVERSATION @h
FROM SERVICE Simple_Service_Init
TO SERVICE 'Simple_Service_Target'
ON CONTRACT Simple_Contract
WITH ENCRYPTION=OFF

--Actually do the send
;SEND ON CONVERSATION @h
MESSAGE TYPE Simple_Msg
('<Hello_Simple_Talk/>')
GO

--As soon as you hit send, the batch in the other window
--should have finished running, and you should see the
--row from the queue corresponding to the message you
--just sent.



/*

-- Cleanup --

We've made a bit of a mess--and it's always good to clean up when we're done
playing with new technologies. Once you've closed the window you opened to try
WAITFOR, the following will wipe the slate clean.

*/

USE master
GO
DROP DATABASE Simple_Talk_SSB
GO

/*

You've now experienced a small taste of what Service Broker offers, both as a
messaging and queuing technology, but there is a lot more power hidden beneath the
surface.

In the next installment we'll delve deeper, looking at the Service Broker locking
semantics, taking messaging across databases, processing messages automatically
with activation stored procedures, and investigating some of the Service Broker
catalog views.

In part three of this series, we'll look at applying the basics to produce some
interesting solutions including priority queues, blocked callbacks, worker retries,
and batching. In that installment I'll also cover some basic performance techniques
for getting the most out of Service Broker.

*/
```