Service Broker Sample: Asynchronous Triggers

by Eitan Blumin on 4 December 2013 in Blog • 15 Comments

Introduction

Service Broker is a cool feature new since SQL Server 2005. Yes, SQL 2014 is just around the corner so it's hardly "new" any more. But in reality, many DBAs and database users are still not aware of this feature, don't know how to utilize it, or are afraid to do so because they're not familiar with it.

In my previous post in the series, Service Broker Sample: Multi-Threading, I showed a rather advanced scenario where we can implement a multi-threading solution inside the SQL Server database engine. In this post, I hope to show a simpler scenario of using Service Broker in SQL Server.

This time, I'll start by handing out the API script itself and give brief explanation on how to use it. Then, if you're interested, you may continue reading for further explanations.

As before, I won't go into too much detail introducing Service Broker. But Microsoft has an excellent and detailed chapter about Service Broker available right here: http://msdn.microsoft.com/en-us/library/bb522893.aspx

The Short Version: Right to the Scripts

Available below is the download of a zip file containing two scripts.

The first one, "SB_AT_Installation.sql", is the **installation script**, responsible for creating all the objects we require for working with asynchronous triggers: the Service Broker procedures, endpoints and queues. Everything you need in order to implement asynchronous triggers in SQL Server.

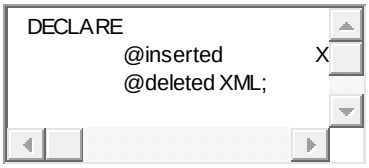
The other script, "SB_AT_Sample.sql", is a **sample script** of how to use this API. For the sake of our example, we'll use the **AdventureWorks2008R2** database which is available for free download here: http://msftdbprodsamples.codeplex.com/releases/view/93587

In that database, we'll actually take an existing trigger **uPurchaseOrderDetail** which is defined on the table **Purchasing.PurchaseOrderDetail**, and convert it into an asynchronous trigger.

API Usage

The installation script provided above is implementing a generic “API” which will allow you to use it for *any table* without any changes to the installed objects. The API will work like this:

1. First, run the installation script on the database where you want the asynchronous triggers.
2. Create a new stored procedure that will implement the logic of your trigger. This procedure must receive two parameters: **@inserted** and **@deleted** – both of type XML. Using XML methods such as nodes(), query() and value(), the procedure will use the data in the @inserted and @deleted parameters as it would use the regular INSERTED and DELETED tables.
3. Create a regular DML trigger on your table, and include the following code inside it:




```
DECLARE
@inserted XML,
@deleted XML;
SELECT @inserted =
( SELECT * FROM inserted FOR XML PATH('row'), ROOT('inserted') );

SELECT @deleted =
( SELECT * FROM deleted FOR XML PATH('row'), ROOT('deleted') );

EXECUTE SB_AT_Fire_Trigger '{YourProcedureName}', @inserted, @deleted;
```

But replace `{YourProcedureName}` with the name of your stored procedure which you created in step 2.

The Long Version: Detailed Explanation

In order to understand what an asynchronous trigger is, we first need to make sure we understand what a normal trigger is.

What is a DML Trigger?

A DML trigger is a programmable routine (like a stored procedure) which is automatically executed when a DML operation (Data Manipulation) is performed on a table (i.e. UPDATE, INSERT, DELETE).

The most important thing that sets triggers aside from stored procedures is the access to special "virtual tables" called **INSERTED** and **DELETED**. When performing an INSERT, the INSERTED table will contain all the new rows that fired the trigger. When performing a DELETE, the DELETED table will contain all the deleted rows. And when performing an UPDATE, the INSERTED table will contain the new values of the updated rows, while the DELETED table will contain all the old values of those same rows.

For further reading, please refer to the Microsoft MSDN resource:
http://technet.microsoft.com/en-us/library/ms178110.aspx

What is an Asynchronous Trigger?

The thing that differentiates asynchronous and normal triggers is that normal triggers will not release the original DML statement (which caused the trigger to fire) before the trigger itself is complete. This means that if the trigger implements some complex logic that takes a while to execute, the users performing the simple DML operations may get surprised when they see that their simple INSERT, UPDATE or DELETE statement takes too long to execute – when in fact it is the trigger that causes the delay.

With asynchronous triggers, on the other hand, the trigger doesn't actually perform the requested work. Instead, the trigger creates a message that contains information about the work to be done and sends this message to a service that performs the work. The trigger then returns.

This way, the program that implements the service (Service Broker in our case) performs the work in a separate transaction. By performing this work in a separate transaction, the original transaction can commit immediately. The application avoids system slowdowns that result from keeping the original transaction open while performing the work.

In this post, I'll provide an example script that implements exactly this kind of logic.

The Design

Since Service Broker only works with binary or XML messages, we'll use XML to send "trigger requests".

General Workflow

The general workflow of our scenario is like this:

1. The user performs an UPDATE operation on a table.
2. An AFTER INSERT trigger on the table is fired. This trigger compiles the contents of the INSERTED and DELETED tables into XML parameters, creates a Service Broker message and sends it to a queue. The original transaction immediately returns.
3. The Service Broker service fires up and processes the messages in the queue independently of the original transaction. It opens up a transaction that will pull the message out of the queue, execute a relevant stored procedure that will use the XML data previously taken from the INSERTED and DELETED tables, and implement the relevant logic.

Understanding the Scripts

The scripts provided for download above are commented to help you understand what's going on. But we'll go over them bit by bit just in case.

The Installation Script

In order to explain the installation script, I'll list all the objects that we're creating and explain the purpose of each object. More detailed information about the implementation can be found in the script itself.

- SB_AT_ServiceBrokerLogs:
  This is a logging table which will be used to log the start and end of each query execution, as well as any errors that may occur.
- SB_AT_HandleQueue:
  This is the activation procedure that Service Broker executes when it creates a "reader instance". In other words, when there's something in the request queue.
  This procedure is the most important part because it's responsible for retrieving messages

from the request queue, executing the relevant stored procedure implementing the trigger logic, and handling any errors that may occur.

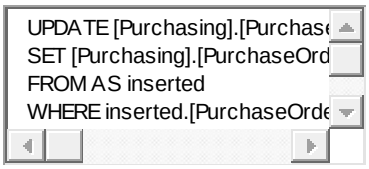
- SB_AT_Request_Queue:
  This is the queue where all trigger requests will be waiting to be processed. Its activation procedure is SB_AT_HandleQueue mentioned above.
- SB_AT_Response_Queue:
  This is the response queue used for responses, but in this sample we won't actually use it.
- [//SB_AT/Message]:
  This is a simple message type of "Well Formed XML". We'll be using it to verify our messages.
- [//SB_AT/Contract]:
  This is a contract defining that both target and initiator must send messages of type [//SB_AT/Message] mentioned above.
- [//SB_AT/ProcessReceivingService]:
  This is the service endpoint working as the "target address" of trigger requests. We will be sending messages into this service. The queue it'll be using to receive messages is SB_AT_Request_Queue mentioned above.
- [//SB_AT/ProcessStartingService]:
  This is the service endpoint working as the "target address" of response messages, or more importantly as the "sender address" since we'll be sending messages out from this service and into [//SB_AT/ProcessReceivingService].
- SB_AT_Fire_Trigger:
  This is the public procedure that will be used for firing an asynchronous trigger, by sending a message from [//SB_AT/ProcessStartingService] to [//SB_AT/ProcessReceivingService].
  You can think of it as an API simplifying our use of all the above objects.
  It receives as parameters the name of the stored procedure which should be executed, the INSERTED table formatted as XML, and the DELETED table formatted as XML (@inserted and @deleted).

The Sample Script

As I mentioned earlier, in this sample we'll take the existing trigger **uPurchaseOrderDetail**, which is defined on the table **Purchasing.PurchaseOrderDetail** in the AdventureWorks2008R2 sample database, and convert it into an asynchronous trigger.

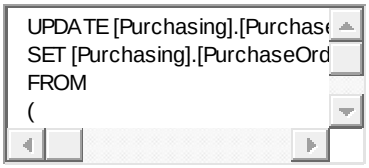
Technically, this means that we'll copy the meat of the trigger's code into a new stored procedure (we'll call it "**Purchasing.usp_AT_uPurchaseOrderDetail**"), and change it in such a way so instead of accessing the INSERTED table, it will access an XML parameter called @inserted and parse it into a relational table.

For example, this code from the original trigger:




```
UPDATE [Purchasing].[PurchaseOrderDetail]
SET [Purchasing].[PurchaseOrderDetail].[ModifiedDate] = GETDATE()
FROM AS inserted
WHERE inserted.[PurchaseOrderID] = [Purchasing].[PurchaseOrderDetail].[PurchaseOrderID]
   AND inserted.[PurchaseOrderDetailID] =
[Purchasing].[PurchaseOrderDetail].[PurchaseOrderDetailID];
```

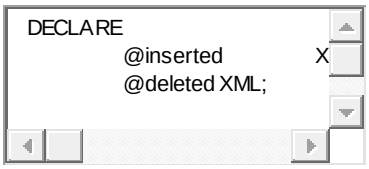
Will be converted to the following code:




```
UPDATE [Purchasing].[PurchaseOrderDetail]
SET [Purchasing].[PurchaseOrderDetail].[ModifiedDate] = GETDATE()
FROM
(
SELECT
  X.query('.').value('(row/PurchaseOrderID)[1]', 'int') AS PurchaseOrderID
, X.query('.').value('(row/PurchaseOrderDetailID)[1]', 'int') AS PurchaseOrderDetailID
FROM @inserted.nodes('inserted/row') AS T(X)
) AS inserted
WHERE inserted.[PurchaseOrderID] = [Purchasing].[PurchaseOrderDetail].[PurchaseOrderID]
   AND inserted.[PurchaseOrderDetailID] =
[Purchasing].[PurchaseOrderDetail].[PurchaseOrderDetailID];
```

Note the usage of the XML methods nodes(), query() and value() which are used for parsing the XML into relational form (click on any of them for more info).

The actual trigger of the table will contain only the following code:




```
DECLARE
@inserted XML,
@deleted XML;
```

```
4
5   SELECT @inserted =
6   ( SELECT * FROM inserted FOR XML PATH('row'), ROOT('inserted') );
7
8   SELECT @deleted =
9   ( SELECT * FROM deleted FOR XML PATH('row'), ROOT('deleted') );
10
11 EXECUTE SB_AT_Fire_Trigger 'Purchasing.usp_AT_uPurchaseOrderDetail', @inserted,
   @deleted;
```

Note how we format the INSERTED and DELETED tables as XML and save their contents in XML variables. Then we pass them on to the stored procedure **SB_AT_Fire_Trigger** together with the name of the stored procedure implementing the actual trigger logic **(Purchasing.usp_AT_uPurchaseOrderDetail).**

Important Notes

Some points I'd like to stress that are important to remember:

- The stored procedure **SB_AT_Fire_Trigger** receives as its first parameter the name of another stored procedure which is supposed to implement the actual trigger logic. Using this method you can use SB_AT_Fire_Trigger for just about any table. Just create a "trigger procedure" and pass it on as the first parameter.
- The first two parameters of your "trigger procedure" (Purchasing.usp_AT_uPurchaseOrderDetail in our sample above) must be **@inserted and @deleted XML parameters**. The procedure should be using these parameters instead of the native INSERTED and DELETED tables. The reason for this is of course that we're not inside the trigger's scope anymore. We're in the scope of a regular stored procedure, and therefore the INSERTED and DELETED tables are not available.
- Since we are not in the context of the trigger anymore, it's also important to stress that **executing the ROLLBACK command will not rollback the DML operation** that fired the trigger. If a ROLLBACK command is integral to the logic of your trigger, you will need to implement it yourself by utilizing the primary key columns of the table (which I hope you have), and joining with the data in the @inserted and/or @deleted parameters.
    - To rollback a DELETE, you can use the data in the @deleted parameter to re-insert the deleted data.
    - To rollback an INSERT, you can use the data in the @inserted parameter to delete the inserted data.
    - To rollback an UPDATE, you can use the @deleted parameter to regain the old data. WARNING: Don't forget to take into account that in order to rollback an UPDATE you must perform another UPDATE, so obviously it will re-fire the trigger again. Take precautions as to not cause an infinite loop of triggers!

Conclusion

This sample should be much simpler than the first one (implementing multi-threading), and I hope it'll give you a better idea on how you can use Service Broker to your advantage in SQL Server. And if not that, then I hope that at least now you have one more copy-paste trick up your sleeve to help you implement asynchronous triggers easily.

http://www.madeiradata.com/service-broker-asynchronous-triggers/