Scheduling Jobs in SQL Server Express

By **Mladen Prajdić** on **27 August 2008** | 18 Comments | Tags: Backup/Restore, Service Broker

Article Series Navigation:

As we all know SQL Server 2005 Express is a very powerful free edition of SQL Server 2005. However it does not contain SQL Server Agent service. Because of this scheduling jobs is not possible. So if we want to do this we have to install a free or commercial 3rd party product. This usually isn't allowed due to the security policies of many hosting companies and thus presents a problem. Maybe we want to schedule daily backups, database reindexing, statistics updating, etc. This is why I wanted to have a solution based only on SQL Server 2005 Express and not dependent on the hosting company. And of course there is one based on our old friend the Service Broker.

New terminology

To achieve scheduling we will use **SQL Server Service Broker**. If you're not familiar with this great addition to the storage engine go read my **previous** **three** **articles** about it. There you'll get familiarized with the terminology and database objects used in this article. Done? OK, let's move on.

So you're familiar with services, queues, activation procedures, messages, contracts, conversations, etc... The new member we have to take a look at is the

Conversation Timer:

```
BEGIN CONVERSATION TIMER ( conversation_handle )
   TIMEOUT = timeoutInSeconds
[ ; ]
```

When the conversation timer is set it waits the number of seconds specified in the timeout and then it sends the **http://schemas.microsoft.com/SQL/ServiceBroker/DialogTimer** message to the local queue that is the part of a conversation. It never sends the message to the remote queue. As the DialogTimer message comes to the queue, the activation stored procedure associated with the queue fires, receives the message from the queue and executes whatever logic we have programmed it to.

Don't mistake the conversation timer for the conversation lifetime! Each part of the conversation can have a different conversation timer set while the conversation lifetime is the time from the beginning to the end of the conversation.

How it works

Let's see how this scheduling infrastructure is built from start in simple bullet points:

1. Create the needed tables for our scheduled jobs information

2. Create the needed stored procedures that handle scheduled jobs

3. Create the needed contract, queue and service

1. Needed tables

We need two tables:

- **ScheduledJobs** stores information about our scheduled jobs

- **ScheduledJobsErrors** stores possible errors when manipulating scheduled jobs

```
CREATE TABLE ScheduledJobs
(
    ID INT IDENTITY(1,1),
    ScheduledSql nvarchar(max) NOT NULL,
    FirstRunOn datetime NOT NULL,
    LastRunOn datetime,
    LastRunOK BIT NOT NULL DEFAULT (0),
    IsRepeatable BIT NOT NULL DEFAULT (0),
    IsEnabled BIT NOT NULL DEFAULT (0),
    ConversationHandle uniqueidentifier NULL

)

CREATE TABLE ScheduledJobsErrors
(
    Id BIGINT IDENTITY(1, 1) PRIMARY KEY,
    ErrorLine INT,
    ErrorNumber INT,
    ErrorMessage NVARCHAR(MAX),
    ErrorSeverity INT,
    ErrorState INT,
    ScheduledJobId INT,
    ErrorDate DATETIME NOT NULL DEFAULT GETUTCDATE()
)
```

2. Needed stored procedures

For our simple scheduling we need three stored procedures. Only the pieces of code are shown here so look at the accompanying script for full code.

First two expose the scheduling functionality we use. The third one isn't supposed to be used directly but it can be if it is needed.

- **usp_AddScheduledJob** adds a row for our job to the ScheduledJobs table, starts a new conversation on it and set a timer on it. Adding and conversation starting is done in a transaction since we want this to be an atomic operation.

```
INSERT INTO ScheduledJobs(ScheduledSql, FirstRunOn, IsRepeatable,
ConversationHandle)
VALUES (@ScheduledSql, @FirstRunOn, @IsRepeatable, NULL)

SELECT @ScheduledJobId = SCOPE_IDENTITY()

...

BEGIN DIALOG CONVERSATION @ConversationHandle
```

```
    FROM SERVICE   [//ScheduledJobService]

    TO SERVICE      '//ScheduledJobService',
                    'CURRENT DATABASE'
    ON CONTRACT     [//ScheduledJobContract]

    WITH ENCRYPTION = OFF;

BEGIN CONVERSATION TIMER (@ConversationHandle)
TIMEOUT = @TimeoutInSeconds;
```

- usp_RemoveScheduledJob performs cleanup. It accepts the id of the scheduled job we wish to remove. It ends the conversation that the inputted scheduled job lives on, and it deletes the row from the ScheduledJobs table. Removing the job and ending the conversation is also done in a transaction as an atomic operation.

```
IF EXISTS (SELECT *
        FROM sys.conversation_endpoints
        WHERE conversation_handle = @ConversationHandle)
    END CONVERSATION @ConversationHandle

DELETE ScheduledJobs WHERE Id = @ScheduledJobId
```

- usp_RunScheduledJob is the activation stored procedure on the queue and it receives the dialog timer messages put there by our conversation timer from the queue. Depending on the IsRepeatable setting it either sets the daily interval or ends the conversation. After that it runs our scheduled job and updates the ScheduledJobs table with the status of the finished scheduled job. This stored procedure isn't transactional since any errors are stored in the error table and we don't want to return the DialogTimer message back to the queue, which would cause problems with looping and **poison messages** which we'd have to again handle separately. We want to keep things simple.

```
RECEIVE TOP(1)
        @ConversationHandle = conversation_handle,
        @message_type_name = message_type_name
FROM ScheduledJobQueue

...


SELECT  @ScheduledJobId = ID,
       @ScheduledSql = ScheduledSql,
       @IsRepeatable = IsRepeatable

FROM    ScheduledJobs
WHERE   ConversationHandle = @ConversationHandle AND IsEnabled = 1

...

-- run our job
EXEC (@ScheduledSql)
```

3. Needed Service Broker objects

For everything to work we need to make a simple setup used by the Service Broker:

- [//ScheduledJobContract] is the contract that allows only sending of the "http://schemas.microsoft.com/SQL/ServiceBroker/DialogTimer" message type.

- ScheduledJobQueue is the queue we use to post our DialogTimer messages to and run the usp_RunScheduledJob activation procedure that runs the scheduled job.

- [//ScheduledJobService] is a service set on top of the ScheduledJobQueue and bound by the [//ScheduledJobContract] contract.

```
CREATE CONTRACT [//ScheduledJobContract]

    ([http://schemas.microsoft.com/SQL/ServiceBroker/DialogTimer] SENT BY
INITIATOR)

CREATE QUEUE ScheduledJobQueue
    WITH STATUS = ON,
    ACTIVATION (
        PROCEDURE_NAME = usp_RunScheduledJob,
        MAX_QUEUE_READERS = 20, -- we expect max 20 jobs to start simultaneously

        EXECUTE AS 'dbo' );

CREATE SERVICE [//ScheduledJobService]
    AUTHORIZATION dbo
    ON QUEUE ScheduledJobQueue ([//ScheduledJobContract])
```

4.Tying it all together

Now that we have created all our objects let's see how they all work together.

First we have to have a valid SQL statement that we'll run as a scheduled job either daily or only once. We can add it to or remove it from the ScheduldJobs table by using our usp_AddScheduledJob stored procedure. This procedure starts a new conversation and links it to our scheduled job. After that it sets the conversation timer to elapse at the date and time we want our job to run.

At this point we have our scheduled job lying nicely in a table and a timer that will run it at our time. When the scheduled time comes the dialog timer fires and service broker puts a DialogTimer message into the ScheduledJobQueue. The queue has an activation stored procedure usp_RunScheduledJob associated with it which runs every time a new message arrives to the queue.

This activation stored procedure then receives our DialogTimer message from the queue, uses the conversation handle that comes with the message and looks up the job associated with that conversation handle. If our job is a run only once type it ends the conversation else it resets the timer to fire again in 24 hours. After that it runs our job. When the job finishes (either succeeds or fails) the status is written back to the ScheduledJobs table. And that's it.

We can also manually remove the job at any time with the usp_RemoveScheduledJob stored procedure that ends the conversation and its timer from our job and then deletes a row from the ScheduledJobs table.

The whole infrastructure is quite simple and low maintenance.

5. How to Schedule Jobs - Example

Here is an example with three scheduled jobs: a daily backup job of our test database, a faulty script and a one time update of statistics. All are run 30 seconds after you add them with the usp_AddScheduledJob stored procedure.

```
GO
DECLARE @ScheduledSql nvarchar(max), @RunOn datetime, @IsRepeatable BIT

SELECT  @ScheduledSql = N'DECLARE @backupTime DATETIME, @backupFile
NVARCHAR(512);
    SELECT @backupTime = GETDATE(),
           @backupFile = ''C:\TestScheduledJobs_'' +
                replace(replace(CONVERT(NVARCHAR(25), @backupTime, 120),
                '' '', ''_''), '':'', ''_'') + N''.bak'';
           BACKUP DATABASE TestScheduledJobs TO DISK = @backupFile;',
        @RunOn = dateadd(s, 30, getdate()),
        @IsRepeatable = 0

EXEC usp_AddScheduledJob @ScheduledSql, @RunOn, @IsRepeatable
GO

DECLARE @ScheduledSql nvarchar(max), @RunOn datetime, @IsRepeatable BIT

SELECT  @ScheduledSql = N'select 1, where 1=1',
        @RunOn = dateadd(s, 30, getdate()),
        @IsRepeatable = 1

EXEC usp_AddScheduledJob @ScheduledSql, @RunOn, @IsRepeatable
GO

DECLARE @ScheduledSql nvarchar(max), @RunOn datetime, @IsRepeatable BIT

SELECT  @ScheduledSql = N'EXEC sp_updatestats;',
        @RunOn = dateadd(s, 30, getdate()),
        @IsRepeatable = 0

EXEC usp_AddScheduledJob @ScheduledSql, @RunOn, @IsRepeatable
GO
```

6. Monitoring

We can monitor our scheduled jobs' conversations currently being processed and their success by running these queries:

```
-- show the currently active conversations.

-- Look at dialog_timer column (in UTC time) to see when will the job be run next
SELECT * FROM sys.conversation_endpoints
-- shows the number of currently executing activation procedures
SELECT * FROM sys.dm_broker_activated_tasks

-- see how many unreceived messages are still in the queue.
-- should be 0 when no jobs are running
SELECT * FROM ScheduledJobQueue with (nolock)

-- view our scheduled jobs' statuses
SELECT * FROM ScheduledJobs  with (nolock)
-- view any scheduled jobs errors that might have happend

SELECT * FROM ScheduledJobsErrors  with (nolock)
```

Conclusion

Here we've seen how to build a simple job scheduler that can schedule jobs once or daily at the time **specified in the database we run this in**. The complete code can be found in this file.

Scheduling Jobs in SQL Server Express - Part 2

By **Mladen Prajdić** on **1 December 2008** | 27 Comments | Tags: Administration, Backup/Restore, Service Broker

Article Series Navigation:

In my previous article Scheduling Jobs in SQL Server Express we saw how to make simple job scheduling in SQL Server 2005 Express work. We limited the scheduling to one time or daily repeats. Sometimes this isn't enough. In this article we'll take a look at how to make a scheduling solution based on Service Broker worthy of the SQL Server Agent itself.

We will try to imitate scheduled jobs provided by SQL Server Agent like making a scheduled job made up of multiple job steps. Every job will have it's own custom job schedule similar to the one you can set up with SQL Server Agent. We'll also be able to add and remove scheduled jobs and job schedules and start and stop a scheduled job.

Note: All scheduled job dates HAVE TO BE handled in UTC time since conversation timer works only in UTC date format.

Security

In part 1 we were limited to making jobs run only in our database. Because of that we couldn't backup any other databases. Since that is a big drawback for a complete scheduling solution we need a fix for this. This happens because the activation stored procedure always runs under the EXECUTE AS security context which is more restrictive than normal db_owner context we use. This is well explained in the Books Online article Extending Database Impersonation by Using EXECUTE AS. There are two ways to go about this.

The first one which is also used in this article is to set our scheduling database to trustworthy:

```
ALTER DATABASE TestScheduledJobs SET TRUSTWORTHY ON
```

Setting the database to trustworthy gives the members of the db_owner role sysadmin privileges. So be careful with this setting.

The second option is to sign the activation stored procedure with a certificate. How to do this is excellently explained by Remus Rusanu, a former developer of Service Broker on his blog here. This is extremely useful if you need finer access control in your database.

Solving the security issue allows us to have one database for all scheduling which is what a proper scheduling engine should have.

Required Tables

We need 4 tables for this to work well. You can add more tables such as job history but you'll have to modify the stored procedures to use them.

ScheduledJobs Table

This holds information about our scheduled jobs such as job name, enabled status, etc...

```
CREATE TABLE ScheduledJobs
(
    ID INT IDENTITY(1,1),
    JobScheduleId INT NOT NULL,
    ConversationHandle UNIQUEIDENTIFIER NULL,
    JobName NVARCHAR(256) NOT NULL DEFAULT (''),
    ValidFrom DATETIME NOT NULL,
    LastRunOn DATETIME,
    NextRunOn DATETIME,
    IsEnabled BIT NOT NULL DEFAULT (0),
    CreatedOn DATETIME NOT NULL DEFAULT GETUTCDATE()
)
```

ScheduledJobSteps Table

This holds the job step name, the SQL statement to run in the step, whether to retry the step on failure and how many times, step duration, etc...

```
CREATE TABLE ScheduledJobSteps
(
    ID INT IDENTITY(1,1),
    ScheduledJobId INT NOT NULL,
    StepName NVARCHAR(256) NOT NULL DEFAULT (''),
    SqlToRun NVARCHAR(MAX) NOT NULL, -- sql statement to run
    RetryOnFail BIT NOT NULL DEFAULT (0), -- do we wish to retry the job step on
failure
    RetryOnFailTimes INT NOT NULL DEFAULT (0), -- if we do how many times do we
wish to retry it
    DurationInSeconds DECIMAL(14,4) DEFAULT (0), -- duration of the step with all
retries
    CreatedOn DATETIME NOT NULL DEFAULT GETUTCDATE(),
    LastExecutedOn DATETIME
)
```

JobSchedules Table

This holds the job's schedule. Multiple jobs can have the same schedule. Here we specify an absolute or relative scheduling frequency. This is mimicking most of the SQL Server Agent scheduling options. For weekly scheduling number 1 is Monday and number 7 is Sunday. This is because the **ISO standard** says that a week starts on Monday.

```
CREATE TABLE JobSchedules
(
    ID INT IDENTITY(1, 1) PRIMARY KEY,
    FrequencyType INT NOT NULL CHECK (FrequencyType IN (1, 2, 3)),
        -- daily = 1, weekly = 2, monthly = 3. "Run once" jobs don't have a job
schedule
    Frequency INT NOT NULL DEFAULT(1) CHECK (Frequency BETWEEN 1 AND 100),
    AbsoluteSubFrequency VARCHAR(100),
```

```
        -- '' if daily, '1,2,3,4,5,6,7' day of week if weekly,
'1,2,3,...,28,29,30,31' if montly
    MontlyRelativeSubFrequencyWhich INT,
    MontlyRelativeSubFrequencyWhat INT,
    RunAtInSecondsFromMidnight INT NOT NULL DEFAULT(0)
        CHECK (RunAtInSecondsFromMidnight BETWEEN 0 AND 84599), -- 0-84599 = 1
day in seconds
    CONSTRAINT CK_AbsoluteSubFrequency CHECK
        ((FrequencyType = 1 AND ISNULL(AbsoluteSubFrequency, '') = '') OR --
daily check
         (FrequencyType = 2 AND LEN(AbsoluteSubFrequency) > 0) OR -- weekly check
(days of week CSV)
         (FrequencyType = 3 AND (LEN(AbsoluteSubFrequency) > 0 -- monthly
absolute option (days of month CSV)
             AND MontlyRelativeSubFrequencyWhich IS NULL
             AND MontlyRelativeSubFrequencyWhat IS NULL)
             OR ISNULL(AbsoluteSubFrequency, '') = '') -- monthly relative option
                ),
    CONSTRAINT MontlyRelativeSubFrequencyWhich CHECK -- only allow values if
frequency type is monthly
         (MontlyRelativeSubFrequencyWhich IS NULL OR
           (FrequencyType = 3 AND
            AbsoluteSubFrequency IS NULL AND
            MontlyRelativeSubFrequencyWhich IN (1,2,3,4,5)) -- 1st-4th, 5=Last
                                                ),
    CONSTRAINT MontlyRelativeSubFrequencyWhat CHECK  -- only allow values if
frequency type is monthly
      (MontlyRelativeSubFrequencyWhich IS NULL OR
       (FrequencyType = 3 AND
        AbsoluteSubFrequency IS NULL AND
        MontlyRelativeSubFrequencyWhich IN (1,2,3,4,5,6,7,-1)) -- 1=Mon to 7=Sun,
-1=Day
                                                )
)
```

For monthly relative scheduling you can easily set the first/second/third/fourth/last day of the week or of the month.

SchedulingErrors Table

This table contains information about any errors that have happened in our job execution. Once a job errors out it will stop without any further execution.

```
CREATE TABLE SchedulingErrors
(
    Id INT IDENTITY(1, 1) PRIMARY KEY,
    ScheduledJobId INT,
    ScheduledJobStepId INT,
    ErrorLine INT,
    ErrorNumber INT,
    ErrorMessage NVARCHAR(MAX),
    ErrorSeverity INT,
    ErrorState INT,
    ErrorDate DATETIME NOT NULL DEFAULT GETUTCDATE()
)
```

Required User-Defined Functions

dbo.GetNextRunTime

This calculates the next time our job will be run based on the schedule we created and the last run time or last ValidFrom time of the job. In the function we use an excellent calendar table F_TABLE_DATE and a user defined function called dbo.F_ISO_WEEK_OF_YEAR that gets the ISO Week number for a date. Both were created by Michael Valentine Jones, a regular on the SQLTeam.com forums. The GetNextRunTime function is quite complex but the comments in code should provide enough information for complete understanding. If they don't, ask questions in the comments.

The following code shows the part of the UDF that calculates the next date to run for the daily scheduling type:

```
-- ...
-- DAILY SCHEDULE TYPE
IF @FrequencyType = 1
BEGIN
    SELECT TOP 1 @NextRunTime = DATE
    FROM (
            SELECT  DATEADD(s, @RunAtInSecondsFromMidnight, DATE)
                AS DATE, ROW_NUMBER() OVER(ORDER BY DATE) - 1 AS
CorrectDaySelector
            FROM    dbo.F_TABLE_DATE(@LastRunTime, DATEADD(d, 2*@Frequency,
@LastRunTime))
         ) t
    WHERE   DATE > @LastRunTime
            AND CorrectDaySelector % @Frequency = 0
    ORDER BY DATE
END
-- ...
```

Required Stored Procedures

These stored procedures make a nice and friendly interface to the scheduling functionality.

- **usp_AddJobSchedule:** Adds a new job schedule.
- **usp_RemoveJobSchedule:** Removes an existing job schedule.
- **usp_AddScheduledJob:** Adds a new scheduled job. For "Run once" job types we don't need a Job Schedule so for @JobScheduleId parameter we have to pass -1 and for @NextRunOn we have to set a date in the future in UTC time.
- **usp_RemoveScheduledJob:** Removes an existing scheduled job.
- **usp_AddScheduledJobStep:** Adds a new scheduled job step for a job. It also calculates the next run time of the job.
- **usp_RemoveScheduledJobStep:** Removes an existing scheduled job step from a job.
- **usp_StartScheduledJob:** Used when starting a new or job or re-enabling an old disabled job by passing a new ValidFrom date and in the activation stored procedure to start the job anew for the next scheduled run.
- **usp_StopScheduledJob:** Stops the scheduled job run by ending the conversation for it and setting it to disabled.
- **usp_RunScheduledJobSteps:** Runs every job step and repeats it the set number of times if that option is enabled. After each successful job step execution the step duration time and last run on time is set.

- **usp_RunScheduledJob:** Activation stored procedure that is **NOT** meant to be run by hand. You should run it by hand only for debugging, when your messages are left in the queue. The stored procedure receives the dialog timer message from the queue and finds the scheduled job that corresponds with its conversation handle. After it gets the Scheduled job it runs its Job steps, sets the dialog timer for the next scheduled value and updates the last run time of the job.

Scheduling Code Examples

```
DECLARE @JobScheduleId INT,
        @ScheduledJobId INT,
        @validFrom DATETIME,
        @ScheduledJobStepId INT,
        @secondsOffset INT,
        @NextRunOn DATETIME

SELECT    @validFrom = GETUTCDATE(), -- the job is valid from current UTC time
         -- run the job 2 minutes after the validFrom time.
         -- we need the offset in seconds from midnight of that day for all jobs
        @secondsOffset = 28800, -- set the job time time to 8 in the morning of
the selected day
        @NextRunOn = DATEADD(n, 1, @validFrom) -- set next run for once only job
to 1 minute from now

-- SIMPLE RUN ONCE SCHEDULING EXAMPLE
-- add new "run once" scheduled job
EXEC usp_AddScheduledJob @ScheduledJobId OUT, -1, 'test job', @validFrom,
@NextRunOn
-- add just one simple step for our job
EXEC usp_AddScheduledJobStep @ScheduledJobStepId OUT, @ScheduledJobId, 'EXEC
sp_updatestats', 'step 1'
-- start the scheduled job
EXEC usp_StartScheduledJob @ScheduledJobId

-- SIMPLE DAILY SCHEDULING EXAMPLE
-- run the job daily
EXEC usp_AddJobSchedule @JobScheduleId OUT,
                        @RunAtInSecondsFromMidnight = @secondsOffset,
                        @FrequencyType = 1,
                        @Frequency = 1 -- run every day
-- add new scheduled job
EXEC usp_AddScheduledJob @ScheduledJobId OUT, @JobScheduleId, 'test job',
@validFrom
DECLARE @backupSQL NVARCHAR(MAX)
SELECT  @backupSQL = N'DECLARE @backupTime DATETIME, @backupFile NVARCHAR(512);
    SELECT @backupTime = GETDATE(),
    @backupFile = ''C:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\Backup\AdventureWorks_'' +
         replace(replace(CONVERT(NVARCHAR(25), @backupTime, 120), '' '', ''_''),
'':'', ''_'') +
            N''.bak'';
    BACKUP DATABASE AdventureWorks TO DISK = @backupFile;'

EXEC usp_AddScheduledJobStep @ScheduledJobStepId OUT, @ScheduledJobId,
@backupSQL, 'step 1'
-- start the scheduled job
EXEC usp_StartScheduledJob @ScheduledJobId

-- COMPLEX WEEKLY ABSOLUTE SCHEDULING EXAMPLE
```

```
-- run the job on every tuesday, wednesday, friday and sunday of every second
week
EXEC usp_AddJobSchedule @JobScheduleId OUT,
                        @RunAtInSecondsFromMidnight = @secondsOffset,
                        @FrequencyType = 2, -- weekly frequency type
                        @Frequency = 2, -- run every every 2 weeks,
                        @AbsoluteSubFrequency = '2,3,5,7'
                         -- run every Tuesday(2), Wednesday(3), Friday(5) and
Sunday(7)
-- add new scheduled job
EXEC usp_AddScheduledJob @ScheduledJobId OUT, @JobScheduleId, 'test job',
@validFrom
-- add three steps for our job
EXEC usp_AddScheduledJobStep @ScheduledJobStepId OUT, @ScheduledJobId, 'EXEC
sp_updatestats', 'step 1'
EXEC usp_AddScheduledJobStep @ScheduledJobStepId OUT, @ScheduledJobId, 'DBCC
CHECKDB', 'step 2'
EXEC usp_AddScheduledJobStep @ScheduledJobStepId OUT, @ScheduledJobId,
  'select 1,', 'step 3 will fail', 1, 2 -- retry on fail 2 times
-- start the scheduled job
EXEC usp_StartScheduledJob @ScheduledJobId

-- COMPLEX RELATIVE SCHEDULING SCHEDULING EXAMPLE
DECLARE @relativeWhichDay INT, @relativeWhatDay INT
SELECT    @relativeWhichDay = 4, -- 1 = First, 2 = Second, 3 = Third, 4 = Fourth,
5 = Last
        @relativeWhatDay = 3 -- 1 = Monday, 2 = Tuesday, ..., 7 = Sunday, -1 =
Day
-- run the job on the 4th monday of every month
EXEC usp_AddJobSchedule @JobScheduleId OUT,
                        @RunAtInSecondsFromMidnight = @secondsOffset, -- int
                        @FrequencyType = 3, -- monthly frequency type
                        @Frequency = 1, -- run every month,
                        @AbsoluteSubFrequency = NULL, -- no aboslute frequence if
relative is set
                        @MontlyRelativeSubFrequencyWhich = @relativeWhichDay,
                        @MontlyRelativeSubFrequencyWhat = @relativeWhatDay
/*
some more relative monthly scheduling examples
run on:
the first day of the month:
  - @MontlyRelativeSubFrequencyWhich = 1, @MontlyRelativeSubFrequencyWhat = -1
the third thursday of the month:
  - @MontlyRelativeSubFrequencyWhich = 3, @MontlyRelativeSubFrequencyWhat = 4
the last sunday of the month:
  - @MontlyRelativeSubFrequencyWhich = 5, @MontlyRelativeSubFrequencyWhat = 7
the second wedensday of the month:
  - @MontlyRelativeSubFrequencyWhich = 2, @MontlyRelativeSubFrequencyWhat = 3
*/
-- add new scheduled job
EXEC usp_AddScheduledJob @ScheduledJobId OUT, @JobScheduleId, 'test job',
@validFrom
-- add just one simple step for our job
EXEC usp_AddScheduledJobStep @ScheduledJobStepId OUT, @ScheduledJobId, 'EXEC
sp_updatestats', 'step 1'
-- start the scheduled job
EXEC usp_StartScheduledJob @ScheduledJobId

-- SEE WHAT GOING ON WITH OUR JOBS
-- show the currently active conversations
-- look at dialog_timer column (in UTC time) to see when will the job be run next
```

```
SELECT GETUTCDATE(), dialog_timer, * FROM sys.conversation_endpoints
-- shows the number of currently executing activation procedures
SELECT * FROM sys.dm_broker_activated_tasks
-- see how many unreceived messages are still in the queue. should be 0 when no
jobs are running
SELECT * FROM ScheduledJobQueue WITH (NOLOCK)
-- view our scheduled jobs' statuses
SELECT * FROM ScheduledJobs  WITH (NOLOCK)
SELECT * FROM ScheduledJobSteps WITH (NOLOCK)
SELECT * FROM JobSchedules  WITH (NOLOCK)
SELECT * FROM SchedulingErrors WITH (NOLOCK)
```

Conclusion

When we look at this solution we can see it's a pretty powerful scheduling engine. If needed it can be extended for hourly or minute-level scheduling but that is an exercise for the reader. The script might look overly complex but once you have it set up it's use is fast and easy. The complete code can be found in this file.

Centralized Asynchronous Auditing with Service Broker

By **Mladen Prajdić** on **16 July 2007** | 4 Comments | Tags: Administration, Service Broker

Article Series Navigation:

Service Broker is a new feature in SQL Server 2005. It is an integrated part of the database engine and it provides queuing and reliable direct asynchronous messaging between SQL Server 2005 instances only. In the future this is planned to be extended to non-SQL Server instances. This article shows how to use Service Broker and triggers to capture data changes.

How Service Broker works

What Service Broker does is it talks or converses with other service brokers. It does that by exchanging messages in a dialog conversation between two service brokers. Imagine two people talking to each other. The words they exchange are messages and their conversation is a dialog. To fully understand the basics we must become familiar with the terminology and what does what. I will show only basic commands. More info can be found in SQL Server Help better known as Books Online.

Commands are explained in the order of needed object creation for Service Broker conversations.

The basis of everything is a **message type**. A message type defines the validation of the XML message that will be performed.

The general syntax is:

```
CREATE MESSAGE TYPE message_type_name
 [ AUTHORIZATION owner_name ]
 [ VALIDATION = {
NONE | EMPTY | WELL_FORMED_XML |
VALID_XML WITH SCHEMA COLLECTION
schema_collection_name
} ]
```

This is how to create a simple message type with validation that conforms to well formed XML:

```
CREATE MESSAGE TYPE [//Audit/Message] VALIDATION = WELL_FORMED_XML
```

Next step is to create a message type **contract**. This **contract** defines which message types are allowed in a conversation. For example, if we take our two people talking from before, a contract means that they are only allowed to talk about sports. Anything else is rejected by both persons as garbage.

The general syntax is:

```
CREATE CONTRACT contract_name
 [ AUTHORIZATION owner_name ]
 ( { { message_type_name | [ DEFAULT ] }
 SENT BY { INITIATOR | TARGET | ANY }
} [ ,...n] )
```

In the SENT BY part we specify which message is allowed to be sent from each point of the conversation. So we can specify that the initiator can talk about sports and women, while the target can only talk about sports.

This is how to create a simple contract type with previously created message type that can be only sent by the initiator:

```
CREATE CONTRACT [//Audit/Contract] ([//Audit/Message] SENT BY INITIATOR)
```

Next comes a **queue.** A **queue** holds every message received by each point in the conversation. Each point of conversation has it's own queue in which the received messages are waiting for processing

The general syntax is:

```
CREATE QUEUE [ database_name. [ schema_name ] . | schema_name. ] queue_name
```

This is how to create a simple queue which takes:

- a stored procedure name to execute when a new message arrives in the queue
- maximum number of concurrently running stored procedures (for very busy queues)
- and user under whose context the procedure will be run

```
CREATE QUEUE dbo.TargetAuditQueue
 WITH STATUS=ON,
 ACTIVATION (
 PROCEDURE_NAME = usp_WriteAuditData, -- sproc to run when the queue receives a
message
 MAX_QUEUE_READERS = 50, -- max concurrently executing instances of sproc
 EXECUTE AS 'dbo' );
```

Every queue is associated with a **service.** A service exposes the functionality of contracts associated with the service to other contracts. It defines which message types the associated queue will receive. Other types are rejected. If no contract is specified then that service can only be an initiator of the conversation so we have to specify which queue it will use and which contrats are allowed.

The general syntax is:

```
CREATE SERVICE service_name
 [ AUTHORIZATION owner_name ]
 ON QUEUE [ schema_name. ]queue_name
 [ ( contract_name | [DEFAULT] [ ,...n ] ) ]
```

This is how to create a simple service with previously created contract on our previously created queue with dbo authorization:

```
CREATE SERVICE [//Audit/DataWriter]
 AUTHORIZATION dbo
 ON QUEUE dbo.TargetAuditQueue ([//Audit/Contract])
```

Building the Centralized Asynchronous Auditing System

Now that we're familiar with the basics of Service Broker we can go on with building our auditing system. Auditing is the means of tracking changes of your data. It provides you with a log of who did what when. In the US and EU it is also required by law for sensitive data.

Auditing is usually done by inserting changed data into the accompanying audit table in the trigger of the source table. Or it can be a part of the Update, Delete and Insert stored procedures in which case we don't need triggers. Simplest way to implement auditing is for every table to have another audit table with the same structure. The downside is that these audit tables grow very fast. This increases the database size and backup/restore times which is a negative side effect. With a lot of tables this becomes cumbersome an hard to maintain. Imagine the work you have to do if you have 10 databases with each having 40 tables. That's 10 x 40 x 2 = 800 tables you have to create. And those are small databases table-wise.

Another option is to use third-party Log Readers but they aren't fun, are they? :)

I went about it differently. I wanted to have only one database that will hold all my audited data from every database I have on the server. This way my other databases would be free of bloated audited data. If i wanted to query the audited data i ccould simply select from one table that holds everything. I also wouldn't want this scenario to impact my performance. How to implement this? Service Broker to the rescue.

Service Broker's reliable asynchronous messaging was the perfect solution. I still used triggers but the technique can be easily used in stored procedures and OUTPUT clause. I created two databases each with its own Service Broker. The auditing database is called **MasterAuditDatabase** and the database to be audited is called **TestDb1**. Both databases have SET TRUSTWORTHY ON which enables us to use cross-database service broker communication without the use of certificates. Each database has its own error table that holds errors that happen in Service Broker communication. Yes, they may happen :)

When an insert, update or delete happens our audit trigger simply takes appropriate data from the inserted and deleted pseudo tables, turns it into our well formed XML message and uses Service Broker to send it to another Service Broker in the Master Audit Database which then saves it to our auditing table. Every time a new message arrives to our target queue, a stored procedure is executed that inserts the queued message into the audit table.

For busy systems only one stored procedure couldn't possibly cope with all incoming messages. That is why we set the MAX_QUEUE_READERS = 50. To allow 50 concurrent (parallel) queue readers. Because this is a completely asynchronous operation there's no impact on performance on the initiator end. The triggers return immediately.

The code can be downloaded and is well documented so I hope it speaks for itself. The code is broken into two scripts.

- The Master Audit Database (the database that holds all audited information) and its Service Broker infrastructure
- Sample audited database and its Service Broker infrastructure

Note: You will need to copy the GUID returned by the first script and use it in the second script.

This method of auditing proved to be very good with no noticeable impact on performance. Of course the master audit database grows fast and horizontal partitioning will be surely needed. It can also be used with multiple servers where one server serves as keeper of audited data from other servers. To do this we'd have to create TCP/IP endpoints and create users, certificates and other security measures which I will show how to do in the next article.

Conclusion

Service Broker is a great addition to SQL Server and it's use has just barely scratched the surface. The future is leaning to service oriented architecture (SOA) and loosly coupled applications. With the coming of .Net 3.5 and adoption of Windows Communication and Workflow Foundations, Service Broker will fit in nicely. So learn it.