

CHAPTER 20



Bits and Pieces

Whew! We've certainly traveled a few roads together. We've taken a very basic—and poor—database structure and transformed it into a robust, well-defined database that could be used as the basis of any contact-based system you care to develop. Now that the database is out of the way, there are a few other things SQL Server offers worth taking a look at. This will be a bit brief, but you can use the overview as a springboard to find out about the things that sound interesting. Okay, let's roll!

Security

A lot of the things we've done throughout this book have required administrator-level permissions—creating a database, for example. When you first installed SQL Server, you added yourself as an administrator. This meant you were added to a security group called *sysadmins*. This is a server security group—anybody who is a member of the *sysadmins* group can perform absolutely any action on the server. This is usually the norm in a development environment, but it certainly is not normal in any other environment! Most of the time, you will have pretty limited permissions, so it's good to know how permissions are broken down.

GRANT, DENY, and REVOKE

It is possible to grant permissions on individual objects and commands using the T-SQL statement `GRANT`. This statement would grant `SELECT` permissions on the `Contacts` table to a user called `Dolly`:

```
GRANT SELECT ON dbo.Contacts TO Dolly;
```

There are two other commands: `DENY` and `REVOKE`. `DENY` temporarily removes permissions, `REVOKE` removes them for good.

Built-in Security Roles

To make things easier to manage, SQL Server provides two sets of roles. Users assigned to these roles will be able to perform a certain set of tasks. The first set of roles deals with server permissions, and the second deals with database rights. Users can be members of multiple roles, although some roles incorporate the permissions of others.

Fixed Server Roles

It is possible to create your own server roles; the nine roles provided as part of SQL Server are known as *Fixed Server Roles*.

- **sysadmin:** This is what we have been using throughout the book. A member of this group can perform any action on the SQL Server.
- **serveradmin:** Members of this group can shut down the server and modify the server configuration.
- **securityadmin:** This group is used to manage logins and associated permissions at the server level. If a securityadmin user has access to a database they can also manage database permissions. Users in this group can use the `GRANT`, `DENY`, and `REVOKE` statements we saw earlier.
- **processadmin:** Do you remember [Chapter 15](#), our transactions chapter, when we blocked our `SELECT` statement? Members of this group can stop processes, so if you think you need this ability, this is the role to ask for.
- **setupadmin:** Not a commonly used role. It is possible to link SQL Server to other servers (this includes SQL Servers, Oracle, and other database systems). This role lets you create these linked servers using T-SQL. Strangely, you have to be a member of the sysadmin group to create a linked server using SSMS.
- **bulkadmin:** If you are a member of this group, you can execute the `BULK INSERT` statement. You can do anything else.



- **diskadmin:** Do you remember when we were discussing database files way back in the early chapters? We talked about the possibility of splitting your database up across multiple files. If you want to do this, you've come to the right group!
- **dbcreator:** Despite its name, this role does not limit its users to just creating databases—they can also alter and drop databases, and restore them via backups.
- **public:** All SQL Server logins (logins created using SQL Server security) belong to the public role by default. This is a kind of catchall role. If a user is attempting to access a particular object and no specific permissions have been granted on that object, the permissions the public role has for that object are inherited by the user. You should be careful with this role, and only assign permissions on objects you want everybody to have access to.

Fixed Database Roles

The Fixed Server Roles let you manage server permissions; the Fixed Database Roles manage individual permissions in individual databases. You may be a member of *db_owner* in one database, and a member of *db_datareader* in another. If you wish, you can create your own database roles—these are known as *flexible database roles*.

There are nine Fixed Database Roles:

- **db_owner:** Equivalent to *sysadmin*, but at the database level. If you are in this group you can do anything within the database, even drop it.
- **db_securityadmin:** Members can manage database permissions, including role membership.
- **db_accessadmin:** Controls who can add or remove access to the database, for all types of user.
- **db_backupoperator:** Used to perform database backups.
- **db_ddladmin:** If you are in this role, you can run any Data Definition Language (DDL) command. These are the commands that create, alter, or drop objects, like `CREATE TABLE`.
- **db_datawriter:** Gives access to the data in all tables of the database. You can insert, update, and delete data if you are in this role.
- **db_datareader:** Allows data in any table to be viewed.
- **db_denydatawriter:** Members of this role cannot modify data in any way.
- **db_denydatareader:** You guessed it—if you are in this role you won't be able to read any data.

You can grant permissions on individual items, which will override the role's permissions. If explicit `SELECT` permissions are granted on the `Contacts` table, it wouldn't matter that you were a member of *db_denydatareader*. You would be able to view the data in `Contacts`, but not in any other table.

Schemas

Schemas can be used as a further security mechanism, but they are also really useful for logically grouping database objects. We've mentioned schemas a couple of times throughout this book, so it's worth taking a quick look at them now.

Let's say we've been asked to create some reporting stored procedures in our **AddressBook** database. We've also been told there will be more reporting requests coming our way. We could just write the stored procedure like this:

```
CREATE PROCEDURE dbo.SelectAllContacts
```

This would put the procedure in the default `dbo` schema. There is no way of knowing if this procedure is used for reporting purposes. We could change its name:

```
CREATE PROCEDURE dbo.ReportSelectAllContacts
```

but this is a bit unwieldy, and relies on developers to follow a convention. The best solution is to create a schema called `Reporting`, and to then create the procedure within that schema. Schemas can be created via



SSMS or T-SQL. To use SSMS, you expand the database name in Object Explorer, then **Security** ➤ **Schemas**. Right-click **Schemas** and choose the **New Schema** option, shown in Figure 20-1.

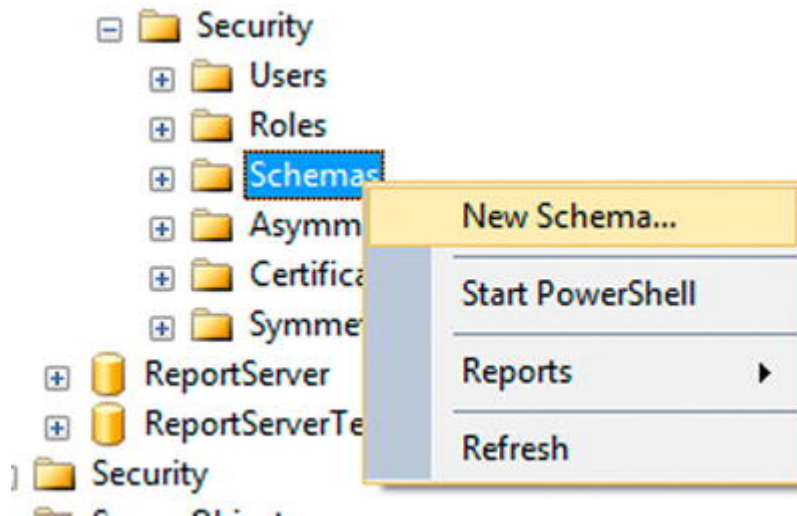


Figure 20-1. Creating a new schema in SSMS

To use T-SQL, you simply run this statement:

```
CREATE SCHEMA Reporting;
```

This will create a schema called `Reporting`. You then create the procedure within that schema by specifying the schema name.

```
CREATE PROCEDURE Reporting.SelectAllContacts
```

When you view the stored procedure in SSMS, you'll see the schema name appears in front of the stored procedure name (Figure 20-2).

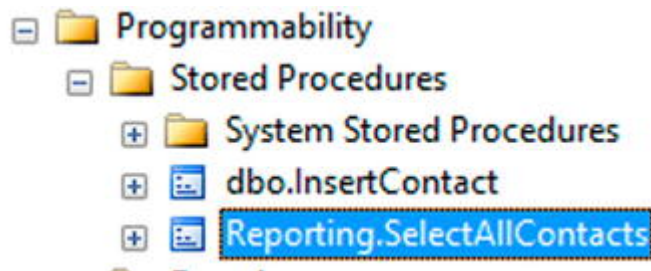


Figure 20-2. A stored procedure created within a schema

All the way through this book, we've added checks to our scripts to see if an object exists, and if it does we drop it before re-creating it. If we tried to run this:

```
IF EXISTS(SELECT 1 FROM sys.procedures WHERE [name] = 'SelectAllContacts')
```

or this:

```
IF EXISTS(SELECT 1 FROM sys.procedures WHERE [name] = 'Reporting.SelectAllContacts')
```

The checks would fail. The first check would expect the procedure to exist in `dbo`, and this would be the only schema it would attempt to check. The second will fail because the name of the object is `SelectAllContacts`, not `Reporting.SelectAllContacts`. `SelectAllContacts` exists within the `Reporting` schema. To correctly detect the object, we need to use a join.



```
IF EXISTS(SELECT 1 FROM sys.procedures SP
INNER JOIN sys.schemas SC
ON SP.schema_id = SC.schema_id
WHERE SP.[name] = 'SelectAllContacts' AND SC.[name] = 'Reporting')
```

You can join any of the `sys` tables we've used for these checks to `sys.schemas` using this technique. Don't forget to check for both the object name and the schema name!

Triggers

Many developers I meet hate triggers with a passion. This is wrong. The trigger is much maligned (unfairly in my opinion). There are times when nothing but a trigger will do. A *trigger* is a piece of code that runs after some action has happened on a table (you can also apply triggers to views, DDL statements, and logon attempts, among others). The triggers that fire on tables are the most commonly used and are known as DML triggers.

A DML trigger is fired whenever an `INSERT`, `UPDATE`, or `DELETE` statement is executed against a table. You can instruct the trigger to perform pretty much any action you wish, even to the point of preventing the action that the original statement executed from happening (e.g., you could run an `INSERT` and have the trigger remove that `INSERT`).

Triggers can be great when you need the values of some columns to be automatically calculated, and this cannot be done with a default. Imagine you needed to record a start and end date against each row in a table, to determine when that particular row was active. Whenever a new row is inserted, the following actions must occur:

- The end date of the previously active row must be set to the current date and time
- The start date of the new row must be set to the current date and time
- The end date of the new row must be set to a particular date in the future

If you implement this using a stored procedure, it will work—but every single insert must be processed by the stored procedure. If anybody inserts a row without using the stored procedure, the dates will go out of sync. If a trigger is implemented, the dates will always be calculated, as the trigger can find the previous row, update it, and update the new row, too.

Triggers are a huge subject in their own right and would justify at least one chapter of a book. If you think you need to use triggers in your database, I point you to the `CREATE TRIGGER` documentation at SQL Server Books Online: <https://msdn.microsoft.com/en-GB/library/ms189799.aspx>.

Profiler and Extended Events

Sometimes, you need to be able to trace what your SQL Server is doing. A particular `SELECT` statement may be taking a while to run and you'd like to obtain the statement so you can walk through it and figure out what is going on. Or you may want to see why a particular statement is being blocked. There are two tools that can provide answers to these questions as well as many more. The old way of doing things is to use SQL Server Profiler, which has been with the product for many years. The new way of tracing is to use Extended Events. We'll take a quick look at both options.

SQL Server Profiler

Profiler is a tool provided with SQL Server that allows you to inspect what is happening inside your server in real time. It is a fantastically powerful utility that gives granular access to certain events. A full description of everything Profiler does would require its own book, so we're just going to introduce you to the tool here. More details can be found at <https://msdn.microsoft.com/en-GB/library/ms181091.aspx>.

Microsoft has announced that Profiler will be deprecated in a future version of SQL Server for everything but SQL Server Analysis Services. As a result you should use Extended Events for all new development work, which offers more facilities than Profiler and is built directly into SSMS. I'm showing Profiler here for completeness.

Starting Profiler

To open Profiler, type **SQL Server 2014 Profiler** into your Start Menu or Start Screen. Once it opens, you be confronted with the rather dull gray screen shown in **Figure 20-3**.



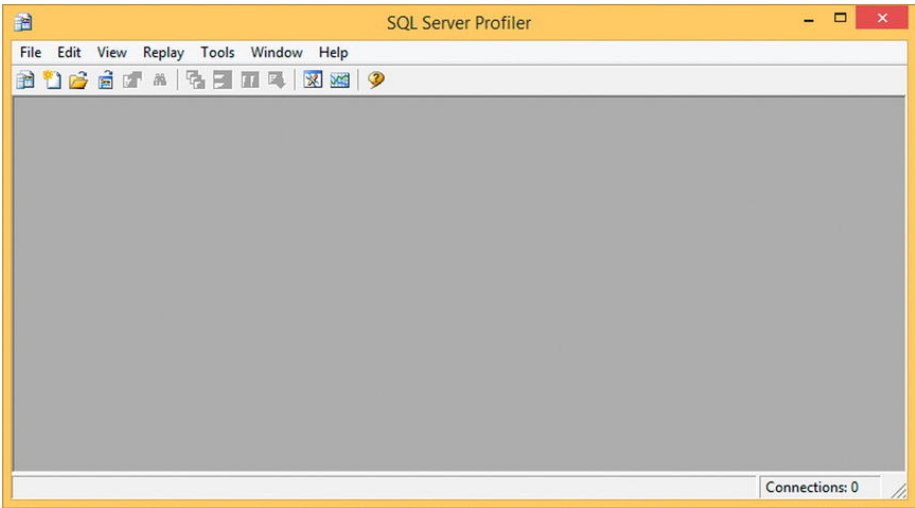


Figure 20-3. The SQL Server Profiler tool

To start tracing events in SQL Server, go to the **File** menu and select the **New Trace** option. You'll be prompted to connect to SQL Server—do this, and the **Trace Properties** window will appear (you can see this in Figure 20-4).

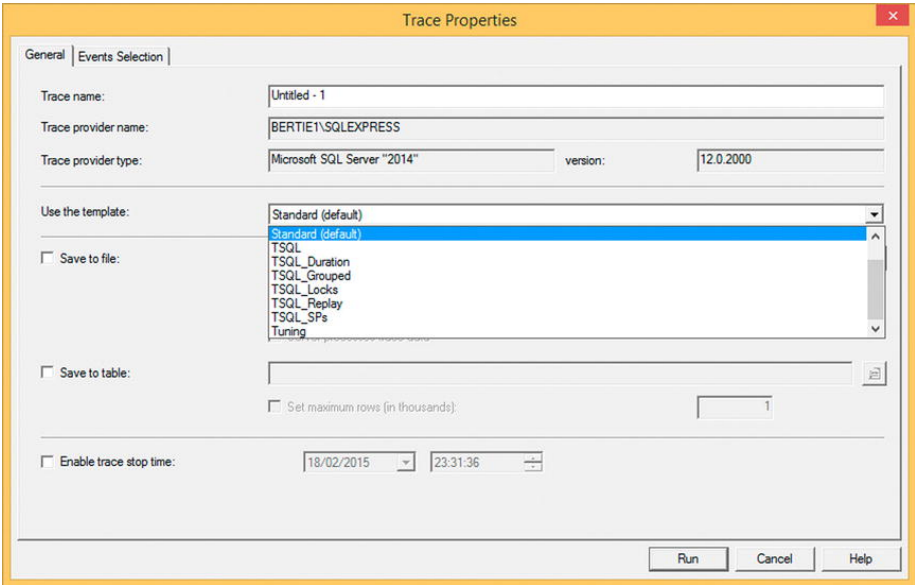


Figure 20-4. Creating a new trace from a standard template

TRACE PERMISSIONS

You require a certain set of permissions to be able to run Profiler. You must be a sysadmin or have ALTER TRACE permissions. You should already be a sysadmin, so Profiler should work for you without any issues.

You'll see you can give the trace a name, among other values. The most interesting item on this screen is the **Use the template** drop-down list. This contains the trace templates: prebuilt templates supplied with SQL Server. **TSQL_Locks**, for instance, allows you to monitor for objects that are being locked, while **TSQL_SPs** captures stored procedures as they execute and process. You can create your own templates should you wish.

Select **TSQL_SPs** and then move over to the **Events Selection** tab. Profiler can capture certain events as part of a trace, such as the details issued when the SP starts and the details issued when the SP completes. There are many types of event; to see them all check the **Show all events** box. Each template only captures certain subset of events.



For our purposes, ensure only the events below **Stored Procedures** have a check mark next to them (just like in [Figure 20-5](#)). It is possible to easily overwhelm yourself with information returned from Profiler, so you need to think carefully about the events you want to capture.

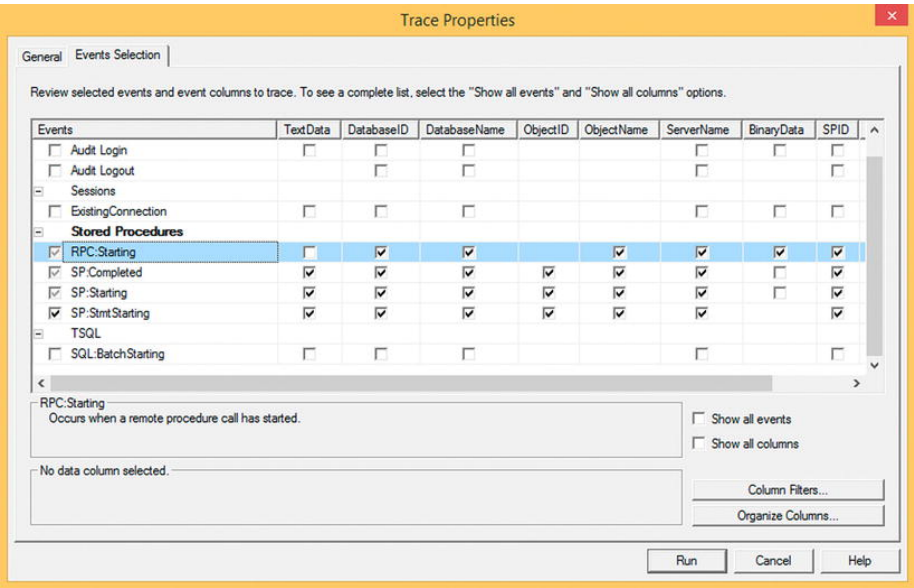


Figure 20-5. The trace events selection screen

You can set filters to limit the data you capture. By default, a trace will capture *everything that happens on your server*. This means every event fired by every database on the server will be captured. It is not likely you'll want to do this, as normally you are trying to capture events for a particular database. To add a filter, click the **Column Filters** button. The filters dialog appears (shown in [Figure 20-6](#)), presenting a number of items you can filter on.

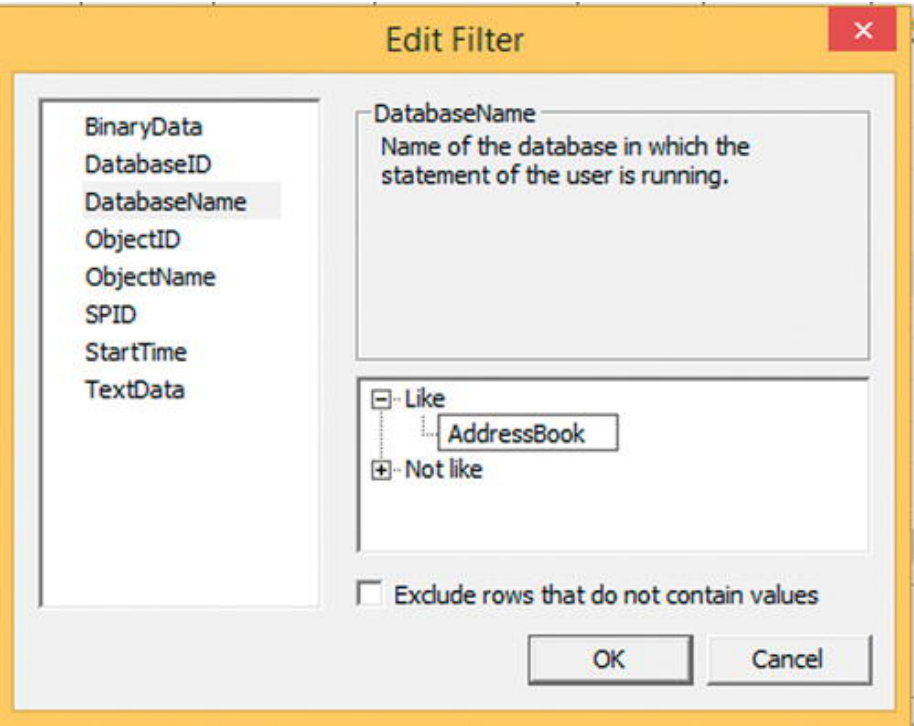


Figure 20-6. Adding filters to a trace

We'll add a filter on **DatabaseName**. Click this item, and then expand **Like** in the box in the bottom right-hand corner. You can then type a value—we'll enter **AddressBook**. You can specify multiple filters; pressing



Enter after typing **AddressBook** would present you with another data entry box.

Click **OK** to clear this dialog, and then click **Run** to start the trace. Nothing particularly exciting happens—just a message showing **Trace Start**, which you can see in **Figure 20-7**.

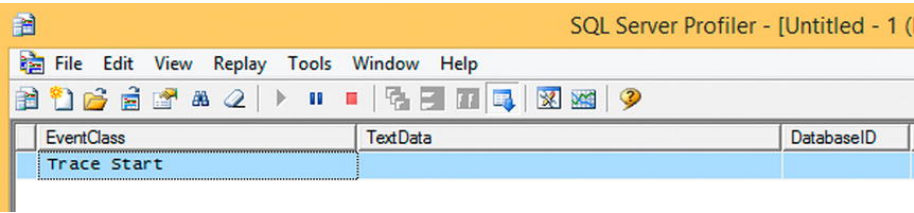


Figure 20-7. A trace that just started

To make things interesting, open up SSMS and execute this script:

```
USE AddressBook

DECLARE @ContactNotes CONTACTNOTE;
DECLARE @ContactIdOUT INT;

INSERT INTO @ContactNotes(ContactId, Note) VALUES (NULL, 'Dolly is a rather cool little dog. She ha

EXEC dbo.InsertContact
@FirstName = 'Dolly',
@LastName = 'McQuillan',
@DateOfBirth = '2001-06-30',
@Notes = @ContactNotes,
@ContactId = @ContactIdOUT;
```

After executing, return to the Profiler window, which should now be showing some events (your output may differ from **Figure 20-8**, but the important thing is you can see some rows).

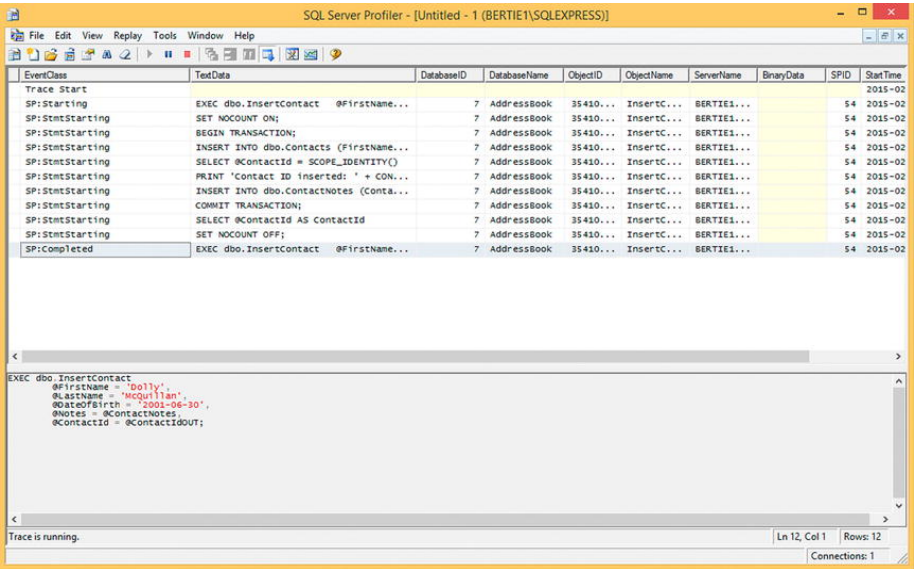


Figure 20-8. A running trace with captured events

At this point, the trace is still running. Click the red stop square on the toolbar to stop the trace. A **Trace Stop** item will appear as the last trace item.

Profiler has captured everything our stored procedure did. There is a **SP:Starting** event first, which displays the full command passed to SQL Server. This is great if you need to execute the procedure manually with the same parameter values to find an error. Next are a bunch of **SP:StmtStarting** events, one for each statement.



in the stored procedure. The right-most column shows the **StartTime** for each statement, so if one statement in particular took a while to execute you would be able to pinpoint on which line the problem was occurring.

The final event is **SP:Completed**, which again displays the full statement executed, and informs you that the procedure has successfully completed its work.

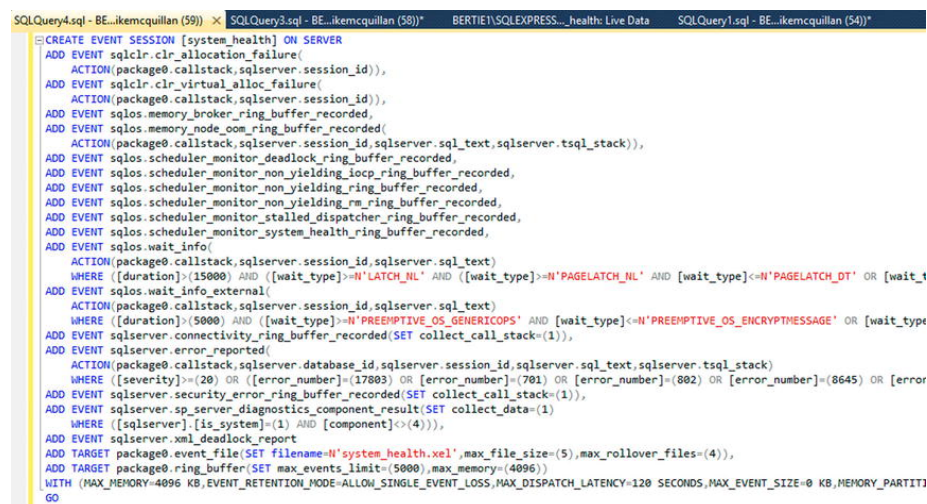
There is obviously a whole lot more to Profiler than has been shown here, but now that you know it's there you can explore!

Extended Events

I mentioned that Profiler has been deprecated, and will be removed from SQL Server in some future version. Its replacement, Extended Events, was introduced in SQL Server 2008. Extended Events is more powerful than Profiler, as it is embedded deeper into the SQL Server product. This allows it to capture more granular information than Profiler.

To create an Extended Events session, expand the **Management** node in Object Explorer, then **Extended Events**. A **Sessions** node will exist below **Extended Events**—if you expand this, you'll see two default sessions have been created. **AlwaysOn_health** is disabled (it has a small red arrow next to it) unless you have turned the session on. This is intended for use with high-availability configurations, so it is unlikely to be enabled.

The other session, **system_health**, is active, and captures various server events for things like memory usage and errors. If you right-click this session and choose **Script Session as ► CREATE To ► New Query Editor Window**, the script generated will show you exactly which events are being captured (Figure 20-9).



```

CREATE EVENT SESSION [system_health] ON SERVER
ADD EVENT sqlclr.clr_allocation_failure(
    ACTION(package0.callstack,sqlserver.session_id)),
ADD EVENT sqlclr.clr_virtual_alloc_failure(
    ACTION(package0.callstack,sqlserver.session_id)),
ADD EVENT sqls.memory_broker_ring_buffer_recorded,
ADD EVENT sqls.memory_node_oom_ring_buffer_recorded(
    ACTION(package0.callstack,sqlserver.session_id,sqlserver.sql_text,sqlserver.tsq_stack)),
ADD EVENT sqls.scheduler_monitor_deadlock_ring_buffer_recorded,
ADD EVENT sqls.scheduler_monitor_non_yielding_io_cp_ring_buffer_recorded,
ADD EVENT sqls.scheduler_monitor_non_yielding_ring_buffer_recorded,
ADD EVENT sqls.scheduler_monitor_non_yielding_re_ring_buffer_recorded,
ADD EVENT sqls.scheduler_monitor_stalled_dispatcher_ring_buffer_recorded,
ADD EVENT sqls.scheduler_monitor_system_health_ring_buffer_recorded,
ADD EVENT sqls.wait_info(
    ACTION(package0.callstack,sqlserver.session_id,sqlserver.sql_text)
    WHERE ([duration]>(15000) AND ([wait_type]>=N'LATCH_NL' AND ([wait_type]<=N'PAGE_LATCH_NL' AND [wait_type]<=N'PAGE_LATCH_DT' OR [wait_t
ADD EVENT sqls.wait_info_external(
    ACTION(package0.callstack,sqlserver.session_id,sqlserver.sql_text)
    WHERE ([duration]>(5000) AND ([wait_type]>=N'PREEMPTIVE_OS_GENERICOPS' AND [wait_type]<=N'PREEMPTIVE_OS_ENCRYPTMESSAGE' OR [wait_type
ADD EVENT sqlserver.connectivity_ring_buffer_recorded(SET collect_call_stack=(1)),
ADD EVENT sqlserver.error_reported(
    ACTION(package0.callstack,sqlserver.database_id,sqlserver.session_id,sqlserver.sql_text,sqlserver.tsq_stack)
    WHERE ([severity]>=(20) OR ([error_number]=17803) OR [error_number]=701) OR [error_number]=802) OR [error_number]=8645) OR [error
ADD EVENT sqlserver.security_error_ring_buffer_recorded(SET collect_call_stack=(1)),
ADD EVENT sqlserver.sp_server_diagnostics_component_result(SET collect_data=(1)
    WHERE ([sqlserver].[is_system]=1) AND [component]<>(4))),
ADD EVENT sqlserver.xml_deadlock_report
ADD TARGET package0.event_file(SET filename=N'system_health.xel',max_file_size=(5),max_rollover_files=(4)),
ADD TARGET package0.ring_buffer(SET max_events_limit=(5000),max_memory=(4096))
WITH (MAX_MEMORY=4096 KB,EVENT_RETENTION_MODE=ALLOW_SINGLE_EVENT_LOSS,MAX_DISPATCH_LATENCY=120 SECONDS,MAX_EVENT_SIZE=0 KB,MEMORY_PARTITION
GO
  
```

Figure 20-9. A generated Extended Events script

We'll create a simple session to record information about login requests to our SQL Server. To begin, right-click **Sessions** and choose **New Session Wizard**. Click **Next** on the introduction page and you'll be asked to provide a name for the session (Figure 20-10). Call it **Capture_Logins** and click **Next**. You don't need to check the **Start the event session at server startup** box, as we're just fooling around. You may want to utilize this option when you are creating a real session, though.



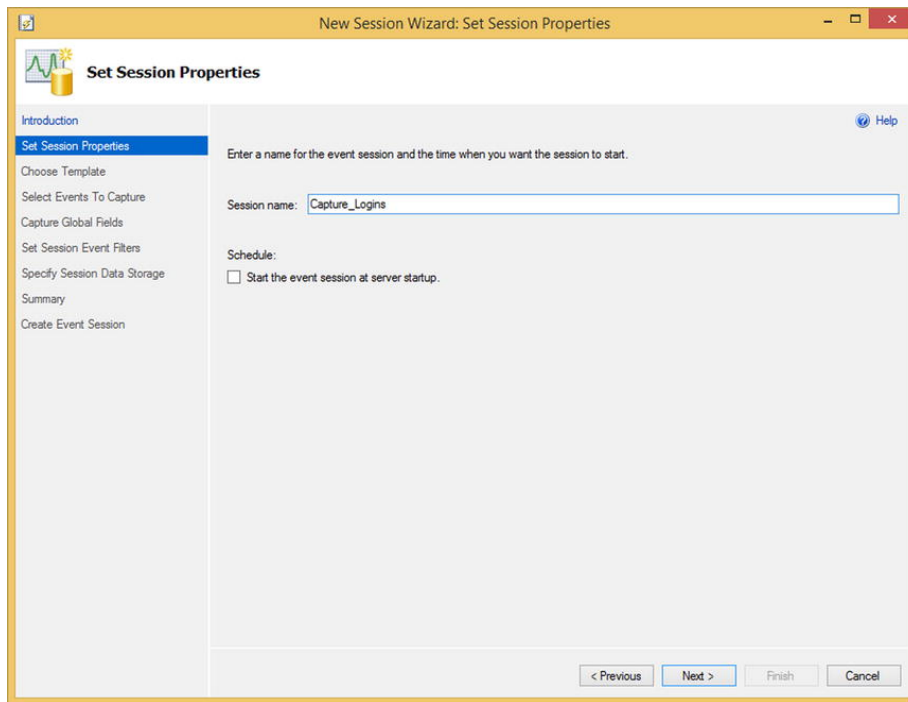


Figure 20-10. Naming a session in the Extended Events wizard

Click **Next** and you'll be asked if you want to use a session template. **Do not use a template** is selected by default. These templates work in a similar way to the templates provided with Profiler, in that they provide you with a prebuilt structure to work with. Click the **Use this event session template** option and select **System Monitoring** ➤ **Connection Tracking**. This captures any logins to the server and provides us with a ready-made session template. A brief description of what the template gives you is displayed (**Figure 20-11**).

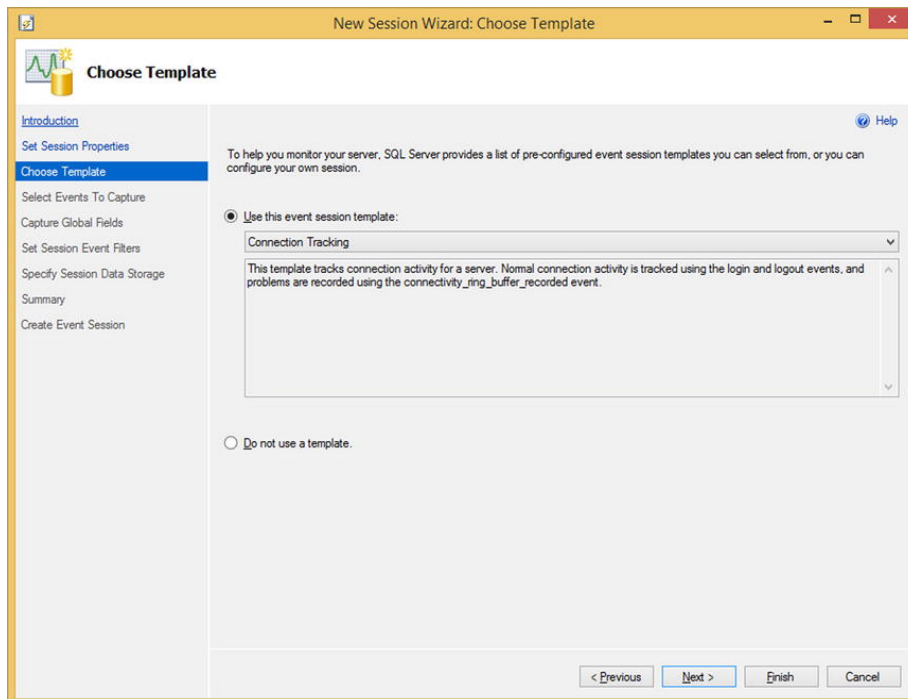


Figure 20-11. Choosing an Extended Events session template

Clicking **Next** displays the Events screen. There are many events you can capture, but because we selected a prebuilt template, three events have already been selected for us: **connectivity_ring_buffer_recorded**, **login**, and **logout** (shown on the right in **Figure 20-12**). This will be fine for our needs, so click **Next** again.



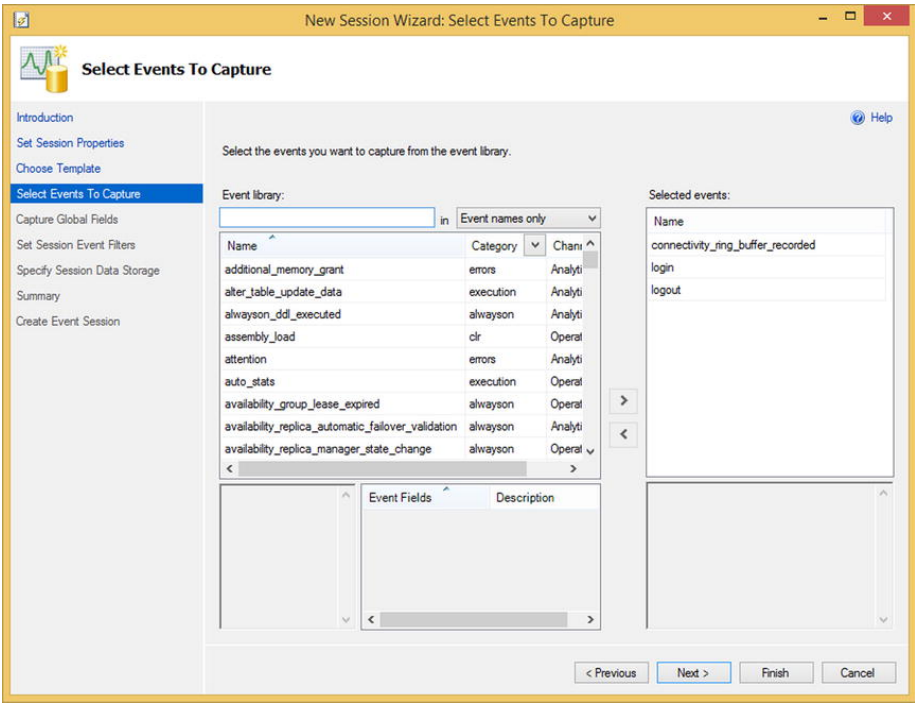


Figure 20-12. Adding events you want to capture

Next up is **Capture Global Fields**, which determines the columns shown in the trace. Again, some of these fields are already checked via our template selection. The default selection is fine, but if you are curious feel free to add any fields you see fit—the list is shown in Figure 20-13. Click **Next** once you are done.

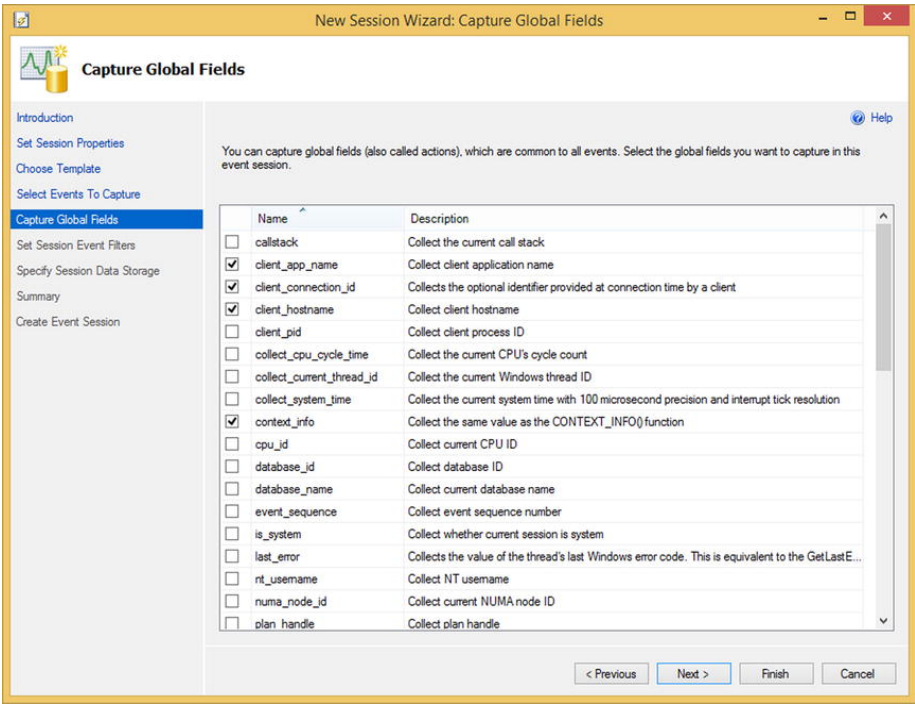


Figure 20-13. Choosing global fields for an event session

Now we come to the **Set Session Event Filters** page, shown in Figure 20-14. If we were capturing queries, we'd state that we want our Extended Events session only to capture events from our **AddressBook** database. For logins, we may wish only to capture login attempts for a particular account. You can add as many filters as you like.



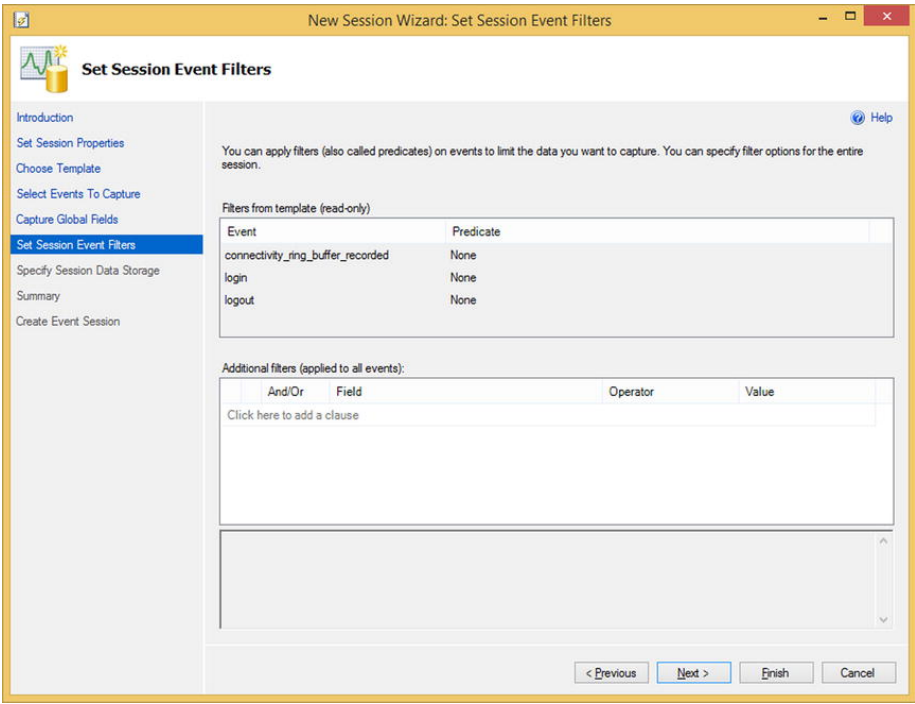


Figure 20-14. Adding event filters

For this demonstration, we don't need any filters, so click **Next**.

At this point, we are nearly done. Clicking **Next** shows the last selection screen of the wizard (Figure 20-15), where we specify how to store the session's data. We can write the session output to a file for future analysis, or we can choose to work with the most recent data (the default). The most recent data are in the ring buffer, which is why that event was included as part of the template. Leave the default values selected. This makes the last 1,000 events available for analysis (you can make this infinite by specifying a value of 0).

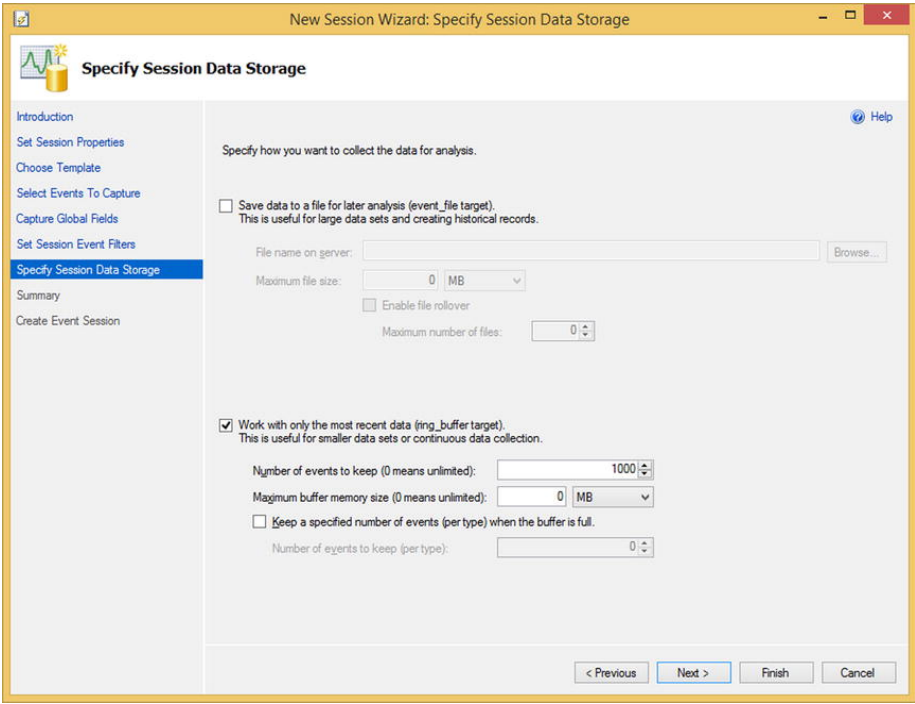


Figure 20-15. Choosing the number of events to store



Clicking **Next** now takes you to the summary page. You can inspect your selections if you wish. There's an interesting button at the bottom of this page named **Script**. Clicking on this generates a script for the session

in a query window. Click **Finish** and you should see a big **Success** message, like the one in Figure 20-16!

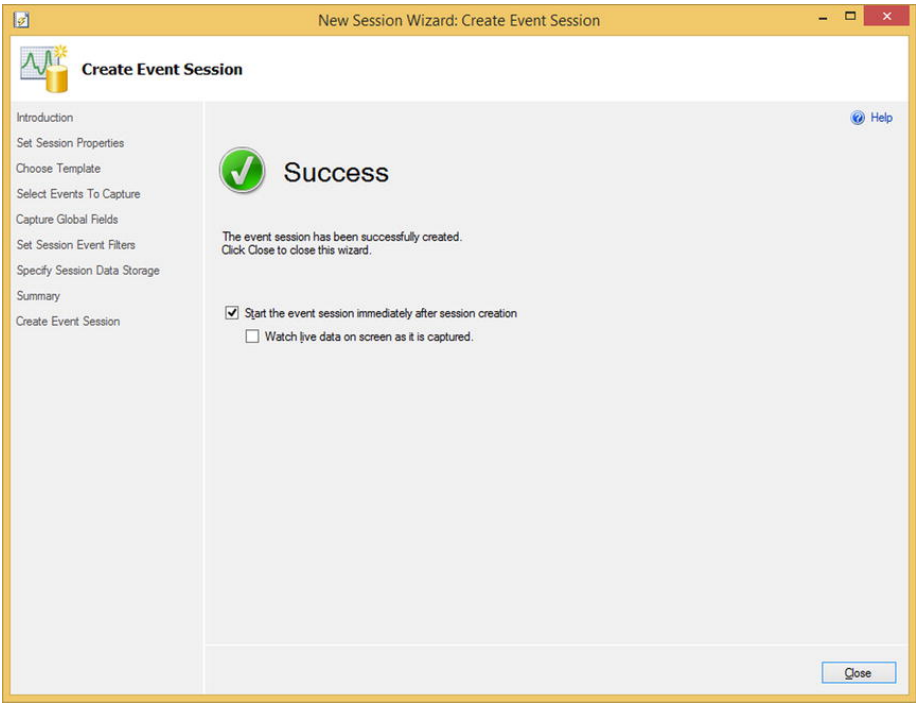


Figure 20-16. Successful creation of an event session

Before clicking **Close**, make sure you click **Start the event session immediately after session creation**. This will start the session and it will begin capturing data.

The session name will appear under **Extended Events** ► **Sessions** in Object Explorer (you may need to right-click this node and refresh to see the new session). Right-click it and choose **Watch Live Data**. The session will appear, empty as no events have been captured yet (Figure 20-17).

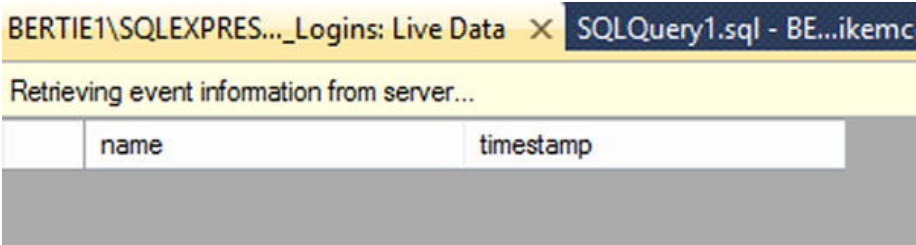


Figure 20-17. A newly running Extended Events session

In Object Explorer, right-click the server name (the topmost item) and choose **Disconnect**. Click **Connect** and reconnect to the server. Extended Events will capture some data linked to the logout/login events you just raised.

Run this and return to the Live Data window. You will see the event as it was captured. Don't be surprised if your event list doesn't exactly match Figure 20-18—as long as some rows appear in there, you'll know everything is working.



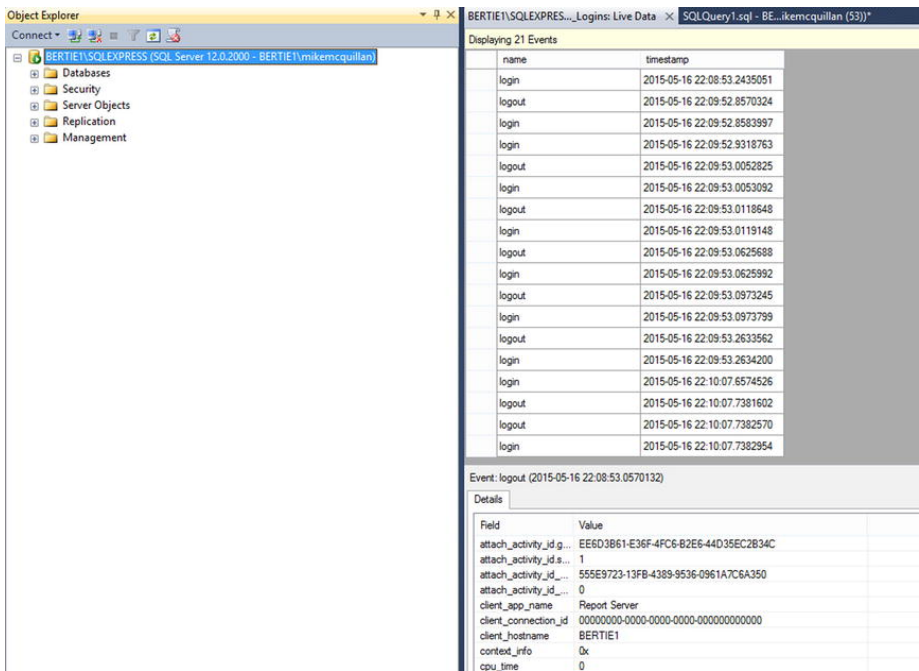


Figure 20-18. Events captured by the Extended Events session

If you want to look at a set of events for a particular Extended Events session, you need to expand the session name in Object Explorer. Right-click the item below it (this unusually named item specifies where the data are being captured to—**package0.ring_buffer** means the data are being stored within SQL Server) and choose **View Target Data**. The data will appear in XML format and you can inspect them. You can see an example in Figure 20-19.

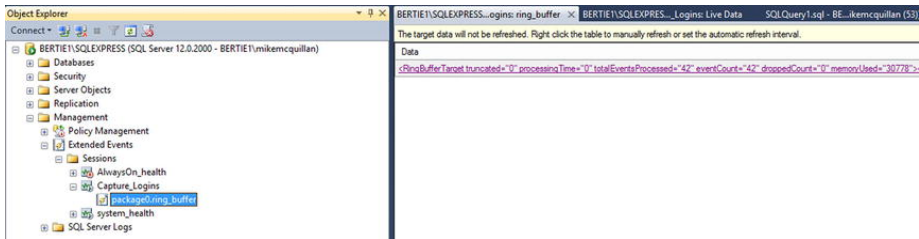


Figure 20-19. Viewing Extended Events in XML format

Summary

Congratulations, you have made it to the end of the book (if you ignore the appendixes!). I hope you’ve learned something useful. This chapter was a bit of a mishmash, but it aimed to give you pointers for some key SQL Server technologies that will definitely benefit your database career.

SQL Server is a huge product, but it can be very rewarding to work with. You may never know it all, but never stop learning and you’ll have a great career. If you feel like you just can’t finish yet, hop on over to Appendix D —there are some exercises for you to undertake.

Thank you for taking the time to read this book—it means a lot. You can contact me at mike.mcquillan@mcqtech.com if you’d like to discuss anything from the book that has raised your interest (or even if you just want to say “Hi!”).

Good luck, and enjoy querying!



PREV

Chapter 19 : Stored Procedures...

NEXT

Appendix A: SQL Data Types

R

S

© 2017 Safari. Terms of Service / Privacy Policy

