Course Overview

Course Overview

Hello there. Welcome to Pluralsight. My name is Mike McQuillan, and this course is Programming SQL Server Database Stored Procedures. Writing database code in an ad hoc fashion can be tiring, inefficient, and worst of all, plain boring, but stored procedures are here to help make things far more exciting. This course will show you just how wonderful stored procedures in SQL Server are and how they can help you greatly improve your database development practices. You'll be introduced to the do's and dont's of stored procedure development, learning how a stored procedure used in the right manner can lead to more efficient and secure code. Along the way, you will see how SQL Server Management Studio can be used to create, edit, and manage your stored procedures. The best stored procedures allow users to control them via parameters. You can use parameters to dynamically change the output of the stored procedure, maybe as a filter on a report, for instance. The course goes into detail about parameters, and not just input parameters. You'll also see more complex parameters used, such as output parameters and table-valued parameters, also known as TVPs. TVPs are amazing, offering a fantastic way of sending multiple records to a stored procedure in a single call. It's like magic. Throughout all of this, we'll be creating plenty of stored procedures to demonstrate the techniques, including procedures that insert data and return data. Some of them might even do both. After learning about refactoring code, the course finishes up with a detailed look at debugging stored procedures. This ranges from using simple print statements to stepping through code line by line. I wonder if you knew you could do that with SQL Server Management Studio? Along the way, you'll see many tips and tricks for making the most of your stored procedures. Thanks for watching. See you at the course.

Creating Your First Stored Procedure

Introduction

Hello there. I'm guessing you're interested in writing code for databases and for Microsoft SQL Server in particular. If you've just thought, yes, I am, you're in the right place. Welcome to this course, Programming SQL Server Database Stored Procedures. My name is Mike McQuillan, and this course is going to introduce you to the powerful programming language built directly into the SQL Server product T-SQL. We're going to concentrate on stored procedures, which are blocks of code that perform one or more actions on the database. Stored procedures are very flexible and very powerful. In fact, they're the most commonly used piece of programmatic functionality available in SQL Server. Whew, that was a mouthful. Let's take a look at what we're going to cover in this course. We're going to begin with a look at what stored procedures are in Creating Your First Stored Procedure. We'll see why they are used, what they can do, and why they are useful. We'll also check out times when they shouldn't be used. Once we have a good idea of why and how we are creating stored procedures, we'll begin to look at implementing them in the next module, Creating Stored Procedures and Using Parameters. We'll begin by creating a simple stored procedure before expanding it to use common, but more advanced techniques, like parameters. Most parameters used by stored procedures are simple primitive types like integers and strings. But did you know you can create your own custom data types in SQL Server and pass these into the stored procedure? It's true. In the snappily named Table-Valued Parameters and Refactoring module, we'll investigate how to create a table value type and how we can use it in a stored procedure. Now, nobody writes perfect code first time. I certainly don't. I often need to fix bugs in my code, and SQL Server comes with a number of ways to assist with that fixing. In Debugging and Troubleshooting Stored Procedures, we will see how various SQL Server features like the print statements and the try/catch statements can be used to make our code more robust and how they can help report errors that occur when our stored procedures are running. We'll even look at how we can step through code line by line. By the end of all that, you should have a solid foundation from which you can begin creating your own stored procedures. Does this sound of interest? Awesome. Let's go ahead and begin programming SQL Server database stored procedures.

A Quick SQL Server Recap

Now, as you are watching this course, it's reasonable to assume you have a pretty solid idea of what SQL Server is, but I'll just give a very quick recap for any newbies amongst us. SQL Server is a massively popular RDBMS, relational database management system. It's been around since the late 1980s and is owned by Microsoft who have

released several versions of the product. Release schedules used to be a bit patchy. There were five years between SQL Server 2000 and SQL Server 2005, for instance, but since 2012, a version has been released roughly every 18 months or so. I mentioned SQL Server is popular, but how can I prove that? Well, job adverts for a good barometer. Look around, and you'll see many developer posts ask for SQL Server expertise. There's also the popular db-engines.com website, which ranks database systems using things like number of website mentions, how often something is mentioned in technical discussions, and relevance in social networks. If you check this website out, you'll find SQL Server is high up on the list and has been for a number of years. There are millions of SQL Server instances out there, and Microsoft continues to aggressively develop the product, so it's not going away anytime soon. Knowing your way around SQL Server will give you an advantage over developers who don't want to pick up database technologies. It's always useful to have more than one string to your bow, but many developers I've met don't bother learning how to write SQL, the language used by most relational databases. They want to concentrate on what they see as real programming languages, like C# and Java. That's exactly what happened to Grace, a developer at Dolly's Dolls, a company that supplies dolls to toy shops. Her team has been working on a contact system, which will allow staff at Dolly's Dolls to capture contact info for their customers, suppliers, and other types of contact. This is the first phase in the redevelopment of the Dolly's Dolls systems. And all was going well until a database developer unexpectedly left the company. Management are scrambling to recruit a new database developer, but in the meantime deadlines are looming, and Grace has been asked to move from website development to database development. This has put Grace into a mild state of panic as she doesn't know anything about database programming. She receives a quick debriefing from her manager. The database structure has been finalized, and it captures everything the system needs. It's also reasonably elegant, and the relationships between the tables are clearly defined. Some test data has also been added to the system. It's possible to capture contact details, addresses for those contacts, phone numbers, notes, roles, and verification details, like driving license and passport details. The only problem is there's no way of adding, updating, or removing data to and from the system. Absolutely no code has been written to support inserts, updates, or deletes. The system can't even read records from the database at the moment. Grace has been asked to write database code that will allow the website to read the contact details out of the system and to allow records to be inserted. She's had to look around the web and keeps coming across the term stored procedure. The common consensus says this is the way to go. She begins to investigate further by fiercely discovering exactly what a stored procedure is and what it can do.

What Is a Stored Procedure?

Most of the popular relational database systems, SQL Server, Oracle, MySQL and the like, support stored procedures. A stored procedure is nothing more than a piece of code that performs some repetitive set of actions. It performs a particular task by executing a set of actions or queries against the database. The code for the stored procedure is stored in the database and can be executed at any time. Stored procedures are typically used to insert

your records into one of more tables, update or delete data from tables, and to generate reports via the SELECT statement. It's actually possible for a stored procedure to do more than one thing. For instance, you might want to run an UPDATE statement which modifies some information, then run a SELECT statement to return the updated statistical information. Stored procedures are useful because they allow the developer to write code once, then execute it many times, just like a developer might do in a stereotypically normal language like C#. It's possible to build up a library of stored procedures for a database over time, all of which provide a programming interface to the database tables. You can even add complex business logic in a stored procedure if you wish. On top of the programming benefits, stored procedures reduce the amount of network traffic. Imagine Grace had to write some T-SQL code which inserted a record, then selected the data created for that record. Without a stored procedure, her code would have to send the INSERT statement first, wait for notification that the insert has succeeded, then send another request for the SELECT statement. With a stored procedure, the code just sends the values for the stored procedure to use, then waits for the response, which will include the output of the SELECT statement. This simple example reduces two network calls to one. But imagine if the code needed to do four or five different things. How much network chatter would compiling the tasks into a stored procedure eliminate? Another useful benefit offered by stored procedures involves security. If T-SQL statements like INSERT and DELETE are being executed directly on the relevant tables, the user must be granted the relevant permissions on the tables. These users have the granted level of permission for the entire table and could wipe out the content of that table if they managed to access it. A stored procedure avoids these kinds of scenarios. For a start, the user would not have access to the table at all. They would have access to execute the stored procedure. The stored procedure will talk to the tables on the user's behalf. This has the huge benefits of simplifying and enhancing security at the same time. It's also worth mentioning that the code is more efficient. As the code always follows a particular pattern, SQL Server can generate an execution plan for the stored procedure, allowing it to execute in the most efficient way possible. Okay, so Grace has a reasonable idea of what stored procedures are, what they can do, and why she should be using them. She tries to find out what stored procedures can't do and pretty much comes up empty. The stored procedure is the principle programmatic method of database communication. You can do virtually anything in a stored procedure. However, just because you can, doesn't mean you should. Grace discovers there are a few things that are frowned upon in stored procedure development. These include limiting how many result sets are returned from a stored procedure. It's possible to return multiple sets from a single stored procedure. Grace could write a procedure, for instance, which returns contact detail, addresses, notes, phone numbers, verification details, and roles. But why return all this information if the calling application doesn't need it? There may be times when returning multiple record sets makes sense, but base these decisions on the logic of your application. If a summary screen displays client information at the top of the screen, but notes at the bottom, it makes sense to return two record sets, one for each set of data. Another thing Grace discovers is that using cursors is a definite no-no. A cursor is an SQL Server object that behaves like a foreach loop. You can loop around the cursor, reading records one by one. It's amazingly rare to have to do this kind of thing in database systems because you have access to the power of set-based logic. This allows you to apply

code to multiple rows at the same time. Instead of looping through on a row-by-row basis, an UPDATE statement to mark all contacts as verified would run in no time at all using one line of code, whilst doing the same thing using a cursor would take a lot of code, be less maintainable, and would also run a lot slower as the size of the data set grew. Grace feels like she's already learned quite a bit about stored procedures. She wants to start looking at some of the more technical aspects, but needs to install SQL Server and the database before she can.

Installing SQL Server and Setting up a Database

Grace decides it's time to open up SQL Server Management Studio and see how all of this stored procedure business really works. Dolly's Dolls already has a development SQL Server instance set up called DOLLYDEV1. Grace doesn't feel confident enough to use that yet. She wants to have a local instance of SQL Server running so she can experiment without breaking anything. There are many ways of using SQL Server. Some of these don't need installation. If you haven't Azure account, you could set up an SQL Server instance in the cloud. There are some limitations compared to the on-premises edition, but most of these will not affect the majority of users. However, using the on-premises version gives you more control of the installation and won't affect your bank balance when you use it. There are a number of SQL Server versions available. Standard and Enterprise cost money and are used in production environments, so you don't want to be using those for experimentation. Express is a limited free edition. But the one to download is the free Developer edition from Microsoft. Developer is essentially the same as SQL Server Enterprise, but isn't licensed for production use. It's great to be able to obtain the full product for no cost. It's one of the reasons why SQL Server is one of the most popular databases out there. It just helps developers learn. The next step is to decide which operating system to use. Hang on, what's that you are saying? SQL Server only runs on Windows, doesn't it? No, it doesn't. That was the case until 2017, but now you can run SQL Server on a number of Linux distributions and also on Docker. Choices, choices. Dolly's Dolls use Windows, so Grace goes down the traditional route and installs SQL Server on her Windows system. With the product installed, Grace tries to access her SQL Server instance. There doesn't seem to be an application installed which lets her use it. She knows SQL Server Management Studio is the application used to interact with SQL Server, but it hasn't been installed. Initially confused, she returns to the download page and reads it. She scrolls down and notices a Tools section. Aha, there's the link for Management Studio. She clicks this and proceeds to download and install Management Studio. Management Studio used to be delivered as part of the main SQL Server download, but has been separated out now to make it easier for developers to choose what they want to install. Although Management Studio is the most common tool used for managing SQL Server, it is possible to use Visual Studio too. However, Management Studio is a fantastic tool, as we'll soon discover. With the installation's complete, Grace fires up Management Studio and connects to her SQL Server instance. She expands the Databases node and sees nothing. No databases. She needs to create the Contacts database. She has already pulled the code down from the company's source code repository. Now she needs to run it. The main Apply script requires SQLCMD, a special mode in Management Studio

that allows SQL Server to run command-line scripts. Grace has to go to the Query menu and select SQLCMD Mode to turn this on. SQLCMD Mode requires a path to grab the SQL files from. A path has to be specified in the Apply script for this to work. It's set to C:\temp\contacts\. Grace has already downloaded the code into this folder, so she runs it, and it succeeds. She refreshes the Databases node and sees a Contacts database in the list. Hooray! Grace has been told how to generate a diagram of the database so she can see how everything fits together. Using the Object Explorer in Management Studio, she expands the Contacts database, right-clicks on Database Diagrams, and selects the New Database Diagram option. As it's the first time she's created a diagram in this database, Management Studio asks if she wants to install the diagrammatic objects. She hits Yes and is taken to the diagram editor, which immediately flashes up a list of tables that can be added to the diagram. Grace, clicks the Add button until all of the tables have been added, then clicks Close to remove the dialog. Now she can see the database diagram. She unpins the Object Explorer to give her a bit more room and inspects the database structure. It's clear that the Contacts table is at the center of everything with all of the other tables hanging off it. The only slightly different structure is the Roles table, which has a many-to-many relationship with the Contacts table via the ContactRoles table. This means a single role could be linked to multiple contacts. This makes sense when you think about it. If there's a role called director, for instance, that role is likely to apply to many of Dolly's Dolls contacts.

Stored Procedure T-SQL Statements

With the database ready to use, Grace wants to create a new stored procedure. She knows there are three T-SQL statements available to manage stored procedures, CREATE PROCEDURE, ALTER PROCEDURE, and DROP PROCEDURE. The names of each of these are pretty obvious. Grace looks at the Microsoft help page and is a bit bewildered. It looks really complicated. However, after a bit of reading and scooting around the web, she realizes the essence of the CREATE PROCEDURE statement is pretty simple. Declare it, add parameters, then add the code. If I had to take a guess, I'd say 95% of all stored procedures followed this structure. The name of the procedure is always supplied first, followed by parameters. Parameters are optional. A stored procedure doesn't need them. They can be very useful though, as we'll see later in the course. The brackets surrounding the parameters are also optional, as are the BEGIN and END statements. I think the BEGIN and END statements give additional clarity to the structure of the code, as do the brackets, so I always include them. The AS keyword is mandatory, as this dictates the start of the stored procedure code. Stored procedure names must be unique; however, you can use the same name if it exists in different schemas. You can think of a schema as a folder which contains all similar objects. You may create a Report schema, for instance, and a Contact schema. There could be a stored procedure in Contact called SelectContact and a procedure with the same name in the Report schema. The actual names as SQL Server would see them would be Report.SelectContact and Contact.SelectContact. It's probably best to keep names unique, even across schemas, but it's worth considering grouping database objects for a particular purpose in a schema. What if you want to change one of your stored procedures? ALTER PROCEDURE is the boy for you. Grace again

checks out the help page. There are no real differences between CREATE PROCEDURE and ALTER PROCEDURE other than ALTER PROCEDURE does not change permissions. If you assign permissions on a user-by-user basis, ALTER PROCEDURE might be the call to use. In my experience, most developers tend to drop a stored procedure and recreate it if necessary, as permissions are granted via groups or roles. In this scenario, ALTER PROCEDURE can pretty much be ignored, but definitely use it if your permissions model demands it. Grace will have a little look at it shortly so she knows how to use it. The last T-SQL statement is DROP PROCEDURE. This is the easiest statement of the lot. You just supply the name of the stored procedure along with an optional IF EXISTS claused. The IF EXISTS only works in SQL Server 2017 and higher. All of the other stuff Grace has looked at so far will work as far back as SQL server 2008

Creating a Stored Procedure

Grace decides to go ahead and create a new procedure just to dip her toe into the water. She wants to create something that returns all contacts in the database. She returns to Management Studio, navigates to the File menu, then New and selects Query with Current Connection. This opens up a blank editor window. Grace types in a simple query to return all records from the Contacts table. She hits the Execute button on the toolbar, and the query runs. Quick tip, hitting F5on the keyboard also runs the query. Grace can see all of the contacts, so she knows the SELECT statement works. She navigates above the statement and begins to type, adding the CREATE PROCEDURE declaration. The code looks good, but the CREATE PROCEDURE statement has a red underline. Grace can't understand why. She decides to run the code anyway and sees the message 'CREATE/ALTER PROCEDURE' must be the first statement in a query batch. At first, she's a little confused. Then she remembers something she read in an internet article. SQL Server sees all statements in the script as a single batch of commands, but certain commands, especially data definition language statements like CREATE PROCEDURE, must exist within their own batch. The USE statement at the top of the script, which tells the script which database to create the procedure in, is the first statement of this batch and is causing the error. The fix here is to add a GO statement between the two commands, which will separate them into two separate batches. With GO, the red underline has gone, and when Grace runs the script, it succeeds. She wants to try the procedure, so she opens up another script window and types in the EXEC command to fire up the stored procedure. She runs the code, and it works. Grace has successfully created her first stored procedure. She returns to the window containing the procedure's code. There are a couple of interesting points to make. First, the USE statements. I can't tell you enough how important this is. Many a time I've created a procedure in the main master database all because I forgot to add a USE statement. Always make sure you are in the correct database. One little extra thing to be aware of though. The USE statement is not supported in SQL Azure, so if you're creating stored procedures there, make sure you select the correct database before running your script. The other thing of note is the semicolon at the end of each statement. This was first introduced around SQL Server 2008, but only for certain statements. It's not mandatory, but

Microsoft recommends you use it as it could become mandatory in future, and indeed, it must be specified for certain types of statements. If you can put yourself in the habit now, it's a good practice to follow, just like Grace is doing here. Grace decides to test out the ALTER procedure statement. When she ran the procedure, she noticed her name was in the Contacts table. She changes the code to only return contacts with the first name of Grace. After changing the stored procedure code over, she executes the change, returns to the window where the EXECUTE statement is, and runs the command again. This time, the stored procedure just returns Grace's record as she's the only Grace in the table. Nice.

Managing Procedures Using SQL Server Management Studio

Grace decides to hunt her new procedure down in the Object Explorer. She expands the Programmability section and sees a subsection called Stored Procedures. Sounds like a good candidate, and she's right. Expanding that displays two items, a System Stored Procedures folder, which can be ignored, and the dbo.SelectContacts procedure. Right-clicking on this provides several options. The procedure can be modified, and you can also view any dependencies it has. Grace checks this, and SQL Server correctly deduces there is dependency upon the Contacts table. You can view objects that are dependent upon the procedure and objects the procedure itself is dependent upon. Grace returns to the Contacts menu and moves to the Script Stored Procedure as section. There are plenty of interesting options in here. She wonders what the DROP And CREATE To option looks like. She selects it and is a little disappointed that the name was very literal. But still, it's a useful feature to have if you want to quickly inspect the code for a stored procedure without resorting to your source control repository. Grace closes the generated script and right-clicks on the Stored Procedures option. She sees two options in the New item, Store Procedured and Natively Compiled Stored Procedure. What's the difference? Natively compiled stored procedures were first introduced in SQL Server 2014. When you create these procedures, they're compiled into the C programming language and are available for immediate use. However, these types of procedures can only access in-memory tables, which are a fairly advanced feature of SQL Server. In-memory tables are very fast and can be accessed faster by a natively compiled stored procedure as opposed to a good-old regular stored procedure like Grace created earlier. We won't be looking at natively compiled stored procedures in this course as they cannot be used to access disk-based tables, and disk is where the vast majority of SQL Server tables are stored, so we'll stick with old-fashioned stored procedures. To finish off her experiment, Grace decides to drop the stored procedure. She returns to the appropriate script and is surprised to see the name of the procedure in the ALTER PROCEDURE statement is underlined in red. Momentarily forgetting about dropping the procedure, she hovers over the underlined name, causing the error to be displayed. The database is reporting that the stored procedure has an invalid object name. But we've just looked at the code for this procedure, so how can it be invalid? After a minor panic, Grace discovers this is because SQL Server hasn't updated the metadata used by its IntelliSense feature. This can be fixed by going to the Edit menu, IntelliSense, and clicking Refresh Local Cache. You can also hit Ctr+Shift+R to solve this

problem. Once this is done, the red underlying disappears. Now Grace can concentrate on the deletion code. The first thing to do is add a DROP PROCEDURE statement at the top. She also changes the ALTER statement back to CREATE. She decides she only wants to run the DROP statement, so Grace highlights it. Doing this ensures Management Studio only runs the selected code. Management Studio reports success, and when Grace tries to execute the procedure, she is told it couldn't be found, which is correct. She returns to her procedure script and runs it to recreate the procedure, but unexpectedly she runs into an error. She's told the procedure cannot be dropped as it doesn't exist. And indeed it doesn't; she just dropped it. As she is using a recent version of SQL Server, the quick fix is to modify DROP PROCEDURE, adding the IF EXISTS clause. This works a treat, and she can execute the procedure again. If a version of SQL Server earlier than 2017 is used, you need to inspect one of the system metadata tables to see if the procedure exists. In this case, you'd need to check sys.procedures to determine if the procedure already existed. Grace quickly modifies the code to use this check. It's fairly involved. You can see why Microsoft introduced the IF EXISTS check. But still, Graces glad she looked this up. It will come in handy if she ever uses an earlier version of SQL Server. Grace really feels like she has achieved something. She's picked up quite a few SQL Server tricks and has created a simple stored procedure. She reports back to her manager who asks Grace to pick up some of the stored procedure backlog. Now it's time for some real work

Summary

This module introduced the concept of stored procedures in Microsoft SQL Server. After being introduced to Grace and Dolly's Dolls, we saw what a stored procedure is, what it can do, and some of the benefits offered by stored procedures, like security and code reuse. We also saw some bad practices and things to avoid, like using cursors. With the theory out of the way, SQL Server was installed and the test database scripts executed, giving Grace a database she could safely play with in a local environment. The module finished up by introducing the T-SQL statements used to create procedures and how procedures can be managed in SQL Server Management Studio. The stored procedure Grace created in this module was very simple, but in the next module, the real work will start, as she creates a procedure to insert records into the Contacts database.

Creating Stored Procedures and Using Parameters

Introduction

Hello! Still interested in stored procedures? Terrific! We saw the basic essentials of stored procedure creation and management in the last module. That has set us up wonderfully well for this module, Creating Stored Procedures and Using Parameters. This is where we'll really start to see how amazingly useful stored procedures are. We'll start off by creating a new parameter-less stored procedure. Once we have put our initial offering together, we'll add parameters, allowing us to provide inputs to the stored procedure. You might think simply adding parameters is the end of the story, but no. There's so much more to parameters. We can make them optional, and we can also use them to return data. We'll see both of these features in action. And we'll also check out how a stored procedure can execute a number of related operations. We'll finish up by investigating some of the system options SQL Server makes available to us and how they affect our stored procedures. By the end of this module, you should feel pretty confident creating stored procedures. Let's make a start.

The Business Requirement

Grace, who as you might remember is just starting out in the SQL Server world, has created one simple stored procedure so far to return all contact details. It was pretty basic and consisted of nothing but a SEELCT statement. It was still a perfectly valid stored procedure, but it wasn't terribly useful, and it wasn't terribly exciting either. But now a real business requirement has come Grace's way, and they need the stored procedure as soon as possible. Isn't that always the way? She's been asked to create a mechanism that allows new contacts to be inserted into the database. The code has to allow the calling program to insert a contact with specified name, date of birth, and allow contact by phone values, then return the details for the generated contact, including the ID. It shouldn't insert a contact if that contact already exists. How could Grace do this? The first option is to manually open the Contacts table in Management Studio and type the values directly into the table. Really, this is a nonstarter as it means users need to have Management Studio installed if they wish to insert or modify data. It also makes it easy for a user to make a big mistake, like accidentally deleting all data in the table. Add to the fact that isn't a customized user interface, and it's clear this is a bad idea. Another option is to write an INSERT statement, embed it into the calling code, and then execute it using a database communication mechanism like ADO.NET. This could meet all of the requirements, but there are severe limitations. For instance, if the database schema changes, it means the application code needs to change as the SQL is embedded in the code. The INSERT statement could be stored in a separate file, allowing it to

be read and executed, but that's giving the job of managing SQL to the application, not the database. It also makes parameterizing the code much more difficult. One last option to consider is using something called an object-relational mapping system, or ORM for short. Examples in the .NET world are Entity Framework and NHibernate. These are software libraries that try to take a lot of the database hassle away from the developer. They aim to prevent the developer from needing to write SQL code, instead generating statements on the fly through an object model. ORMs can be a good solution, but they require the model to be updated if the database schema changes. This usually requires a recompilation of the application. ORMs offer good parameterization facilities and hide a lot of the database implementation from developers; however, they also take a lot of control away from the developer. The SQL the ORM generates can be overly complicated and inefficient. They are worth looking at, but probably won't fulfill your needs if you want total control over your database access. Okay, so we've seen a number of ways the insert contact functionality could be implemented, none of which were hugely convincing. That leaves just one candidate. You guessed it, the humble stored procedure. A procedure can safely handle the three core requirements. The T-SQL INSERT statement can be used to perform the inset, and a SELECT statement will take care of returning the ID of the generated contact. There are other ways to return the ID, as we'll soon discover. Checking whether a contact exists is a little more involved, but shouldn't be overly onerous. Let's join Grace as she makes a start.

The Insert Contact Stored Procedure

Grace has opened up SQL Server Management Studio and is staring at an empty new query window. She adds a USE Contacts statement to ensure the procedure is created in the correct database and then adds the batch separator GO statement. She now adds the CREATE PROCEDURE statement, calling the procedure dbo.InsertContact. It's possible to include spaces in procedure names, but if you do that, you'll have to wrap the name in square brackets every time you call it, and it will cause no end of confusion to people using the procedure. Don't do it! Upper camelCase is generally used, and it works quite well. Grace completes the core stored procedure structure by adding AS BEGIN and END. Now she has an outline, but she has to populate it with code. The first requirement stipulated the calling program must be able to specify the values to be inserted for the contact. She declares four variables to hold these values, FirstName, LastName, DateOfBirth, and AllowContactByPhone. All variables in SQL Server begin with an @ symbol, and they should all be named sensibly, no spaces or anything else that could cause confusion. Any valid SQL Server type can be used to set the data type for a variable. Grace wants to make sure the variable types match the columns they are going to be inserted into. She expands the Columns for the Contacts table in Object Explorer and sets the variable types to match. This is a really important implementation point. Look at the FirstName column. It accepts up to 40 characters. By setting the @FirstName variable to the same length, we guarantee we cannot overflow the column. If our first name was, say, 60 characters in length, we could pass in a string that is too big for the column, which might be rejected or cut off at 40 characters. Either way, that's

not what we want. With the data type set correctly, Grace assigned some values to them. Let's hope this procedure doesn't turn into a nice mess. With Stan Laurel already and waiting to go, Grace adds an INSERT statement. This works by declaring the table we want to insert into, followed by the columns to insert values into, and finally, the values we want to add to those columns. Well, Grace has written the stored procedure to insert the code. Great! She executes the code and sees that it has run successfully. Now to test this stored procedure.

Executing and Testing a Stored Procedure

Grace opens a new query window and writes the code to test the InsertContact stored procedure. The command to use when you want to execute a stored procedure is, rather obviously, EXECUTE, although many people use EXEC for short. When Grace types this in, she is surprised to see that the name of the InsertContact procedure is underlined in red. She's confused. Is there an error? Hovering over the underlined word displays the error. The stored procedure doesn't exist. But Grace knows she's just created it. There's no need to worry. All that's happened here is Management Studio's IntelliSense feature hasn't updated its cache, so it doesn't know about this new procedure. This could easily be fixed either by pressing the keys Ctrl+Shift+R together or by going to the Edit menu, choosing IntelliSense, then clicking on Refresh Local Cache. Magically, the red line disappears. Grace now feels confident that she can run the stored procedure. She executes it and a message is displayed, (1 row affected). So, something involving one row has happened, but was it what Grace expected? And why are there two messages stating one row has been affected? Hold that thought. She writes a SELECT statement to pull back all of the contacts, but orders it in descending contact ID order. The contact ID is the primary key of the table and is automatically incremented when a new record is added, so the most recently added contact will have the biggest contact ID. She highlights the SELECT statement and runs it. Why did she highlight it? That's a nifty little Management Studio feature. If you highlight a particular section of code, just that section of code will be executed. The results have appeared, and we can see Stan Laurel is at the top of the list. Great! So inserting a contact works. Grace runs the procedure again, and Stan Laurel now appears twice, but there was only one Stan Laurel. Grace has created the basic procedure, but she now needs to move on to the next part of the development, allowing the calling program to specify which values are to be inserted. For that, she'll need some parameters.

Adding Parameters to a Stored Procedure

Okay, so I can't imagine a stored procedure that continually adds Stan Laurel to a database table is going to be the next big thing, so Grace needs to change things up. Fortunately, she made a good start when she declared the variables in the first version of the procedure. Parameters are nothing more than variables which can be passed in by the calling program. It's possible to declare parameters for all of the common SQL Server data types, such as the string types like char and varchar, numeric types such as int and decimal, and other data types like dates and XML.

As we'll see in a later module, you can even create your own types and pass those in. Parameters are important as they make your code flexible and reusable. A stored procedure adding Stan Laurel every time it runs is pretty useless, but a stored procedure that can be told who is being added is a very enticing prospect. Before adding the parameters, you need to ask yourself what the parameter is for. In this case, Grace needs parameters for the contact's first name, last name, date of birth, and telephone contact setting. I mentioned the variables she has already declared. These can quickly be converted into parameters. Parameters are nothing more than variables, so they follow the same conventions, begin with an @ symbol and follow standard naming rules. Grace returns to her code, navigates to the beginning of the AS line, and hits Enter to add a new line. She adds opening and closing brackets. These will hold the parameters. Do you remember the basic outline of a stored procedure we saw in the previous module? It had parameters in brackets. Grace is following this template right now. She highlights the variable declaration section and cuts the code, pasting it between the brackets. A few errors appear. They are easy to solve. Parameters do not need the DECLARE keyword. They are automatically declared by the brackets. Grace removes this keyword. The semicolon is underlined, but that's an easy one too. The semicolon is still present at the end of the final parameter. That's not required either. Deleting it soon solves the issue. Now the stored procedure name is underlined. This time, it's not because Management Studio thinks it doesn't exist, but because it knows it does exist. Grace will worry about that one in a moment. For now, she wants to finish off the code. Her last action is to remove the SELECT statement that sets the variables to the Stan Laurel values. She wipes this code, leaving the main body of the stored procedure as a simple INSERT statement. Grace executes the CREATE PROCEDURE statement, expecting the stored procedure to be created, but she forgot about the error message highlighted in the code. The stored procedure already exists, so she needs to add the DROP PROCEDURE IF EXISTS line before it is created. This will ensure the procedure is dropped before the CREATE PROCEDURE statement executes. After adding this line, the code runs successfully. She returns to the query window and runs the test. It blows up. Grace is being told a parameter called @FirstName is required. Yes, that's right. The parameters she has added are mandatory. There are two ways to pass the parameters when testing in Management Studio, passing the values directly or specifying the names and values of the parameters as key-value pairs. I prefer the latter as it makes things more obvious. Here's Grace's first effort in which she doesn't specify the parameter names. Now, this works, but it doesn't really show what's happening. Grace isn't that keen either. She decides to include the parameter names. This makes things much easier to read. We can see exactly what value has been assigned to what parameter. Grace notices her procedure is underlined in red again, so she refreshes the cache again. This time, Management Studio was complaining because parameters were specified. IntelliSense hadn't realized there's a new parameterized version of this stored procedure in town. Grace runs the code again, and Terry Thomas successfully becomes a contact of Dolly's Dolls. The procedure is working really well. Grace can run it with any combination of values, and they will be added to the table. It's important that data of the correct type are passed to the appropriate parameter; otherwise, an error will occur. Strings should always be wrapped in single quotes, and dates need to be presented in the format required for your database instance. Numbers do not need single quotes, and neither do bit parameters. If

the wrong type of data is passed, a data conversion error may appear, or worse, the data could be automatically converted to the target data type and inserted. Not necessarily what you might want to happen, so always make sure you specify your values correctly.

Optional Parameters

Grace likes this procedure, but she's looking at the structure of the Contacts table and realizes she needs to make another change. All of the columns in the table except DateOfBirth state not null, meaning empty values cannot be stored in those columns; they are mandatory. DateOfBirth does allow nulls though. In other words, it's optional. The system will allow contacts to be created without a date of birth. How can she represent this in her stored procedure? Simple, by using optional parameters. It's possible to assign a default value to a parameter. This default value acts as a placeholder. If no value is passed for the parameter when the stored procedure is called, the default value is used. The code can then check for this default value and determine what course of action to take. Throw an error, insert the data, maybe even convert the data. Making a parameter optional is amazingly straightforward. Grace returns to her code and adds = NULL to the @DateOfBirth parameter. That's it. The @DateOfBirth parameter is now optional. If it isn't passed, NULL will be set as the date of birth value. The default value can be any value suitable to the data type. First of January 1900 would've been equally valid. Avoid so-called magic values like this if you can though. They generally add confusion as they look like real values. NULL exists in the database for a reason. It means empty. So if a value is supposed to be empty, set it to NULL and ensure it stays empty. Grace runs the code, and as usual by now, it succeeds. She heads back to the test script and changes the parameter values, entirely removing the @DateOfBirth parameter. She runs the script, and look, Norman Wisdom puts in an appearance in the table without a date of birth. Everything is coming together now. Let's stop for a moment and check out the original requirements again. Grace needed to allow the calling program to insert a contact with a specified name, date of birth, and allow contact by phone values, and then she needed to return the details for the generated contact, including the new ID for the contact. Finally, the stored procedure should not insert a contact if that contact already exists. Well, the first item is complete, but Grace hasn't implemented the other requirements yet. Let's remedy that right now.

Retrieving Record Identifiers

The stored procedure already inserts the new record. Now it needs to return it. This means Grace needs to obtain the ID of the new record. This involves making the procedure do an additional piece of work. Now, stored procedures can be small or big. A procedure might perform multiple different operations whilst it is running. This is absolutely fine as long as what the procedure is doing is an atomic piece of work. By that, I mean the stored procedure should only perform one task. That task might be made up of lots of subtasks, a bit like the stored procedure Grace is creating

now. That's fine, as long as the collection of subtasks all work together to complete the principal purpose of the procedure. Don't create procedures that do a lot of different things. They won't make any sense to you or your colleagues. Separate out the tasks and create individual stored procedures for each. More is better than less in this case. Now, back to Grace. She is considering how to obtain the contact ID generated for the new record. She has been told that the contact ID is an identity column. After some searching on the web, she's learned that these columns are normal numeric columns, but are automatically populated by the database when a record is added. The name comes from the IDENTITY clause, which automatically assigns the next incremental value to the ID. Grace has also learned that SQL Server provides a number of system functions, all of which can be used in your scripts. These used to be known as global variables, but Microsoft seems to have gone in for a trendy name change. There are a number of system functions available, but the one Grace is interested in is @@IDENTITY. This returns the last identity value inserted into the database, and it sounds like the answer to Grace's problem. She returns to her script and declares an @ContactId variable at the beginning of the procedure. She then pops to the bottom of the code and adds the line to set @ContactId to the @@IDENTITY value. She follows this with a SELECT statement, returning the details for the newly inserted contact. The SELECT statement is filtered using the @ContactId value. If you're wondering why she didn't just write WHERE ContactId = @@IDENTITY, it's all to do with good practice. @@IDENTITY could change if somebody else inserts a record, so you need to store the value as soon as you can, which means the statement straight after the insert. That should do the trick. She runs the script to update the procedure and runs the test script again. However, things don't go as planned. Two record set are returned. The first is the one executed by the stored procedure, returning the contact details. Bob Monkhouse was inserted, but Stan Laurel is rearing his head again. Grace can see that Bob Monkhouse was inserted, but why was Stan Laurel returned? Weird! Confused, she tries again, adding June Whitfield. This time, Oliver Hardy comes back. Are Laurel and Hardy trying to take over the database? Grace stares and notices that Oliver Hardy is ContactId 23. Stan Laurel was ContactId 22. It seems somehow that the wrong identity value is being returned from the code, but why? She speaks to one of her more senior colleagues who looks at the code with her. At first, they are baffled, but then the senior developer remembers. The old database developer added a trigger to the Contacts table when he was experimenting. Maybe that's causing the problem. They expand the Triggers node in Management Studio, and lo and behold, there is indeed a trigger. It inserts a default record into the ContactAddresses table and is clearly not meant to be part of the codebase. Grace runs a quick query against the ContactAddresses table and can see that records 22 and 23 correspond to the address specified in the trigger. The mystery is solved. Triggers run whenever something happens to a record in a particular table. The type of action can be specified, so a trigger might run before an update or after a delete or, as in this case, after an insert. What's happening here is Grace's code is inserting a contact. At that point, the @@IDENTITY value is set to the new contact ID. But the trigger then runs. It inserts a new contact address record. The identity column on this table is activated, and that value is assigned to @@IDENTITY. As that's the last value assigned, that's the value used in the SELECT @Contact = @@IDENTITY line. So clearly, @@IDENTITY isn't safe to use. We cannot guarantee the correct value is going to be returned. @@IDENTITY

returns the last identity value inserted into the entire database, whereas Grace just wants the last identity value inserted into a particular table. Her colleague thinks there's a way to do this and advises her to look at Microsoft documentation on @@IDENTITY. Maybe that will point the way. Grace reads the document and is interested when it mentions alternatives to @@IDENTITY. There are two other options, IDENT_CURRENT and SCOPE_IDENTITY. IDENT_CURRENT sounds of interest. It is limited to a particular table, but it applies to any session and any scope. This means if two calls inserted a record at the same time, one of them could receive the wrong identity value. Highly unlikely, but it's a possibility. SCOPE_IDENTITY, however, only returns the identity value within the current scope. That means it can only return the identity value inserted by the stored procedure. Just what Grace wants. She decides to give it a try. She returns to Management Studio and closes the trigger code. She pauses for a moment, then decides to delete the trigger. It's just causing issues and creating messy data. And now that she thinks about it, the trigger has been the cause of the duplicate 1 row affected messages she has been seeing when running her queries. With the trigger deleted, Grace switches back to her stored procedure code and changes at @@IDENTITY to SCOPE_IDENTITY. SCOPE_IDENTITY is a metadata function, which is why the brackets are needed. Grace applies the changes, then runs her test script, inserting Terry Scott. This time, the new Terry Scott record is returned. Success at last! Grace's stored procedure is now returning the contact details. However, the requirements implied that maybe just the contact ID should be returned. How could Grace do that?

Output Parameters

Grace's first option is to simplify the SELECT statement. She could just shorten it to return the ContactId column, but there is another way, an output parameter. So far, all of the parameters we've seen have been input parameters. They stipulate the values that are passed into the stored procedure, and the code then acts upon them. But it's possible for a stored procedure to have output parameters too. With these, a user assigns a variable to each output parameter. The code in the stored procedure assigns a value to the parameter, which is then assigned by the EXEC call to the variable. The caller can then use the value in that variable in their code. Grace decides to try an output parameter. The first thing she needs to do is declare the output parameter itself. This is initially declared just like an input parameter. She calls it @ContactId. Grace adds the word OUTPUT after the data type, which rather obviously turns the parameter into an output parameter. As soon as the @ContactId parameter was added, the existing @ContactId variable was underlined in red. That's because this variable is now a duplicate of the output parameter. As a result, it's no longer required and can safely be deleted. That's the only code change required as the line assigning SCOPE_IDENTITY to the @ContactId parameter remains viable. That was a simple change, but will it work? Grace applies the procedure changes and switches to the test script. She runs this and hits an error because the @ContactId parameter is now mandatory. Calling a procedure with an output parameter is not as simple as assigning a value like the input parameters. Firstly, a variable has to be declared. Grace calls this @ContactIdOut. It should use the same data type as the output parameter, which in this case is an integer. The output parameter is

then added to the call with a new variable assigned as the value. To check if the correct value is assigned, Grace modifies the SELECT statement to return the contact details using a WHERE clause to filter on the @ContactIdOut variable. If the output parameter is working as expected, that will hold the inserted identity value, and as a result, should cause the newly inserted record to be returned. Grace is just about to run this, but notices her new parameter declaration is underlined in red. She hovers over the line to see the error message and realizes this is another case of the IntelliSense metadata not keeping up to speed. She hits Ctrl+Shift+R on the keyboard, and the red underline disappears. Shazam! Grace now runs the test, and it doesn't work. Grace is a bit nonplussed. She thought she'd done everything correctly. She adds another SELECT statement to return the value of @ContactIdOut. Running the script again reveals the value is NULL. What has she done wrong? Well, it's an easy mistake to make. The EXEC statement has to be explicitly told which parameters are output parameters. To ensure the value is returned from the output parameter, the OUTPUT keyword must be added to the end of the output parameter declaration. With this in place, running the script again succeeds. Now Grace has options. She can choose to return the contact ID via a SELECT statement, an output parameter, or both. As she navigates around Management Studio, she accidentally clicks on the Messages tab. It displays a number of messages outputted by her code. Grace vaguely recalls the old database developer telling her these kinds of messages should be disabled. She decides to perform a quick investigation in how to do this.

Using SET Options

Grace wants to remove the messages that appear when her stored procedure is running. The stored procedure is outputting the row counts affected by each statement. The 1 row affected messages are being returned by the various SELECT statements in the test script and stored procedure as they are all filtered on ContactId. The INSERT statement in the stored procedure is also returning the same message. When it runs, one row is inserted, so the row count for that statement is 1. She has been advised to take a look at the SET options available in SQL Server. These options only affect the current session. The database provides a good number of SET options for developers. Grace visits the help page and scrolls through. Some of the options are quite interesting. SET DATEFORMAT, for instance, determines the format in which dates must be specified. SET IDENTITY_INSERT allows the currently executing script to manually insert a value into an identity column. The one that looks of most interest to Grace is SET NOCOUNT. That's the SET command used to turn the informational messages she's been seeing on or off. Including SET NOCOUNT ON in a script disables the count, which stops the messages being returned. To turn messages back on, you use SET NOCOUNT OFF. Now, you're probably wondering why the heck Grace is bothered about this. What's the big deal? Well, if you have a fairly big stored procedure, lots of these messages could be generated. If SET NOCOUNT ON isn't declared at the top of the procedure, all of those messages will be sent over the network to the calling program, along with the actual procedure output. It's very unlikely the development team will know or care about these extra messages, so why bother sending them? Better to make a slight performance gain and turn them

off. Try to put yourself in the habit of setting NOCOUNT ON at the top of your procedures. It's a really good practice to follow. Consider using SET NOCOUNT OFF when you are debugging and testing. It can come in really handy. There are benefits to using it during development. Now that Grace knows what NOCOUNT is for, she adds SET NOCOUNT ON as the first line of her stored procedure. It's considered good practice to enable a default at the end of the procedure, so she adds SET NOCOUNT OFF as the last line and then executes the code. She runs the test script and checks the messages. There are still two row count messages present. There were four messages last time, so two of them have gone. Why are the other two still there? It's because of the SELECT statements in the test script. The first SELECT returns the newly inserted contacts row. The second SELECT returns one row, the value of @ContactIdOut. Grace comments out those lines and runs the script again. Sure enough, no informational messages, just the completed successfully message. Many database teams using SQL Server will have a specific set of rules dictating which SET options are used and how they should be implemented. It's worth learning what each of the SET options can do. They could be very useful for you in the future.

Calling a Procedure from Another Procedure

Grace's INSERT procedure is just about done. She decides to create another stored procedure to return details for a specified contact. This is another function required for the contacts application, and as she knows what she is doing with parameters now, she thinks it will only take a few minutes to write. In Management Studio, Grace opens a new window, adds the USE database statement, and then a DROP PROCEDURE IF EXISTS line. She caused the procedure SelectContact. It has one parameter, @ContactId. The first line is SET NOCOUNT ON to prevent the row count messages from being returned. Then comes the main line in the procedure, a SELECT statement returning the appropriate columns from the Contacts table. The statement is filtered on the ContactId column using the value in the @ContactId parameter. The last line sets NOCOUNT back to OFF, a pretty simple procedure. Grace runs this and is told the procedure was created successfully. She opens up another window and writes a quick script to test it. She tries it with a valid contact ID, and everything works as expected. She tries a different ID, and that returns correctly too. Happily, it also returns an empty row set when an invalid contact ID is supplied. Cool! With the SelectContact procedure in place, Grace wants to make one final amendment to her InsectContact procedure. She moves to the line after she assigns SCOPE_IDENTITY to the @ContactId parameter, removes the existing SELECT statement, and writes an EXEC statement, calling the new SelectContact procedure. She passes @ContactId to it. Grace has just created a stored procedure chain. InsertContact is now dependent upon SelectContact. She runs this, then runs the InsectContact test script again. The contact details are returned. Great stuff! By having the SELECT statement in a stored procedure, it centralizes the code and makes it available to all elements of the database. If a change was needed, say SelectContact needed to return an additional column, this could now easily be done without affecting any of the calling procedures. Clever stuff, Grace. After refreshing IntelliSense, she sits back for a moment and takes stock of what she has achieved so far. From the position of absolute beginner, she has managed to create two useful

stored procedures, one to insert records and another to return them. Better yet, she's managed to link the two together to enhance the insert. Just one requirement to go now, ensuring a contact who already exists is not inserted again.

Adding Business Logic

Like any other programming language, T-SQL supports control flows. IF statements and WHILE loops are just two of the useful features you can use to dictate how your stored procedure should execute. The IF statement is what Grace needs to use to detect if a contact has already been added to the Contacts table. Some business logic is required to identify the new contact. If the contact already exists in the table, nothing should happen. The record will not be inserted as it already exists. If the contact does not exist, he or she should be added to the table. This can be done by inspecting the parameter values and searching for them in the Contacts table. If we ignore the @ContactId output parameter, there are four input parameters to hold FirstName, LastName, DateOfBirth, and AllowContactByPhone values. The first three can uniquely identify a person. It's possible for more than one person to have the same name and date of birth, but it's unlikely. The @AllowContactByPhone parameter is just a Boolean flag and does not form part of the user's personal information. So Grace decides to write a check using the FirstName, LastName and DateOfBirth values. To do this, she needs to check if a record with those values already exists in the Contacts table. She needs an IF statement for this used in conjunction with the EXISTS clause. IF EXISTS can accept a SELECT statement, which will return true if the contact exists. The check can actually be written as NOT EXISTS, meaning an ELSE clause won't be needed. Grace modifies her stored procedure again. She writes an IF statement above the INSERT line. The check isn't complicated. It says do not attempt to insert a contact if a contact that matches the values in the FirstName, LastName and DateOfBirth parameters already exists. The code in the procedure is wrapped in a BEGIN and END block, which denotes the scope of the IF statement. The SelectContact stored procedure is outside the scope and still runs at the end of the procedure. This is to guarantee the output from the INSERT procedure is always the same. This is very important. Your procedures should almost always return a consistent output regardless of the control flow followed by the procedure. This makes it easy for calling programs to handle the output. Now that looks good. Grace runs it. You might recall way back at the start of this module Stan Laurel was added to the database several times. Grace runs a quick query to pull back Stan's details. Well, yes, he's certainly in there more than once. She amends the procedure call to use Stan as the first name, Laurel as the last name, and a date of birth. She doesn't expect this record to be inserted. And hooray! It isn't inserted, just as Grace expected. To be on the safe side, she changes the date of birth and runs it again. As expected this time, the new record is inserted correctly. Running the procedure again with the exact same parameters doesn't insert anything. Marvelous stuff! Marvelous! Grace has successfully implemented all of the requirements for the InsertContact stored procedure. Her first proper stored procedure has been successfully completed. She knows this is just the tip of the

iceberg though. The next request on her list is asking for a stored procedure that can insert multiple records at the same time. Can this even be done? Grace reaches for her coat. That one can wait until tomorrow.

Summary

This module has introduced most of the core concepts you need to write useful stored procedures. We began by looking at how we could supply insert functionality to our users. After looking at options, including SQL Server Management Studio and object-relational mapping libraries, a stored procedure implementation was selected due to its simplicity and reusability. Grace then created a parameter-less version of the procedure, which continually inserted the same record. She added some parameters, making the procedure flexible and able to insert any contact. She even made one of the parameters optional. We then saw how to return the newly added contact record. @@IDENTITY was investigated, and discounted, leading Grace to use SCOPE_IDENTITY. She used this to return the contact ID firstly in a select statement and then via an output parameter. The module also showed how one stored procedure can call another, promoting code reuse and reducing the amount of code we need to write. The things you've seen in this module probably cover 75% of the stored procedures you're ever going to need to write. Most stored procedures are variants on inserts, updates, deletes, and selects, often with some business logic applied. One of the things you may need to do that hasn't been covered is accept a parameter that contains multiple values. Fortunately, that's what we're going to check out in the very next module.

Table-valued Parameters and Refactoring

Introduction

Welcome. In this module of Programming SQL Server Database Stored Procedures, we'll learn all about table-valued parameters and refactoring. Last time out, we covered stored procedure creation. These procedures started as rather basic efforts, but became more complex as they were developed. All the procedures so far have been created from scratch and use simple input-output parameters. Now we'll look at some of the more advanced concepts. We'll start by checking out a procedure another developer has written to insert multiple records in a single stored procedure call. We might need to make a few modifications. After we've sorted that procedure out, we'll see how a table-valued type, or TVP, can be used to pass multiple records to a stored procedure. With a TVP-enabled stored procedure in the bag, we'll see how to populate the TVP and pass it to the aforementioned stored procedure. Along the way, we'll investigate some other ways of processing multiple records and why a table-valued parameter might be the best option. Are you ready to see if you can conform to type? Wonderful! Let's say hello to Grace again.

Inheriting a Stored Procedure

Oh dear, things haven't been great for Grace since we saw her last. A crisis on another project meant she was moved off stored procedure development for a while, just as she was starting to enjoy herself. Worse, a junior developer was asked to write a stored procedure which allows multiple contact notes to be added to the database. The junior developer has something which sort of works, but it's a bit of a disaster. Anyway, the crisis is over on the other project, and Grace has been asked to pick up the contact note stored procedure. She decides to see how the procedure works before checking out the code. She kicks into SQL Server Management Studio and writes a test query to add some notes records to the most recently added contact. The original developer has told her multiple notes can be passed as a comma-separated list, so she makes sure she passes her notes in that format. She runs the test code to call the procedure, and it appears to work. A record set containing three records is returned, each containing the appropriate notes. Well, that's superb! Grace wonders what all the fuss was about. Looks like this procedure is done. Then she considers that the original developer himself didn't sound overly convinced with his work. Maybe the code is dodgy. She navigates to the procedure in Management Studio and opens up the code. Well, this looks a bit involved. What's going on here? Grace starts to walk through the code. The first thing she sees are the variable declarations. There's a TABLE variable and a VARCHAR. The use of the TABLE variable is obvious from the very next line. The STRING_SPLIT function is used to split up the note string via the comma separator, inserting

the split values into the table variable. If we assume a string of Hello,Goodbye was passed in, the table variable would currently hold two rows, one with the value Hello, another with the value Goodbye. Seems straightforward enough so far. Uh-oh, here's something Grace hasn't encountered in the wild before, a cursor. A cursor takes a collection of records and allows them to be processed on a sequential basis. It works by fetching the record at the top of the pile. If a record is found, the developer needs to store the values for the record in some variables. These can then be worked on in the script. Once the work is done, the developer tells the cursor to move to the next record. Another check is performed to determine whether the record was found and then the process starts all over again. Writing a cursor is quite involved. There are a number of distinct steps that need to happen. Firstly, the cursor must be declared, then opened. With the cursor now accessible, an attempt to obtain the first record held in the cursor must be made. A check is necessary to determine if the cursor returned a record, and if the record is returned, that record must be processed. Then the loop goes round again. An attempt to obtain the next record is made, another check to see if a record was found is made, and so on. This process continues until all records have been read and processed, after which the cursor can be closed and deallocated. That's quite a number of steps and seems like very hard work. The cursor in the InsertContactNotes stored procedure has been declared to use the @NoteTable table variable. It pulls the first value using FETCH NEXT with @@FETCH_STATUS used to determine if the cursor returns a record. If this returns a 0 value, it means a record was successfully obtained. This causes the WHILE loop to begin processing. The current note value read from the cursor is inserted into the ContactNotes table, and the code moves on to pick up the next record. The loop will continue picking up records and inserting them until @@FETCH_STATUS doesn't return 0. Once this happens, the loop will exit, as all records have been processed. The cursor is closed and deallocated to tidy things up. If these two closing actions are not performed, the cursor will continue to hold onto memory even after the procedure has completed. Whoa, that's a fair amount of code. Grace has to read up on cursors before she fully understands the code. She isn't convinced the original developer completely understood what he was doing, especially after multiple websites tell her cursors should be avoided. Grace thinks she can reduce the code and begins to look for an alternative approach.

Alternatives to Cursors

Grace is becoming a dab hand at searching the Microsoft website. She's discovered there are some alternatives to cursors out there, and she's also read up a bit more on why cursors are a bad thing. Let's look at the alternatives first. The first option is to use a WHILE loop instead of a cursor. This still processes records one by one, but it does reduce the amount of code the developer needs to write. It might even perform a little better too, as it doesn't have the overhead of setting up and closing the cursor. So an improvement, but it's still not great. Looping around records is something we expect to do in a traditional programming language, but working like this in a relational database is effectively throwing away the power of the database engine. Relational databases are designed to work on thousands of records at the same time, not on one record at a time. This is the set-based logic we came across

earlier in the course, and it allows the database engine to work on several records at the same time. This knowledge leads us on to the best of the options, inserting the record straight into the table in one go. In other words, inserting a set of records as one single batch. Grace has modified the code to do this. Just look how small the procedure is now. It's down to two statements, an INSERT statement inserting the STRING_SPLIT call straight into the ContactNotes table and the SELECT statement to return the notes for the requested contact. That's set-based logic at work. Grace is feeling pretty pleased with herself. She's ditched a load of useless code. She runs the procedure and tries out the test script again. It works really well. She wonders what will happen if a note contains a comma. Will it be inserted as part of the note? She adds some notes containing commas and runs the script again. Oh no, not good. Grace was expecting three notes to be inserted, but instead five notes have gone in. The code is just separating out the notes using any comma. Maybe there's a simple way around this. The input string could be split on two commas. Grace modifies the STRING_SPLIT call to do this, applies the change, and runs the test script again. Nice try, but no cigar. STRING_SPLIT only works with a single character. What a disaster. Actually, it isn't a disaster. This is not a good way to go about inserting multiple records via a stored procedure in SQL Server. For one thing, your input might be limited by the size of the parameter variable. The @Notes parameter in the stored procedure uses MAX, so has a theoretically unlimited size. But if there was a limit, the string will be cut off as soon as the limit was reached. Also, we put all of the responsibility of generating the input string onto the calling application. This really isn't good, but is there a better way? You bet there is!

User-defined Data Types

The better way involves two steps, the first of which involves creating a custom data type. SQL Server allows developers to define their own custom data types using the CREATE TYPE statement. There are a number of different types you can create, but we're only interested in the two most commonly used varieties, creating aliases for primitive types or creating custom data types. In the Contacts database Grace has inherited lies a table called ContactVerificationDetails. This contains two fields which would hold verification details, driving license number and passport number. We could create an alias to define the fields for the British driving license, which is 16 characters in length. It would be easy to create an alias for a 16-character string. All we would have to do is write CREATE TYPE dbo.DrivingLicense FROM CHAR(16) NOT NULL. The NOT NULL is optional, and if it is not specified, the type will support null values. It's better to specify null explicitly if that's your intention. Once the type has been created, you can use it in any script or table you like. In this script, a new variable of type DrivingLicense is created. The variable is assigned a value and then selected. The data is retained, just like the data in any other type. If the value is longer than 16 characters, it is cut off, and just the first 16 characters are returned. The type is underlined because the IntelliSense cache hasn't been updated. A quick Ctrl+Shift+R will sort this out. Custom types exist within individual databases via the Object Explorer in Management Studio. Expand the Programmability node, then Types, and then User-Defined Data Types to see these. If you have a type you no longer need, you can drop it using the DROP TYPE

statement, although you're only allowed to drop a type if no other objects are using it. We haven't created any other objects using the DrivingLicense type, so there are no dependencies on it. Running DROP TYPE for DrivingLicense would see SQL Server happily deleted. That's great, and it's good to know you can create type aliases. Unfortunately, it doesn't help Grace pass in multiple notes in a safe and consistent way. But don't worry, SQL Server has the technology. Did you notice the Object Explorer node called User-Defined Table Types? Bingo! That's what is going to help Grace out of her current mess. A user-defined table type is nothing more than a table variable. Because it has been explicitly defined, SQL Server allows it to be used like any other data type. The type can contain as many columns as you want, and you can insert as many rows into it as required, too. This gives us a consistent way of passing data into stored procedures and also ensures the calling application simply needs to create a data table using the same structure, then pass the data table as a structured parameter to the stored procedure. Grace has read up on table-valued parameters, TVP for short, on Microsoft's documentation site and believes this is the answer to her problems. She kicks up a new script and creates a ContactNote table type. This only contains one column called Note because that's all it needs. You can see it has been defined just like any table would be. Grace could have added more columns if she'd needed them. She runs the script, and the type is created. Refreshing the User-Defined Table Types node in Management Studio reveals the new type. Expanding it shows the Columns, Keys, Constraints, and Indexes. Yes, you can index and constrain the type if you wish and even define primary keys. Often, you won't need these extra features, but it's handy to know they have your back should the moment arise. Grace doesn't need any of the extra stuff. She returns to the InsertContactNotes store procedure and goes about modifying it to use the TVP. This is surprisingly easy. Grace changes the type of the @Notes parameter from VARCHAR(MAX) to ContactNote. The only other change is to replace the field name and the FROM STRING_SPLIT line with FROM @Notes. That's it. In just two lines, Grace has made the procedure more flexible and more robust, or so she thinks. She runs this so she can test it and hits a compilation error. The TVP must be declared with the READONLY option. What's this all about? Well, it's all about how the data is passed. The data held in the TVP is held in SQL Server's tempdb. This means the data only needs to be held in one place. It is not explicitly passed into the stored procedure with the TVP. Rather, it's passed by reference. The data is held in tempdb, a reference is passed into the stored procedure, and the data can be read from tempdb via the stored procedure. This is done to prevent copies of large sets of data from being continually created, which could have a negative effect on the system. A TVP could contain a lot of data, so you really don't want to be creating multiple copies of it. Grace adds the READONLY keyword to the end of the parameter and runs her script again. This time the build is successful. Before we leave the READONLY discussion forever, it's useful to know that there is a way to modify the TVP data within the stored procedure if you really need to. All you have to do is declare a variable using the TVP type within the stored procedure. With this in place, write an INSERT statement to insert the data into the variable. You can then modify the data in the TVP as much as you want. A nifty little trick.

Calling a Stored Procedure with a Table-valued Parameter

Grace doesn't need to modify the data, and she's already ran her CREATE PROCEDURE script, so she's ready to test it. The first thing her test script does is declare a variable to hold the data using the ContactNote table value type she created earlier. Then she inserts some data into the TVP. This is just a normal INSERT statement. There's nothing unique about it. Remember, a TVP is just a table. Grace makes sure she adds some notes with commas in so she can prove the new functionality handles them. With the data all ready to go, she can call the stored procedure. This is the typical stored procedure call we've seen throughout the course, with the @TempNotes variable passed as the value of the @Notes parameter. Grace takes a deep breath and hits the Execute button. Whoa! Total success. The stored procedure worked perfectly. Grace is really pleased. Not only has she removed a huge heap of poorly performing code, she's learned a much better way of passing in multiple records. If she wanted to, she could expand the procedure even further. The TABLE value type could be modified to include a ContactId column. If the stored procedure was then changed to just accept a contact table parameter, notes for multiple contacts could be inserted concurrently. That's one of the things I like about Grace, always trying to improve things. She's fulfilled the requirements for this stored procedure, but in this case, success has led to more work. You guessed it, some other developers have written stored procedures that aren't working anywhere near as well as Grace's InsertContactNotes efforts. Guess who management has called in to debug the faulty stored procedures? Grace tries to smile. It seems a database developer's work is never done.

Summary

It's fair to say that using a table-valued type is one of the more advanced SQL Server development topics, and it's something worth knowing about. It can really help you stand out from the crowd. The module began by showing the old-fashioned and badly performing way of passing in multiple records to a stored procedure using a comma-delimited string. We looked at the problems this could introduce and also saw why cursors are a bad thing. Cursor alternatives were investigated, including WHILE loops and table-valued parameters. After a quick look at creating aliases for simple data types, the module concluded by modifying a stored procedure to use a table-valued parameter. Grace's database skills have really come along, and she's feeling good. Next up, she's going to debug some existing stored procedures. Let's hope she can stamp those bugs out.

Debugging and Troubleshooting Stored Procedures

Introduction

Since Grace started working on the database, a lot of progress has been made on the contact system project. Grace has become the point of contact for all database development and has been helping some of the other developers with the creation of stored procedures. The database now contains quite a number of stored procedures as a result. Unfortunately, a number of bugs are present in some of these stored procedures, and guess who's been asked to fix them? We'll follow Grace's progress in this final module, Debugging and Troubleshooting Stored Procedures. Specifically, we'll find out how to use the PRINT statement to output simple messages, providing us with useful feedback when a stored procedure is running. Next, we'll check out the debugging tools available to us and how they can support our development work. We'll also look at how we can handle errors in code using TRY/CATCH blocks. Part of this investigation will demonstrate how the error functions built into SQL Server can inform us of any errors that have occurred and how return codes can be used. We'll finish up by looking at how to deal with transactions correctly and using defensive coding. Grace can't wait to fix these books. Grab your keyboard and hang on.

The Print Statement

Somebody has written a procedure to insert an address for a contact. Part of the code involves capitalizing the first letter of the street and city values. The code is pretty straightforward. It uses the UPPER function to make the first letter capitalized and then the LOWER function to ensure the rest of the string is in lowercase. Like the other insert procedures we've seen, the record is inserted and returned. Apparently, one of the test users has reported the string formatting in this procedure isn't working properly. Adresses are being inserted and then returned as different values. Sounds a bit weird. Grace writes a quick script to test the procedure so she can see what it's doing. Whoa! Those results are very odd. The house number and postcode looked good, but the street and city look incredibly weird. The street value passed was Downing Street in lowercase. It's been stored in uppercase. A D has been added to the front, and the last letter has disappeared. The city value is even worse. It somehow inserted the street value and done all sorts of weird things to it. This is definitely not what was requested. So Grace returns to the code and decides to bring the PRINT statement into play. Her first job is to convert the procedure into a standard T-SQL script. She comments out the CREATE PROCEDURe statements and brackets, adding a DECLARE above the parameters

to turn them into variables. After commenting out the AS BEGIN part, the INSERT and SELECT statements, and the END statement, she has reduced the code to the part that isn't working. The last thing she does is copy the value she passed in using her test script and paste these into the code. Now this code can run, and it will just perform the conversions on the value specified. This is a really useful technique when manually working through a faulty stored procedure. You reduce the code, fix the problem, restore the code, and test. When Grace first runs this code, it just returns a Commands completed successfully message. Not particularly helpful. She needs to add some PRINT statements to output what is going on. She decides to split out the two parts of each line into two separate PRINT statements. The first PRINT shows the value of the UPPER(LEFT call on the Street variable. The second displays the other part of the Street variable conversion, LOWER(LEFT. She does exactly the same for the City values. Now she can run the code and hopefully see what is going wrong. She runs the script, and the PRINT statement shows the string she asked the database to display. The PRINT statement can only return strings, but it is possible to chain strings together with the plus sign, as Grace has done here. The key thing to remember is PRINTs can only output strings, char, nchar, varchar, and so on. If you want to output other types, like numbers and dates, you need to convert these to strings in the PRINT statement. Okay, what's going on with the procedure output? The first conversion should return a capital D. Check. That looks good. Next should be the owning street of Downing street. That certainly isn't working. The last character has disappeared, and the letter has not been removed from the front. Grace looks at this line and sees the problem immediately. LEFT is being used, not RIGHT. She changes this in the PRINT statement, which should now return the correct value. What about the City value? Well, the left part looks good, L for London. The right part is completely wrong. It's returning part of the street. Looks like somebody did a straight copy and paste job. Very lazy. Grace replaces the Street values with City and switches the LEFT call to RIGHT. Another run shows both values are now returning the correct results. The PRINT statement has proven its worth. Grace switches the code to match the PRINT statements, then adds two new PRINT statements underneath the conversion showing the converted values. An execution of this shows everything is now working just as expected. Amazing, Grace! Now she can start restoring the code. But before doing that, she runs the code as a script. After restoring the INSERT and SELECT statements, the script is executed. We can see DOWNING STREET has been inserted, but it's all in uppercase. No need for the PRINT statement to fix this issue. It's obvious what's happening. The UPPER function has been used in the INSERT statement. Somebody really wasn't thinking when they created this procedure. Grace deletes the UPPER function call and runs the script again. Now the values are inserted as expected. The street of Downing street really should have a capital S on it, but the team have asked for just the first letter to be capitalized. That's something Grace might have to bring up with them later. Now that things are working as expected, it's time to restore the stored procedure. It's just a matter of restoring the commented out code and removing the DELCARE statement. The SELECT statement, which sets the variable values, is deleted; otherwise, every row's address will be set to 10 Downing Street. Now the procedure is ready to go. Grace has left the PRINT statements in for now just until she has confirmed the procedure works. You can safely leave PRINT statements in the code, but you really don't want to do this in production code. Much like SET NOCOUNT we saw

earlier in the course, you don't want to retain unnecessary metadata over the network. Make sure you remove the PRINT statements once you are done with them. Grace returns to the test script she originally created and runs it. There's a new version of Downing street in the table. Clicking into the Messages tab shows that the PRINT statement is still diligently outputting messages. Grace returns once more to the stored procedure code, deletes the PRINT statement, and recreates the stored procedure. One more run of the test script succeeds, and this time the Messages tab has not outputted any messages at all. That's one store procedure sorted out.

Debugging With Visual Studio

PRINT is a great way of inspecting how your code is running, but sometimes it just isn't enough. Now and again, you might sit and look at a problem for half an hour or an hour and not be able to see what is going wrong. When this kind of thing happens, you need to go to the next level of debugging, stepping through the code as it is running. Unfortunately, SQL Server Management Studio no longer has the tools to do this. Microsoft removed them as they were not heavily used. But don't fret, Visual Studio has everything you need to do some top debugging. If you don't have Visual Studio installed, just head on over to the download page. The Community edition has all the features of Visual Studio Professional, but it's free. You can develop database projects in Visual Studio and do lots of cool SQL Server-related things. But for now, we'll stick with learning how to use it for debugging. Just make sure you choose the Data storage and processing option when installing. If you've ever developed in a programming language like C# or Java, you are probably very familiar with the idea of stepping through your code. This is where you execute the code line by line, allowing you to determine what value is held by what variable at any particular point in time. By doing this, you can identify issues in the code that would otherwise be hard to find. That's what we're going to do with our stored procedures. Grace is now looking at a stored procedure that has been written to insert a role for a contact. It isn't working properly, and there are a few points where it could go wrong. When she runs it, something odd is happening. The first time she runs the code, she's told there's a FOREIGN KEY constraint error indicating the role or the contact record being used does not exist. If she executes the code again, a different error appears, a unique key violation, and it cannot insert NULL into the Contacts table message. Apart from the first execution, running the procedure again and again consistently delivers the same error message. Why was it different the first time around? Grace quickly writes two test queries, one to list out the roles in the database and another to show the roles assigned to ContactId 22. The Comedian role is present, but it has not been assigned to the contact. Grace deduces that the first execution added the Comedian role to the database, but failed to assign it to the contact for some reason. The following executions failed in a different way because the Comedian role already existed. She deletes the Comedian role record to test this theory and runs the script again. She was right. The original FOREIGN KEY error is back. Running the two test queries shows the records remain in the same state. The Comedian role has been added to the Roles table, but not linked to the contact. Grace is now very clear about what is going on. There are two distinct errors to solve. She deletes the Comedian role record again so she can sort out the first problem. As you have no

doubt realized by now, Grace is very good at figuring things out. She has looked into the debugging tools available to her and has learned she needs to use Visual Studio for debugging. Fortunately, she already has it installed and is in a position to start debugging. She chooses to continue without code as no project is needed when debugging SQL Server code. Now it's time to connect Visual Studio to the SQL Server. Grace clicks on the View menu and chooses SQL Server Object Explorer. Yep, you saw that right, the Object Explorer is directly accessible from Visual Studio. Grace needs to add her SQL Server to the Visual Studio Object Explorer. She right-clicks on the SQL Server node and chooses the Add SQL Server option. From here, she can add an SQL Server from any valid source. She is using a local instance, so she expands the Local option. For some reason, Visual Studio always seems to display the same local instance twice. Grace chooses the first listing of the instance, leaves all of the default options set, and click the Connect button. Voila! Grace's database instance is now accessible in Visual Studio. From this point, everything works the same as in Management Studio's Object Explorer. Grace expands the Databases node, goes into the Contacts database, and expands the Programmability node until she can find the InsertContactRole stored procedure. This is the one that she wants. Visual Studio makes debugging very simple. Grace simply right-clicks on the stored procedure name and clicks the Debug Procedure option. A dialog appears asking for parameter values. You can enter any values you like here, and you won't be stuck with them either. Visual Studio will generate a test script for debugging purposes once these values have been entered. Grace knows ContactId 22 the Comedian role demonstrate the problem she is facing, so she enters those values and clicks OK. And abracadabra, like magic, the test script I was just telling you about appears. The first line is highlighted in yellow, denoting it is the line about to be executed. Grace can now decide how to execute the code. She can run the entire script, step over or into the code line by line, or stop running the code. The various options can be found in the Debug menu. If Grace chooses to step over the code, the debugger will not display the code within the stored procedure. It will step over it and will move to the next statement in the current script. Step Out could be used within the stored procedure. If we were debugging within the stored procedure and decided to step out, the debugger would jump back out to the calling script. The option Grace wants to use is Step Into. Using this, the debugger jumps from the current script to the piece of code it is calling, whether that's a stored procedure, a function, a trigger, or any other programmatic object. Grace hits F11 on her keyboard to step through the code. Once she presses F11 on the EXEC line, the code for the stored procedure appears. Grace can now use F10 for Step Over or F11 for Step Into to walk through the code. The code for the procedure is pretty simple. It declares a transaction, inserts the new role, and inserts the ContactRoles record, linking the contact to the role. It finishes off by returning the roles for the contact. The debugger starts by hitting the BEGIN TRANSACTION line. Grace hovers her mouse pointer over the @RoleTitle parameter. This shows the current value of that parameter. You can hover over any parameter or variable to see the current value. If you prefer not to use hovering, it's possible to add watches to certain variables. You do this in the Watch window, which you can display by going to the Windows submenu of the Debug menu. Once the Watch window is visible, simply type the names of the variables you want to watch, and their values will appear in the Watch window. This is a great way of seeing how values change as your code is executed. Grace moves past the INSERT Roles line. That succeeds!

Nothing wrong there. The RoleId is picked up too. The value is updated in the Watch window. Before she steps past the INSERT ContactRoles line, she checks the values in the Watch window. Grace thinks the error is occuring on this line. It's the table involved in the FOREIGN KEY constraint error. After looking at it for a moment, she spots the problem. The values are being inserted the wrong way around. @RoleId is being inserted into the ContactId column, and @ContactId is being inserted into the RoleId column. Suddenly it all makes sense. No wonder she was seeing a FOREIGNforeign key error. The debugger has done its job. Grace hit the Stop button to exit the debugger and switches back to SQL Server Management Studio. In the stored procedure code, Grace switches the variables around and applies the changes. After running the test script again, the INSERT succeeds. She can see the Comedian role has been added and then correctly linked to the contact in the ContactRoles table. This is good stuff. But Grace hasn't forgotten that the procedure was failing differently the second time it ran. She runs it again, and the unique and null letters are our back, back, back. The Visual Studio debugger has done its job. It's helped Grace walk through the code and identify the initial error in the code. The stored procedure now works if the role doesn't already exist, but it still bombs out if the role does exist. Looks like Grace will need to amend the code further to sort this problem out.

Handling Errors with Try/Catch

Everything Grace has done so far has been aimed at identifying problems with the code. The PRING statement told her what was happening on a complete run-through of the code, whilst the debugging features of Visual Studio helped her step through the code line by line. Now that the code that has been written is working, even if it is somewhat incomplete, it needs to be modified to be able to handle errors. Unfortunately, despite our best efforts, we cannot always guarantee that the code we write will perform correctly. Factors we may be unaware of could affect the code, permissions accidentally being removed or a disk being unavailable, for instance. On top of that, factors we are aware of could also have a negative effect. For instance, the Roles table has a unique constraint on it. This prevents the same role name from being inserted into the table more than once, and it's the reason the INSERT ContactRoles procedure is throwing the unique key violation error. We can deal with unexpected errors in a variety of ways. The first step is to add error handling to the code. We write the code in such a way that if an error occurs we can capture it and output a useful error message, allowing the support team to fix the issue. The T-SQL language provides a few options to help us manage errors at runtime. Until SQL Server 2005 came along, the nefarious GOTO statement had to be used to handle runtime errors efficiently. The developer defined an ErrorBlock, and after every statement that could fail, the value of the @@ERROR system function needed to be checked. If this was 0, everything was okay, and the code could proceed. If something had gone wrong, @@ERROR would return an error number, which could be used to return an error. Those of us of a certain age will know GOTO should be treated like a vampire, hold a cross up to it whenever you see it. GOTO was a bad paradigm at the time and remains so today. It leads to what was infamously called spaghetti code, where it was very difficult to follow the flow of code as it was jumping all over the

place. Happily, SQL Server 2005 introduced a much better way of handling errors, the TRY/CATCH block. This was brought over from languages like C# and allows the developer to wrap code that could fail in a TRY block. If something goes wrong whilst the code is executing, an error is raised and causes the code in the CATCH block to execute. The TRY block contains code that tries to execute, and if it fails, the exception is caught and handled in the CATCH block. Grace has decided to add a TRY/CATCH block to the INSERT ContactsRoles stored procedure so things like unique exceptions will be handled correctly. She adds a BEGIN TRY statement to the code and ends this after COMMIT TRANSACTION. She declares BEGIN CATCH straight after END TRY. These effectively form a statement by themselves. You might notice a semicolon is not specified after END TRY, and you're not allowed to specify one either. Within the BEGIN CATCH, you can do whatever is required to correctly handle the error. In this block, Grace does two things. She outputs a PRINT message to return the details of the error and returns a failure value. The PRINT statement is interesting. It uses functions called ERROR_PROCEDURE and ERROR_MESSAGE. There are other useful error functions that can be used too. These include ERROR_LINE, which returns the line the error occurred on; ERROR_NUMBER, which returns the number of the error; and ERROR_STATE and ERROR_SEVERITY, which return additional information about the error. The two functions Grace has used provides very useful information. ERROR_PROCEDUREoutputs the name of the stored procedure or trigger in which an error occurred whilst ERROR_MESSAGE returns a string containing the error that caused the CATCH block to be executed. These error functions are very useful and can help identify runtime problems in your code. The second line in the CATCH block utilizes something we haven't discussed, return codes.

Return Codes

When a stored procedure executes, it always returns an integer value. Even if this isn't assigned, it is always returned. A successful execution always returns 0. If any other value is returned, that value has been generated inside the code of the stored procedure. Let's take a look at the RETURN line Grace has added to the CATCH block. This returns -1. She has done this so any program calling the stored procedure can check the return value. If it is -1, the calling program will know it needs to output an error. Underneath the CATCH block is another new line, RETURN 0. If this line is reached, it confirms a successful execution. RETURN, if specified, is the last thing to execute in a stored procedure. If the CATCH block is entered, the RETURN -1 line will be fired, ending the stored procedure execution, so the RETURN 0 success line will never be reached. Let's see how this works In practice. Grace has applied the changes to the stored procedure and has now returned to her test script. She removes the SELECT lines from the test script and declares a new variable, @RetVal. She assigns this to the execution of the stored procedure. Finally, she writes a PRINT statement to output the return code. Note the CONVERT statement in the PRINT statement. Only strings can be outputted by the PRINT statement, so an integer must be converted to a string before it can be displayed. You can return any valid integer as the return code of a stored procedure. For example, your code might run 10 different checks, each of which would return a different return code if a particular check failed.

Whatever program is calling the stored procedure could then perform a particular action based upon the return code. Grace runs the updated code, and the unique constraint error is indeed handled correctly. We know this because the error is displayed in black. Unhandled errors are outputted it in red. Worryingly, an unhandled error still exists. Grace will deal with that in a moment. The last line shows the return code, which is the expected value of -1. So, the TRY/CATCH block is returning correctly, and the return codes are working as expected, but what about the unhandled exception?

Handling Failed Transactions

The stored procedure Grace is working on uses transactions. There is a very simple rule to follow with transactions. Whenever you open a transaction, you must close it either by committing it or rolling it back. If an error might occur when a transaction is open, you need to ensure the transaction is handled correctly. Grace inspects her code and sees she made an error when she modified it. The main block of code starts with BEGIN TRANSACTION. This happens before the BEGIN TRY. But the COMMIT TRANSACTION is closed within the TRY block. If an error occurs which causes the CATCH block to be executed, the COMMIT TRANSACTION will never run, meaning the transaction will never be closed. This is really bad as it can block other users from being able to access the table. Transactions block access to the table whilst they update records, so they should be held open for the minimal time possible. Grace needs to do two things. move the BEGIN TRANSACTION line into the TRY block and then add code to roll back the transaction if something goes wrong. Looking at the error in the test script demonstrates the problem. The code has executed and left transactions open. These open transactions could block access to the Roles or ContactRoles tables, which would require a manual rollback with the ROLLBACK statement. Grace modifies the code. After she moves the BEGIN TRANSACTION line, she starts to modify the CATCH block. Above the PRINT statement, Grace adds a check to see if any transactions are open. There is a system function to do this called @@TRANCOUNT. This returns the number of currently open transactions in the session. The statement to roll back these open transactions is rather imaginatively called ROLLBACK TRANSACTION. If @@TRANCOUNT returns a value of 1 or higher, ROLLBACK TRANSACTION will roll back the database, resetting the changes implemented by the code within the transaction, leaving everything in a good state. There's actually one further change that can be made. We've just discussed keeping transactions as short as possible. There's no need for the SEELCT statement to form part of the transaction. This can run once the transaction is complete, Grace lifts the COMMIT TRANSACTION above the SELECT statement ,which means the transaction is now open for the minimal time possible. Now that all of the code is in place, Grace can try it out. After applying the stored procedure changes, she runs the test script again. Unexpectedly, an unhandled error is returned. Grace panics until she realizes the error happened because a transaction was still open from the last faulty execution. This execution has rolled back the old transaction, and if Grace had scrolled across a little, she would have seen the current transaction count is now 0. With the transactions

all closed, she runs the test again. At last, all of the unhandled errors have been resolved. All that's left to do now is modify the stored procedure so it works properly.

Defensive Coding

It's a good thing that Grace has handled all of the errors in the stored procedure, but there's still one glaring problem. It doesn't actually work. Let's see what the procedure should do. First, it should check if the RoleTitle passed in exists. If it does exist, the RoleId should be stored in the @RoleId variable. If the RoleTitle does not exist, it should be inserted with the new identity value assigned to the @RoleId variable. At the moment, the code is simply attempting to insert the RoleTitle every time it has passed in, which is why the unique check on the table is causing the CATCH block to execute. With the RoleId correctly obtained, the next check is to ensure the Contact and RoleId combination doesn't already exist. If it doesn't exist, the INSERT INTO the ContactRoles table can be executed. The code then finishes up by returning the role list for the contact. There's a mistake here too. That filter on RoleId will stop all roles for the contact from being returned. It has to go. Grace makes the change for the role INSERT first. This requires an IF EXISTS check. Grace actually does this in a very nice manner. Instead of using IF EXISTS, she uses IF NOT EXISTS. This eliminates the need for an else branch. She can just write code to perform the insert. She changes the scope identity call to pick up the ID from the table using the RoleTitle. The penultimate change is to add an IF EXISTS check before the ContactRoles INSERT. Grace again saves herself a bit of typing by using the IF NOT EXISTS check again. The final change involves removing the errant @RoleId filter from the SELECT statement, which Grace spotted earlier. That should be it, one fully functional and error handling stored procedure coming up, hopefully. Grace returns to her test script and runs it with her fingers crossed. Surely everything will work this time. It does! Stan Laurel has been linked to the Comedian role. Yay! She runs it again and again and again. But no matter how many times she runs it, only one record is returned. She executes one more check, changing the role to Actor. She runs it, and just the two records are returned, Comedian and Actor. Running it again and again and again still returns two records. Wow! Grace puts her feet up. She still has a lot of work to do on this database, but Dolly's Dolls have been delighted with her progress. They've offered her a new role as a database developer, which comes with a higher salary. Grace is thrilled and looks forward to completing work on the rest of the database. Nice work, Grace.

Summary

That completes an epic journey through stored procedure debugging and troubleshooting. We've certainly covered a whole heap of functionality. The PRINT statement is great for obtaining immediate feedback from your code as you are writing and testing it. You can use it at various points in your code to identify potential issues. However, if PRINT isn't quite doing it for you, the debugging tools available via Visual Studio might just come to your rescue. The debugger is really powerful, allowing you to keep a watch on variables and parameters and to step through your

code line by line. The debugging tools are seemingly not widely used for database programming, but they are really useful, so keep them in mind. Sometimes exceptions can occur in your code that you could never envisage, network access disappearing, for instance. Likewise, things like check constraints can cause errors to occur in your code if they aren't handled. To avoid these problems, use TRY/CATCH blocks. These allow you to wrap potentially problematic pieces of code in a TRY block and handle any exceptions they raise in a CATCH block. The code inside the CATCH block can be used to retain useful information using the built-in error functions and return codes. These pieces of functionality can help inform the calling program what went wrong when the procedure was called. Another key part of error handling is dealing with open transactions. Leaving transactions open is a really bad thing as it could lock other people and applications out of the database. It could even lead to lost data. You don't want to be the cause of this. You might be made to stand in the corner wearing the cone of shame. Avoid this by checking for open transactions using @@TRANCOUNT. We completed our investigation into error handling by looking at defensive coding. This involves writing code directly into a stored procedure which checks for a possible error condition and works around it if necessary. This means the error can never occur as the code will deal with the condition and follow a different path. That's all folks! You have successfully completed this course, Programming SQL Server Database Stored Procedures. I hope you now feel comfortable writing and testing stored procedures. I'm not a big fan of social media, but you can reach me on LinkedIn. So, if you have any questions, look me up on there, and I'll be happy to help. Thanks very much for watching. I hope you enjoyed the course. Enjoy programming.