CHAPTER 3

Database Basics

In this chapter, we are going to look at databases: what they are, the types available to us, and how we manage them. We'll also take a look at some basic server settings, including security.

What Is a Database?

Think of a database as a container, with different compartments. Have you ever seen a neatly partitioned lunchbox? It might have a section for your sandwiches, a section for your salad, and a section for a tasty chocolate bar. Databases are very similar, except they don't have anything as tasty as a chocolate bar! Databases instead contain database objects. Some of the more common objects you will find in a database are the following:

- **Tables**: Used to store data.
- **Views**: An SQL statement that acts as a table. You'll learn about SQL statements throughout this book, and you'll actually write our first statement in this chapter (remember, SQL stands for Structured Query Language).
- **Stored procedures**: These execute code within your database and can be used to modify data in your tables. They have many other uses, too.
- **Functions**: A piece of code that performs a particular task, (e.g., returning the last day of a specified month).

There are lots of other types of objects, but in daily use you'll come across the items in this list on a constant basis.

Types of Databases

SQL Server uses two types of databases:

- **System databases**: These are databases that SQL Server requires to operate correctly. We'll take a look at these in a moment.
- **User databases**: These are created by users of SQL Server, and store any data required by those users or the organizations that own the servers. We'll be working with user databases for the majority of this book.

System Databases

There are five system databases.

- **Master**: This database stores all of the information needed for your SQL Server to function correctly. Logins and server configurations are two of the items stored here. If this database is unavailable your SQL Server will not start and you will not be able to log in to it.
- **MSDB**: We met the SQL Server Agent in **Chapter 1**, which acts as the scheduler for SQL Server. The **MSDB** database stores details about the jobs executed by the scheduler, along with job history. Backup details and history are also stored in this database. If something goes wrong with this database, your SQL Server will function, but certain aspects of it, such as scheduled jobs and backups, may fail (e.g., backups may complete, but won't be recorded).
- **Model**: The **Model** database is a template database, and every user database created starts as a copy of the Model database.
- **Resource**: This is the only database you cannot see via the Object Explorer in SQL Server Management Studio. It stores system objects, such as the **sys** tables (you'll see more of the **sys** tables as you work through the book).
- **TempDB**: You can think of this database as a scratchpad, accessed by all users of the SQL Server. Whenever you do something in SQL Server that requires a temporary object to be created, it will be created in this database. This database is deleted and recreated every time the server is restarted.

Note that if you have sysadmin permissions (more on permissions in Chapter 20), you can make changes in any of the preceding databases except for **Resource**. I cannot stress enough that you should never modify any of these databases, with the possible exception of **Model**. As I mentioned earlier, **Model** is used as a template for all user databases you create (more on this in just a moment). If your organization has certain standard tables (e.g., an Auditing table), it could be added to the **Model** database. Any new databases will then automatically contain this table.

Caution **Model** is the exception to the rule: never modify a system database!

To see the system databases, open SSMS and log in as we did at the end of Chapter 2. Find the **Databases** node and expand it, then expand **System Databases**. As Figure 3-1 shows, you'll see all of the system databases there except for **Resource**.

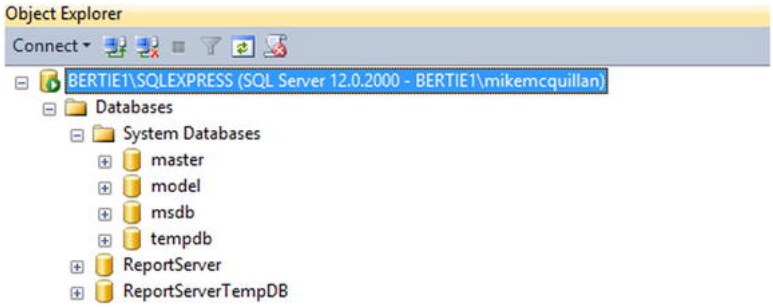

Figure 3-1. *The system databases in Object Explorer*

The eagle-eyed among you may have noticed two `ReportServer` databases, sitting below `System Databases`. What are these? When I installed SQL Server in Chapter 2, I installed SSRS, too. SSRS uses these databases to store reports and data used during report generation. Interestingly, they exist as normal user databases, not system databases. You shouldn't modify anything in these databases if you are using SSRS, though. Doing so may break your SSRS installation.

User Databases

Now we come to the real start of our journey into SQL Server time and space. As a developer, you will spend most of your time creating objects within user databases. But you won't actually spend that much time creating them; after all, once a database is there, it's there! There are lots of options for creating databases but we'll stick to the simple path, and then afterward I'll give you a brief overview of the other things you can do.

There are a number of ways to create a SQL Server database (and as we continue through the book, you'll find there are multiple ways of doing just about anything). We'll take a look at two options:

- Creating a database through SQL Server Management Studio (SSMS)

- Creating a database using scripting

We'll also explore when we might use either of these options.

Creating a Database Using SSMS

At last! It's time to start using SQL Server. If you don't already have SSMS open and you have a Start Menu or a Start Screen, type in *SQL Server*—you should see **SQL Server 2014 Management Studio** in the list, depending upon the version of SQL Server you installed.

Open this up and you'll be presented with the login prompt shown in Figure 3-2.

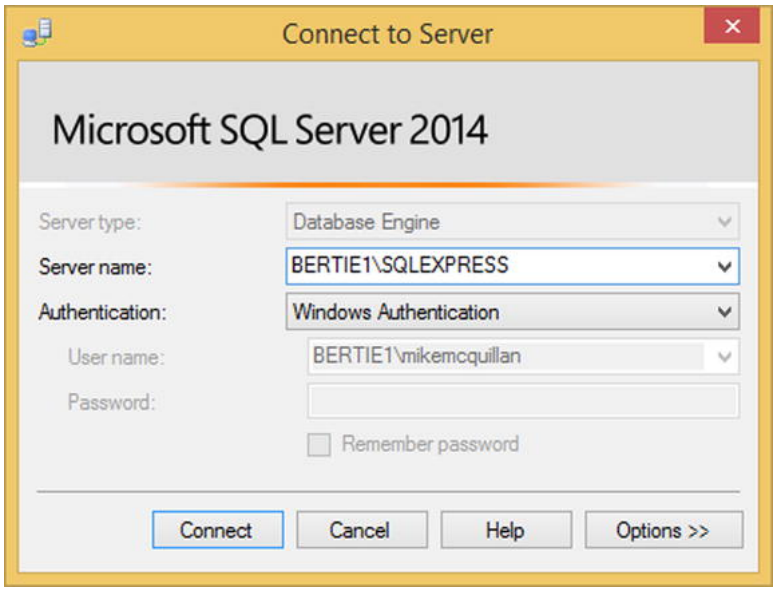

Figure 3-2. *The SQL Server login dialog*

You'll see there are a few drop-down boxes within the login prompt. You should populate these as follows:

- **Server type**: Type *Database Engine*. You could also log in to SSIS, SSRS, and SSAS. Database Engine is the Relational Database Engine.
- **Server name**: The name you gave the server when you installed SQL Server (e.g., *BERTIE1*). If you installed the server as a default instance, you can just type *(local)* (including the parentheses) or a full stop/period here. If you installed the SQL Server as a named instance, as in **Figure 3-2**, you need to provide the full name of the instance, which is in the format *SERVER NAME\INSTANCE NAME*, (e.g., `BERTIE1\SQLEXPRESS`). For more information about default and named instances, refer to Chapter 2.
- **Authentication**: There are two types of authentication available; *Windows Authentication* and *SQL Server Authentication*. As I've said, my recommendation is to never use SQL Server Authentication, as you are opening up another potential route into your server. Windows Authentication integrates with your Windows server accounts. If your server runs as part of a larger network domain, any user account within the network's Active Directory can be granted access to SQL Server. This makes permissions easy to manage as you can allocate permissions to a group in Active Directory, and then just add users to that group as necessary. SQL Server Authentication uses individual usernames and passwords within SQL Server itself. So you would create a user account for yourself through SSMS and use that to log in (you can map logins to a SQL Server user account should you wish). If you followed the instructions in Chapter 2, you should only have configured Windows Authentication, so that is the one to choose.

If you did configure SQL Server Authentication and want to use a SQL Server account, the **User name** and **Password** boxes will become enabled. Enter the default user credentials:

- **User name**: **sa**
- **Password**: whatever you specified during installation

sa stands for System Administrator. This is a well-known account in SQL Server, and as a result could be used to compromise your server, as a hacker only needs to guess the password.

Once you are happy with everything, click **Connect** and you should be logged into SQL Server. If the login fails a prompt will appear; correct your login details and try again. Once login is successful the login prompt disappears and you will see your server name over on the left in the Object Explorer. You may learn to love the Object Explorer!

Click the + symbol next to your server name. A list of items will appear beneath it. The first item you can see is **Databases**. All available databases on the server are listed below this element. Click the + symbol next to **Databases**. An item called **System Databases** should appear underneath (other items may also appear; just ignore them).

Expand **System Databases** and you will see **master, model, msdb** and **tempdb** listed. Apart from being under the **System Databases** branch they don't actually look any different to user databases.

We're going to create our database in a folder called `c:\temp\sqlbasics`. If this folder doesn't exist on your computer, please create it before we proceed.

THE SQLBASICS FOLDER

As you work through the book I'll be asking you to save files you create in SQL Server Management Studio. These files, also known as scripts, will allow you to create and destroy the database we create together. The book will refer to a folder called `c:\temp\sqlbasics`. Feel free to use a different folder if you wish, but don't forget to save the files to it!

Right, let's create that database. Right-click the **Databases** node and a pop-up menu will appear as per **Figure 3-3**. Choose **New Database**.

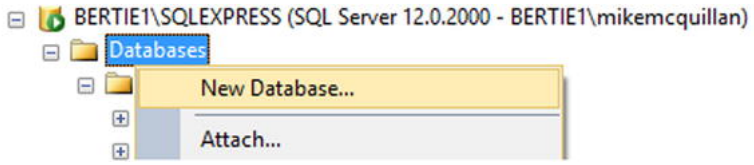

Figure 3-3. *Creating a new database in SSMS*

A dialog will appear. The first thing we need to do is provide a database name. This may sound obvious, but database names are very important! The name should accurately describe the purpose of the database. I've seen many databases where the name of the database did not correlate to its purpose. This can make things difficult for developers and database administrators (DBAs), especially if they are new to the job. When you are creating objects or writing code, follow this maxim:

Always think about the next person.

The next person could be you! You need to be confident that you could come back to your code in a few months and pick it back up with minimal difficulty. We'll continue to reinforce this maxim throughout the book.

DBAS AND DEVELOPERS

Database Administrators—or DBAs for short—are the people tasked with ensuring SQL Server keeps on running. A DBA maintains SQL Server, fixes it when things are going wrong, and backs it up. A DBA is often responsible for testing and releasing your scripts, too.

A developer writes SQL code that is released onto SQL Server. Quite often, the same person will be a DBA and a developer (this has happened to me). Larger organizations will always have a team of DBAs dedicated to managing their SQL Server instances.

Back to database creation. We are going to create a simple address book database, which we'll build up throughout the rest of the book, so let's call our database **AddressBook**. Database names can include special characters, numbers, and even spaces. However, using these special characters and spaces is not a good idea. It leads to more typing and may cause issues with any systems that need some level of backward compatibility. As an example, look at these two queries (don't worry about what they do for the time being):

```
SELECT * FROM AddressBook.dbo.Contacts

SELECT * FROM [Address Book].dbo.Contacts
```

The second example uses the name **Address Book**, with a space. The existence of this space means we must wrap the name in square brackets. If we didn't do this, the query would not work.

In short, use simple and obvious names for your databases, and don't use any special characters—just capitalize each word in the name. This is known as CamelCase.

Type *AddressBook* into the **Database** name box. The next box allows you to specify the **Owner** of the database. This is set to *<default>*, which means the person creating the database is the owner. This translates to **dbo** within the database, of which more later. It is not particularly common to change the owner of a database, unless your organization has specific security requirements, so don't worry about this too much.

When you typed in the **Database** name, you may have noticed the **Database** files section underneath auto-populated (just like Figure 3-4). You should be looking at a table with columns for Logical Name, File Type, Filegroup, and so on. This table contains the files that make up our database.

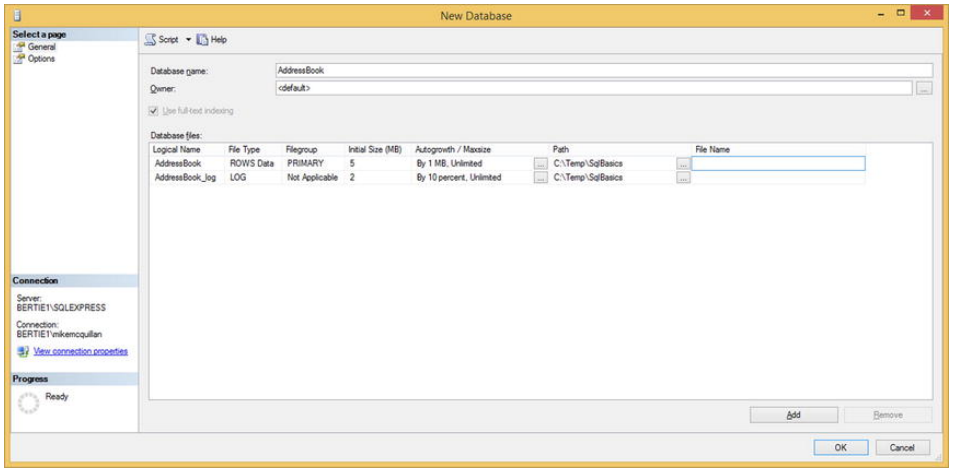

Figure 3-4. *The New Database dialog*

We'll take a quick look at what each of these columns is for:

- **Logical Name**: The name assigned to the file for user identification purposes. This name is used to restore the file, for example.
- **File Type**: There are a few file types, but mainly you'll deal with data files and log files. In the list you can see, one data file (ROWS Data) and one log file (LOG) are present.
- **Filegroup**: A filegroup can house multiple files, and is used to split up a database across disks. These are used for performance reasons. We'll touch on them shortly.
- **Initial Size (MB)**: How big the files should be when they are created. You need to figure this out by assessing how much data you expect the database to contain. Be aware that the file sizes you can see in Figure 3-4 may differ from the default file sizes assigned to your database files.
- **Autogrowth/Maxsize**: If the file reaches the initial size specified, SQL Server will use the values you specify here to autogrow the file, and also to limit the size of the file. You can set Maxsize to unlimited—more on this in the "Creating a Database Using T-SQL" section later in this chapter.
- **Path**: The folder where the database files will be created.
- **File Name**: The actual name of the physical file. This will be created automatically from the database name (it usually matches the logical name).

Click **OK** and after a couple of seconds the dialog will disappear (if you encounter an error, make sure you created the `c:\temp\sqlbasics` folder correctly). If you look under **Databases** in Object Explorer now, you'll see you have an **AddressBook** database!

Excellent, you have your first database. Let's see the files SQL Server created. Open Windows Explorer and go to `c:\temp\sqlbasics`. You should see two files (shown in Figure 3-5):

- `AddressBook.mdf`
- `AddressBook_Log.ldf`

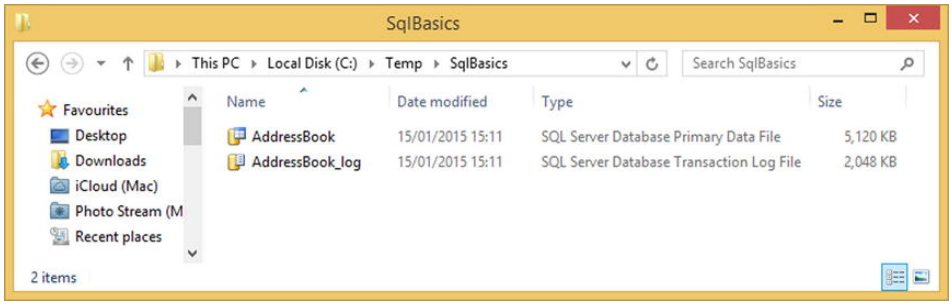

Figure 3-5. *The AddressBook database files*

It is highly likely that most of the databases you come across will use this simple structure. The MDF file is a Master Data File, and it is the file where SQL Server will store your data. The LDF file is a Log file that SQL Server uses to keep your database consistent (we will look at the log in Chapter 15).

There is a third type of file, the Secondary Data File (NDF). You need to create a filegroup if you want to use a secondary data file. You would use secondary data files when you want to maximize performance. You could put some of your tables in the Master Data File on Hard Disk 1, some files in a Secondary Data File on Hard Disk 2, and some files in another Secondary Data File on Hard Disk 3. This would maximize data reading and writing to/from the database as three hard disks could be used at the same time. You could also choose to back up or restore individual data files using their filegroups. This kind of configuration is not uncommon, but it is a bit more work to configure and manage. The performance gains are worth it if you have such a requirement.

If you are in doubt, always go with a simple database structure first. You can modify the structure in the future as your database grows.

Dropping a Database Using SSMS

Now that we've created our database, let's delete it!

Dropping a database using SSMS is scarily easy. Right-click the **AddressBook** database and click **Delete** in the pop-up menu that appears. A dialog appears, prompting you to confirm the deletion (Figure 3-6).

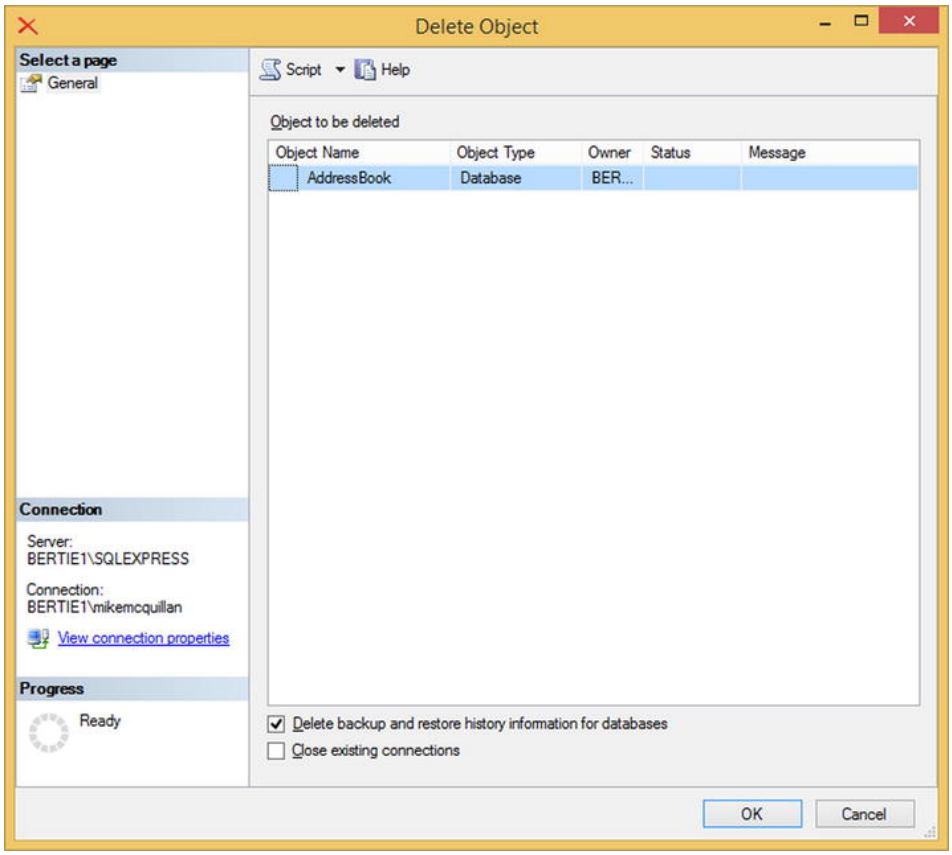

Figure 3-6. *Dropping a database using SSMS*

Note the two checkboxes at the bottom of the dialog shown in Figure 3-6:

- **Delete backup and restore history information for databases**: This will clear out the relevant history from the MSDB database. It is checked by default.
- **Close existing connections**: This is unchecked by default, and for good reason. If this box is unchecked and you attempt to drop the database, you will be prevented from doing so. You shouldn't use this option in production. However, quite often you will want to drop a development version of a database, and it is possible developers will be attached to the database. Checking this box in that scenario would be a valid use of this option.

You shouldn't need to change the defaults to drop the **AddressBook** database. Just click **OK** and the database should disappear from Object Explorer. If you browse to `c:\temp\sqlbasics`, you should find that the **AddressBook** MDF and LDF files have also disappeared.

Unless you had a backup, that database is now unrecoverable! So be really careful when dropping databases.

Okay, so now you know how to create (add) and drop (delete) databases using SSMS. However, SSMS is not always an option for us. What if you've been asked to build a database that can be deployed on different environments? You might have to create the database on a User Acceptance Testing environment, and then deploy it to a production environment. It is common to deploy to three or four environments before going into production. If you are using SSMS to manually create your databases and related objects, what are you going to do? Put a document together listing which buttons and boxes the deployment team needs to click and complete? Good luck with that!

No, you can't use SSMS for multi-environment deployments. For that, you need scripts. Scripts are the magic potions of the SQL Server world. You can manage just about anything from a script, and it can be executed in the blink of an eye on multiple servers.

Creating a Database Using T-SQL

In a moment, we'll create our first T-SQL script—to recreate our **AddressBook** database. Just before we do, it's worth understanding exactly what T-SQL is. SQL stands for Structured Query Language, and it is the standard language used to query databases—it is used by every major RDBMS (Relational DataBase Management System, a logical extension of a DBMS). It forms the basis of most RDBMS languages.

T-SQL is SQL Server's version of SQL. Oracle, a competing RDBMS, has a language called PL/SQL. You will find that many SQL statements you write for SQL Server will work in Oracle, and vice versa. But each RDBMS also implements many custom features, such as .NET integration in SQL Server and the `FOR LOOP` in Oracle.

On to the script. SSMS isn't just a tool used to manage databases; it is a fully fledged code editor, too. We'll use it for all of our scripting throughout the book. To create your first script, click the **File** menu, choose **New**, and click **Query with Current Connection** (alternatively, press Ctrl+N your keyboard or click the **New Query** button on the top menu).

An empty code window opens. What a world of possibilities in front of us!

Let's create a T-SQL script to create the database. As we are creating a database, this will be a Data Definition Language (DDL) script. There are two types of SQL statements. DDL statements are statements that create, modify, or delete database objects like tables and stored procedures. Data Manipulation Language (DML) statements work on data only, allowing you to select, insert, update, or delete data for your tables.

The DDL statement we use to create a database is `CREATE DATABASE`. This statement is huge and has lots of options. We are not going to look at all of these options. We'll just build up the statement we need bit by bit. Start by typing this:

```
CREATE DATABASE AddressBook;
```

Do you think this will work? Try it! Press F5 to run the statement. As Figure 3-7 demonstrates, you should see the message **Command(s) completed successfully**:

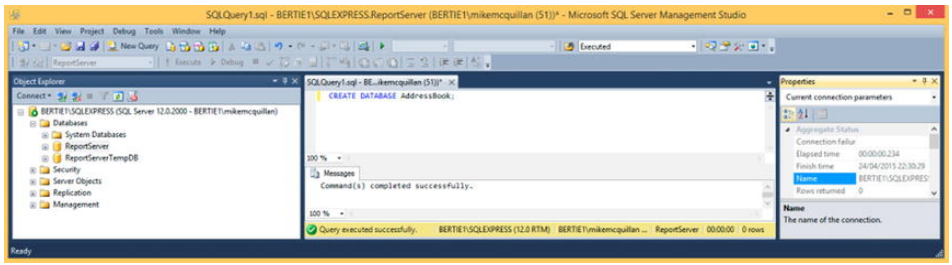

Figure 3-7. *Basic `CREATE DATABASE` T-SQL statement*

Amazing; it does succeed. Yes, those three simple words are all you need to create a database with the default values (from the **Model** database, referenced earlier). When we opened up the **New Database** dialog earlier and created the database, this is pretty much the statement SQL Server executed in the background.

If you refresh the **Databases** node of your Object Explorer, the **AddressBook** database will be visible. Cool!

THE SEMICOLON

You might have spotted that our `CREATE DATABASE` statement ended with a semicolon. This is put here to denote the end of a statement. SQL Server doesn't force you to use this at the moment other than in a few unique cases, such as declaring Common Table Expressions (CTEs). It will become required in a future version, so it's best to enter into the habit of finishing all of your statements with a semicolon now. Then you'll be ready for the brave new world! You don't need to put a semicolon after the `GO` command—if you do the code won't work. This is because `GO` is technically not part of your code; it separates your code into batches.

Out of curiosity, do you think we can create another database called **AddressBook**? Press F5 to rerun our statement. This time we see an error message—in red, just like a teacher marking our homework (see Figure 3-8)!

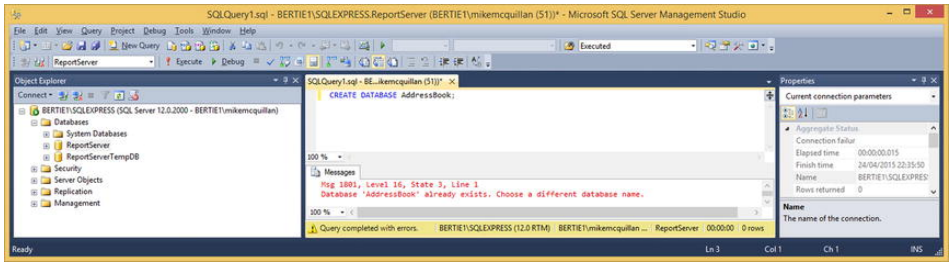

Figure 3-8. *"Database already exists" error*

So, we cannot create two databases with the same name. This makes perfect sense, as how would you be able to tell them apart?

The database we have at the moment is not suitable for our requirements, as we want to customize the file names and sizes. We therefore need to drop our database. Replace the `CREATE DATABASE` statement with:

```
DROP DATABASE AddressBook;
```

Press **F5** to run it. The database will be removed, as Figure 3-9 shows.

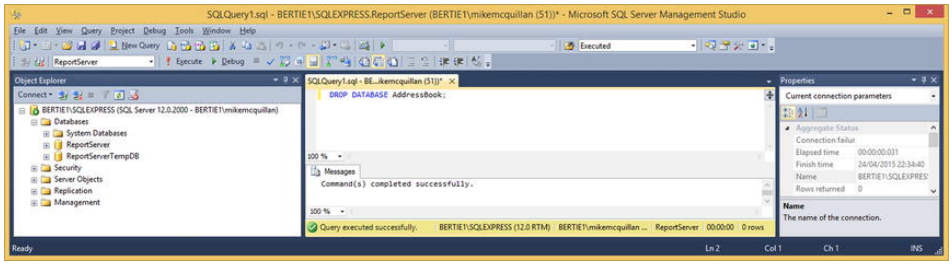

Figure 3-9. *DROP DATABASE in T-SQL*

As you can tell, it is as easy to drop a database as it is to create one. Be careful with the DROP command!

Now, before we write our complete `CREATE DATABASE` statement, let's consider what we would like to create:

- A database called **AddressBook**
- One data file, 10MB in size
- One log file, 2MB in size
- The data file should autogrow by 50%, and should be able to grow as large as it needs
- The log file should autogrow by 2MB, but should be limited to 100MB in size

MAXIMUM FILE SIZE (MAXSIZE)

There are many schools of thought about the maximum size of database and log files. My view is you should not set size limits on files; you should always let them grow as large as needed. In my experience, when a size limit is set, SQL Server quite often hits it well before the developers or DBAs expected it to be hit. Once that happens the database stops working.

When the transaction log is backed up, it is truncated and all of its space is made available again. This doesn't happen with database files, so you need to be confident about the size of your data. You can set up alerts that will monitor your files proactively, ensuring you are warned when they hit a limit (e.g., only 10% of space is left).

This is a pretty standard database configuration. We'll start by typing in the three words we have already seen, along with `ON PRIMARY`:

```
CREATE DATABASE AddressBook
ON PRIMARY;
```

This statement will not work; the `ON PRIMARY` is stating that the file we are about to specify is the primary data file. A database can only have one primary file. We'll add the details for the data file.

```
CREATE DATABASE AddressBook
ON PRIMARY
(
NAME = 'AddressBook',
FILENAME = 'C:\temp\sqlbasics\AddressBook.mdf',
SIZE = 10MB,
MAXSIZE = UNLIMITED,
FILEGROWTH = 50%
);
```

We've now met our requirements for the data file. We've named the file, given it a full file name so SQL Server knows where to create it, and specified its size and growth details. As it is, this is a valid statement and the database would be created, with a default log file, if it was executed. We don't want a default log file, as we can't control where it should be placed. Let's add the log file specification.

```
CREATE DATABASE AddressBook
ON PRIMARY
(
NAME = 'AddressBook',
FILENAME = 'C:\temp\sqlbasics\AddressBook.mdf',
SIZE = 10MB,
MAXSIZE = UNLIMITED,
FILEGROWTH = 50%
)
LOG ON
(
NAME = 'AddressBook_Log',
FILENAME = 'C:\temp\sqlbasics\AddressBook_Log.ldf',
SIZE = 2MB,
MAXSIZE = 100MB,
FILEGROWTH = 2MB
);
```

That's all we need, and as you start out with SQL Server it is likely this is all you need to know about creating a database. There are many more options; search for *T-SQL CREATE DATABASE* on the Internet for a full outline on MSDN (the Microsoft Developer Network web site).

Press F5 and, just like in **Figure 3-10**, your database will be created.

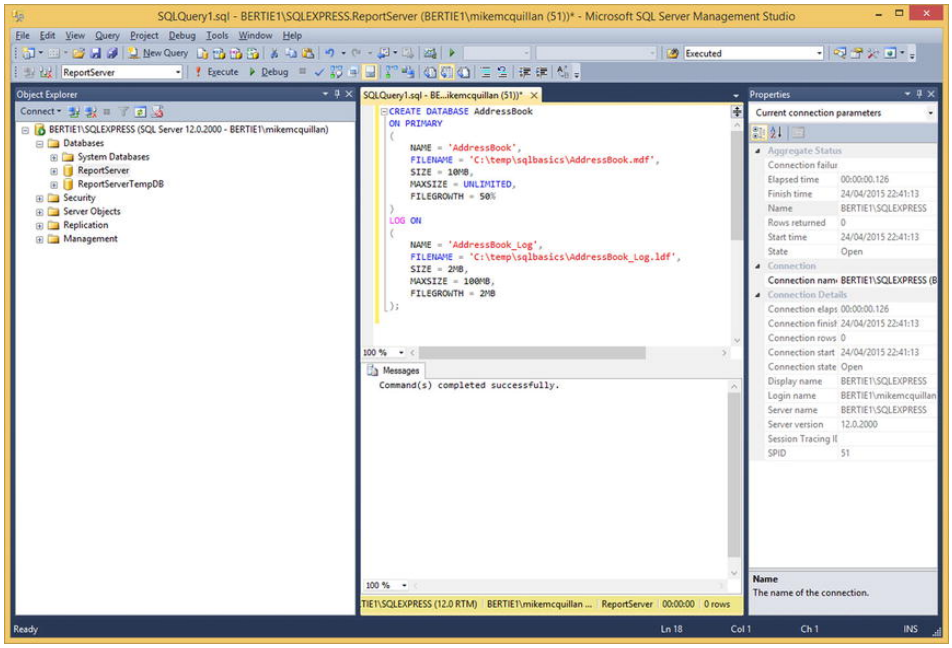

Figure 3-10. *Complete* `CREATE DATABASE` *in T-SQL*

Press F5 again and you'll see an error, telling you the database already exists.

Now, remember why we've created a script: for reusability purposes. A script allows us to deploy to multiple environments. However, DBAs don't take kindly to scripts failing! So we need to add a mechanism that will allow the script to complete successfully, even if the database already exists.

What we need to do is:

- Check if the database exists
- If it doesn't exist, create it
- If it does exist, do nothing

Do you remember the **Resource** system database? This holds a set of **sys** tables, which hold information about your server and databases. We can use this information to check if the database exists.

Amend your `CREATE DATABASE` script so it looks like this:

```
USE Master;
IF NOT EXISTS (SELECT 1 FROM sys.databases WHERE [name] = 'AddressBook')
BEGIN

CREATE DATABASE AddressBook
ON PRIMARY
(
NAME = 'AddressBook',
FILENAME = 'C:\temp\sqlbasics\AddressBook.mdf',
SIZE = 10MB,
MAXSIZE = UNLIMITED,
FILEGROWTH = 50%
)
LOG ON
(
NAME = 'AddressBook_Log',
FILENAME = 'C:\temp\sqlbasics\AddressBook_Log.ldf',
SIZE = 2MB,
MAXSIZE = 100MB,
FILEGROWTH = 2MB
);

END;

GO
```

You can run this as many times as you want. The script will not fail, and the database will only ever be created once. But what is this script doing? It's actually using some concepts we'll be looking at in more detail later on.

Figure 3-11 shows the complete statement. The first line is now `USE Master`. This gives the focus to the **Master** database, so what we are saying is apply this script using the **Master** database. As this is the parent system database, it makes sense to use this as a default. We'll be using the `USE` statement much more in later chapters.

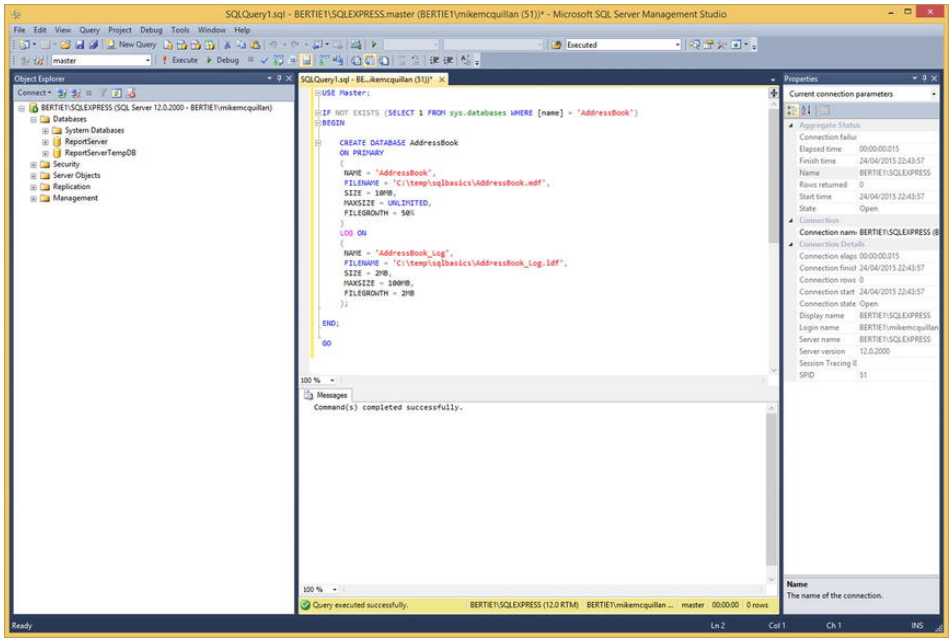

Figure 3-11. *Complete* `CREATE DATABASE` *statement in T-SQL*

The next section of code is interesting; it is the code that checks if the database exists.

```
IF NOT EXISTS (SELECT 1 FROM sys.databases WHERE [name] = 'AddressBook')
```

The `SELECT 1...` statement will return a single column containing the literal numeric value of `1` if a database called **AddressBook** exists on the server. If you highlight that piece of code and run it (in SSMS, if you highlight a block of code and run it, only the highlighted code will be executed), you should see a result of `1` (assuming you have created your database). If the database does not exist, no result is returned.

`IF NOT EXISTS` says if no result of `1` is returned from the `sys.databases` query, execute the `CREATE DATABASE` statement. This is now enclosed in a `BEGIN...END` block to link it to the `IF` statement. In short, what we have is:

```
IF (Condition is true)
BEGIN
       Do Something
END
```

We could extend this by adding an `ELSE` to do something else if the database does already exist, but we'll look at `ELSE` when we are writing SQL statements in later chapters.

The very last line is `GO`. SQL Server executes T-SQL code in batches. A batch is just one or more T-SQL statements. The `GO` command signifies the end of a batch. It is good form to put this in at the end of each batch of commands. When we look at SQLCMD in **Chapter 6** you'll see how useful `GO` can be.

Create a new folder in `c:\temp\sqlbasics` called `apply`. Then save this script as `c:\temp\sqlbasics\apply\01 - Create AddressBook Database.sql`.

Congratulations, you have a database ready to use, and you've just created your first script!

Dropping A Database Using T-SQL

DBAs are very keen on rollback scripts. These are scripts that undo whatever your "apply" scripts did (the `CREATE DATABASE` script we just created is an apply script).

We'll create a rollback script to drop the database, should it exist. Open a New Query Window (Ctrl+N) and type the following:

```
USE Master;

IF EXISTS (SELECT 1 FROM sys.databases WHERE [name] = 'AddressBook')
BEGIN
DROP DATABASE AddressBook;
END;

GO
```

This is pretty similar to the `CREATE DATABASE` script, with a few key differences. Instead of `IF NOT EXISTS`, we now have `IF EXISTS`. The `SELECT 1...` query is still the same. So now we are saying that if the query returns `1`, perform the action. The action in this case is to drop the database. Run this as many times as you want. It will never fail, but the first time it runs, it will drop the database.

NOT EXISTS AND EXISTS

The `EXISTS` keyword is used to check if some condition evaluates to true—for example, `IF EXISTS (SELECT 1 FROM sys.databases WHERE [name] = 'AddressBook')` returns `true` if the **AddressBook** database exists. `NOT EXISTS` inverts this, so if we have `IF NOT EXISTS (SELECT 1 FROM sys.databases WHERE [name] = 'AddressBook')`, the statement will only return `true` if the database does not exist. These constructs are very powerful and can help you control the flow of your code.

To finish off, save this script. In `c:\temp\sqlbasics`, create a folder called `rollback`. Save this script as `c:\temp\sqlbasics\rollback\01 - Create AddressBook Database Rollback.sql`. It's a good convention to keep the rollback script file names the same as the apply scripts, but with `Rollback` either prepended or appended to the name. This makes it easy for you and others to match the scripts up.

Note Always ensure your rollbacks leave the system in the same state as your apply script found it.

Advanced Database Scripting

We're all done with database creation. It is worth noting again that the `CREATE DATABASE` statement is extremely powerful and has lots of options, so once you are comfortable with basic database creation, please take a look at the MSDN documentation to check out the more advanced ways you can use this statement (`https://msdn.microsoft.com/en-us/library/ms176061.aspx`).

There is also an `ALTER DATABASE` statement. This can do most of what the `CREATE DATABASE` statement does, along with a few other things, but on existing databases. Again, take a look on MSDN (`https://msdn.microsoft.com/en-us/library/ms174269.aspx`).

Almost any database object you can create will offer the three statements `CREATE`, `ALTER`, and `DROP`. You'll see more examples of this pattern later in the book.

Creating Scripts Automatically

If you think writing scripts is going to be a bit onerous, you can use SSMS to automatically generate scripts for you. Ensure you can see the AddressBook database in Object Explorer and right-click it. Move over the option that reads **Script Database as**, which is highlighted in **Figure 3-12**.

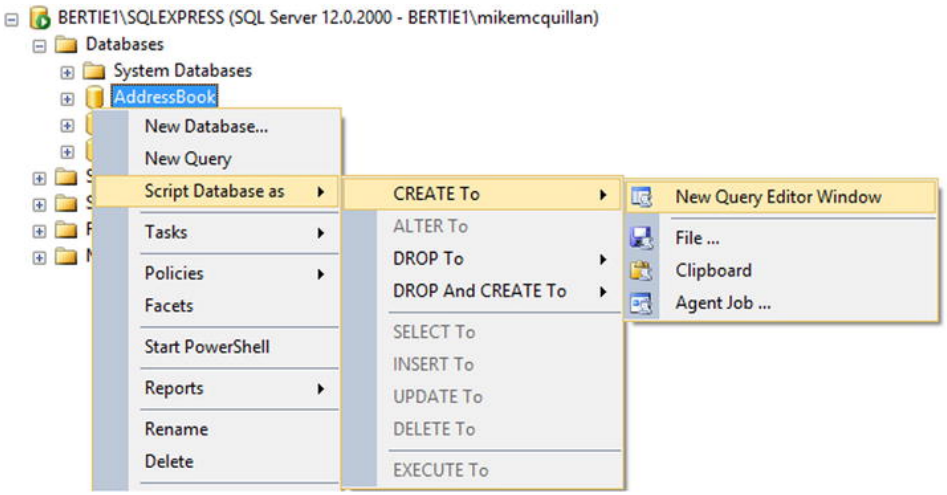

Figure 3-12. *Automatically creating a CREATE DATABASE script*

There are a number of options in here, all of which will create an appropriate script for you. This can be a great time-saver. Move over the **CREATE To** option and choose **New Query Editor Window**. This will open a New Query Window, populated with the T-SQL code. You can choose to save the script as a file, should you wish.

Figure 3-13 shows the generated script. You'll notice that the script generated is much longer and more complicated than the script we created. That's because the script has created statements to set every single database setting; we just accepted the defaults for all of these. But the end result of running this script will be the same; it would create the database. Have a play with this option to determine which scripts you can generate.

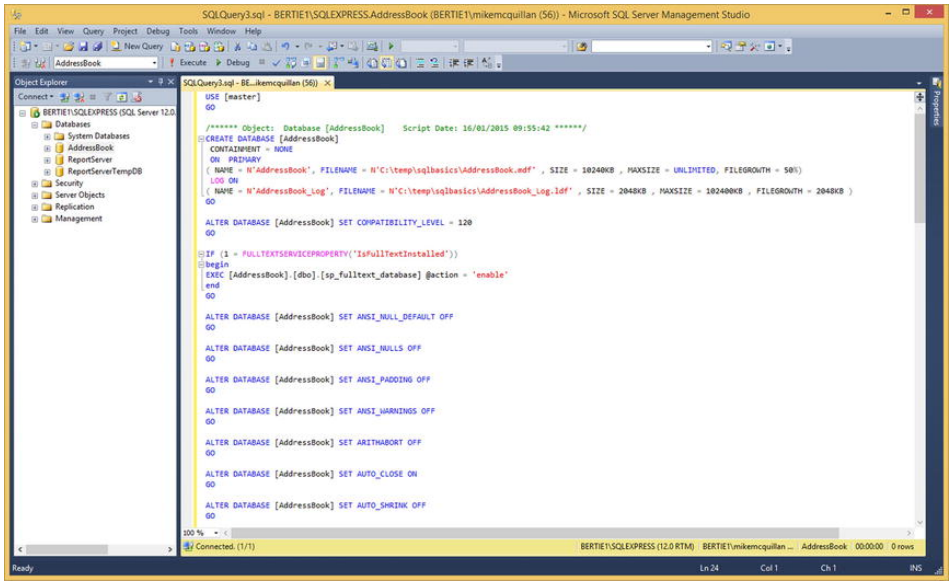

Figure 3-13. *The generated CREATE DATABASE script*

Any scripts you generate will only include the appropriate `CREATE`, `ALTER`, or `DROP` statement; it won't include an `IF EXISTS` statement like the one we created earlier.

Database Properties

To finish off this chapter, we'll take a look at some of the important properties SQL Server maintains per database, and how we can programmatically find out what these properties are.

If you right-click the **AddressBook** database and choose the **Properties** option on the context menu (it is the last item), a dialog appears listing lots of information about your database (Figure 3-14): things like the date on which the database was last backed up, the owner of the database, and the size of the database.

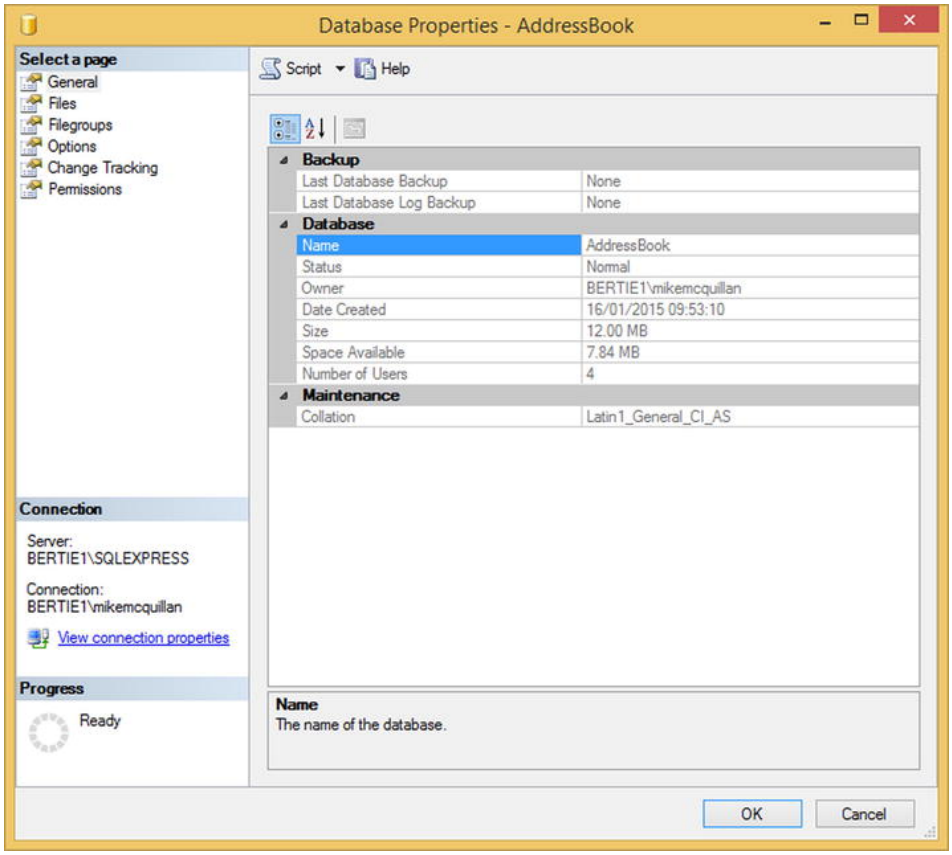

Figure 3-14. *The Database Properties dialog*

We'll take a quick look at three important properties.

- **Status**: The current state of the database. Usually you want this to be **Online**, which means the database is online and available for use. If a database is unavailable it may be **Offline**, and if it is being restored it could be in the **Restoring** state.
- **Recovery Model**: This option is found on the Options page on the left-hand side of the Database Properties dialog. This setting is key. There are three recovery models, and they dictate how the transaction log works. **Full** means all data is written to the transaction log. You won't lose any data if the data file fails. **Bulk Logged** means normal operations are logged, but bulk copy operations (operations involving lots of rows) are minimally logged. You could lose data here if you haven't backed up and a bulk copy operation has occurred. The final option is **Simple**. This doesn't log anything and is usually a bad idea.**Full** is the most common model used and is what should be used in production. If you plan to execute some pretty large inserts/updates/deletes against the database, consider temporarily switching to **Bulk Logged** or **Simple**.
- **Collation**: Collation is shown on the General page, and is also shown on the Options page. On the Options page, it can be changed to a different collation, if desired. Collation dictates how language is used in the database. There are lots of collation options, which dictate things like case sensitivity and accent sensitivity. Collations can affect whether a value like "`Dolly`" is treated the same as "`dolly`". It is important to ensure these stay consistent across your servers and databases to prevent issues with cross-database referencing (if required).

These are more advanced topics, but it is important to know they exist. There are lots of other properties available for you to manage. If you need to obtain a particular property value as part of a script, you can use the `DATABASEPROPERTYEX` system function (we'll talk about functions in Chapter 16). This example returns the Status and the Collation:

```
SELECT  DATABASEPROPERTYEX('AddressBook', 'Status') AS Status,
        DATABASEPROPERTYEX('AddressBook', 'Collation') AS Collation;
```

You can use this to return any database property. Just pass in the name of the database and the name of the property you want. You can find a full list of properties by going to `https://msdn.microsoft.com/en-us/library/ms186823.aspx`. A demonstration of the executed statement can be seen in Figure 3-15.

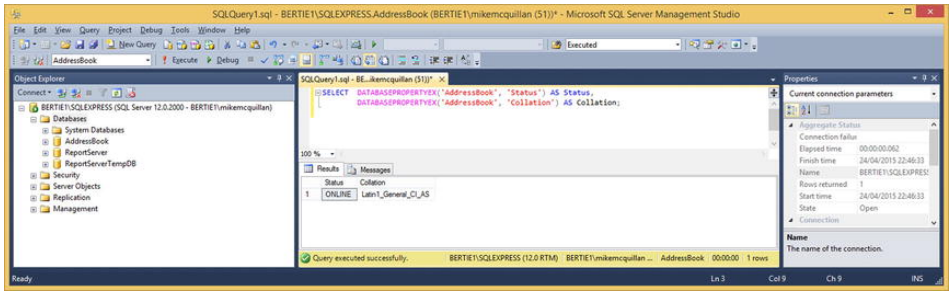

Figure 3-15. *Displaying database properties using T-SQL*

Summary

Well done, you have created a database! In a number of different ways, I might add. Unfortunately, a database without any content is like an empty glass. You need to fill your database up with useful objects that will allow it to store data. We'll take a look at the first and most important of these—tables—in the next chapter.

PREV
Chapter 2 : Obtaining and Installing SQL Server

NEXT
Chapter 4 : Tables