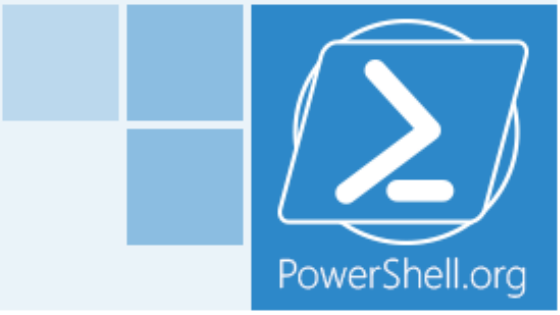
Creating Historical and Trend Reports with PowerShell and SQL Server

by Don Jones

curated by
Matt Johnson



Making Historical and Trend Reports in PowerShell

by Don Jones

Visit PowerShell.org to check for newer editions of this ebook.

PowerShell.org ebooks are works-in-progress, and many are curated by members of the community. We encourage you to check back for new editions at least twice a year, by visiting PowerShell.org. You may also subscribe to our monthly e-mail TechLetter for notifications of updated ebook editions. Visit PowerShell.org for more information on the newsletter.

Feedback and corrections, as well as questions about this ebook's content, can be posted in the PowerShell Q&A forum on PowerShell.org. Moderators will make every attempt to either address your concern, or to engage the appropriate ebook author.

What This Book is About

This book assumes that you need to create reports that utilize stored data - typically, historical data on resource consumption or something like that. If you just need to pump out quick inventory reports, like system configurations or something, check out *Creating HTML Reports in PowerShell*. That's a lot simpler.

I know a lot of folks find themselves in the position of having to "do it yourself" when there are perfectly great tools out there - problem being those tools cost money, and The Boss won't spend money. Let's be clear that The Boss is stupid, or at least penny-wise and pound-foolish, because you're going to spend enough time building your own reporting tools that, strictly based on your salary, a tool would be cheaper. But at least, if you're going to build your own reporting tools, you can do a good job of it!

I know a lot of administrators use Excel for stuff like this. The scripts I've seen, where people are manipulating Excel through its Component Object Model (COM) interface, manipulating cells and inserting charts and graphs... oy. The scripts I've seen. It's a lot of work. It's a kludge. It's a hack. I know Excel seems easy to start with, but the curve is logarithmic: every little thing you want to add - another chart, another page, another data series - gets harder and harder. *Excel isn't a database*, nor it is a reporting system. Yes, it's free (well, you've already bought it, so it doesn't cost extra), but you're using it for entirely the wrong purposes. "But it's easy to learn!" I'm told by its fans. Well, yes - to a point. But if some administrators added up all the time they've spent hacking Excel and dealing with its API's shortcomings, they'd have had time to learn something better instead.

Hence, this book.

We're going to build a system that uses PowerShell to collect data and add it to a SQL Server database. We're going to use SQL Server Reporting Services to build reports off of that database - reports which, I might add, can be scheduled and delivered and are generally better-looking than anything you could do in Excel. *And it's going to cost us nothing*. I'm going to give you tools that make this *easy*, and in large part hide all the underlying SQL Server complexity. I'm a nice guy that way (donations cheerfully accepted on http://PowerShellBooks.com if you're appreciative).

Ditch Excel. Let's do this reporting thing *right*.

System Requirements

All you're going to need is SQL Server 2012 Express with Advanced Services, which you can download *for free* from http://www.microsoft.com/en-us/download/details.aspx?id=29062. If you already have a SQL Server on your network, and it has the SQL Server Reporting Services (SSRS) installed, that'll work as well. Any version of SQL Server with SSRS, in fact, will do the job. 2012 Express, in particular, can be installed on Windows 7, Windows Server 2008 R2, Windows Server 2008 (SP2), Windows 8, and Windows Server 2012. There are 32-bit and 64-bit editions to suit any environment, and you only need a couple gigabytes of RAM to run it well.

You'll need .NET Framework 3.5 SP1 and 4.0 installed as a pre-requisite.

Of course, you'll need Windows PowerShell. For the purposes of this book, I'm assuming you're running PowerShell v3. I'm not sure if anything in here won't work in 2.0, but I'm not testing it. 3.0 runs on the same operating systems, with the same requirements, as SQL Server 2012 Express, so just download it, install it, and you should be good to go.

There's no need for a dedicated server infrastructure. If you want, you can just install SQL Server on your desktop or laptop - it's fine for what we're doing.

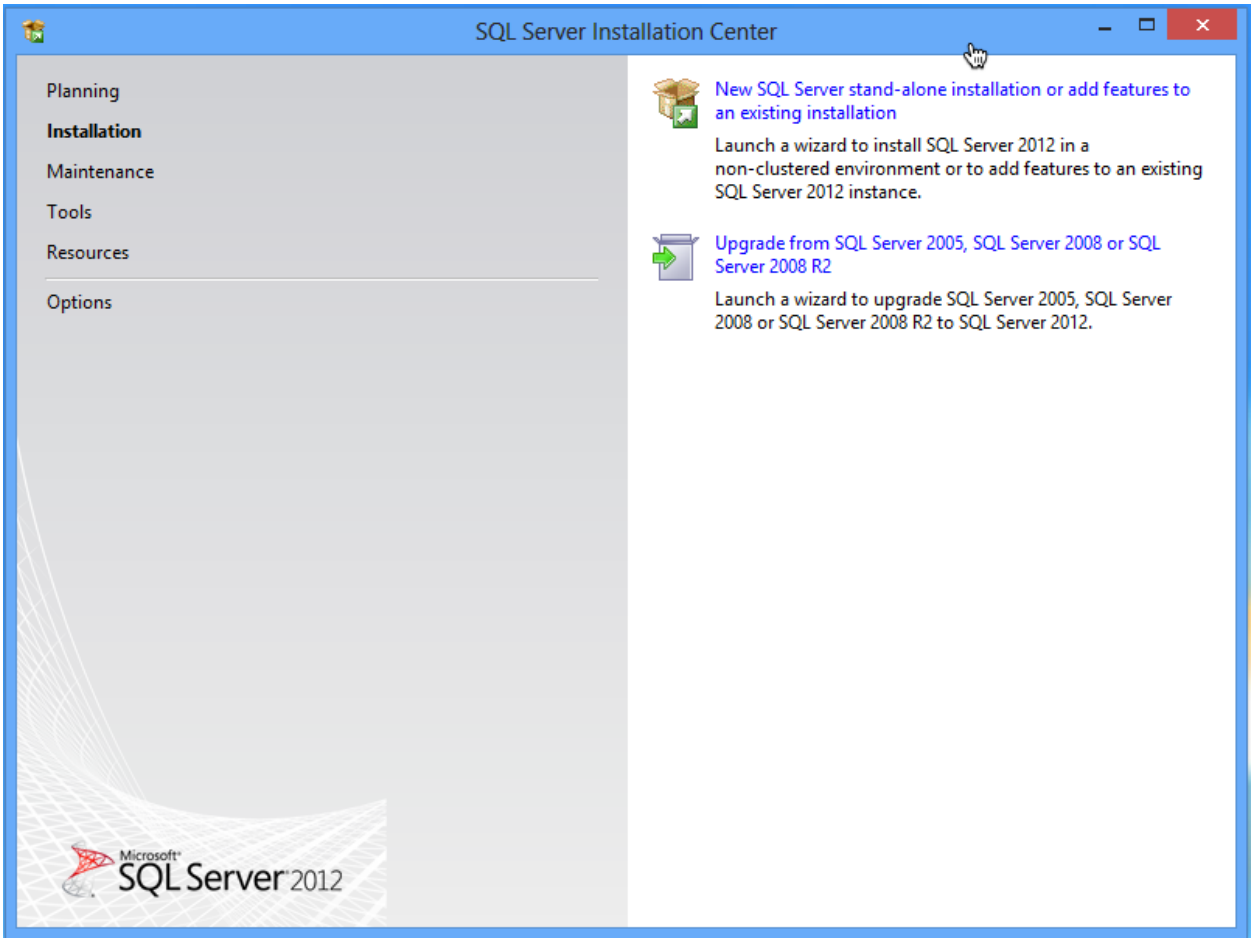
Installing SQL Server 2012 Express

If you already have a suitable SQL Server on your network, you can skip this chapter. I'm assuming that you're installing this on your desktop or laptop, and that you'll be the only one using it, so we're going to do a very lightweight, easy version of the installation. The tools I'm going to provide you are built to quickly and easily connect to a local SQL Server 2012 Express instance, although they're also capable - with a bit more syntax from you - of connecting to any SQL Server anywhere on your network.

As a bonus, as I walk you through this installation, I'm going to point out a few SQL Server concepts. You don't necessarily need them, but it'll make troubleshooting easier, and it'll make you a generally better admin. Having a few basic SQL Server concepts under your belt can't possibly hurt, given how widespread SQL Server is in any modern organization.

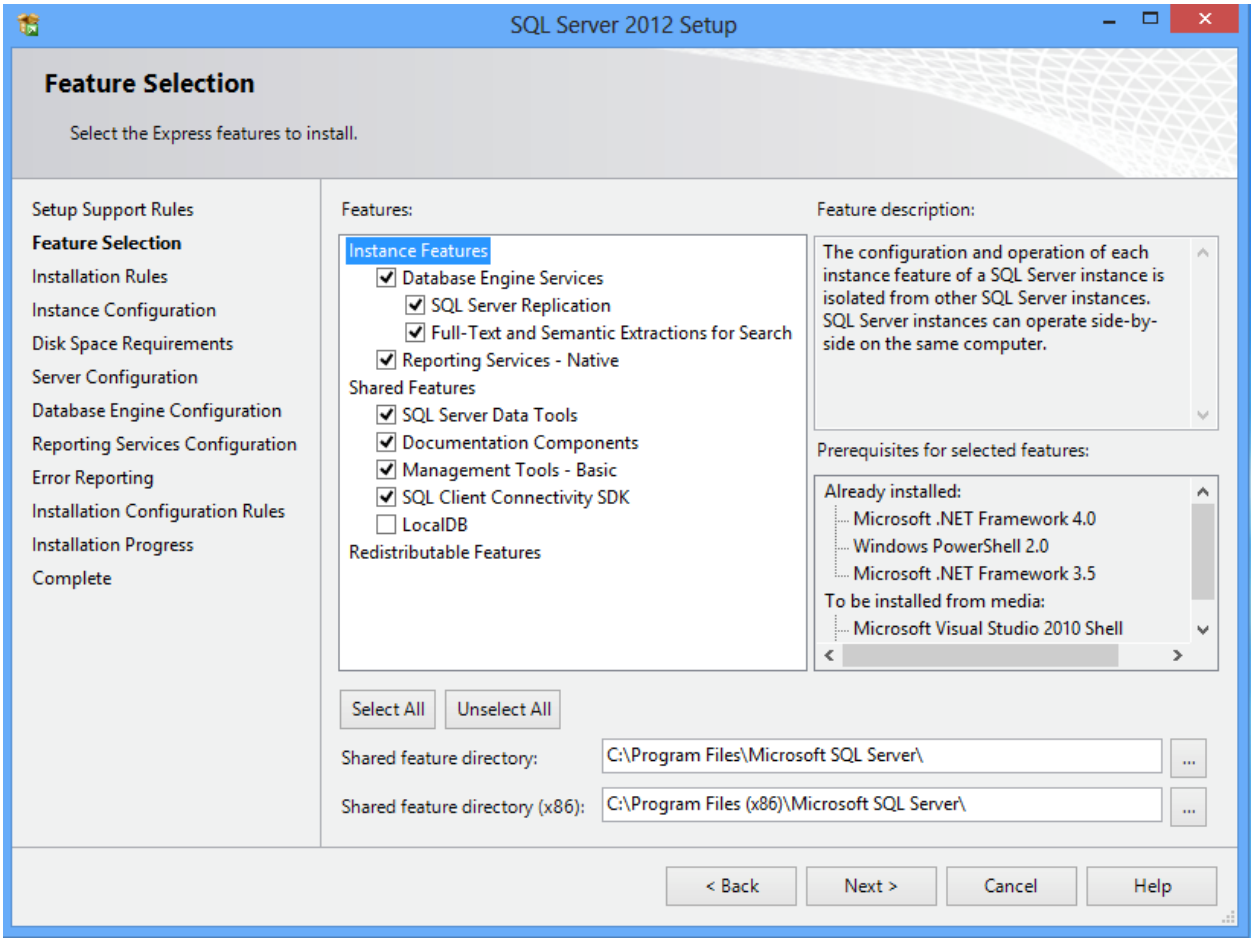
Starting after your SQL Server 2012 Express (with Advanced Services) download completes, you're going to double-click the installer to kick things off. If your computer has User Account Control (UAC) enabled, make sure you run the installer as Administrator, because that's what you'll be running PowerShell as. It's important that SQL Server be available to the same account that you use to launch PowerShell.

You're going to do a new stand-alone installation, so click that link:

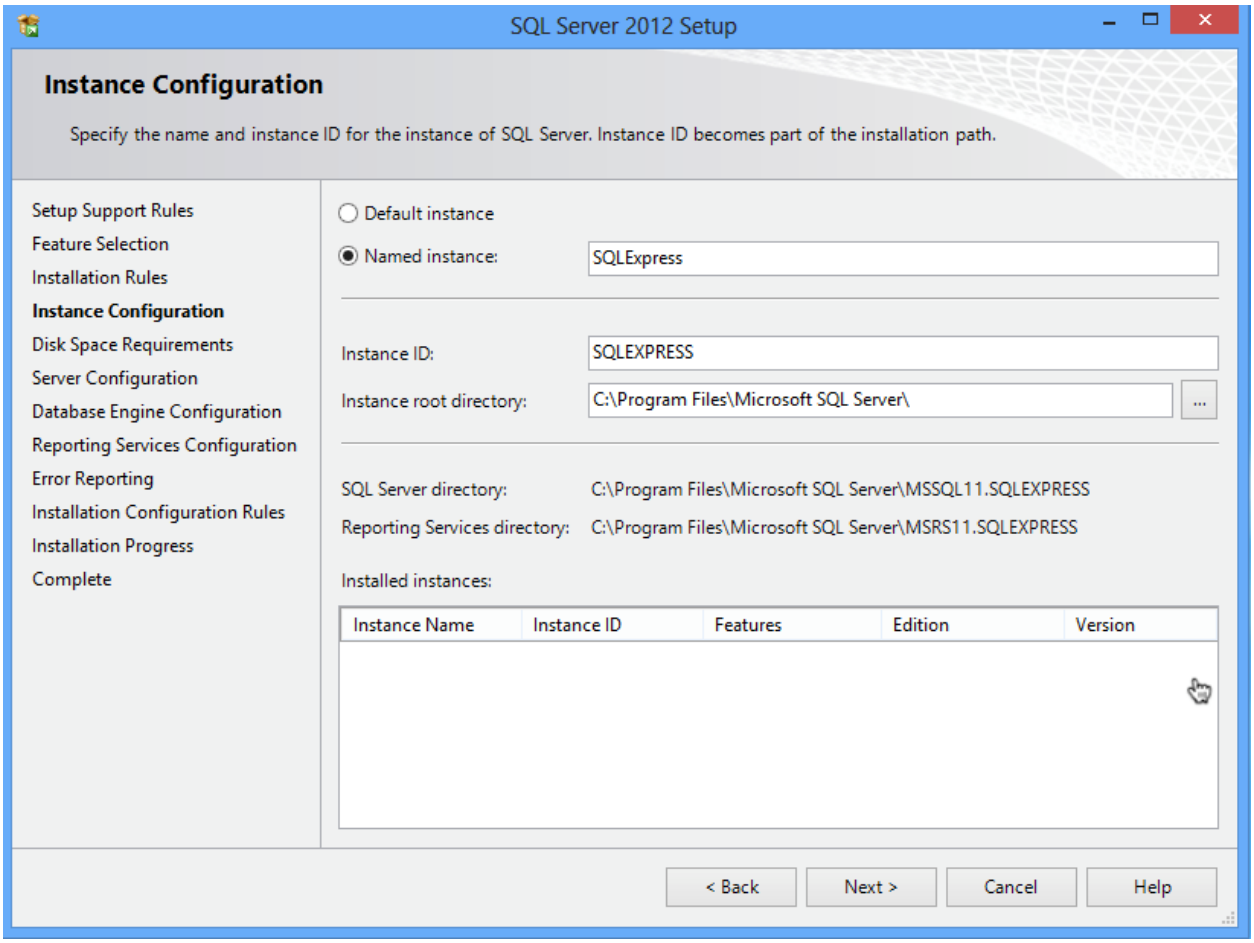


By the way, it's generally fine to have SQL Server 2012 installed alongside another version, should you happen to have another version already installed.

SQL Server's installation wizard will go through several screens, including a license agreement, some pre-requisite checks, blah blah blah. It's a pageant. Here's where you come in:

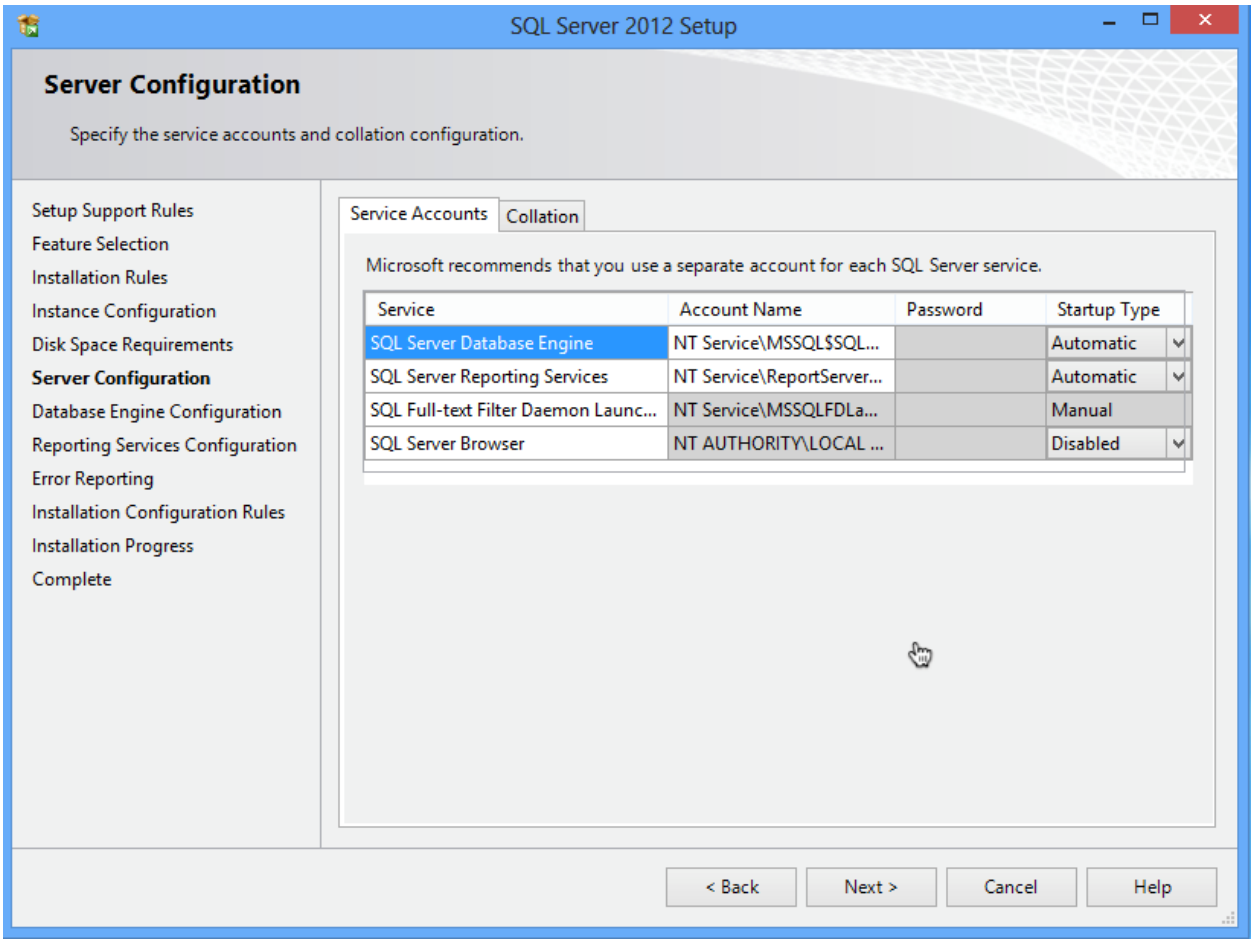


Those defaults are pretty much what you want. Click Next.

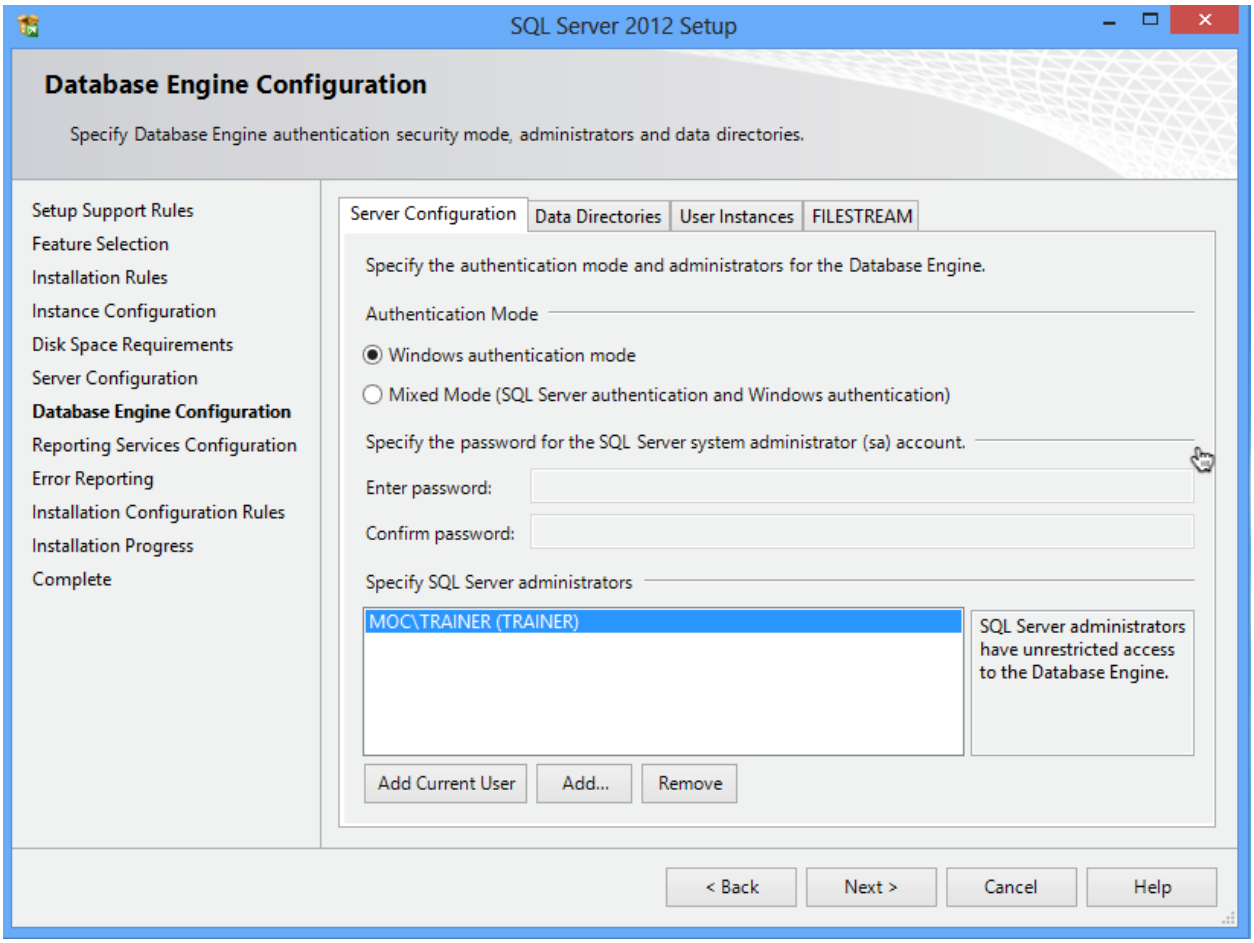


This is an important bit. SQL Server Express installs a *named instance* of itself, named SQLExpress. This will fail if you already have an instance running under that name. A named instance is basically a copy of SQL Server that requires you to specify your computer name *and* the instance name in order to connect. Whatever you put here, *remember it*. I'm going to assume you're sticking with the default SQLEXPRESS.

Confirm your disk space requirements on the next screen, and get to the server configuration:

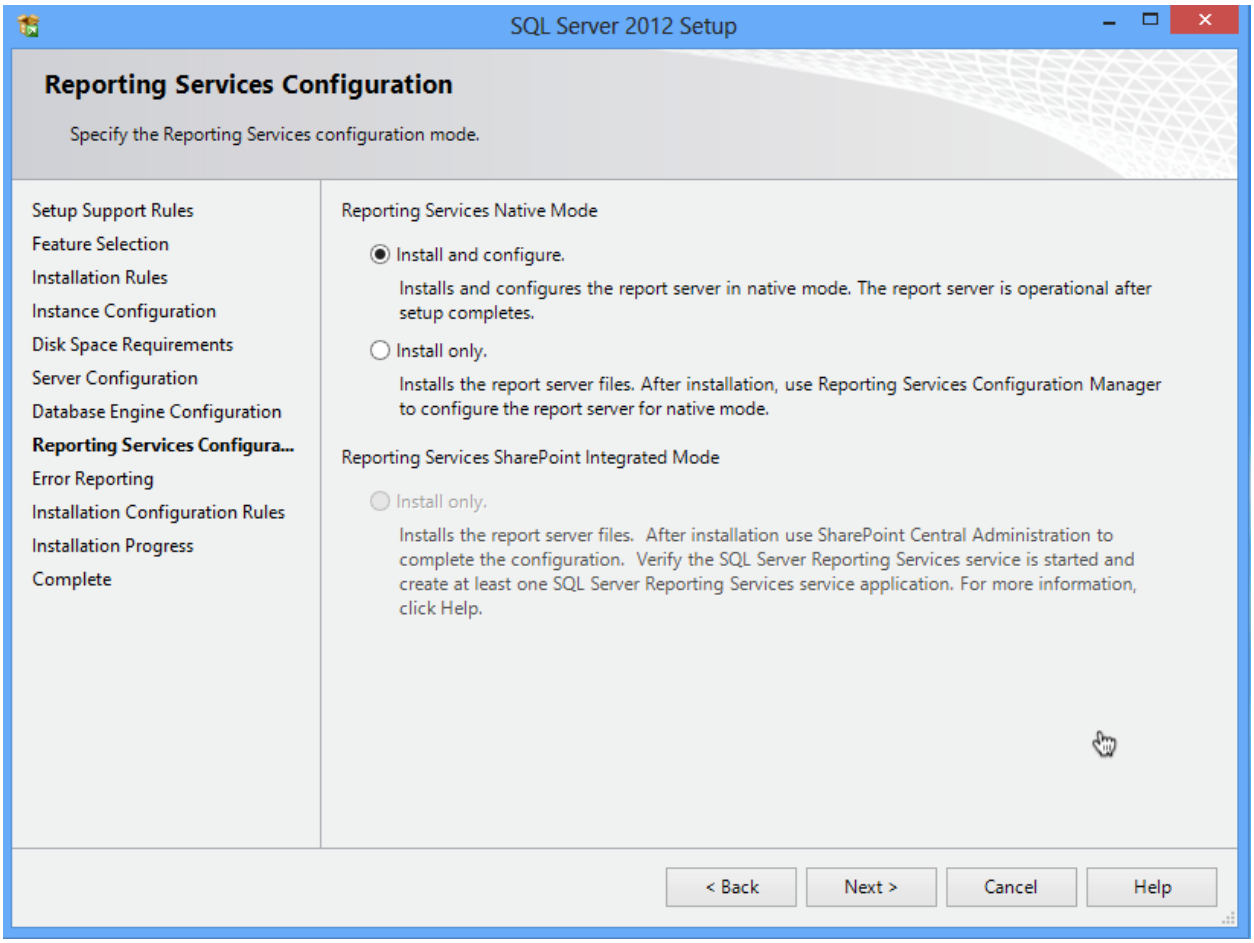


For our purposes, the defaults are fine. Next is the Database Engine configuration:



Again, the defaults are fine if you're just installing SQL Server to support this reporting functionality. It's going to default to Windows authentication, meaning it uses your current login, and it's going to add your current login account as an administrator of SQL Server.

This is important: SQL Server has this feature called "user instances." It means that, if a non-admin tries to run SQL Server, they get their own isolated instance of it. It's all magical and transparent, *but it won't include the databases from other instances.* So it's *really important* to make sure that the "Specify SQL Server administrators" box lists the user account(s) that you'll use to run PowerShell. Otherwise we'll never get everything to match up. You can add accounts, including domain accounts, if needed. You're just giving these accounts permissions inside your local SQL Server installation, so it's no big danger or security risk.



Stick with the default, to install and configure. From there, you're done. "Next" your way through the rest of the wizard and go get a cup of coffee.

Note: If you don't select "install and configure" then Reporting Services won't be set up for use. After installation, you'll have to create a Reporting Services database and do a bunch of other stuff. You may have to do that if you're starting with an existing SQL Server installation where SSRS isn't already configured. Setting that up is beyond the scope of this book.

Setting Up a Database

Now, there are a *lot* of ways you could set up a database. I'm assuming, if you're reading this book, that you (a) know relatively little about SQL Server, (b) want to more or less keep it that way, and (c) don't want to deal with much complexity. So I'm going to have you create a *single* database in which all of your historical data will be stored. The tools I'm giving you will take care of creating their own tables in which to store data, so we just need to give them a database in which to do so.

Asking for a Database

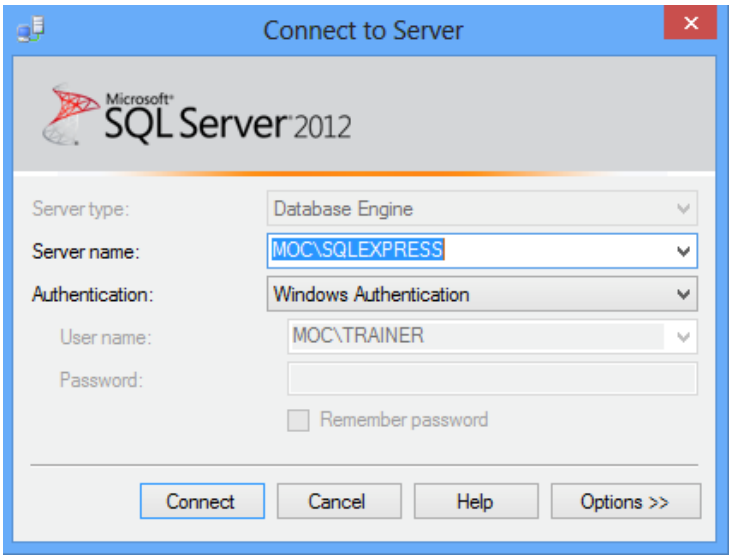
If you're planning to use a SQL Server instance that's already running in your environment, you'll need to get the DBA to set up a database for you (if you're the DBA, then you'll know how). Have them:

- Add a Windows login, to SQL Server, for whatever account you use to run PowerShell.
- Create a database. Name it anything - you'll just need to know them name.
- Add your Windows login to the database's db_owner (database owner) database role.

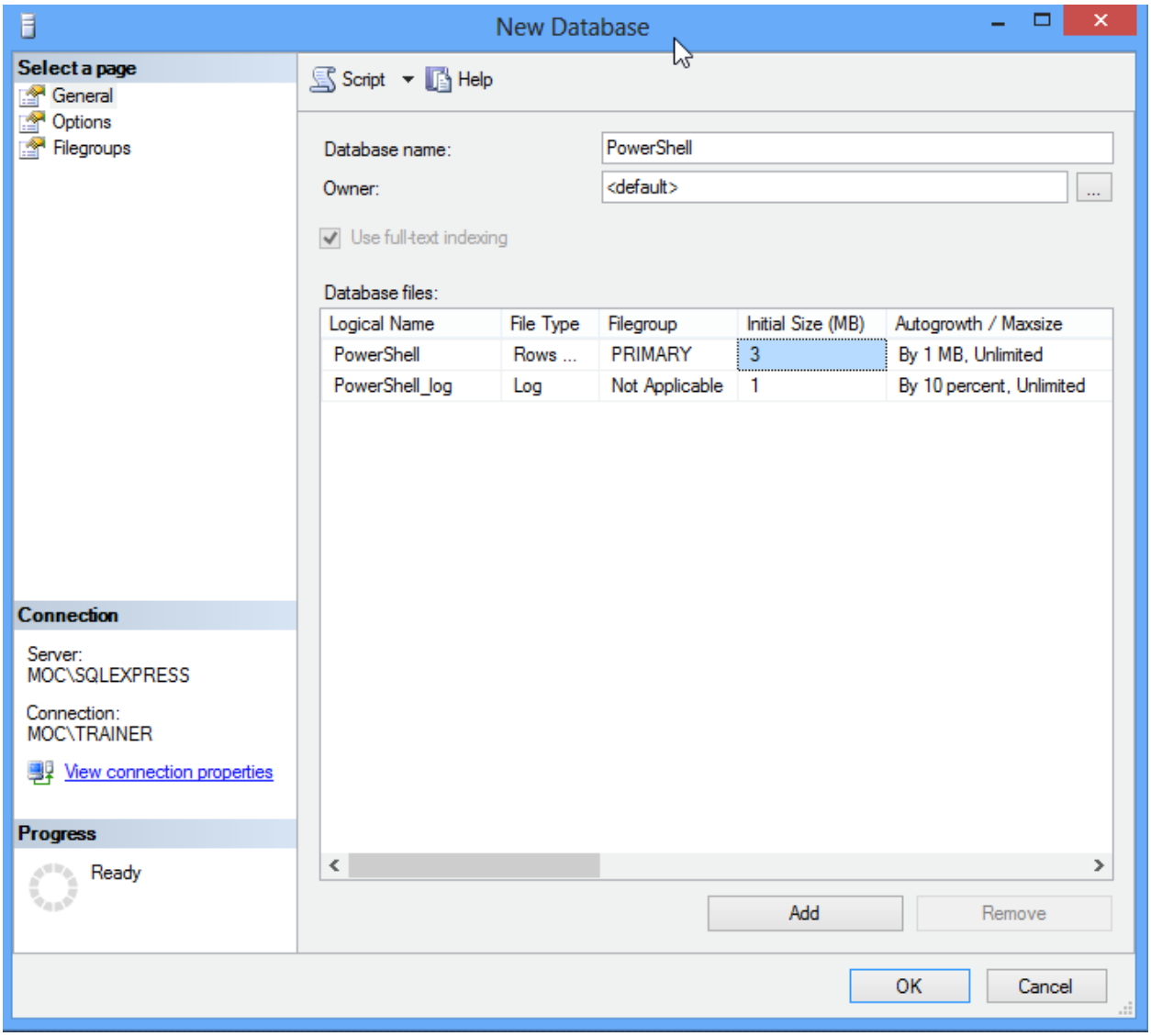
They're going to have to tell you the server name also, and if your database isn't in the default instance you'll need to know the instance name.

Making a Database

If you're using a local SQL Server Express instance, start by opening SQL Server Management Studio. Log in using Windows Authentication and whatever account you use to run PowerShell - you won't provide a password here. Notice that it's pointed at my computer name ("MOC") and instance name ("SQLEXPRESS").



Right-click the Databases folder, select New Database, and fill in the dialog box.



You can pretty much accept the defaults after supplying a database name (letters only to keep it simple; I've used "PowerShell"). I'm skipping over a *lot* of SQL Server details, here; suffice to say that these defaults result in a pretty maintenance-free database. Scroll to the right a bit to change the location of the database files.

If you're worried about backing up your data, you can read up on how to use SQL Server to run backups. I'm assuming that you're not putting anything mission-critical in here, so that it doesn't need backed up; if you're concerned about backups then the right thing to do is use a full, backed-up SQL Server instance that's on a network server someplace. SQL Server Express is kind of intended for "I won't miss it much if it's gone" kind of work.

Collecting Data

Data collection is where you'll be using PowerShell. Keep in mind that one of the goals of this book is to provide you with tools that make storing the data in SQL Server as transparent and easy as possible. With that in mind, the data must be collected and formatted in a way that supports transparent and easy. So I'm going to establish some standards, which you'll need to follow.

As an example, I'm going to write a script that retrieves local drive size free space information (on my systems, SAN storage looks like a local drive to Windows, so it'll capture that space for SAN-connected volumes). I'm writing the data-collection piece as a PowerShell advanced script; if you're not familiar with those, check out *Learn PowerShell Toolmaking in a Month of Lunches* (http://manning.com/).

There are a few important things to notice about this:

- The data-collection function outputs a single custom object. This is the only thing my data-storage tools will accept.
- The custom object has a custom type name, "Report.DiskSpaceInfo." The first part of that, "Report." is mandatory. In other words, every function you write must output a custom object whose type name starts with the word "Report" and a period. The second part of the type name is defined by you, should represent the kind of data you're collecting, and *must be unique within the database.*
- Be sure to include a ComputerName property if appropriate, so that you can identify which computer each piece of data came from.
- You should always include a "Collected" property, which is the current date. Don't convert this to any kind of display format - you'll worry about that when building the report. Right now, you just want a raw date/time object.
- Property names should consist of letters, maybe numbers, and nothing else. No spaces, no symbols. It's "FreeSpace," not "Free Space (bytes)."
- You need to be a bit careful about what data you collect. My tools are capable of dealing with properties that contain dates (like "collected"), strings of up to 250 characters, and 64-bit integers (capable of storing numbers up to around 9,000,000,000,000,000,000). Anything else will result in un-handled errors and possibly lost or truncated data. Strings using double-byte character sets (DBCS) like Unicode are fine.
- You don't have to explicitly include ComputerName or Collected properties. However, if you do, I'll automatically tell SQL Server to index those (I'll also index any Name property you pass in) for faster reporting, since you'll most often be sorting and filtering on those fields.

Planning is crucial here. Once your first data is added to the SQL Server database, *you lock in the properties that will be stored.* Here, its going to be computer name, drive letter (DeviceID), size, free space, and the collected date. I can never add more information to my DiskSpaceInfo repository without deleting the existing data and starting over. So the moral of the story is, *think hard about what you want to collect and report on.*

When run, my function outputs something like this:

```
FreeSpace    : 43672129536
Size         : 64055406592
ComputerName : MOC
Collected    : 11/17/2012
DeviceID     : C:
```

Don't be tempted to make this look fancy - remember, it's go to go into SQL Server. When you build your reports, you can pretty this up as much as you want.

I'm naming my script C:\Collect-DiskSpaceInfo.ps1. You'll see that script name later.

Storing Your Data

Included with this book should be a script module named SQLReporting. This needs to go in a very specific location on your computer:

My Documents (make sure it's *My* Documents, not *Public* Documents)

WindowsPowerShell

Modules

SQLReporting

SQLReporting.psm1

If you don't have the folder and filename structure exactly correct, this won't work. After putting the module there, open a new PowerShell console and run **Import-Module SQLReporting** to make sure it imports correctly. If you get an error, then you didn't put it in the right spot.

I'm assuming that, by this point, you have a command or script that can produce the objects you want to store, and that those have an appropriate type name as outlined in the previous chapter. For the sake of example, my script name is C:\Collect-DiskSpaceInfo.ps1.

I'm also assuming that you've installed the SQLReporting module as outlined above.

Saving to Your Local SQL Server Express Instance

If you're taking the easy way out, and saving to a local SQL Server Express instance named SQLEXPRESS, do this:

```
C:\Collect-DiskSpaceInfo.ps1 |
Save-ReportData -LocalExpressDatabaseName PowerShell
```

That's it. Of course, this presumes you've already created the "PowerShell" database (which could be named anything you want; I outlined that earlier in this book). In my example, the objects output by Collect-DiskSpaceInfo.ps1 have a type name of "Report.DiskSpaceInfo," so they'll be stored in a table named DiskSpaceInfo within the PowerShell database. That table gets created automatically if it doesn't exist.

Saving to a Remote SQL Server

This is a bit trickier, but not too much. You'll need to start by constructing a *connection string*. They look something like this:

```
Server=myServerAddress;Database=myDataBase;Trusted_Connection=True;
```

You fill in your server name or IP address. For example, to access the default instance on SERVER1, the server address would just be SERVER1. To access an instance named MYSQL on a server named SERVER2, the server address would be SERVER2\MYSQL. That's a backslash. You also fill in your database name. Using the connection string looks like this:

```
C:\Collect-DiskSpaceInfo.ps1 |
Save-ReportData -ConnectionString
"Server=myServerAddress;Database=myDataBase;Trusted_Connection=True;"
```

In other words, just pass the connection string instead of a local express database name. The necessary table will be created if it doesn't already exist.

Reading Your Data

PowerShell isn't really meant to read the data from your SQL Server database; SSRS will do that. But, if you want to quickly test your data, you can do so. You'll need a -LocalExpressDatabaseName or -ConnectionString parameter, exactly as described under "Storing Your Data." You also need to know the original object type name that went into the database, such as "Report.DiskSpaceInfo" in my example.

```
Get-ReportData -LocalExpressDatabaseName PowerShell -TypeName Report.DiskSpaceInfo
```

That's it. The appropriate objects should come out of the database. From there, you can sort them, filter them, or whatever using normal PowerShell commands. But the real action is when you use SSRS to read that data and construct reports!

Collecting Performance Data

I'm adding this chapter to the book somewhat under protest. I *know* performance data is something folks want to collect; PowerShell just isn't an awesome way to do it.

PowerShell can only pull performance data *as its happening*. So, let's say you grab your computer's CPU utilization *right now*. It's 3%. What's that tell you about your CPU utilization? Nothing. Your computer could average 80%, and you just happened to catch it on its slowest millisecond of the day. So performance data has to be captured in a series of samples, often over a long period of time, and that's where I'd argue PowerShell isn't well-suited.

Imagine you're capturing data from a remote computer: You've got to make a WMI connection, or a Remoting connection to use CIM, because those are the only real ways PowerShell can easily access performance counters. Both your computer and the remote one are running a copy of PowerShell to do that (in the case of Remoting, at least), which means they've spun up the .NET Framework, blah blah blah. It's layers of software; performance-capturing software is usually designed to be small and lightweight, and PowerShell is neither of those. That isn't a dig on PowerShell; it's just not what the shell was designed for.

But I know a lot of admins are going to want to do this anyway. "The Boss won't buy SCOM, and we have to do something." I'd argue that you *don't* have to do "something" other than tell the The Boss, "no, you can't do that, it's not how the software works and you're creating more performance load than you're measuring, fool," but I suppose being employed can sometimes overrule technical accuracy.

To minimize impact, I'm going to offer an approach that runs scripts right on the machines you're measuring, rather than capturing data remotely. Yes, you're building a "performance monitoring agent," and that should bother you, because you probably didn't get into this industry to be a professional software developer. But that's what this task entails.

(By the way, if all this ranting sounds a little grumpy, I really only get to write these free ebooks on Sunday mornings and I haven't had much coffee yet.)

Methodology

I'm going to write a script that captures performance data every 5 seconds, for a full minute, right on a local computer. The intent is that you'd schedule this under Task Scheduler to run whenever needed. I'd suggest running it during known peak utilization times, maybe 2-3 times a day. Don't get carried away and schedule it to run every 5 minutes - you're going to generate more data than can possibly be useful.

I'm going to write the actual performance-measuring bit as a function, and then call the function at the bottom of the script. That means you just schedule the script (actually, you'd schedule **PowerShell.exe -filename script.ps1**) to run. If you need to collect multiple bits of performance, you'd add a function for each bit, and call each one in the script.

The data is going to be stored in a remote SQL Server database, and if you plan to have multiple servers doing so, you're going to want a real SQL Server edition, not the Express one (which is limited in how many connections it can accept, amongst other restrictions).

Every server on which you install these is going to need my SQLReporting module. I'd honestly suggest putting the module in \Windows\System32\WindowsPowerShell\v1.0\Modules\SQLReporting\SQLReporting.psm1, just to ensure the module is available for all user accounts - including whatever account Task Scheduler is using to run the

script. This isn't a "best practice;" the best practice would be to put the module somewhere *not* in System32, and then add that path to the PSModulePath environment variable. That, however, leads us down a path of instruction and explanation that's out of scope for this book. So I'm taking the cheater's way out and using System32.

Finding a Counter

I'm going to rely on the PowerShell v3 Get-Counter command to access performance data. I'm starting by listing all available counter *sets*:

```
Get-Counter -ListSet *
```

Using this, I discover the System set, which looks interesting. So I take a deeper look:

```
 PS C:\> Get-Counter -ListSet System


CounterSetName     : System
MachineName        : .
CounterSetType     : SingleInstance
Description        : The System performance object consists of counters that
apply to more than one instance of a
                     component processors on the computer.
Paths              : {\System\File Read Operations/sec, \System\File Write
Operations/sec, \System\File Control
                     Operations/sec, \System\File Read Bytes/sec...}
PathsWithInstances : {}
Counter            : {\System\File Read Operations/sec, \System\File Write
Operations/sec, \System\File Control
                     Operations/sec, \System\File Read Bytes/sec...}



PS C:\> Get-Counter -ListSet System | Select-Object -ExpandProperty Counter
\System\File Read Operations/sec
\System\File Write Operations/sec
\System\File Control Operations/sec
\System\File Read Bytes/sec
\System\File Write Bytes/sec
\System\File Control Bytes/sec
\System\Context Switches/sec
\System\System Calls/sec
\System\File Data Operations/sec
\System\System Up Time
\System\Processor Queue Length
\System\Processes
\System\Threads
\System\Alignment Fixups/sec
\System\Exception Dispatches/sec
\System\Floating Emulations/sec
\System\% Registry Quota In Use
```

All interesting stuff. Note that this is a SingleInstance set, meaning there's only one of these on the system. A MultiInstance set comes in bunches - for example, the Process set is MultiInstance, with one instance of the counters for each running process. If you want to accurately grab data, you've got to decide which process you want to grab it for. System, in fact, is the only SingleInstance set I found on my machine. So I'm not going to use it. Might as well show you the hard stuff, right?

How about IPv4 instead?

```
 PS C:\> Get-Counter -ListSet IPv4 | Select-Object -ExpandProperty Counter
\IPv4\Datagrams/sec
\IPv4\Datagrams Received/sec
\IPv4\Datagrams Received Header Errors
\IPv4\Datagrams Received Address Errors
\IPv4\Datagrams Forwarded/sec
\IPv4\Datagrams Received Unknown Protocol
\IPv4\Datagrams Received Discarded
\IPv4\Datagrams Received Delivered/sec
\IPv4\Datagrams Sent/sec
\IPv4\Datagrams Outbound Discarded
\IPv4\Datagrams Outbound No Route
\IPv4\Fragments Received/sec
\IPv4\Fragments Re-assembled/sec
\IPv4\Fragment Re-assembly Failures
\IPv4\Fragmented Datagrams/sec
\IPv4\Fragmentation Failures
\IPv4\Fragments Created/sec
```

Fun stuff there. Let's play. Here's my script, C:\PerformanceCheck.ps1:

```
$Datagrams = Get-Counter -Counter "\IPv4\Datagrams/sec" -SampleInterval 5 -
MaxSamples 12
$DataGrams | Select-Object -Property
@{n='Collected';e={$_.CounterSamples.Timestamp}},
                                    @{n='ComputerName';e={Get-Content
Env:\COMPUTERNAME}},

@{n='IPv4DatagramsSec';e={$_.CounterSamples.CookedValue}}
```

The basic methodology here is to get the counters into a variable, and you can see that I've specified 12 samples every 5 seconds, for a total of 1 minute. You then need to pull out the "CookedValue" property from each counter sample. I've also grabbed the sample timestamp and named it "Collected," and the current computer name, to correspond with what my SQLReporting module needs. The results of this:

```
Collected              ComputerName               IPv4DatagramsSec
---------              ------------               ----------------
11/18/2012 7:58:42 AM  MOC                        0.19922178698138
11/18/2012 7:58:47 AM  MOC                        825.908768304232
11/18/2012 7:58:52 AM  MOC                        711.73267889951
```

That's almost ready to be piped right to my Save-ReportData command, which will need to be given a

complete connection string to a SQL Server database. The problem right now is that Select-Object doesn't give me a way to apply the proper custom TypeName to the objects, so I'll have to modify my script a bit:

```
$Datagrams = Get-Counter -Counter "\IPv4\Datagrams/sec" -SampleInterval 5 -MaxSamples 3
$DataGrams |
ForEach-Object {
    $props = @{'ComputerName'=(Get-Content Env:\COMPUTERNAME);
               'IPv4Datagrams'=($_.CounterSamples.CookedValue);
               'Collected'=($_.CounterSamples.TimeStamp)}
    $obj = New-Object -TypeName PSObject -Property $props
    $obj.PSObject.TypeNames.Insert(0,'Report.IPv4DatagramsPerSec')
    Write-Output $obj
} |
Save-ReportData -Conn
"Server=myServerAddress;Database=myDataBase;Trusted_Connection=True;"
```

It was still worth doing the first version of the script as a quick-and-dirty test, but this second version will save the data into a SQL database.

So there you have it. Run that script every so often - not too often, mind you - and you'll have performance data in a SQL Server database. From there, you can start building reports on it.

Making a Report

I want to start by saying that SQL Server Reporting Services (SSRS) is a pretty powerful set of tools. While it'd be cool if this short little ebook gave you complete SSRS coverage, it ain't gonna happen. Hop on to Amazon (or wherever you buy your books) and you'll find a number of books, each hundreds of pages thick, covering SSRS in detail. What I'm going to attempt to give you is the crash course.

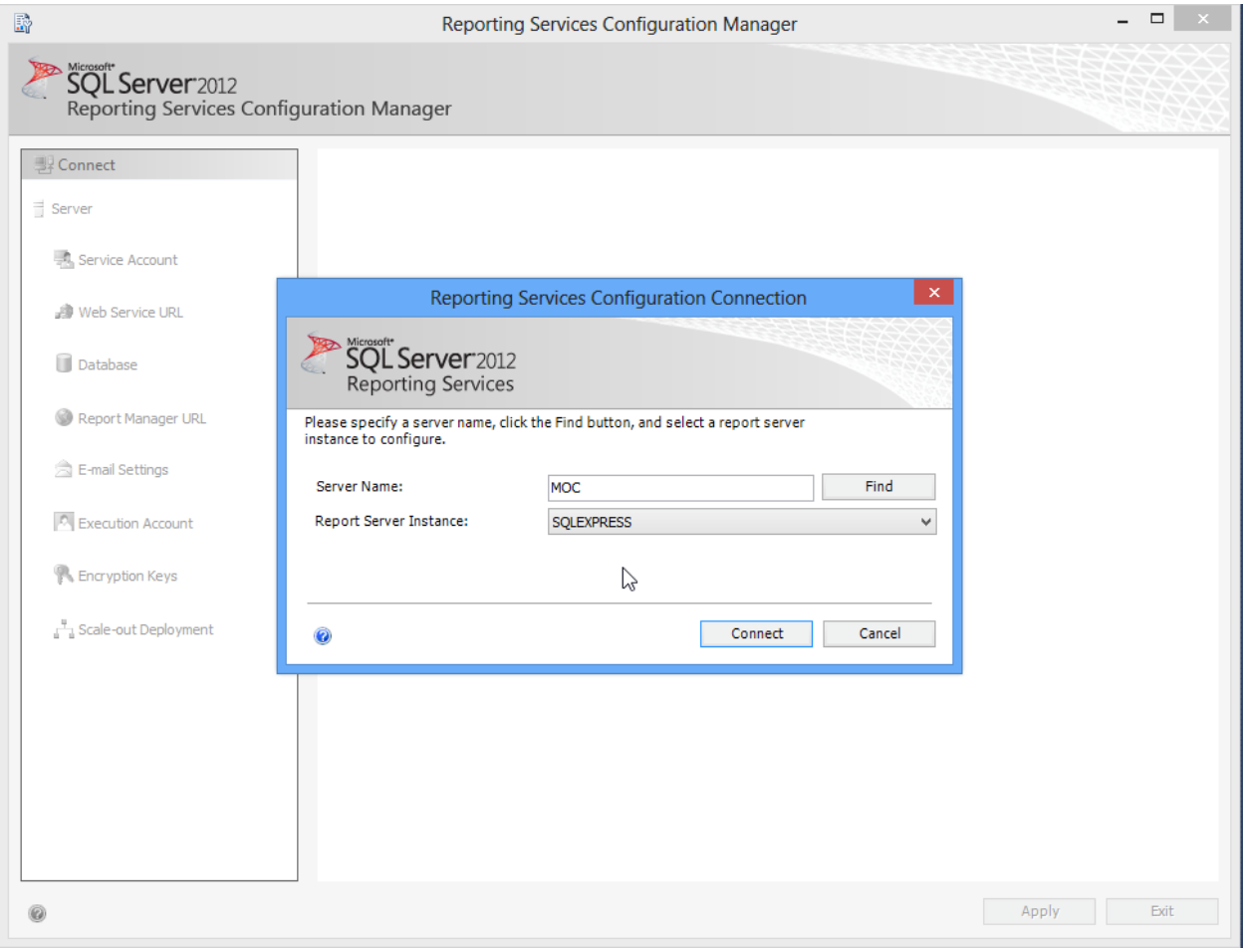
But before I do, let me reiterate something from the beginning of this book: ***SQL Server Reporting Services is worth your time to learn.*** Yes, you already know Excel, or Access, or whatever you're using these days. SSRS is better. SSRS can tap into *any* SQL Server-based data, along with data living in other database platforms. It's powerful. It offers scheduled report generation and delivery, a Web-based reporting console for end-user self-service, and a lot more. This is totally a tool you want to become proficient with, because the investment in time *will be paid back a hundredfold*, I promise.

OK. Let's go.

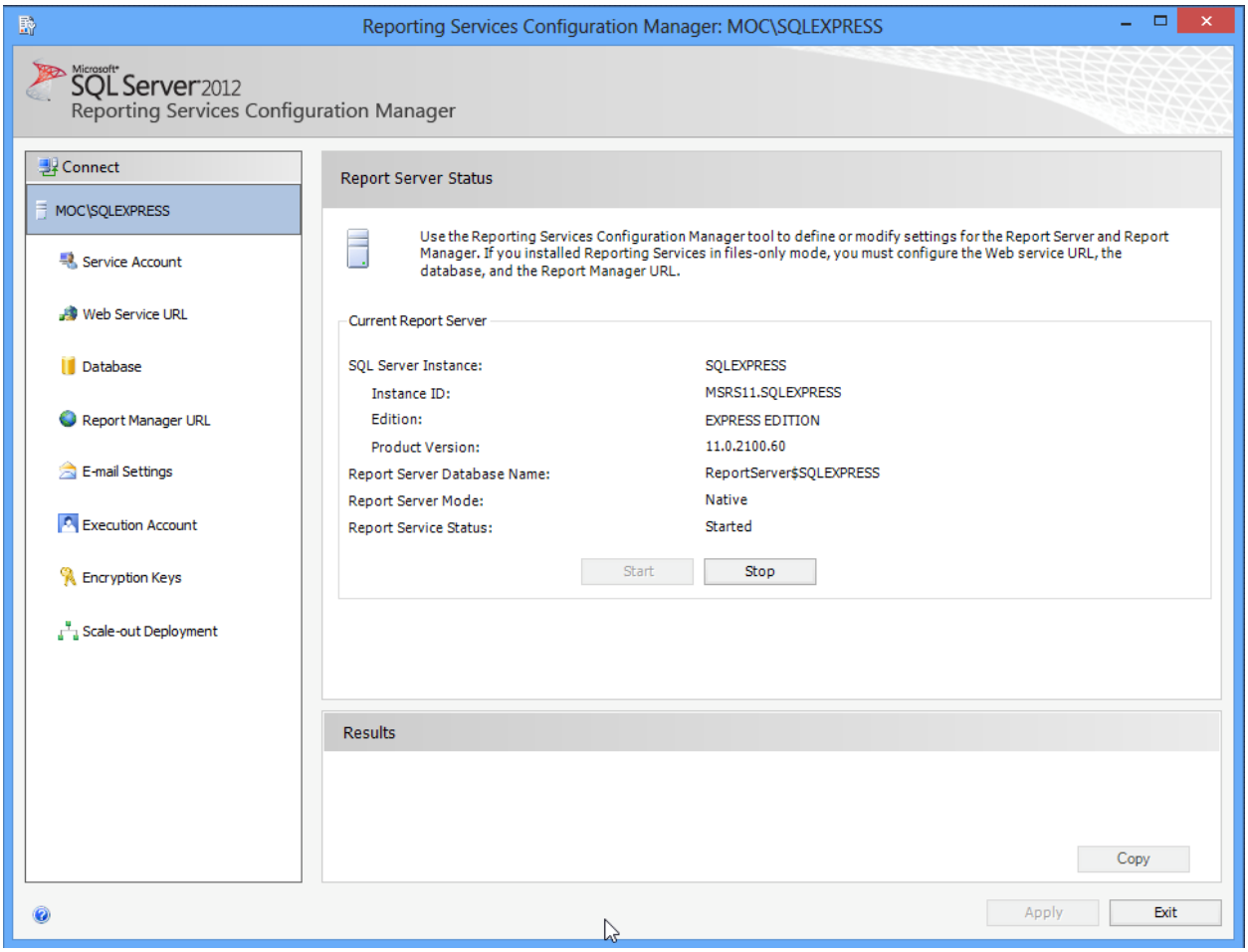
I've populated my PowerShell table with a year's worth of disk information from three computers, queried monthly from each. I'd like to produce a trend-line report, so that I can start to predict when I'll get low on space.

Verifying Reporting Services

I'm going to start by launching the Reporting Services Configuration Manager. I'll log in:



And then just verify that all of the services and whatnot are running properly.



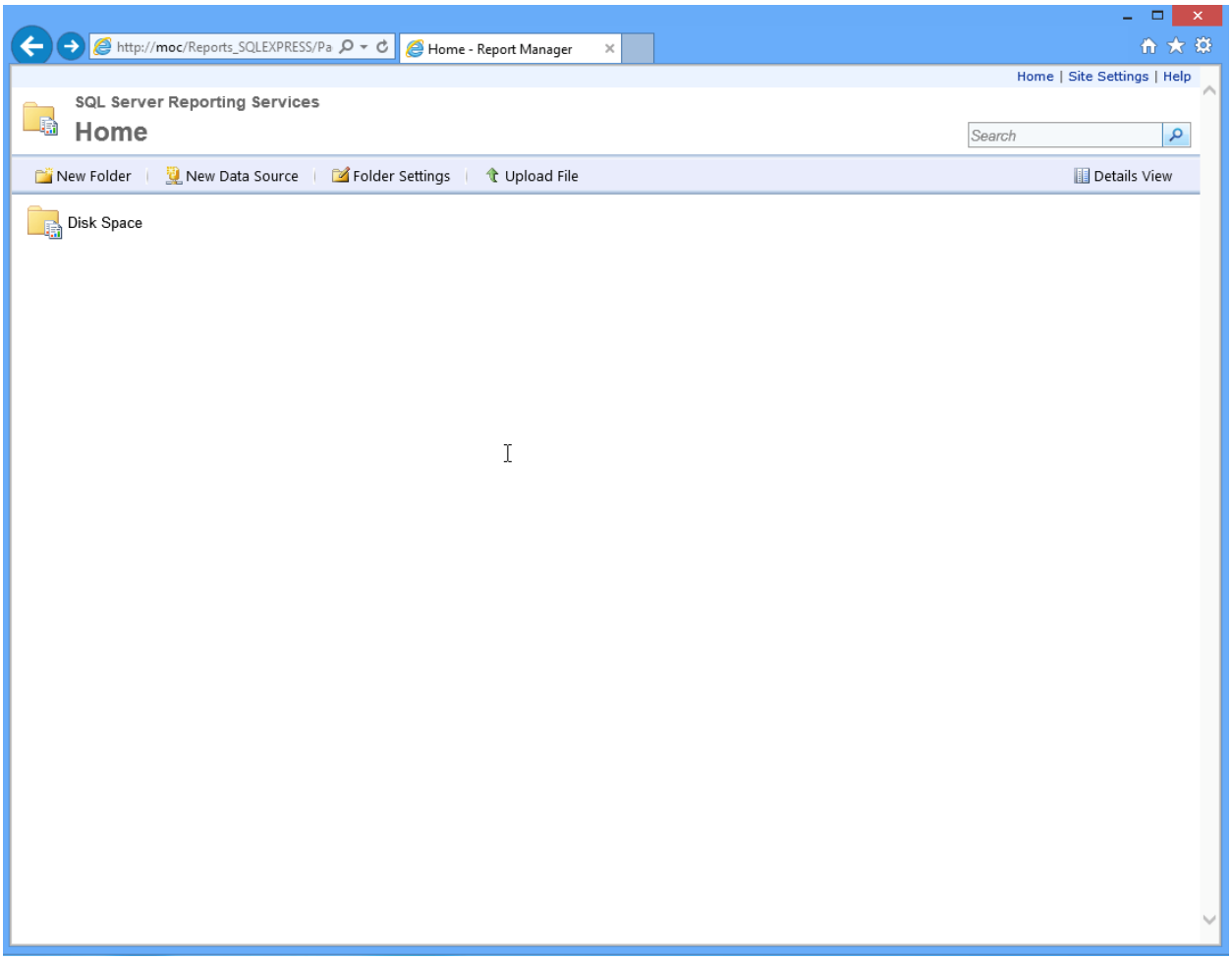
Note that this is also where you can set up the Web Service URL, the Report Manager URL, modify e-mail settings, change the account report-generation runs under, and so forth. I'm just verifying that everything is started and running.

Accessing Report Manager

Also note that Reporting Services has its own embedded Web server; it doesn't depend on IIS. If you go to the "Report Manager URL" tab, you'll find a link you can click to get to the Report Manager Web site. Report Manager can be used to do most of what I need; note that the version of SQL Server Management Studio included with SQL Server Express *cannot* provide this functionality. If you happen to have a full version of SQL Server Management Studio, you can use it to connect to Reporting Services. Since I don't, I'll work with the Web-based Report Manager.

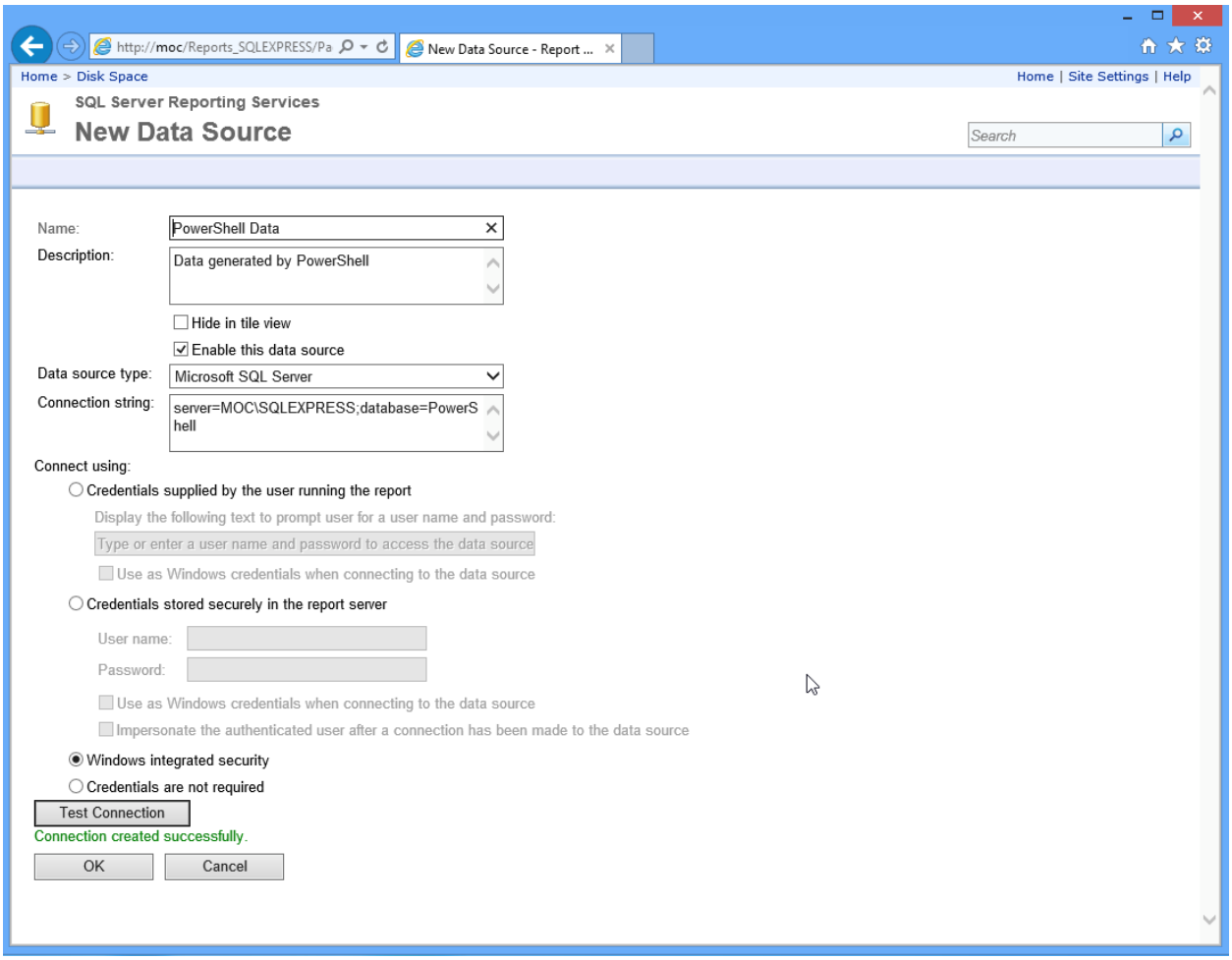
Note: I'm going to breeze through this report-creating stuff - you'll find a more complete tutorial at http://msdn.microsoft.com/en-us/library/ms167559%28v=sql.110%29.aspx.

Also note: I had to explicitly run Internet Explorer "as Administrator" to get Reporting Services to recognize me. Because I didn't feel like playing with permissions for this ebook, I just went with that.



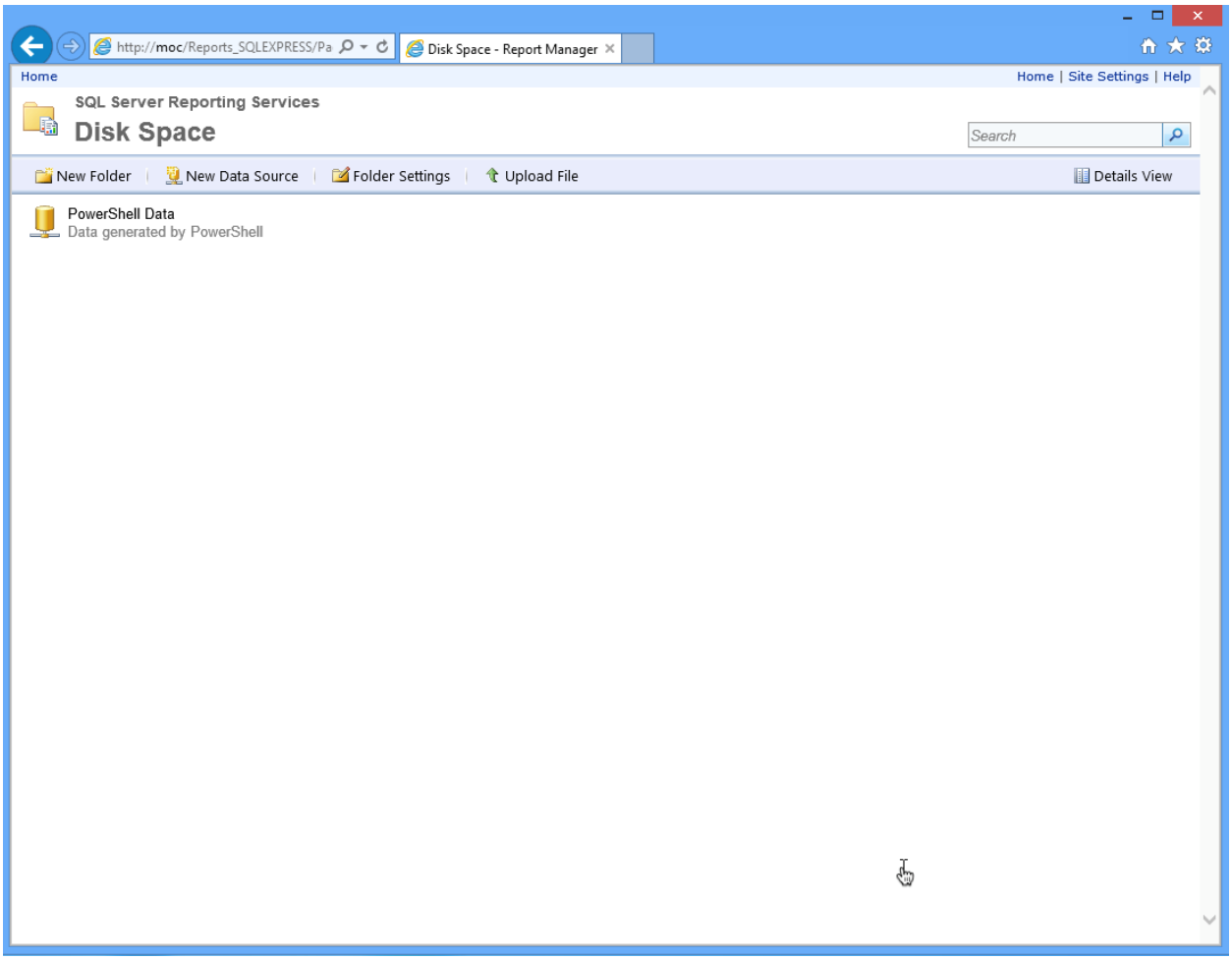
I'm going to start by creating a new folder called "Disk Space." SSRS lets you apply permissions to folders, so you can grant other folks access to this. You can use the "Site Settings" link to assign site-wide permissions, and that'll let non-administrators get in, if you like. Obviously, that makes sense mainly if you're running this on a central SQL Server, and not on your own desktop or laptop.

Next, we have to create a data source within this folder. This is the database where we'll be pulling our data from. Because my SQLReporting module stores all of your data in a single database (named "PowerShell" in my examples), you could create an SSRS folder for *all* of your reports, since they'll all share that same data source.



As you can see, I just needed to specify the server and database portion of the connection string. I've selected Windows Integrated Security, meaning my login account will be used to access the data. Always use the Test Connection button to make sure this is working.

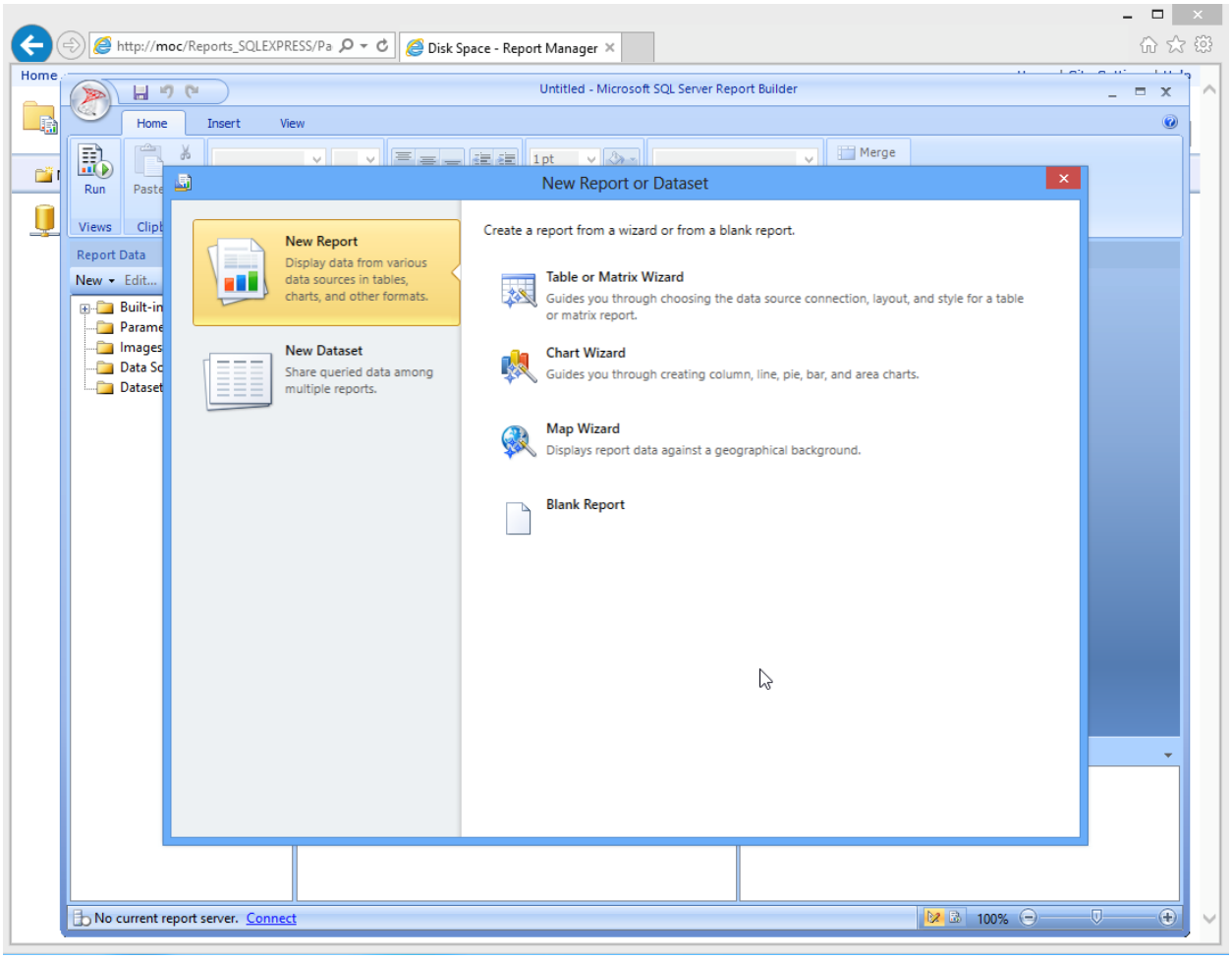
Note that we aren't actually going to use this data source, but I wanted to show you how to create one because once you start really getting into SSRS, having predefined sources can be very handy.



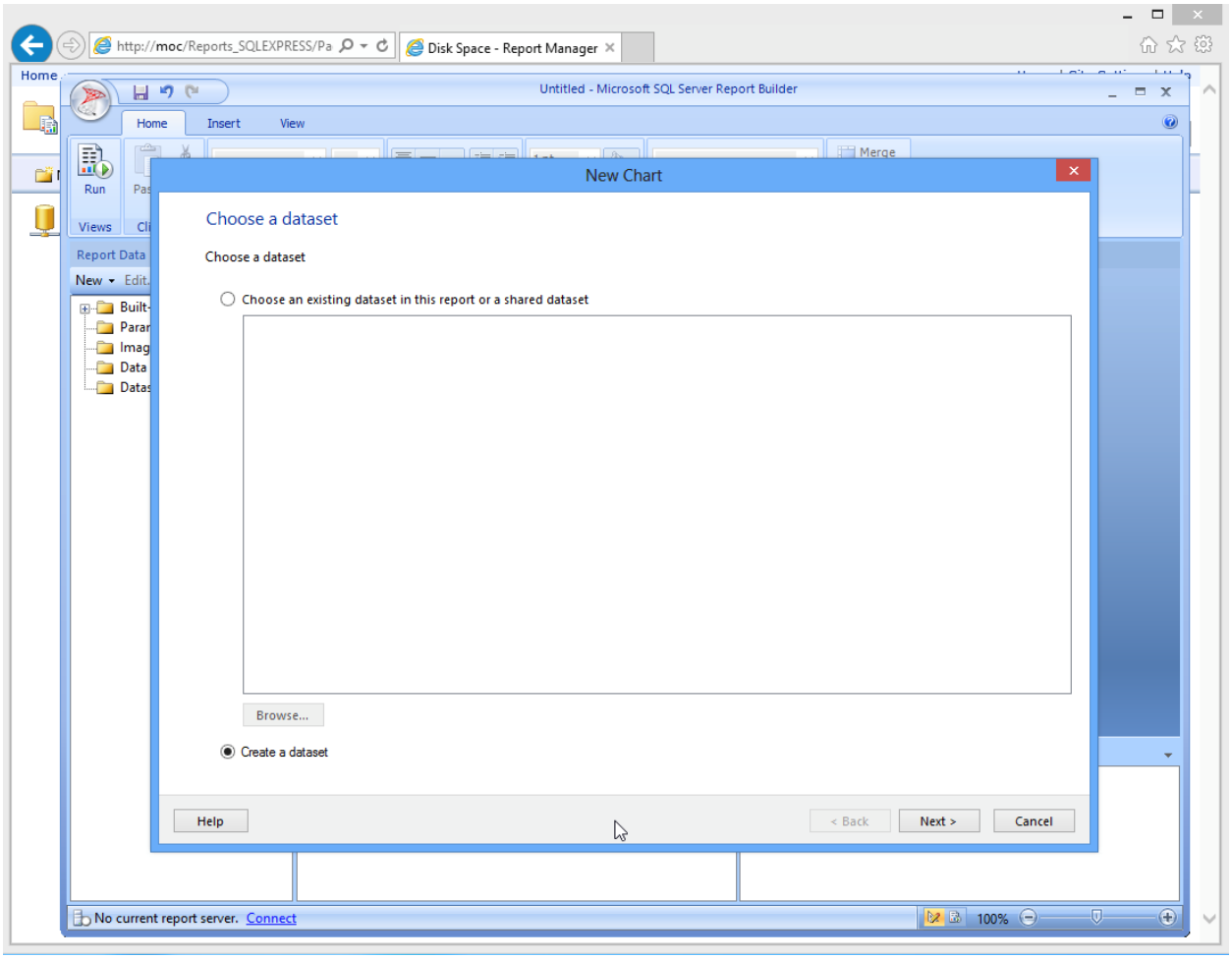
Back in the folder, we can see the new data source.

Building a Report

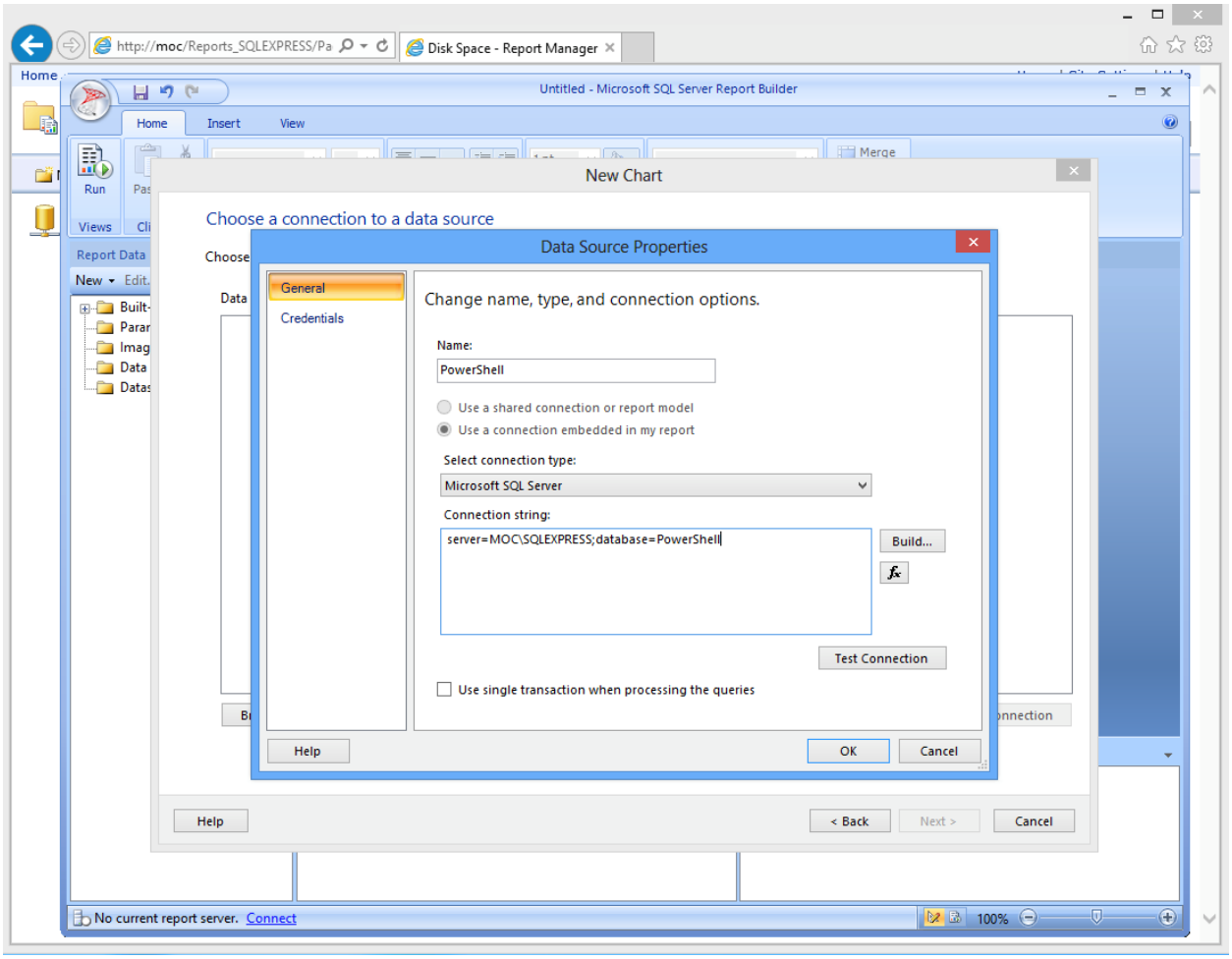
But now we've got to take a bit of a departure. You can't actually *design* reports in Report Manager; you just *manage* them. So we need to create a report definition, and to do that we're going to need a tool. Go to http://www.microsoft.com/en-us/download/details.aspx?id=29072 and get Report Builder. When installing, don't sweat the URL prompt - we can save our reports as .RDL files instead of deploying via HTTP.



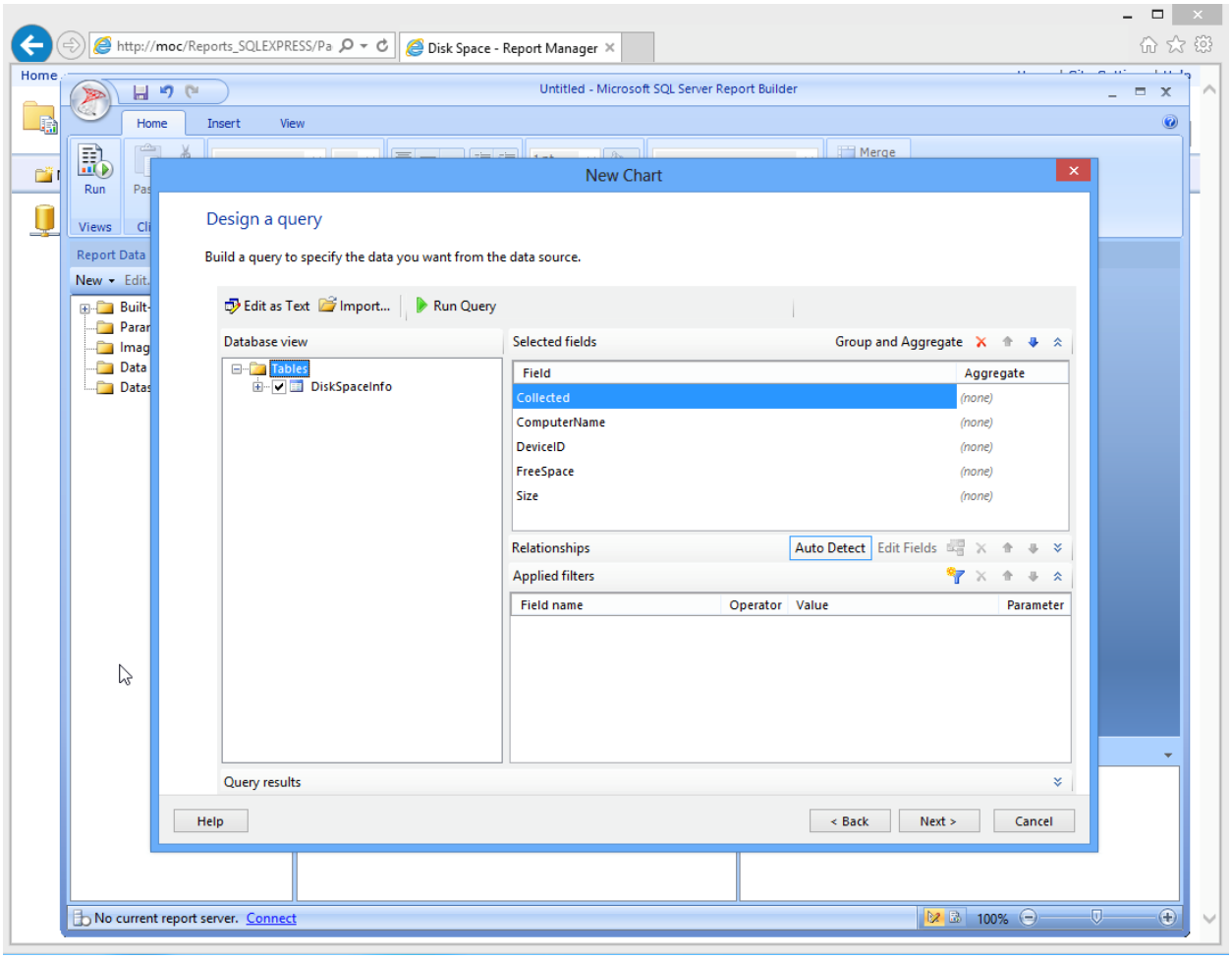
I'm going to pick the Chart Wizard for my new report.



I need to create a new dataset (because this isn't linked to the Reporting Services installation, there's no access to any shared datasets created there).

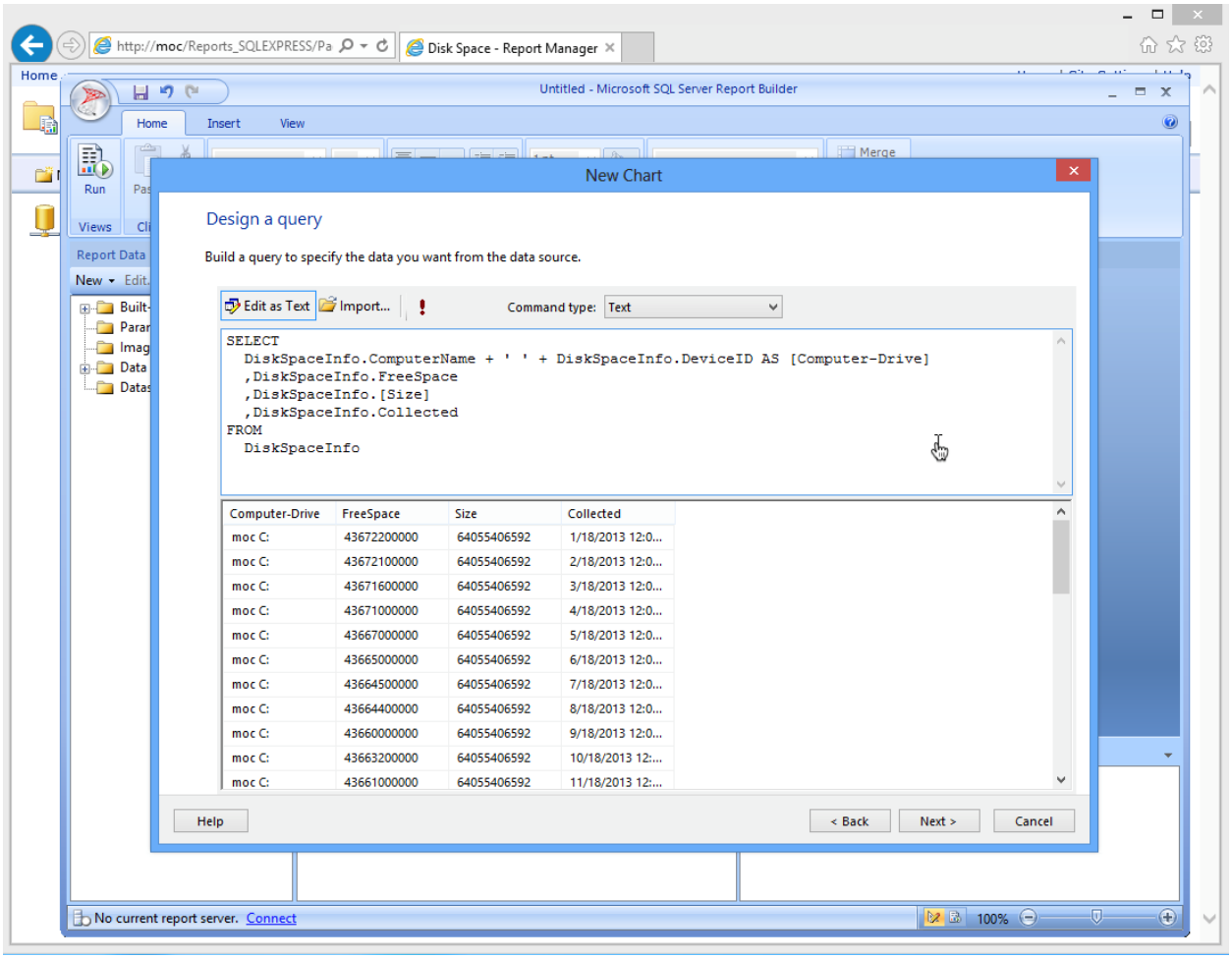


This is a lot like the data source setup we did in Report Manager, which is one reason I wanted to show it to you there first. Always test the connection before proceeding!

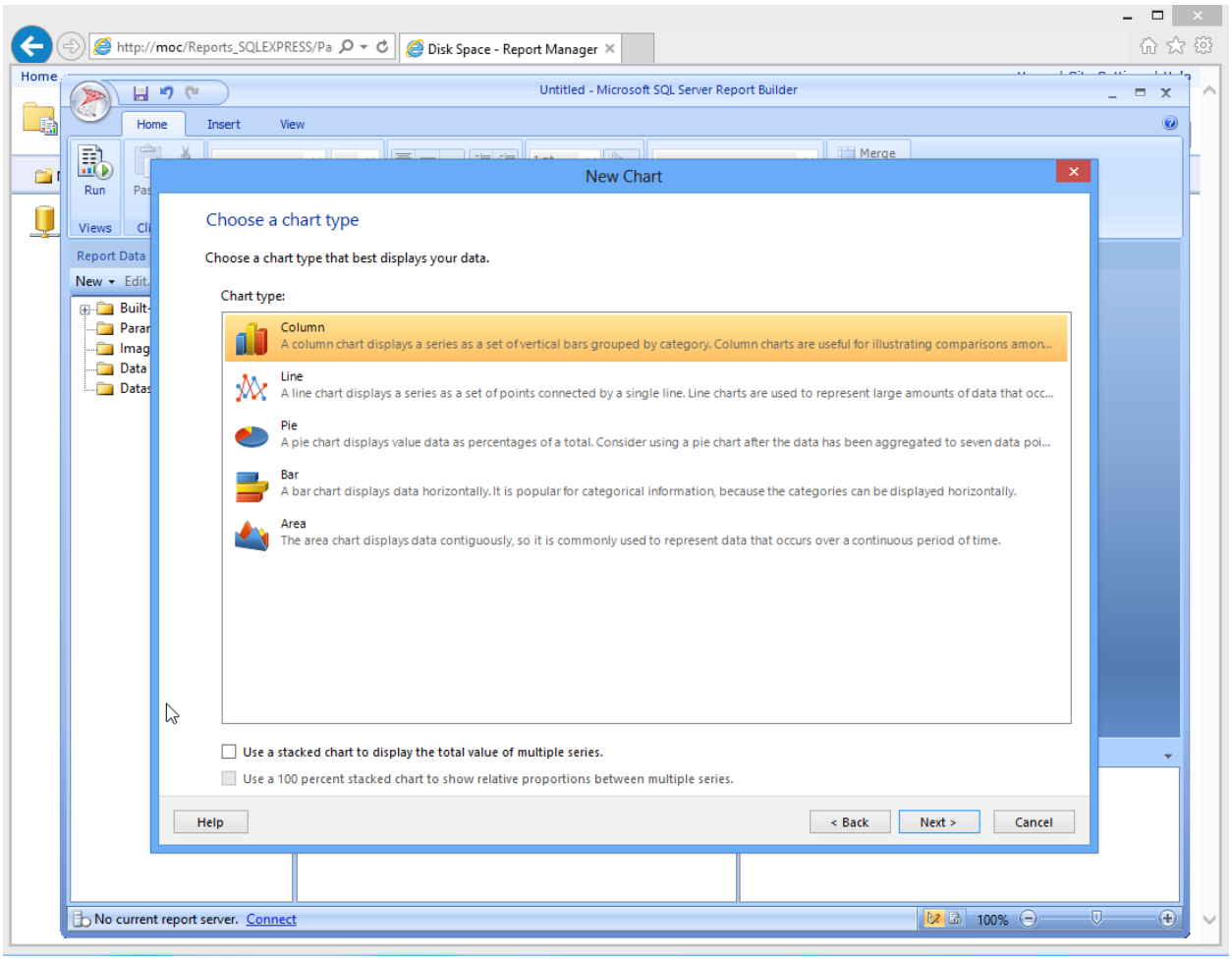


Choose the table that contains the data you need. You can rearrange the fields using the little blue arrows, and then change grouping and aggregating of specific fields (like generating averages, min, max, and so on). Because I have only one table, there are no relationships to mess with, and I don't want to filter out any of the data.

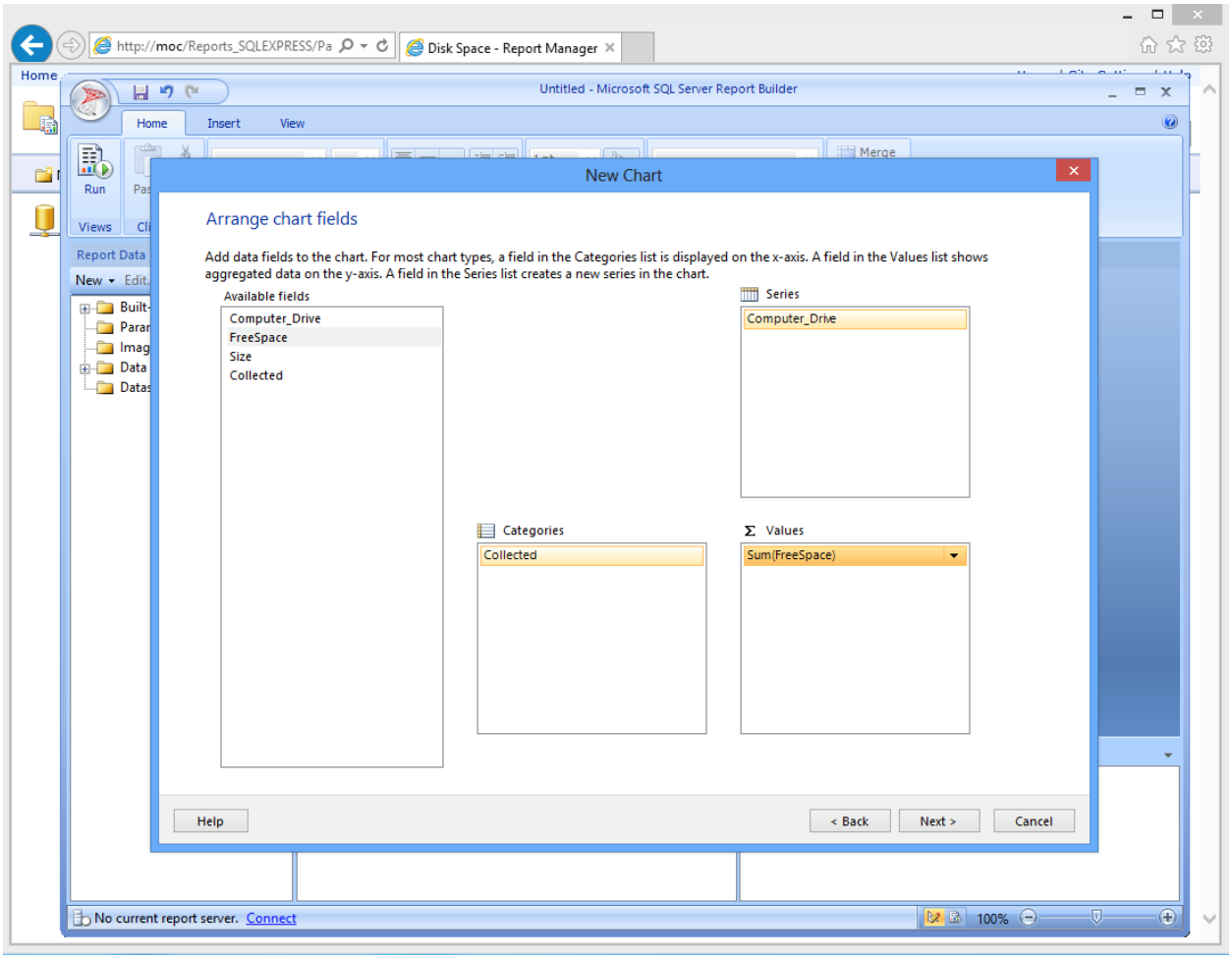
If you're good with SQL Server, you can click "Edit as Text" to manually edit the SQL query instead of using the GUI. I'm actually going to do that, so that I can combine the ComputerName and DeviceID fields into a single field.



I've named this combined field "Computer-Drive," and I clicked the "!" button to run the query and test the results. That's just what I want.

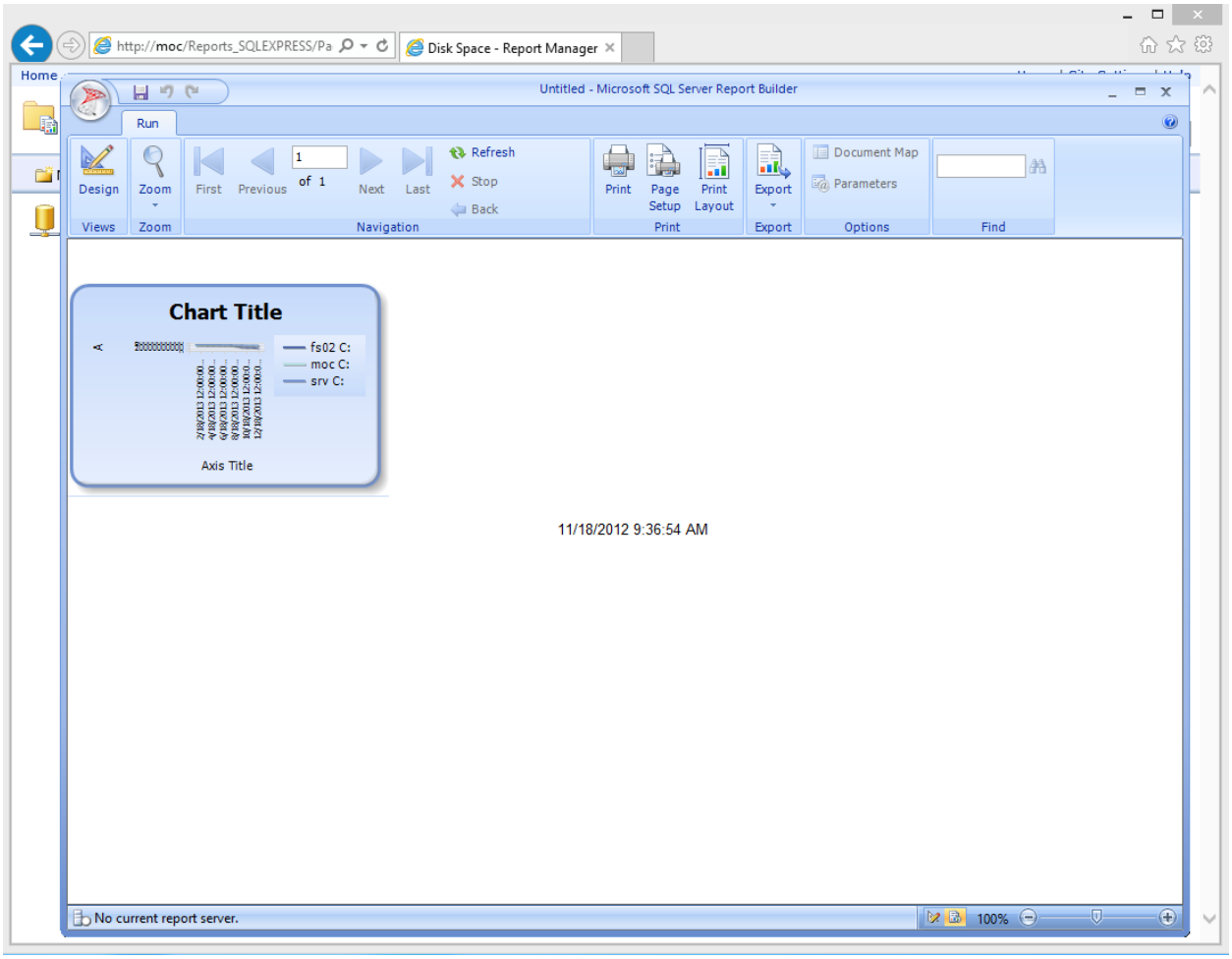


I'm going to choose the Line chart type next, so that I can get a trend line.

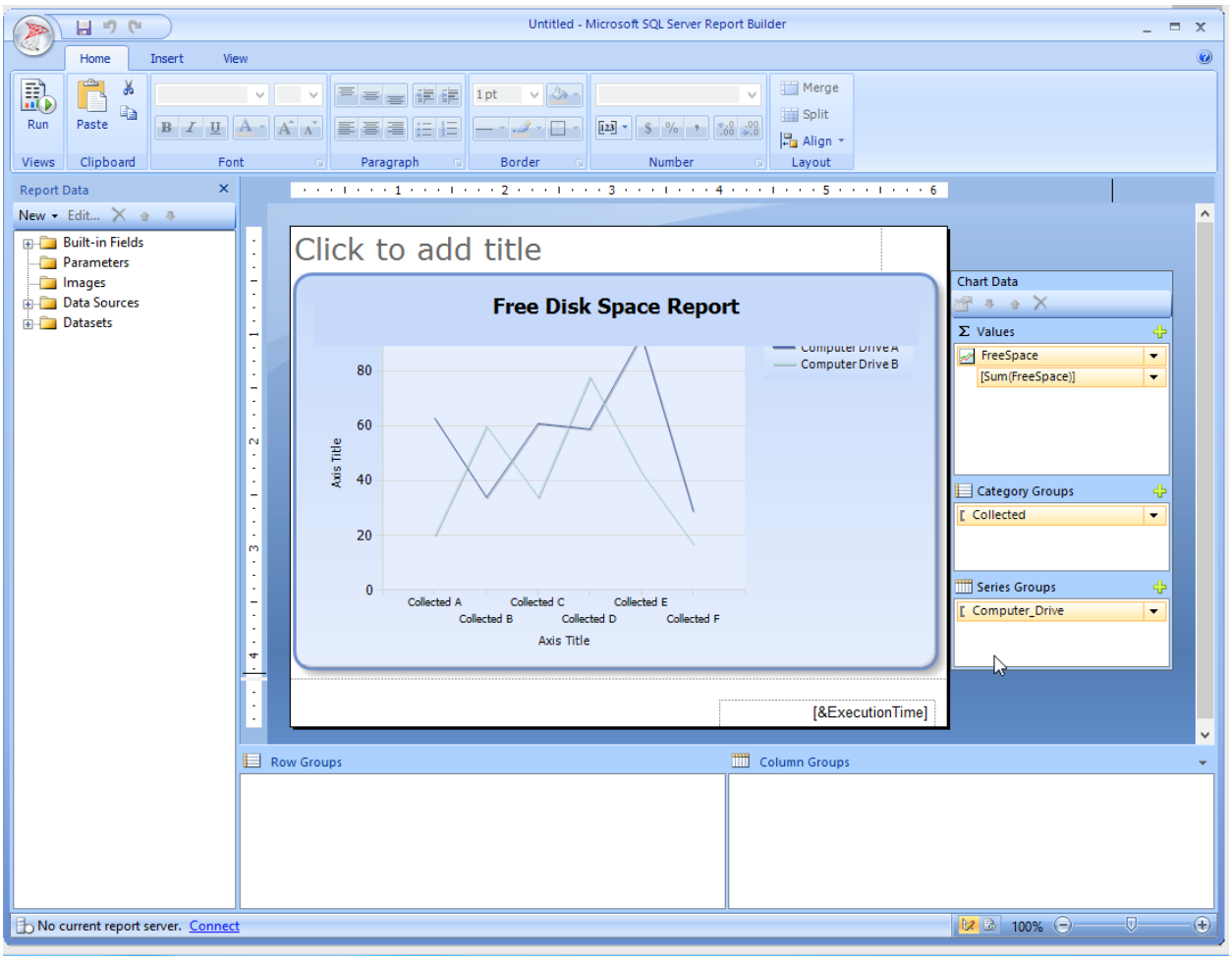


Next, I decide which bits of data go where. The "Computer_Drive" column will be series, meaning each Computer/Drive combination will have its own line on the chart. "Collected" will form the horizontal axis, showing time, and the sum of the FreeSpace *for each value of Collected* will form the vertical axis of the chart. Because I'll only have one free space value per computer per day, there won't actually be any summing happening.

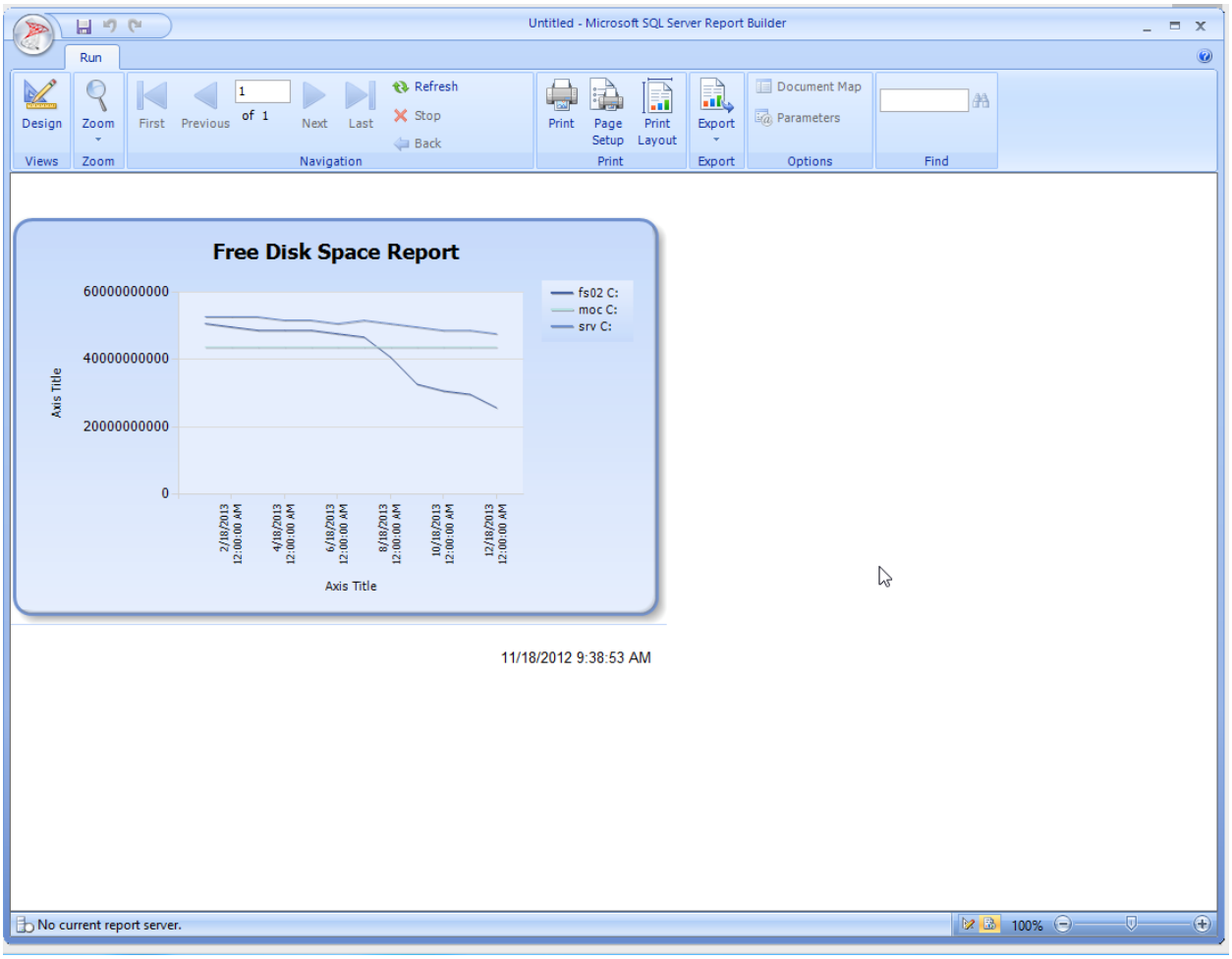
The next screen lets you pick a visual style for the chart.



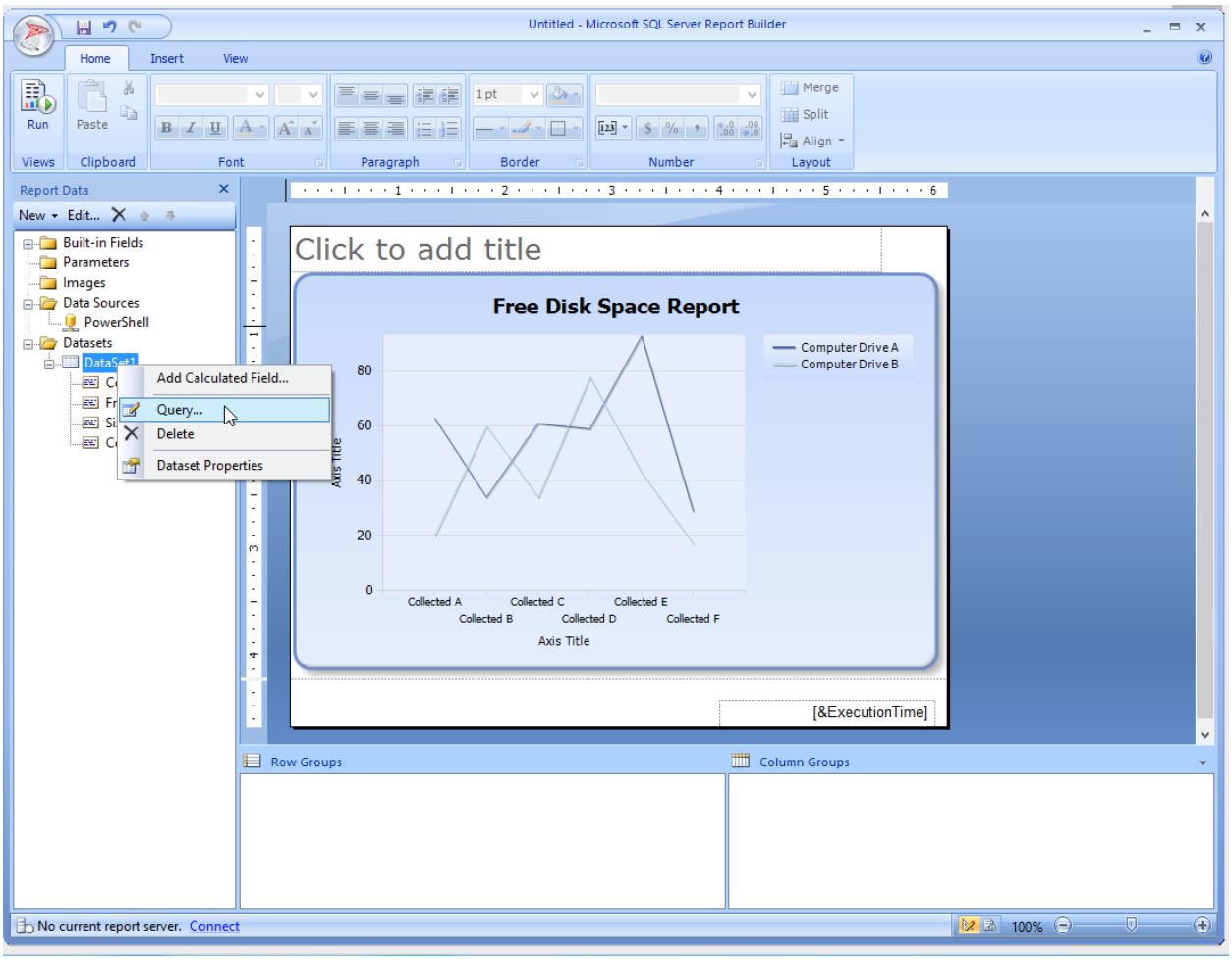
Er, yeah. After running the report (using the "Run" button in the ribbon), I'm not impressed. Back to Design.



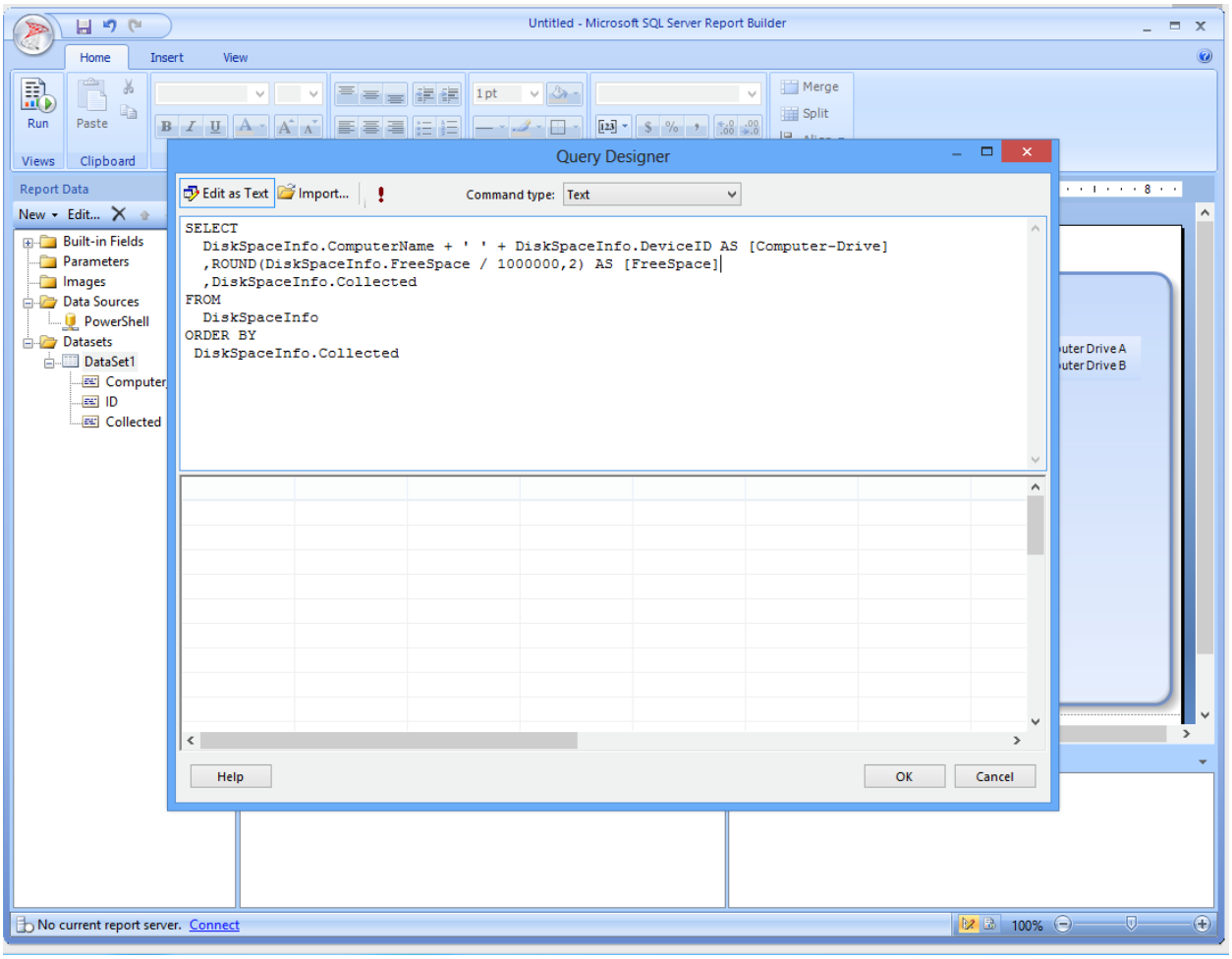
A little resizing, and double-clicking to edit the chart title, might help.



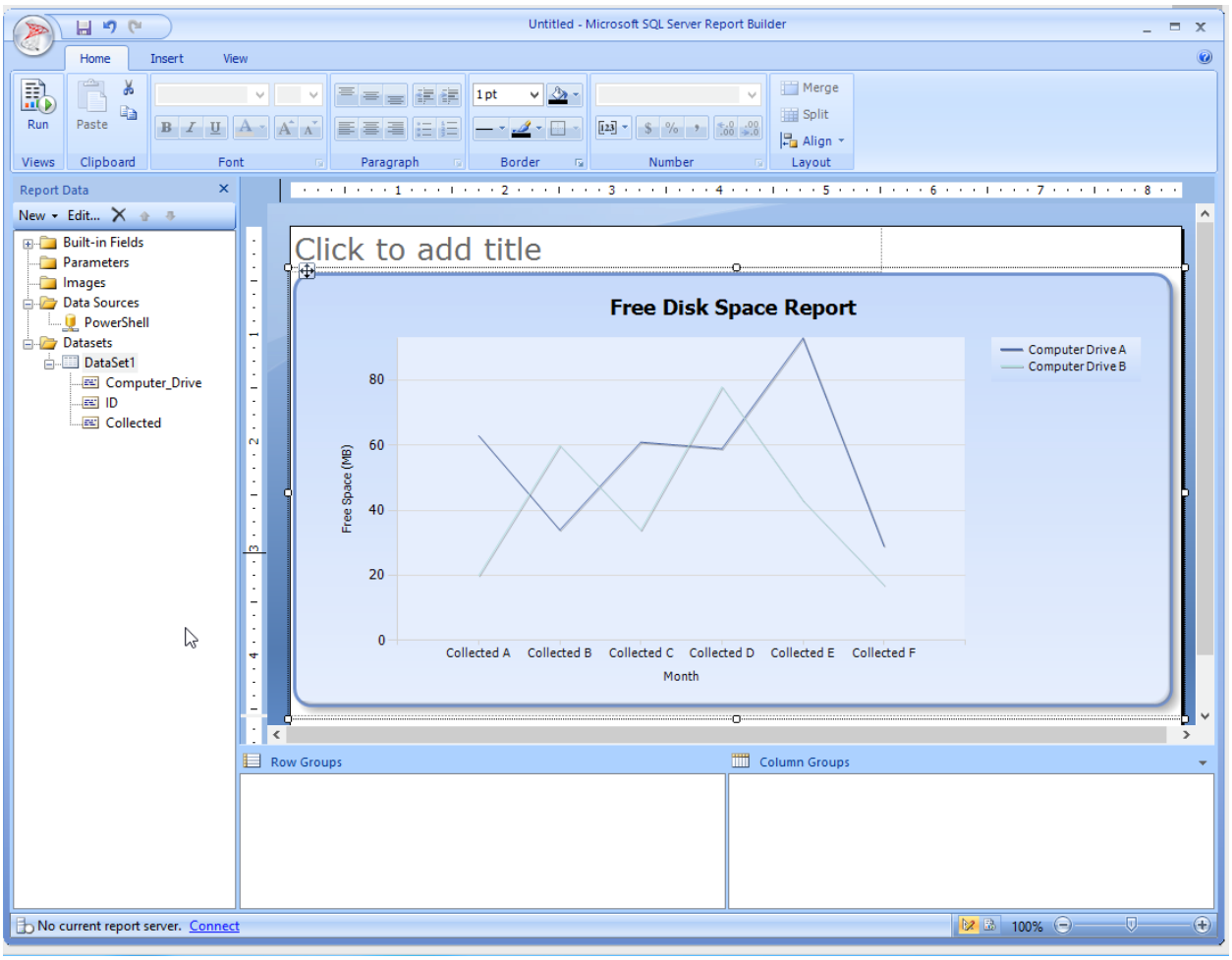
W00t! We're definitely getting there. Now's when I could think of some potential changes to make.



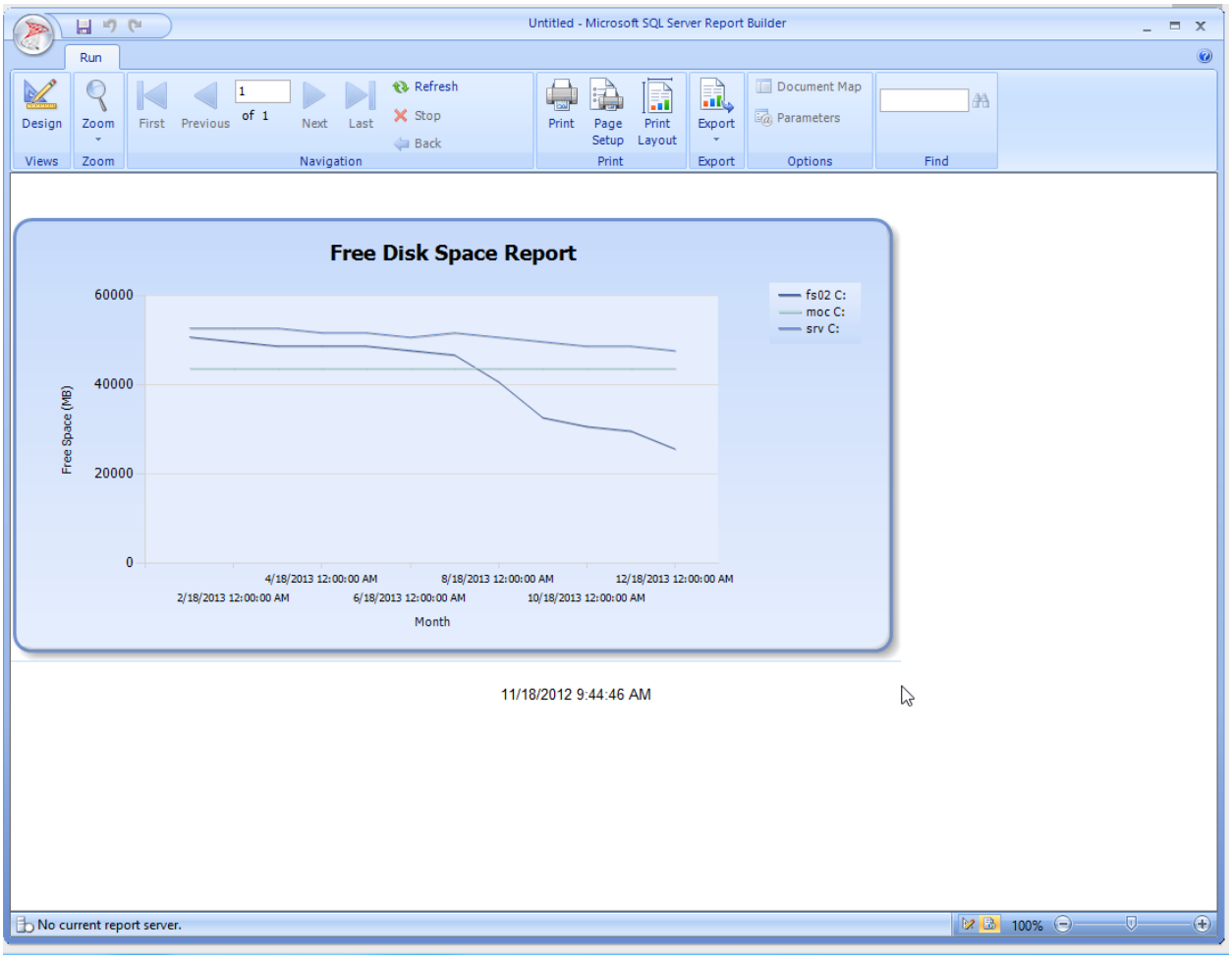
I want to edit my query a bit, so I'll right-click my data set and edit the Query.



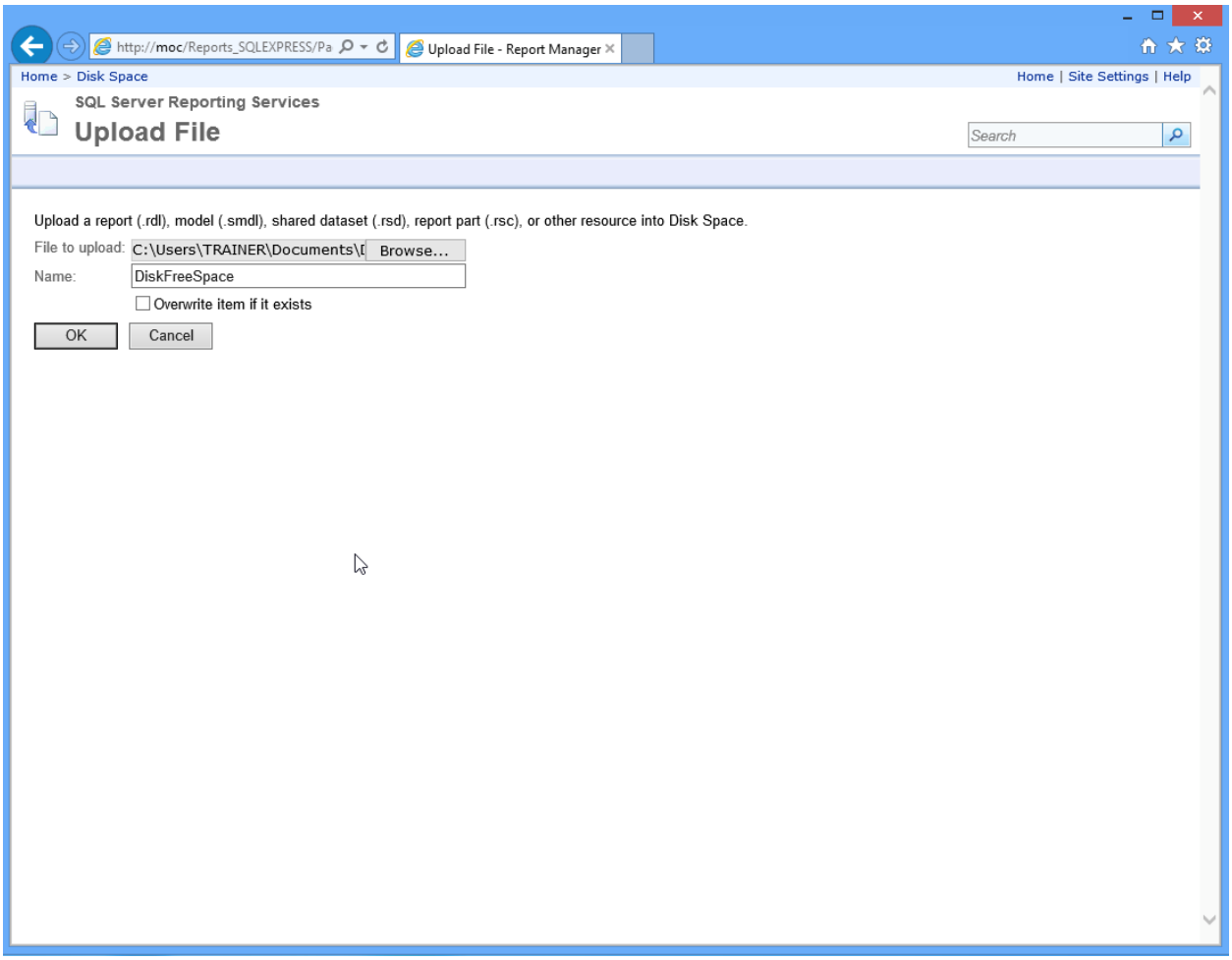
I've removed Size, because I'm not really using it. I've added an ORDER BY clause to make sure the data appears in date order, and I've added a calculation to display free space in megabytes instead of bytes, rounded to two decimal places. Note that I've been careful to name the resulting field the same name - "FreeSpace" - by using the AS option. Because my report is already looking for FreeSpace, it's important that the field continue to exist by that name.



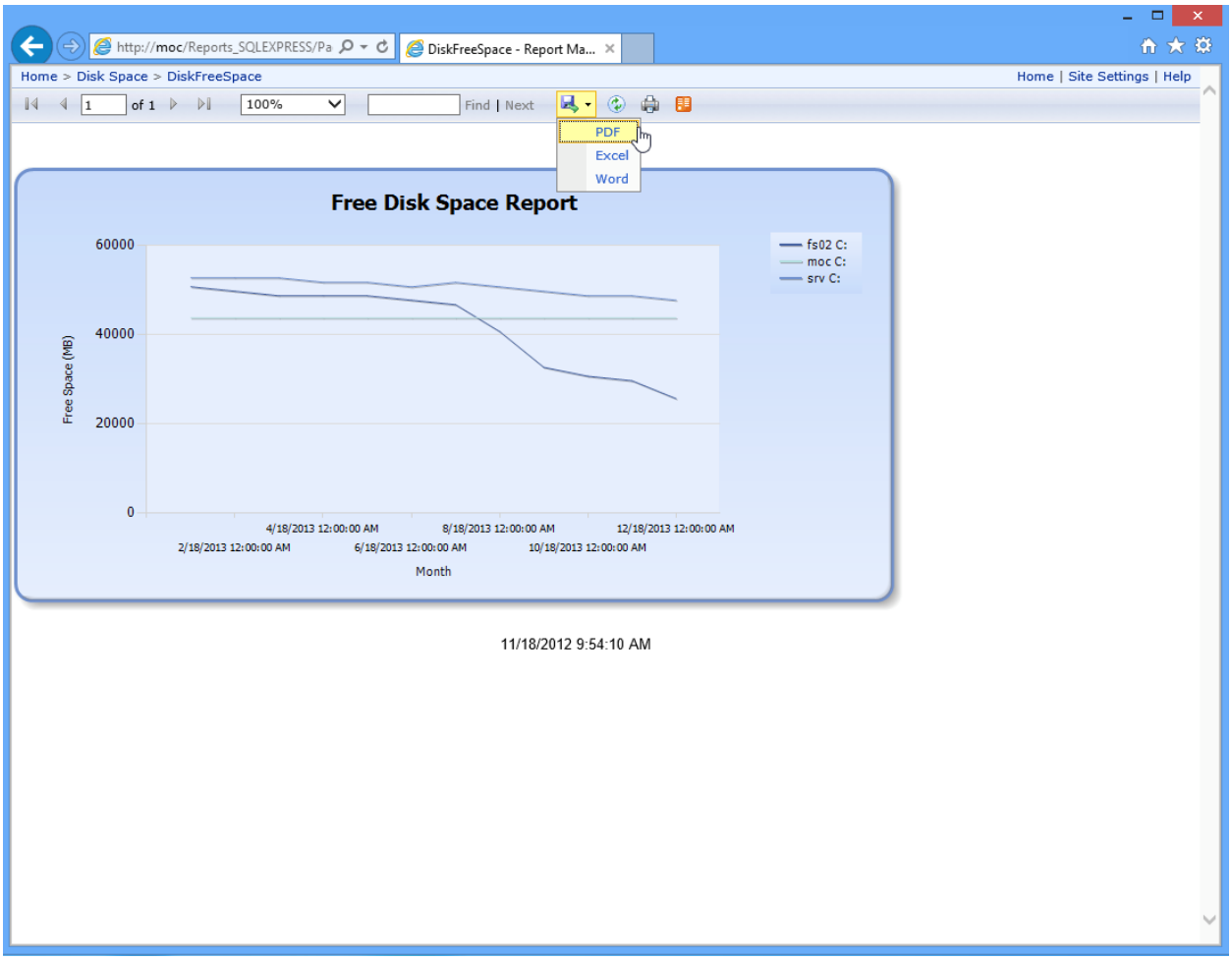
I've edited the axis titles of the chart and made it a bit bigger.



Not bad. I could continue tweaking this - maybe using the T-SQL date/time functions to generate nicer-looking dates along the "X" axis - but let's call it "good" for now. I'll switch back into design mode and save this as DiskFreeSpace.rdl on the file system.



Back in the Web-based Report Manager, in my folder, I'll click Upload File to upload that .RDL file.



I can click the now-uploaded report to run it, and gain access to export options, like PDF.

The Case for "Real" Reporting Services

This Express edition of Reporting Services is missing a few key features. I'll argue that *most* organizations probably already have a full edition of SQL Server someplace, and if you can get SSRS installed (it doesn't cost anything extra, it's just a feature), then you gain access to a lot of awesome abilities - like being able to schedule reports, let people subscribe to them, basically taking everything off your hands once the report is designed.

SSRS also supports "Report Parts," which are kind of like mini-report chunks that can be re-used in other reports. By building a library of such parts, you can then construct meta-reports that contain lots of information. That's why I think SSRS is such a good investment of your time - there's so much you can do to reduce your own future workload. The full SSRS also has a Web-based report designer, meaning you can define data sources (such as those you populate with PowerShell), and then let other folks build their own reports right in a Web browser.

SSRS also integrates with SharePoint, meaning reports and whatnot can be published to a SharePoint installation. Again, this is all about letting you *define* the reports, setting up a routine to put data into tables (which I've hopefully made pretty easy), and then *never touching it again*. Get reporting off your plate entirely by building dashboards and reports that happen automatically, and which report consumers can access on their own.