INSTANT

Short | Fast | Focused

Oracle Database and PowerShell How-to

Utilize the power of Microsoft's powerful scripting engine to automate database tasks with Oracle from PowerShell

Geoffrey Hudik

[PACKT] PUBLISHING

Instant Oracle Database and PowerShell How-to

Utilize the power of Microsoft's powerful scripting engine to automate database tasks with Oracle from PowerShell

Geoffrey Hudik

BIRMINGHAM - MUMBAI

Instant Oracle Database and PowerShell How-to

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2013

Production Reference: 1180113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-858-1

www.packtpub.com

Credits

Author
Geoffrey Hudik

Reviewer
Laurent Schneider

Acquisition Editor
Martin Bell

Commissioning Editor
Maria D'souza

Technical Editors
Priyanka Shah
Vrinda Amberkar

Project Coordinators
Priya Sharma
Esha Thakker

Proofreader
Aaron Nash

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

Cover Image
Conidon Miranda

About the Author

Geoffrey Hudik is a developer with 14 years of experience. He has worked with a variety of companies including Alltel Information Services, JPMorgan Chase, TeamHealth, and Eventbooking.com. These days, he mostly works with C#, ASP.NET MVC and web technologies, Windows development, CTI, XAML, Oracle, SQL Server, PowerShell, Kinect, and mobile technologies including Windows Phone and iOS. He records his thoughts on his blog at `http://geoffhudik.com/tech` and on Twitter (`@thnk2wn`).

I would like to thank Jim Tilson for the expert Oracle help and general support over the years, as well as for much of the script and techniques in the recipe *Automating SQL*Plus (Advanced),* Additional thanks go out to Eddie Frederick, Bryant Brabson, and Jim Christopher for PowerShell assistance, general advice, and inspiration.

About the Reviewer

Laurent Schneider spends his time tuning Oracle databases and developing Oracle applications. When not working on Unix systems, he enjoys using PowerShell to start SQL*Plus scripts and to execute PL/SQL database procedures.

Laurent is working as a Senior Database Administrator for a leading wealth manager in Switzerland.

Laurent is the author of the blog, `http://laurentschneider.com`, and the book, *Advanced Oracle SQL Programming*.

When not at work, you will find Laurent playing chess tournaments or skiing in the Alps with his kids Dora and Loïc.

I would like to thank Priya Sharma for her confidence and her patience. Thanks to the entire Packt Publishing team for their dedication.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface 1

Instant Oracle Database and PowerShell How-to 7

Setting up your environment (Simple) 7
Accessing Oracle (Simple) 8
Connecting and disconnecting (Simple) 12
Retrieving data (Simple) 16
Filtering and exporting data (Simple) 20
Adding records (Simple) 24
Importing sets of records (Medium) 28
Updating and deleting records (Simple) 32
Executing database procedures (Medium) 35
Organizing and invoking scripts (Medium) 40
Script automation and error handling (Medium) 44
Creating reusable script modules (Advanced) 50
Automating SQL*Plus (Advanced) 55
Exploring ODT assemblies (Advanced) 60
Visualizing data (Advanced) 64

Preface

So why use Oracle and PowerShell together in the first place? Often there is a need to automate some work with Oracle that would be tedious to do manually in a large, graphical application such as Toad or SQL Developer. Command-line tools such as SQL*Plus exist, but do not provide the richness that PowerShell offers with its object-oriented pipeline that leverages the .NET framework and core Windows components.

PowerShell offers a lightweight yet very capable alternative to interacting with Oracle in and with Windows. It can be faster than a bloated DMBS GUI application, less work than programming an ad-hoc Oracle utility application, and more powerful than plain text shell scripting environments. PowerShell can be beneficial both for simple, ad-hoc Oracle operations and for complex scripts to automate large batch operations. You do not need to be a programmer to use it; it can be well suited for application developers, DBAs, report writers, data analysts, and other roles.

Oracle can feel more difficult on Windows in general in areas such as installation, configuration, support, stability, and tooling and this can extend to PowerShell. Secondly, unlike with SQL Server, there is no native set of PowerShell cmdlets to ease working with Oracle. With the scripting shell and database coming from competing companies, this may not change soon.

The good news is that through some elbow grease and leveraging various libraries and PowerShell's flexibility, you can manage Oracle from script without too much trouble.

What this book covers

Setting up your environment (Simple), covers the prerequisites such as what you will want to have installed and configured to get started.

Accessing Oracle (Simple), teaches you to access Oracle through loading ODP.NET and other libraries.

Connecting and disconnecting (Simple), covers connecting with and without TNS names, TNS discovery, config file connections, secure credentials, and more.

Retrieving data (Simple), introduces you to the basics of selecting and enumerating data using DataTables, DataReaders, and related objects.

Filtering and exporting data (Simple), builds on the basics from retrieving data with filtering and finding data from the retrieved results and outputting it to different formats.

Adding records (Simple), shows how to insert records and retrieve record IDs.

Importing sets of records (Medium), teaches how to batch import new records from other data stores.

Updating and deleting records (Simple), covers the basics of modifying and removing records.

Executing database procedures (Medium), discusses invoking functions, procedures, and package procedures.

Organizing and invoking scripts (Medium), demonstrates how to organize scripts into functions and reusable script files and how to invoke and debug scripts.

Script automation and error handling (Medium), shows how to run scripts in an automated fashion. It also covers scheduled tasks, transcription, logging, error handling, and notification of results.

Creating reusable script modules (Advanced), takes script files to the next level with more powerful and reusable PowerShell scripting.

*Automating SQL*Plus (Advanced)*, covers executing more complex SQL such as large, combined DDL scripts using SQL*Plus from PowerShell.

Exploring ODT assemblies (Advanced), shows how to use the Oracle Developer Tools for Visual Studio library from PowerShell to take advantage of rich Oracle objects.

Visualizing data (Advanced), shows how to use libraries to visualize data from Oracle with graphs, charts, and so on.

What you need for this book

You should have a solid understanding of Oracle, SQL, and relational database concepts before getting started. You should also have at least a basic familiarity with PowerShell; while you can learn PowerShell from this book, the focus is on using Oracle from PowerShell and not on introducing PowerShell basics. It will also be helpful if you have any experience with Microsoft's .NET framework but this is not required. The focus of this book will be interacting with Oracle libraries from PowerShell, and not Oracle concepts or PL/SQL development.

The first recipe, *Setting up your environment*, will address software and setup requirements for getting started. Some later chapters will have additional requirements when applicable.

Who this book is for

The primary target audiences for this book are application developers who work with Oracle databases and Oracle DBA type roles. Other applicable audiences might include report developers, data analysts, data architects, or QA personnel.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Invoke `Select-TNS` to list all entries or pipe it to `Where-Object` to filter it down."

A block of code is set as follows:

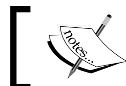
```
function Select-TNS
{
    $enu = New-Object Oracle.DataAccess.Client.
OracleDataSourceEnumerator
    Write-Output $enu.GetDataSources()
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

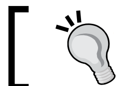
```
$dt = New-Object System.Data.DataTable
[void]$da.Fill($dt)
return ,$dt
```

Any command-line input or output is written as follows:

```
PS > Set-ExecutionPolicy RemoteSigned
```



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant Oracle Database and PowerShell How-to

Welcome to *Instant Oracle Database and PowerShell How-to*. In this book you will learn how to interact with Oracle databases from PowerShell, including connection and discovery, querying, and modifying data and objects, and executing database logic. With the help of this book you can avoid manual labor and painful applications, and start leveraging the strengths of PowerShell scripting and .NET to automate Oracle database tasks.

Setting up your environment (Simple)

Before jumping into PowerShell, make sure your system is ready to begin.

Getting ready

At a minimum, you will want the following to get started with Oracle and PowerShell:

- **Accessible Oracle database**: The examples in this book are tested primarily against the sample HR schema provided with *Oracle 11g Express edition*, a free download from `www.oracle.com`.
- **PowerShell 2.0** (or later): This is included with Windows 7 and Windows Server 2008 R2, and also available as a free download for older Windows OS versions.
- **Oracle Data Access Components** (**ODAC**): This is a free download from `www.oracle.com`. Preferably use Oracle Developer Tools for Visual Studio (for later use in this book). Examples are tested primarily against using the 32 bit ODAC distribution. The 64 bit version is not required for use on a 64 bit OS.

- **DBMS Application**: A DBMS tool is technically optional but will help in browsing and verifying results. **Toad for Oracle** is a popular choice; another one is SQL Developer, a free download from `www.oracle.com`.

How to do it...

Follow these steps to run PowerShell scripts:

1. Ensure you start an x86 PowerShell host session on a 64 bit OS if the 32 bit version of ODAC is installed; otherwise use an x64 session.
2. In PowerShell, ensure output of `Get-ExecutionPolicy` is not `Restricted`. If it is, run the following command:

   ```
   PS > Set-ExecutionPolicy RemoteSigned
   ```

How it works...

With these tools, you will have what you need for most of the examples in this book.

By configuring PowerShell's execution policy in this manner you will have necessary permission to run scripts and load configuration files.

There's more...

Some later examples will specify additional setup as required.

For more on execution policy, run the following help command in PowerShell:

```
PS > get-help set-executionpolicy -detailed
```

Accessing Oracle (Simple)

In this recipe we will look at loading libraries to create objects to interact with Oracle.

Getting ready

Open a PowerShell session for the appropriate processor architecture, x86 or x64 according to your ODAC installation. See the preceding recipe, *Setting up your environment (Simple)* for more information. Often it is easiest to start with the PowerShell **ISE** (**Integrated Scripting Environment**) for a better editing and debugging experience.

How to do it...

1. The Oracle Data Provider for .NET (`ODP.NET`) is included with ODAC in `Oracle.DataAccess.dll` and is generally the best choice for accessing Oracle. The first step is loading this library into memory:

   ```
   $odpAssemblyName = "Oracle.DataAccess, Version=2.112.3.0,
   Culture=neutral, PublicKeyToken=89b483f429c47342"
   [System.Reflection.Assembly]::Load($odpAssemblyName)

   GAC    Version        Location

   ---    -------        --------

   True   v2.0.50727     C:\Windows\assembly\GAC_32\Oracle.DataAccess
   \2.112.3.0__89b483f429c47342\Oracle.DataAccess.dll
   ```

2. Once the assembly is loaded you can start creating types. If you are not sure what object types are available, use the following code to enumerate public types in the `Oracle.DataAccess` assembly.

   ```
   $asm = [appdomain]::currentdomain.getassemblies() | where-object
   {$_.FullName -eq $odpAssemblyName}
   $asm.GetTypes() | Where-Object {$_.IsPublic} | Sort-Object {$_.
   FullName } | ft FullName, BaseType | Out-String
   ```

3. With the assembly loaded, and the types to create known, you can now start creating objects. A logical place to start is with a connection object:

   ```
   $conn = New-Object Oracle.DataAccess.Client.OracleConnection
   ```

How it works...

First the .NET Framework's `Assembly.Load` method is called and a fully qualified assembly name is specified. By default PowerShell 2.0 runs against the .NET Framework 2.0 so the 2.x version of Oracle.DataAccess is specified in `$odpAssemblyName`. The parts of the assembly name can be found by inspecting `%WINDIR%\assembly\` in Windows Explorer. Because the output was not explicitly captured or discarded, details of the loaded assembly are shown.

Next, the `AppDomain.CurrentDomain.GetAssemblies` method is invoked and the list is filtered down to the Oracle.DataAccess assembly. This verifies that the assembly loaded and provides a reference to it. Finally it gets the types in that assembly, narrows them down by public types, sorts the data by full name, and formats the resulting table with the full name and base type properties of each Oracle type that is available.

There's more...

To discard the output of the assembly load, use either `[void]` or a pipe to `Out-Null`.

```
[System.Reflection.Assembly]::Load($odpAssemblyName) | Out-Null
[void][System.Reflection.Assembly]::Load($odpAssemblyName)
```

Alternatively, capture the output to a variable and format another way:

```
$odpAsm = [System.Reflection.Assembly]::Load($odpAssemblyName)
"Loaded Oracle.DataAccess from {0}" -f $odpAsm.Location
```

Here are three other ways of loading ODP.NET:

- `[Reflection.Assembly]::LoadWithPartialName("Oracle.DataAccess")`

  ```
  $filename = "C:\Oracle\product\11.2.0\client_1\odp.net\bin\2.x\
  Oracle.DataAccess.dll"
  ```

- `[void][Reflection.Assembly]::LoadFile($filename)`
- `[void][Reflection.Assembly]::LoadFrom($filename)`

`LoadWithPartialName` is convenient but blindly loads an unknown version of `ODP.NET`. Both `LoadFile` and `LoadFrom` can also lead to issues. See `http://msdn.microsoft.com/en-us/library/dd153782.aspx` for assembly loading best practices. Using `Load` with an explicit, full assembly identity is safest, despite more typing.

Now, let's find out how to get help with `ODP.NET`, and some other ways of accessing Oracle.

Getting help with ODP.NET

Oracle's ODP documentation can be accessed via the Start Menu, with Windows Explorer under `ORACLE_BASE\ORACLE_HOME\ODACDoc\DocumentationLibrary`, or online at `http://docs.oracle.com`.

Also, PowerShell's `Get-Member` (alias `gm`) is useful for inspecting available members of objects; for example, use `$conn | gm` to list all the properties and methods available on the `OracleConnection` that was created.

Loading ODP.NET 4.0

The ODAC installation installs two versions of Oracle.DataAccess, one for .NET Framework 2 and another for PowerShell 2.0, which was built before .NET Framework 4.0. As such, you cannot directly load Version 4.0 assemblies by default. Generally the Version 2.0 will suffice. If there are specific 4.0 features you want, you have a few options:

- Change a global registry setting that forces all .NET 2.0 assemblies to run under .NET 4 (from PowerShell or otherwise); this has a number of serious consequences and should almost always be avoided.

- Create or update `powershell.exe.config` and `powershell_ise.config` in `$pshome` (that is `%WINDIR%\SysWOW64\WindowsPowerShell\v1.0`) for x86 and/or x64 to run PowerShell sessions under .NET 4.0; this is better than the first option, but still applies globally. See `http://poshcode.org/2045` for details.
- Use custom script code to selectively run a given script block or file as a .NET 4.0 child process from within PowerShell's .NET 2.0 context. This has the least potential impact but can make running and debugging more difficult. For example: `https://gist.github.com/882528`.
- Use PowerShell 3.0, which uses .NET 4.0 by default. This is the best option.

If PowerShell 3.0 is available that is the path of least resistance. Otherwise the PowerShell configuration modifications are most likely the next best route, unless you have various other PowerShell scripts or cmdlets (built-in or custom) that may not execute correctly when run under the .NET Framework 4.0 runtime.

Once configured, load the assembly as before but specify the 4.X version number.

```
$odpAssemblyName = "Oracle.DataAccess, Version=4.112.3.0,
Culture=neutral, PublicKeyToken=89b483f429c47342"
[System.Reflection.Assembly]::Load($odpAssemblyName)
```

To get the assembly name and other details here, you can open `%WINDIR%\Microsoft.NET\assembly` in Windows Explorer, navigate to `Oracle.DataAccess.dll`, and open the DLL in a dissembler such as dotPeek by JetBrains.

Using OLE DB

One alternative to ODP.NET is using **OLE DB** (included with ODAC). In this case the assembly is already loaded; just start creating the types.

```
$conn = New-Object System.Data.OleDb.OleDbConnection
```

The only real advantage of OLE DB is your scripts are more portable in that your data access script logic can be reused for databases other than Oracle. This comes at the cost of slow performance with COM interop, the loss of some Oracle optimizations, and less functionality than ODP.NET.

Using Microsoft's Data Provider for Oracle

Microsoft's .NET Framework Data Provider for Oracle (`System.Data.OracleClient`) is deprecated and generally should not be used. It does provide a more lightweight footprint than a full ODAC install but has a number of performance and functionality limitations and is not being maintained.

```
[System.Reflection.Assembly]::Load("System.Data.OracleClient,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089")
$conn = New-Object System.Data.OracleClient.OracleConnection
```

Other ways to access Oracle

Using an **ODBC** driver (included with ODAC) is an older technique but could be useful in conjunction with systems that do not support newer drivers. Another option is loading a .NET data access DLL from PowerShell. This could be either a standard .NET class library or one that references `System.Management.Automation.dll` and exposes cmdlets for friendly PowerShell usage.

You can also use Oracle's SQL*Plus client from PowerShell to send commands and receive outputs via pipes. This book will focus primarily on using ODP.NET directly from PowerShell. Despite some heft, ODP.NET provides the best performance and functionality and staying in a PowerShell script context prevents more involved work of developing compiled DLLs.

Connecting and disconnecting (Simple)

In this recipe we will explore making connections to Oracle.

Getting ready

Start by gathering any database connection details such as server and user information. The example connection string is against the HR sample database included with Oracle Express. To connect to this database you may need to adjust the user ID and/or password in the connection string according to how you set it up. Refer to the *Oracle Database Express Edition Getting Started Guide* for more information.

How to do it...

The code in this section assumes that ODP.NET has already been loaded as described in the preceding recipe. Let's look at connecting without using **TNS** names:

```
function Connect-Oracle([string] $connectionString = $(throw
"connectionString is required"))
{
    $conn= New-Object Oracle.DataAccess.Client.OracleConnection($conn
ectionString)
    $conn.Open()
    Write-Output $conn
}

function Get-ConnectionString($user, $pass, $hostName, $port, $sid)
{
    $dataSource = ("(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST={0})
(PORT={1}))(CONNECT_DATA=(SERVICE_NAME={2})))" -f $hostName, $port,
$sid)
    Write-Output ("Data Source={0};User Id={1};Password={2};Connection
```

```
Timeout=10" -f $dataSource, $user, $pass)
}

$conn = Connect-Oracle (Get-ConnectionString HR pass localhost 1521
XE)
"Connection state is {0}, Server Version is {1}" -f $conn.State,
$conn.ServerVersion
$conn.Close()
"Connection state is {0}" -f $conn.State
```

How it works...

First a `Connect-Oracle` function is defined to create and open a connection. It requires a connection string by throwing an error if it is not set. An `OracleConnection` objection is then created using that connection string and the connection is opened. Finally, the connection object is written to output with `Write-Output $conn`; this could have also been done with `return $conn` or just `$conn`. Using `return` can be deceiving because any output of the function that is not captured will become part of the function return value and not just what follows the `return` statement.

`Get-ConnectionString` takes in the key server and user data as arguments and uses the `-f` format operator to concatenate a connection string for ODP.NET.

Next these functions are invoked with `$conn = Connect-Oracle (Get-ConnectionString HR pass localhost 1521 XE)`; note the parentheses to ensure the appropriate order of operations and output. Also note that you may need quotes for some argument values in certain cases (that is, `$pass`). Finally the connection will be closed, with some connection properties being output both before and after. The connection code could be reduced but breaking the steps into functions makes the script more maintainable and reusable.

There's more...

For connection string help, try `http://connectionstrings.com/oracle`.

Let's explore some additional connection techniques to use.

Connecting with TNS Names

To connect via TNS you can change the `$dataSource` variable in `Get-ConnectionString` to point to a valid TNS Names alias.

1. For example, `$dataSource = "LOCALDEV"` with `LOCALDEV` defined in `ORACLE_BASE\ORACLE_HOME\Network\Admin\tnsnames.ora` can be defined as follows:

   ```
   LOCALDEV =
     (DESCRIPTION =
       (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1521))
   ```

```
    (CONNECT_DATA =
      (SERVICE_NAME = xe)
    )
  )
```

2. TNS entries can be browsed using a function as follows:

```
function Select-TNS
{
    $enu = New-Object Oracle.DataAccess.Client.
OracleDataSourceEnumerator
    Write-Output $enu.GetDataSources()
}
```

3. Invoke `Select-TNS` to list all entries or pipe it to `Where-Object` to filter it down:

```
Select-TNS | where-object {$_.InstanceName -like '*DEV*'} | ft
```

Reading connection strings from config files

1. Often connection strings are stored in app config files, such as the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="AppConnect"
         connectionString="Data Source=LOCALDEV;User
Id=HR;Password=pass;Connection Timeout=10"/>
  </connectionStrings>
</configuration>
```

You can easily read the connection string from such a file as follows:

```
function Get-ConfigConnectionString(
    [string] $filename = $(throw "filename is required"),
    [string] $name = $(throw "connection string name is
required"))
{
    $config = [xml](gc $filename)
    $item = $config.configuration.connectionStrings.add | where
{$_.name -eq $name}
    if (!$item) { throw "Failed to find a connection string with
name '{0}'" -f $name}
    return $item.connectionString
}

$connectString = Get-ConfigConnectionString .\App.config
AppConnect
```

The `Get-ConfigConnectionString` function uses `gc` (an alias for `Get-Content`) to read the file's XML text and then creates an `XmlDocument` from the result with `[xml]`. It then looks for and returns the specified connection string, or throws an exception if the specified connection was not found.

2. Some connection strings may be stored in `machine.config` under `%WINDIR%\Microsoft.NET\Framework\[Version]\Config\` where `[Version]` is the .NET Framework version (i.e. v2.0.50727 or v4.0.30319). These can be accessed with `Get-ConfigConnectionString` or via using the `ConfigurationManager` class:

```
$config = [System.Configuration.ConfigurationManager]::OpenMachine
Configuration()
$connectString = $config.ConnectionStrings.
ConnectionStrings["AppConnect"]
```

If securing connection strings is a concern, you might use `Secure-String` if the account encrypting and decrypting is the same, or use `Library-StringCrypto.ps1` by Steven Hystad.

Prompting for connection credentials

If a user-interactive script is acceptable there may be times where you want to prompt for credentials to use in building a connection string:

```
function Get-Password
{
    $securePass = Read-Host "Enter Password" -AsSecureString
    $bstr = [System.Runtime.InteropServices.Marshal]::SecureString
ToBSTR($securePass)
    $pass = [System.Runtime.InteropServices.Marshal]::PtrToStringA
uto($bstr)
    [System.Runtime.InteropServices.Marshal]::ZeroFreeBSTR($bstr)
    return $pass
}

$user = read-host "Enter username"
$pass = Get-Password
$connectString = "Data Source=LOCALDEV;User
Id={0};Password={1};Connection Timeout=10" -f $user, $pass
```

This script defines a `Get-Password` function that uses the `–AsSecureString` switch of `Read-Host` to get a secure password string. It then does a couple of conversions and cleanup using methods in `System.Runtime.InteropServices.Marshal` class to return a plain text version of the secure password.

It then prompts for a username using `Read-Host`, invokes the function to get the password, and injects those values into the connection string.

Retrieving data (Simple)

In this recipe we will look at the basics of retrieving data from Oracle in PowerShell.

Getting ready

Use code from the preceding recipes to load ODP.NET and establish a connection.

How to do it...

1. First create a function to return a .NET `DataTable` instance with a connection and a `SQL SELECT` statement:

```
function Get-DataTable
{
Param(
    [Parameter(Mandatory=$true)]
    [Oracle.DataAccess.Client.OracleConnection]$conn,
    [Parameter(Mandatory=$true)]
    [string]$sql
)
    $cmd = New-Object Oracle.DataAccess.Client.
OracleCommand($sql,$conn)
    $da = New-Object Oracle.DataAccess.Client.
OracleDataAdapter($cmd)
    $dt = New-Object System.Data.DataTable
    [void]$da.Fill($dt)
    return ,$dt
}
```

2. Connect as in the preceding recipe and call the function passing the connection and SQL:

```
$conn = Connect-Oracle (Get-ConnectionString)
$dt = Get-DataTable $conn "select employee_id, first_name, last_
name, hire_date from employees where job_id = 'SA_MAN'"
```

3. Output a heading showing how many records were returned and then the raw data:

```
"Retrieved {0} records:" -f $dt.Rows.Count
$dt | ft -auto
```

4. Iterate through the data and perform some processing:

```
foreach ($dr in $dt.Rows)
{
    $eligible = [DateTime]::Now.AddYears(-5) -ge $dr.hire_date

    if ($eligible)
    {
        Write-Output ("{0} {1} is eligible" -f $dr.first_name,
$dr.last_name)
    }
}
```

5. When finished, close the connection:

```
$conn.Close()
```

6. Run the script. The output will vary by date but should be similar to the following:

```
Retrieved 5 records:

EMPLOYEE_ID FIRST_NAME LAST_NAME HIRE_DATE
----------- ---------- --------- ---------
        145 John       Russell   10/1/2004 12:00:00 AM
        146 Karen      Partners  1/5/2005 12:00:00 AM
        147 Alberto    Errazuriz 3/10/2005 12:00:00 AM
        148 Gerald     Cambrault 10/15/2007 12:00:00 AM
        149 Eleni      Zlotkey   1/29/2008 12:00:00 AM

John Russell is eligible
Karen Partners is eligible
Alberto Errazuriz is eligible
```

How it works...

`Get-DataTable` first defines the connection and SQL parameters more formally within a `Param` block and uses a `Parameter` attribute before each to specify additional metadata indicating that the parameters are required (preferred over using `throw`).

Next, an `OracleCommand` object is created using the connection and SQL string parameters passed in. That command is passed in as the `selectCommand` argument when constructing the `OracleDataAdapter` on the next line. A new `DataTable` object is then created and it is populated using the data adapter with `[void]$da.Fill($dt)`; `void` prevents the `Fill` output so the function only has a single value being returned. The adapter can also fill a `DataSet` when multiple tables are involved.

The last line of the function returns the `DataTable` using `return ,$dt`; the comma ("unary comma") is important as it prevents PowerShell from unrolling the collection of rows of `DataTable`. By default PowerShell unrolls enumerable types, meaning that without the comma the return type of this method would actually be `Object[]` (the `Object` array of `DataRow`) or a `DataRow` if only one record.

Generally this unrolling is desirable, as it is useful with piping so script blocks downstream in the pipeline can start operating on items one at a time for faster output and perceived performance. In the case of this script this is avoided since a small amount of data is completely loaded in memory, there is no heavy processing, and the function implies returning a consistent type of `DataTable` (with all of its functionality available).

`Get-DataTable` is passed the open connection object and an inline SQL string; for more complex SQL you might want to use a multiline string or read the SQL from a file. The output from the function is captured in the `$dt` variable and the script outputs the count of rows and then uses `$dt | ft -auto` to format the `DataTable` output as a table with the columns auto-sized to the data.

Finally the script uses a `foreach` loop to iterate through each `DataRow` and perform processing and close the connection when finished. The processing here is an arbitrary calculation of eligibility based on being hired for 5 years or longer, with output based on being eligible. Note that with PowerShell we were able to use `$dr.hire_date` to refer to the `hire_date` field without having to use `$dr["hire_date"]`.

There's more...

There are a few more basics to look at including other ways to iterate the data and handling null checking.

Unrolling and null checking

Let's look at an example where we are not preventing the unrolling of the collection:

1. First we replace `return ,$dt` in `Get-DataTable` with `return $dt` (removing the comma).
2. Next we change the code between the `Connect-Oracle (Get-ConnectionString)` and `$conn.Close()` lines to the following:

```
$sql = "select city, state_province from locations order by city,
state_province"

Get-DataTable $conn $sql | foreach {
    $cityState = $_.city
    if ($_.state_province -ne [DBNull]::Value)
    {
        $cityState += ", " + $_.state_province
```

```
    }
    Write-Output $cityState
}
```

In this example each `DataRow` that was unrolled from the `DataTable` is piped to the `ForEach-Object` cmdlet (`foreach` alias) one at a time. Note that all the data is still loaded into memory inside the `DataTable` in one database call; this just iterates over them in a "lazy fashion".

For more information on the differences between the `foreach` statement used in the first example with the `ForEach-Object` cmdlet used here, see: `http://poshoholic.com/2007/08/21/essential-powershell-understanding-foreach/`.

The other item of note in this script is the State/Province field is checked to ensure it is not equal to null, using `[DBNull]::Value` and not `$null`.

Using a DataReader

At times it may be beneficial to read one record at a time from Oracle, usually when iterating through a large number of records.

1. We can do this using a function similar to `Get-DataTable` but returning a `DataReader` instead:

```
function Get-DataReader
{
Param(
    [Parameter(Mandatory=$true)]
    [Oracle.DataAccess.Client.OracleConnection]$conn,
    [Parameter(Mandatory=$true)]
    [string]$sql
)
    $cmd = New-Object Oracle.DataAccess.Client.
OracleCommand($sql,$conn)
    $reader = $cmd.ExecuteReader()
    return ,$reader
}
```

2. We invoke the function in a similar fashion:

```
$sql = "select city, state_province from locations order by city,
state_province"
$reader = Get-DataReader $conn $sql
```

The iteration changes to use `$reader.Read()` and string fields are read using the `DataReader`'s `GetString` method, combined with `GetOrdinal` to get the position

of the field, and `IsDBNull` to check for null values:

```
while ($reader.Read())
{
    $city = $reader.GetString($reader.GetOrdinal("city"))
    $stateProvince = $null

    if (!$reader.IsDBNull($reader.GetOrdinal("state_province")))
    {
        $stateProvince = $reader.GetString($reader.
GetOrdinal("state_province"))
    }

    "City is '{0}', State/Province is '{1}'" -f $city,
$stateProvince
}
```

3. After iterating over the data, we close the DataReader before closing the connection:

```
$reader.Close()
```

Reading in this manner is more tedious than filling a `DataTable` but the memory savings may be worth it in cases where you are working with large amounts of data.

Filtering and exporting data (Simple)

In this recipe we will look at filtering and sorting data and outputting data to different file formats.

Getting ready

Complete all previous recipes including *Retrieving data (Simple)*.

How to do it...

1. Since we want to output files to the script's directory, define this function:

```
function Get-ScriptDirectory
{
    if (Test-Path variable:\hostinvocation)
    {
        $FullPath=$hostinvocation.MyCommand.Path
    }
    else
    {
```

```
        $FullPath=(get-variable myinvocation -scope script).value.
Mycommand.Definition
    }
    if (Test-Path $FullPath)
    {
        return (Split-Path $FullPath)
    }
    else
    {
        $FullPath=(Get-Location).path
        Write-Warning ("Get-ScriptDirectory: Powershell Host <" +
$Host.name `
            + "> may not be compatible with this function, the
current directory <" `
            + $FullPath + "> will be used.")
        return $FullPath
    }
}
```

2. Establish a connection as in past recipes:

```
$conn = Connect-Oracle (Get-ConnectionString)
Define a SQL select statement. Here we get employee with manager
data:
$sql = @"
SELECT e.employee_id,
       e.first_name || ' ' || e.last_name employee,
       j.job_title, e.hire_date,
       m.first_name || ' ' || m.last_name manager
FROM   employees e
       JOIN jobs j
         ON e.job_id = j.job_id
       JOIN employees m
         ON e.manager_id = m.employee_id
"@
```

3. Populate a `DataTable`, set its table name, and close the connection:

```
$dt = Get-DataTable $conn $sql
$dt.TableName = "Employee"
$conn.Close()
```

4. Add the script's directory to the location stack and remove any `*.HR.*` output files there. This is a clean-up step for files we will create shortly.

```
Push-Location (Get-ScriptDirectory)
Remove-Item *.HR.* -force
```

5. Output the raw data from the `DataTable` to text files formatted as a list and a table, and to XML and CSV formats:

```
$dt | Format-List | Out-File .\EmployeesList.HR.txt
$dt | Format-Table -auto | Out-File .\EmployeesTable.HR.txt
$dt.WriteXml(".\Employees.xml")
$dt | Export-CSV .\Employees.csv
```

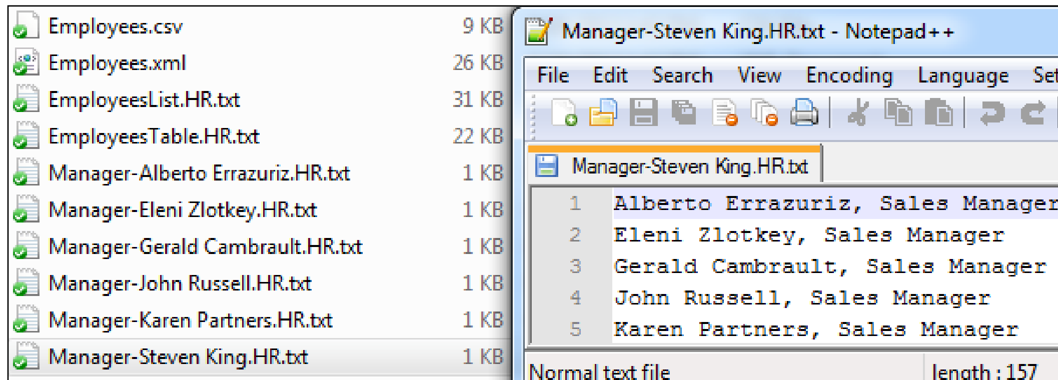
6. Create custom output files by piping the `DataTable` to `Where-Object` to filter it down, `Sort-Object` to order it, and `ForEach-Object` to iterate and process it, appending employee data to text files:

```
$dt | Where-Object {$_.job_title -like '*Sales*' -and $_.hire_date
-ge [datetime]"02/01/2004"} `
    | Sort-Object {$_.manager, $_.employee} `
    | ForEach-Object { `
      Add-Content (".\Manager-{0}.HR.txt" -f $_.manager) ("{0},
{1}" -f $_.employee, $_.job_title)
    }
```

7. Invoke the current directory to launch Windows Explorer to see the output files created and remove the script's directory from the location stack:

```
Invoke-Item .
Pop-Location
```

8. Run the script and inspect the files that were created:



How it works...

The `Get-ScriptDirectory` function handles some subtle differences between PowerShell hosts to return the directory where the script is running. This will be used later on to create output files. Alternatively we could hardcode the path directly or change the directory to the desired location with `Set-Location`.

The SQL string is defined using PowerShell's **here-string** construct with the `@` symbol before and after the string. This makes it easy to define a formatted, multiline string.

After loading the `DataTable`, it is given a name with `$dt.TableName = "Employee"`; this is required later to create the XML file with `$dt.WriteXml` as it needs a table name to know what tag name to use for each XML record.

`Push-Location` (or `pushd` for short) is then used to add the script's directory to the location stack and change the current directory location to that path. This makes it easier for file I/O to come, as full filenames will not have to be specified. Next any files in this directory matching a `*.HR.*` pattern are removed with `Remove-Item`; this ensures that on any subsequent script runs we start with a clean slate and does not append duplicate data to existing files.

Next, the raw data from the `DataTable` is output using different formats. First the data is formatted as a list with `Format-List` and piped to `Out-File` to create a list-formatted text file. In a similar fashion, a table-formatted text file is produced using `Format-Table`. An XML file is written using `$dt.WriteXml(".\Employees.xml")`. This could also be done with `($dt | ConvertTo-Xml -NoTypeInformation).Save(".\Employees.xml")`, but that's more verbose and produces much larger output that isn't formatted as well. Lastly an Excel-readable CSV file is produced via a pipe to `Export-CSV`.

Finally the data in the `DataTable` is filtered down with `Where-Object` to those records that have a job title like `*Sales*` and a hire date greater than or equal to `02/01/2004`. These matches are then sorted with `Sort-Object` by the manager's name and then the employee's name. Next `ForEach-Object` is used to iterate over these results and perform processing. In this example the processing simply consisits of every manager building text files for his/her employees by appending the employee name and job title with `Add-Content`.

Finally the script uses `Invoke-Item .` to launch the current directory in Windows Explorer and `Pop-Location` (or `popd` for short), to remove the script's directory from the location stack, and change back to the original directory before `Push-Location (Get-ScriptDirectory)` was invoked.

There's more...

Another popular export option is HTML; let's take a look at that.

Exporting to HTML

One common HTML formatting option is using `ConvertTo-HTML` as follows:

```
$filename = join-path (Get-ScriptDirectory) Employees.html
if (Test-Path($filename)) {Remove-Item $filename -force}

$head = @"
<style>
table { font-family: "Lucida Sans Unicode", "Lucida Grande", Sans-
```

```
Serif;
    font-size: 12px; background: #fff; margin: 45px; border-collapse: 
collapse;
   text-align: left; }
th { font-size: 14px; font-weight: normal; color: #039; padding: 10px 
8px;
   border-bottom: 2px solid #6678b1; }
td { border-bottom: 1px solid #ccc; color: #669; padding: 6px 8px; }
tbody tr:hover td { color: #009; }
</style>
"@

$dt | ConvertTo-HTML -head $head `
    -Property employee_id, employee, job_title, hire_date, manager `
    | Out-File $filename
Invoke-Item $filename
```

This code builds an HTML head section into `$head` to provide some nice styling of the table and specifies that with the `-head` parameter of `ConvertTo-HTML`. It also specifies what property names to include with `-Property` to prevent other properties of the `DataTable` from being included (such as `RowState`, `ItemArray`, and so on).

Adding records (Simple)

In this recipe we will look at adding new records to Oracle tables with PowerShell.

Getting ready

To run the sample against Oracle's sample HR schema, you may need to first disable the `SECURE_EMPLOYEES` trigger. See `README.txt` in this recipe's sample code for more details.

How to do it...

1. Create the following `Add-Oracle` function:

```
function Add-Oracle
{
Param(
    [Parameter(Mandatory=$true)][Oracle.DataAccess.Client.
OracleConnection]$conn,
    [Parameter(Mandatory=$true)][string]$sql,
    [Parameter(Mandatory=$false)][System.Collections.Hashtable]
$paramValues,
    [Parameter(Mandatory=$false)][string]$idColumn
```

```
)
    $cmd = New-Object Oracle.DataAccess.Client.
OracleCommand($sql,$conn)
    $cmd.BindByName = $true
    $idParam = $null

    if ($idColumn)
    {
        $cmd.CommandText = "{0} RETURNING {1} INTO :{2} " -f $cmd.
CommandText, $idColumn, $idColumn
        $idParam = New-Object Oracle.DataAccess.Client.
OracleParameter
        $idParam.Direction = [System.Data.
ParameterDirection]::Output
        $idParam.DbType = [System.Data.DbType]::Int32
        $idParam.Value = [DBNull]::Value
        $idParam.SourceColumn = $idColumn
        $idParam.ParameterName = $idColumn
        $cmd.Parameters.Add($idParam) | Out-Null
    }

    if ($paramValues)
    {
        foreach ($p in $paramValues.GetEnumerator())
        {
            $oraParam = New-Object Oracle.DataAccess.Client.
OracleParameter
            $oraParam.ParameterName = $p.Key
            $oraParam.Value = $p.Value
            $cmd.Parameters.Add($oraParam) | Out-Null
        }
    }

    $result = $cmd.ExecuteNonQuery()

    if ($idParam)
    {
        if ($idParam.Value -ne [DBNull]::Value) { $idParam.Value }
else { $null }
        $idParam.Dispose()
    }

    $cmd.Dispose()
}
```

2. Create an `Add-Employee` function to set up the `Add-Oracle` call:

```
function Add-Employee(
     [Parameter(Mandatory=$true)][string]$firstName,
     [Parameter(Mandatory=$true)][string]$lastName,
     [Parameter(Mandatory=$true)][string]$email,
     [Parameter(Mandatory=$true)][DateTime]$hireDate,
     [Parameter(Mandatory=$true)][string]$jobId,
     [Parameter(Mandatory=$false)][string]$phoneNumber
)
{
    $params = @{ 'FIRST_NAME'=$firstName; 'LAST_NAME'=$lastName;
'EMAIL'=$email; `
        'HIRE_DATE'=$hireDate; 'JOB_ID'=$jobId; 'PHONE_
NUMBER'=$phoneNumber }

    $sql = @"
INSERT INTO EMPLOYEES (
    EMPLOYEE_ID,
    FIRST_NAME,
    LAST_NAME,
    EMAIL,
    HIRE_DATE,
    JOB_ID,
    PHONE_NUMBER)
VALUES (
    EMPLOYEES_SEQ.NEXTVAL,
    :FIRST_NAME,
    :LAST_NAME,
    :EMAIL,
    :HIRE_DATE,
    :JOB_ID,
    :PHONE_NUMBER
)
"@
    $employeeId = Add-Oracle $script:conn $sql $params EMPLOYEE_ID
    $employeeId
}
```

3. Call the `Add*` functions to insert some records:

```
$conn = Connect-Oracle (Get-ConnectionString)

Add-Oracle $conn "INSERT INTO JOBS (JOB_ID,JOB_TITLE) VALUES
('CFO', 'Chief Financial Officer')"
```

```
$employeeId = Add-Employee -firstName "Joe" -lastName "De Mase"
-email "joe@company.com" `
    -hireDate ([DateTime]::Now.Date) -jobId "CFO"
if ($employeeId) { "Added employee; new id is $employeeId" }

Add-Oracle $conn `
    "INSERT INTO COUNTRIES (COUNTRY_ID,COUNTRY_NAME,REGION_ID)
VALUES (:COUNTRY_ID, :COUNTRY_NAME, :REGION_ID)" `
    @{ 'COUNTRY_ID'='BG'; 'COUNTRY_NAME'='Bulgaria'; 'REGION_
ID'=1; }

$conn.Close()
$conn.Dispose()
```

How it works...

The `Add-Oracle` function expects a connection and a SQL `INSERT` statement at a minimum. Optionally an associative array of name/**value pairs** (a dictionary that's a `Hashtable`) can be specified in `$paramValues` to set any **bind variables** that may be in the SQL for the values to be inserted. Bind variables help in preventing SQL injection attacks and avoid parsing. If `$paramValues` is specified, it is enumerated to set up the `Parameters` collection of the `OracleCommand` used for the insert.

Note that only the parameter names and values are set and other properties such as `DbType` will be automatically resolved.

For situations where you are inserting into a table that has a numeric primary key column and you want the record id back after the insert, specify the name of the primary key column with the `$idColumn` parameter. If set, the function will create an output parameter and modify the command text to return the id column value into that parameter. After the SQL is executed with `$cmd.ExecuteNonQuery()`, the parameter value is output for the function return value; any `DBNull.Value` (that is, error situation) is converted to `$null` to ease any conditional checking of the value.

`Add-Oracle` function is more generic in nature to handle the common insert cases; in some scenarios you may need more fine-grained control over the Oracle parameter collection than this function provides.

The `Add-Employee` function simply makes inserting an employee record easier and reusable via creating the `$params` dictionary and SQL with matching bind variables, passing that along to `Add-Oracle`, and returning the employee id of the new record.

`Add-Employee` uses the `OracleConnection` object in the script level variable `$conn` as opposed to passing in the connection as a parameter as is done in `Add-Oracle`. Here the idea is that more generic functions rely just on parameters and the more specific ones may use script level variables. The `$script:conn` syntax is used to clarify that this variable is in script scope.

The calls to the `Add` functions exercise different scenarios of inserting records, with and without bind variables and record ID retrieval. The insert into the `COUNTRIES` table isn't related to adding an `EMPLOYEES` record in this example but simply provides another example and test case.

There's more...

One thing this script adds over scripts in previous recipes is disposal of the different ODP.NET objects such as `OracleParameter`, `OracleCommand`, and `OracleConnection`. While the `Dispose` calls are not technically needed, it is best practice for responsible use of resources.

While calling `Dispose()` on the `OracleConnection` object calls `Close()`, it is often best to include both calls just to be explicit. Also note that with PowerShell there is no native using statement block as there is in C# or VB.NET to automatically call `Dispose()`, regardless of error or success. A Try/Catch/Finally block could be used to achieve the same result.

In the following recipe we will look at inserting records in more of a bulk fashion.

Importing sets of records (Medium)

In this recipe we will look at adding records in batches from external sources. First we will look at importing country data using `OracleBulkCopy`, from an XML file shown in the following figure:

countries.xml

```
1  <?xml version="1.0" standalone="yes"?>
2  <countries>
3    <country code="AF" iso="4">Afghanistan</country>
4    <country code="AL" iso="8">Albania</country>
5    <country code="DZ" iso="12">Algeria</country>
6    <country code="AS" iso="16">American Samoa</country>
```

to the following COUNTRIES table:

COUNTRIES ×

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL

Actions...

	COLUMN_NAME	NULLABLE	COLUMN_ID	DATA_TYPE	COMMENTS
1	COUNTRY_ID	No	1	CHAR(2 BYTE)	Primary key of countries table.
2	COUNTRY_NAME	Yes	2	VARCHAR2(40 ...	Country name
3	REGION_ID	Yes	3	NUMBER	Region ID for the country. Foreign key to region_id column in the departments table.

Getting ready

See previous recipes for the Add-Oracle, Get-ScriptDirectory, and connection functions used in this recipe.

How to do it...

1. Load the countries XML file into a DataTable and add a REGION_ID column:

```
$regionId = 5
$ds = New-Object System.Data.DataSet
$srcFilename = (join-path (Get-ScriptDirectory) countries.xml)
[void]$ds.ReadXml($srcFilename)
$regionCol = New-Object System.Data.DataColumn("REGION_ID", [int])
$regionCol.DefaultValue = $regionId
$regionCol.ColumnMapping = [System.Data.MappingType]::Attribute
$countryDT = $ds.Tables["country"]
$countryDT.Columns.Add($regionCol)
```

2. Connect and insert a region record for the countries to be imported:

```
$conn = Connect-Oracle (Get-ConnectionString)
Add-Oracle $conn "INSERT INTO REGIONS (REGION_ID, REGION_NAME)
VALUES ($regionId, 'Test Region')"
```

3. Set up an OracleBulkCopy object:

```
$bulkCopy = New-Object Oracle.DataAccess.Client.
OracleBulkCopy($conn) `
    -property @{DestinationTableName = "COUNTRIES";
BulkCopyTimeout = 300; NotifyAfter = 50}
[void]$bulkCopy.ColumnMappings.Add("code", "country_id")
[void]$bulkCopy.ColumnMappings.Add("country_Text", "country_name")
[void]$bulkCopy.ColumnMappings.Add("region_id", "region_id")
```

4. Register an event to be notified as record batches are inserted:

```
Register-ObjectEvent -InputObject $bulkCopy -EventName
OracleRowsCopied -SourceIdentifier BatchInserted `
    -MessageData $countryDT.Rows.Count -Action {
        $rowsCopied = $($event.sourceEventArgs.RowsCopied)
        $msg = "Inserted $rowsCopied records so far"
        $percentComplete = ($rowsCopied / $($event.MessageData) *
100)
        Write-Host $msg
        Write-Progress -Activity "Batch Insert" -Status $msg
-PercentComplete $percentComplete
    } | Out-Null
```

5. Call `WriteToServer` to start the import and time the duration:

```
$sw = [System.Diagnostics.StopWatch]::StartNew()
$bulkCopy.WriteToServer($countryDT)
$sw.Stop()
("Inserted {0} records total in {1:#0.000} seconds" -f $countryDT.
Rows.Count, $sw.Elapsed.TotalSeconds)
```

6. Perform teardown:

```
Unregister-Event -SourceIdentifier BatchInserted
$bulkCopy.Dispose()
$conn.Close()
$conn.Dispose()
```

7. Run the script. The sample output is as follows:

```
Inserted 50 records so far
Inserted 100 records so far
Inserted 150 records so far
Inserted 200 records so far
Inserted 210 records total in 0.052 seconds
```

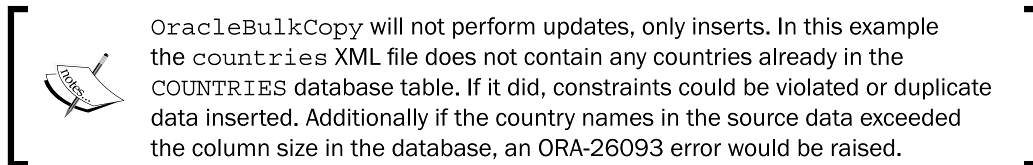
How it works...

First a `DataSet` is created and loaded using the `ReadXml` method and the `countries.xml` file in the current directory. This XML file is a slimmed down version from `http://madskristensen.net/post/XML-country-list.aspx` and was not created with a `DataSet` but the `DataSet` infers the schema from the data. A `REGION_ID` column is added since the source data doesn't contain this value but we want it in the `COUNTRIES` table. A default value of `5` ensures each row in the table gets this value.

After connecting and inserting the test region using `Add-Oracle` from the previous recipe, an `OracleBulkCopy` object is created and passed the connection. `OracleBulkCopy` is much more efficient at bulk loading data from a data source into an Oracle table, than iterating through data and inserting records one at a time. The `DestinationTableName` is set to the `COUNTRIES` table, a `BulkCopyTimeout` of 5 minutes is set (`300 seconds`), and we ask to be notified after every 50 records are processed with `NotifyAfter`. Next, the column mappings are set with `ColumnMappings.Add` calls, where the first argument is the source column name in the `DataTable` and the second is the destination column name in the Oracle table. The column mappings are required when there are any differences between column names, ordinal positions, or column counts between the source and destination. Note that `REGION_ID` has to be mapped even though the name matches and `[void]` prevents outputting the mapping object created.

Next, an event listener is registered for the `OracleRowsCopied` event of the `OracleBulkCopy` object in order to output progress information during the bulk import. In this sample there are only a couple hundred rows in the source XML file and they are processed so quickly that the progress bar is not really visible, short of setting a breakpoint. You will often be inserting considerably more records and the progress information may be more useful. The code in the `-Action` block defines the event logic of referencing the number of rows copied so far, the total row count, and the completion percentage, and writing that to the host output and progress. Note that `$countryDT.Rows.Count` is passed as the `-MessageData` parameter, as variables outside the `Action` block cannot be reliably accessed. This may work from the PowerShell ISE but not from the PowerShell console.

Finally the `WriteToServer` method invokes the bulk insert operation. We pass it the countries' `DataTable` but it can take different data sources including an array of `DataRow`, a `DataReader` or an `OracleRefCursor` type. A `StopWatch` object is used to calculate the duration and afterwards we clean up by unregistering the event, closing the connection, and disposing objects.



`OracleBulkCopy` will not perform updates, only inserts. In this example the `countries` XML file does not contain any countries already in the `COUNTRIES` database table. If it did, constraints could be violated or duplicate data inserted. Additionally if the country names in the source data exceeded the column size in the database, an ORA-26093 error would be raised.

There's more...

In some cases with a limited amount of records to insert, iterating over data in a file and inserting records one at a time may be acceptable performance wise for the flexibility of custom processing.

For very large data sets you may need to look at automating SQL *Loader from PowerShell or using Oracle's External Tables feature.

Another popular technique is using array binding, which we will look at next. You may need to experiment with different techniques and compare performance and trade-offs.

Using array binding for bulk inserts

Another common option for efficient bulk inserts is using array binding to insert records in one batch operation based on arrays of values:

```
$regions = Import-CSV (join-path (Get-ScriptDirectory) Regions.csv)
$regionIds = New-Object int[] $regions.Count
$regionNames = New-Object string[] $regions.Count
$index = 0

foreach ($r in $regions) {
    $regionIds[$index] = $r.RegionId; $regionNames[$index++] =
$r.RegionName
}

$cmd = New-Object Oracle.DataAccess.Client.OracleCommand( `
    "insert into regions values (:region_id, :region_name)",$conn)
$idParam = $cmd.Parameters.Add(":region_id", [Oracle.DataAccess.
Client.OracleDbType]::Int32)
$nameParam = $cmd.Parameters.Add(":region_name", [Oracle.DataAccess.
Client.OracleDbType]::Varchar2)
$idParam.Value = $regionIds; $nameParam.Value = $regionNames
$cmd.ArrayBindCount = $regions.Count
$trans = $conn.BeginTransaction()
$cmd.ExecuteNonQuery()
$trans.Commit()
```

The sample code for this recipe includes a simple `Regions.csv` file with region ID and region name. This file is read with `Import-CSV` and empty `int` and `string` arrays are created with a size matching the record count. Next, the arrays are filled by looping over the loaded CSV data. Bind variable parameters are used as before in the recipe, *Adding Records* but this time the values of the parameters are set to the arrays instead of single values. The other key here is setting the `ArrayBindCount` to the number of records. Additionally, a transaction is created to ensure all records are inserted or rolled back.

Updating and deleting records (Simple)

In this recipe we'll look at invoking SQL commands to make changes to Oracle, mostly with `UPDATE` and `DELETE` statements but we will also look at related topics such as modifying an Oracle schema.

How to do it...

1. Define the function `Invoke-Oracle`:

```
function Invoke-Oracle
{
Param(
    [Parameter(Mandatory=$true)][Oracle.DataAccess.Client.
OracleConnection]$conn,
    [Parameter(Mandatory=$true)][string]$sql,
    [Parameter(Mandatory=$false)][System.Collections.
Hashtable]$paramValues,
    [Parameter(Mandatory=$false)][switch]$passThru
)
    $cmd = New-Object Oracle.DataAccess.Client.
OracleCommand($sql,$conn)
    $cmd.BindByName = $true

    if ($paramValues)
    {
        foreach ($p in $paramValues.GetEnumerator())
        {
            $oraParam = New-Object Oracle.DataAccess.Client.
OracleParameter
            $oraParam.ParameterName = $p.Key
            $oraParam.Value = $p.Value
            $cmd.Parameters.Add($oraParam) | Out-Null
        }
    }

    $result = $cmd.ExecuteNonQuery()
    $cmd.Dispose()

    if ($passThru) { $result }
}
```

2. Load ODP.NET and connect to Oracle as in previous recipes.
3. Make some calls to `Add-Oracle` with various `INSERT`, `UPDATE`, and `DELETE` SQL statements:

```
$affected = Invoke-Oracle $conn -PassThru `
    "UPDATE HR.JOBS SET MIN_SALARY = MIN_SALARY + 1000 WHERE
UPPER(JOB_TITLE) LIKE '%MANAGER%'"
"Updated {0} job record(s)" -f $affected

$cities = "Jacksonville", "Orlando", "Tampa", "Miami", "Del Ray
```

```
Beach"
$inserted = 0
foreach ($city in $cities) {
    $inserted += Invoke-Oracle $conn "INSERT INTO HR.LOCATIONS
(LOCATION_ID, CITY, STATE_PROVINCE) `
        VALUES (HR.LOCATIONS_SEQ.NEXTVAL, :CITY, :STATE_PROVINCE)"
`
        @{ 'CITY'=$city; 'STATE_PROVINCE'='Florida' } -PassThru
}

"Inserted $inserted location records"

Invoke-Oracle $conn "UPDATE HR.LOCATIONS SET COUNTRY_ID = `
    (SELECT COUNTRY_ID FROM HR.COUNTRIES WHERE COUNTRY_NAME =
'United States of America')"

"Deleted {0} location records" -f (Invoke-Oracle $conn `
    "DELETE FROM LOCATIONS WHERE STATE_PROVINCE = 'Florida'"
-PassThru)
```

4. Run the script. Output is as follows:

```
Updated 6 job record(s)
Inserted 5 location records
Deleted 5 location records
```

How it works...

The `Invoke-Oracle` function is very similar to the `Add-Oracle` function from the two previous recipes, except it does not include any code to get back record ids of any records inserted. Another difference is `Invoke-Oracle` includes a `-PassThru` switch parameter that can be specified to return the number of records affected by the statement; by default the function does not emit output.

There's more...

`Invoke-Oracle` can also be used for multi-statement **PL/SQL** blocks:

```
$sql = @"
BEGIN
INSERT INTO HR.REGIONS (REGION_ID, REGION_NAME) VALUES (5, 'Africa');
INSERT INTO HR.COUNTRIES(COUNTRY_ID, COUNTRY_NAME, REGION_ID) VALUES
('BU','Burundi', 5);
END;
"@
Invoke-Oracle $conn $sql
```

In addition to inserts, updates, and deletes, `Invoke-Oracle` also works fine for simple **DDL** operations such as `CREATE TABLE` or **DCL** operations such as `GRANT` or `REVOKE`:

```
Invoke-Oracle $conn "ALTER TABLE HR.DEPARTMENTS ADD NOTE VARCHAR2(100)
NULL"
Invoke-Oracle $conn "ALTER TABLE HR.DEPARTMENTS DROP COLUMN NOTE"
```

Updating with a DataAdapter

Sometimes you may want to select data into a `DataTable` using an `OracleDataAdapter` as in the recipe *Retrieving data (Simple)*, make some changes in PowerShell, then update those back to the Oracle table without manually creating an UPDATE statement and command:

```
$sql = "SELECT EMPLOYEE_ID, MANAGER_ID, SALARY FROM HR.EMPLOYEES WHERE
DEPARTMENT_ID = 90"
$cmd = New-Object Oracle.DataAccess.Client.OracleCommand($sql,$conn)
$da = New-Object Oracle.DataAccess.Client.OracleDataAdapter($cmd)
$cmdBuilder = new-object Oracle.DataAccess.Client.OracleCommandBuilder
$da
$dt = New-Object System.Data.DataTable
[void]$da.Fill($dt)

foreach ($dr in $dt.Rows) {
    if ($dr.manager_id -eq 100) {$dr.salary += $dr.salary * .10}
}

"Updated {0} records" -f $da.Update($dt)
```

An `OracleCommandBuilder` is created, and it is passed the data adapter that it will use to automatically create an `UpdateCommand` to be used when the `Update` method is called on the `OracleDataAdapter` to persist the changes. The command builder will use metadata from the `DataTable` and `SelectCommand` (`$cmd`) to construct the update command. It will listen for changes to the data table to track what changed.

This method only works for single-table updates and you may find that it generates SQL that you might not expect. You can inspect `$cmdBuilder.GetUpdateCommand()` to see the update statement it generates.

Executing database procedures (Medium)

In this recipe we will explore executing a database package procedure that returns multiple **Ref Cursors** and loading that data into a `DataSet`.

Getting ready

To run the code, first execute `EMPLOYEE_PACKAGE.pls` in a DBMS tool. This sample package SQL file is included with the code for this recipe.

How to do it...

1. Create a function to return a connection object:

```
function New-Connection
{
    $dataSource = "(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)
(HOST=localhost)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=xe)))"
    $connectionString = ("Data Source={0};User
Id=HR;Password=pass;Connection Timeout=10" -f $dataSource)
    New-Object Oracle.DataAccess.Client.OracleConnection($connecti
onString)
}
Create a function to set up a command for executing a database
procedure:
function New-ProcCommand ($procedure, $parameters)
{
    $cmd = New-Object Oracle.DataAccess.Client.
OracleCommand($procedure, (New-Connection))
    $cmd.CommandType = [System.Data.CommandType]::StoredProcedure
    $parameters | foreach {$cmd.Parameters.Add($_) | Out-Null}
    $cmd
}
```

2. Create helper functions for creating `OracleParameter` objects:

```
function New-Param ($name, $type, $value,
    $size = 0, $direction = [System.Data.
ParameterDirection]::Input)
{
    New-Object Oracle.DataAccess.Client.OracleParameter($name,
$type, $size) `
        -property @{Direction = $direction; Value = $value}
}

function New-CursorParam ($name)
{
    New-Param -name $name -type ([Oracle.DataAccess.Client.
OracleDbType]::RefCursor) `
        -direction ([System.Data.ParameterDirection]::Output)
}
```

3. Add a function to return a data reader for reading procedure Ref Cursors:

```
function Get-ProcReader ($procedure, $parameters)
{
    $cmd = New-ProcCommand $procedure $parameters
    if ($cmd.Connnection.State -ne [System.Data.
ConnectionState]::Open)
    {
        $cmd.Connection.Open()
    }
    ,$cmd.ExecuteReader()
}
```

4. Define a function to invoke a `LOAD_EMPLOYEE` procedure from `EMPLOYEE_PACKAGE` and load the results into a `DataSet`.

```
function Get-EmployeeDataSet ($employeeId)
{
    $procedure = "HR.EMPLOYEE_PACKAGE.LOAD_EMPLOYEE"
    $params = @(
        (New-Param -name "I_EMPLOYEE_ID" -type ([Oracle.
DataAccess.Client.OracleDbType]::Int32) -value $employeeId)
        (New-CursorParam -name "O_EMPLOYEES")
        (New-CursorParam -name "O_LOCATIONS")
    )
    $ds = New-Object System.Data.DataSet("EmployeesDataSet")
    $empDT = $ds.Tables.Add("EMPLOYEES")
    $locDT = $ds.Tables.Add("LOCATIONS")
    $reader = Get-ProcReader $procedure $params
    $empDT.Load($reader)
    $locDT.Load($reader)
    $reader.Close(); $reader.Dispose()
    $ds
}
```

5. Add code to invoke the function and output the rows returned:

```
$ds = Get-EmployeeDataSet -employeeId 203
$ds.Tables["EMPLOYEES"]
$ds.Tables["LOCATIONS"]
$ds.Dispose()
```

6. Run the script. Sample output follows.

```
EMPLOYEE_ID     : 203
FIRST_NAME      : Susan
LAST_NAME       : Mavris
EMAIL           : SMAVRIS
PHONE_NUMBER    : 515.123.7777
```

```
HIRE_DATE       : 6/7/2002 12:00:00 AM
JOB_ID          : HR_REP
JOB_TITLE       : Human Resources Representative
SALARY          : 6500
COMMISSION_PCT  :
MANAGER_ID      : 101
MANAGER_NAME    : Neena Kochhar
DEPARTMENT_ID   : 40
DEPARTMENT_NAME : Human Resources

LOCATION_ID    : 2400
STREET_ADDRESS : 8204 Arthur St
POSTAL_CODE    :
CITY           : London
STATE_PROVINCE :
COUNTRY_ID     : US
```

How it works...

The `New-Connection` function is slightly different than past connecting code in that it defines the connection string in the same method and doesn't automatically open the connection it creates.

`New-ProcCommand` expects a procedure name and an array of `OracleParameter` objects and it creates the command, sets the command text to the procedure name, creates and associates the `OracleConnection` object, set the command type, adds each parameter to the `Parameters` collection, and returns the command.

The `New-Param` helper method saves some typing when creating `OracleParameter` methods later via creating the object and defaulting some of the property values such as size and duration. Likewise the `New-CursorParameter` goes a step further with setting the parameter type and direction as well.

`Get-ProcReader` makes the call to create the command object. As it doesn't control the creation of the command or connection, it also checks the state of connection and opens it if it isn't open, though the connection would always be closed with the current definition of the functions. Finally it invokes the procedure with `ExecuteReader` and prevents unrolling the returned results with the leading comma, as a data reader return value is expected.

`Get-EmployeeDataset` sets the procedure name and the array of `OracleParameter` objects that `Get-ProcReader` expects. It also sets up a new `DataSet` via adding `DataTable`s where the Ref Cursor results from the procedure will be stored. The `DataSet` and `DataTable`s are given friendly names to make working with the results easier. Further `DataSet` set up may often be desirable, this is a minimum. The data tables are then loaded with the reader, in the order the cursors are opened in the database procedure. Finally the clean up is done, the `DataSet` is returned, the function is invoked, and results are output.

There's more...

In this example we executed a package procedure that returned Ref Cursors but we could use the same or similar code to invoke functions or procedures inside or outside a package that perform other tasks such as inserting data or returning simple values. The sample code for this recipe includes other script files with these examples. Let's look at invoking a function.

Invoking a function

ODP.NET uses positional parameter binding by default. When calling a function we want to set `$cmd.BindByName = $true` for the command (in `New-ProcCommand`). Afterwards the following code is used to invoke the `GET_EMPLOYEE_MANAGER` function in the `EMPLOYEE_PACKAGE` to return the manager name of a given employee ID:

```
function Invoke-Proc ($procedure, $parameters)
{
    $cmd = New-ProcCommand $procedure $parameters
    if ($cmd.Connnection.State -ne [System.Data.
ConnectionState]::Open) {$cmd.Connection.Open()}
    $cmd.ExecuteNonQuery() | Out-Null
    $cmd.Connection.Close(); $cmd.Connection.Dispose(); $cmd.Dispose()
}

function Get-EmployeeManager ($employeeId)
{
    $params = @(
        (New-Param -name "i_employee_id" -type ([Oracle.DataAccess.
Client.OracleDbType]::Int32) -value $employeeId)
        (New-Param -name "RETURN_VALUE" -type ([Oracle.DataAccess.
Client.OracleDbType]::Varchar2) `
            -direction ([System.Data.ParameterDirection]::ReturnValue
) -size 46)
    )
    Invoke-Proc "HR.EMPLOYEE_PACKAGE.get_employee_manager" $params
    $params[1].Value
}

"Manager of employee id 200 is {0}" -f (Get-EmployeeManager 200)
```

This produces the following output:

```
Manager of employee id 200 is Neena Kochhar
```

Alternatively, without using `BindByName`, `CommandType.Text` could be used instead of `CommandType.StoredProcedure` and the function could be called inside a BEGIN/END PL/SQL block.

Organizing and invoking scripts (Medium)

In this recipe we will look at organizing script code into multiple script files for better reusability and maintainability.

Getting ready

Complete previous recipes first. Use the sample code files included with this recipe or create them manually in the same directory, copying code from the samples as needed.

How to do it...

1. Create `Oracle.DataAccess.ps1` for common ODP.NET code. Some examples from this file included with this recipe's sample code are as follows:

```
function Get-ODPAssemblyName
{
param ( [Parameter(Mandatory=$true)][validateset(2,4)]
        [int] $version)
    $a = @{}
    $a["2"] = "Oracle.DataAccess, Version=2.112.3.0,
Culture=neutral, PublicKeyToken=89b483f429c47342"
    $a["4"] = "Oracle.DataAccess, Version=4.112.3.0,
Culture=neutral, PublicKeyToken=89b483f429c47342"
    $a[$version.ToString()]
}

function Get-ODPAssembly
{
    [appdomain]::currentdomain.getassemblies() `
        | ? {$_.FullName -like 'Oracle.DataAccess,*'} `
        | select -first 1
}

function Load-ODP
{
param (
    [Parameter(Position=0, Mandatory=$true)]
    [validateset(2,4)]
    [int] $version,
    [Parameter(Position=1)]
    [switch] $passThru
)
    $asm = [System.Reflection.Assembly]::Load((Get-ODPAssemblyName
```

```
$version))
    if ($passThru) { $asm }
}

function Select-ODPTypes
{
    (Get-ODPAssembly).GetTypes() | ? {$_.IsPublic} | sort {$_.
FullName }
}
```

2. Create a file `Utilities.ps1` for common code often used in conjunction with ODP.NET. A partial sample from the sample code is as follows:

```
function Get-ConfigConnectString
{
Param( [Parameter(Mandatory=$true)] [string]$filename,
       [Parameter(Mandatory=$true)] [string]$name )
    $config = [xml](cat $filename)
    $item = $config.configuration.connectionStrings.add | where
{$_.name -eq $name}
    if (!$item) { throw "Failed to find a connection string with
name '{0}'" -f $name}
    $item.connectionString
}
```

3. Create an `App.config` file with a connection string in the same directory as the scripts; use the sample included with the code for this recipe or see the recipe *Connecting and disconnecting (Simple)*.
4. Create a file `IncludeTest.ps1` to test the included functions. In this file first add the following code to include the common function libraries:

```
. .\Utilities.ps1
. .\Oracle.DataAccess.ps1
```

5. Add some code to list out the functions that were included that have ODP in the function name:

```
dir function: | ? {$_.Name -like '*ODP*'} | ft
```

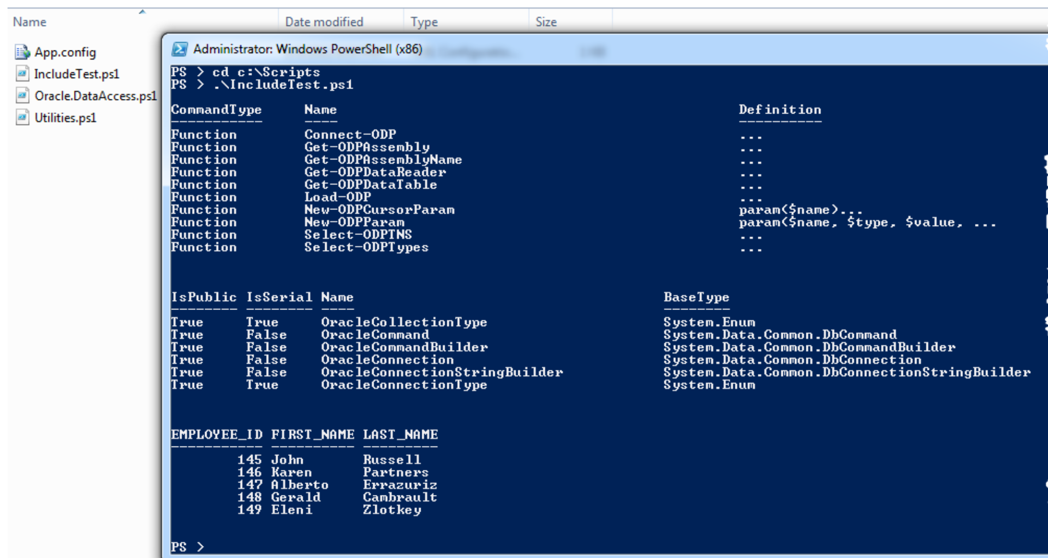
6. Load ODP.NET 2x and list types that begin with "`OracleCo`":

```
Load-ODP -version 2
Select-ODPTypes | ? {$_.Name -like 'OracleCo*'} | ft
```

7. Connect and retrieve some data:

```
$conn = Connect-ODP (Get-ConfigConnectString .\App.config
AppConnect)
$dt = Get-ODPDataTable $conn "select employee_id, first_name,
last_name from employees where job_id = 'SA_MAN'"
$dt | ft -auto
$conn.Close(); $conn.Dispose()
```

8. Open a PowerShell command-line host, change the directory to the location of the scripts, and run `IncludeTest.ps1` with `.\IncludeTest.ps1`:



How it works...

First we moved common ODP.NET functions to `Oracle.DataAccess.ps1` so they can be reused across multiple scripts and we can start building a useful library of Oracle functions. Next we did the same for `Utilities.ps1` for any common functionality that is often used with Oracle script code but isn't Oracle-specific. This allows each script using these script libraries to focus on specific tasks and not setup, helpers, and infrastructure.

Next , we included the contents of these script files into `IncludeTest.ps1` with the following:

```
. .\Utilities.ps1
. .\Oracle.DataAccess.ps1
```

This is referred to as **dot sourcing** the scripts, which will pull all the functions and any variables into the same runtime scope of `IncludeTest.ps1`. Note there is no leading space before the period and then a space between the period and the filename. In this case we are expecting these included scripts to be in the current directory (the `.\`) – not necessarily the directory of the calling script (`IncludeTest.ps1`).

Usually the included files will be somewhere else and you may need to provide a full path to the files being included, such as:

```
"C:\Scripts\Oracle.DataAccess.ps1"
```

Alternatively, you could modify your environment path either more permanently in Windows or temporarily with PowerShell, as follows:

```
$env:Path += ";C:\Scripts"
. Oracle.DataAccess.ps1
```

Finally you might temporarily change directory to a known location using `pushd` and `popd` before and after including the script files, or use other techniques such as relative pathing from the script's directory using the `Get-ScriptDirectory` function from the recipe *Filtering and exporting data (Simple)*.

After dot sourcing the script files, `IncludeTest.ps1` listed the functions in scope using `dir function:` and the results showed the various ODP functions that we defined and were loaded with the include. If we ran `dir function:` after the script finished executing, the functions would no longer be available. However, if we ran `IncludeTest.ps1` using dot sourcing as well (`. .\IncludeTest.ps1`), the functions would still be in scope for use even after the script finished executing.

The rest of the script remains similar to what we've looked at in previous recipes. One difference is the use of more shorthand syntax such as `?` in place of `Where-Object`, `sort` instead of `Sort-Object`, and so on. More importantly we've refactored some existing common functions a bit and added new ones, such as `Load-ODP`, which uses a `ValidateSet` attribute as `[ValidateSet(2,4)]` to ensure that the version parameter is either `2` or `4`, to load the corresponding .NET framework version of Oracle.DataAccess.

There's more...

Care should be taken when dot sourcing files as problems can arise. For one, any script level variables of the included scripts are globally available for `get` and `set` access in the script they are included into. A careful naming convention should be chosen to ensure it's obvious what functions do, and that there aren't function name collisions between scripts. Finally, these functions should generally be generic in nature, with parameters to help prevent making too many assumptions about the script code using the functions.

The PowerShell profile

If you inspect `$profile` in PowerShell you'll get back a profile filename where common script will be loaded automatically whenever you start a PowerShell session:

```
PS > $profile
C:\Users\Geoff\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.
ps1
```

You could dot source these common script libraries inside the profile script so they are included automatically when starting PowerShell. You might even invoke some functions afterwards such as `Load-ODP`. These files do not exist by default and there is a different profile for each host; for example the ISE uses a `Microsoft.PowerShellISE_profile.ps1` filename. You may want to create and dot source a `SharedProfile.ps1` file that defines the shared profile script for all hosts.

Script automation and error handling (Medium)

In this script we'll look at automating a PowerShell script with a scheduled task, logging, and error handling. We will query employee data where the commission percentage exceeds a given threshold and export that data to a CSV file for later analysis.

Getting ready

Complete the previous recipes first. Copy updated versions of `Oracle.DataAccess.ps1` and `Utilities.ps1` from the sample code files for this chapter that includes parameter support in `Get-ODPDataTable` and a new function `Get-CanTranscribe`. Likewise copy `App.config` and change the connection string as necessary.

How to do it...

1. Create a `Query.sql` file to define the SQL we want to execute:

```
select
  e.employee_id,
  e.first_name || ' ' || e.last_name employee,
  e.email,
  e.hire_date,
  e.job_id,
  e.salary,
  e.commission_pct,
  d.department_name department
from
```

```
  HR.employees e
join
  HR.departments d
  on e.department_id = d.department_id
where
  e.commission_pct  > :commission_threshold
```

2. Create `HighCommission.ps1` and add the following script to it:

```
param (
[Parameter(Mandatory=$true)][string]$outputPath,
[Parameter(Mandatory=$false)][decimal]$commissionThreshold = .25)

$errorActionPreference = "Continue"
```

3. Add a function to `HighCommission.ps1` to define set host line widths:

```
function Set-HostSettings
{
    if ($Host -and $Host.UI -and $Host.UI.RawUI) {
        $ui = (Get-Host).UI.RawUI
        $winSize = $ui.WindowSize; $buffSize = $ui.BufferSize
        $buffSize.Width = 120; $ui.BufferSize = $buffSize
        $winSize.Width = 120; $ui.WindowSize = $winSize
    }
}
```

4. Define an `Invoke-Script` function in `HighCommission.ps1`:

```
function Invoke-Script
{
    try {Set-HostSettings} catch {}
    "Processing started. Commission threshold is
$commissionThreshold, output path is $outputPath"

    "Checking existence of output path $outputPath"
    if (!(Test-Path $outputPath -PathType Container)) {
        throw "outputPath '$outputPath' doesn't exist"
    }

    "Importing Oracle.DataAccess.ps1"
    . (join-path $_scriptDir Oracle.DataAccess.ps1)

    $sqlFilename = (join-path $_scriptDir Query.sql)
    $configFile = (join-path $_scriptDir App.config)
    $outputFilename = (join-path $outputPath HighCommission.csv)
```

```
    if (Test-Path($outputFilename)) {
        "Removing old copy of $outputFilename"
        Remove-Item $outputFilename -Force
    }

    "Loading ODP.NET 2x"; $asm = Load-ODP -version 2 -passthru
    "Loaded {0}" -f $asm.FullName

    "Reading SQL from $sqlFilename"
    $sql = (Get-Content $sqlFilename -EV err) | Out-String
    if ($err) {
        Write-Warning "SQL read failed; bailing"; throw $err
    }
    $sql

    "Reading connection string from $configFile"
    $connectString = (Get-ConfigConnectString $configFile
AppConnect)
    "Connecting with $connectString"
    $conn = Connect-ODP $connectString

    try {
        "Running SQL using threshold of $commissionThreshold"
        $dt = Get-ODPDataTable $conn $sql @{ 'commission_
threshold' = $commissionThreshold; }

        "Retrieved {0} record(s):" -f $dt.Rows.Count
        $dt | ft -auto

        "Exporting data to $outputFilename"
        $dt | Export-CSV -NoTypeInformation $outputFilename
        $dt.Dispose()
    }
    catch [System.Exception] {
        throw "Failed to run SQL: " + $_.Exception.ToString()
    }
    finally {
        "Closing connection"
        $conn.Close(); $conn.Dispose()
    }
}
```

5. Add the following to `HighCommission.ps1` to set up the `Invoke-Script` call:

```
try {
    $script:_scriptDir = (Get-ScriptDirectory)
    . (join-path $_scriptDir Utilities.ps1)

    if (Get-CanTranscribe) {
        try { Stop-Transcript | out-null } catch { }
        Start-Transcript -path (join-path $_scriptDir
"HighCommissionLog.txt")
    }

    Invoke-Script
}
catch [System.Exception] {
    $error = ("Script failed with error: {0}{1}{1}Script: {2}
Line,Col: {3},{4}" `
        -f $_.Exception.ToString(), [Environment]::NewLine,
$_.InvocationInfo.ScriptName, `
        $_.InvocationInfo.ScriptLineNumber, $_.InvocationInfo.
OffsetInLine)
    Write-Error $error; Exit 100
}
finally {
    if (Get-CanTranscribe) { Stop-Transcript }
}
```

6. Open the **Task Scheduler** via *Win + R*, type `Taskschd.msc`, then press *Enter*.
7. Click **Create Basic Task...** from the **Actions** panel. Supply a task name and click **Next** until you get to the **Start a Program** step.
8. For the **Program/script** field, enter the appropriate path to PowerShell:

```
%SystemRoot%\syswow64\WindowsPowerShell\v1.0\powershell.exe
```

9. For **Add arguments**, use the following, changing the paths as needed:

```
-NonInteractive -Command "& c:\Scripts\HighCommission.ps1
-outputPath 'C:\Dropbox\Report Outbox\' -commissionThreshold
.3; exit $LASTEXITCODE"
```

10. Run the scheduled task, and then inspect the output files `HighCommissionLog.txt` and `HighCommission.csv`.

How it works...

This script defines script level parameters that control what directory the results will be output to and what commission percentage threshold must be exceeded for an employee to be included in the results. To be explicit, the script sets `$ErrorActionPreference` to `Continue` (the default). This behaviour results in non-terminating errors being output and script execution continuing; other valid values are `SilentlyContinue`, `Inquire`, and `Stop`. Likewise `$WarningPreference` can be set the same way for handling warnings. Preferences can be overridden when invoking **cmdlets** as follows: `Remove-Item $outputFilename -Force -ErrorAction SilentlyContinue`.

At the bottom of the script, all of the processing code (mostly `Invoke-Script`) is enclosed in a try/catch block so we can respond to any error and stop any transcription that was started regardless of success or failure. Transcription automatically sends all output of the script to a specified file without having to explicitly direct output for each command to a file. Since transcription is not supported in the ISE, the function `Get-CanTranscribe` (in `Utilities.ps1`) checks the host name and doesn't use transcription if running from the ISE (useful while debugging). In the catch block the built-in `$_.Exception` and `$_.InvocationInfo` variables are used to format the error message. The message is written to error output with `Write-Error` and a non-zero `Exit` statement is used so the scheduled task will report a failure when the script fails.

In the `Invoke-Script` function, `Set-HostSettings` is called to expand the host line length from 80 to 120 characters. This is done so the output doesn't wrap as often in the transcription log. Since this isn't critical, an empty catch block supresses any errors. Next, the function throws an error if a valid directory was not passed in with the `$outputPath` script parameter. The next error check comes with the following:

```
$sql = (Get-Content $sqlFilename -EV err) | Out-String
if ($err) {
        Write-Warning "SQL read failed; bailing"; throw $err
}
```

This uses the `–ErrorVariable` argument (`EV` for short) to send any error to the specified variable, the `$err` variable in this case. Note that `–EV err` is used and not `–EV $err`. This technique is another error handling approach that can be used on a cmdlet by cmdlet basis. Here we turn any non-critical error reading the SQL from a file into both a warning and a critical error.

Finally, we surround executing the SQL query (`Get-ODPDataTable`) with a try/catch block to ensure the connection gets closed regardless of the success or failure of the SQL query. The other important aspect of this script is output that's written every step of the way, indicating what is about to happen and the result of what was just executed. This is critical for troubleshooting since the script runs unattended in the background.

When setting up the scheduled task the PowerShell.exe arguments were set as follows:

```
-NonInteractive -Command "& c:\Scripts\HighCommission.ps1 -outputPath
'C:\Dropbox\Report Outbox\' -commissionThreshold .3; exit $LASTEXITCODE"
```

The `–NonInteractive` argument is set since we won't be running the script in a user-interactive way. The `–Command` argument is used instead of the `–File` argument since we need to pass parameters to the script. We first tell PowerShell to run the script `HighCommission.ps1` with *"& c:\Scripts\HighCommission.ps1*. The `outputPath` parameter for `HighCommision.ps1` is then specified, using single quotes since the directory has a space in the path. Next the default `.25` value for the script's `commisionThreshold` parameter is overridden with a `.3` value. Finally, a semicolon is used to separate a second statement of `exit $LASTEXITCODE`, which is needed to kick any error exit code up the chain, through PowerShell.exe's exit code, and onto the scheduled task result.

There's more...

You may notice the PowerShell command-line host popping up briefly when running the scheduled task. You may be able to avoid that by changing the user the scheduled task runs under to `NT AUTHORITY\SYSTEM`. The user should be carefully considered in terms of having enough permissions to perform the script actions but not more permissions than necessary (as may be the case with `SYSTEM`). Consider using a service account and use caution when using an account where the password will eventually expire.

When developing a script to be run as a scheduled task and when modifying it, it's best to run it within PowerShell first, instead of from the scheduled task. If there are syntax or certain other errors, you may not get any transcription output and any console that pops up may disappear before you have time to see the error. If the script works from the console but not from within the scheduled task, screen captures or screencasts can be used to snag any temporarily visible error message. From there additional isolation changes can be made and additional output writes added to help diagnose issues.

Another error tool is the built-in `$Error` variable. You can inspect `$Error[0]` to see the most recent error or larger indexes to see prior ones. Finally you can query the `$?` automatic variable; if it returns `$false`, the last command failed.

For scheduled tasks such as this one, sending an e-mail using `System.Net.Mail.SmtpClient` can be useful. In this example in this recipe, attaching `HighCommission.csv` on success and `HighCommission.log` on error would be a useful enhancement.

Creating reusable script modules (Advanced)

In this recipe we'll turn prior script libraries into more reusable script modules.

Getting ready

Complete the previous recipes first. See sample code files or prior recipes for additional functions such as `Get-DataTable` and `Set-CommandParamsFromArray`.

How to do it...

1. Evaluate PowerShell module paths with `$env:PSModulePath`:

   ```
   PS > $env:PSModulePath.Split(";")
   C:\Users\Geoff\Documents\WindowsPowerShell\Modules
   C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
   ```

2. In your `%USERPROFILE%\Documents\WindowsPowerShell\Modules\` folder, create the folder `Oracle.DataAccess` (create the `Modules` folder if it does not exist).
3. In the `Oracle.DataAccess` folder create the file `Oracle.DataAccess.psm1` (note the `psm1` extension, rather than `ps1`).
4. At the top of the file define a parameter for the version of ODP.NET to use, a variable to save a connection object, and a `Load` function:

   ```
   param ([Parameter(Mandatory=$true)][validateset(2,4)]
   [int] $OdpVersion)
   $SCRIPT:conn = $null

   function Load {
   param (
       [Parameter(Position=0, Mandatory=$true)]
       [validateset(2,4)] [int] $version,
       [Parameter(Position=1)] [switch] $passThru
   )
       $name = ("Oracle.DataAccess, Version={0}.112.3.0,
   Culture=neutral, PublicKeyToken=89b483f429c47342" -f $version)
       $asm = [System.Reflection.Assembly]::Load($name)
       if ($passThru) { $asm }
   }
   ```

5. Create a connection function with built-in Help documentation, and a connection method to connect via TNS:

```
<#
.SYNOPSIS
Connects to oracle via a connection string.

.DESCRIPTION
Creates a new Oracle Connection and opens it using the specified
connections string.
Created connection is stored and not returned unless -PassThru is
specified.

.PARAMETER ConnectionString
The full connection string of the connection to be created and
opened.

.PARAMETER PassThru
If -PassThru is supplied, the created connection will be returned
and not stored.

.EXAMPLE
Connect to oracle with a connection string and store the
connection for later use without outputting it.

Connect "Data Source=LOCALDEV;User Id=HR;Password=Pass"

.NOTES
If -PassThru isn't used, the connection will be available for
later operations such as Disconnect, without having to pass it.
#>
function Connect {
[CmdletBinding()]
Param(
[Parameter(Mandatory=$true)] [string]$ConnectionString,
[Parameter(Mandatory=$false)] [switch]$PassThru )
    $conn= New-Object Oracle.DataAccess.Client.OracleConnection($C
onnectionString)
    $conn.Open()
    if (!$PassThru) {
        $SCRIPT:conn = $conn
        Write-Verbose ("Connected with {0}" -f $conn.
ConnectionString)
    }
    else {
        $conn
```

```
    }
}

function Connect-TNS {
[CmdletBinding()]
Param(
[Parameter(Mandatory=$true)] [string]$TNS,
[Parameter(Mandatory=$true)] [string]$UserId,
[Parameter(Mandatory=$true)] [string]$Password,
[Parameter(Mandatory=$false)] [switch]$PassThru )
    $connectString = ("Data Source={0};User Id={1};Password={2};"
-f $TNS, $UserId, $Password)
    Connect $connectString -PassThru:$PassThru
}
```

6. Define functions to resolve a connection and to disconnect:

```
function Get-Connection ($conn) {
    if (!$conn) { $conn = $SCRIPT:conn }
    $conn
}

function Disconnect {
[CmdletBinding()]
Param(
    [Parameter(Mandatory=$false)]
    [Oracle.DataAccess.Client.OracleConnection]$conn)
    $conn = Get-Connection($conn)
    if (!$conn) {
        Write-Verbose "No connection is available to disconnect
from"; return
    }
    if ($conn -and $conn.State -eq [System.Data.
ConnectionState]::Closed) {
        Write-Verbose "Connection is already closed"; return
    }
    $conn.Close()
    Write-Verbose ("Closed connection to {0}" -f $conn.
ConnectionString)
    $conn.Dispose()
}
```

7. Add `-WhatIf` support to a function for executing non-query methods:

```
function Invoke {
[CmdletBinding(SupportsShouldProcess = $true)]
Param(
```

```
    [Parameter(Mandatory=$false)][Oracle.DataAccess.Client.
OracleConnection]$conn,
    [Parameter(Mandatory=$true)][string]$sql,
    [Parameter(Mandatory=$false)][Hashtable]$paramValues,
    [Parameter(Mandatory=$false)][switch]$passThru
)
    $conn = Get-Connection($conn)
    $cmd = New-Object Oracle.DataAccess.Client.
OracleCommand($sql,$conn)
    Set-CommandParamsFromArray $cmd $paramValues
    $trans = $conn.BeginTransaction()
    $result = $cmd.ExecuteNonQuery(); $cmd.Dispose()

    if ($psCmdlet.ShouldProcess($conn.DataSource)) {
        $trans.Commit()
    }
    else {
        $trans.Rollback(); "$result row(s) affected"
    }

    if ($passThru) { $result }
}
```

8. At the bottom of the script, use `Export-ModuleMember` to define the functions to be made public and automatically load ODP.NET:

```
Export-ModuleMember -Function Connect,Connect-TNS,Disconnect,Get-
DataTable,Invoke
Load -version $OdpVersion
```

9. Open a PowerShell command prompt and import the module:

```
PS > Import-Module Oracle.DataAccess -Prefix Oracle -ArgumentList
2
```

10. List the commands available in the module:

```
PS > Get-Command -Module Oracle.DataAccess | ft -auto

CommandType Name                Definition
----------- ----                ----------
Function    Connect-OracleTNS   ...
Function    Get-OracleDataTable ...
Function    OracleConnect       ...
Function    OracleDisconnect    ...
Function    OracleInvoke        ...
```

11. Get help for `OracleConnect` (output matches function comments):

```
PS > Get-Help OracleConnect
```

12. Make a connection with a TNS entry name of `LOCALDEV`:

```
PS > Connect-OracleTNS LOCALDEV HR pass
```

13. See how many rows would have been updated using `WhatIf`:

```
PS > OracleInvoke -sql "UPDATE HR.EMPLOYEES SET MANAGER_ID = 114
WHERE MANAGER_ID = 108" -WhatIf
What if: Performing operation "OracleInvoke" on Target "LOCALDEV".
5 row(s) affected
```

14. Disconnect using `–Verbose`:

```
PS > OracleDisconnect -verbose
VERBOSE: Closed connection to Data Source=LOCALDEV;User Id=HR;
```

15. Optionally remove the module when finished with it:

```
PS > Remove-Module Oracle.DataAccess
```

How it works...

PowerShell modules provide better componentization than normal script files. First, any script level variable in a module won't conflict with any same-named variable outside the module and module variables cannot be accessed outside the module. Second, modules allow selectively exporting certain commands as public while keeping other helper type functions private to the module. Another module benefit (among others) is easily adding and removing modules by name without worrying about the filename.

First we "installed" the module under `My Documents\WindowsPowerShell\Modules`, where the subfolder name is the name of the module and the module filename matches the module folder name. We could have modified the `PSModulePath` environment variable to point to another location, such as a company name under Program Files. If we wanted to temporarily install the module we could also add it via its full filename.

An `OdpVersion` module parameter defines which .NET version of the ODP.NET assembly to load and the load call is made at the end of the module, automatically loading ODP.NET when the module is imported so functions are ready for use. Note that when the module is imported, `–ArgumentList 2` is used; unlike with script or function parameters, we can't use named parameters when importing modules. Also note the `–Prefix Oracle`, which prefixes `Oracle` to the names of the nouns in the names of the imported members; `Get-DataTable` becomes `Get-OracleDataTable` for example. This helps in avoiding name conflicts and aids in function description, meaning, and grouping. Public module functions should follow a Verb-Noun naming convention and use standard PowerShell verbs or a warning will be output when the module is imported.

Above the `Connect` function, comment-based help is used to support `Get-Help` calls, which is always a good idea for shared modules. In this function and others, the `[CmdletBinding()]` attribute is used to enable use of `Write-Verbose` and `Write-Debug`, `-WhatIf`, and other common parameters support such as `–ErrorVariable`. The `Connect` function was changed to store the connection object in a script level variable (unless `–PassThru` is present) so it is available for other functions needing a connection later, without needing to pass it each time. Functions like `Disconnect` use the connection passed in if set; otherwise, they use the script level connection object.

The `Invoke` function sets the `SupportsShouldProcess` property of the `CmdletBinding` attribute to `$true` to enable `–WhatIf` support so we can test a non-query SQL method to see how many rows would be affected if committed. The `ShouldProcess` method is called on the built-in `$psCmdlet` object and is given the connection's data source as the target name for output. When this method returns `$false`, it means `–WhatIf` was not supplied and the transaction will get committed. Otherwise, the transaction is rolled back and the number of affected rows is output.

After the functions, `Export-ModuleMember` lists the items we want publicly exposed. Since functions such as `Load` and `Get-Connection` were not included, they can only be used inside the module as helpers. `Export-ModuleMember` can be passed multiple items, called multiple times, and used for other items like aliases and variables.

There's more...

This recipe just scratches the surface of what can be done with PowerShell modules, advanced functions, and Oracle. We could port and refactor more prior Oracle functions to cmdlet functions, define aliases, support pipeline input, accept script blocks as parameters, tackle other Oracle operations, and much more.

Automating SQL*Plus (Advanced)

In this recipe we'll look at using SQL*Plus (included with ODP.NET) from PowerShell to execute larger blocks of SQL, such as combined DDL scripts to update multiple procedures and views. ODP.NET is great for basic select, insert, update, and delete statements, executing procedures and the like but it breaks down for executing larger, complex batches of scripts such as making many object changes for an application deployment. Issues with linefeeds, BEGIN/END blocks, and multiple objects in one file can occur with ODP.NET but are well-handled in SQL*Plus.

Getting ready

Ensure you have a TNS name entry set up for Oracle Express on localhost. See the *Connecting and disconnecting (Simple)* recipe for more details.

How to do it...

1. Create `SqlPlus.ps1` and add the main driver script that will iterate through subfolders of the script's directory and execute all SQL files it finds. Each subfolder will be named to match an Oracle user/schema:

```
function Main {
    $server = read-host "Server TNS name"
    pushd (Get-ScriptDirectory)
    $userDirs = @(gci | ? {$_.PsIsContainer -and $_.Name -ne
"Logs"})

    foreach ($dir in $userDirs) {
        $user = $dir.Name
        $password = Get-Password($user)
        $sw = [System.Diagnostics.StopWatch]::StartNew()
        $runCountBefore = $script:_runCount
        Invoke-Scripts $server $user $password
        $sw.Stop()
        "Ran $($script:_runCount - $runCountBefore) script(s) " +
            "for $user@$server in $($sw.Elapsed.TotalSeconds)
second(s)"
    }

    popd
    "Finished. Ran $script:_runCount script(s) total"
}

$script:_dateId = "{0:MM-dd-yyyy.hh-mm-ss}" -f (Get-Date)
$script:_runCount = 0
Main
```

2. Define a utility function to prompt for, and return a password:

```
function Get-Password($user) {
    $pwd = read-host -AsSecureString "Password for $user@$server"
    [System.Runtime.InteropServices.Marshal]::PtrToStringAuto(`
             [System.Runtime.InteropServices.Marshal]::SecureStrin
gToBSTR($pwd))
}
```

3. Define a function to iterate through the files in the subfolder matching the user/schema, execute each, and move executed files to a completed folder:

```
function Invoke-Scripts($server, $user, $password) {
    "Running scripts for $user on $server"
    pushd $user
```

```
    $files = @(gci | where {!$_.PsIsContainer})
  $count = 0

    foreach ($file in $files) {
        write-progress -activity "Running scripts for $user" `
      -currentoperation $file.name -status Executing `
      -PercentComplete (100*$count++/$files.count)
        Invoke-Script $server $user $password $file
        New-Directory ".\Completed"
        move-item -path $file.fullname -destination ".\Completed"
-Force
    }

    write-progress -activity "Running scripts for $user" `
        -status Complete -completed
    popd
    "Completed running $count script(s) for $user on $server"
}
```

4. Define a function to run a given SQL file using `SQL*Plus`:

```
function Invoke-Script($server, $user, $password, $file) {
    $logDir = ("..\Logs\{0}\{1}" -f $script:_dateId, $user)
    New-Directory $logDir
    $logFile = join-path $logDir ($file.basename + ".html")

    "Running $file against $user@$server"
    (Get-SqlPlusSQL $file.fullname) | sqlplus.exe -L -M "HTML ON
SPOOL ON" `
    -S "$user/""$password""@$server" >> $logfile 2>$1
    "Ran $file against $user@$server. Details at $logfile"
    $script:_runCount++

    if ($LASTEXITCODE -ne 0) {
        write-error ("ERROR executing {0}!" -f $file.FullName)
        invoke-item $logFile; exit
    }
}
```

5. Define a function that will read the contents of a SQL file and surround it with SQL*Plus commands to get the desired output:

```
function Get-SqlPlusSQL($filename) {
@"
    whenever sqlerror exit sql.sqlcode
    set echo off
    set termout off
```

```
        $(gc $filename -readcount 0 | Out-String)
        commit;
        exit
    "@
    }
```

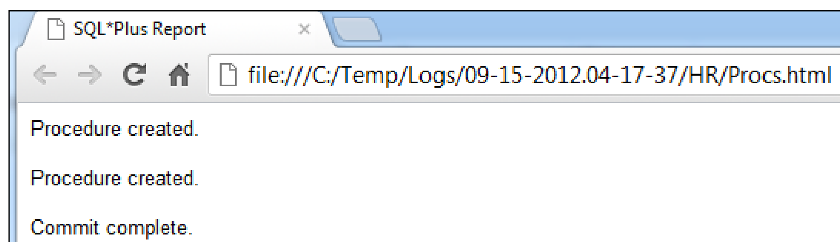
6. Define a helper function to create a new directory:

```
function New-Directory($dir) {
    if (!(test-path $dir)) {
        new-item $dir -type directory | Out-Null
    }
}
```

7. In the folder of `SqlPlus.ps1`, create a subfolder named `HR`
8. In a DBMS tool, connect to the sample `HR` database of Oracle Express and expand `Procedures` in the `HR` schema. Select `ADD_JOB_HISTORY` and `SECURE_DML` and export the DDL to a combined `Procs.sql` file in the `HR` folder. Likewise export `EMP_DETAILS_VIEW` to `EMP_DETAILS_VIEW.sql`.
9. Execute the script and supply a TNS name and a password when prompted:

```
Administrator: Windows PowerShell (x86)
PS > .\SqlPlus.ps1
Server TNS name: localdev
Password for HR@localdev: ****
Running scripts for HR on localdev
Running EMP_DETAILS_VIEW.sql against HR@localdev
Ran EMP_DETAILS_VIEW.sql against HR@localdev. Details at ..\Logs\09-15-2012.04-17-37\HR\EMP_DETAILS_VIEW.html
Running Procs.sql against HR@localdev
Ran Procs.sql against HR@localdev. Details at ..\Logs\09-15-2012.04-17-37\HR\Procs.html
Completed running 2 script(s) for HR on localdev
Ran 2 script(s) for HR@localdev in 1.1653406 second(s)
Finished. Ran 2 script(s) total
PS > _
```

10. Verify the LAST DDL date was updated on these objects in the database.
11. Explore the `Logs` directory created and open the log files:



How it works...

In the Main script we start by prompting for a server TNS name entry with Read-Host and changing to the script directory. Next, Get-ChildItem (gci) is invoked to get the items in the script directory that are folders ($_.PsIsContainer) not named Logs. For each directory, the directory name is used for user/schema name, a password is read from the host, and these values are passed to Invoke-Scripts to run the SQL files in that subdirectory.

Invoke-Scripts iterates through the files in the folder and calls Invoke-Script to execute each. It also takes care of progress reporting and moving executed scripts to a Completed subdirectory of the user/schema folder. This is more important if the SQL files have table schema changes and data migrations that may not be intended to be run multiple times. Note we are not recursively iterating folders and files here. If script files needed to be run in a certain order, we'd want to add a numeric prefix to the files so they were iterated over and executed in the right sequence.

Invoke-Script first sets up a log filename for the script to be executed and invokes Get-SqlPlusSQL to build an adjusted SQL statement for the script before piping that SQL to SQL*Plus. Since sqlplus.exe is in the ORACLE_HOME directory and that's in the PATH environment variable, we don't need to specify the path to it. The –L parameter tells sqlplus.exe to only attempt to log on once; otherwise any bad credentials could cause an input prompt and hang the script. The –M parameter indicates we want HTML output. The –S parameter invokes silent mode to prevent echoing of commands, banners, and prompts. Credentials are passed to sqlplus.exe using the user/password@server format. Output is redirected to $logfile and 2>$1 directs standard error to standard output. After this, $LASTEXITCODE is checked; 0 indicates success otherwise it is the Oracle error number. Any error is written with Write-Error, the log file is launched using Invoke-Item, and the script immediately terminates with exit since one script failure could impact other scripts to be executed.

The Get-SqlPlsSQL function uses whenever sqlerror exit sql.sqlcode to ensure $LASTEXITCODE will be set in the event of an error. It then sets echo and termout to off and includes the SQL contents of $filename with cat. A commit statement is added in case $filename contains insert, update, or delete statements without a commit; SQL*Plus should auto commit on exit as a default but here we are being explicit. Finally, exit is used to terminate SQL*Plus.

This approach provides a generic way to automate running any number of SQL scripts with PowerShell and SQL*Plus. In this example we're just reapplying some procedures and views but the same script could be used to execute hundreds of DML, DDL, or DCL changes that are common for application deployments.

Exploring ODT assemblies (Advanced)

This recipe will explore use of the Oracle.Management.Omo assembly included with **Oracle Developer Tools** for Visual Studio (**ODT**). This assembly is intended for internal use within Visual Studio so you will not find documentation or support for it and any code you write against it may break upon future ODT updates. Even so, this Oracle .NET library provides such rich functionality that these drawbacks may be acceptable.

Getting ready

You will need ODT installed; see the recipe *Setup Your Environment*. You will need to be able to load .NET 4.0 assemblies from PowerShell. See the *Loading ODP.NET 4.0* section in the *Accessing Oracle (Simple)* recipe for more.

How to do it...

1. Load the `Oracle.Management.Omo` assembly:

   ```
   PS > $odtAsmName = "Oracle.Management.Omo, Version=4.112.3.0,
   Culture=neutral, PublicKeyToken=89b483f429c47342"
   PS > $asm = [Reflection.Assembly]::Load($odtAsmName)
   ```

2. Look at the types in the assembly that start with "`Con`":

   ```
   PS > $types = $asm.GetTypes() | ? {$_.IsPublic}
   PS > $types | ? {$_.Name -like 'Con*'} | sort {$_.FullName } | ft
   -auto FullName

   FullName
   --------
   Oracle.Management.Omo.ConnectAs
   Oracle.Management.Omo.Connection
   Oracle.Management.Omo.ConstraintBase
   Oracle.Management.Omo.ConstraintCollection
   Oracle.Management.Omo.ConstraintInitialState
   Oracle.Management.Omo.ConstraintType
   ```

3. See what properties are on the connection object, as shown in the following screenshot:

```
Administrator: Windows PowerShell (x86)
PS > ($types | ? {$_.Name -eq "Connection"}).GetProperties() `
>>     | sort {$_.Name} -desc | ft -auto Name, PropertyType
>>

Name                                PropertyType
----                                ------------
UserID                              System.String
UseOptimizationLevelForDebug        System.Boolean
TnsAlias                            System.String
State                               Oracle.Management.Omo.ObjectState
ServerVersion                       System.String
Role                                Oracle.Management.Omo.DBAPrivilegeType
ProxyUserpasswd                     System.String
ProxyUsername                       System.String
PreserveCase                        System.Boolean
Password                            System.String
```

4. Create the connection object:

```
$conn = New-Object Oracle.Management.Omo.Connection `
    -property @{UserID="HR"; Password = "pass"; TnsAlias =
"LOCALDEV"}
```

5. Inspect methods on the connection object:

```
Administrator: Windows PowerShell (x86)
PS > $conn | gm | ? {$_.MemberType -eq "Method"} | sort -desc {$_.Name} | ft -auto Name, Definition

Name                           Definition
----                           ----------
UpdateIncludeSchemas           System.Void UpdateIncludeSchemas()
ToString                       string ToString()
SetObjectStateToInitialized    System.Void SetObjectStateToInitialized()
RevokeRoles                    System.Void RevokeRoles(string[] Roles, string[] Grantee)
Refresh                        System.Void Refresh()
Open                           System.Void Open()
Initialize                     System.Void Initialize()
```

6. Open the connection and initialize it to prepare for digging deeper:

```
$conn.Open(); $conn.Initialize()
```

7. Get a collection of table objects and list table names:

```
PS > $tables = $conn.GetTables($false, $true)
PS > $tables | sort {$_.Name} | foreach {$_.Name}
COUNTRIES
DEPARTMENTS
EMPLOYEES
JOB_HISTORY
JOBS
LOCATIONS
REGIONS
```

8. Get the `DEPARTMENTS` table and initialize it:

```
PS > $deptTable = $tables["DEPARTMENTS"]
PS > $deptTable.Initialize()
```

9. Display column information for the `DEPARTMENTS` table:

```
PS > $deptTable.Columns | ft -auto Ordinal,Name, IsNullable,
>>     @{Label="Type"; Expression={"{0} ({1})" -f $_.DataType.
OracleType, $_.DataType.Size}}
>>

Ordinal Name            IsNullable Type
------- ----            ---------- ----
      1 DEPARTMENT_ID        False NUMBER (0)
      2 DEPARTMENT_NAME      False VARCHAR2 (30)
      3 MANAGER_ID            True NUMBER (0)
      4 LOCATION_ID           True NUMBER (0)
```

10. Generate a CREATE TABLE script for `DEPARTMENTS`:

```
PS > $deptTable.GetCreateSQLs($true)[0]
CREATE TABLE "HR"."DEPARTMENTS" (
  "DEPARTMENT_ID" NUMBER(4,0) NOT NULL,
  "DEPARTMENT_NAME" VARCHAR2(30 BYTE) NOT NULL,
  "MANAGER_ID" NUMBER(6,0) NULL,
  "LOCATION_ID" NUMBER(4,0) NULL,
  CONSTRAINT "DEPT_LOC_FK"
    FOREIGN KEY ( "LOCATION_ID")
       REFERENCES "HR"."LOCATIONS" ( "LOCATION_ID")
    ENABLE
    VALIDATE,
  CONSTRAINT "DEPT_MGR_FK"
    FOREIGN KEY ( "MANAGER_ID")
       REFERENCES "HR"."EMPLOYEES" ( "EMPLOYEE_ID")
    ENABLE
    VALIDATE)
  STORAGE (
    NEXT 1048576 )
```

11. Get `DEPARTMENT` table data and filter it down:

```
PS > $deptDS = $deptTable.GetData($true)
PS > $deptDS.Tables["DEPARTMENTS"] | ? {$_.department_id -lt 40} |
ft -auto DEPARTMENT_ID, DEPARTMENT_NAME

DEPARTMENT_ID DEPARTMENT_NAME
------------- ---------------
           10 Administration
           20 Marketing
           30 Purchasing
```

12. Close the connection when finished:

```
PS > $conn.Close()
```

How it works...

The `Oracle.Management.Omo` assembly is located under the `ORACLE_BASE\ORACLE_HOME\odt` folder and installed into the GAC (that is, `%WINDIR%\Microsoft.NET\assembly\GAC_32\Oracle.Management.Omo`). After loading the assembly, we explore the types in the assembly with `$types = $asm.GetTypes()`. Spotting a `Connection` object in the results, we see what properties are available on it with `($types | ? {$_.Name -eq "Connection"}).GetProperties()`. With the properties known, we create an `Omo.Connection` object and pass it connection details. We then inspect methods available to us on the connection object using `Get-Member` (gm) using `$conn | gm | ? {$_.MemberType -eq "Method"}`. We notice both `Open` and `Initialize` methods on the connection so we invoke both in that order. If we didn't call `Initialize` we'd get an error later when we tried to drill down into further use of the connection. Most of the objects in this assembly work this way; you can get back basic details without having to initialize but to drill into child objects you have to initialize the parent.

Next we get back a rich collection of table objects with `$tables = $conn.GetTables($false, $true)`, passing `$false` for the refresh parameter and `$true` to fetch dependencies. From there we iterate through the tables and print out table names. Next we reference the `DEPARTMENTS` table with `$deptTable = $tables["DEPARTMENTS"]` and call `Initialize()` on it so we can access column data. We pipe the columns to `Format-Table` (`ft`) and use an expression to concatenate sub object properties into the results: `@{Label="Type"; Expression={"{0} ({1})" -f $_.DataType.OracleType, $_.DataType.Size}}`.

Next we generate a CREATE table script with `$deptTable.GetCreateSQLs($true)[0]`; `$true` indicates the schema name should be appended and we access item 0 since the method returns an `ArrayList`. Finally, we invoke the `GetData` method on the table (passing `$true` to append the schema name in the query) to get a `DataSet` created with the table data and then we close the connection.

There's more...

This is only a small fraction of `Oracle.Management.Omo` usage. We can query, alter, and otherwise interact with nearly all Oracle types. If you can do it through Visual Studio using Server Explorer and related integration points, you can probably do it through PowerShell. While we could do much of this ourselves using `Oracle.DataAccess` and interacting with Oracle system objects, that would involve a lot of raw plumbing already done for us inside this ODT assembly. To explore this assembly more easily, create a project in Visual Studio that references the `Oracle.Management.Omo assembly`, or use a .NET disassembler tool such as JetBrain's dotPeek to explore all the objects and their members.

There is some risk and trial and error using this approach but you may find the reward worth it. Note ODT includes other assemblies such as `Oracle.VsDevTools.dll` but `Oracle.Management.Omo` is the only logical choice from PowerShell.

Visualizing data (Advanced)

This recipe will explore visualizing Oracle data with charting using PowerShell.

Getting ready

Use `Oracle.DataAccess.psm1` from the recipe *Creating reusable script modules (Advanced)*, also included with the sample code files for this recipe. You will need to be able to load .NET 4.0 assemblies from PowerShell to use Microsoft Charting, which is included with .NET Framework 4.0. See the *Loading ODP.NET 4.0* section in the *Accessing Oracle* recipe for more. Alternatively, you can install Microsoft Charting for .NET Framework 3.5 from `http://www.microsoft.com/en-us/download/details.aspx?id=14422` without any changes for PowerShell v2.

How to do it...

1. Create `Visualize.ps1` and add these script parameters:

```
param (
[string]$outputFile = ".\Salaries.png",
[string]$outputFormat = "PNG",
[bool]$interactive = $true,
[bool]$openImage = $false)
```

2. Define a function to get average salary data by job id into a `DataTable`:

```
function Get-SalaryData {
    Import-Module .\Oracle.DataAccess.psm1 -ArgumentList 2
    $sql = "select job_id, round(avg(salary), 2) avg_sal from
hr.employees group by job_id order by job_id"
    Connect-TNS -TNS LOCALDEV -UserId HR -Password pass
    $dt = Get-DataTable -sql $sql
    Disconnect; $dt
}
```

3. Load the assemblies we'll need for charting:

```
[void][Reflection.Assembly]::LoadWithPartialName("System.Windows.
Forms")
[void][Reflection.Assembly]::LoadWithPartialName("System.Windows.
Forms.DataVisualization")
Create a chart object and set up the title:
```

```
$chart = New-object System.Windows.Forms.DataVisualization.
Charting.Chart `
    -property @{
        Width=800; Height=400; BackColor=[System.Drawing.
Color]::Transparent
        Dock = [System.Windows.Forms.DockStyle]::Fill
    }
$chartTitle = $chart.Titles.Add("Average Salaries by Job Code")
$chartTitle.Font = new-object drawing.font("calibri",18,[drawing.
fontstyle]::Regular)
```

4. Set up the chart area:

```
$chartArea = New-Object System.Windows.Forms.DataVisualization.
Charting.ChartArea
$chartArea.Area3DStyle.Enable3D = $true
```

5. Set up the Y-axis of the chart area:

```
$yAxis = $chartArea.AxisY
$yAxis.Title = "Salaries"
$yAxis.Interval = 5000
$yAxis.LabelAutoFitMinFontSize = 16
$yAxis.LabelStyle.Font = new-object drawing.
font("calibri",14,[drawing.fontstyle]::Regular)
$yAxis.TitleFont = new-object drawing.font("calibri",18,[drawing.
fontstyle]::Regular)
```

6. Set up the X-axis of the chart area:

```
$xAxis = $chartArea.AxisX
$xAxis.Interval = 1
$xAxis.LabelAutoFitMinFontSize = 16
$xAxis.Title = "Job Codes"
$xAxis.LabelStyle.Font = new-object drawing.
font("calibri",14,[drawing.fontstyle]::Regular)
$xAxis.TitleFont = new-object drawing.font("calibri",18,[drawing.
fontstyle]::Regular)
```

7. Add the chart area:

```
$chart.ChartAreas.Add($ChartArea)
Set up the chart data binding:
$series = $chart.Series.Add("Data")
$series.XValueMember = "job_id"; $series.YValueMembers = "avg_sal"
$series["DrawingStyle"] = "Cylinder"
$chart.DataSource = $dt; $chart.DataBind()
```

8. Set up a form to place the chart onto and show it:

```
$form = New-Object Windows.Forms.Form -property `
    @{ Text=$chartTitle.Text; Width=900; Height=600;
    StartPosition = [Windows.Forms.FormStartPosition]::CenterScree
n }
$form.controls.add($Chart)
```

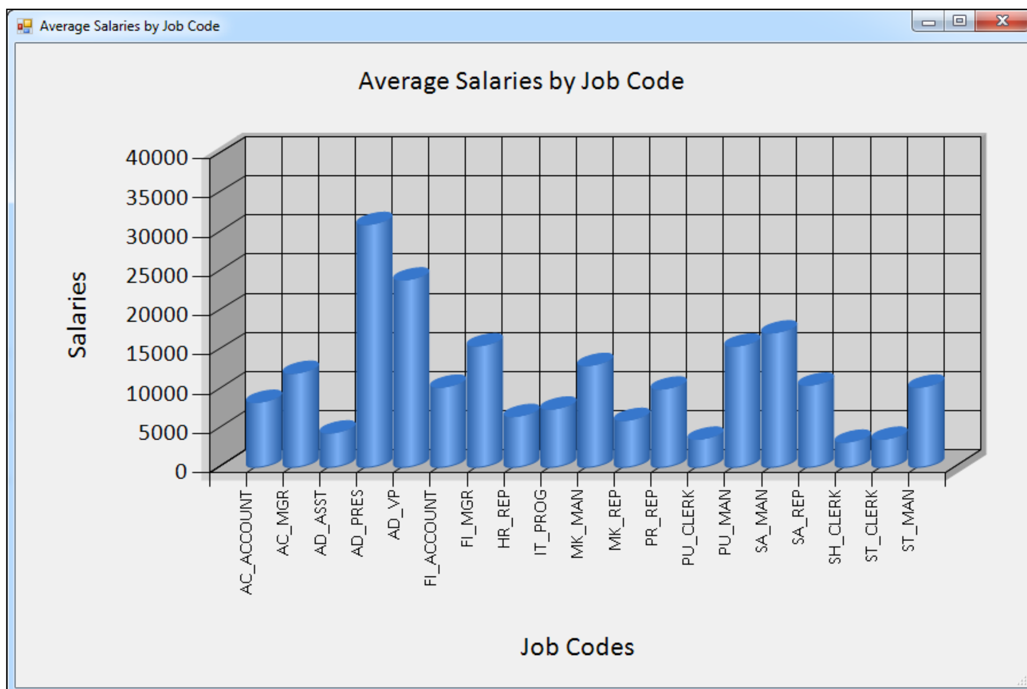
9. Save and open an image of the rendered chart if so configured:

```
if ($outputFile -and $outputFormat) {
    $chart.SaveImage($outputFile, $outputFormat)
    if ($openImage) { ii $outputFile }
}
```

10. If configured, show the chart on a windows form:

```
if ($interactive) {
    $form.Add_Shown({$Form.Activate()})
    $form.ShowDialog()
    $form.Dispose()
}
```

11. Change the directory to the location of `Visualize.ps1` and run the script. The result is as follows:



How it works...

At the top of the script, some parameters with default values are defined that allow the script to create (and optionally open) an image of the chart, and/or show the chart on a Windows form.

The `Get-SalaryData` function imports the `Oracle.DataAccess.psm1` module created for the recipe *Creating reusable script modules (Advanced)* but in this case, the module is imported by filename, looking for it in the current directory. SQL is defined to get the average salary by job ID and return a `DataTable` of the results.

Next, the assemblies are loaded for Windows Forms and Windows Forms Charting to create the screen and chart for display. The first step is creating and setting properties on a `System.Windows.Forms.DataVisualization.Charting.Chart` object, followed by setting up the chart area that is set up for 3D rendering.

The Y-axis of the chart area is then set up with an `Interval` of `5000` so we get a reasonable amount of salary axis labels and tick marks on the left vertical axis. The script also adjusts the font size for better readability. The X-axis is set up in a nearly identical format, except an `Interval` of `1` is used to ensure every job id/code is rendered. Finally the chart area object is added to the chart's `ChartAreas` collection.

Chart data binding is set up next, with the `job_id` column of the `DataTable` (`$dt`) being specified for the X-axis value member and `avg_sal` used for the Y-axis. Lastly the `DataSource` of the chart is set to the `DataTable`, and `DataBind()` is called.

To show the chart, the script sets up a new `Windows.Forms.Form` object onto which the control will be added to the form's `Controls` collection. If the `$outputFile` and `$outputFormat` script parameters are set, the `SaveImage` method of the chart is used to create the chart image. If the `$openImage` script parameter is `$true`, `Invoke-Item (ii)` is used to open the image in the default associated application. Finally if the `$interactive` script parameter is set, a `Shown` event handler is added with `Add_Shown` that will activate the form to bring it to the front when shown; then the window is shown as a dialog and the form is disposed.

There's more...

There is a lot more that can be done with the Microsoft Chart Controls for the .NET Framework, both with the data and the visual appearance of the chart.

There are a number of other tools that can be used to visualize data in PowerShell:

- JavaScript/JQuery/HTML/CSS can be output using PowerShell, leveraging JavaScript libraries and web browsers to create and render the chart. For example, the JQuery Visualize plugin can be found at `http://www.filamentgroup.com/lab/update_to_jquery_visualize_accessible_charts_with_html5_from_designing_with/`.
- The SeeShell PowerShell module provides a professional, supported PowerShell component for data visualization at `http://www.codeowls.com/SeeShell/Features`.
- For network graphs, NetMap from Microsoft Research can be used: `http://www.dougfinke.com/blog/index.php/2008/08/31/microsoft-research-netmap-and-powershell/`.
- A number of third-party component vendors provide control suites that included sophisticated charting controls. A few examples are as follows:
    - `http://www.telerik.com/`
    - `http://www.infragistics.com/`
    - `http://www.devexpress.com/`
    - `http://www.componentone.com/`

Thank you for buying
Instant Oracle Database and PowerShell How-to

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

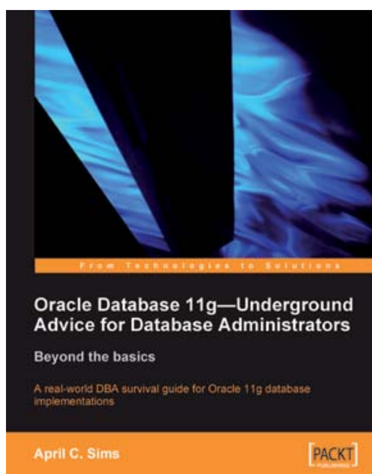
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



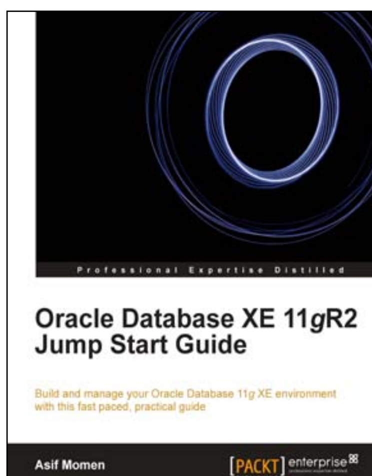
Oracle Database 11g – Underground Advice for Database Administrators

ISBN: 978-1-849680-00-4 Paperback: 348 pages

A real-world DBA survival guide for Oracle 11g database implementations

1. A comprehensive handbook aimed at reducing the day-to-day struggle of Oracle 11g Database newcomers
2. Real-world reflections from an experienced DBA—what novice DBAs should really know
3. Implement Oracle's Maximum Availability Architecture with expert guidance

---



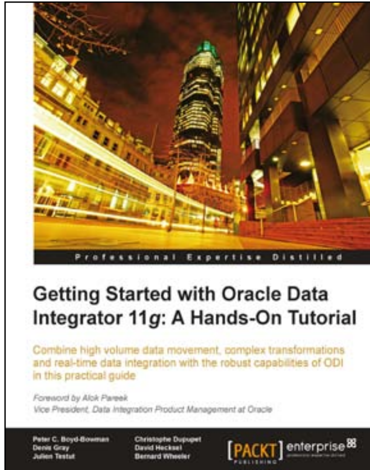
Oracle Database XE 11gR2 Jump Start Guide

ISBN: 978-1-84968-674-7 Paperback: 146 pages

Build and manage your Oracle Database 11g XE environment with this fast paced, practical guide

1. Install and configure Oracle Database XE on Windows and Linux
2. Develop database applications using Oracle Application Express
3. Back up, restore, and tune your database
5. Includes clear step-by-step instructions and examples

Please check **www.PacktPub.com** for information on our titles



Getting Started with Oracle Data Integrator 11g: A Hands-On Tutorial

ISBN: 978-1-849680-684 Paperback: 384 pages

Combine high volume data movement, complex transformations and real-time data integration with the robust capabilities of ODI in this practical guide

1. Discover the comprehensive and sophisticated orchestration of data integration tasks made possible with ODI, including monitoring and error-management

2. Get to grips with the product architecture and building data integration processes with technologies including Oracle, Microsoft SQL Server and XML files

---

Oracle 11g R1/R2 Real Application Clusters Essentials

ISBN: 978-1-849682-66-4 Paperback: 552 pages

Design, implement, and support complex Oracle 11g RAC environments for real world deployments

1. Understand sophisticated components that make up your Oracle RAC environment such as the role of High Availability, the RAC architecture required, the RAC installation and upgrade process, and much more!

2. Get hold of new Oracle RAC components such as the new features of Automatic Storage Management (ASM), performance tuning, and troubleshooting.

Please check **www.PacktPub.com** for information on our titles