Analyze Your Kindle Book Sales with KDP Sales Reports

Introduction

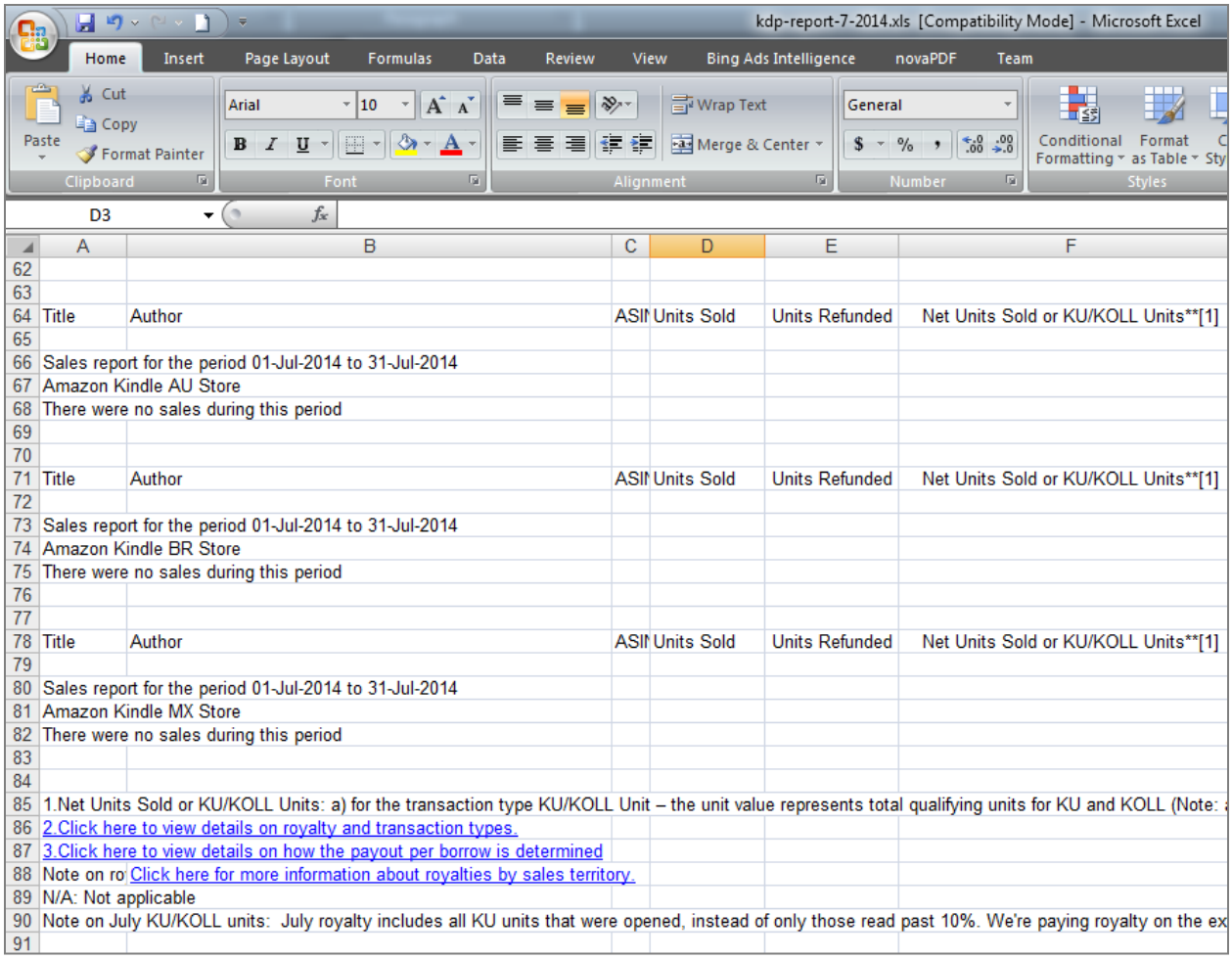
Amazon's Kindle Direct Publishing (KDP) is a free, self-serve publishing platform that allows authors and publishers to make digital versions of their books available to a worldwide audience through the Kindle Store. In exchange for this service, Amazon keeps a percentage of each sale and gives the rest to the author or publisher.

Rather than merely showing a summary of the royalties earned, Amazon provides authors and publishers with a detailed breakdown of sales for each title, including but not limited to the sale date, marketplace (Amazon US, UK, AU, DE, etc.), number of units sold, and number of refunds. The KDP Dashboard also features a couple of graphs showing sales and page-reads for a given period of time.

Unfortunately, however, the reports that you can generate on-the-fly within the Dashboard won't show any information that's older than 90 days. If you want to go further back, you'll have to download a separate Excel 2003 file (.xls) for each month of each year. While it's not a big deal to look at a single report here and there, looking at all of the reports for multiple years at once and trying to make sense of them all can be a burden if your only tool is Microsoft Excel operating in "Compatibility Mode."

While it's possible to merge the files into one by using a macro, that experience might ruin your day for a lot of reasons, not the least of which is that the structure of the reports is inconsistent. The columns that you see in one month's report may not exist in another month's report, due to some change or other that Amazon has made in the Kindle publishing model. For example, newer reports may show a book's enrollment and performance in a type of program or promotion that wasn't available a year ago.

In addition to mismatched columns, there are also hyperlinks and notes in the "Title" column that you probably wouldn't want in the final product, assuming you were to merge the files. You can also see from the following image that each Amazon marketplace has its own section of rows and columns in the same worksheet.



	A	B	C	D	E	F
62						
63						
64	Title	Author	ASIN	Units Sold	Units Refunded	Net Units Sold or KU/KOLL Units**[1]
65						
66	Sales report for the period 01-Jul-2014 to 31-Jul-2014					
67	Amazon Kindle AU Store					
68	There were no sales during this period					
69						
70						
71	Title	Author	ASIN	Units Sold	Units Refunded	Net Units Sold or KU/KOLL Units**[1]
72						
73	Sales report for the period 01-Jul-2014 to 31-Jul-2014					
74	Amazon Kindle BR Store					
75	There were no sales during this period					
76						
77						
78	Title	Author	ASIN	Units Sold	Units Refunded	Net Units Sold or KU/KOLL Units**[1]
79						
80	Sales report for the period 01-Jul-2014 to 31-Jul-2014					
81	Amazon Kindle MX Store					
82	There were no sales during this period					
83						
84						
85	1.Net Units Sold or KU/KOLL Units: a) for the transaction type KU/KOLL Unit – the unit value represents total qualifying units for KU and KOLL (Note:					
86	2.Click here to view details on royalty and transaction types.					
87	3.Click here to view details on how the payout per borrow is determined					
88	Note on ro Click here for more information about royalties by sales territory.					
89	N/A: Not applicable					
90	Note on July KU/KOLL units: July royalty includes all KU units that were opened, instead of only those read past 10%. We're paying royalty on the ex					
91						

By contrast, the reports generated on-the-fly by selecting a pre-determined time period and clicking the **Generate Reports** button are much more pleasing. The columns are consistent from one report to the next, and the transactions are ordered by date, as opposed to marketplace.

Here's what a generated report looks like (with some details censored to protect my innocence):

	A	B	C	D	E	F	G	H
1	Royalty Date	Title	Author Name	ASIN	Marketplace	Royalty Type	Transaction Type	Units Sold
2	Jan 07, 2016				Amazon.com	70%	Standard	1
3	Jan 07, 2016				Amazon.com	70%	Standard	1
4	Jan 05, 2016				Amazon.com	70%	Standard	1
5	Jan 02, 2016				Amazon.com	70%	Standard	1
6	Dec 14, 2015				Amazon.co.jp	N/A	Free - Promotion	2
7	Dec 13, 2015				Amazon.de	N/A	Free - Promotion	1
8	Dec 13, 2015				Amazon.co.uk	N/A	Free - Promotion	1
9	Dec 13, 2015				Amazon.com	N/A	Free - Promotion	1
10	Dec 12, 2015				Amazon.de	N/A	Free - Promotion	1
11	Dec 12, 2015				Amazon.com	N/A	Free - Promotion	6

Despite the fact that the generated reports are much cleaner, there's still the problem of getting data older than 90 days into this nice format. Depending on how many monthly reports you have, consolidating the data in the Excel files by hand would be a one-time hardship (that could be outsourced to a clerical freelancer). After that, it would be easy to generate and download a new report every 30 or 90 days (or every week, if you like), but the next problem you'll face is that your royalty reports will be all dressed up with nowhere to go.

A situation like this warrants the need for a database with a user-friendly front-end that would allow one to upload a data file at the click of a button, after which it would be imported into the database. **KDP Sales Reports** is a web application that does just that. It's not simply a storage bin for royalty reports, though. You can choose from several queries that are designed to present data in meaningful ways.

For example, if you wanted to know which of your books are making the most money, you can select the "Total royalties per book (all-time)" query, which sorts the titles according to their earnings. If you run a publishing company, or if you're an individual author with several pen names spanning multiple fiction genres, then you might be interested in seeing your earnings ranked by author name. Since the Kindle Store is not limited to Amazon.com, you may also be curious about your sales in countries other than the U.S., in which case, you can choose the "Royalties from non-US markets" query.

The **demo of KDP Sales Reports** will be available until my free trial of Microsoft Azure expires. It was developed with ASP.NET 4.0, VB.NET, and SQL Server 2008. The application's **web.config** file shows the ASP.NET version as 4.5.2 because that's the version required by Azure, but since my version of Visual Studio (2010 Professional) isn't compatible with 4.5.2, I had to make this change to the configuration after moving the application to Azure.

Ideally, an application of this nature would reside behind a login screen. In its current form, KDP Sales Reports is meant to be used by one author or publisher, since there's only one database and no mechanism to distinguish who is uploading what data or who has access to said data. It's just a proof of concept, so feel free to take the **source code** and run with it.

In the GitHub repository, you'll also find a sample data file (**RoyaltyReportJanuary2016.txt**) that you can import into the demo version of KDP Sales Reports. The application accepts a tab delimited .txt file as input, rather than an Excel (.xls) file, because even though Excel files can be imported into SQL Server, it's feasible to do so only in a desktop environment where Excel is installed.

Excel is not meant to run in a server environment, and not everyone who uses Amazon KDP has access to Excel (or SQL Server, for that matter). Such people can, however, open .xls files in a non-Microsoft spreadsheet program and convert them to tab delimited text files.

When you run the queries in the demo, you'll notice a lot of big-name authors whose books appear to be making money hand over fist. The database is a lie. My own royalties are unremarkable, so if I were to show the demo with real data, there wouldn't be much to see. Therefore, I decided to build a dummy data set for a fictitious publishing company that shall remain nameless.

Building the Dummy Data Set

When searching for a ready-made list of books and authors, I found this interesting collection on Wikipedia. From there, I copied and pasted nearly 200 entries into Notepad to get rid of the formatting and links, and then copied and pasted the Notepad content into a new Excel file. Since I wanted only the titles and authors, I deleted the third column (Literary Reference). The remaining two columns became Title and Author Name, respectively.

The KDP royalty reports that are generated on-the-fly have the following columns. The values of columns shown in orange are fixed, whereas the values in other columns are variable and/or calculated:

- Royalty Date
- Title
- Author Name
- ASIN
- Marketplace
- Royalty Type
- Transaction Type

- Units Sold
- Units Refunded
- Avg. List Price without Tax
- Avg. File Size (MB)
- Avg. Offer Price without Tax
- Avg. Delivery Cost
- Royalty
- Currency

To create fake data for the remaining columns, I built a Windows Forms application to do it for me because I didn't want to endure the mind-numbing exercise of coming up with a unique set of royalty details for each of 195 titles. So, the idea was to create a new spreadsheet, add the fixed values where appropriate, and then add the remaining data by creating sets of random-but-reasonable values and performing calculations on them.

The Windows Forms application is not included in the GitHub repo. It would have been a .exe file, and .exe files tend to inspire a fear of the unknown in the security conscious. However, I'll unleash the code here for the curious.

Creating a Unique ASIN for Each Title

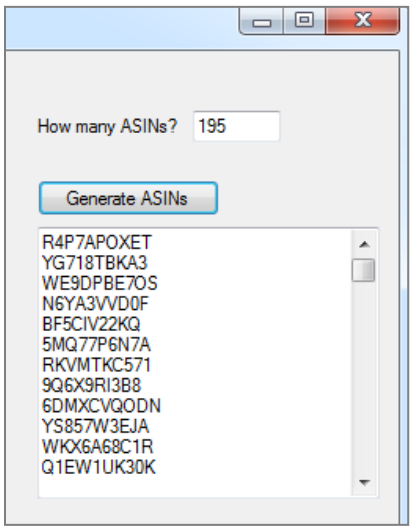
Every product on Amazon has an Amazon Standard Item Number (ASIN) consisting of 10 alphanumeric characters where letters are uppercase. Most, if not all, ASINs that I've seen begin with the letter B, but for our purposes, any 10-character alphanumeric string will do.

The following code assumes that the form (**Form1.vb [Design]**) has a Button and a TextBox named `btnAsin` and `txtAsin`, respectively. There's also a TextBox (`txtQty`) for accepting the desired number of ASINs. I could have hard-coded this number, but I decided to make the code "reusable" and to add some input validation for good measure. For reference, `txtAsin` is a MultiLine TextBox with the ScrollBars property set to Vertical, and `txtQty` has a MaxLength of 8 characters.

```
Private Sub btnAsin_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
btnAsin.Click
        ' Check for text
        If txtQty.Text.Trim() = String.Empty Then
            MessageBox.Show("Please enter a positive integer with no more than 4
digits.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
        End If
        If txtQty.Text.Trim() <> String.Empty Then
            ' Check for integer
            Dim qty As String = txtQty.Text.Trim()
```

```
            Dim pattern As String = "^\d+$"
            If Not Regex.IsMatch(qty, pattern) Then
                MessageBox.Show("The text must be an integer.", "Error",
MessageBoxButtons.OK, MessageBoxIcon.Error)
            End If
            ' Check for integer > 0
            If Int32.Parse(qty) = 0 Then
                MessageBox.Show("Please enter a positive integer.", "Error",
MessageBoxButtons.OK, MessageBoxIcon.Error)
            End If
            ' Check number of digits
            Dim int As Integer = Int32.Parse(txtQty.Text.Trim())
            Dim str As String = int.ToString()
            Dim strLen As Integer = str.Length
            If strLen > 4 Then
                MessageBox.Show("Please enter an integer with no more than 4 digits.",
"Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
            Else
                ' Create the ASINs
                Dim list As New List(Of String)
                For i As Integer = 1 To int
                    list.Add(GenerateAsin())
                Next
                ' Get rid of duplicates, however unlikely they are to occur
                ' (since the RNGCryptoServiceProvider class was used to
                ' create the ASINs instead of a run-of-the-mill
                ' random number generator).
                ' http://stackoverflow.com/questions/418817/pros-and-cons-of-rngcryptoserviceprovider
                list.Distinct()
                Dim sb As StringBuilder = New StringBuilder()
                For j As Integer = 0 To list.Count - 1
                    sb.Append(list.Item(j) & Environment.NewLine)
                Next
                txtAsin.Text = sb.ToString().Trim()
            End If
        End If
End Sub

Private Function GenerateAsin() As String
        Dim chars() As Char = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ".ToCharArray
        Dim length As Integer = 9 ' if 10, the ASIN will be 11 characters
        Dim data(1) As Byte
        Dim rng As New RNGCryptoServiceProvider
        rng.GetNonZeroBytes(data)
        ReDim data(length)
        rng.GetNonZeroBytes(data)
        Dim asin As StringBuilder = New StringBuilder(length)
        For Each b As Byte In data
            asin.Append(chars(b Mod (chars.Length - 1)))
        Next
        Return asin.ToString()
End Function
```

Here's what the output looked like for the 195 ASINs that were needed for the dummy data set:



After creating the ASINs, I used Excel's **RANDBETWEEN** function (followed by a "Fill Down") to get the book prices and file sizes:

A2 | =RANDBETWEEN(499, 1199)/100

	A	B	C
1	AvgListPriceWithoutTax	AvgFileSizeMB	
2	11.51	1.94	
3	7.36	1.26	
4	7.03	0.44	
5	7.1	1.26	
6	6.37	1.24	
7	6.35	0.93	
8	9.73	1.95	
9	5.13	0.44	
10	11.89	1.33	
11	10.95	1.55	
12	10.23	1.74	
13	10.87	1.27	
14	5.68	0.81	
15	6.27	0.5	
16	7.17	1.19	
17	7.23	0.81	
18	5.82	1.48	

B2 =RANDBETWEEN(4, 200)/100

	A	B	C
1	AvgListPriceWithoutTax	AvgFileSizeMB	
2	11.03	1.25	
3	10.57	1.69	
4	8.42	1.78	
5	9.33	0.41	
6	8.77	1.93	
7	11.36	1.03	
8	6.94	0.32	
9	11.71	1.58	
10	7.86	0.94	
11	6.3	0.32	
12	9.28	1.45	
13	9.42	0.29	
14	7.76	1.25	
15	5.82	1.26	
16	5.32	1.97	
17	8.73	0.55	
18	11.21	1.47	

In the screen shots above, the column headings are slightly different than those in the bulleted list on pages 4-5. The reason is that in order to create the Royalty Date values, I wanted to move the existing data from Excel to SQL Server so that I could randomly select a set of rows for each day of a given month and year. Because column names in a database table can't have any spaces or special characters, I modified the headers accordingly.

Creating the Royalty Dates

The royalty reports that can be generated on-the-fly in the KDP Dashboard are ordered by date. Each row corresponds to one transaction, so transactions that occur on the same date share the same Royalty Date value.

To mimic a real-world scenario, I wanted multiple book sales to happen on each day of a given date range, but I didn't want the same set of books to be purchased every time. My solution was to import the spreadsheet into SQL Server on my local machine via the Windows Forms application, execute a stored procedure to choose a random sample of rows for each date, and paste the data into a new spreadsheet.

This new spreadsheet would serve as the master spreadsheet that would be converted to a tab delimited text file and "uploaded" to KDP Sales Reports running on localhost, where it would

then be imported into the SQL Server database, also on my local machine. Later, I would migrate the entire database to Microsoft Azure.

Previously, I mentioned that importing a .xls file into SQL Server has to be done on a machine where Excel is installed and that a web server is not the place to do it. In that case, the .xls file would need to be converted to either a tab delimited .txt file or a comma separated value (.csv) file before the data could be imported. For the dummy data set, it was appropriate to use a Windows Forms application that to connects to Excel. I then used the SqlBulkCopy class of the .NET Framework to import the data that I had gathered so far.

Below is the complete code for **Form1.vb** of the Windows Forms application, minus the code for generating the ASINs (see page 6). It assumes that **Form1.vb [Design]** has a Button named `btnSqlBulkCopy`, a DataGridView named `dgvSales` , and another Button named `btnRandom` that, when clicked, will call SQL Server and retrieve a random sample of data. To accomplish this, I used a BackgroundWorker, which has two methods associated with it: `bwRandom_DoWork` and `bwRandom_RunWorkerCompleted`.

The BackgroundWorker also creates values for the **UnitsSold** column of the spreadsheet. After assembling the sales data for the months of October, November, and December 2015, I added a Button named `btnRefunds` and wrote the code for getting the **UnitsRefunded** values.

```
Imports System
Imports System.Collections
Imports System.Collections.Generic
Imports System.ComponentModel
Imports System.Data
Imports System.Data.DataTableExtensions
Imports System.Data.OleDb
Imports System.Data.SqlClient
Imports System.Globalization
Imports System.IO
Imports System.Linq
Imports System.Math
Imports System.Security.Cryptography
Imports System.Text
Imports System.Text.RegularExpressions
Imports System.Threading
Imports AmazonBooks.Configuration
Imports AmazonBooks.SqlDataAccess
Imports AmazonBooks.StoredProcedures

Public Class Form1

Public Sub New()
        InitializeComponent()
        bwRandom.WorkerReportsProgress = False
        bwRandom.WorkerSupportsCancellation = False
End Sub
```

```
Private Sub btnSqlBulkCopy_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
btnSqlBulkCopy.Click
        Dim sqlTable As String = "Books"
        Dim dir As String = "C:\Users\MyUsername\Desktop"
        Dim excelFileName As String = "FakeSalesData_InputForSQL.xlsx"
        Dim excelSheetName As String = "[Sheet1$]"

        ' Different versions of Excel have different connection strings.
        ' This is the one for Excel 2007.
        Dim excelConnectionString As String = "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=" & Path.Combine(dir, excelFileName) & ";Extended Properties=""Excel
12.0;HDR=YES"""

        ' Bulk copy the data from Excel to the designated SQL table.
        Using conn As New OleDbConnection(excelConnectionString)
            Dim cmd As New OleDbCommand(("SELECT Title, AuthorName, ASIN,
AvgListPriceWithoutTax, AvgFileSizeMB FROM " & excelSheetName), conn)
            conn.Open()
            Using dr As OleDbDataReader = cmd.ExecuteReader()
                Using bulkCopy As New SqlBulkCopy(DbConnectionString)
                    bulkCopy.DestinationTableName = sqlTable
                    bulkCopy.ColumnMappings.Add("Title", "Title")
                    bulkCopy.ColumnMappings.Add("AuthorName", "AuthorName")
                    bulkCopy.ColumnMappings.Add("ASIN", "ASIN")
                    bulkCopy.ColumnMappings.Add("AvgListPriceWithoutTax",
"AvgListPriceWithoutTax")
                    bulkCopy.ColumnMappings.Add("AvgFileSizeMB", "AvgFileSizeMB")
                    Try
                        bulkCopy.WriteToServer(dr)
                        MessageBox.Show("Successfully added data to the database",
"Success", MessageBoxButtons.OK, MessageBoxIcon.Information)
                    Catch ex As Exception
                        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error)
                    End Try
                End Using
            End Using
        End Using
    End Sub

Private Sub btnRandom_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
btnRandom.Click
        ' Tell the BackgroundWorker to start working.
        bwRandom.RunWorkerAsync()
End Sub

Private Sub bwRandom_DoWork(ByVal sender As Object, ByVal e As DoWorkEventArgs) Handles
bwRandom.DoWork
        Dim bw As BackgroundWorker = TryCast(sender, BackgroundWorker)
        If bw IsNot Nothing Then
            ' Months to be included in DataSet (Month[# days]):
            ' October[31]
            ' November[30]
            ' December[31]
            ' Start with October 2015, and return to the code to change
            ' the following values for the remaining months (then run
            ' the code again by clicking the btnRandom button).
```

```
            Dim year As Integer = 2015
            Dim mont As Integer = 10
            Dim days As Integer = 31
            Dim ds As DataSet = New DataSet("BookSales")

            ' Each day of the chosen month will have its own DataTable.
            For i As Integer = 1 To days
                ' The SQL stored procedure takes the sale date as a parameter,
                ' so put the date in the correct format.
                Dim saleDate As String = mont.ToString() & "/" & i.ToString() & "/" &
year.ToString()
                Dim saleDateTime As DateTime = Convert.ToDateTime(saleDate)
                Dim saleDateStr As String = saleDateTime.ToString("yyyy-MM-dd")

                Dim cmd As New SqlCommand
                cmd.CommandType = CommandType.StoredProcedure
                cmd.CommandText = CN_GetRandomBooks
                cmd.Parameters.AddWithValue("@SaleDate", saleDateStr)
                Try
                    Dim dt As DataTable = ExecuteSelectCommand(cmd)
                    ' We need to add the column "UnitsSold" to the DataTable
                    ' and populate it with random numbers.
                    ' First, add the new column.
                    Dim col As DataColumn = New DataColumn("UnitsSold",
GetType(System.Int32))
                    dt.Columns.Add(col)
                    ' Get a List(Of Integer) containing randomly generated book sales.
                    ' The number of items in the list should be equal to dt.Rows.Count.
                    ' We will then iterate through the list and add the book sales to the
                    ' DataTable in the UnitsSold column for the specified row.
                    Dim unitsSold As List(Of Integer) = GenerateUnitsSold(dt.Rows.Count)
                    For j As Integer = 0 To dt.Rows.Count - 1
                        Dim currentRow As DataRow = dt.Rows.Item(j)
                        currentRow("UnitsSold") = unitsSold.Item(j)
                    Next
                    ' Now that the DataTable has been updated with the sales numbers, add
                    ' it to the DataSet that will hold all of the DataTables (one
                    ' DataTable for each day of the specified month).
                    ds.Tables.Add(dt)
                Catch ex As Exception
                    e.Result = ex.Message
                End Try
            Next
            ' Pass the DataSet to RunWorkerCompleted.
            e.Result = ds
        End If
    End Sub

Private Sub bwRandom_RunWorkerCompleted(ByVal sender As Object, ByVal e As
RunWorkerCompletedEventArgs) Handles bwRandom.RunWorkerCompleted
        If e.Error IsNot Nothing Then
            MessageBox.Show(e.Error.Message & Environment.NewLine & "Method:
bwRandom_DoWork", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
        Else
            Dim ds As DataSet = CType(e.Result, DataSet)
            ' Merge the DataTables in the DataSet into one DataTable
            For i As Integer = 0 To ds.Tables.Count - 1
                ds.Tables(0).Merge(ds.Tables(i))
```

```
            Next
            ' Bind the DataTable to a DataGridView
            dgvSales.DataSource = ds.Tables(0)
            dgvSales.AllowUserToAddRows = False
        End If
End Sub

Private Function GenerateUnitsSold(ByVal numBooks As Integer) As List(Of Integer)
        Dim units As New List(Of Integer)
        Dim r As Random = New Random()
        For i As Integer = 0 To numBooks - 1
            units.Add(r.Next(1, 200)) ' generate from 1 to 200 sales
        Next
        Return units
End Function

Private Sub dgvSales_CellFormatting(ByVal sender As Object, ByVal e As
DataGridViewCellFormattingEventArgs) Handles dgvSales.CellFormatting
        ' First column in dgv is for RoyaltyDate
        If e.ColumnIndex = 0 Then
            Dim dat As New Date
            If Date.TryParse(e.Value.ToString(), dat) Then
                ' Put the date in the desired format. If you don't
                ' specify a format, the DataGridView will use
                ' MM/dd/yyyy.
                e.Value = dat.ToString("yyyy-MM-dd")
                e.FormattingApplied = True
            End If
        End If
End Sub

Private Sub btnRefunds_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
btnRefunds.Click
        Dim refunds As New List(Of Integer)
        Dim r As Random = New Random()
        For i As Integer = 0 To 1000
            refunds.Add(r.Next(0, 6)) ' generate from 0 to 6 refunds
        Next
        Dim dt As New DataTable
        dt.Columns.Add("UnitsRefunded", GetType(System.Int32))
        Dim row As DataRow
        For j As Integer = 0 To refunds.Count - 1
            row = dt.NewRow()
            row("UnitsRefunded") = refunds.Item(j)
            dt.Rows.Add(row)
        Next
        dgvSales.DataSource = dt
End Sub

End Class
```

`AmazonBooks` is the name of the Windows Forms application, as well as the Namespace for the class files shown at the top of **Form1.vb**. `AmazonBooks.Configuration` is a class that I used for holding the SQL Server connection string, while `AmazonBooks.SqlDataAccess` is a

class with methods for accessing data. `AmazonBooks.StoredProcedures` is for keeping the names of SQL stored procedures all in one place. This way of organizing data access code is something that I picked up from an excellent book by Deborah Kurata called *Doing Objects in Visual Basic 2005*.

The code for each class file is below, with sensitive information replaced by asterisks.

Configuration.vb

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common

Public NotInheritable Class Configuration

    ' Returns the database connection string
    Public Shared ReadOnly Property DbConnectionString() As String
        Get
            Dim bldr As New SqlConnectionStringBuilder()
            bldr.DataSource = "******\sqlserver2008" ' server name
            bldr.InitialCatalog = "AmazonBooks" ' database name
            bldr.UserID = "*****"
            bldr.Password = "************"
            bldr.IntegratedSecurity = True ' If True, Windows Authentication will be used
                                           ' If False, UserID and Password will be used
            Return bldr.ConnectionString
        End Get
    End Property

End Class
```

SqlDataAccess.vb

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports AmazonBooks.Configuration

Public Shared Function ExecuteSelectCommand(ByVal cmd As SqlCommand) As DataTable
        Dim dt As DataTable
        Using cnn As New SqlConnection(DbConnectionString)
            cnn.Open()
            cmd.Connection = cnn
            Dim rdr As SqlDataReader = cmd.ExecuteReader()
            dt = New DataTable
            dt.Load(rdr)
            rdr.Close()
        End Using
        Return dt
End Function
```

StoredProcedures.vb

```
Imports Microsoft.VisualBasic

Public Class StoredProcedures

    Public Const CN_GetRandomBooks As String = "procGetRandomBooks"

End Class
```

procGetRandomBooks

```
USE [AmazonBooks]
GO
/****** Object:  StoredProcedure [dbo].[procGetRandomBooks]    Script Date:
01/20/2016 23:52:28 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[procGetRandomBooks]
(
@SaleDate DATE
)
AS

DECLARE @RowsLowerBound INT;
DECLARE @RowsUpperBound INT;
DECLARE @NumTitles INT;

SET @RowsLowerBound = 10;
SET @RowsUpperBound = 150;
SET @NumTitles = ROUND(((@RowsUpperBound - @RowsLowerBound - 1) * RAND() +
@RowsLowerBound), 0);

SELECT TOP(@NumTitles) @SaleDate AS RoyaltyDate, Title, AuthorName, [ASIN],
AvgListPriceWithoutTax, AvgFileSizeMB
FROM Books
ORDER BY CHECKSUM(NEWID());
```

The "Books" Table Design (as shown in SQL Server Management Studio)

Column Name	Data Type	Allow Nulls
BookID	int	
Title	nvarchar(300)	
AuthorName	nvarchar(50)	
ASIN	nchar(10)	
AvgListPriceWithoutTax	smallmoney	
AvgFileSizeMB	decimal(5, 2)	

Filling in the Remaining Dummy Data

Once all of the data obtained by running the above code was pasted into Excel, I started doing some calculations on the columns. To get NetUnitsSold, I subtracted UnitsRefunded from UnitsSold. This resulted in a handful of negative integers, so I doctored the numbers a bit to make the values turn positive.

For the AvgOfferPriceWithoutTax, I used the same values as the AvgListPriceWithoutTax column. The AvgDeliveryCost was calculated by multiplying the AvgFileSizeMB by 0.15, which is the cost per MB (in U.S. dollars) that Amazon charges for delivering a Kindle book to the customer.

The Royalty amount is equal to the AvgListPriceWithoutTax multiplied by 0.7 because the Royalty Type is 70%, meaning that Amazon pays the author or publisher 70% of the list price. Amazon pays different royalty percentages for different marketplaces and different list prices, but I chose a flat 70% for the RoyaltyType.

Unfortunately, my efforts to format the date in the DataGridView were in vain. When I copied and pasted the values from the DataGridView into the spreadsheet, Excel automatically converted them to the same format that the DataGridView would have used if I hadn't specified otherwise. To fix this, I selected a custom date format by right-clicking on the cells and choosing "Format Cells."

The values in the Marketplace, RoyaltyType, TransactionType, and Currency columns are variable, but I all I did to fill them in was to choose the initial values and use Excel's "Fill Down" feature to complete the rest of the columns. In the case of Marketplace, I made sure that the Currency was appropriate for the given marketplace (e.g., for Amazon.co.uk, the Currency had to be GBP). Similarly, I set the AvgOfferPricewithoutTax to 0.00 where the TransactionType was "Free – Promotion."

When all was said and done, I ended up with about 6600 rows in the spreadsheet.

The KDP Sales Reports Web Application

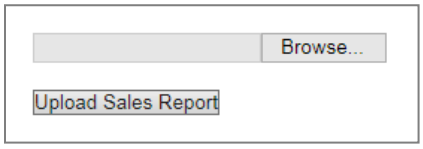
KDP Sales Reports consists of a single web page, where an author or publisher can upload a royalty report as a tab delimited text file. There are twelve database queries from which to choose. When a query is selected, input options may or may not appear, depending on whether or not the SQL stored procedure takes any parameters. A "Run Query" button will also appear.

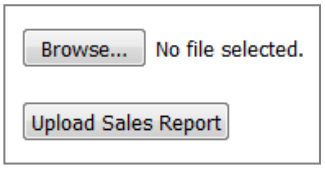
The Web Controls (buttons, input fields, etc.) for each query are organized in separate ASP.NET Web User Controls, which are akin to PHP includes. A Web User Control has a **.ascx** extension, whereas an ASP.NET web page has a **.aspx** extension. Web User Controls also have their own code-behind files with either a **.ascx.vb** extension for VB or a **.ascx.cs** extension for C#.

Uploading a Data File

The ASP.NET Framework provides a convenient FileUpload control, which has the familiar "Browse" button seen on many web pages where users can upload images and other files. Clicking on the **Browse** button will bring up a file explorer, and you can select the data file that you want to import. As mentioned, you'll find a dummy data file in the GitHub repository that you can use to test the application.

The appearance of the FileUpload control differs according to browser. The images below correspond to Internet Explorer 11 (left) and Firefox 43 x64 (right). Nowhere in my code do I use the words "No file selected." Apparently, it's a Firefox thing.





Importing the Data

Clicking the **Upload Sales Report** button initiates checks on the input file to make sure that it's of the correct type and size. If the file is acceptable, the application saves it to the server using a

new file name based on a GUID so that any files with the same original name are not overwritten.

When saving a spreadsheet as a tab delimited text file, there may be one or more blank lines at the end of the file's contents. This can be problematic whenever such a file needs to be processed in some way, so the application removes leading and trailing whitespace before doing anything with the data.

After approving and cleaning the file, KDP Sales Reports converts the tab delimited data into XML format. The main reason is because I wanted to add a new column to the data set that I would not expect the end user to include on his or her own for my programming convenience. The other reason is that it's an opportunity to get some vindication for VB.NET by showing off its "XML literals" feature.

The new column is **RefundRate**, and it's values are equal to the UnitsRefunded divided by the UnitsSold. Refund rates can be very telling. For instance, if you notice that a particular book is getting a lot of refunds, you may be prompted to investigate why that's happening. It might be that your readers just didn't care for your latest tale of intrigue, but it could have been something like a Kindle formatting error that made the book practically unreadable. You can also look to the reviews for insight, as many Amazon customers prefer the "one-star review" method to get the attention of authors when a Kindle book has technical issues. If refund rates are high, you might also consider how the book was promoted. Did you put it in front of the right audience, or was your advertising untargeted?

Once the data are in XML format, the web application imports the data into SQL Server via SqlBulkCopy. This time, however, the data source is an ADO.NET DataTable object constructed from the XML data. In the Windows Forms application, a .xlsx file was imported directly into the database without a middleman, so to speak.

Running the Queries

When a query is selected via RadioButton, a PlaceHolder in the .aspx page will show the appropriate User Control. If the query is executed, then a Repeater will show the resulting data. Selecting another query at this point will cause the User Control to disappear and another one to take its place. If you want to return to any query that you've already run, you can select that query again, and the persisted data will appear.

The GitHub repository contains a **.sql** file with all of the stored procedures. You can open this file in SQL Server Management Studio if you have it, or you may use a text editor.

A Guided Tour of the Source Code

The lone web page of KDP Sales Reports is **default.aspx**. (Many websites use index.html or index.php as the home page, but ASP.NET is a little different.) The "code-behind" file, where all of the magic happens, is named **default.aspx.vb**.

Default.aspx has several ASP.NET Web Controls, which are server-side HTML controls that are programmatically accessible. Incidentally, normal HTML tags can be accessed in the same way by ASP.NET, as long as the `runat="server"` attribute is added to the HTML tag. The code for **default.aspx** is too cumbersome to show in a PDF due to some very long lines, so hopefully a table showing the controls referenced in **default.aspx.vb** will suffice.

Control Type	Tag	Control Name (ID)
FileUpload	`<asp:FileUpload />`	`fupReport`
Button	`<asp:Button />`	`btnUpload`
Label	`<asp:Label></asp:Label>`	`lblStatus`
RadioButton	`<asp:RadioButton />`	`rad1`
RadioButton	`<asp:RadioButton />`	`rad2`
RadioButton	`<asp:RadioButton />`	`rad3`
RadioButton	`<asp:RadioButton />`	`rad4`
RadioButton	`<asp:RadioButton />`	`rad5`
RadioButton	`<asp:RadioButton />`	`rad6`
RadioButton	`<asp:RadioButton />`	`rad7`
RadioButton	`<asp:RadioButton />`	`rad8`
RadioButton	`<asp:RadioButton />`	`rad9`
RadioButton	`<asp:RadioButton />`	`rad10`
RadioButton	`<asp:RadioButton />`	`rad11`
RadioButton	`<asp:RadioButton />`	`rad12`
Label	`<asp:Label></asp:Label>`	`lblInstructions`
PlaceHolder	`<asp:PlaceHolder></asp:PlaceHolder>`	`phrOptions`

In addition to the controls in the table, the **default.aspx** page also has code resembling the following, placed just above the DOCTYPE declaration:

```
<%@ Page Language="VB" AutoEventWireup="false" CodeFile="default.aspx.vb" Inherits="_default"
MaintainScrollPositionOnPostback="true" %>
<%@ Register src="controls/ControlName1.ascx" tagname="ControlName1" tagprefix="uc1" %>
<%@ Register src="controls/ControlName2.ascx" tagname="ControlName2" tagprefix="uc2" %>
.
.
.
<%@ Register src="controls/ControlName11.ascx" tagname="ControlName11" tagprefix="uc11" %>
<%@ Register src="controls/ControlName12.ascx" tagname="ControlName12" tagprefix="uc12" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
```

User Controls must be "registered" in the .aspx page where they will be used so that the page is "aware" of them. KDP Sales Reports has 12 User Controls, one for each database query. Here's an example of a User Control's .ascx file:

```
<%@ Control Language="VB" AutoEventWireup="false" CodeFile="QueryTotalRoyaltiesOfAuthorControl.ascx.vb"
Inherits="controls_QueryTotalRoyaltiesOfAuthorControl" %>
<asp:DropDownList ID="ddlAuthors" runat="server" CssClass="aspnet_dropdownlist"></asp:DropDownList><br
/><br />
<asp:Button ID="btnRunQuery" runat="server" Text="Run Query" /> <asp:Label ID="lblQueryStatus"
runat="server"></asp:Label><br /><br />
<asp:Panel ID="pnlData" runat="server" CssClass="data">
<asp:Repeater ID="rptData" runat="server">
    <HeaderTemplate>
        <table width="100%" cellspacing="0">
            <tr>
                <td class="heading">Author Name</td>
                <td class="heading" align="right">Total Royalties (USD)</td>
            </tr>
    </HeaderTemplate>
    <ItemTemplate>
            <tr>
                <td><%# Eval("AuthorName")%></td>
                <td align="right"><%# Eval("TotalRoyalties", "{0:c}")%></td>
            </tr>
    </ItemTemplate>
    <FooterTemplate>
        </table>
    </FooterTemplate>
</asp:Repeater>
</asp:Panel>
```

When the Button (`btnRunQuery`) is clicked, the application will query the database and receive a DataTable object in return, which will be "bound" to a Repeater control. As its name implies, it allows data to be displayed in a repeatable format, or template. The code-behind file of a User Control looks similar to a code-behind file for an ASP.NET web page.

QueryTotalRoyaltiesOfAuthorControl.aspx.vb

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
```

```
Imports System.Web
Imports System.Web.UI.WebControls
Imports KDPSalesReports.Configuration
Imports KDPSalesReports.SqlDataAccess
Imports KDPSalesReports.StoredProcedures

Partial Public Class controls_QueryTotalRoyaltiesOfAuthorControl
    Inherits System.Web.UI.UserControl

    Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs) Handles Me.Load
        If Not Page.IsPostBack Then
            ' Bind a list of author names to a DropDownList.
            Dim dt As DataTable = GetAuthors()
            ddlAuthors.DataSource = dt
            ddlAuthors.DataTextField = "AuthorName"
            ddlAuthors.DataValueField = "ValueField"
            ddlAuthors.DataBind()
        End If
    End Sub

    Private Sub btnRunQuery_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
btnRunQuery.Click
        pnlData.Visible = True
        Try
            Using cnn As New SqlConnection(DbConnectionString)
                cnn.Open()
                Dim cmd As New SqlCommand
                cmd.Connection = cnn
                cmd.CommandType = CommandType.StoredProcedure
                cmd.CommandText = CN_TotalRoyaltiesOfAuthor
                cmd.Parameters.Add(New SqlParameter("@AuthorName",
SqlDbType.NVarChar)).Value = ddlAuthors.SelectedItem.ToString()
                Dim da As SqlDataAdapter = New SqlDataAdapter(cmd)
                Dim dt As DataTable = New DataTable()
                da.Fill(dt)
                rptData.DataSource = dt
                rptData.DataBind()
            End Using
        Catch ex As Exception
            lblQueryStatus.Text = ex.Message
            lblQueryStatus.CssClass = "status_error"
        End Try
    End Sub

    Private Function GetAuthors() As DataTable
        Dim cmd As New SqlCommand
        cmd.CommandType = CommandType.StoredProcedure
        cmd.CommandText = CN_GetAuthors
        Dim dt As DataTable = ExecuteSelectCommand(cmd)
        Return dt
    End Function

End Class
```

The other User Controls look very similar, so I'm not going to show them all. The `Configuration`, `SqlDataAccess`, and `StoredProcedures` class files are along the same

lines as described earlier when discussing the Windows Forms application. **Default.aspx.vb** handles the file upload, data conversion, and data import procedures. To keep things simple, I've left out the Imports statements and boring input validation stuff in the code shown from this point onward. There's a lot of code, so I'll walk through it one chunk at a time.

At the top, there are a few variable declarations:

```
Private _newFileNameNoExt As String
Private _txtPath As String
Private _xmlPath As String
Private _radQuerySelected As RadioButton
```

Due to the stateless nature of HTTP, server-side objects have to be recreated every time a page is requested, as well as when a postback occurs. Dynamic controls that are created at runtime must also be recreated every time an ASP.NET page posts back, such as in response to a Button's Click event. In order to persist data and handle events associated with the User Controls, these controls need to be instantiated in the page's **Init** event (the second phase of the ASP.NET page life cycle).

Below is the code for the Init event. In it (no pun intended), you can see that the User Controls are loaded and added to the PlaceHolder in **default.aspx**. The **CheckedChanged** event handlers for the RadioButtons are also assigned. A single CheckedChanged method (`radQuery_CheckedChanged`) handles the CheckedChanged event for each RadioButton.

```
Private Sub Page_Init(ByVal sender As Object, ByVal e As EventArgs) Handles Me.Init
        ctrlRad1 = Page.LoadControl("controls/QueryTotalRoyaltiesPerBookControl.ascx")
        ctrlRad2 = Page.LoadControl("controls/QueryTotalRoyaltiesOfAuthorControl.ascx")
        ctrlRad3 = Page.LoadControl("controls/QueryTotalRoyaltiesByAuthorControl.ascx")
        .
        .
        ctrlRad12 = Page.LoadControl("controls/QueryRefundRatesControl.ascx")
        phrOptions.Controls.Add(ctrlRad1)
        phrOptions.Controls.Add(ctrlRad2)
        phrOptions.Controls.Add(ctrlRad3)
        .
        .
        phrOptions.Controls.Add(ctrlRad12)
        AddHandler rad1.CheckedChanged, AddressOf radQuery_CheckedChanged
        AddHandler rad2.CheckedChanged, AddressOf radQuery_CheckedChanged
        AddHandler rad3.CheckedChanged, AddressOf radQuery_CheckedChanged
        .
        .
        AddHandler rad12.CheckedChanged, AddressOf radQuery_CheckedChanged
End Sub
```

When the page loads, the User Controls are invisible. They don't appear unless and until a RadioButton is selected, and only the relevant control will be displayed at that time. To prevent the controls from appearing the first time that the page is loaded while allowing them to appear upon postback, the **IsPostBack** property of the page can be set in the page's **Load** event. We can iterate through the controls in the PlaceHolder and hide all of the User Controls.

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs) Handles Me.Load
        If Not Page.IsPostBack Then
            For Each c As Control In phrOptions.Controls
                If TypeOf (c) Is UserControl Then
                    c.Visible = False
                End If
            Next
        End If
End Sub
```

The CheckedChanged event of a RadioButton is similar to a Click event of a Button. When the RadioButton is selected, any code that you want to execute should be placed in the RadioButton_CheckedChanged method. In KDP Sales Reports, selecting a RadioButton sets the value of `_radQuerySelected` so that the application can keep track of which RadioButton was chosen. Based on the UniqueID property of the selected RadioButton, the appropriate User Control will be made visible, while all of the other User Controls will remain hidden. Any instructions for the user regarding the query will show up in a Label (`lblInstructions`).

```
Private Sub radQuery_CheckedChanged(ByVal sender As Object, ByVal e As EventArgs)
        Dim rb As RadioButton = CType(sender, RadioButton)
        If rb IsNot Nothing Then
            If rb.Checked = True Then
                _radQuerySelected = rb
                If _radQuerySelected.UniqueID = "rad1" Then
                    lblInstructions.Text = String.Empty
                    ctrlRad1.Visible = True
                    ctrlRad2.Visible = False
                    ctrlRad3.Visible = False
                    ctrlRad4.Visible = False
                    ctrlRad5.Visible = False
                    ctrlRad6.Visible = False
                    ctrlRad7.Visible = False
                    ctrlRad8.Visible = False
                    ctrlRad9.Visible = False
                    ctrlRad10.Visible = False
                    ctrlRad11.Visible = False
                    ctrlRad12.Visible = False
                End If
                If _radQuerySelected.UniqueID = "rad2" Then
                    lblInstructions.Text = "Select an author from the list:"
                    ctrlRad1.Visible = False
```

```
        ctrlRad2.Visible = True
        ctrlRad3.Visible = False
        ctrlRad4.Visible = False
        ctrlRad5.Visible = False
        ctrlRad6.Visible = False
        ctrlRad7.Visible = False
        ctrlRad8.Visible = False
        ctrlRad9.Visible = False
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
    End If
    If _radQuerySelected.UniqueID = "rad3" Then
        lblInstructions.Text = String.Empty
        ctrlRad1.Visible = False
        ctrlRad2.Visible = False
        ctrlRad3.Visible = True
        ctrlRad4.Visible = False
        ctrlRad5.Visible = False
        ctrlRad6.Visible = False
        ctrlRad7.Visible = False
        ctrlRad8.Visible = False
        ctrlRad9.Visible = False
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
    End If
    If _radQuerySelected.UniqueID = "rad4" Then
        lblInstructions.Text = String.Empty
        ctrlRad1.Visible = False
        ctrlRad2.Visible = False
        ctrlRad3.Visible = False
        ctrlRad4.Visible = True
        ctrlRad5.Visible = False
        ctrlRad6.Visible = False
        ctrlRad7.Visible = False
        ctrlRad8.Visible = False
        ctrlRad9.Visible = False
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
    End If
    If _radQuerySelected.UniqueID = "rad5" Then
        lblInstructions.Text = String.Empty
        ctrlRad1.Visible = False
        ctrlRad2.Visible = False
        ctrlRad3.Visible = False
        ctrlRad4.Visible = False
        ctrlRad5.Visible = True
        ctrlRad6.Visible = False
        ctrlRad7.Visible = False
        ctrlRad8.Visible = False
        ctrlRad9.Visible = False
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
    End If
    If _radQuerySelected.UniqueID = "rad6" Then
```

```
        lblInstructions.Text = String.Empty
        ctrlRad1.Visible = False
        ctrlRad2.Visible = False
        ctrlRad3.Visible = False
        ctrlRad4.Visible = False
        ctrlRad5.Visible = False
        ctrlRad6.Visible = True
        ctrlRad7.Visible = False
        ctrlRad8.Visible = False
        ctrlRad9.Visible = False
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
    End If
    If _radQuerySelected.UniqueID = "rad7" Then
        lblInstructions.Text = "Select the year first, and then the month:"
        ctrlRad1.Visible = False
        ctrlRad2.Visible = False
        ctrlRad3.Visible = False
        ctrlRad4.Visible = False
        ctrlRad5.Visible = False
        ctrlRad6.Visible = False
        ctrlRad7.Visible = True
        ctrlRad8.Visible = False
        ctrlRad9.Visible = False
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
    End If
    If _radQuerySelected.UniqueID = "rad8" Then
        lblInstructions.Text = String.Empty
        ctrlRad1.Visible = False
        ctrlRad2.Visible = False
        ctrlRad3.Visible = False
        ctrlRad4.Visible = False
        ctrlRad5.Visible = False
        ctrlRad6.Visible = False
        ctrlRad7.Visible = False
        ctrlRad8.Visible = True
        ctrlRad9.Visible = False
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
    End If
    If _radQuerySelected.UniqueID = "rad9" Then
        lblInstructions.Text = "Select the year first, and then the month:"
        ctrlRad1.Visible = False
        ctrlRad2.Visible = False
        ctrlRad3.Visible = False
        ctrlRad4.Visible = False
        ctrlRad5.Visible = False
        ctrlRad6.Visible = False
        ctrlRad7.Visible = False
        ctrlRad8.Visible = False
        ctrlRad9.Visible = True
        ctrlRad10.Visible = False
        ctrlRad11.Visible = False
        ctrlRad12.Visible = False
```

```
            End If
            If _radQuerySelected.UniqueID = "rad10" Then
                lblInstructions.Text = "Select a year:"
                ctrlRad1.Visible = False
                ctrlRad2.Visible = False
                ctrlRad3.Visible = False
                ctrlRad4.Visible = False
                ctrlRad5.Visible = False
                ctrlRad6.Visible = False
                ctrlRad7.Visible = False
                ctrlRad8.Visible = False
                ctrlRad9.Visible = False
                ctrlRad10.Visible = True
                ctrlRad11.Visible = False
                ctrlRad12.Visible = False
            End If
            If _radQuerySelected.UniqueID = "rad11" Then
                lblInstructions.Text = String.Empty
                ctrlRad1.Visible = False
                ctrlRad2.Visible = False
                ctrlRad3.Visible = False
                ctrlRad4.Visible = False
                ctrlRad5.Visible = False
                ctrlRad6.Visible = False
                ctrlRad7.Visible = False
                ctrlRad8.Visible = False
                ctrlRad9.Visible = False
                ctrlRad10.Visible = False
                ctrlRad11.Visible = True
                ctrlRad12.Visible = False
            End If
            If _radQuerySelected.UniqueID = "rad12" Then
                lblInstructions.Text = String.Empty
                ctrlRad1.Visible = False
                ctrlRad2.Visible = False
                ctrlRad3.Visible = False
                ctrlRad4.Visible = False
                ctrlRad5.Visible = False
                ctrlRad6.Visible = False
                ctrlRad7.Visible = False
                ctrlRad8.Visible = False
                ctrlRad9.Visible = False
                ctrlRad10.Visible = False
                ctrlRad11.Visible = False
                ctrlRad12.Visible = True
            End If
        End If
    End If
End Sub
```

Running a query may not be the sole reason for an end user's visit to **default.aspx** of KDP Sales Reports. Before playing with the database, there must be some data, and that's where the **Upload Sales Report** (`btnUpload`) button comes in. The code discussed in the following pages

belongs to the **btnUpload_Click** method. The complete code, including the input validation, is available in default.aspx.vb in the GitHub repository.

Assuming that the tab delimited file is accepted by the application, the next step is to create a new file name for it because we want to save it to the server. The new name is based on a GUID to avoid overwriting files with the same name, and it will be stored in a Private variable that is declared at the top of the code file.

Before saving the file with its new name, the code specifies a path for both the output .txt and .xml files and checks for the existence of the directory where the .txt file will be saved. The FileUpload control (`fupReport`) has a convenient **SaveAs** method.

```
_newFileNameNoExt = GenerateFileName()
_txtPath = Server.MapPath("txt/" & _newFileNameNoExt & ".txt")
_xmlPath = Server.MapPath("xml/" & _newFileNameNoExt & ".xml")

If Not Directory.Exists(Server.MapPath("txt/")) Then
    Directory.CreateDirectory(Server.MapPath("txt"))
    fupReport.SaveAs(_txtPath)
Else
    fupReport.SaveAs(_txtPath)
End If
```

Now we need to read the text file and get rid of any blank lines that it may have.

```
Dim txtOld As String = String.Empty ' the original, multiline string
Dim txtNew As String() = Nothing ' a string array that will hold the lines of txtOld
Try
    ' It's important to set the Encoding to Default. Setting it to Unicode
    ' causes a NullReferenceException when filling the XDocument.
    txtOld = File.ReadAllText(_txtPath, Encoding.Default).Trim
    ' Add each line of txtOld to a string array.
    txtNew = txtOld.Split(CChar(Environment.NewLine)) Catch ex As Exception
    lblStatus.Text = ex.Message
    lblStatus.CssClass = "status_error"
    Exit Sub
End Try
```

It's finally time to transform the data into XML and save it as a .xml file. To create an XML file, first create an XDocument with the desired structure. An XDocument is a representation of an XML document.

```
' Note to C# programmers: Don't try this at home. XML literals are exclusive to VB.NET.

Dim xdoc As XDocument = <?xml version="1.0" encoding="utf-8"?>
                        <RoyaltyReport>

                        </RoyaltyReport>
```

Next, create an XElement called `<Sales>`, and use LINQ to populate each `<Sale>` within `<Sales>`.

```
Dim el As XElement = _
<Sales>
    <%= From records In txtNew _
        Let values = Split(records, ControlChars.Tab) _
        Select _
        <Sale>
            <RoyaltyDate><%= values(0) %></RoyaltyDate>
            <Title><%= HtmlEncode(values(1).Replace(Chr(34), String.Empty)) %></Title>
            <AuthorName><%= HtmlEncode(values(2)) %></AuthorName>
            <ASIN><%= values(3) %></ASIN>
            <Marketplace><%= values(4) %></Marketplace>
            <RoyaltyType><%= values(5) %></RoyaltyType>
            <TransactionType><%= values(6) %></TransactionType>
            <UnitsSold><%= values(7) %></UnitsSold>
            <UnitsRefunded><%= values(8) %></UnitsRefunded>
            <RefundRate><%= (Convert.ToDecimal(values(8)) /
Convert.ToDecimal(values(7))).ToString() %></RefundRate>
            <NetUnitsSold><%= values(9) %></NetUnitsSold>
            <AvgListPriceWithoutTax><%= values(10) %></AvgListPriceWithoutTax>
            <AvgFileSizeMB><%= values(11) %></AvgFileSizeMB>
            <AvgOfferPriceWithoutTax><%= values(12) %></AvgOfferPriceWithoutTax>
            <AvgDeliveryCost><%= values(13) %></AvgDeliveryCost>
            <Royalty><%= values(14) %></Royalty>
            <Currency><%= values(15) %></Currency>
        </Sale> _
    %>
</Sales>
```

Since we have a tab delimited text file, we need to split by the tab character. If we had a comma delimited (.csv) file, then we would split by comma, but that gets a little dicey when you're dealing with fields that contain one or more commas.

`<Title>` and `<AuthorName>` are HTML-encoded because they may contain illegal XML characters. They may also contain one or more commas, in which case they will be surrounded by quotation marks in the input text file. To get rid of them, we'll replace a double quote (") with an empty string. The ASCII character code for a double quote is 34. Since VB.NET requires double quotes to be placed around strings, escaping quotation marks can get confusing, so it's simpler and easier to use the appropriate character code in place of a double quote that occurs within the pair of outer double quotes that denotes a string.

<RefundRate> is a calculated value equal to UnitsRefunded divided by UnitsSold. As mentioned, RefundRate is not one of the column headings supplied by Amazon in their generated reports, so this had to be added at runtime.

We can now add <Sales> to the XDocument and save the XML file. Saving the file is actually optional, since we can do what we need to do while the data are being stored in the XDocument object.

```
' Add <Sales> to the XDocument.
xdoc.Root.Add(el)

' (Optional) Save the file.
If Not Directory.Exists(Server.MapPath("xml/")) Then
    Directory.CreateDirectory(Server.MapPath("xml"))
Else
    xdoc.Save(_xmlPath)
End If
```

If you save an XML file for viewing at a later date, you can load it programmatically as follows:

```
Dim xml As XDocument = XDocument.Load(pathToYourXmlFile)
```

The next step is to build a DataTable object and fill it with the XML data. This DataTable will be the data source for **SqlBulkCopy**.

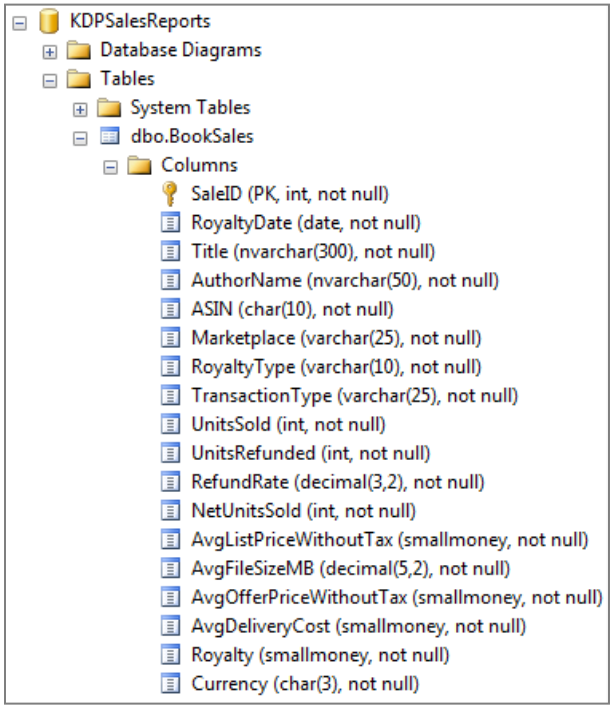
```
Dim sales = xdoc.Root.<Sales>(0)
Dim dt As New DataTable
If sales.Elements.Count > 0 Then
    dt.TableName = "BookSales" ' must match database table name
    ' The System data types must be compatible with the SQL data types.
    ' Here's a helpful guide for future reference:
    ' http://kambiz-na.blogspot.com/2009/09/mapping-sql-data-type-to-system-type.html
    dt.Columns.Add("RoyaltyDate", GetType(System.DateTime))
    dt.Columns.Add("Title", GetType(System.String))
    dt.Columns.Add("AuthorName", GetType(System.String))
    dt.Columns.Add("ASIN", GetType(System.String))
    dt.Columns.Add("Marketplace", GetType(System.String))
    dt.Columns.Add("RoyaltyType", GetType(System.String))
    dt.Columns.Add("TransactionType", GetType(System.String))
    dt.Columns.Add("UnitsSold", GetType(System.Int32))
    dt.Columns.Add("UnitsRefunded", GetType(System.Int32))
    dt.Columns.Add("RefundRate", GetType(System.Decimal))
    dt.Columns.Add("NetUnitsSold", GetType(System.Int32))
    dt.Columns.Add("AvgListPriceWithoutTax", GetType(System.Decimal))
    dt.Columns.Add("AvgFileSizeMB", GetType(System.Decimal))
    dt.Columns.Add("AvgOfferPriceWithoutTax", GetType(System.Decimal))
```

```
    dt.Columns.Add("AvgDeliveryCost", GetType(System.Decimal))
    dt.Columns.Add("Royalty", GetType(System.Decimal))
    dt.Columns.Add("Currency", GetType(System.String))

    ' Get the data for each <Sale> in <Sales>. (XML literals are cool, aren't they?)
    For i As Integer = 0 To sales.Elements.Count - 1
        Dim rDate As DateTime = Convert.ToDateTime(sales.<Sale>.<RoyaltyDate>(i).Value)
        Dim rDateStr As String = rDate.ToString("yyyy-MM-dd") ' SQL date format
        ' Don't forget to HtmlDecode anything that was encoded earlier.
        Dim title As String = HtmlDecode(sales.<Sale>.<Title>(i).Value)
        Dim auth As String = HtmlDecode(sales.<Sale>.<AuthorName>(i).Value)
        Dim asin As String = sales.<Sale>.<ASIN>(i).Value
        Dim mark As String = sales.<Sale>.<Marketplace>(i).Value
        Dim rType As String = sales.<Sale>.<RoyaltyType>(i).Value
        Dim tType As String = sales.<Sale>.<TransactionType>(i).Value
        Dim sold As String = Int32.Parse(sales.<Sale>.<UnitsSold>(i).Value)
        Dim refd As String = Int32.Parse(sales.<Sale>.<UnitsRefunded>(i).Value)
        Dim rate As String = Decimal.Parse(sales.<Sale>.<RefundRate>(i).Value)
        Dim nus As String = Int32.Parse(sales.<Sale>.<NetUnitsSold>(i).Value)
        Dim price As String =
Convert.ToDecimal(sales.<Sale>.<AvgListPriceWithoutTax>(i).Value)
        Dim size As String = Convert.ToDecimal(sales.<Sale>.<AvgFileSizeMB>(i).Value)
        Dim offer As String =
Convert.ToDecimal(sales.<Sale>.<AvgOfferPriceWithoutTax>(i).Value)
        Dim dev As String = Convert.ToDecimal(sales.<Sale>.<AvgDeliveryCost>(i).Value)
        Dim roy As String = Convert.ToDecimal(sales.<Sale>.<Royalty>(i).Value)
        Dim cur As String = sales.<Sale>.<Currency>(i).Value

        ' Create a new DataRow and populate it.
        Dim row As DataRow = dt.NewRow()
        row("RoyaltyDate") = rDateStr
        row("Title") = title
        row("AuthorName") = auth
        row("ASIN") = asin
        row("Marketplace") = mark
        row("RoyaltyType") = rType
        row("TransactionType") = tType
        row("UnitsSold") = sold
        row("UnitsRefunded") = refd
        row("RefundRate") = rate
        row("NetUnitsSold") = nus
        row("AvgListPriceWithoutTax") = price
        row("AvgFileSizeMB") = size
        row("AvgOfferPriceWithoutTax") = offer
        row("AvgDeliveryCost") = dev
        row("Royalty") = roy
        row("Currency") = cur

        ' Add the DataRow to the DataTable
        dt.Rows.Add(row)
    Next
End If
```

Importing the data into SQL Server with SqlBulkCopy is very simple, but there are things that can go wrong. The Microsoft documentation for the SqlBulkCopy.ColumnMappings property

states, "If the data source and the destination table have the same number of columns, and the ordinal position of each source column within the data source matches the ordinal position of the corresponding destination column, the ColumnMappings collection is unnecessary."

Sometimes official documentation doesn't represent reality. Although we have a one-to-one correspondence between the columns in the DataTable object and the columns in the database table (as shown below in SQL Server Management Studio), it's a good idea to use ColumnMappings anyway. Incidentally, the primary key column doesn't count as a column for the purposes of SqlBulkCopy.



Failure to use ColumnMappings often results in bewildering error messages about data types, despite the data types being correct. For example, my not using ColumnMappings the first time around resulted in the following error: "The given value of type String from the data source cannot be converted to type date of the specified target column."

This was related to the RoyaltyDate column, which is a "DateTime" in the DataTable and a "date" in the database, but the input was a string. Normally, there's no problem sending a date string to SQL Server because it implicitly converts it to a date, as long as the string is in the date format that the database uses (yyyy-MM-dd, in this case).

Simply adding the ColumnMappings without making any changes to the actual data in terms of type or formatting allowed the SqlBulkCopy operation to go without a hitch.

```
Try
    Using cnn As New SqlConnection(DbConnectionString)
        cnn.Open()
        Using bulkCopy As New SqlBulkCopy(cnn)
            ' Import the data
            bulkCopy.DestinationTableName = dt.TableName
            ' (SourceColumn, DestinationColumn)
            bulkCopy.ColumnMappings.Add("RoyaltyDate", "RoyaltyDate")
            bulkCopy.ColumnMappings.Add("Title", "Title")
            bulkCopy.ColumnMappings.Add("AuthorName", "AuthorName")
            bulkCopy.ColumnMappings.Add("ASIN", "ASIN")
            bulkCopy.ColumnMappings.Add("Marketplace", "Marketplace")
            bulkCopy.ColumnMappings.Add("RoyaltyType", "RoyaltyType")
            bulkCopy.ColumnMappings.Add("TransactionType", "TransactionType")
            bulkCopy.ColumnMappings.Add("UnitsSold", "UnitsSold")
            bulkCopy.ColumnMappings.Add("UnitsRefunded", "UnitsRefunded")
            bulkCopy.ColumnMappings.Add("RefundRate", "RefundRate")
            bulkCopy.ColumnMappings.Add("NetUnitsSold", "NetUnitsSold")
            bulkCopy.ColumnMappings.Add("AvgListPriceWithoutTax",
"AvgListPriceWithoutTax")
            bulkCopy.ColumnMappings.Add("AvgFileSizeMB", "AvgFileSizeMB")
            bulkCopy.ColumnMappings.Add("AvgOfferPriceWithoutTax",
"AvgOfferPriceWithoutTax")
            bulkCopy.ColumnMappings.Add("AvgDeliveryCost", "AvgDeliveryCost")
            bulkCopy.ColumnMappings.Add("Royalty", "Royalty")
            bulkCopy.ColumnMappings.Add("Currency", "Currency")
            bulkCopy.WriteToServer(dt)
        End Using
        ' Remove duplicate records
        Dim cmd As New SqlCommand
        cmd.Connection = cnn
        cmd.CommandType = CommandType.StoredProcedure
        cmd.CommandText = CN_RemoveDuplicates
        cmd.ExecuteNonQuery()
    End Using
    lblStatus.Text = "Your file was successfully uploaded and imported into the
database."
    lblStatus.CssClass = "status_ok"
Catch ex As Exception
    lblStatus.Text = ex.Message
    lblStatus.CssClass = "status_error"
End Try
```

It's bad form to allow duplicate rows in a database, so the application removes duplicates after every import. Dupes may occur when the royalty reports have overlapping dates.

Here's the SQL stored procedure that removes duplicate rows using a common table expression (CTE):

```
USE [KDPSalesReports]
GO
/****** Object:  StoredProcedure [dbo].[procRemoveDuplicates]    Script Date:
01/22/2016 07:55:49 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[procRemoveDuplicates]
AS
WITH BookSales_CTE
AS
(
      SELECT *, ROW_NUMBER() OVER (PARTITION BY RoyaltyDate, Title,
AuthorName ORDER BY RoyaltyDate) AS NumRecords
      FROM BookSales
)
DELETE FROM BookSales_CTE WHERE NumRecords > 1;
```

Upon finishing and debugging the code for `btnUpload_Click`, I tried to import the dummy data set, which had about 6600 rows in the tab delimited text file. On my local machine (see specs below), it took about a minute and a half for SqlBulkCopy to do its thing. Not bad, considering that the average royalty report for an individual author (or the royalty report for an "average" individual author) will contain a lot fewer than 6k rows.

Development Environment:

- Windows 7 Ultimate x64 SP1
- Visual Studio 2010 Professional
- SQL Server 2008 SP1 (Standard Edition)
- .NET Framework 4.0 and IIS 7.0
- AMD Phenom II x4, 3.00 GHz
- 8 GB RAM

Summary

KDP Sales Reports offers a simple way for authors and publishers to store and analyze their generated royalty reports from Amazon, regardless of whether or not they have Microsoft Excel installed on their computers. Even if they do, having the ability to query a database on-demand is more appealing than figuring out how to get the same results from a spreadsheet.

One of the application's shortcomings, however, is that some of the result sets are rather large, so reading the data requires a lot of scrolling. A future version of KDP Sales Reports could stand to have paging functionality added to the Repeater control. Alternatively, the Repeater could be replaced with another control that has built-in support for paging.

It might also be worth exploring how to automate the process of generating and importing the royalty data on a scheduled basis. On the other hand, prosperous Kindle book authors don't need to be prodded to check their stats every month; it becomes a daily addiction.