Chapter 22 – Printing From Within VB

# Objectives:

* Using the PrintForm Control
* Using the Printing Common Dialogs
* Creating an Application That Prints
* Introduction to the Report Designer

This chapter provides an overview of printing from within VB.NET. The various print dialog controls will be examined, culminating in an application that shows how to use them. The chapter concludes with an introduction to using Microsoft’s RDLC Report Designer and Reporting control.

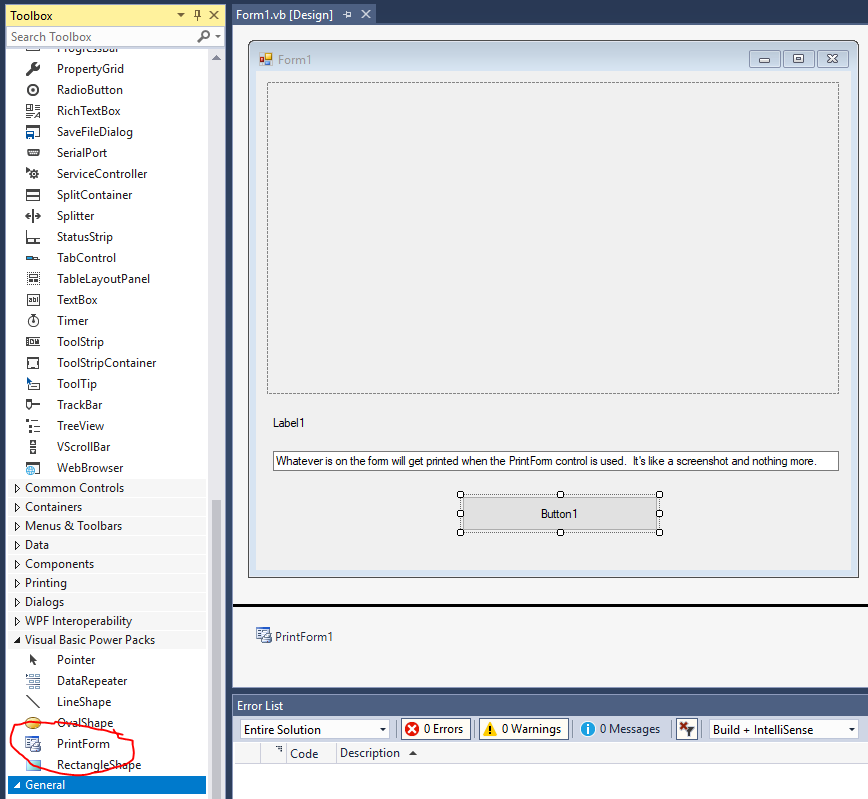
As commonplace as the need for printing is, you might think it will be a breeze. Unfortunately, as we will see, that is not at all the case. Even with what’s built-in, very little provides much functionality without work on our part. Wiring up an application to print anything other than a simple screenshot is going to involve considerable effort.

# Using the PrintForm Control

This is the weakest of the printing techniques that we will look at in this chapter. The PrintForm Control is a pre.-NET throwback. Essentially all that this control does is send a snapshot of a form to a printer. Nothing fancy, nothing exciting, but it’s quick and easy.

This control does not come with Visual Studio directly. You add it to Visual Studio when you install the Visual Basic Power Packs. We discussed downloading and installing those controls back in Chapter 10 when we introduced the ideas of components and working with the items that are in the toolbox. You did install the Power Packs, right? If not, you’ll need to do so to use the PrintForm control.

Here’s a sample application that has a PictureBox, Label, TextBox and a Command button place on it. All the default names for the controls were used. A PrintForm control was dragged and dropped on the form as well. The control shows up in the invisible controls tray below the form. The PrintForm control can be found in the Visual Basic Power Packs section of the toolbox once we install and add those controls:



Here’s the code behind the scenes:

'Chapter 22 - Program 1

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

'Load up the image from the Debug folder and set the label

PictureBox1.ImageLocation = Environment.CurrentDirectory &

"\At Smithy's HDR.jpg"

PictureBox1.SizeMode = PictureBoxSizeMode.Zoom

PictureBox1.Load()

Label1.Text = "At Smithy's (HDR)"

End Sub

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'The next line allows us to set where we want to print (printer,

'file or preview)

'If you don't set anything, the printer is used by default

PrintForm1.PrintAction = Printing.PrintAction.PrintToPreview

'This actually prints

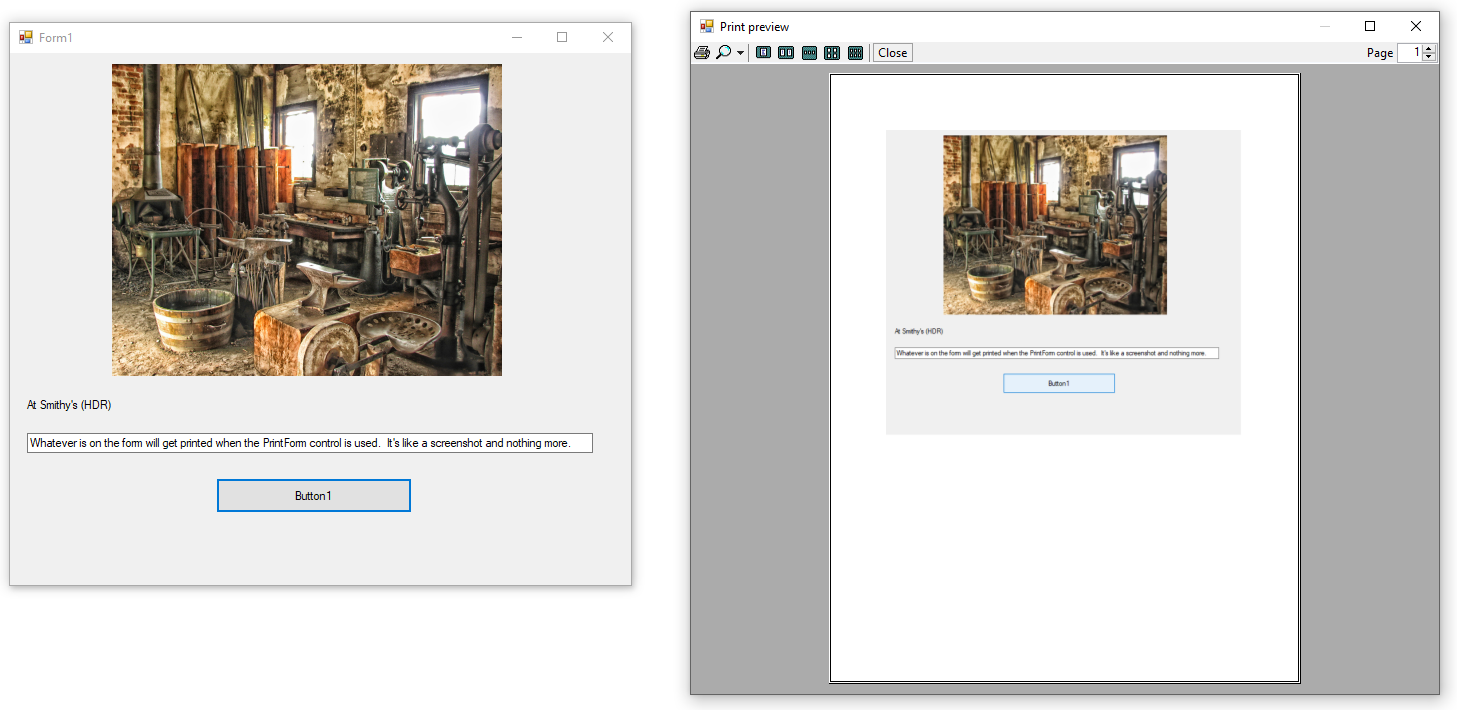
PrintForm1.Print()

End Sub

End Class

Besides printing the form, about the only other functionality you really have is deciding where you want your form to be sent: to a file, the printer or a print preview. We set our application up to send things to the print preview. So after clicking the Command button, the standard system Print Preview dialog appears and you can zoom in/out, close or print the form. There’s not much else you can do on that dialog since the form is only one page long…

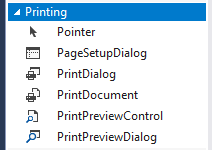
When you run the program and click the Command button, here’s the application’s form and the Print Preview dialog that appears:



# Using the Printing Common Dialogs

The last application’s PrintForm control has the ability to show a Print Preview dialog. This is one of the common dialogs that are available to us. We didn’t discuss the printing common dialogs when we initially examined the OpenFile, SaveFile, Font and Color dialogs because I wanted to look at all the printing pieces at once.

As you may have noticed, there are several printing related dialog boxes that are available under the Printing toolbox section in VB:

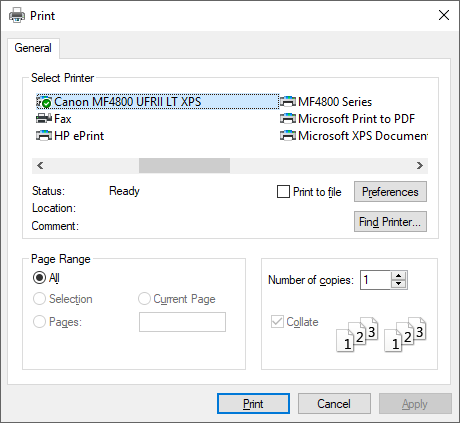


Much of the drudgery work for handling the setup of printing can be eliminated through the use of these controls. Notice that I said setup work – that’s all these dialogs do. You must write the code to perform the printing as it’s not built in.

This section of the chapter will introduce each dialog box and its related properties.

## PrintDialog

The PrintDialog control allows the user to select the printer to print to, the number of copies of a print job to produce and the specification of which pages of a print job are to be printed:

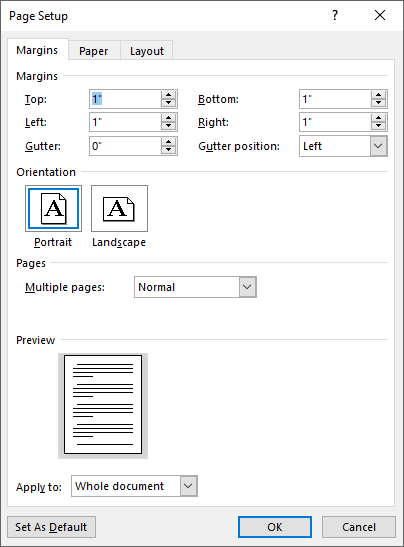


Here are the commonly accessed properties in PrintDialog box:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| AllowCurrentPage | Gets or sets a value indicating whether the current page option button is shown |
| AllowPrintToFile | Gets or sets a Boolean value that specifies whether the Print to File checkbox is shown |
| AllowSelection | Gets or sets a Boolean value that specifies whether the dialog shows the ability to print user specific pages |
| AllowSomePages | Gets or sets a Boolean value that specifies whether the Pages option textboxes are shown |
| Document | Gets or sets a value that specifies the PrintDocument object that the dialog box used to obtain PrinterSettings |
| PrinterSettings | Gets or sets the PrinterSettings object that the dialog box modifies |
| PrintToFile | Gets or sets a Boolean value specifying whether or not the Print to File box is checked |

## PageSetupDialog

The PageSetupDialog control is responsible for allowing the user to select the size and the source of the paper used in printing, the orientation of the printing and the margins to leave on the page. This dialog cannot be viewed until after a PageSettings object has been created:



Here are the more commonly used properties on the PageSetupDialog box:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| AllowMargins | Gets or sets a Boolean value that specifies whether the margins section of the dialog box is enabled |
| AllowOrientation | Gets or sets a Boolean value that specifies whether orientation section of the dialog box is shown |
| AllowPaper | Gets or sets a Boolean value that specifies whether the paper section of the dialog box is shown |
| AllowPrinter | Gets or sets a Boolean value that specifies whether the printer button is shown |
| Document | Gets or sets a value that specifies the PrintDocument object from which the dialog gets the page settings |
| EnableMetric | Gets or sets whether English or Metric margin settings should be displayed |
| MinMargins | Gets or sets a value that specifies the minimum margins (in hundredths of an inch) that the user can select |
| PageSettings | Gets or sets a value that specifies the page settings object that the dialog will modify |
| PrinterSettings | Gets or sets the PrinterSettings object that the dialog modifies if the user clicks the Printer button |

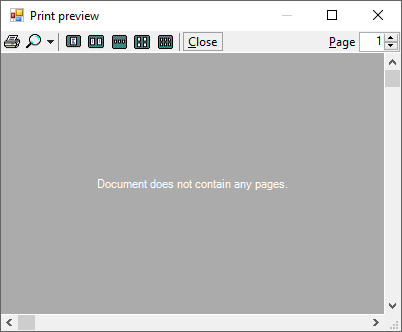
Again, at a minimum, you must set up a PageSettings object prior to trying to show the PageSetupDialog, as in:

PageSetupDialog1.PageSettings = New System.Drawing.Printing.PageSettings

PageSetupDialog1.ShowDialog()

## PrintPreviewDialog

The PrintPreviewDialog control allows the user to preview how a document's pages will appear when they are printed:



This control has more functionality than some of the other printing related common dialogs. It will allow a user to zoom in/out on a document, see one or multiple pages simultaneously, jump to a particular page in the document and “send” the job to a printer (you still have to write the code for this).

Here are the more commonly used properties on the PrintPreviewDialog box:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| AutoZoom | Gets or sets a value indicating whether resizing the control or changing the number of pages shown will automatically adjust the Zoom property |
| Columns | Gets or sets the number of pages displayed horizontally across the screen |
| Document | Gets or sets a value indicating the document to preview |
| Rows | Gets or sets the number of pages displayed vertically down the screen |
| StartPage | Gets or sets the page number of the upper left hand page |
| UseAntiAlias | Gets or sets a value indicating whether printing uses the anti-aliasing features of the operating system |
| Zoom | Gets or sets a value indicating how large the pages will appear |

There is also a PrintPreviewControl that is wrapped up in the PrintPreviewDialog that we can add to our application natively if we desire. The PrintPreviewControl is identical to the PrintPreviewDialog’s main canvas area minus the button menu. There is no discussion of this control in this chapter – in practice, most developers use the full PrintPreviewDialog if they wish to provide print preview capabilities.

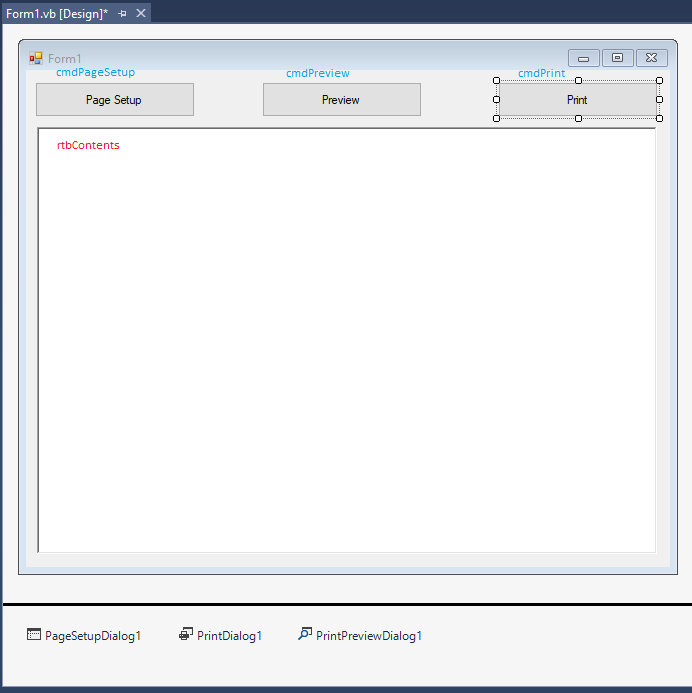
# Creating an Application that Prints

Printing in .NET… How can I put this succinctly? Sucks! Yeah, that’s pretty accurate in my opinion. We are so dependent on the capability of printing things out that you would think this facility would be built in and easy-peasy. After all, look at how many other complex things we’ve performed that have required so little effort. Printing is quite a complex process and it will clearly involve many of the items that we discussed including graphics contexts, common dialog controls and some more pieces of the graphics namespace we haven’t looked at before. If we try to boil printing down into a series of steps, this is what that list would look like:

1. We have to get an instance of the PrintDocument class (either drag and drop it from the toolbox or add it through code).
2. Set the PrintDocument up to use the printer settings that we are interested in. We can get those printer settings through a PrintDialog or set them through code.
3. For every page that is to be printed, the PrintDocument’s PrintPage event will trigger. We will receive a System.Drawing.Graphics graphics context object for the printer canvas and your handler code will print out the single page; there is a flag indicating if there will be more pages to print after the current page is printed. We can only “handle” one printed page at a time.
4. We will use the Print method in PrintDocument to actually start the work up.

As you can see, you have a lot of work to do. This is why printing is one of the last chapters covered in the class. You need to understand so many other things before you can worry about trying to print. The printing common dialogs only get us some basic information about the printer and what not – we have to write a bunch of code to actually take the contents of a document and send it to the printer. That requires that we have an understanding of what a graphics context is because we will write the contents of the document onto the printer’s graphics context, which will then be written to physical paper.

Let's write a simple Notepad type program that utilizes the PrintDialog, PageSetupDialog and PrintPreviewDialogs (the default names were used). You will also need to add three Command buttons for Page Setup (cmdPageSetup), Print Preview (cmdPrintPreview) and Print (cmdPrint). Finally, add a RichTextBox control (rtbContents) to the form – this will be where the user can type in the document that they want printed. Finally, let's write it so that the program really prints – the dialogs aren’t going to handle any of that for us. Here’s the form:



Here's the code behind the application:

'Chapter 22 - Program 2

'Notice we are bringing in some of the Drawing namespace here.

'It's not 2D, but many of the concepts are the same in order

'to get our document to print.

Imports System.Drawing.Printing

Public Class Form1

'Define a print document and a printer settings object for use

'in the various dialog boxes. We have to declare any object of

'type PrintDocument WithEvents since this object needs to be able

'to fire its PrintPage event. We will respond to this event by

'writing the code to print the page out.

Private WithEvents myPrintDocument As New

System.Drawing.Printing.PrintDocument()

Private CurrentPrinterSettings As New

System.Drawing.Printing.PrinterSettings()

Private Sub cmdPageSetup\_Click(sender As Object, e As EventArgs)

Handles cmdPageSetup.Click

'Remember that we have to set the PageSetupDialog's Document

'property to a valid PrintDocument before trying to display it.

'We will set the document to our print document.

PageSetupDialog1.Document = myPrintDocument

PageSetupDialog1.ShowDialog()

End Sub

Private Sub cmdPreview\_Click(sender As Object, e As EventArgs)

Handles cmdPreview.Click

'Remember that we have to set the PrintPreviewDialog's Document

'property to a valid PrintDocument before trying to display it.

'We will set the document to our print document.

PrintPreviewDialog1.Document = myPrintDocument

PrintPreviewDialog1.ShowDialog()

End Sub

Private Sub cmdPrint\_Click(sender As Object, e As EventArgs)

Handles cmdPrint.Click

'Remember that we have to set the PrintDialog's Document

'property to a valid PrintDocument before trying to display it.

'We will set the document to our print document. We also need

'to have some place to store the PrinterSettings associated

'with how the current document will be printed on the currently

'selected printer.

PrintDialog1.Document = myPrintDocument

PrintDialog1.PrinterSettings = CurrentPrinterSettings

'If the user presses the OK button on the PrintDialog page,

'then we will tell the Document to print. When the document

'executes the print method, it will raise the PrintPage event

'which we will need to handle.

If (PrintDialog1.ShowDialog() = DialogResult.OK) Then

myPrintDocument.Print()

End If

End Sub

Private Sub myPrintDocument\_PrintPage(sender As Object, e As

PrintPageEventArgs) Handles myPrintDocument.PrintPage

'Here's where we have to write the code that tells how to

'print the current document. I know you probably hoped that

'VB magically knew how to do this, but you can't have

'everything done for you...

'We can use the DrawString to print the RichTextBox's contents

'out to the printer's graphics context.

e.Graphics.DrawString(rtbContents.Text, rtbContents.Font,

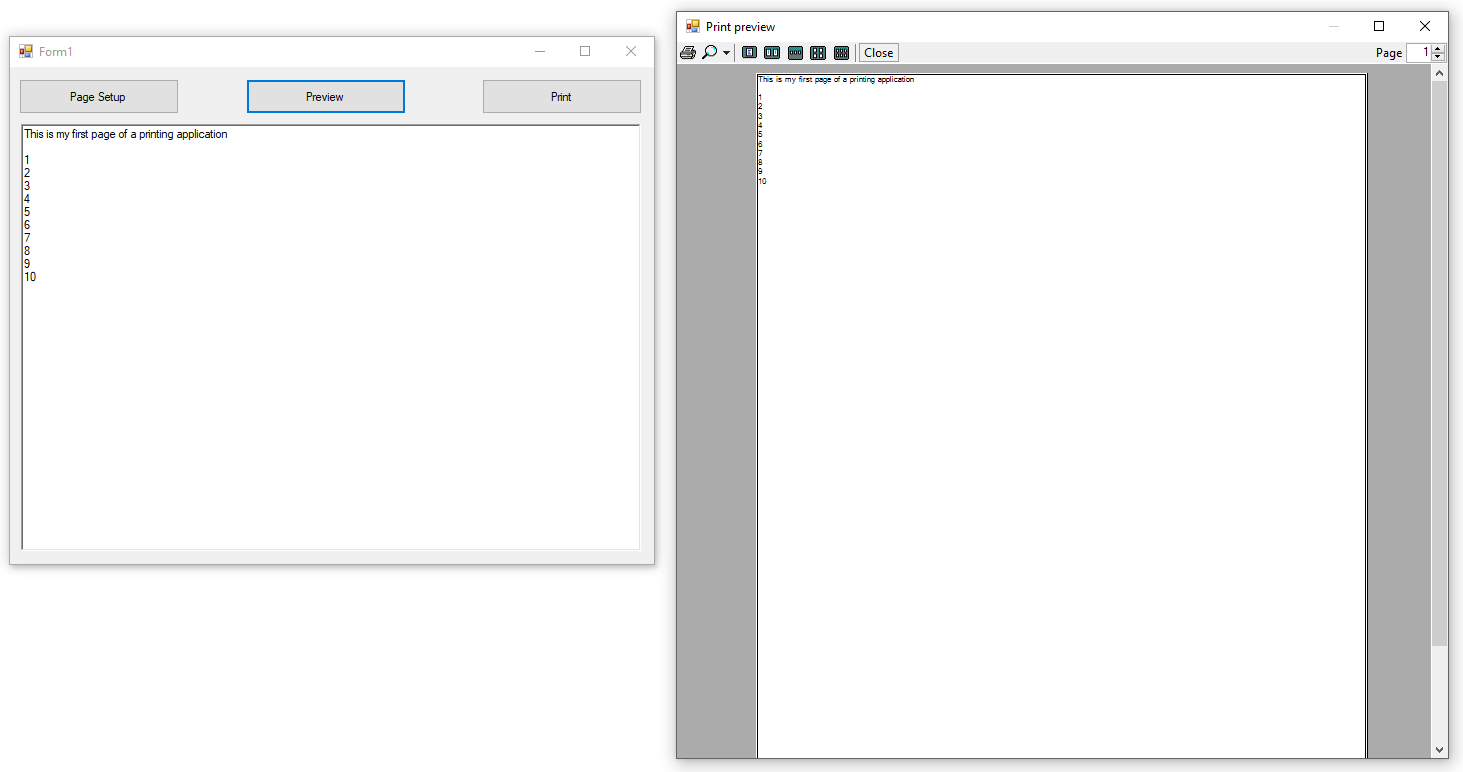
Brushes.Black, 0, 0)

End Sub

End Class

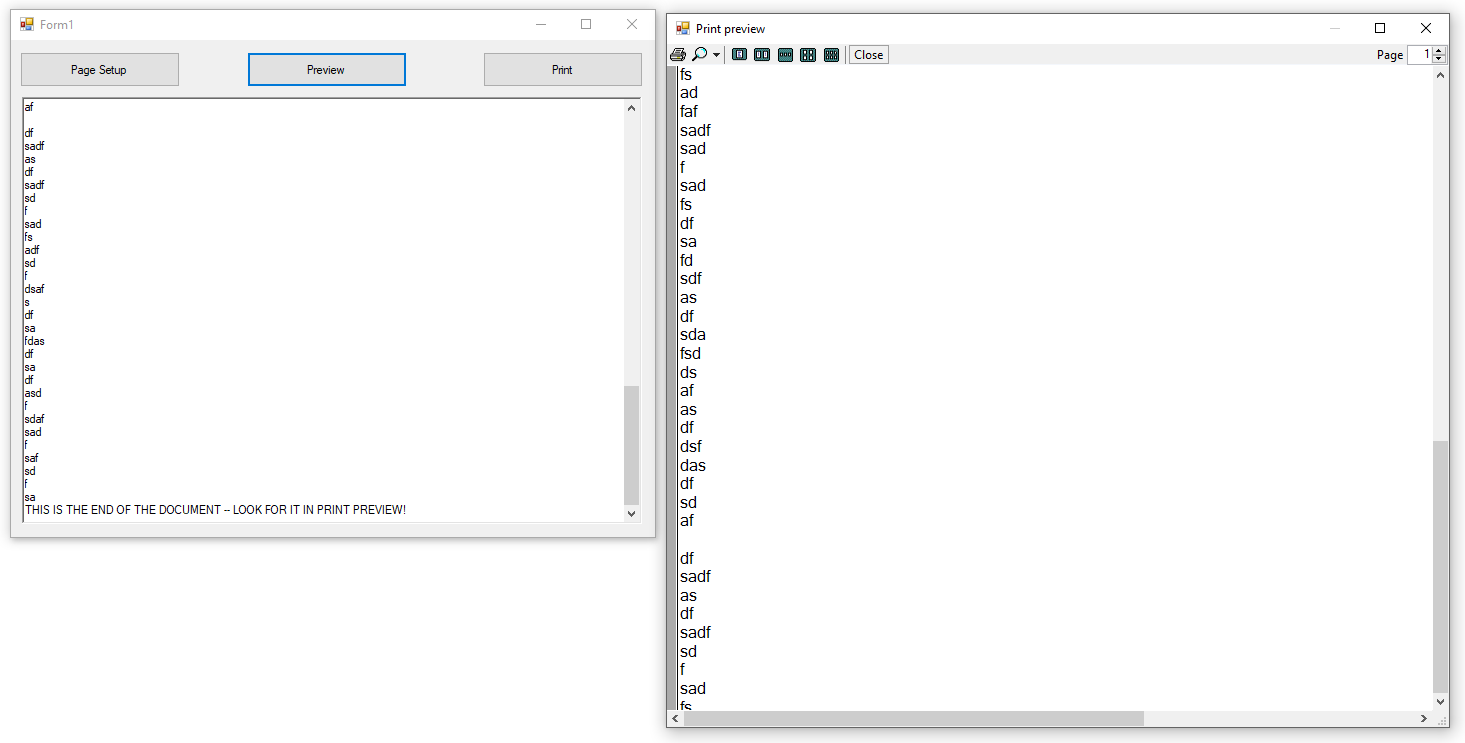
You probably noticed that the toughest part of the program was actually writing the code that designates how to print the document out. VB will assist you in the printing dialogs, getting your user's choices about how to print and so forth, but it still comes down to you having to handle the printing chore. Still, even the example that we looked at wasn't too bad to write.

Here's a screenshot of the application running with its PrintPreview dialog open:



If you type in some text into the RichTextBox and press the Print button, pick a printer and press Print, your document does come out. You’re probably wondering why I made printing out to be so bad, if all we had to do was write one lousy DrawString command in the PrintPage method. Hang on, it’s coming…

Now, create a document that’s more than one page long. Try printing and you’ll notice that only the first page worth comes out (ditto for the previewing). I ran the application and typed in a couple of hundred lines – and I put a special note on the last one. Notice that you don’t get to see it (nor will you) in either the PrintPreview or when you print the document:



This is where the printing process becomes kind of nasty – you are responsible for counting what you’ve printed and determining whether there is more to print. If so, **you** have to notify the system. Let’s take a look at a modified version of the program above which correctly handles multiple pages:

'Chapter 22 - Program 3

Imports System.Drawing.Printing

Public Class Form1

Private WithEvents myPrintDocument As New

System.Drawing.Printing.PrintDocument()

Private CurrentPrinterSettings As New

System.Drawing.Printing.PrinterSettings()

'This is where we will copy the RTB's contents to when we need

'to print or to printpreview

Dim docText As String

Private Sub cmdPageSetup\_Click(sender As Object, e As EventArgs)

Handles cmdPageSetup.Click

PageSetupDialog1.Document = myPrintDocument

PageSetupDialog1.ShowDialog()

End Sub

Private Sub cmdPreview\_Click(sender As Object, e As EventArgs)

Handles cmdPreview.Click

'Going to start a preview, so grab the latest copy

'of the RTB's contents

docText = rtbContents.Text

PrintPreviewDialog1.Document = myPrintDocument

PrintPreviewDialog1.ShowDialog()

End Sub

Private Sub cmdPrint\_Click(sender As Object, e As EventArgs)

Handles cmdPrint.Click

'Going to start a print, so grab the latest copy

'of the RTB's contents

docText = rtbContents.Text

PrintDialog1.Document = myPrintDocument

PrintDialog1.PrinterSettings = CurrentPrinterSettings

If (PrintDialog1.ShowDialog() = DialogResult.OK) Then

myPrintDocument.Print()

End If

End Sub

Private Sub myPrintDocument\_PrintPage(sender As Object, e As

PrintPageEventArgs) Handles myPrintDocument.PrintPage

'We need to know how many characters and lines can fit on a page.

'Essentially we have to do some math and to count...

Dim intCharactersOnPage As Integer = 0

Dim intLinesPerPage As Integer = 0

' Sets the value of charactersOnPage to the number of characters

' of stringToPrint that will fit within the bounds of the page.

e.Graphics.MeasureString(docText, Me.Font, e.MarginBounds.Size,

StringFormat.GenericTypographic, intCharactersOnPage, intLinesPerPage)

' Draws the string within the bounds of the page

e.Graphics.DrawString(docText, Me.Font, Brushes.Black,

e.MarginBounds, StringFormat.GenericTypographic)

' Remove the portion of the string that has been printed.

docText = docText.Substring(intCharactersOnPage)

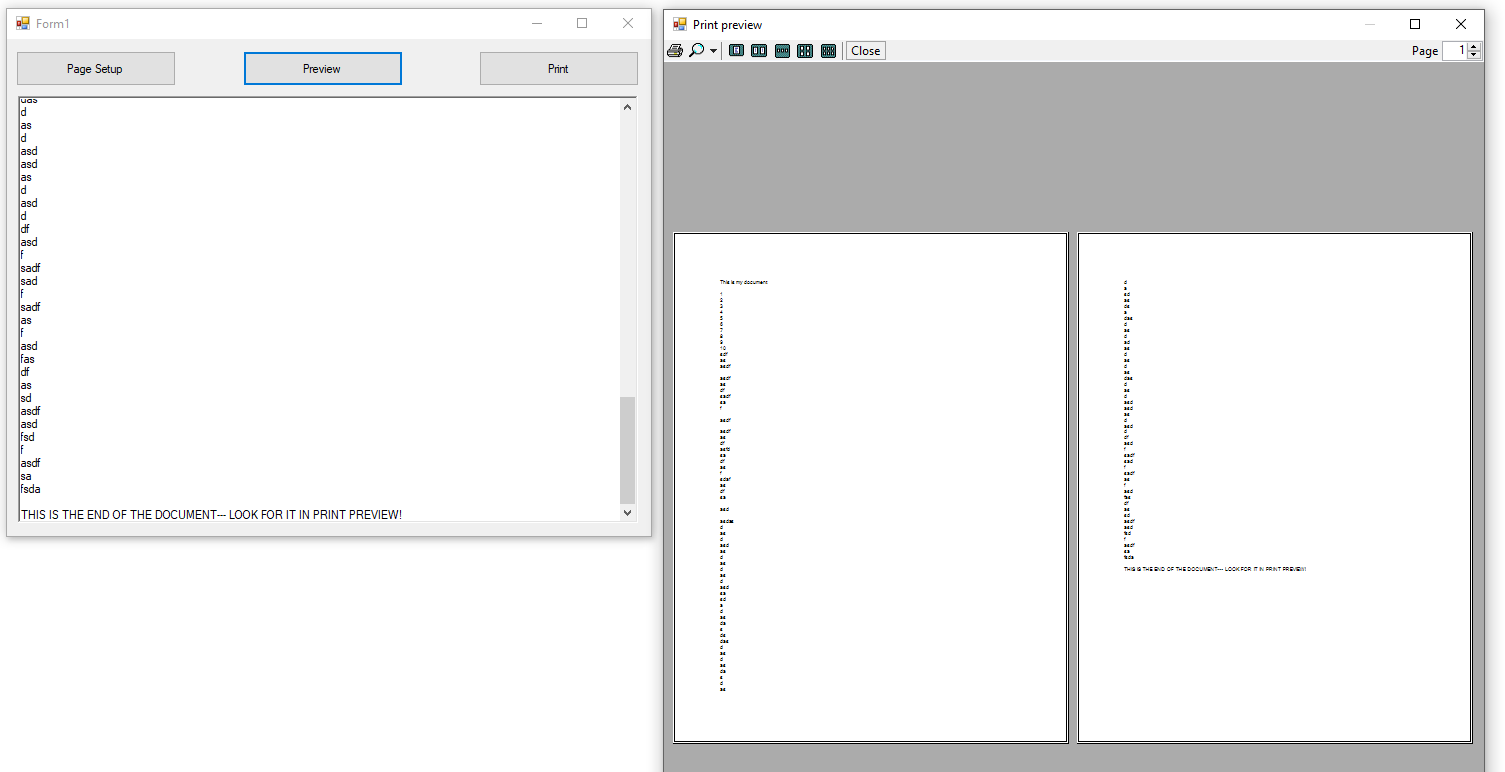
' Check to see if more pages are to be printed.

e.HasMorePages = docText.Length > 0

End Sub

End Class

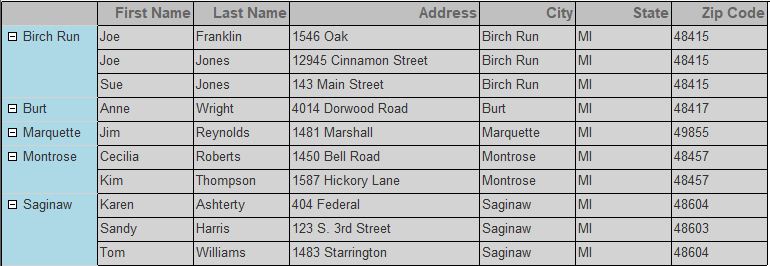
Here is a screenshot showing that PrintPreview has two pages in it:



If we try to print this document out, we now get both pages. From this little introduction, you should be able to see why so many people avoid building printing in from scratch in their applications: it’s tough. We didn’t consider different fonts, margins, graphics, any of that…it’s a complicated problem to solve.

# Introduction to the Report Designer

How would you like to be able to do reporting on databases that look like this?

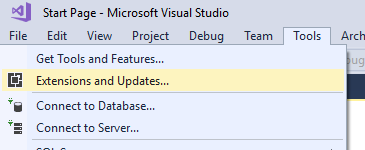


After having just looked at the amount of work to just print the contents of a RichTextBox control out, you're probably not convinced that you want to spend the effort to do this. Fortunately, you don't have to. VB has access to a report designer and a report viewer control that give you a bunch of functionality. They were designed with the express purpose of taking the drudgery out of creating and writing complex reports, usually the sort that you would run off from databases. The downside is that Microsoft no longer directly bundles this software with Visual Studio. We are going to need to add some additional pieces in so that we can get this functionality.

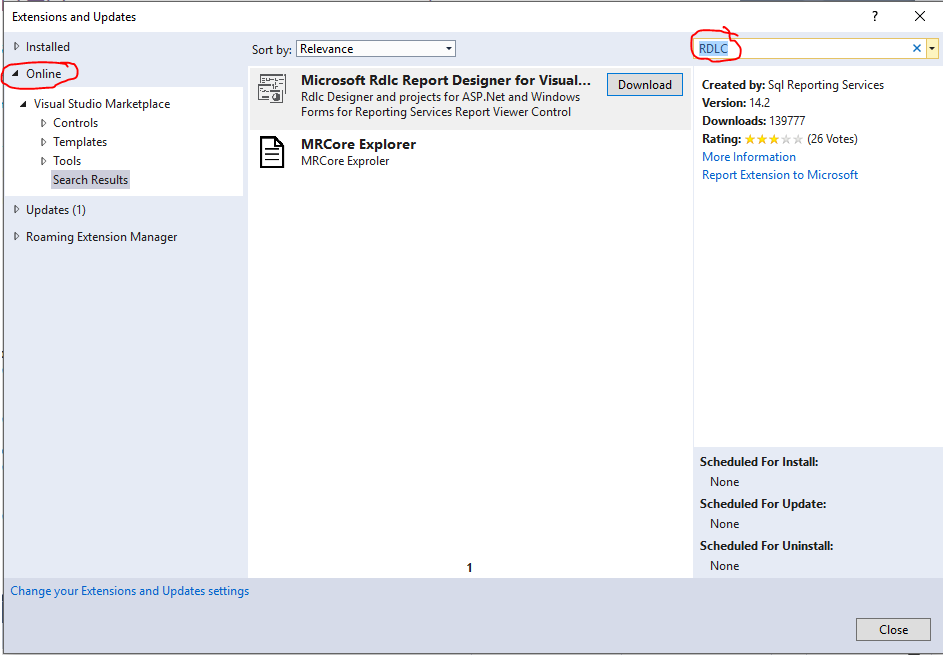
## Installing the Report Designer

I can’t answer why Microsoft takes away things that used to be installed natively with Visual Studio. Furthermore, they aren’t even consistent with where you need to go to access this additional functionality. Understand that these instructions work for the current version of Visual Studio, but it doesn’t mean things will stay this way in the next version…

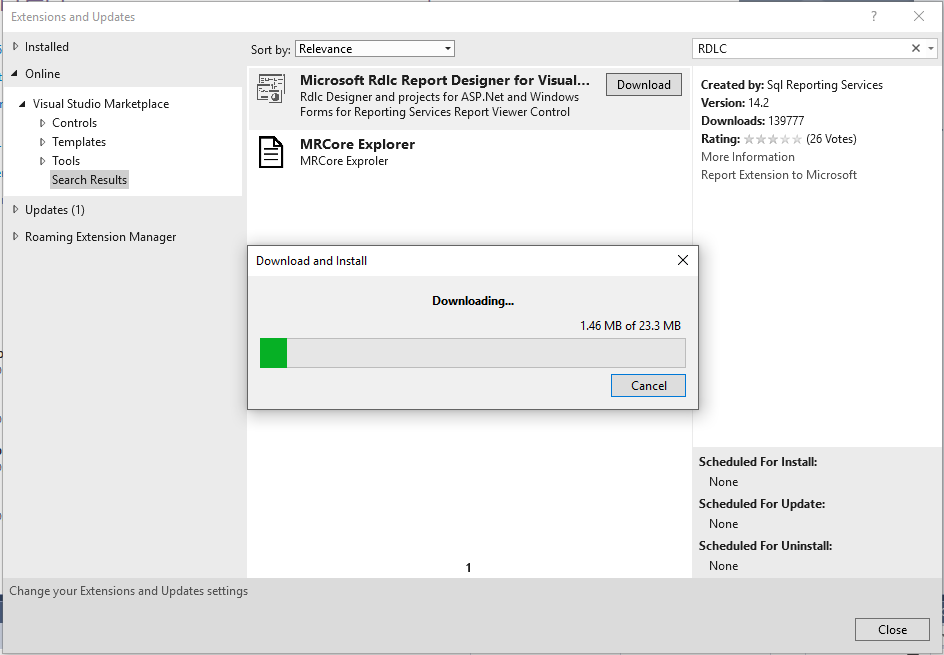
Start up Visual Studio but don’t open or create any new projects. Instead click on Tools🡪Extensions and Updates:



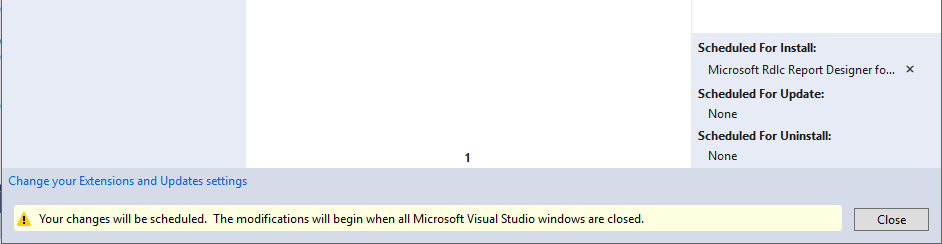
From the dialog that appears, click on the Online tab in the left pane (I know the screenshot below shows something different, but that’s due to the search results) and type RDLC in the right pane’s search box. After a couple of seconds, you should see some results in the middle pane:



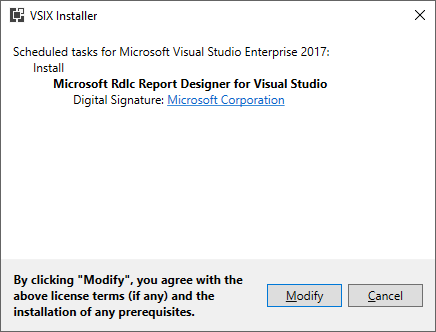
The Microsoft Rdlc Report Designer for Visual Studio is what we are looking for. You’ll notice that there is a Download button next to the item in the middle pane. Go ahead and click on it:



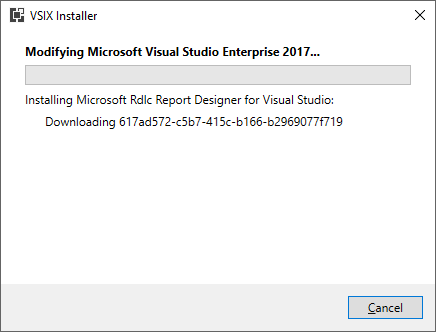
You’ll see the progress as the extension is being downloaded. Once it is complete, the progress bar will disappear and you’ll see a message displayed on the bottom of the Extensions and Updates manager:



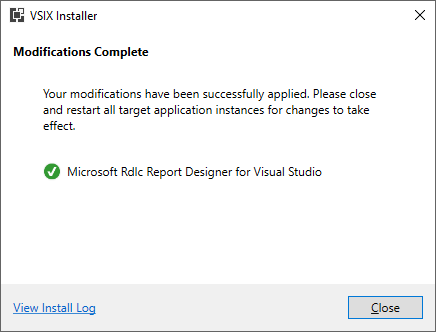
You can click Close and shut down Visual Studio. Once Visual Studio has closed, the Extensions manager will start the installation process of the RDLC extension. You’ll be prompted if you want to Modify your Visual Studio (meaning install the extension) or Cancel the installation. Choose Modify:



It’s all automated, so you just can sit back and wait for it to complete:



Everything has been installed when you see this screen:

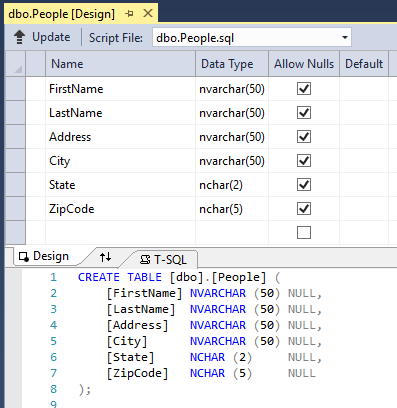


Click close and you’re done with this piece. The RDLC Report Designer only needs to be installed once and it has included a Reporting Project as well as some new items that we can add to a project: Report Wizard and Report. So far, so good – we’ll work with these items as we need them.

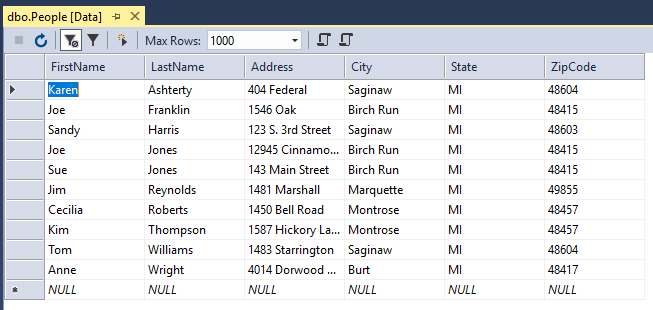
## Setting up a Test Database

Before we get into the reporting items, we need some data to work with. Since you are a pro at creating SQL Server Database files, we will use one in our demo. I created a new SQL Server Database file called RptAddresses in the C:\DB folder.

Here’s the schema for the table that I created, called People. I didn’t use spaces in any field names – be careful as spaces can cause problems with the report designer:



And here’s the data:



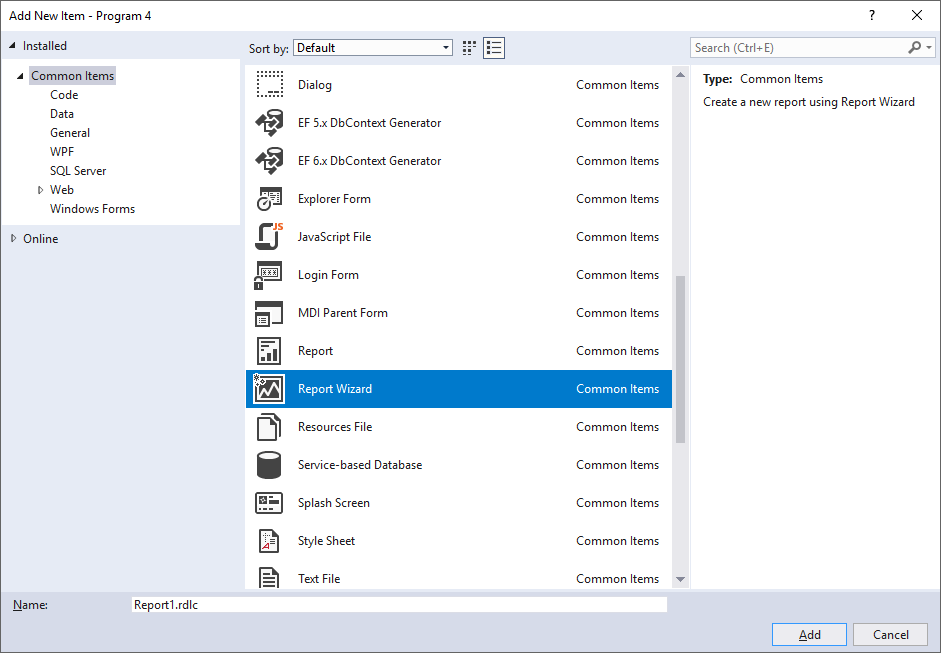
Now we are ready to get to our report building work since we have installed the necessary tools and we have a database available. It’s important to note that this report designer toolset we are working with will work with any database, not just SQL Server. You can hook it up to Oracle, mySQL, Access, et cetera. So, if you learn how to use it once, you’re pretty much set when it comes to reporting. Having said that, like most powerful things, there is a learning curve. And, as with most of these more esoteric topics, there are a few books around that will teach you how to use the report designer.

Let’s begin by creating a new Windows Forms application in Visual Studio (I am calling mine Program 4). By default, we will get our standard Form1 built. Just leave that form alone for the moment – we will eventually come back and work with it.

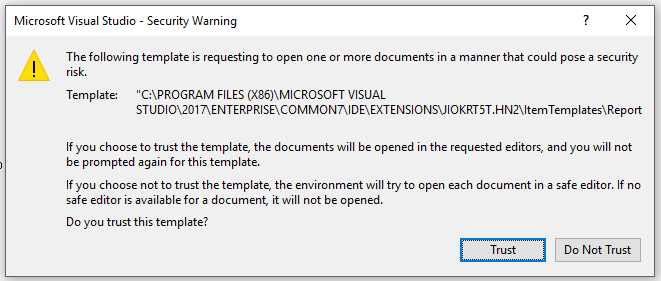
The next thing that we need to do is add a Report Wizard to our project. This will allow us to point to a database and return the data that we want in a DataSet. We’ll point our report designer at the DataSet and construct our report from there.

## Using the Report Design Wizard

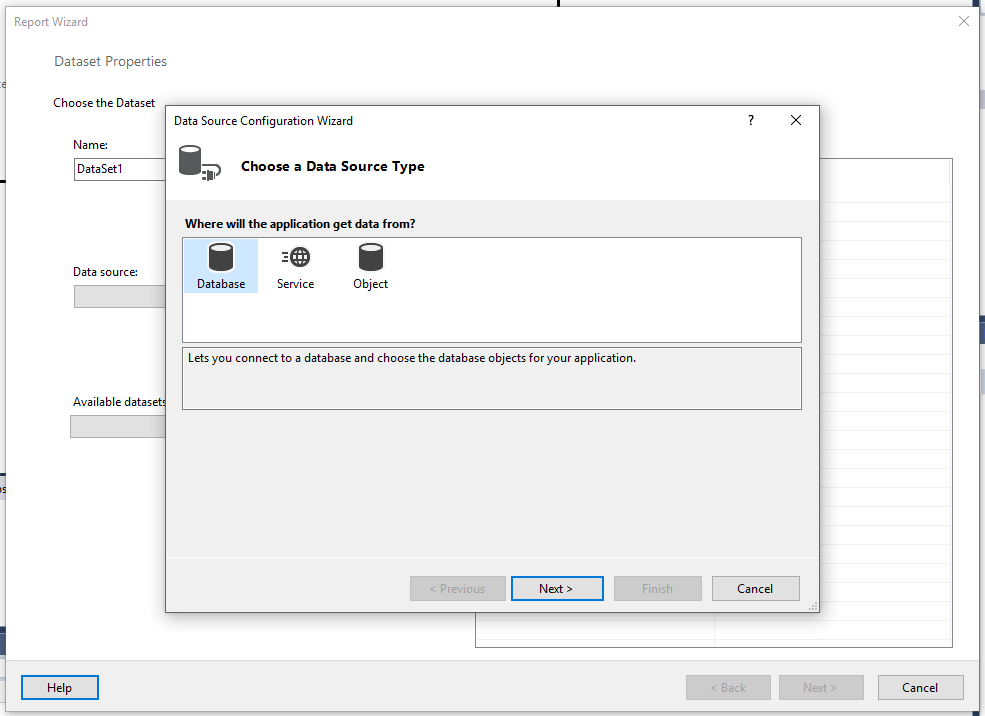
Start the Report Wizard by clicking on the menu item Project🡪Add New Item. Then click on the Report Wizard, not Report (both items were added by installing the Report Designer extension):



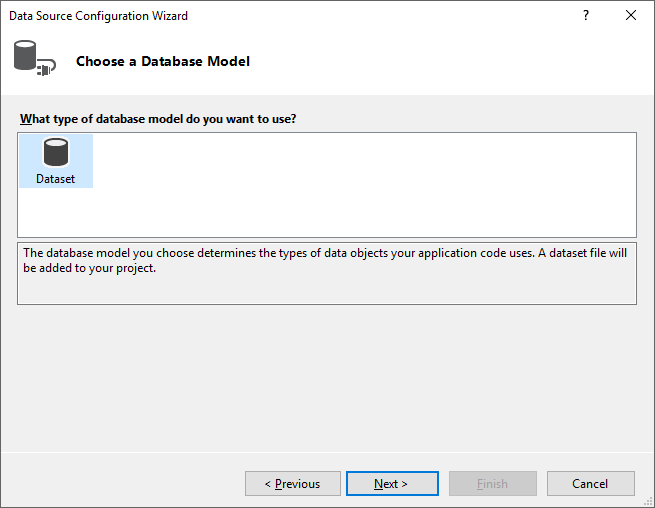
I will leave the report’s name alone (Report1.rdlc will be what the wizard creates for us) and just click on the Add button. This will launch the report designer wizard. You may be asked about security permissions. The template was created by Microsoft, so you can click on the Trust button:



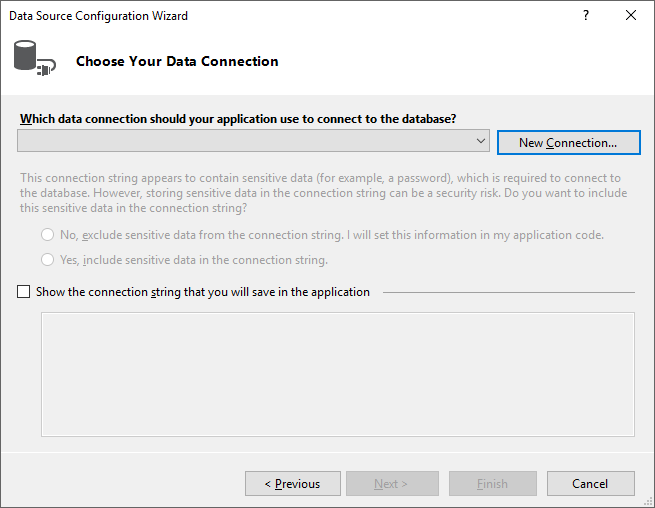
The report wizard begins by asking you to create a data connection to the data that you want to see in your report. We have worked with this in detail in the databases chapter, so I am not going to explain the minutiae as to what each step accomplishes here. I will just go through the process. Make sure that Database is selected and press Next:



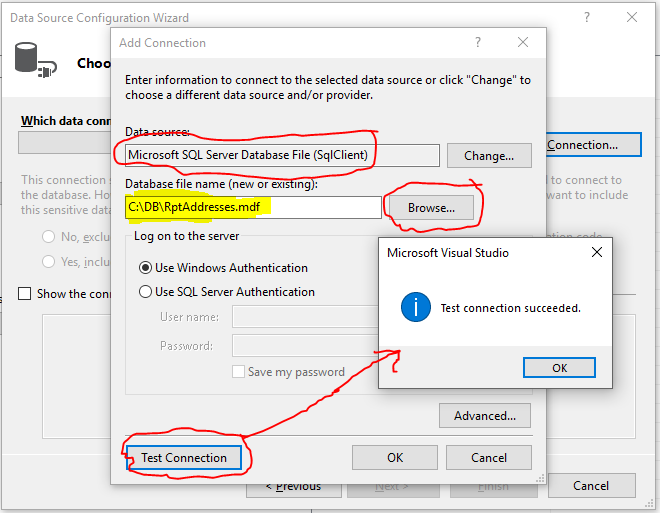
We want our results to end up in a DataSet, which is the default (and only choice), so click Next:



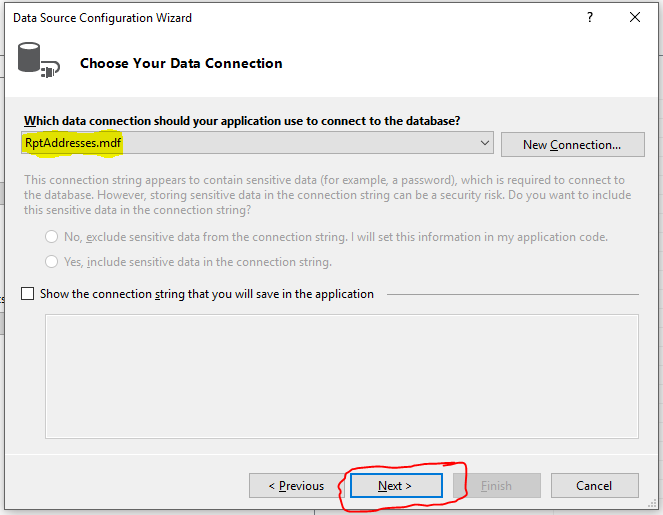
We now need to create a new connection to our database, so click the New Connection button:



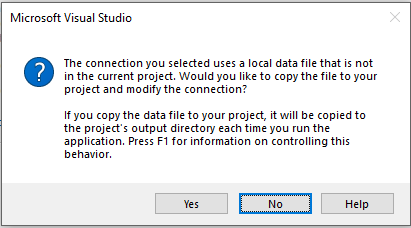
Make sure that Microsoft SQL Server Database file is the type selected in the Data Source type. Then press the Browse button to point at the RptAddresses database file that we created. Finally, make sure that everything is set up right by clicking on Test Connection. Click the OK button once the success message appears and then the OK button back on the Add Connection form:



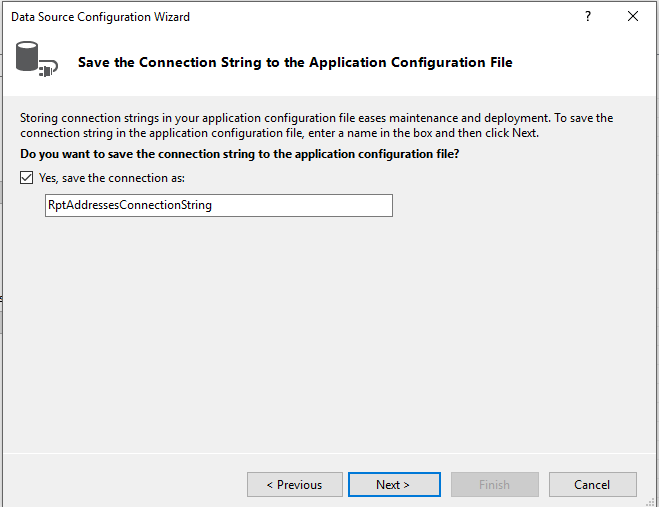
We are taken back to the Data Source Configuration Wizard screen with the Data Connection filled in, so we just press the Next button here:



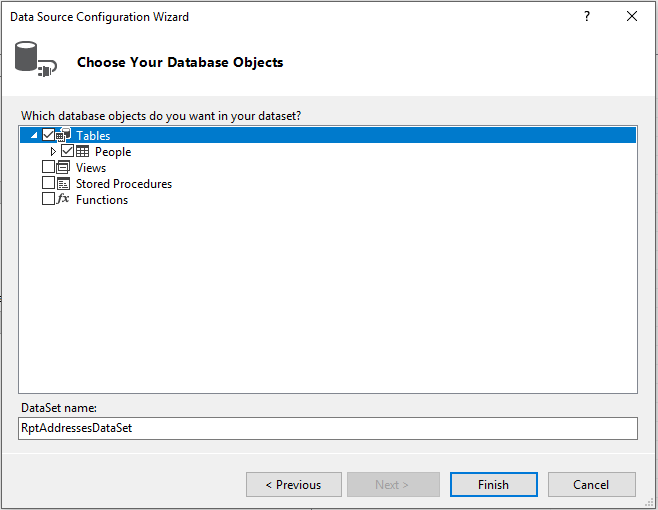
We are then prompted about creating a local copy of the database. Answer No:



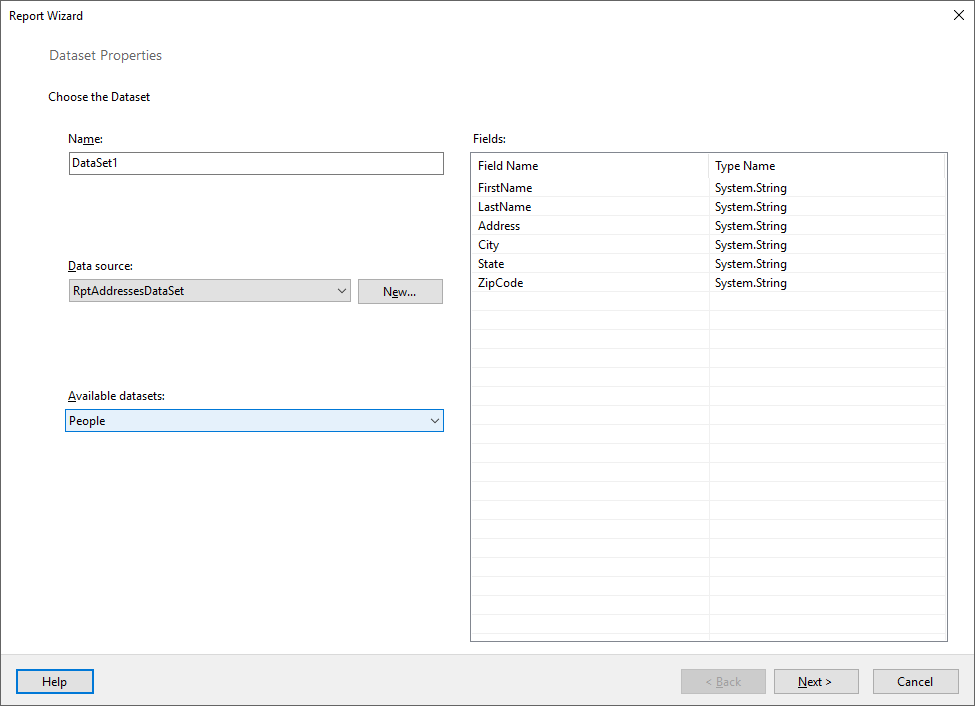
Click Next on the dialog regarding saving the connection string information:



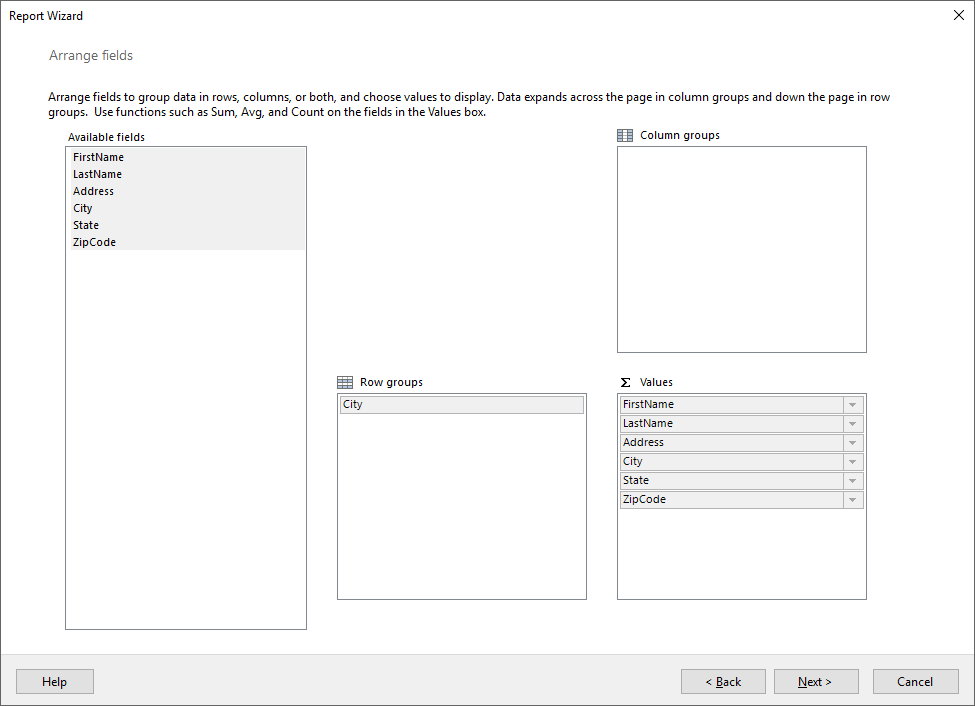
Now we are shown the tables in the database. We will select the People table by checking on it and then click the Finish button to end the Data Source Configuration Wizard:



All the work up to this point should have been very familiar to you as we looked at creating a data connection back in the database chapter. From here on out though, this is now the Report Design Wizard. At this point it shows us a summary of where our data is coming from and what data fields we have to work with. Since it’s pointing at the right data source, press the Next button:

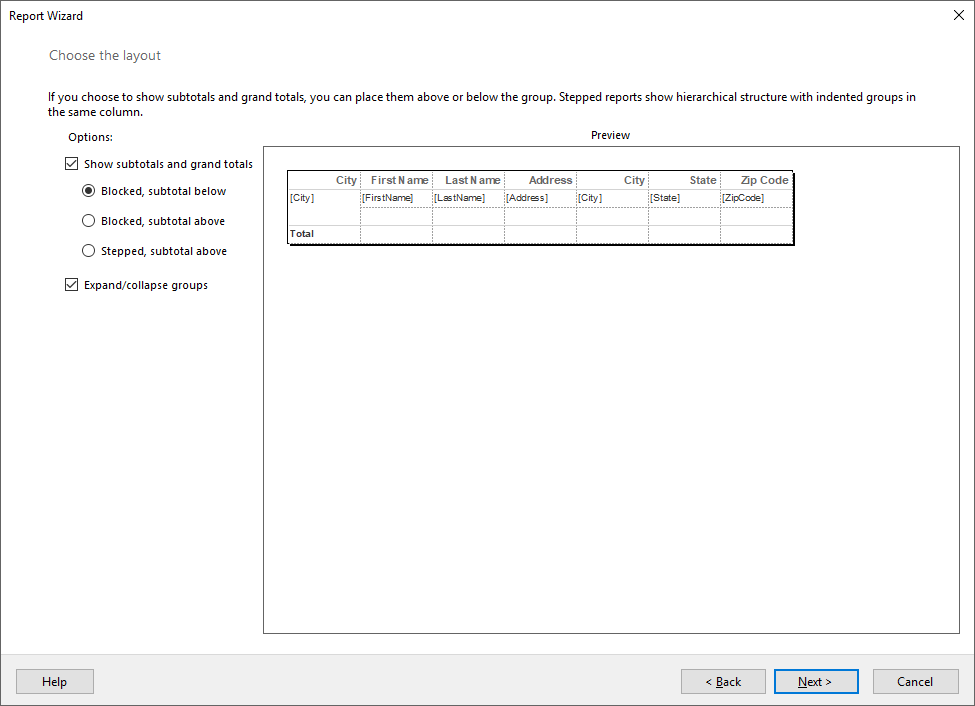


We now see the main report designer screen which will layout our report. We will group our addresses by the city where the address is located. We handle this but dragging the City field from the Available Fields box to the Row Groups box. We also need to list all the fields that we want printed on each row of report output, so handle this by dragging all fields (including City) to the Values box. This is what the form should look like once it’s completed:

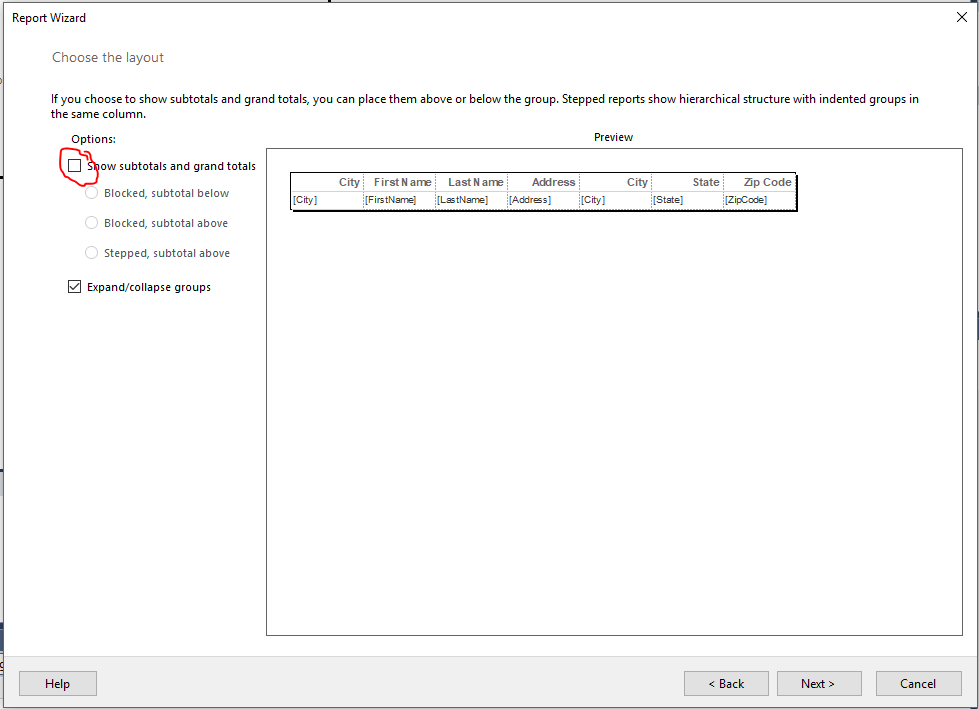


The report looks pretty reasonable for a first attempt, so let’s click the Next button.

The wizard then asks about how you want totals arranged:

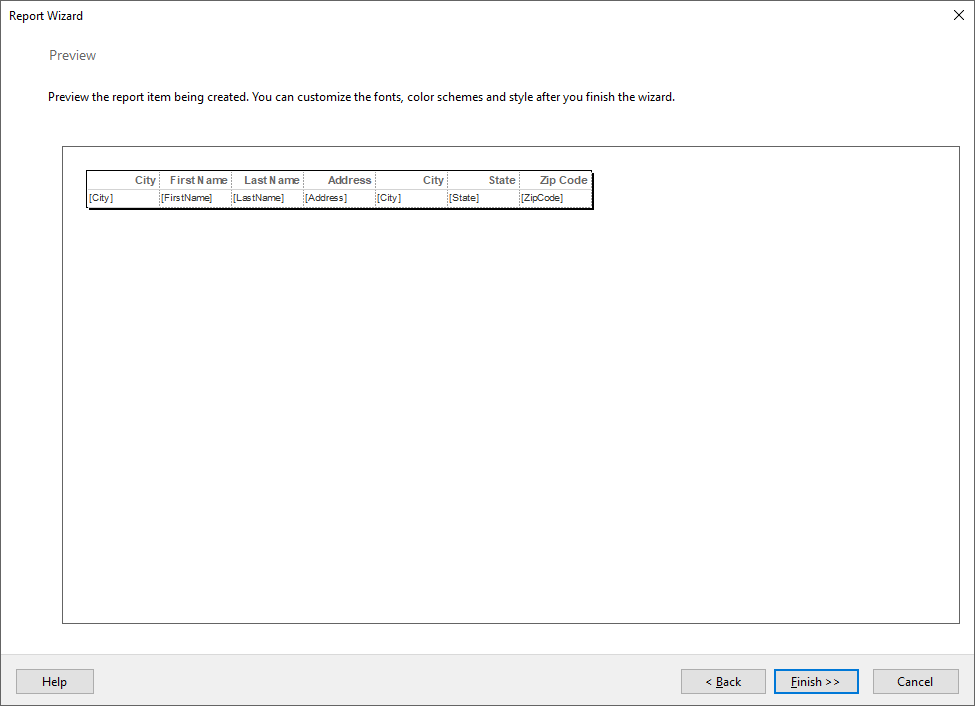


Since this is a report of addresses, it probably doesn’t make a lot of sense to have any subtotals, so uncheck that box. We may want to expand/collapse our groups so we will leave that one alone though. Here’s the form after I’ve made the changes:



Go ahead and click the Next button.

The wizard is now ready to generate our report. Notice that it tells us that we can customize colors, fonts and so on after it’s created. Let’s click on the Finish button:



We’re taken back to VB and we are shown the report that the wizard generated for us. We could go ahead and run the report wizard again (Project🡪Add New Item🡪Report Wizard menu option) if we wanted to create a second report. In reality, we could continue to use the wizard to generate reports ad nauseum. Do realize that there’s nothing that the wizard did that you couldn’t have done by yourself. You could have chosen to add a blank report (the menu option Project🡪Add New Item🡪Report), configured your own data source and then dragged and dropped items from the data source onto the report canvas. It’s just that the wizard is a quick way to get to a basic working report that we can then customize.

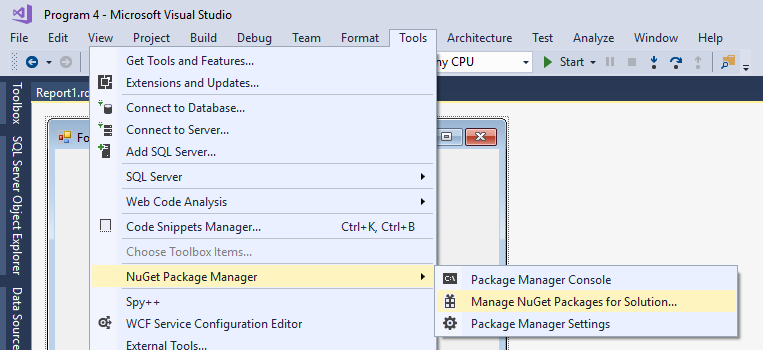
The next important step: SAVE YOUR PROJECT! Really – do it now!

We probably would like to be able to view the report that we created at some point. To do this, we need to add a ReportViewer control to our form. Unfortunately, just like the report wizard, it’s not installed by default. Also, it would have made too much sense to bundle and permanently install the report viewer control with the report wizard. The bottom line is that you don’t have the control on your computer. The good news is we can go and install it. The bad news is that you will have to re-install the control in every project where you want to use a report viewer. Whhhaaaaattttttt?

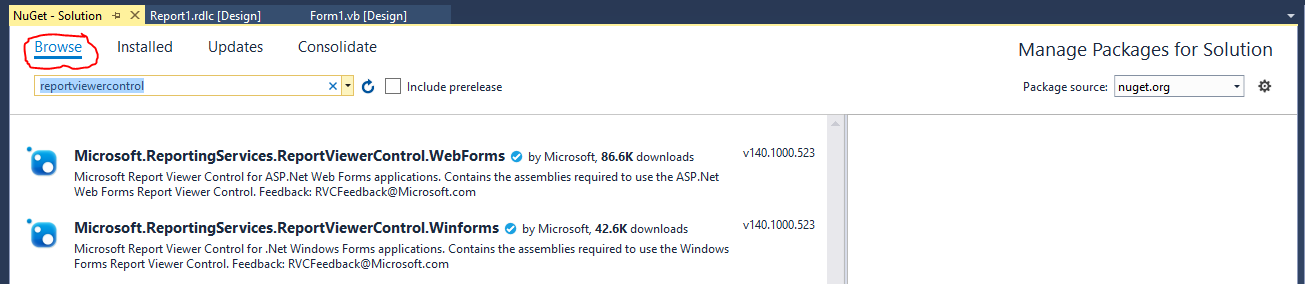
Yeah, Microsoft, in its efforts to go open source, have decided to employ NuGet to allow the installation of small project items. Remember how that the report wizard was installed using Extensions and Updates? Anything that we install that way will be permanently installed into Visual Studio and available whenever and wherever we want to use it, up until we uninstall the extension. So, what this means is that you have to install the report designer once.

The report viewer though, is a control. It’s small in size and typically used on a per-project basis, e.g. some projects may need the report viewer whereas others will not. Consequently, Microsoft has chosen to make this a NuGet downloaded entity that gets added into your project. It’s really not hard to download or install; it’s just a hassle that you can’t permanently add it to the main Toolbox forever. Oh well, the price of progress I guess…

I warned you before, but I am going to do it again: make sure your project is saved at this point just in case the network goes belly up while NuGet’s getting the control. To access NuGet, follow the menu option of Tools🡪NuGet Package Manager🡪Manage NuGet Packages for Solution:

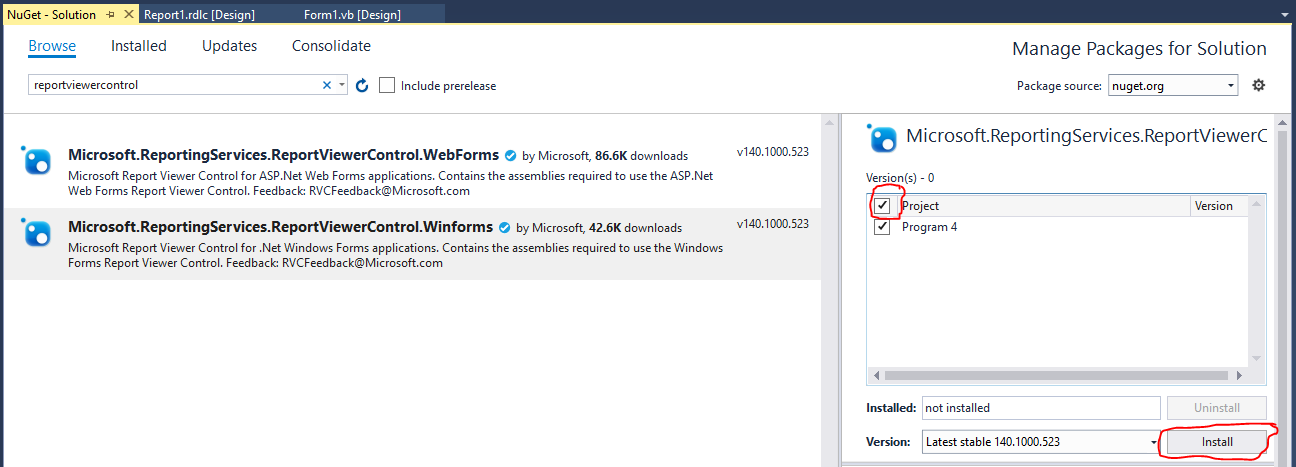


The NuGet package manager will start up. When it appears, click on the Browse tab and type in reportviewercontrol in the search textbox. You should see a couple of matches appear in the lower region after a second or two:

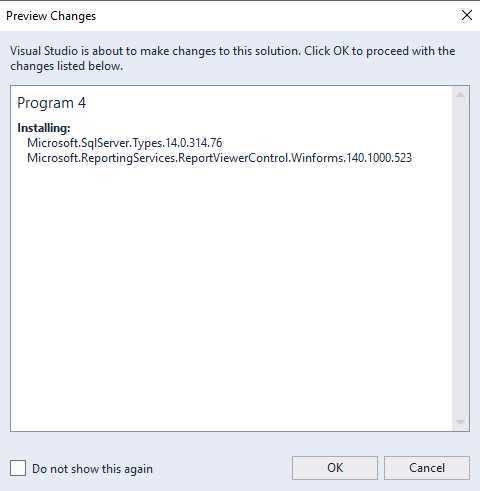


You will want to click on the item that ends in “.Winforms” Notice that you also have the ability to add reports to ASP websites too. You would use exactly the same report wizard, following exactly the same steps – how’s that for cool and adding to what you can display on the web? The only difference would be that when you get to this point in a web project, selecting the viewer control, you would pick the Webforms version instead of the Winforms one. Since we are writing desktop applications, we’ll again need to use Winforms.

Go ahead and click on the Winforms entry in the NuGet window. As soon as you do, the right pane of the NuGet Package Manager changes to show you your project. Click the checkbox to the left of Project. This will automatically check both the Project and whatever the name of your project is (remember mine is called Program 4). You then need to press the Install button to have NuGet run out across the network, download the control and install it under your project:

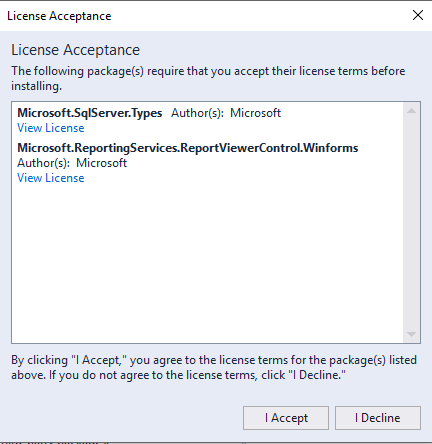


Visual Studio lets you know what NuGet plans on doing for you (and gets your approval) before it starts anything:



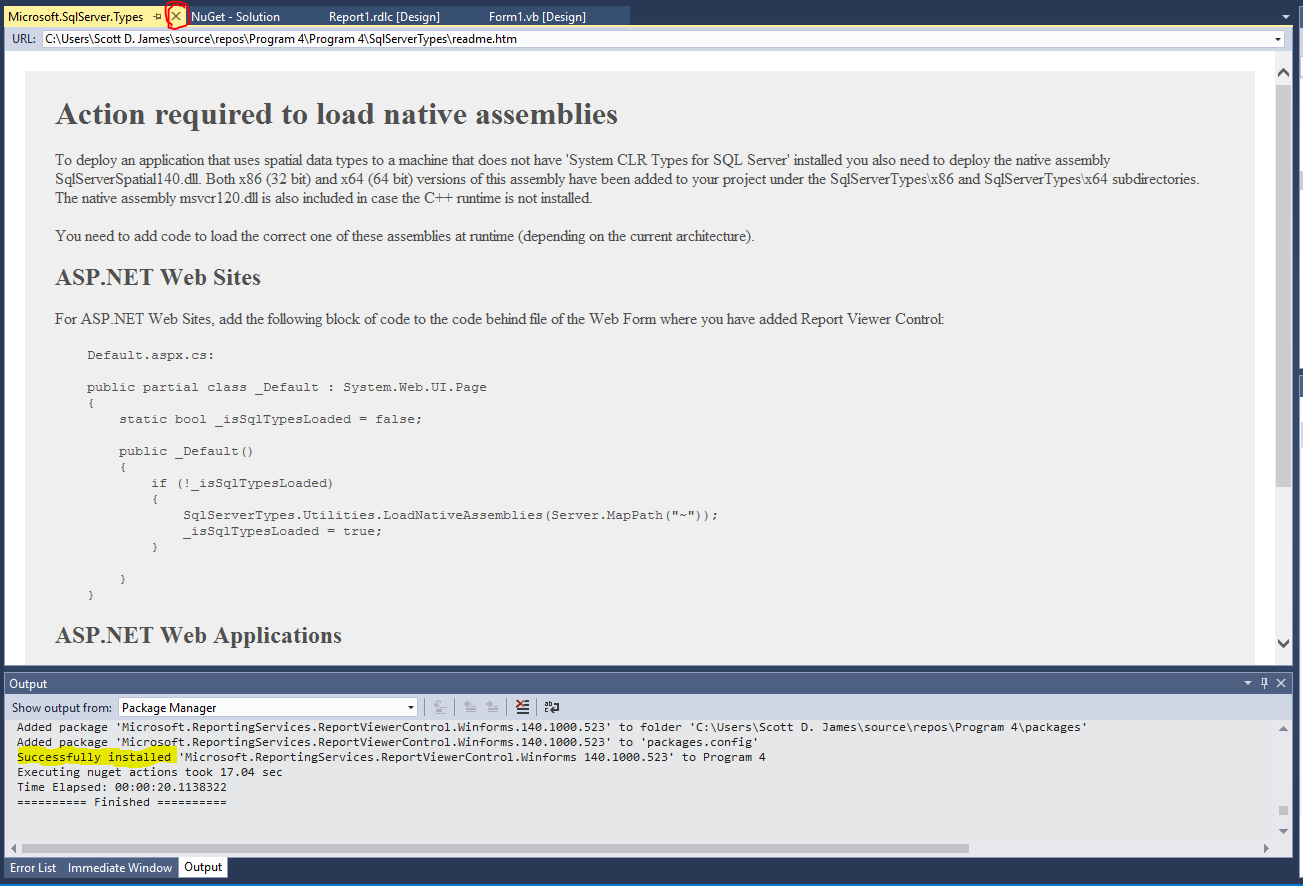
Click the OK button to let NuGet start the installation process.

You are then shown a licensing screen:

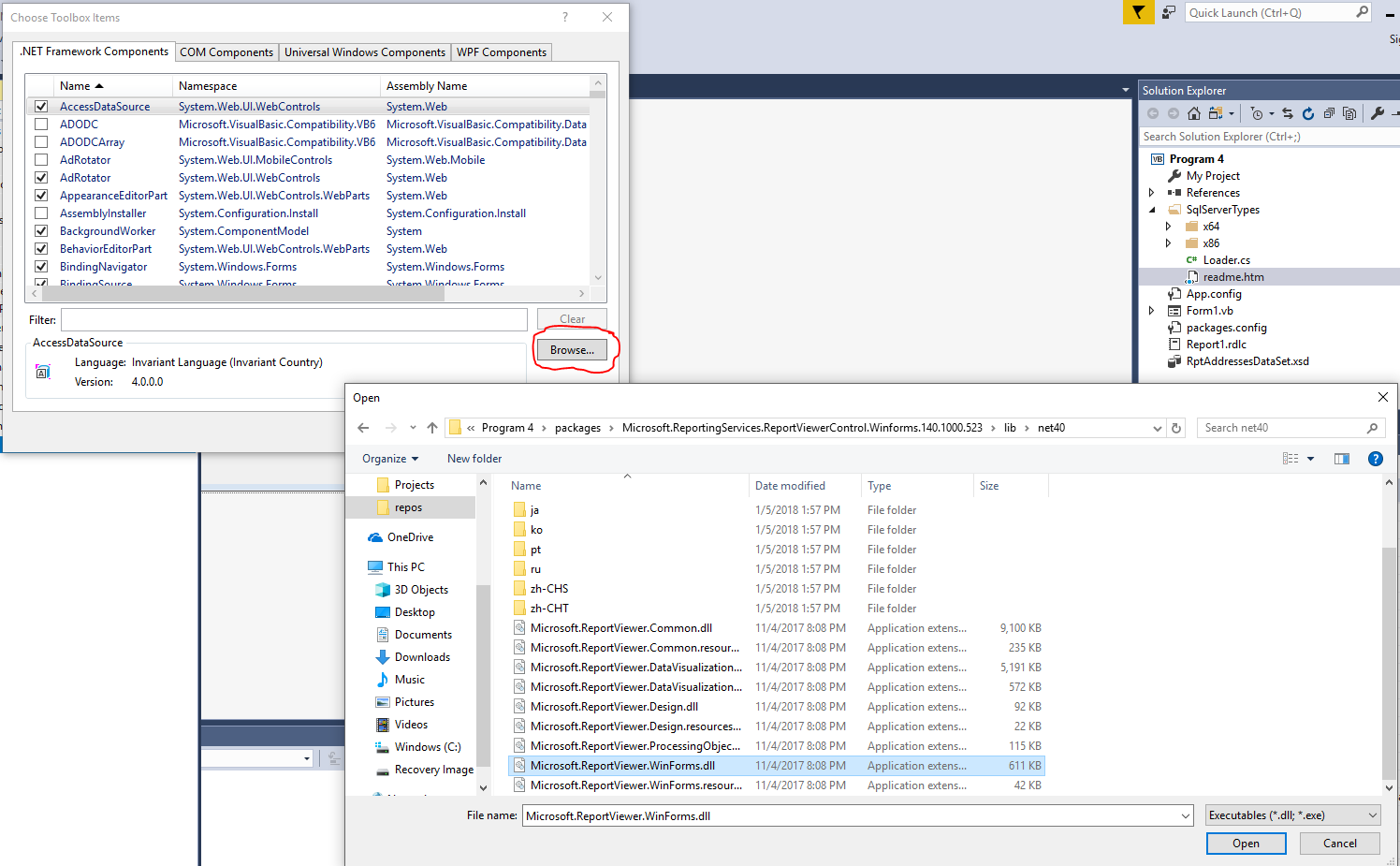


Assuming that you want the control installed, click “I Accept.”

Once the installation has finished, the NuGet Package Manager screen will disappear and you’ll return to the main Visual Studio window. There will be a pop-up HTML document that appears talking about adding in some stuff you don’t need. You can just close that window. Also, down in the Output window, you should see that the ReportViewerControl installed successfully:

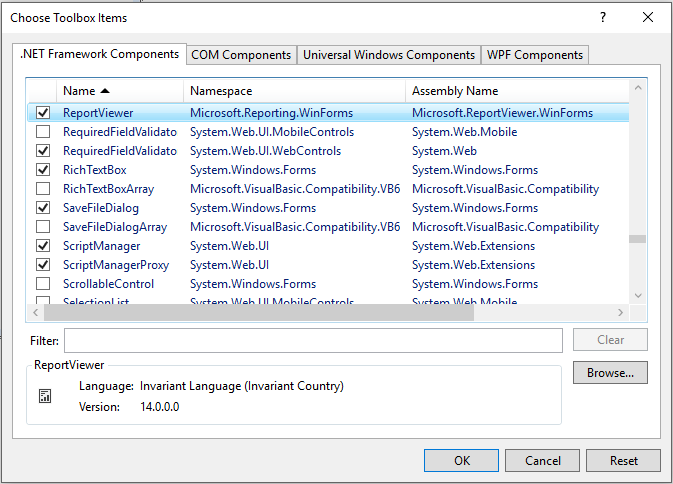


We’re almost there… We now need to access the viewer control so that we can put it on our form. You already know how to add this to the toolbox from our work with building our own controls. You’ll need to use the Tools🡪Choose Toolbox Items menu choice. Click on the Browse button from the .NET Framework Components when the Choose Item dialog appears:



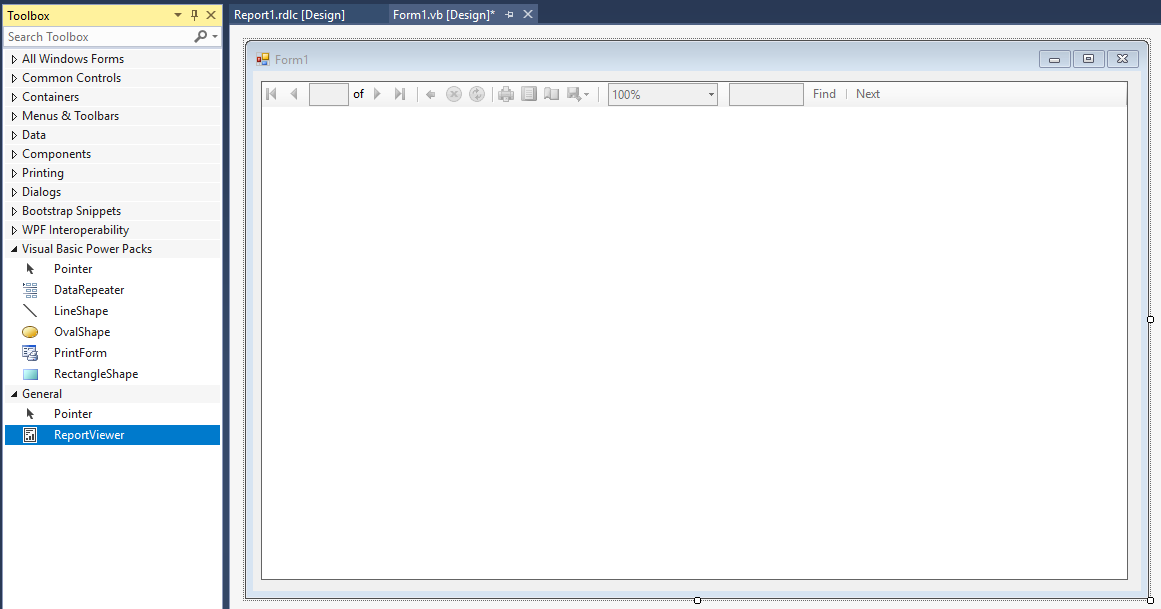
You will then navigate to the NuGet package directory in your solution (the path is shown in the OpenFileDialog above), click on the Microsoft.ReportViewer.WinForms.dll file and then click on the Open button

The ReportViewer control will appear checked in the list of selected controls on the Choose Toolbox Items Manager:

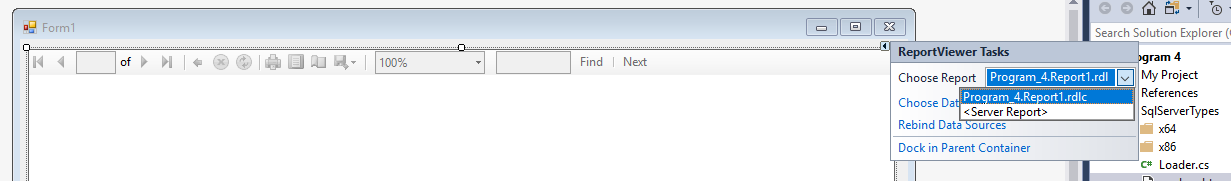


Click the OK button to return to your form.

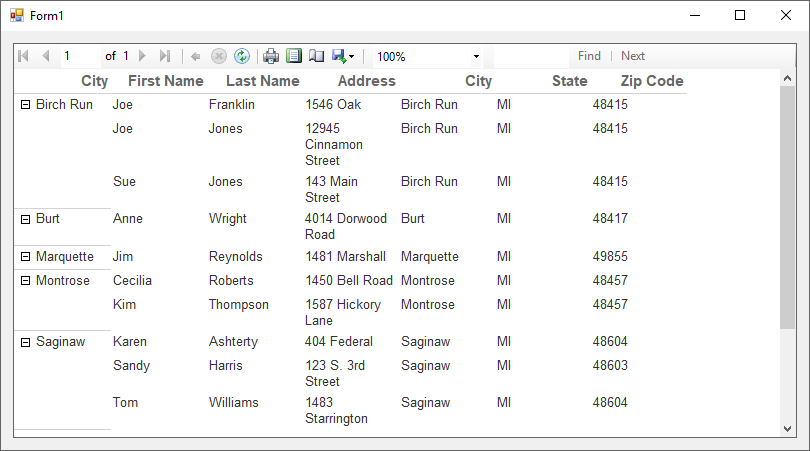
You will now have the ReportViewer control in your toolbox. Drag and drop a copy to your form:



There is a Quick Tasks arrow on the ReportViewer control. Go ahead and click on this. One of the tasks is to choose which report to view. In the dropdown box, select the report (.rdlc file) that we created earlier in the project.

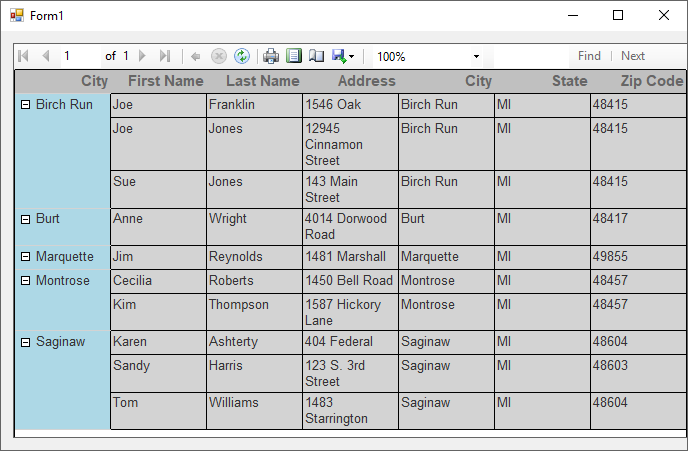


Finally, the moment of suspense – run the application. It’ll take a few seconds to start up and then you’ll see your report:

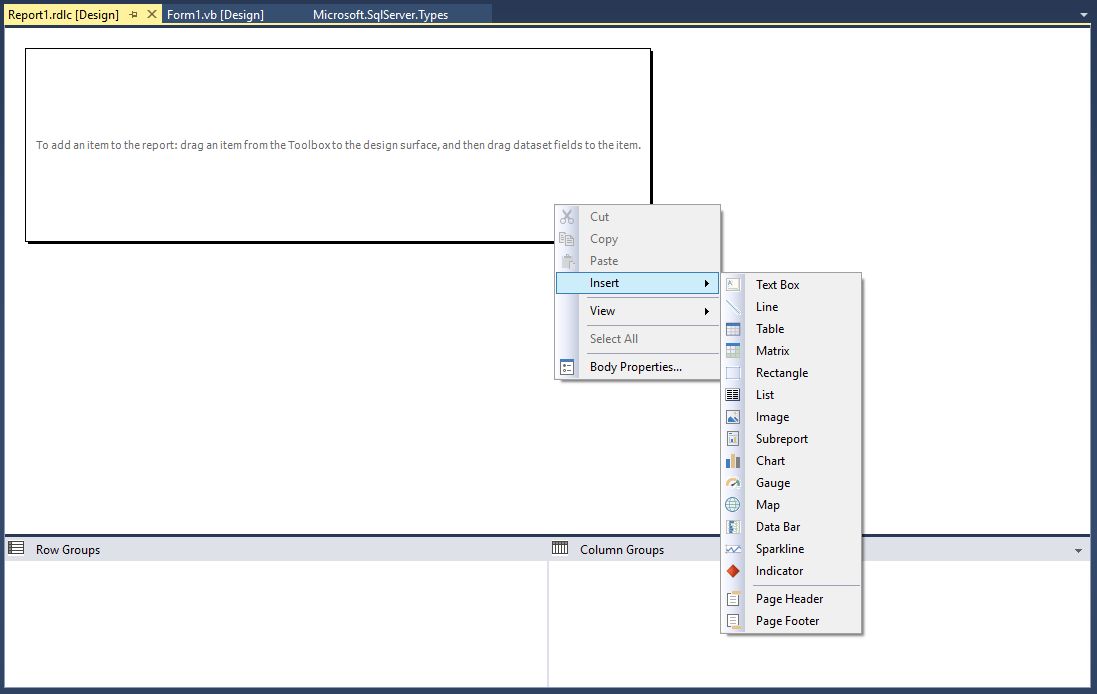


Realize that data is grouped by City and that we can expand/contract those groups. The report also handles pagination automatically. Users have the choice of printing and exporting (you can export to PDF, Excel or Word) the report from the viewer control too. Additionally, zooming and searching facilities are built in. Not too bad considering that we didn’t really do *that* much work. Can you imagine building this report yourself and trying to print it based on what we learned earlier?

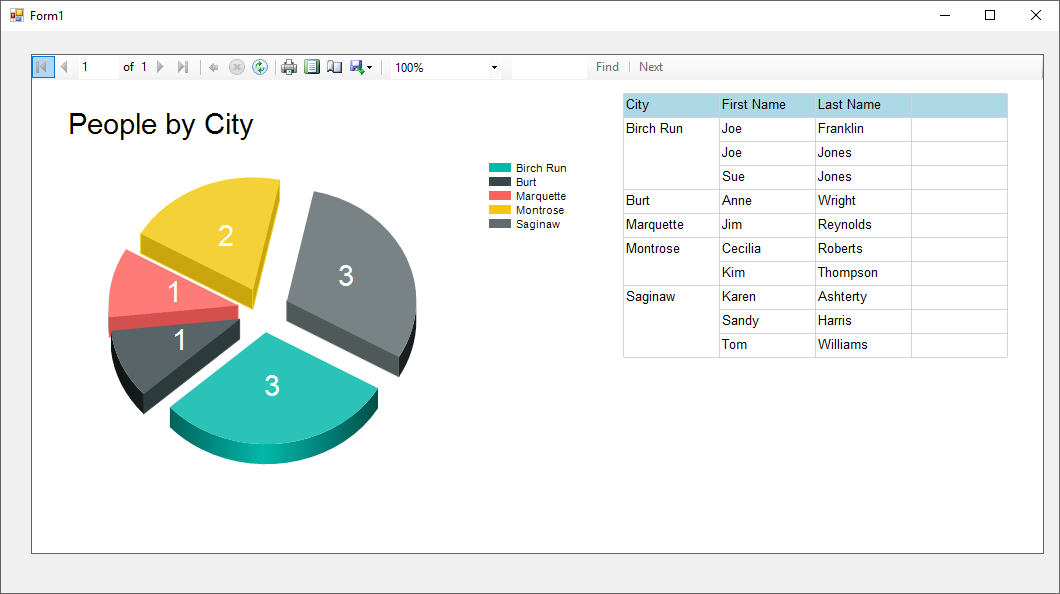
With a little bit of playing around, we can even really customize the style of the report if we choose (Program 5):



The Report Wizard is pretty limited in its capabilities. A Report’s blank canvas gives you far more functionality:



Notice that we can add different kinds of items such as charts, gauges and maps, so our reports do not have to be purely textual in nature. There is a **lot** to learn about using this control, so I would recommend that you spend some time looking for resources to help you get started. In case you’re interested, here’s a graph report of the number of people who live in each city (Program 6):



Some of you may have heard of Crystal Reports or your company may use it for their reporting purposes. Much of what you can do with Crystal Reports can also be done in the report designer. Both have the ability to create expressions that are SQLish in nature. If you use one of the reporting tools, learning the other comes quite easily.

Microsoft now has a different approach that it would like you to take with regard to reporting. This is possibly why the reporting pieces we used in this chapter are no longer bundled with Visual Studio and why you had to go get them manually. Microsoft would like you to use its SQL Server Reporting Services (SSRS). SSRS looks a lot like the report builder; the report files end in .rdl, whereas ours ended in .rdlc. To me the biggest drawback to SSRS is that you need to have a Reporting Services server active to serve the reports. The good old report designer that we used has no such requirement or restriction. Everything can be file based and located on one machine sans server.

Regardless of which reporting facility you learn, you’ll see that they are quite comprehensive and powerful with what they can do. Jumping from one to another has a few caveats, but many of the main concepts are the same. In any case, you’ll have to agree that this is much better than using any of the earlier printing techniques that we learned about!

Don’t be alarmed that I didn’t show you any source code form behind the scenes of the project. There’s not much there to look at – wizards, you know. If you are desperate though, here’s the code behind the first report application we wrote:

'Chapter 22 - Program 4

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

'TODO: This line of code loads data into the 'RptAddressesDataSet.People'

'table. You can move, or remove it, as needed.

Me.PeopleTableAdapter.Fill(Me.RptAddressesDataSet.People)

Me.ReportViewer1.RefreshReport()

End Sub

End Class

Most of the heavy lifting was done in the hidden designer file:

<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> \_

Partial Class Form1

Inherits System.Windows.Forms.Form

'Form overrides dispose to clean up the component list.

<System.Diagnostics.DebuggerNonUserCode()> \_

Protected Overrides Sub Dispose(ByVal disposing As Boolean)

Try

If disposing AndAlso components IsNot Nothing Then

components.Dispose()

End If

Finally

MyBase.Dispose(disposing)

End Try

End Sub

'Required by the Windows Form Designer

Private components As System.ComponentModel.IContainer

'NOTE: The following procedure is required by the Windows Form Designer

'It can be modified using the Windows Form Designer.

'Do not modify it using the code editor.

<System.Diagnostics.DebuggerStepThrough()> \_

Private Sub InitializeComponent()

Me.components = New System.ComponentModel.Container()

Dim ReportDataSource1 As Microsoft.Reporting.WinForms.ReportDataSource =

New Microsoft.Reporting.WinForms.ReportDataSource()

Me.ReportViewer1 = New Microsoft.Reporting.WinForms.ReportViewer()

Me.RptAddressesDataSet = New Program\_4.RptAddressesDataSet()

Me.PeopleBindingSource = New

System.Windows.Forms.BindingSource(Me.components)

Me.PeopleTableAdapter = New

Program\_4.RptAddressesDataSetTableAdapters.PeopleTableAdapter()

CType(Me.RptAddressesDataSet,

System.ComponentModel.ISupportInitialize).BeginInit()

CType(Me.PeopleBindingSource,

System.ComponentModel.ISupportInitialize).BeginInit()

Me.SuspendLayout()

'

'ReportViewer1

'

ReportDataSource1.Name = "DataSet1"

ReportDataSource1.Value = Me.PeopleBindingSource

Me.ReportViewer1.LocalReport.DataSources.Add(ReportDataSource1)

Me.ReportViewer1.LocalReport.ReportEmbeddedResource =

"Program\_4.Report1.rdlc"

Me.ReportViewer1.Location = New System.Drawing.Point(12, 12)

Me.ReportViewer1.Name = "ReportViewer1"

Me.ReportViewer1.ServerReport.BearerToken = Nothing

Me.ReportViewer1.Size = New System.Drawing.Size(784, 395)

Me.ReportViewer1.TabIndex = 0

'

'RptAddressesDataSet

'

Me.RptAddressesDataSet.DataSetName = "RptAddressesDataSet"

Me.RptAddressesDataSet.SchemaSerializationMode =

System.Data.SchemaSerializationMode.IncludeSchema

'

'PeopleBindingSource

'

Me.PeopleBindingSource.DataMember = "People"

Me.PeopleBindingSource.DataSource = Me.RptAddressesDataSet

'

'PeopleTableAdapter

'

Me.PeopleTableAdapter.ClearBeforeFill = True

'

'Form1

'

Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)

Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font

Me.ClientSize = New System.Drawing.Size(808, 419)

Me.Controls.Add(Me.ReportViewer1)

Me.Name = "Form1"

Me.Text = "Form1"

CType(Me.RptAddressesDataSet,

System.ComponentModel.ISupportInitialize).EndInit()

CType(Me.PeopleBindingSource,

System.ComponentModel.ISupportInitialize).EndInit()

Me.ResumeLayout(False)

End Sub

Friend WithEvents ReportViewer1 As Microsoft.Reporting.WinForms.ReportViewer

Friend WithEvents PeopleBindingSource As BindingSource

Friend WithEvents RptAddressesDataSet As RptAddressesDataSet

Friend WithEvents PeopleTableAdapter As

RptAddressesDataSetTableAdapters.PeopleTableAdapter

End Class