CHAPTER 1

An Overview

This chapter puts Microsoft SQL Server 2000 Reporting Services in context. You'll learn about its architecture and its place in the world and get a brief overview of its features and functionality. This *is* an important chapter because it helps you understand why Reporting Services is more than just an alternative (or supplement) to Crystal Reports and how it can make your life easier.

We know you're chomping at the bit to get started—and perhaps you've already installed the software, snooped around, and tried out a report or two. Hopefully you've extracted a copy of the symmetric encryption key and have the backup off-site, along with the backup file password. Yes? Good. Now we're ready to dispel any prejudices or misconceptions your explorations have generated.

Security is (or should be) very important to you; at least we know your boss is concerned with security. All along the way, we'll show you how to secure your data and reports in ways that the documentation doesn't mention or only touches upon. We'll discuss encryption and topics that seem pretty far removed from "reporting," but without these protections, you might as well post reports exposing your company secrets on the 20-foot newsreader board in Times Square.

NOTE We've spent quite a bit of time creating audio-annotated screen capture demonstrations of aspects of Reporting Services. You'll find these demos on the accompanying DVD. We call these Guide me! videos. If you need help on a section, look for the Guide me! icon in the margin.

It is difficult to contain my genuine heartfelt enthusiasm and appreciation when I attempt to succinctly encapsulate what Microsoft SQL Server 2000 Reporting Services is and provides. Just bear in mind that I don't do marketing doublespeak; I am British, and we are the modest world masters of understatement.

–Peter

What Is Microsoft SQL Server 2000 Reporting Services?

At long last, developers have a brand-new server-based reporting solution from Microsoft built from the ground up, almost completely but not entirely in .NET managed code. Without being *too* enthusiastic, we can say that Reporting Services enables extremely straightforward, centrally managed, rapidly developed, easily extended, scalable, secure, fixed as well as interactive, proactive as well as passive database reporting to a variety of (extensible) output formats, from *any* database or data source (and not just SQL Server) whose learning curve is hardly a curve; it's almost flat. <Whew!> Best of all, it's free! Well, let's qualify that statement a little: it's included with the SQL Server Standard or better license at no additional cost.

Not having taken the Microsoft red pill,[1] we can freely draw comparisons to other reporting products on the market. One of the first that undoubtedly flashes into everyone's minds is Crystal Reports. In contrast to Crystal Reports, Reporting Services is a product that Microsoft definitely got right from the outset. We see a new age dawning of assault on Crystal practitioners. Stand by to be assimilated—resistance is futile.

I first saw what Microsoft was hatching with Reporting Services (code-named Rosetta) in February 2003. Back then it was planned as a release feature of the next version of SQL Server (code-named Yukon). After we and the other folks privileged with this sneak preview had relocated the shreds of our blown-off socks, we gave our feedback and petitioned to have Reporting Services

Okay, if you are at the pinnacle of pushing Crystal to its functional feature limits, you may well find that Crystal offers additional functionality over Microsoft Reporting Services, even if it's expensive and awkward to learn and use. However, we think that the vast majority of developers and power users in the market want reporting that's easy to use, learn, and develop with. Microsoft Reporting Services certainly delivers that.

What Is the Source of the Report Data?

Before we go any further, we want to reinforce the message that although the product name includes the moniker "SQL Server 2000" as in "Microsoft SQL Server 2000 Reporting Services," in fact SQL

[1] Or been on their payroll.

released now, as in right now, with the current version of SQL Server 2000. Microsoft appeared to take that feedback to heart, and in June 2003 it was publicly announced at the TechEd Conference in Dallas: Rosetta for SQL Server 2000 would indeed be released at the end of 2003. We're both glad that Rosetta was untied from Yukon because we don't expect this innovative version of SQL Server to appear for another year (or so).

–Peter

Server 2000 is required only as the catalog repository for the deployed (managed) reports and their metadata, and not necessarily as the *only* data source that those reports can be based on.

In fact, right out of the box Reporting Services includes built-in support to generate reports from a number of data sources through what the Reporting Services folks call ***Data Processing extensions***. Think of them as another form of .NET Data Providers; they're based on the same technology. Data is extracted from SQL Server and Oracle as well as any other database management system (DBMS) that can be accessed via either OLE DB providers or ODBC drivers using Data Processing extensions. This means that most commercial data stores are accessible because there are managed .NET Data Providers as well as OLE DB or ODBC drivers for virtually every serious data source. These providers are available from Microsoft or from the DBMS manufacturers and third-party data provider vendors such as DataDirect (and others).

Reporting Services is designed with an open architecture. No, Microsoft does not publish the source code,[2] but under the covers, Reporting Services' Data Processing extensions utilize a subset of easily understood .NET Data Provider interfaces. If you can get or write a .NET Data Provider for a custom data source (something that's moderately easy), Reporting Services can connect to and generate reports on that custom data source.[3] We'll discuss this in more detail in Chapter 13 where we'll show you our implementation to report on the Windows system event logs.

Reporting Services: The Main Components

In a nutshell, to author a report, the developer creates a ***report definition*** using the Report Designer add-in to Visual Studio .NET 2003 or a third-party authoring tool.[4] While under development, each

[2] Like *that* will happen…

[3] Writing a .NET Data Provider in managed code is much easier than writing an ODBC driver, unlike writing an OLE DB data provider, which requires a Ph.D. in nuclear physics and abstract reasoning.

[4] See the DVD for a list of these tools as well as working examples of their RDL generators.

Delving into the intermediate language (IL) of these components (which is, of course, expected of a potty-trained geek like me) leads one to conclude that C# was the developer's language of choice.

–Peter

Yes, but those folks were C++ geeks and not smart enough to learn Visual Basic .NET.

–Bill

OK, Bill, I'll let you win. The Visual Basic .NET geeks can be smarter, if that's what you want, but the C# geeks will remain richer. "Developers who program primarily in C# earn 26 percent more than those who develop primarily in Visual Basic .NET." (www.fawcette.com/vsm/2003_06/magazine/features/salarysurvey/)

–Peter

Yes, but C# programs take 28 percent longer to write, so they spend more time at the office while the Visual Basic programmers are out having a life.

–Bill

Nice try, but if you read the article, this was based on hourly rates.

–Peter

report definition is maintained in its own XML file, which is given the file extension of "RDL". We'll tell you more about RDL later on; however, succinctly, RDL files contain the layout, graphics, connection, and query information as well as Report Parameter definitions and almost all other report logic. Using the Report Designer, developers can author, tune, and refine reports without having to access the Report Server. At design time, reports can be "previewed" using the Report Designer, which provides a close approximation of what the finished report will look like when rendered by the Report Processor.

Once a report definition is complete, the report is published, or "deployed," to a Report Server where it becomes a compiled ***managed report***. At this point, the DBA or report administrator can decide how and to whom a report is to be made available.

The Report Server exposes a SOAP[5] interface, through which these managed reports can be programmatically manipulated. Straight out of the box, there is a fully functional Report Manager. This is an ASP-based application that sits on top of the Report Server's SOAP interface and provides a user-friendly GUI environment with which to manage reports. Additionally, there is a Report Scripting Utility (rs.exe) that enables you to execute scripts that make calls to the SOAP interface.

We'll show you how to launch managed reports from code or simply tune up or customize the existing Report Manager. When it's time to view a managed report, the compiled report definition is fed to the Report Processor, which fetches the data, merges it with the report layout, renders the report into a selected format, and streams back the result. Reports can be processed on demand or according to customizable schedules. It's all really very simple. But don't close the book yet. There are a few more details that you'll want to know.

Reporting Services is composed of a number of well-integrated components, most of which are written in .NET managed code. Some of these components we'll be exploring, and we shall see that they consist of open and extensible subcomponents. The overall look and feel of the Reporting Services Report Designer is reminiscent of web (ASP) application development in Visual Studio .NET or the

[5] We discuss how to manipulate reports through SOAP in Chapter 9¾.

Report Designer in Microsoft Access—but it's far more flexible. To build a report you first define the location of the source of its data, lay out some controls on the report surface, and bind those controls to the data. For much of this, it's all drag and drop. When the report is generated, the Report Server extrudes the report in much the same way that the ASP.NET engine returns HTML from a web page. But Reporting Services does not stop there. It can also render reports in a variety of other ways, as we'll soon explain.

Here are the main components of Reporting Services:

- A Visual Studio .NET 2003 add-in ***Report Designer*** for authoring reports. As we'll show you in Chapters 3, 6, and 7, this is alarmingly reminiscent of the Report Designer in Microsoft Access.[6] It extends Visual Studio .NET 2003 to permit developers to define Reporting Services DataSets, specify a query and layout, and preview and deploy the report definitions, which are saved to report definition (RDL) files in the project and then when deployed, are saved to the Report Server database as intermediate language (IL).
- An ASP.NET XML Web Service–based ***Report Server*** works in conjunction with a .NET Windows System ReportServer Service to process and provide managed reports in a variety of rendered output streams and stores its configuration, processing information, and other metadata in SQL Server.
- An ASP.NET web application–based ***Report Manager*** for centralized report management is an important tool for DBAs, developers, and users. It's designed for report management and is an ideal and secure platform for users to select and launch reports. The Report Manager permits developers or database administrators (DBAs) to define or modify Data Sources, locate and organize reports, and create ***Subscriptions***, which permit reports to be e-mailed on a scheduled basis. We detail the Report Manager in Chapter 4.

Figure 1.1 gives you the basic idea how the basic Reporting Services components fit together.

[6] It's almost as if both SQL Server Reporting Services and Access were written by the same company.

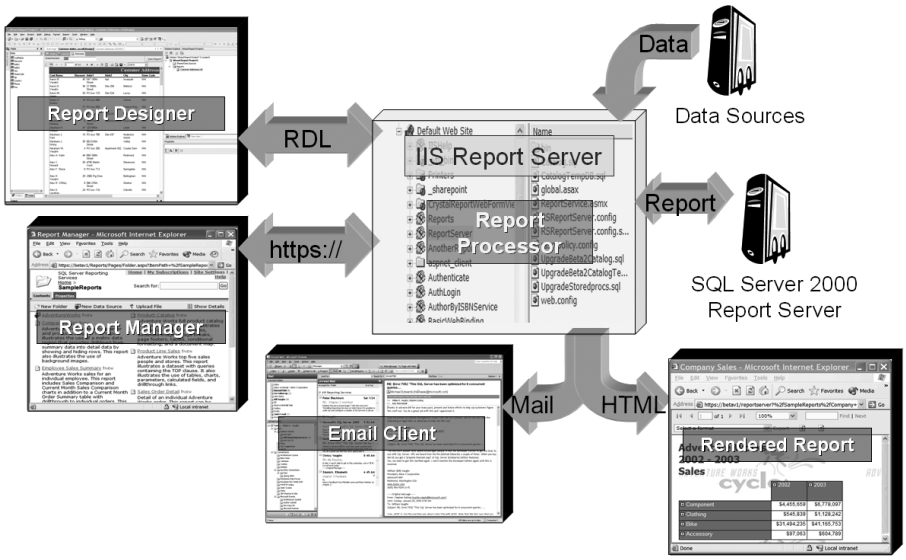

Figure 1.1 SQL Server Reporting Services Overview

Writing Reports: The Report Designer

WYSIWYG—what you see is what you get. Goodness, haven't we come a long way since the Apple Macintosh?
–Peter

Peter does not remember (he was in primary school) when WYSIWYG appeared on CPT word processors, which predate the Apple Macintosh.
–Bill

Reporting Services comes with a WYSIWYG Report Designer add-in to Visual Studio .NET 2003 (as shown in Figure 1.2) to aid in the construction of report definitions. The Report Designer provides a new set of Business Intelligence Project types containing two out-of-the-box templates: a Report Project wizard and a blank report project—we'll show you how to create your own Report Templates in Chapter 9. No, Visual Studio .NET doesn't have to be the Rolls Royce (or Cadillac) version to host the Reporting Services add-in. The inexpensive Visual Basic .NET 2003 Standard (or Academic) version is sufficient.

The Report Designer is as intuitive and easy to use as the Microsoft Access Report Designer, but it's far more powerful. The typical report-authoring approach is to create a Reporting Services DataSet (which, by the way, is not to be confused with an ADO.NET DataSet). This DataSet is connected to the source of the data through a Reporting Services Data Source, which is effectively a Connection string. Once the underlying data is provided for the report, the next step is to visually lay out the report controls, binding them to the underlying data,

customize their properties, and preview the report. All of this can be done without having to involve the Reporting Services server.

Once you're ready, you can ask the Report Designer add-in to "deploy" either the report definition or the whole report project, which might contain several report definitions, to the designated IIS[7] server (where your Reporting Services virtual directories are hosted). This creates one or more managed reports on your server—one for each report in your project—and creates any Data Sources that are needed. At this point, users can execute any reports your report DBA has enabled to be visible. This chapter gives you an overview of this process, and Chapters 6 through 9 deal with the process in greater depth.

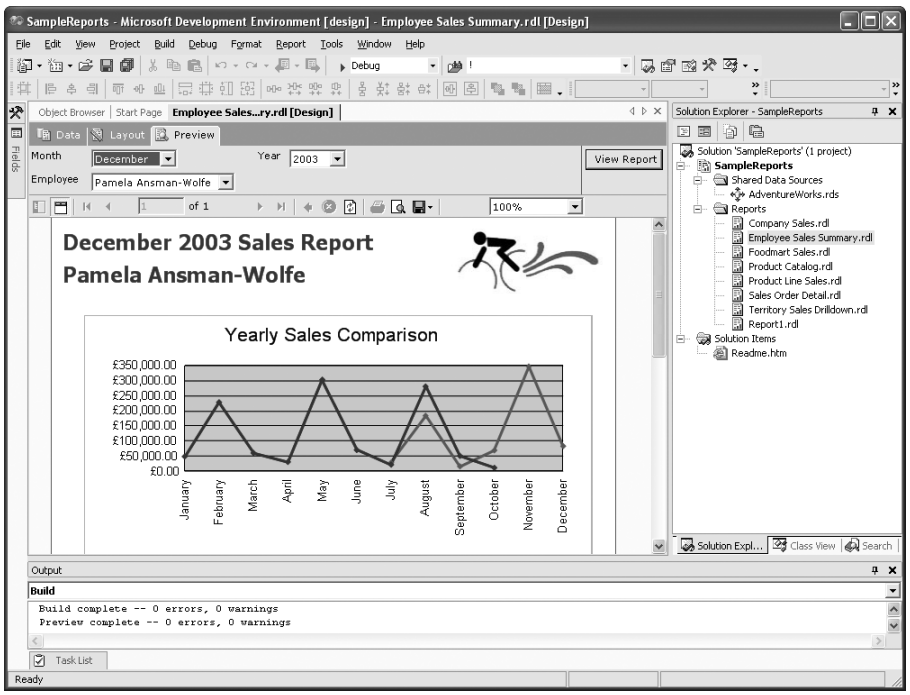

Figure 1.2 The Visual Studio .NET Report Designer Previewing a Report—Rendered in the UK

[7] Folks, we assume you know what most of the acronyms like IIS, ODBC, and ASP mean. If you don't, you're going to have considerable trouble with what we're talking about. We'll define the new ones, but the rest can be Googled pretty easily.

Define the Data Source

Usually, when creating a new report definition, you first wire up the report to its data source or sources. Data sources are typically SQL Server, Oracle, or other DBMS-style servers, but they can be anything exposed with a .NET Data Provider, or by ODBC or OLE DB. This means you can get report data from flat text files, spreadsheet files, Microsoft Access MDB files, or virtually any source of data. Sure, a report can extract information from several independent data sources, but if you need to `JOIN` data together from different data sources you'll need to do that through SQL Server's linked database facility.

Once you choose your data source, you need to create a report Data Source that connects to it and manages the credentials needed to access its data. Building a report's Data Source is very similar to creating an ADO.NET Connection string, and the Report Designer makes intuitive use of the very same Visual Studio .NET 2003 Data Link applet[8] dialog when creating report Data Sources. This applet assists in the validation and testing of the Connection string, driver choice, user credentials, and other driver-specific or provider-specific options. Sure, a report can have several Data Sources. A report's Data Source can also be created as a stand-alone object that can be shared among several reports (a "shared" Data Source), or alternatively it can be directly embedded and thus specific to a report. In Chapter 5, and again in Chapter 6, we'll discuss the security implications of embedded passwords in Data Sources and managed reports. Frankly, you won't see a single chapter that doesn't mention security in one context or another.

Create a DataSet

Typically, with a Data Source in place, you can create a query to populate a Reporting Services DataSet with data on which to base a report. So, as in ADO.NET, you need to create a `SELECT` query to return data from one or more tables (or views) or (better still) name a stored procedure and its parameters. One point to keep in mind: until the SQL Server 2005 version, the Reporting Services "DataSet"

[8] The Data Link applet is the dialog that appears when you create a new connection in the Server Explorer or double-click on a UDL file.

does not expose the same functionality as the ADO.NET DataSet. You can't "just" take an existing ADO.NET DataSet and pass it to a Reporting Services Report—at least not yet (unless you build a custom Data Processing extension, as we discuss in Chapter 13).

The Report Designer offers two ways to help build the SQL for your query: the default "generic" query designer, and the more familiar GUI-based Visual Design Tools' Query Builder. Figure 1.3 shows the generic query designer. It permits you to enter any ad hoc SQL query desired, including stored procedures—just change the command type on the toolbar over to stored procedure. To execute the SQL entered, click on the red ! icon. The generic designer might not be as easy to use as its GUI-based cousin, but it does not balk at complex queries as long as the syntax is understood by the target DBMS that the Data Source connects to.

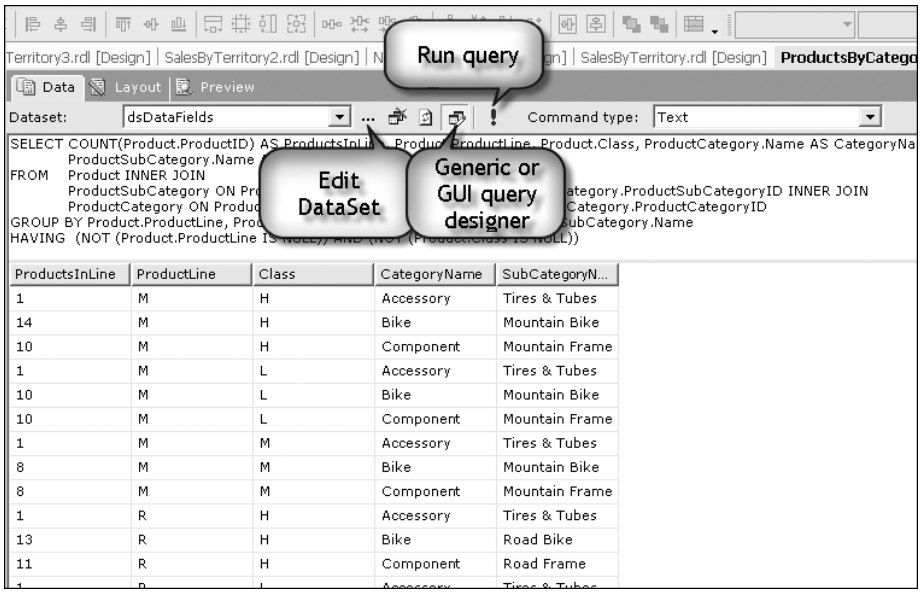

Figure 1.3 The Generic Query Designer

You toggle the designer of choice by clicking the  icon on the toolbar, as shown in Figure 1.3. The problem with this icon is that its tool tip always says "generic query designer" instead of toggling as it should. We expect this problem to disappear quietly in a Service Pack, although it didn't appear to be fixed in Service Pack 1.

The GUI-based query designer—which you may already be familiar with if you've used Access, Visual Basic 6.0, or Visual Studio .NET—makes it a breeze to create simple table-based DataSets (see Figure 1.4). Once you venture beyond the bounds of the GUI-based designer's ability to parse your query, you'll have to go back to the generic designer or resort to using stored procedures or views.

You get an *A* if you noticed we said DataSets, plural. Yes, a report definition can use any number of separate DataSet objects—each obtaining data from the DBMS to which they are connected through their respective Data Sources. And yes, their underlying queries can have input and output parameters, which can be bound to lookups, can be fixed values, can be strongly typed, can have defaults, can have configurable prompt strings, can be included in expressions, and much, much more.

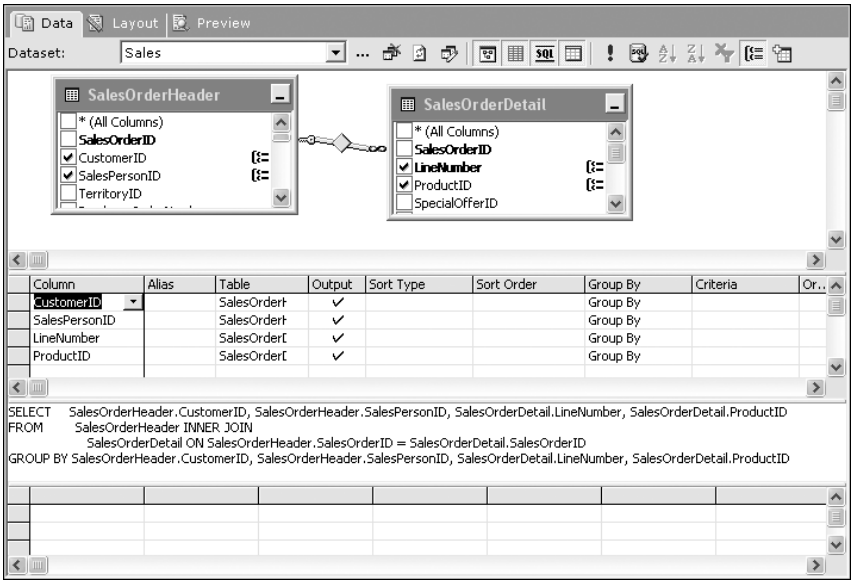

Figure 1.4 The GUI-Based Data Tools Query Designer

The cognoscenti[9] who read our last book, *ADO.NET Examples and Best Practices for C# Programmers*, might have thought that the DataSet we're talking about here is the *classic*[10] ADO.NET DataSet. However, the Reporting Services DataSet doesn't implement all the features you may have seen before in an ADO.NET DataSet; it implements only the streamlined subset of the features that the development team needed immediately. For the next version release of Reporting Services, we are confident that the DataSet element will be more fully implemented.

One of the features that isn't there at the moment is DataSet support for multiple resultsets. However, you can take several approaches if you want to work with multiple resultsets; for example, you might write a custom Data Processing extension or deal with this entirely on the server in a stored procedure by performing an ugly call out to a middleware component.

Visually Lay Out the Report Controls

The Visual Studio .NET Report Designer add-in provides nine report "controls" (called ***Report Items***) on the Toolbox palette (see Figure 1.5). The spec and BOL usually refer to these as controls, so we will too. Just understand that these "items" are not the same as the controls you use with other Visual Studio projects, and so you can't, at present, add custom controls to the toolbox and use them within a report definition. We understand that extensibility is a feature that's not too far off.

These report controls can be dragged and dropped from the Toolbox palette onto the report definition layout surface. (Behind the scenes, the report's RDL definition is synchronized as the report design is visually manipulated and configured.)

[9] cognoscenti (noun): A person with superior, usually specialized knowledge or highly refined taste; a connoisseur.

[10] Ha! DataSets have formally been around only since October 2001, and already we're calling them classic.

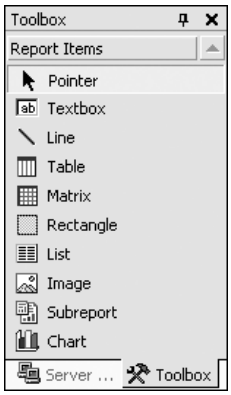

Figure 1.5 The Report Designer Toolbox Report Controls

The Layout tab is where you can most easily set the properties of your report's controls, just as you might do when designing a web or Windows form. Reporting Services permits you to place these controls virtually anywhere on the report. Be careful, though. Sizing the report can be tricky, as we'll discuss in Chapter 7.

Thankfully, there are enough report controls to build most types of report. For example, you'll find a Chart control that can build charts for every appetite, including line, column, area, and carborich (bar, pie, and doughnut), along with the scatter, bubble, and stock types, including colorful 3D and explosive effects. Yes, these are the same chart types you've feasted on for years. We dedicate all of Chapter 8 to the Chart control.

You'll also find ways to lay out your report data using tabular layouts. For example, Reporting Services includes a Table control (shown in Figure 1.6) that lets you merge cells, create and sort multiple grouping rows, and enable report interaction with compelling drilldown into other areas of the same report and drill through onto other reports. For example, if you wanted the rendered report to be filtered on a particular postal code, you'll find it's easy to create a dropdown list or simple dialog to enter this parameter at runtime, just before the report is rendered. That's because, just like programs, these reports can be interactive. Yes, we also said "merge cells," and, what's more, it's possible to embed charts within tables, or tables within tables.

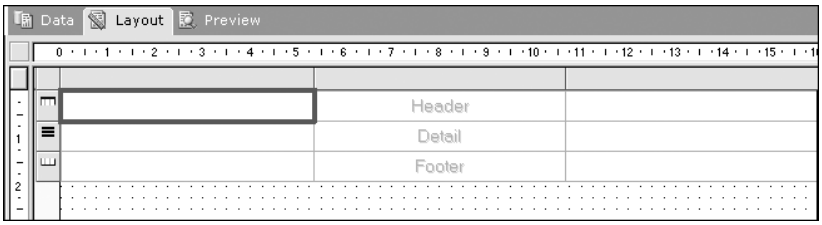

Figure 1.6 The Table Control in the Visual Studio .NET IDE

Another slick feature of the Visual Studio .NET 2003 Reporting Services Report Designer is that it supports dragging a Field into a cell in the Detail, Header, or Footer section of a Table control. When you do, the Designer does a lot of the binding legwork for you. For example, if you drag a Field into a cell in the Footer section, the Report Designer places the Field name in the Header cell and binds a sensible aggregate into the Footer cell; if it's not the aggregate you had in mind, you can always override it.

Oh, Fields! We forgot to mention that once a report has one or more DataSets, the Designer places their columns (AKA ***Fields***) into a palette indexed by a DataSet combo box (as shown in Figure 1.7). This makes it easy to drag and drop selected Fields directly onto the report definition, at which time the Report Designer automatically wires up the bindings for you—just as with the Table control. Nope, there's nothing preventing you from the drudgery of manually binding controls to DataSet Fields if that's what helps you sleep better (or if you're being paid by the hour).

Figure 1.7 The Fields Palette Window

There's also another tabular data layout report control: the Matrix control (see Figure 1.8). And no, it's not supplied or licensed from the Wachowski Brothers, although it does provide a somewhat transcendental interactive user experience to intuitively drill into columns and rows. And no, it does not show data in long, scrolling columns of unrecognizable green characters, unless Mr. Smith gets involved.[11] Chapter 8 discusses the Matrix control.

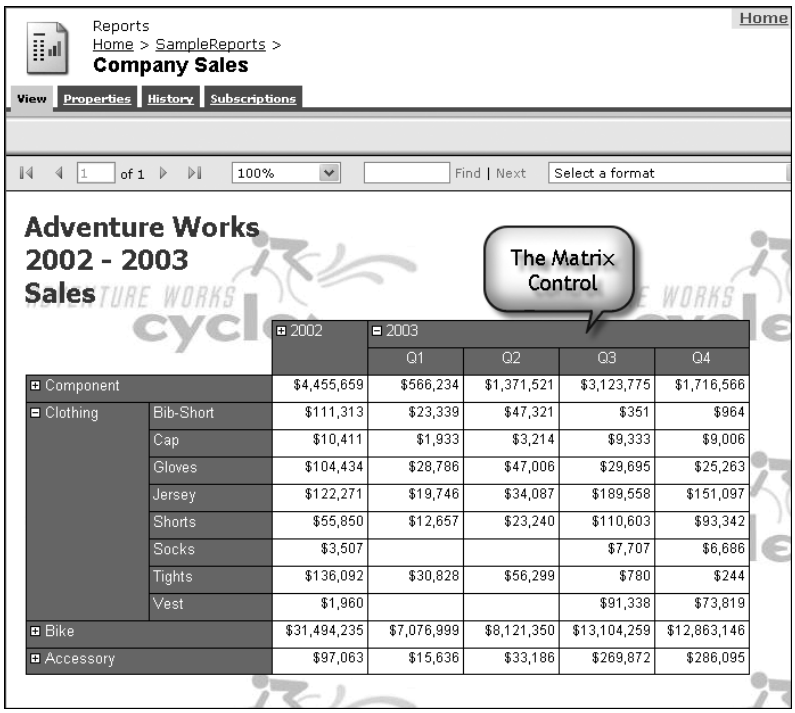

Figure 1.8 The Matrix Control Rendered in a Report

[11] Now, if you haven't heard of *The Matrix*, and the characters Neo, Trinity, and Mr. Smith mean nothing to you, then you have certainly led a geek-sheltered life and have missed out on a great trilogy of films. Take a look at http://whatisthematrix.warnerbros.com/.

We almost forgot to mention the most fundamental report controls from the Report Items toolbar because we simply took them for granted. As you know, no report would be complete without Line and Rectangle controls as well as data-bindable Textbox and Image controls.

In traditional report writers (such as Microsoft Access or Crystal Reports), you work with horizontal bands in the report that span the page and repeat for each row of data being shown. In contrast, Reporting Services uses a more Visual Basic forms-like approach. Here report controls can be hosted, or "parented," in data regions, which grow at runtime while other controls are automatically repositioned to make room. The advantage of this approach is that you can define multiple floating data regions, and not just above and below each other, as you do in banded reports. These data regions can appear virtually anywhere in the body of a report in any configuration (as shown in Figure 1.9). Think of the report definition as simply a container, or "form," for the parts of a report (the display controls) that sit between the report header and footer.

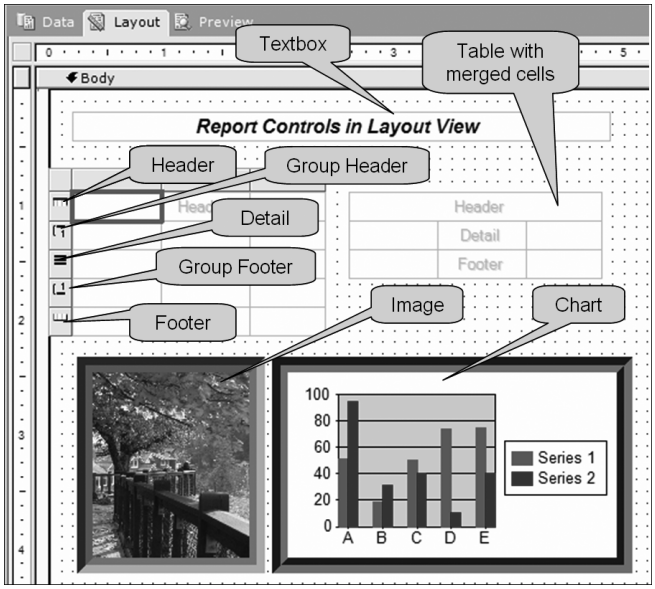

Figure 1.9 The Layout View with Two Side-by-Side Table Controls, an Image Control, and a Chart Control

As is to be expected, in Reporting Services, Report Designer can specify ***PageHeader*** and ***PageFooter*** sections in the report definition. These go at the top and bottom of report pages, and, naturally, a main ***Body section*** is positioned between the page header and page footer. Chapter 7 has a diagram (Figure 7.3) that shows how these areas are laid out. It's worth emphasizing that the body is not bindable to a DataSet, so the body can't repeat once for each record as the body does in Microsoft Access reports. Although this takes some folks by surprise, it's one of the most empowering features of Reporting Services.

We've mentioned the Table and Matrix tabular controls, and these certainly repeat for each record within themselves. However, not all reports call for tabular representations; many have a free-form repeating requirement, which Reporting Services provides via the List control. List controls can host other controls, and the List repeats once for each record in the DataSet to which it is bound.

In Reporting Services jargon, List, Table, and Matrix Report Items (controls) are ***data regions***. What makes the tool so powerful is that these data regions can be laid out side by side (as shown in Figure 1.9), or even defined hierarchically within one another. The latter feature supports data that you want to be independently sorted, grouped, and filtered at the data region level.

You can lay out reports that need snaking columns (like a newspaper) by setting the *Columns* property on the report definition to the number of columns you want. Reports can also contain ***Subreport*** controls, which are basically reports within reports—and the nesting can get several layers deep before choking. There's also a wizard that takes the tough work out of translating and importing reports from Microsoft Access. A minor disappointment for us is that there isn't any out-of-the-box wizardry to translate, import, and then flush Crystal Reports out of our systems. We'll be happy to discuss converting your Crystal Reports (and Access Reports). See the website 12 for details.

Advanced Features: Programmability in the Report Designer

What a report looks like on the screen (how it's ***rendered***) is controlled by a plethora of report definition and report control properties and

[12] See www.SQLReportingServices.NET.

the chosen Rendering extension. Although you can set many of these properties to permanent fixed values at design time, you can also set many of them to Visual Basic .NET expressions, which are evaluated when a managed report is prepared for rendering. It's also possible to make custom Visual Basic .NET code methods for use in any expression; to do this, you create Visual Basic .NET shared functions and paste them into the report's *Code* property. We show how to do that in Chapters 7 and 9.

This programmability makes it possible to do convoluted things, such as conditionally changing the text color in a Textbox control, but only when rendered by a specific user, only when the values are within certain ranges, or only when rendered on a Thursday afternoon between 3 and 4 PM by a certain user. This would be handy if you are a report programmer going into a review meeting with your boss at 3:30 and want to make an impact. Of course, the business logic possibilities are endless, and you'll probably have more serious requirements to implement!

No, you don't need to be jump qualified to be considered a "paraprogrammer."

–Bill

But it sure helps to be jump qualified if you are going to write poor queries for reports that take every cycle of CPU—so that when the DBA and supporting security officers show you the exit on the 42nd floor (the window) you can survive to bring another corporation's servers to its knees. So it's a good idea to either learn how to program properly or how to jump.

–Peter

Why use Visual Basic .NET and not C# (or both)? Well, behind the scenes a single .NET assembly is created for each managed report (using `System.CodeDom`). This assembly contains all the expressions and properties as property methods. It was a lot easier for Microsoft's developers to require that one and only one language assembly be compiled and used. Visual Basic .NET was chosen as being a slightly more user-friendly expression language, especially with paraprogrammers. By default, these expressions embedded in the report definition are given only Execute permission; this helps to prevent someone from creating a ***Trojan report*** by using an expression squirreled away that deletes files in the background, installs a virus, or bumps up your salary by 45 percent. By default, the Report Server does not permit access to the file system from custom code in expressions. Yes, a System Administrator might relax this, but this breach of the security dike would then apply to all expressions in all managed reports. We strongly advise against it, because there are other, more controlled ways to solve this problem.

What if you want to create some custom code that can be used by *all* reports? Well, the facility exists to create your own custom code class library assemblies in any Common Language Runtime (CLR) language, strongname them, reference them in the report

definition's *References Collection* property, instantiate any specific classes, and subsequently use the class methods, or shared functions, in report expressions. The System Administrator must install each of these assemblies and grant them specific permissions. These can be as restrictive as None, denying any ability to run; can be Execute only, which relaxes a little more to permit the code to run in a restricted sandbox; can be completely unrestricted Full Trust; or any combination of specific permissions. All this is achieved through code access security (CAS). If a System Administrator granted sufficient privileges to an assembly, a *Mission Impossible*–style report could be created that, when rendered, would promptly *fdisk* the system in 30 seconds. But such a scenario is enabled only if the System Administrator has permitted it by breaching the security seals and leaving the door open. In Chapter 9 we'll show in detail how to work with expressions and custom assemblies and how to assign strong-names, and set code access security permissions.

We suggest using the principle of least privilege—that is, grant only the permissions needed to do the immediate tasks and no others. Full Trust permission should not be given arbitrarily, and this requires administrators to use fine-grained permissions. For example, if an assembly needs permission to open a certain file, it should be granted permissions only to that file and not to the whole hard drive, nor should it be fully trusted.

Document Maps

For large reports that extend over many pages, it's possible to create a navigable hierarchical tree view called a ***Document Map***. Populating the nodes of this tree is as easy as putting a value into the *Label* property on any of the controls to be included in the Document Map (see Figure 1.10). This can be a fixed literal value, or, as just discussed, it can result from a calculated expression. If any labels have been set in any of the controls, the Document Map is deployed in a pane when the generated report is rendered in a browser. The *Label* property can also be bound to a DataSet Field. We'll show you this in action in Chapter 7.

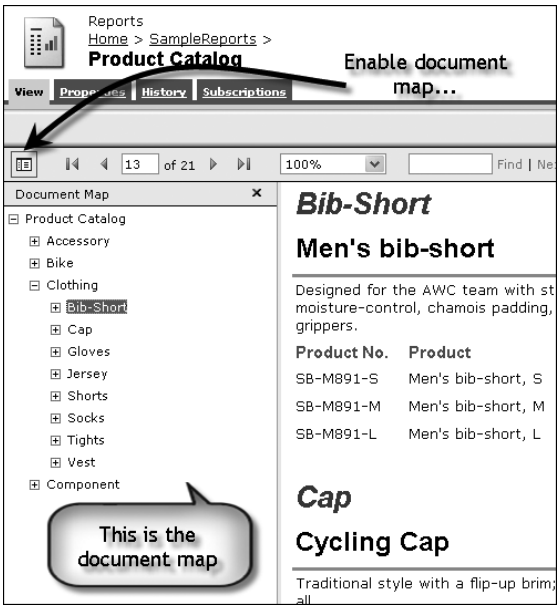

Figure 1.10 A Document Map and Report as Rendered in a Browser

Previewing in the Designer

After you have visually laid out the report controls on the report definition, the Report Designer lets you render and debug the report locally using data collected from the Data Source(s). When using the Preview tab in the designer as shown in Figure 1.11, the Report Designer does not access the Reporting Services server—it uses its own renderer. After you deploy your reports, recheck how the Reporting Services Rendering extension displays the report—we've noticed some subtle (and some not so subtle) differences.

TIP When you design reports, be sure to limit the number of rows to be used when previewing the report in the Report Designer. If you don't, there's a good chance you'll lock up Visual Studio .NET while it tries to render a 50,000-page report. In Chapter 3 we discuss how to limit the number of rows returned in a query, but while you're practicing, be sure to use a parameter-focused query or a `TOP` expression. If you really do want all 50,000 pages, don't forget to remove any row restriction before you deploy the report.

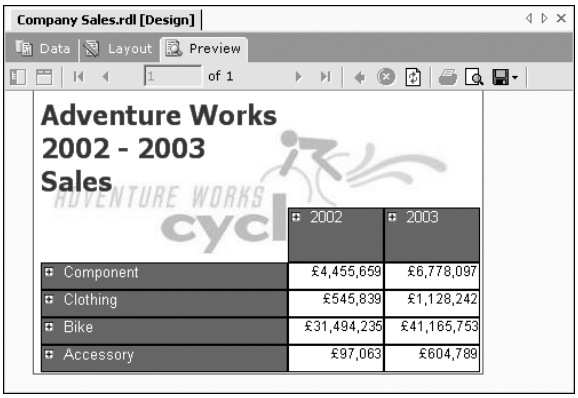

Figure 1.11 Previewing a Report in the Visual Studio .NET IDE

Previewing with the RSReportHost Utility

You can preview your report in the Report Designer Preview pane (as shown in Figure 1.11) by clicking the Preview tab in the Designer. This is great most of the time, but if you're developing with custom assemblies that you are also editing and redeploying, you *won't* want to use this technique. That's because the Visual Studio .NET process loads the custom assembly and doesn't unload it until you close down the Visual Studio .NET development environment. As a result, you won't be able to update the custom assembly you're working on until you've restarted Visual Studio. To address this issue, Microsoft provides the RSReportHost utility, which you call by setting the Solution Configuration to DebugLocal and selecting Debug on the project, or on a specific report. (Note that if you try this out with the Microsoft-supplied samples, there is not a "DebugLocal" Solution Configuration option, so you may have to create one—the crucial part is ensuring that the *TargetServerURL* property on the project remains blank. This ensures that when you try to debug, the RSReportHost starts instead. Figure 1.12 shows the Project properties page for the "Charting" project setup to use RSReportHost to preview the report design.

Figure 1.12 Setting the Report Project Properties for the "Charting" Report

When the RSReportHost utility starts, it loads itself and your custom assembly into a new process space, and it renders the report (as shown in Figure 1.13). When you close RSReportHost, that process space is unloaded, along with your custom assembly, so you'll then be able to work on any desired changes to your custom assembly.

Note that in rendered generated reports, currency formatting can adapt automatically to the current locale of the user browsing the report. That's the way that the Microsoft-supplied samples have been setup in the sample report definitions, but it is *not* the way we advise working. Guess where Figure 1.11 and Figure 1.13 were rendered? Unfortunately, these values are stored in the database using the currency datatype, not even as dollar amounts and not British pounds sterling. If you manage different types of currencies, we recommend that you explicitly embed any currency symbol directly into the Textbox control's *Format* property, possibly even based on the result of evaluating a report expression. If you are going to format a Textbox as "currency," placing a "c" in the Textbox's *Format* property (as Microsoft did in its samples) has the following effect: If the *Language* property is not set on that textbox, or it's set to "default," the report renderer defers to the *Language* property setting on the

report definition. If the *Language* property is not set or left as "default," the language of the user's web browser is used as the language from which the currency symbol and formatting would be used. Clear? Leaving the rendering choice of currency symbol to the browser could be problematic, to say the least—especially if the rendered report is an invoice!

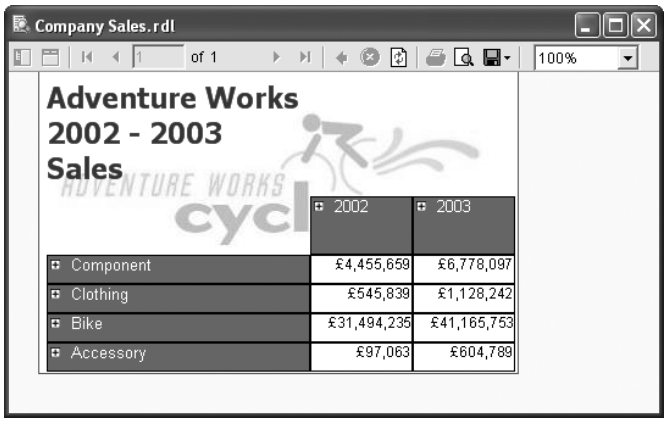

Figure 1.13 Previewing in the RSReportHost Utility

RDL: Report Definition Language

As we mentioned earlier, while you're designing a report, the Report Designer maintains the definition of the report in an "open" human-readable Report Definition Language (RDL) file (see Figure 1.14). RDL files are laid out in an XML format using an open published XSD schema. (Developers can indeed extend RDL by adding their own elements and attributes.) When a report definition is being edited visually in the Report Designer, the underlying RDL file is automatically kept synchronized. Sure, it's also possible to edit the RDL file directly, and those changes are immediately reflected in the Designer. Diehard gurus won't use the Report Designer; they'll use Visual Notepad, or even Edlin![13]

[13] Edlin, for the young whippersnappers, was an old command-line text editor. In fact, it still works in the command prompt shell.

The more serious corollary of an open RDL standard is that other report design tools can be built that extrude RDL or translate other proprietary reporting file formats to RDL. You'll then be able to verify the resulting RDL against the published XSD schema. You may even have ideas about enabling your own custom application to create or modify RDL files.

Figure 1.14 A Sample of RDL Code in a Report Definition

RDS: Report Data Source Files

Report Data Source (RDS) files provide a mechanism to share Data Sources on a Report Server scale. Like report Data Sources, ***shared Data Sources*** hold a Connection string and user credentials. These ***user credentials*** provide access to the underlying data—they're simply the User ID and Password or options that prompt the user for user name and password. These credentials determine whether the DBMS permits your report's DataSets to access the database and further grant or deny access to the underlying database tables, views, functions, or stored procedures. Shared Data Sources can be created in the Report Designer (in Visual Studio .NET), in the Report Manager, or programmatically through the SOAP interface (as we describe in Chapter 9¾). However, the Report Designer only

permits a restricted subset of Report Data Source features to be configured. There is much greater capability (albeit a more awkward interface) when configuring deployed report-specific or shared Data Sources with the Report Manager.

There are a number of mechanisms you can use to specify, manipulate, and persist connection credentials. In Chapter 3 we show you how to create a shared Data Source using the Report Designer wizard, and in Chapter 4 we illustrate how to manipulate Data Sources in the Report Manager. Chapter 5 discusses serious security issues in a deployed production environment, and Chapters 6 and 7 discuss the use of Data Sources in the development environment. After reading these chapters, you'll see that each credential persistence approach has far-reaching consequences. To ensure that you don't leave the door open to Trojan reports or other security risks, you should get a good handle on this information before launching your first production environment.

The Report Server

After you've used the Report Designer to design and build a report definition or report project (which might contain any number of report definitions, shared Data Sources, and resources such as graphic files), you deploy it by sending its RDL (or RDS if a shared Data Source, or the resource file if a graphic) to the ASP.NET-based ***Report Server***. Before attempting to deploy a report, it's crucial to configure two properties on each report project in Visual Studio .NET 2003: the target URL of the Report Server, and the notional folder to which you wish to deploy. You'll also want to reference an appropriate StartItem report, as shown in Figure 1.15.

To deploy a project or a single report within a project, right-click the project or report name and choose Deploy. Once a report definition is complete and deployed to the Report Server, it becomes a ***managed report***.

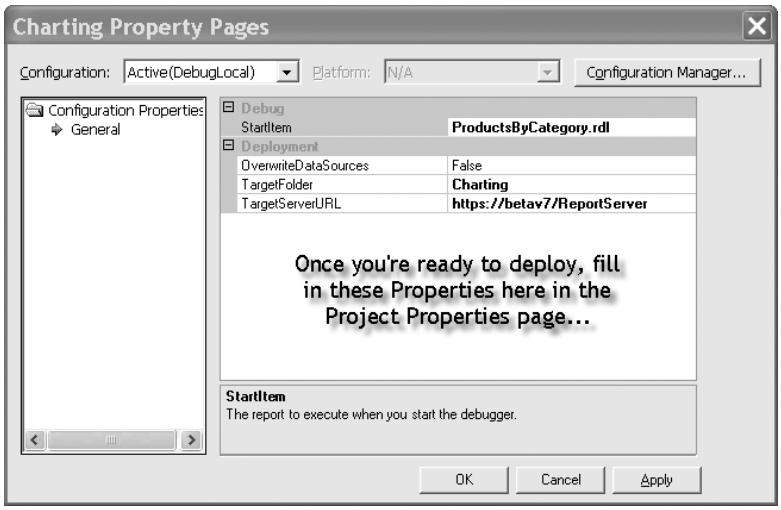

Figure 1.15 The Project Properties Page—Ready for Deployment

I say "XML-like" because although some of these files look like XML, if you try to put in XML comments—for example, `<!--A Comment-->`—this can choke SQL Server, and the exception messages can be pretty cryptic. We'll discuss these configuration files later in this chapter.
–Peter

The Report Server exposes Reporting Services' SOAP endpoint, and Visual Studio .NET deploys the report and shared Data Sources and resource files by simply making calls to SOAP methods. All these report definitions, shared Data Sources, and resource files are organized in the Report Server in a logical folder hierarchy, over which is an extensive role-based security system integrated with Computer/Domain Users and Groups.

If you're developing your own tools or being paid by the hour and you want to build your own .NET application to deploy reports, you can call the SOAP interface from your own programs. We'll show you how in Chapter 9¾.

The Report Server uses a SQL Server 2000 database to retain almost all of its metadata. In fact, the only modifiable metadata retained outside SQL Server are any custom assemblies and supporting files you may have written that the report references, along with a handful of XML-like configuration files for the web and Windows services and security policies.

SQL Server and SOAP

Although Reporting Services uses SQL Server as its own data store and you can get at the Reporting Services' own tables and stored procedures, we should warn you that the *only* officially supported interface is SOAP. The reason is that as the product progresses, Microsoft might (is likely to) change/tune/alter/repair/enhance/refine the underlying schema of the Reporting Services database and the definitions of the stored procedures. Microsoft has confirmed that the SOAP interfaces will be backwardly supported but says that the use of SQL Server by Reporting Services for its data store is a (publicly undocumented) implementation detail subject to change. So if you feel inclined to fiddle directly within the Reporting Services ReportServer database with T-SQL, be warned that a next version or Service Pack could (and probably will) break your code. This is a little frustrating because calling out to a SOAP interface from T-SQL in SQL Server 2000 is not as trivial as it will be in the next version of SQL Server (SQL Server 2005).

Rendering Extensions

In addition to SOAP-type HTTP and HTTPS calls, the Report Server's ASP.NET web server component dutifully responds to specially formatted URL Request Strings by streaming requested reports over HTTP or HTTPS in an amazing number of rendering formats. The rendering formats that a Report Server can generate are governed by the Rendering extension assemblies installed and registered on the system. Out of the box there are six Rendering extensions: HTML (3.2, 4.0, MHTML), XML, CSV, IMAGE (BMP, EMF, GIF, JPEG, PNG, TIFF), EXCEL, and PDF.[14]

Stop and think for a moment about where we are. At this point, we have the ability to design and render a report in a variety of formats, which can target a number of potential platforms and devices. We should also emphasize that there is no additional separate license fee to pay to Adobe for the PDF report generation, and there is a way to extract an HTML table fragment stream, which you can

[14] Sometimes we assume too much—as in we assume you know what these acronyms mean. If you don't, try this website: `www.geocities.com/ikind_babel/babel/babel.html`. It defines all of them.

incorporate directly into one of your own web pages without having to use an IFrame control.[15]

Does anyone still have a WAP phone? Do yourself a favor and get a Smartphone or PocketPC phone. I was one of the first to use WAP technology and blew a load of dough writing WML servers and translators. Thanks, Nokia, for nothing!
–Peter

But as if that were not enough, in true TV shopping style—"Wait! There's more!"—you also get the ability to create and register any of your own custom Rendering extensions. So, for example, if you have a commercial requirement that your reports be rendered on Wireless Application Protocol (WAP)-enabled phones, you can write a .NET Rendering extension assembly for Reporting Services to do so. But be warned—developing Rendering extensions is hard, at least in our opinion. Configuring the Report Server to use a new Rendering extension is as simple as placing your custom rendering assembly in the right folder and delicately editing a few XML configuration files. We suggest you investigate the existing Rendering extensions, such as using Extensible Stylesheet Language Transformations (XSLT) with an XML-rendered report. This might satisfy your requirements without the need to write a new extension.

Delivery Extensions

So far we've talked about being able to extract a rendered report stream in response to a specially formatted URL, but the Report Server need not always be shy and passive. Reporting Services does have the ability to break out of its box and start spamming preconfigured and presumably consenting recipients over SMTP[16] at preset scheduled times or only when the data changes.

And if benevolent spam is not enough, the Report Server can even proactively save rendered reports onto a UNC[17] file share on a certain schedule or when data changes. Yes, it even takes care of any credentials it might need to use to access that share, and it can set access credentials on the saved file to restrict access to a certain computer or domain user or group. It can also be configured to overwrite any previous file or to incrementally number files so as not to overwrite any previous file.

---

[15] An IFrame is a pesky little web control that enables you to host one web page inside another. Not all web browsers support IFrames.

[16] SMTP stands for Simple Mail Transfer Protocol (e-mail).

[17] UNC stands for Universal Naming Convention. These are shares that start with `\\<Server>\<share>`.

Both delivery features are provided through what are called ***Delivery extensions***, and yes, you can write your own .NET Delivery extension assembly to proactively deliver reports elsewhere. For example, you can send to an FTP[18] service, post it using NNTP[19] to a newsgroup, or dispatch an SMS message to a WAP-enabled phone, with the embedded link pointing to where you use your WAP Rendering extension. The extensible delivery possibilities are endless, and they are, for the most part, much easier to develop than Rendering extension assemblies.

What About Printing Delivery Extensions?

We have, however, discovered a small fly in the ointment. In the initial release version there isn't an out-of-the-box Delivery extension to send reports directly to a printer. Thankfully, the Microsoft development team has not left us completely high and dry; they provide project code samples in C# and Visual Basic .NET that show how to make a Delivery extension that targets a printer.

If you stop to think for a moment about how many printers there are in the world and how many different printer drivers there are to take advantage of the plethora of hardware-specific configurable options—such as duplex printing, paper size selection, input and output trays, color printing, margin settings, collation, stapling, binding, faxing, scaling, even the number of copies that should be printed—it would have been a mammoth undertaking for the development team to have created a generic printer Delivery extension that would cater to every possible option, and time constraints pushed this part of the development out to the next version. Our guess is that printer manufacturers and third-party ISVs[20] will write their own printer Delivery extensions that take advantage of their hardware's features to make them fully available to Reporting Services. The Delivery extension sample provided by the Reporting Services developers works by rendering the report as an image and sending that to the printer. This is certainly not the most efficient way to solve the problem—being more

[18] File Transfer Protocol
[19] Network News Transport Protocol
[20] Independent software vendors

"crude but effective," as Seven of Nine might say.[21] All of this said, we are working on a server-side printing extension that can be called directly from the Report Manager when viewing a report, so be sure to check our website www.SQLReportingServices.NET for news.

The SQL Server Agent and ReportingServicesService.exe Windows Service

The Report Server is not alone when it comes to its ability to push report content through Delivery extensions. It's aided and abetted by two friends: the SQL Server Agent and another Windows system service that you'll see in your task manager as the tautological ReportingServicesService.exe, which is a .NET Windows system service. These provide an infrastructure that enables possible load balancing in a web farm environment when you're pushing scheduled content out via Delivery extensions. (For more on web farms, see Chapter 2.)

While the SQL Server Agent creates entries in the ReportServer database's Event table at scheduled times, it's the ReportingServicesService.exe Windows system service that polls the database every 10 seconds to see if there are any entries in this table and takes appropriate action. Chapter 2 tells you how you can configure the polling interval.

The Report Processor

The ***Report Processor*** is the thoroughbred workhorse[22] of the Reporting Services team (see Figure 1.16). It's only used to process managed (deployed) reports. It's responsible for querying data, assembling the report into the Intermediate Format (IF) Binary Large Object (BLOB), and extruding the report to the client. This is where the various extensions come into play. For example, a Data Processing extension is used during report processing to query the report's Data Sources. Data and layout are combined by the Report

[21] Don't know what the ASP.NET guru Doug Seven would say, though.

[22] It's not a donkey or obstinate ass that's difficult to get to do what you want.

Processor and saved to the database in the IF. Only at this stage does the Rendering extension come into play. The benefit of this approach is that all the querying and processing work is done once, after which different renderers can be called—all of which can work from the same IF BLOB. This Intermediate Format can be persisted behind the scenes in the ReportServer database to enable reports to be rendered from cached versions, historic versions, or Snapshots. The fundamental difference between immediate in-demand, cached Snapshots and history is simply where and how long a generated report is stored and managed. (You'll learn more about report history and Snapshots shortly.) The key to the Intermediate Format is that all the heavy lifting is done once, and that it's device (renderer) independent.

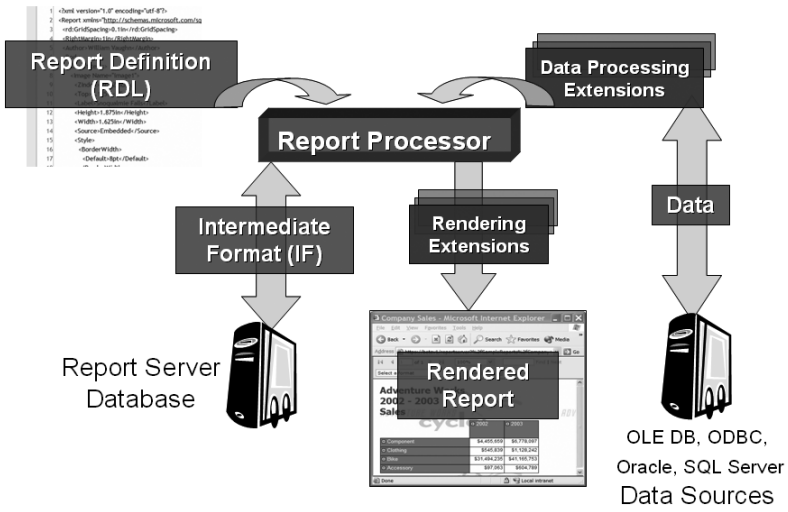

Figure 1.16 The Report Processor

Caching Intermediate Format

To obtain some performance gains and scalability, the Report Server can cache the generated report IF in a SQL table within the Report Server database—assuming that the credentials used in the report's Data Sources are not derived from Windows Authentication security

(SSPI) or by prompting the user. If caching is enabled on a managed report, the report can be delivered much more quickly after it has been requested the first time because the Report Server doesn't have to query all the Data Sources again; it need only render the IF into the requested stream type. Reports can be cached for a configurable number of integer minutes to a maximum of 2,147,483,647—approximately 4,083 years, by which stage we doubt you'll still be bothered (assuming your system stays up that long). For a higher degree of control, the cache can be expired on a report-specific schedule or on a schedule shared with other objects.

For parameterized cache-enabled reports, the Report Server creates IF images and caches the report each time it's requested by means of parameter values that have not already been used, compiled, and cached. In other words, the cache is uniquely indexed on a report and on the parameter values with which it was executed. Reports remain in the cache until they expire or are explicitly flushed from the cache.

Caching Snapshots

The first time a report is executed, it can take a very long time, especially if a lot of number crunching is required in the report's queries or if you asked for all 50,000 customers. Microsoft has provided an alternative to deal with this contingency: the ability to populate the cache with an Intermediate Format Snapshot from a report executed at a scheduled time. Again, the schedule can be specific to a report or can be shared with other objects and must use hard-coded credentials. This approach is designed to support preparation of reports in off-peak hours.

Caching History

The Report Server can maintain a historic archive of Intermediate Format report Snapshots that can be repeatedly rendered at any time in the future—assuming that the system is still up when it's time to render the report. This ***history*** can be unlimited—or, more accurately, limited—by what SQL Server can provide in storage space. The Report Server can generate Snapshot reports on shared schedules or on a report-specific schedule.

The Report Execution Timeout Governor

Preparing the Intermediate Format for reports can take considerable time, especially if there is a lot of data to collect or if complex processing is required. Fortunately, the Report Server provides the option to time out the execution of a report and cancels it with a *polite*[23] message to the user. This Report Execution Timeout setting is set in seconds, and there is a default for the whole Report Server; this default can be overridden on a report-by-report basis.

The default timeout is close to 60 seconds. The Report Server evaluates running jobs at 60-second intervals. Every 60 seconds, the server compares actual process time against the report execution timeout value. If the processing time for a report exceeds the report execution timeout value, report processing stops. If you specify a timeout value that is less than 60 seconds, the report may execute in full if processing starts and ends during the quiet part of the cycle when the Report Server is not evaluating running jobs. For example, if you set a timeout value of 10 seconds for a report that takes 20 seconds to run, the report will process in full if report execution starts early in the 60-second cycle. You can set the *RunningRequestsDbCycle* setting in the RSReportServer.config file to change how often running jobs are evaluated.

The Report Manager

So far we've given an overview of designing reports and deployment to a Report Server. We've said little about how these deployed reports can be managed while in the Report Server, nor how a user might choose a report to run. Well, to do this Microsoft provides a Report Manager ASP.NET web application. The Report Manager provides an attractive interface that acts as a wrapper to the URL calls and SOAP methods of the Report Server.

[23] Perhaps a politically correct translation of what the DBA would really like to say to users who execute queries that bring their 8-way processor/3-terabyte RAM system to its knees.

The Report Manager can be accessed on its default URL of http/https://<server>/Reports. What users see rendered in their web browsers depends on the security permissions they have been assigned; reassuringly, users can perform administrative functions *only* if they have Administrative privileges. In Chapter 4 we'll discuss Report Manager security and explain how to setup Administrative and User account roles. The Report Manager really comes into its own as a user application that enables users to choose from and execute the managed reports that they have permissions to access.

Managing Security Permissions

The Reporting Services security permissions system is extensive and is based on assigning ***roles*** to Windows domain, local machine user, or group accounts. Here's how it works. At the lowest level of granularity, the various ***tasks*** that can be performed are identified, and these tasks can be aggregated into various defined roles by an Administrator or someone with appropriate privileges.

Out of the box there are four defined roles—Browser, Content Manager, My Reports, and Publisher—providing permissions very much along the lines of what their names imply. Because these roles are constructed from a selectable list of task operations that can be performed, it is possible to define new roles consisting of different subsets of tasks. Keep in mind that by default ordinary users have no rights; only those who have Administrator rights can access these roles unless you modify the permissions for your selected accounts.

Managing Folders

As mentioned earlier, the Report Server maintains managed reports and shared Data Sources in notional folders, and the Report Manager lets users navigate these folders. Depending on security permissions, the person browsing can create new folders and can delete, move, hide, rename, and edit the descriptions of folders or reports. Of course, if the person browsing doesn't have permission to see a report or folder, then it isn't visible when the *user* views the Report Manager. In addition, it is possible to set permissions for

users and configure the Report Server to maintain folders on a per-user basis on a model similar to the My Documents folder on the Windows file system. This model enables users to have their own My Reports folders for their own reports (independent of other users) that they alone can see.

Managing Shared Data Sources

Subject to permissions, any shared Data Sources can be managed by a user through the Report Manager. Users can change credentials and Connection strings in the Data Sources on a live Report Server without invoking a development tool.

Managing Linked Reports

One of the cool Report Server features that the Report Manager exposes is the ability to create, name, delete, and manage ***linked reports***. This is a report description that doesn't have any RDL of its own. The concept here is very similar to the way a shortcut operates in the Windows file system. The Report Manager enables linked reports to have different default values, different visibility for parameters, and different security than those of the underlying report to which they are linked, and it provides a placeholder for a user to execute. Alas, for a linked report, it is not possible to establish Data Sources or credentials that differ from the ones used by the real underlying report. Now that *would* be interesting.

Managing Shared Schedules

We have referred to schedules several times in different contexts. The Report Manager provides a mechanism for creating system-wide schedules that can be shared wherever a schedule can be used and can be managed from a central point.

Managing the Cache, Snapshots, History, and Report Execution Timeout

When we discussed the Report Server, we mentioned caching Snapshots and creating historic Snapshots. Yes, you can use the Report Server SOAP interface to configure these, but you'll probably

be relieved to know that the Report Manager provides a UI wrapper around the SOAP interface to let you easily configure these and many other features without needing to resort to programming.

At the site level, someone with Administrator privileges can create defaults using the Report Manager to set the maximum execution time in seconds that a server should spend rendering each report before it decides to abandon it and return a message to the user. This site default can also be overridden on each report.

Subscriptions

Users with the requisite permissions can ***subscribe*** to managed reports using the Report Manager. This means that selected users can configure a managed report to be delivered to them via one of the installed Delivery extensions (such as Report Server File Share) or by e-mail on a personal or a shared schedule. With additional permissions a user can configure a ***data-driven subscription***, where configuration information and distribution lists can be derived from a query on a table.

Help

All the pages that constitute the Report Manager have context URL links to HTML help pages. If the Report Manager is going to be deployed in the default location where users select and execute reports, it is reassuring to know that an out-of-the-box help system is provided. If you are going to build your own report management system, you'll need to budget time to explain how it works and possibly provide your own online help system.

Extending the Report Manager

The Report Manager is intended to be a closed, black-box solution. Because of this Microsoft has not released the source code. In our opinion, the Report Manager is a delightful application, but there are some rough edges that you may find that you want to smooth out. In this respect, not all the settings that one might like to adjust from the Report Manager are provided. Although some of these may be added at a later stage, at the moment a System Administrator may have to delicately futz around with the XML-like configuration files

or find a custom application that does it. Another irritation that comes to mind is that there is very little control over the layout of the ASP.NET controls that are used to collect the parameters in a report.

Microsoft's official position is that you should make your own Report Manager if you don't like the way the Report Manager works. To some extent, Microsoft helps by providing some sample starter applications. The problem is that it's a heck of a lot of work to build all the functionality that the Report Manager application provides when you want to add only a few bells and whistles.

Well, because we are not on the red pill we spoke of earlier, we can be mischievous and tell you how to pry open the Report Manager black box and provide the degree of customization you might want. This certainly takes much less effort than creating a whole Report Manager application. Of course this approach is hostage to Microsoft changing things, as it is at liberty to do in the future. However, the approach we'll show you in Chapter 11 enables you to make easy modifications now, and if Microsoft comes back and breaks your toys later (as we told you it might), at least you'll be able to incorporate them into your own Report Manager should you be forced to build one.

Versions of Reporting Services

There are four versions of Reporting Services to choose from: the Standard Edition, the Enterprise Edition, the Developer Edition, and the Evaluation Edition. Except for the Standard Edition, all provide the same feature set and differ only in EULA[24] license terms and time-bomb duration before the code stops working:

- The Evaluation Edition stops working after 120 days and is licensed only for evaluating the product.
- The Developer Edition differs from the Enterprise Edition only in that it's licensed for development but not for production use.

[24] End-User License Agreement

- The Standard Edition and the Enterprise Edition are the only editions licensed for production use and for using the features described in this chapter.
- The Standard Edition is limited in that it does not support web farms, custom security extensions, report history, and data-driven subscriptions, and it's limited to not more than four processors and 2 GB of RAM.

Summary

This chapter gives you a brief tour of Reporting Services and enough detail to get you started toward a better understanding of the technology. Reporting Services runs on an IIS system and requires a SQL Server 2000 system to manage its databases. Visual Studio .NET plays a role as the host for the Report Designer add-in, and it can be used to build Data Sources, queries, and report definitions and to deploy them. The underlying Report Server and its Data Processing and Rendering extensions help capture report data from a variety of data sources and publish managed reports in a variety of formats. Next, we'll show you how to install Reporting Services.

NOTE Microsoft recommends that you avoid use of the underscore character in computer names as the Report Server does not persist the session state information on these systems—unless they have been patched with Internet Explorer Patch MS01-055. See "Preparing to Install" in the Reporting Services documentation for more information.[25]

[25] `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/RSinstall/htm/gs_installingrs_v1_8k82.asp`