Excel Services

Develop A Calculation Engine For Your Apps

Vishwas Lele and Pyush Kumar

This article discusses:

- Excel as a server-based application
- The Excel Services architecture and APIs
- Creating managed user-defined functions
- Building custom solutions with Excel Services

This article uses the following technologies:

Excel Services

Code download available at: ExcelServices2007_08.exe (226KB)

Contents

O
rganizations use Microsoft® Excel® to perform complex calculations and to visualize information using charts, pivot tables, and the like, and to perform many other custom tasks. But in the past, if you wanted to implement a calculation engine, you needed to enlist the services of a developer who would use algorithms provided by your business analysts to design the code. Now, with the Excel Services technology in Office SharePoint® Server 2007, business analysts themselves can implement the calculation engine formulas they need, reducing the cost of implementation and making maintenance of the calculation algorithms easier than before. In addition, with Excel Services the custom algorithms in an Excel workbook can run on a Web server, allowing users to access them remotely. As you might imagine, this means many more users can take advantage of the software from many more locations.

Excel Services Architecture

Let's see how the Excel Services architecture enables such flexibility. Excel Services consists of three tiers—a Web front end, an application server, and a database (see Figure 1). The SharePoint content database forms the database tier. To enable the server-side Excel behavior, you place the workbook at a trusted SharePoint location or on a network file share. Some functionality (such as additional security features) is only available through SharePoint.

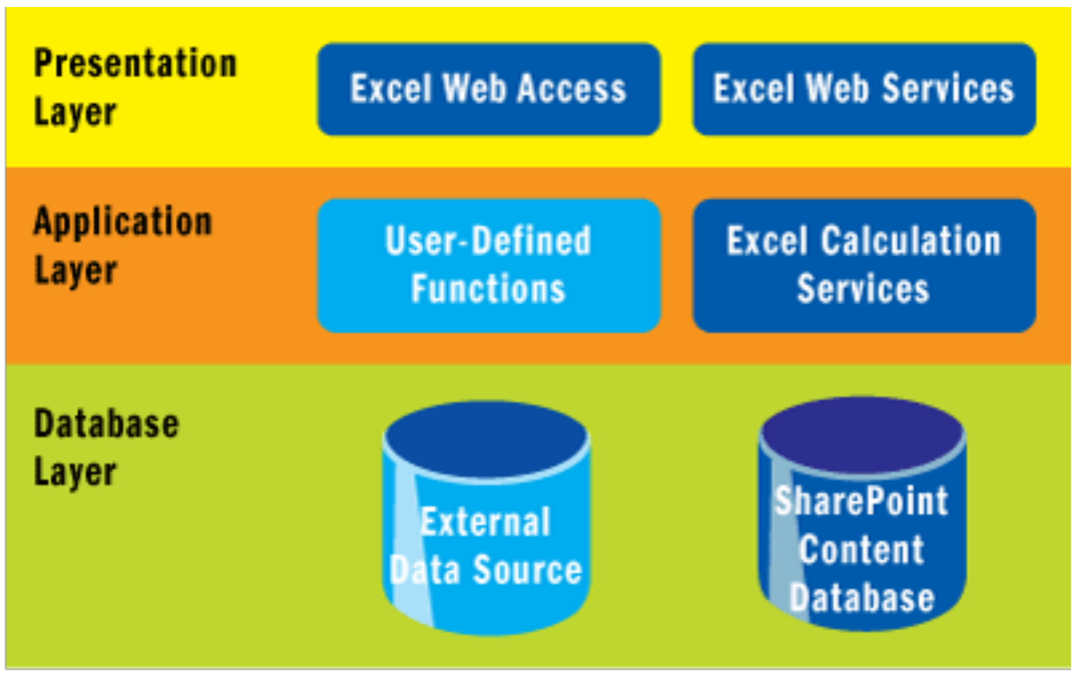


Figure 1 Excel Web Services Architecture

The application server consists of Excel Calculation Services, which is responsible for loading a given workbook and performing any required calculations. Workbook instances can be connected to external data sources.

The Web front end is responsible for rendering the relevant portions of the workbook in HTML via a SharePoint Web Part. The Web front end is also responsible for exposing the Web service endpoints that allow remote access to the Excel Calculation Services.

An important aspect of the Excel Services architecture is that it is integrated with SharePoint 2007. As noted earlier, to enable some of the server-side behavior the workbook needs to be stored inside a SharePoint content database. This makes it possible to take advantage of SharePoint content management features such as versioning, check-in/check-out, and security roles and permissions in the context of Excel workbooks.

Similarly, the Excel Calculation Services is based on the SharePoint Shared Service Provider (SSP) model. SSP is a mechanism for packaging SharePoint functionality as a service that makes it easy to administer and use across different sites. As a result, it is possible to reuse an Excel Calculation Service instance across SharePoint sites, as well as manage it via a SharePoint administration site.

You should also note that Excel Services imposes some restrictions over Excel. Macros and unmanaged code-based add-ins, such as Visual Basic for Applications (VBA) code are not supported by Excel Services. Instead, Excel Services supports managed server-side user-defined functions (UDF), an interface that allows invocation of custom calculations from inside a server-side workbook.

Later in this article, we will look at a UDF code example. The restriction related to unmanaged code add-ins can be overcome by building a managed UDF to wrap unmanaged code. The restriction on the use of VBA and macros is hard, but it may be a boon in disguise as it prevents the server-side calculation logic from becoming unwieldy.

The Excel Services API

Now let’s look at the Web service-based Excel Services API used to interact with a server-side workbook, using the code snippet in Figure 2 as the basis for the discussion that follows. Note that code has been elided for clarity.

To begin, we need to make the workbook accessible to the clients. Any workbook saved to a location on the server can be accessed through the Excel Service API, a part of Microsoft.Office.Excel.Server.WebServices. The Excel 2007 client makes publishing the workbook easier through publish functionality. The benefit of using the publish mechanism is that you can control which parts of a workbook (sheets, views, pivot tables, and so on) are accessible via the Excel Services API. The primary class in the API is the ExcelService class, shown in Figure 2. This class represents an in-memory, server-side instance of a workbook. To enable multiple users to interact concurrently with a workbook, a session-based access model has been implemented. Each user opens a separate session with a workbook using the OpenWorkbook method of the ExcelService class. The OpenWorkbook method returns a unique session ID associated with the opened session. This session identifier needs to be supplied when invoking any subsequent methods to interact with the opened workbook. To set a named range inside a workbook, you can use the RangeCoordinates class to define the boundaries of the named range. The SetRange takes RangeCoordinates and the corresponding array containing values to be passed in as parameters. A variation of the SetRange method is the SetRangeA1 method, which uses Excel range specification "A1" instead of the range coordinates used by SetRange. Once all the required range values have been specified, CalculateWorkbook can be invoked to force the workbook to compute the formulas. It is possible to cancel the most recent CalculateWorkbook method by invoking a CancelRequest method. You can use the GetRange method to obtain calculated values from a range in the open workbook. Once all the calculated values have been retrieved, you close the workbook session using the CloseWorkbook method.

Excel Services can be extended by adding UDFs, which are accessible as cell formulas similar to the built-in Excel functions. To create a UDF, you need to create a Microsoft® .NET Framework assembly that contains at least one class that is marked with the UdfClassAttribute and at least one method marked with the UdfMethodAttribute. Please refer to the code snippet below. Here we define ConvertToUpper as a UDF method. After appropriately registering the UDF, the ConvertToUpper function can be inside an Excel Services workbook instance:

```
using Microsoft.Office.Excel.Server.Udf;

[UdfClass]
public class Util
{
    [UdfMethod]
    public string ConvertToUpper(string name)
    {
        return name.ToUpper();
    }
}
```

We've covered only a small portion of the Excel Services API here. For additional details, refer to the MSDN®documentation.

A Custom Solution with Excel Services

The primary motivation for developing a custom solution is to allow business analysts to author calculations (such as financial models) directly as Excel formulas. Until now, business analysts have mostly relied on documenting algorithms as pseudocode in text. The pseudocode was then translated into code by developers. Using Excel Services, we were able to overcome some of the limitations inherent in the process of creating formulas in Excel and, as a result, eliminated the need for developers to convert the pseudocode into real code.

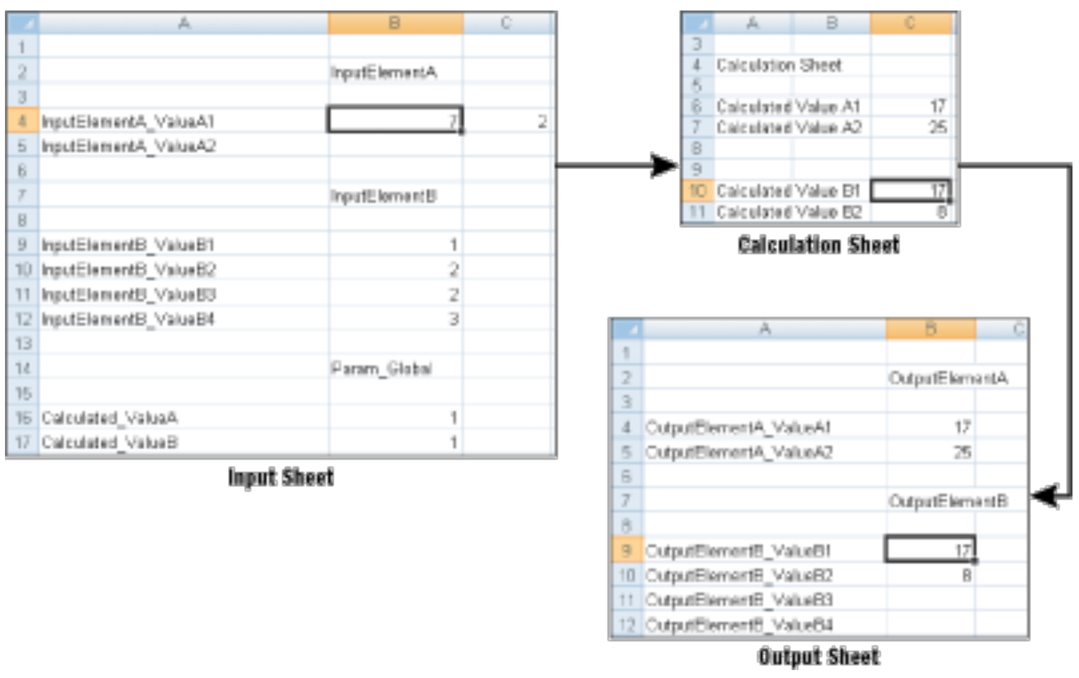
The key challenge in allowing non-developers to author calculation logic was to strike a balance between flexibility and ease of authoring, with the ability to enforce a robust structure. To provide the structure, we needed a way to define an input and output "interface" that represents the data contract for a calculation algorithm. Business analysts would be limited to the named ranges that are part of the data contract for flowing data in and out of the calculation instance.

The obvious choice was to define the data contract using named cells or ranges constructs within Excel. Named ranges not only accord the required level of granularity for authoring calculations within Excel, they are also the fundamental data structure on which Excel Services API methods such as SetRange and GetRange are based. However, the challenge with using named ranges is that there is no standard format or language, such as XML Schema Definition (XSD) or Web Services Description Language (WSDL), to define the interface. Moreover, named ranges (and consequently Excel Service API methods) are inherently type unsafe. For instance, there is no way to enforce datatype checking on a given named range. Finally, there is no built-in way to enforce the contract across the calculations (inside Excel) and the Excel Services client program.

To overcome these limitations, we developed a custom two-part solution. The first part is an Excel pre-compiler designed to generate the named ranges based on a defined interface. The second part is a generic Excel Web service client that invokes the calculation inside a workbook, while adhering to the interface.

The Excel Pre-Compiler

XML Schema Definition with all its semantic richness and simplicity, seemed the ideal choice for defining the interface. We decided to use the XML Schema constructs to define the input and output contracts. Next, we needed a way to translate the XML Schema into named ranges. We first looked at the possibility of using the XMLMap feature introduced with Excel 2003. XMLMap allows cells inside Excel to be mapped to the elements of an imported XML Schema. Unfortunately, the XMLMap capability is not available to Excel Services. So the alternative was to create the named ranges inside an Excel workbook. We developed a pre-compiler component that generated a template workbook with the required named ranges based on the schema. The generated template workbook has three sheets—one each for input, output, and calculation. The input sheet contains named ranges that correspond to the input for the calculation. Similarly, the output sheet contains named ranges that correspond to the output of the calculation, and the calculation sheet is where the calculations are placed (see Figure 3).



	A	B	C
1			
2		InputElementA	
3			
4	InputElementA_ValueA1	7	2
5	InputElementA_ValueA2		
6			
7		InputElementB	
8			
9	InputElementB_ValueB1	1	
10	InputElementB_ValueB2	2	
11	InputElementB_ValueB3	2	
12	InputElementB_ValueB4	3	
13			
14		Param_Global	
15			
16	Calculated_ValuaA	1	
17	Calculated_ValuaB	1	

	A	B	C
3			
4	Calculation Sheet		
5			
6	Calculated Value A1		17
7	Calculated Value A2		25
8			
9			
10	Calculated Value B1		17
11	Calculated Value B2		8

	A	B	C
1			
2		OutputElementA	
3			
4	OutputElementA_ValueA1	17	
5	OutputElementA_ValueA2	25	
6			
7		OutputElementB	
8			
9	OutputElementB_ValueB1	17	
10	OutputElementB_ValueB2	8	
11	OutputElementB_ValueB3		
12	OutputElementB_ValueB4		

Figure 3 Workbook Input, Calculation, and Output Sheets
(Click the image for a larger view)

As we stated earlier, programming constructs such as looping are not available to Excel Services. The pre-compiler compensates for such limitations by transforming the input XML fields into a format that is accessible without the need for complex programming constructs. For example, XSD element collections can be transformed into dimensions of the named range. Figure 4 depicts an XSD snippet that is part of the input data contract for a calculation engine.

Elements TypeA and TypeB are part of the input to the calculation. Note the custom attributes RangeHeight and RangeWidth that define the dimensions of the named ranges. The pre-compiler uses this information to generate the named range dimensions. The pre-compiler can also dereference index fields into separate columns—where each column represents an index value.

A noteworthy aspect of the pre-compiler is the ability to preserve the existing calculations while regenerating the input and output sheets. As depicted in Figure 5, developing a calculation algorithm is an iterative process. Business analysts and developers work together to define the initial data contract. During the course of development of the workbook, input and output contracts may need to be modified, and the workbook must be regenerated for these changes to be applied. The pre-compiler supports such iterative development by preserving the calculations sheet while regenerating the workbook.

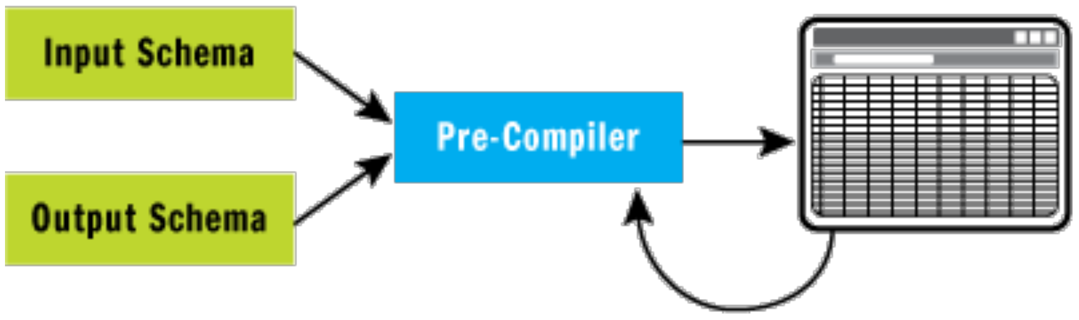


Figure 5 Generating a Workbook Using the Pre-Compiler

The input sheet shown in Figure 3 has the pre-generated named ranges. The calculations sheet has calculations that reference the named ranges defined on the input sheet. The output sheet, in turn, references the calculations from the calculation sheet.

The Excel Web Service Client

The primary role of the Excel Web service client is to invoke calculations inside the workbook using the Excel Services API. In doing so, it interprets the XSD-based input contract, mapping the schema elements into appropriate named ranges. Once the calculation is complete, the client maps the output named ranges back into an XSD-based output contract. Figure 6 depicts the role of the Excel Web service client. A typed DataSet (based on the input schema contract) is passed in as input. Data contained within the DataSet is mapped to input named ranges. Once the calculation is complete, the output named ranges is used to populate the output DataSet. The Excel Web service client is responsible for applying the rules defined for the pre-compilers. It is also possible to inject custom data transformations to alter the aforementioned mapping between XSD and named ranges. Recall our earlier discussion on the need for compensating the lack of programming constructs available to workbook authors. Custom data transformations allow mapping to be altered to make it easy for business analysts to author the calculation logic.

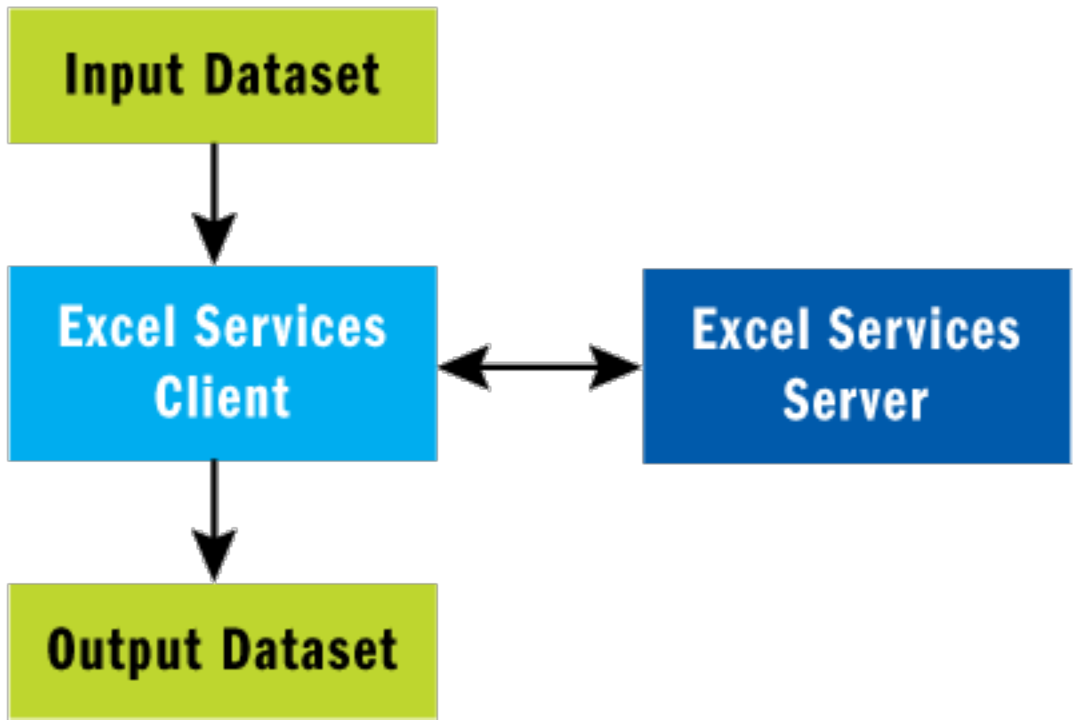


Figure 6 Invoking the Custom Calculation Engine

Digging into the Code

The solution we’ve built is composed of four projects. The PreCompiler project houses all the code to generate a workbook with the required named ranges based on the input and output schema. It takes into account the aforementioned custom attributes such as RangeHeight while generating the named ranges. The PreCompiler project in turn relies on SpreadsheetML (an XML-based dialect used to represent information inside a spreadsheet) for generating the workbook. The SpreadsheetML project contains simple classes that wrap the SpreadsheetML components such as Workbook, Worksheet, and so forth.

The Client project, as the names suggests, is the Excel Web service client code. It sets the values for the input named ranges, forces the workbook to recalculate, and retrieves the values for output named ranges. You will recall that we talked about the need to transform the data to make it easier for business analysts to develop the calculations. In order to allow data transformation to be customized for each calculation engine, we externalized the data transformation logic by defining an IDataTransformer Interface, as shown here:

```
public interface IDataTransformer
```

```
{
    object    getRangeData(string RangeName);
    object[]  getRangeData(string RangeName, int width);
    object[,] getRangeData(string RangeName, int width, int height);

    string getInputSchema();
    string getOutputSchema();

    string getInputSchemaPrefix();
    string getOutputSchemaPrefix();
}
```

The name of an assembly that contains an implementation of the IDataTransformer interface is passed in as input to the client program. The client program in turn calls back the appropriate methods on the class implementing the IDataTransformer interface. In this way, the client program obtains the values that are used to populate the named ranges. The implementation logic inside the IDataTransformer methods is responsible for converting the data residing inside the input DataSet to appropriate named range values. For example, some rows from a DataTable may need to be filtered before populating the appropriate named ranges. Or the rows within the DataTable might need to be sorted before passing them along to the workbook. All such data transformation needs can be met using the IDataTransformer interface.

One other class that is important to discuss here is the ExcelServiceFacade. This class hides the Excel Service API details from the caller. The other important function of this class is to combine individual SetRange calls into one aggregated SetRange call. This is crucial for reducing the network latency as each invocation of SetRange causes a round-trip to the server. By exposing a local SetRange call that is ultimately converted into one aggregated SetRange call, ExcelServiceFacade can dramatically improve the response times. Figure 7 depicts the relevant ExcelServiceFacade code. An internal buffer is maintained by the ExcelServiceFacade class that is appended each time a "local" SetRange call is invoked. Once all the input named ranges have been populated, the internal buffer is sent up to the server as part of a single call. A similar mechanism is used when retrieving the output named values after the calculation is complete. Rather than retrieve the values from the output named ranges individually, we simply retrieve all the values on the output sheet in one fell swoop.

Performance and Scalability

We found the actual execution of the calculations within the workbook to be very fast. In an unscientific test conducted with a workbook containing a nontrivial set of calculations involving close to one thousand named ranges, the response time was under one second; the majority of that time was spent on round-trips to the Excel server. As the load on the system grows in terms of the complexity of the calculation and the number of concurrent executions, it is possible to scale the solution by leveraging the various topology options offered by Excel Services. Different topology options allow you to select where each of the logical Excel Services layers (presentation, application, and database) are placed. For smaller setups (mainly used for testing purposes), it is possible to deploy all three layers on a single server. For a medium setup, the presentation and application layers can be installed on a single server, and the database layer on a separate server. For large setups, you can install each of the three layers on separate servers. Additionally, you can scale out the presentation layer by adding more servers using a network load balancer. The application layer comprising the Excel Calculation Services can also be scaled out using the load balancing schemes supported by the SSP framework. Figure 8 depicts a large setup where each layer is installed on a separate server. Furthermore, presentation and application layers are scaled out using load-balancing schemes.

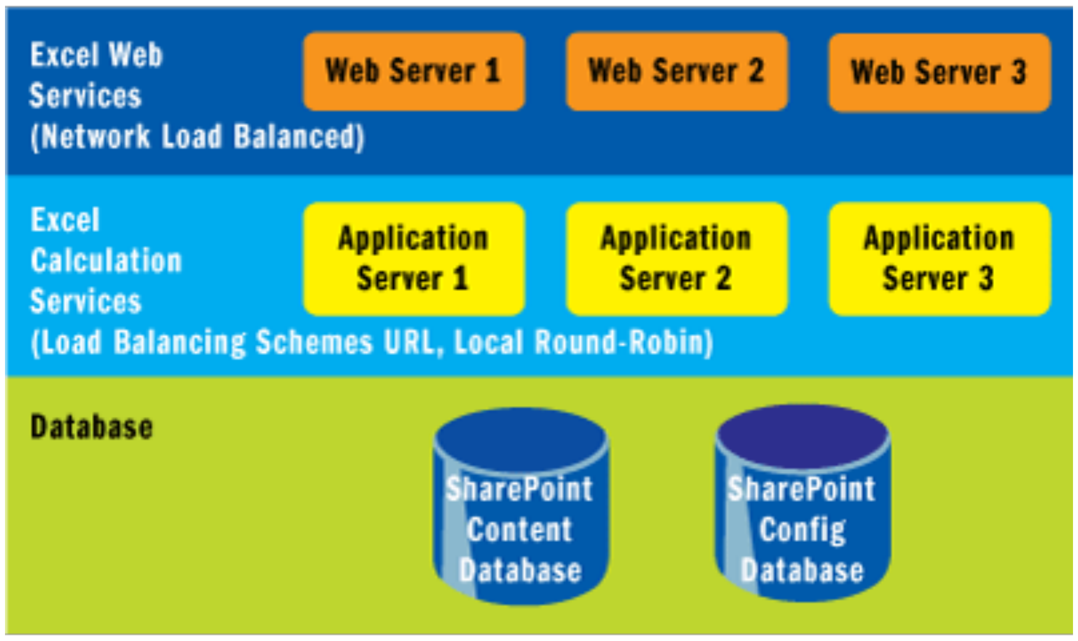

Excel Web Services (Network Load Balanced)
Web Server 1
Web Server 2
Web Server 3
Excel Calculation Services (Load Balancing Schemes URL, Local Round-Robin)
Application Server 1
Application Server 2
Application Server 3
Database
SharePoint Content Database
SharePoint Config Database

Figure 8 Excel Web Services Large Setup

For computationally intensive workloads, it is also possible to combine Excel Services with the Compute Cluster Server to seamlessly distribute work to compute nodes, as shown in Figure 9.

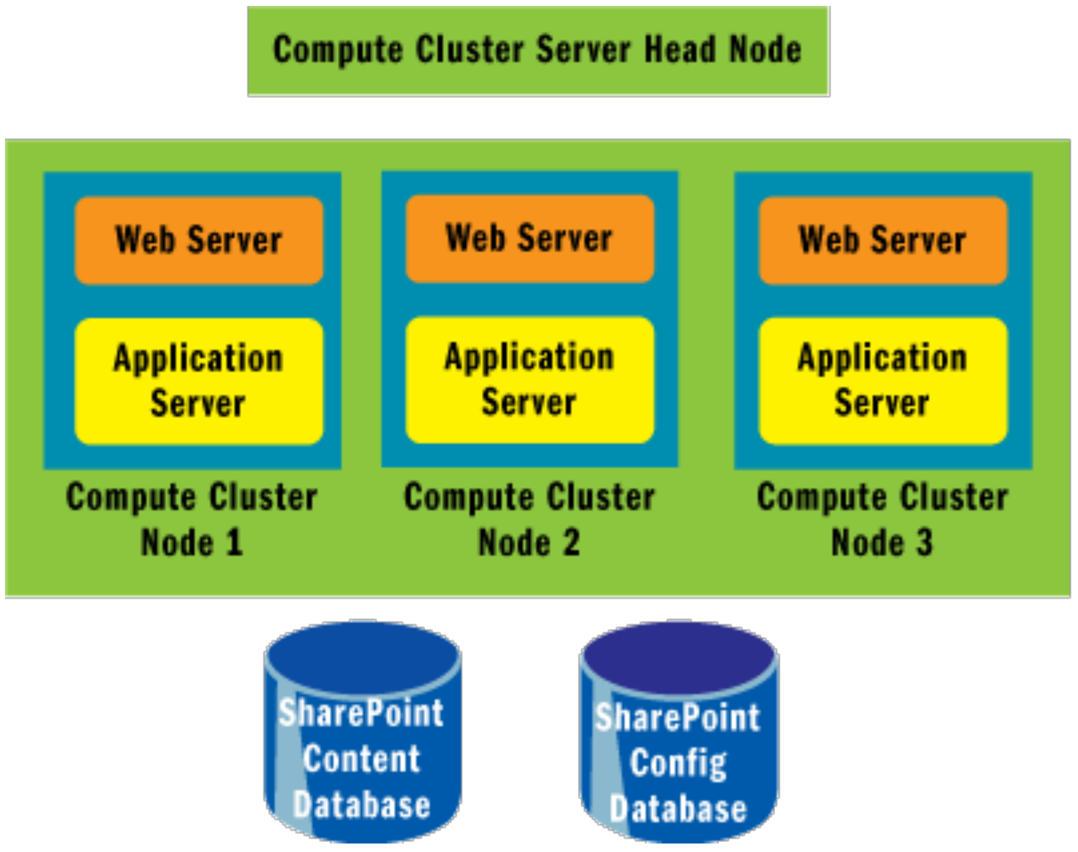


Figure 9 Excel Services High-Performance Setup

As you can see, a custom solution for implementing calculation engine components using Excel Services is a great productivity boon. It allows you to provide your users with anywhere-access to custom workbook functions, eliminates the need for a developer to implement the logic, and lets you scale out your solution as needed. Give it a try. We're sure you'll enjoy the increased flexibility and productivity you'll gain. For more information, see the "Resources" sidebar.

Resources

- Services Technical Overview
- Determine Resource Requirements to Support Excel Services

- Creating XML Mappings in Excel 2003

Vishwas Lele is a CTO at Applied Information Sciences (AIS) in Reston, VA. He assists organizations in envisioning, designing, and implementing enterprise solutions that are based on Microsoft .NET technologies. Vishwas is the Microsoft Regional Director for the Washington, DC, area. He can be reached at vlele@acm.org.

Pyush Kumar is a Lead Systems Architect for Watson Wyatt Worldwide. He's been working most recently on grid computing and large-scale software design for the .NET Framework. You can reach him at pyush.kumar@watsonwyatt.com.
