TOC

About This Guide

This is an archived copy of the v23 documentation provided for Logi Info v23.3 and its service packs.

Notice: Archived Documentation

This documentation is provided as a courtesy reference for a version of our software that is no longer under active development or support. The information contained herein is offered without warranties of any kind, either expressed or implied, including but not limited to warranties of accuracy, completeness, or fitness for a particular purpose.

While this archived material may assist with understanding historical functionality, please be aware that the software described is no longer maintained at this version level and may contain outdated or inaccurate information. Images may not reflect currently supported modules, support sites, or third party products. This software may not be compatible with current versions of previously compatible third party products.

To access and upgrade to current software solutions and receive ongoing support, please contact our customer support team. They can assist you in migrating to the latest appropriate software version that meets your needs. Our support team is available to help ensure a smooth transition to actively maintained alternatives that provide the functionality and reliability you require.

Development Best Practices

This topic enumerates the "best practices" that are taught in our training sessions and relate to development using Logi Studio. The items included here are based on the experiences of hundreds of Logi developers and offer new developers a head start in learning correct and valuable Logi development techniques.

1. KISS: Begin with the end in mind

Keep it simple. Start to design your report definition by visualizing the finished report or web page. If necessary, sketch it out on paper to understand the broad page geography and to identify separate regions.

2. DevNet is your BFF

DevNet is your best source for documentation, reference, and samples.

3. Always turn on Debugger Links

During the development process, turn on and leave on the Debugger Links debugging option.

4. Speeling matters, cAse matters

Pay close attention to spelling accuracy and the correct use of case. Always assume that case is significant, even when dealing with SQL syntax (case-sensitivity may have been enabled for the database server).

5. Think "Waltzing @~"

When working with tokens, think of the famous Australian bush ballad, "Waltzing Matilda", and let it remind you to start every token with an @ sign and, most importantly, to end it with a ~ (tilde).

6. Always edit your definitions in Studio

Studio provides the best environment for development and will catch or prevent many errors.

7. Save early, save often

Save your work in Studio frequently to avoid a loss due to power outages or equipment failure. Back-up your work at the end of a productive day.

8. Always specify the data type

Always set the Data Type attribute value for sorting and group elements and others that use it.

9. Use regular file names

Don't use spaces or special characters in file names.

10. Provide a unique ID for every element

Provide a unique ID for *every* element, even when it's optional, and *don't* use any special characters in the ID. Use a logical and consistent naming scheme for element IDs. For more information, see Element Naming Conventions.

11. For best performance, do it in SQL

If possible, do as much grouping, sorting, and qualifying as possible in SQL (i.e. on the database server), instead of with datalayer child elements.

12. Filters and Conditions always "Filter Through"

In general, data filtering is hierarchical and cumulative: later filtering begins with the results of any earlier filtering. Security Filters are an exception to this rule.

13. Use Divisions Liberally

Use Division elements often to provide structure to your report definition, making it easier to maintain, copy, or rearrange grouped elements. Divisions also make it easy to apply Style, Security, Conditions, and Show Modes to groups of elements. Use the global constant, `rdOutputHtmlDivTagDefault`, to specify the default rendering result when the Output HTML DIV Tag of the Division element is not set explicitly.Division elements also include "HTML Attribute Params", enabling you to apply your

application to work with other frameworks or libraries easier. With the HTML Attribute Params, you have the option to include "style" parameters. For more information, see "Divisions" on page 709.

14. When exporting, always "Go Native"

When building exports into your report definition, always select those elements with "Native" in their names, such as Export Native Excel and Export Native Word.

15. Always specify the Connection ID

Always set the Connection ID attribute value for datalayers and other elements that use it.

16. Quote Text comparisons, not Numeric comparisons

When making comparisons in Condition attributes, in SQL queries, and in script functions, always quote (or double-quote) text, but not numbers. For example, place double-quotes around a request variable and a literal when comparing text: `"@Request.Name~" = "Fairfax"`but not around those comparing numbers: `@Request.InvoicePrice~ < 100`

17. "Column" in an Attribute Name Means No Token Required

If an *attribute name* includes the word "Column", *don't* use an @Data token in the value; just use the column name by itself.

18. Use Shared Elements to Reuse Code, Save Time

Use Shared Elements to optimize the reuse of repetitive code and to provide "single-source" maintenance. Always collect your shared elements into a separate report definition of their own - do not mix them into other report definitions.

19. Give Excel Export a Worksheet Name

When exporting to Excel, always provide a name for the Pattern Block element's Worksheet attribute.

20. Multi-Selects need IN and SingleQuote Token

When working with user inputs that allow multiple selections, use the IN( ) function in your SQL WHERE clause and use the

@SingleQuote token as a prefix to wrap the resulting comma-delimited input values in single quotes in the query. For example,

```
SELECT * FROM MyTable WHERE ColorIN(@SingleQuote.Request.MySelectList~)
```

21. Test Templates with Dummy Data

Always populate Excel Template support files with dummy data to test formatting and formulae.

Improve Application Performance

Want to speed up your reports and analyses? Check out these tips for improving performance:

- Find Out What's Taking So Long
- Filtering and Calculations in SQL Queries
- Use Compare Filters, Not Condition Filters
- Avoid "Every Row" Calculations in Attributes
- Cache Data for Input Select and check box List
- Datalayers: Sequential or Simultaneous?

Find Out What's Taking So Long

In Studio, you can turn on debugging and view the Debugger Trace Report after a report definition runs, see "Debug Reports" on page 56. It provides detailed information about every step in the report execution. Here's an example portion of the Debugger report and how you can use it to see how long processing takes:

Event	Event Detail	Value	Time *
Get DataLayer.SQL	ID = dlOrderSQL		.060
	Compute Data Operation Plan	View Data Operation Plan	.153
	Data Operation Group 1 of 1	View Group Details (1 Items)	.159
	Connection ID	connNW	.160
	Connection Type	SqlServer	.160
	Source	Select * From Orders Inner Join [Order Details] On Orders.OrderID = [Order Details].OrderID Inner Join Employees On Employees.EmployeeID = Orders.EmployeeID	.161
	Process SQL	Open connection	.161
	Process SQL	Send SQL command to data source	.162
	Process SQL	Received response	.182
	Process SQL	Schema built	.185
	Processing SQL	Requesting first row	.206
	Processing SQL	Retrieved first row	.207
	Scripting	5 = 1	.216
	Scripting	6 = 1	.218
	Scripting	4 = 1	.219
	Scripting	3 = 1	.221
	Scripting	9 = 1	.225
	Scripting	1 = 1	.228
	Scripting	8 = 1	.233
	Scripting	2 = 1	.237
	Scripting	7 = 1	.289
	Process SQL	Connection closed	1.657
	Process SQL	Database server took 0.905 seconds to process the query	1.657
	Processed Row Count	345	1.658
	Processed Data	View File Stream Data (10,312,018 bytes)	1.658
	Finalize DataLayers		1.672
	Data Engine	Data processing completed.	1.672

In the right-most column, you can see the execution times, with an embedded chart, for each step. It took until 1.672 seconds into report execution to retrieve that data and have it ready for analysis, based on a SQL query without a WHERE clause. The report instead uses a **Condition Filter** element to filter the datalayer after the query returns the data.

Event	Event Detail	Value	Time *
Get DataLayer.SQL	ID = dlOrderSQL		.054
	Connection ID	connNW	.055
	Connection Type	SqlServer	.055
	Source	Select * From Orders Inner Join [Order Details] On Orders.OrderID = [Order Details].OrderID Inner Join Employees On Employees.EmployeeID = Orders.EmployeeID WHERE Orders.EmployeeID = 1	.056
	Process SQL	Open connection	.057
	Process SQL	Send SQL command to data source	.057
	Process SQL	Received response	.073
	Process SQL	Schema built	.076
	Processing SQL	Requesting first row	.077
	Processing SQL	Retrieved first row	.078
	Process SQL	Connection closed	.313
	Process SQL	Database server took 0.187 seconds to process the query	.313
	Processed Row Count	345	.314
	Processed Data	View Memory Stream Data (10,312,018 bytes)	.314
	Finalize DataLayers		.324
	Data Engine	Data processing completed.	.325

The example above shows a similar query, however this one includes a WHERE clause and a filter element is *not* used. We can see that it only took until 0.325 seconds into report execution, more than *five times* faster than the first example, to have the data ready for analysis. Clearly, this is the faster approach.

You can use the Debugger Trace Page to find out where processing takes the longest and to see how different code changes and element combinations affect performance.

Filtering and Calculations in SQL Queries

As we saw in "Find Out What's Taking So Long" on page 31, it took a lot longer to use an element to filter data compared to using a WHERE clause in the SQL query. This is generally true: if the datasource is SQL-based or accepts queries, you should do all the filtering and calculations that you can *in the query*. The database server is going to process them much more quickly than the Logi application on the web server can.

However, many datasources do not accept queries and that's one of the reasons we have data filtering and manipulation elements.

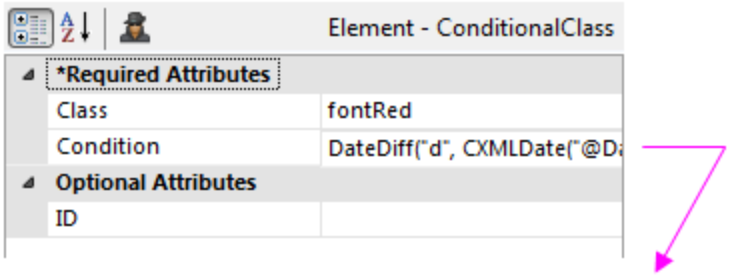
Use Compare Filters, Not Condition Filters

If you have to filter the data after it's been retrieved, use **Compare Filter** elements, *not* **Condition Filter** elements. Compare filters are implemented as a native part of the Logi Engine whereas Condition Filters execute script and that takes a lot longer to run. It's not unusual to see performance improvements of 25-50% when Condition Filter elements are replaced with Compare Filter elements.

Avoid "Every Row" Calculations in Attributes

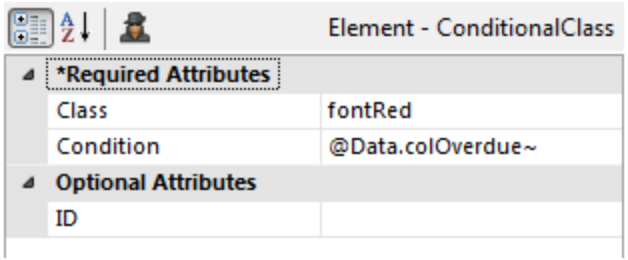
Attributes that can evaluate expressions are very useful. However, for best performance and especially when the evaluation will run for every datalayer row, try doing the evaluation *outside* of the application. For example, the **Condition** attribute expects a *True* or *False* value and you'll get better performance if you can calculate that value outside of the application, for example, inside a SQL query.

Suppose you want to use a time-difference calculation to apply a **Conditional Class** element to data: if a date is 10 days over-due, set the data value color to Red.




```
DateDiff("d", CXMLDate("@Data.ShippedDate~"), CXMLDate("@Data.RequiredDate~")) > 10
```

You could put a complete DateDiff( ) expression into its **Condition** attribute, as shown in the image above. After the data is retrieved, this calculation will run repeatedly, once for every single row in the datalayer.




```
SELECT *,
CASE WHEN DATEDIFF(day, Orders.RequiredDate, Orders.ShippedDate) > 10
THEN 'True'
ELSE 'False'
END AS colOverdue
FROM Orders
```

However, for best performance when using a SQL datasource, do your DateDiff( ) calculation in your SQL query and create a data column containing *True* or *False* based on the results. Then use that data column value in the Condition attribute, as shown above. Once again, this is especially significant when the comparison has to be run for *every row* of data.

Cache Data for Input Select and check box List

When implementing **Input Select List** and **Input check box List** elements that will be used frequently, and populating them with live data (rather than static options), you may want to use **DataLayer.Cached** to retrieve the data. This will limit the number of queries required.

If you must use a long-running SELECT DISTINCT query on a large table to populate the list data, set up a scheduled Process Task to refresh the cached data during off-hours.

Datalayers: Sequential or Simultaneous?

The Logi Server Engine will usually run multiple datalayers in the same definition *simultaneously*, in separate server process threads, to provide the best performance. You can see this happening by examining the Debugger Trace Report: each datalayer running in its own thread will have a "Thread for Datalayer ID" reference.

An exception to this is when datalayers are used under multiple **Local Data** elements. The Local Data elements (and their datalayers) are run *sequentially*, one after the other, in the top-to-bottom order of the elements in the definition. This is done so that the data from one datalayer can be used as criteria in a query in the next datalayer. There is no way to avoid this other than planning your application carefully to minimize the use of multiple Local Data elements, if data retrieval performance becomes an issue.

Hello World! Tutorial

This topic is for new Logi developers. It guides you through the process of building a simple Logi Info application.

This topic contains the following sections:

Requirements

This tutorial assumes that you have:

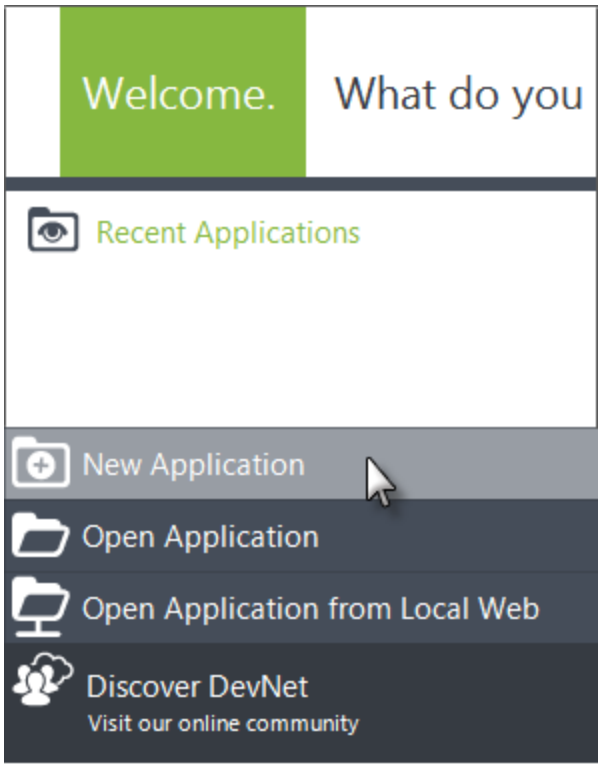
- Microsoft IIS web server and .NET Framework 4.x installed, - OR -
- A supported Java web server and the Oracle JDK or OpenJDK 8, 11, 12, 13, or 14

  Oracle has changed its Java usage policies - see *Java Usage Policy* for important information.
- And an un-expired trial license, or a regular or OEM license file, installed in the *<Logi Info installation folder>*\LogiStudio folder, which defaults to:

  `C:\Program Files\LogiXML IES Dev\LogiStudio`.

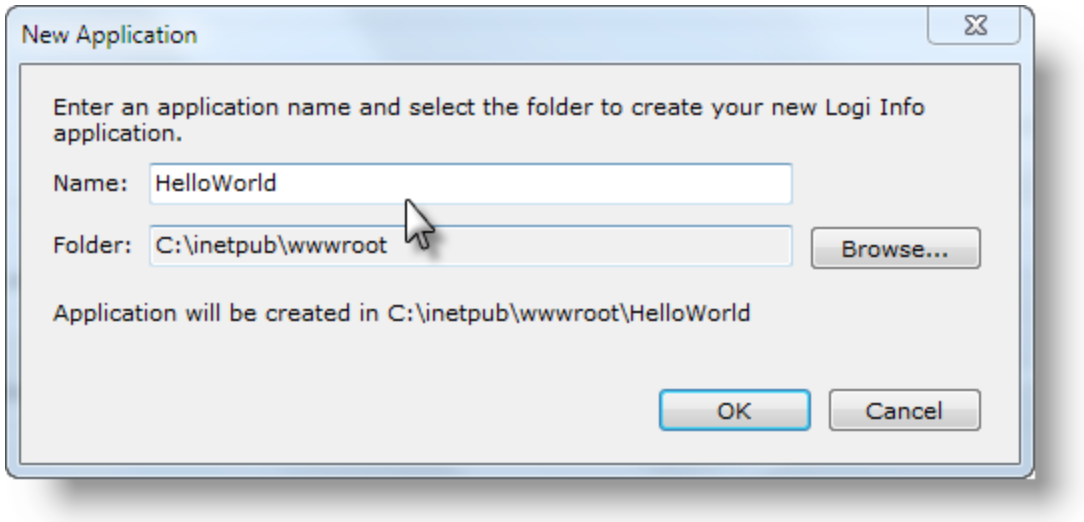
Tutorial

Begin by launching **Logi Studio**, using its desktop icon or Start → All Programs → Logi Info → Studio.



1. In Logi Studio, close the Getting Started dialog box and click **New Application** in the Welcome Panel, as shown above.

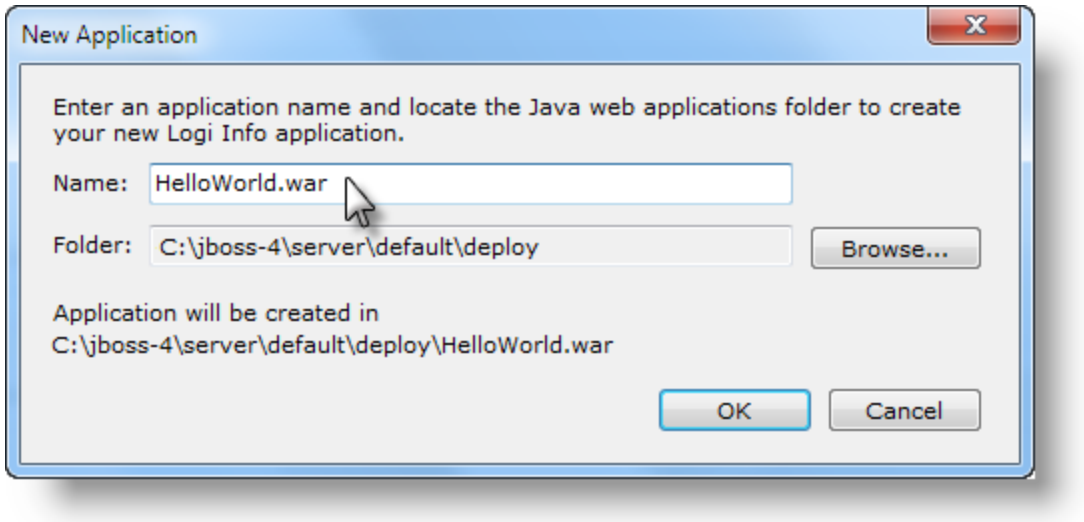
2. Logi Studio is capable of producing both .NET and Java web applications. For this tutorial, we'll select the **.NET** option, as shown above; however, comments are included in the following steps for Java applications.



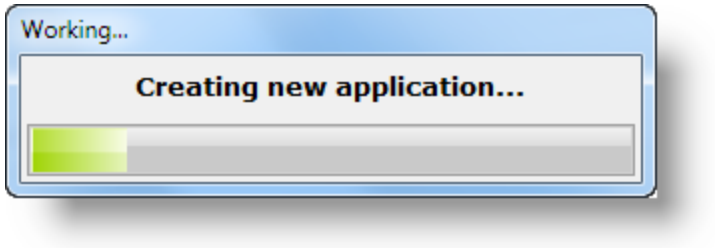
3. Provide an application name: *HelloWorld* - DO NOT insert a space between the two words and be sure to use the exact capitalization shown!

   Choose the location for the new HelloWorld application folder. If you're using the IIS web server, for simplicity, use the default, `C:\Inetpub\wwwroot`, and the wizard will create the new Logi application folder there.

   If you're using a Java-based web server, browse to its standard web application location. For example, for a default installation of Tomcat 8 this would be: `C:\Program Files\Apache Software Foundation\Tomcat 8.0\webapps`.
   Port *8080* will be used by default in the application's URL.

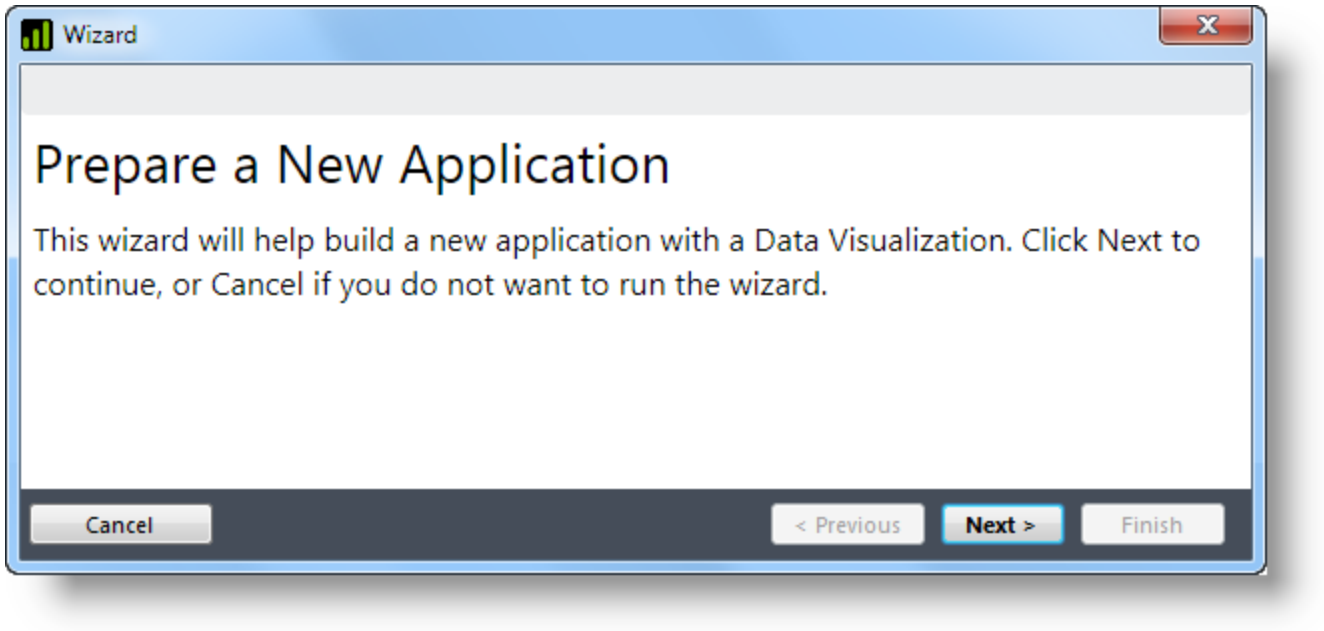


For a JBOSS server, and other Java servers that use auto-deploy, append a ".war" extension to the application folder name, as shown above.

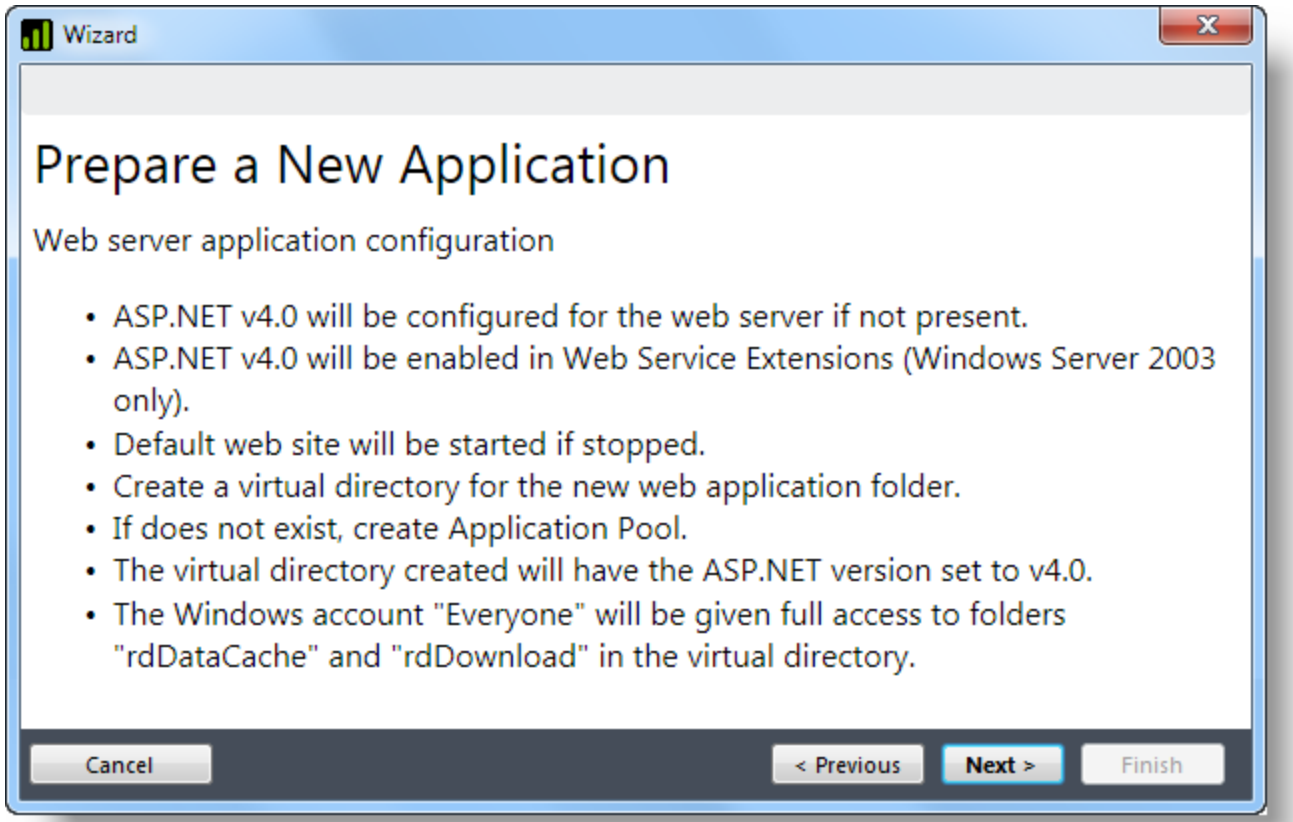


Click **OK** to create the application. The progress indicator, shown above, will appear and required sub-folders and files will be created in your HelloWorld folder. Java web servers may complain as the wizard adds files to the folders; this is normal

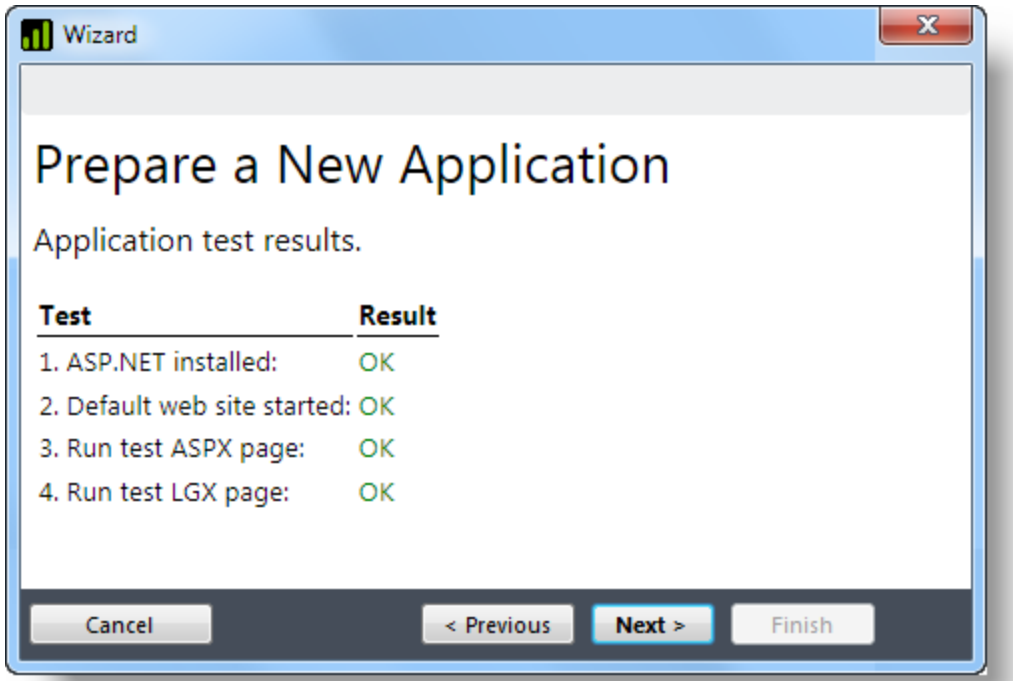
behavior.



4. The **Prepare a New Application Wizard**, shown above, will start. The next step is to configure the basic application settings and register it with your web server. Click **Next**.

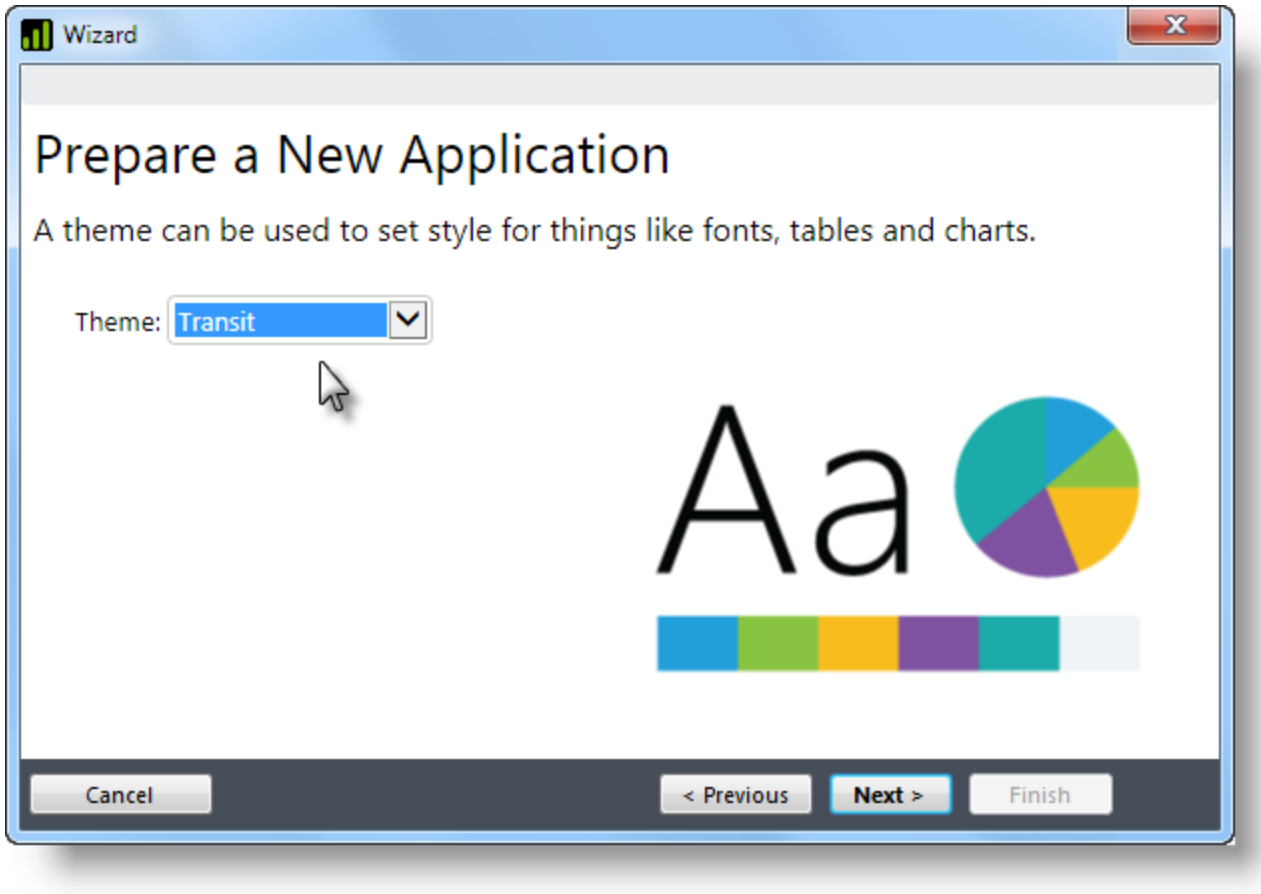


5. The wizard will now run the web server diagnostics and configure your application, described in the dialog box shown above. Click **Next**.

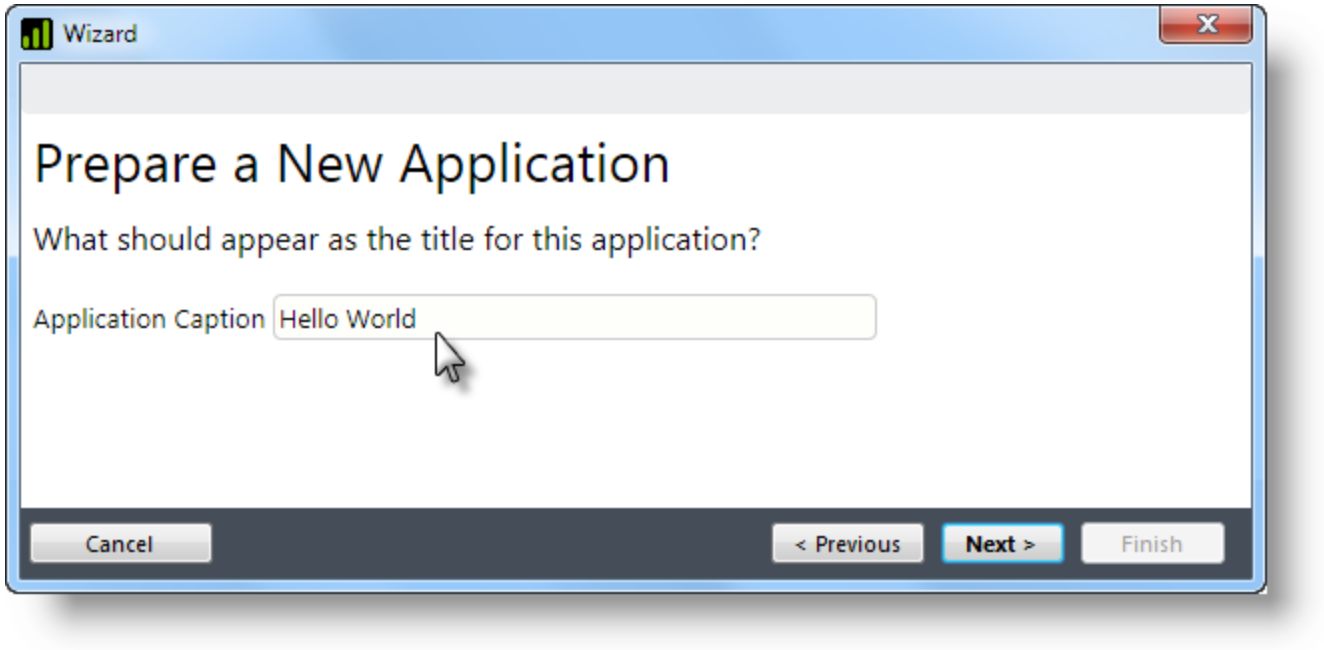


The results of the diagnostic tests will displayed, as shown above. If all test results are "OK", click **Next.**

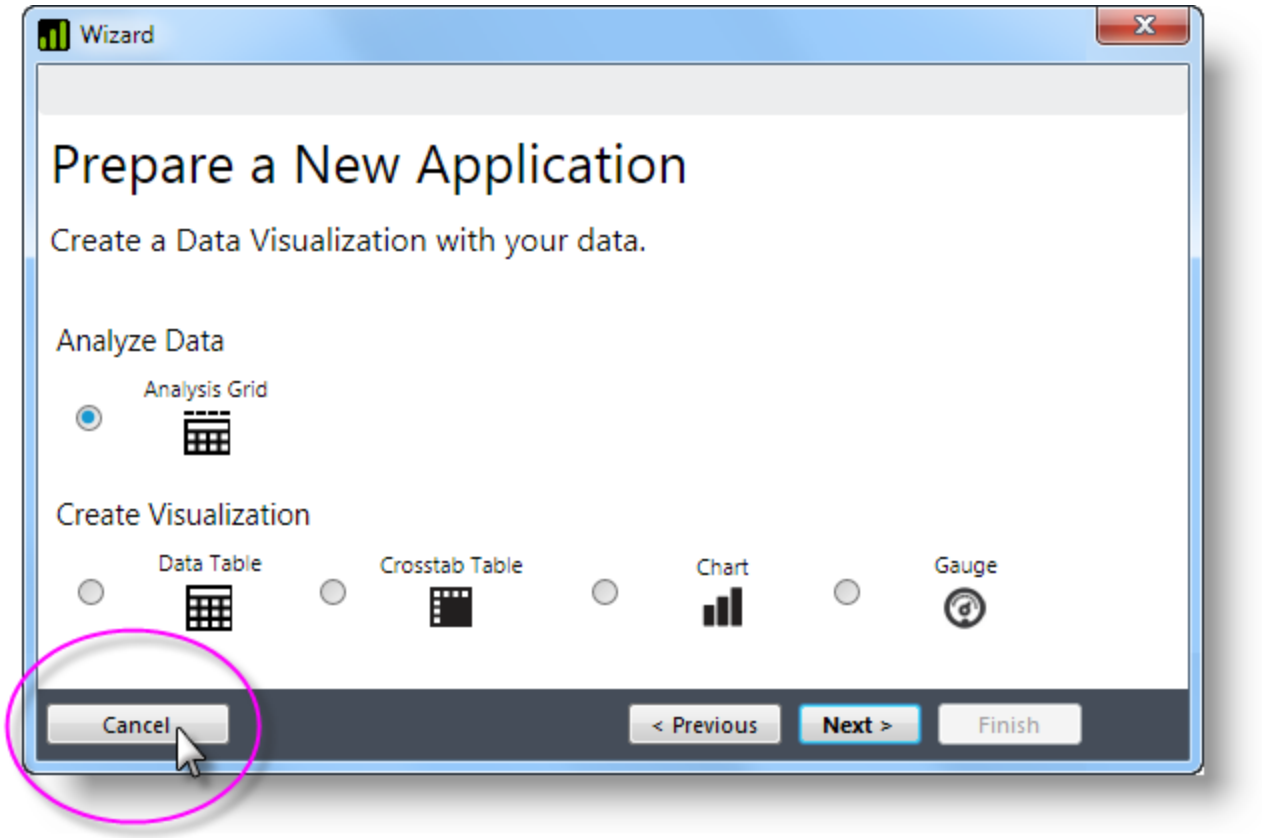
The failure of any test indicates that something is wrong with your web server environment's ability to run the new application. This could be caused by a number of things, such as failing to install the new Logi version as an Administrator, or having incorrect file permissions on the application folder, or failing to have the correct version of .NET installed. Contact Logi Support for assistance.



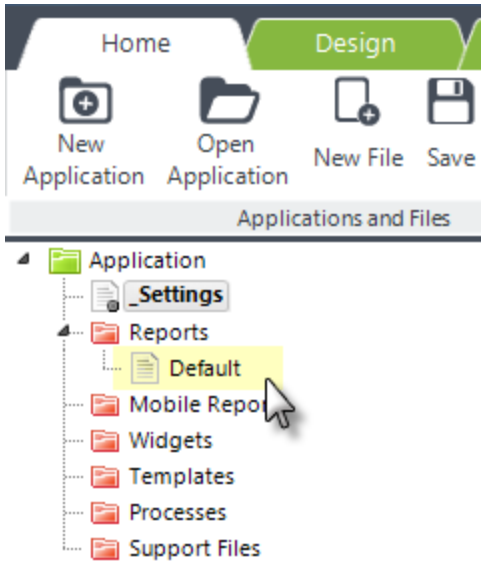
6. Next, you'll be asked to select a **Theme** for your application. Use the default, *Signal,* and click **Next**.



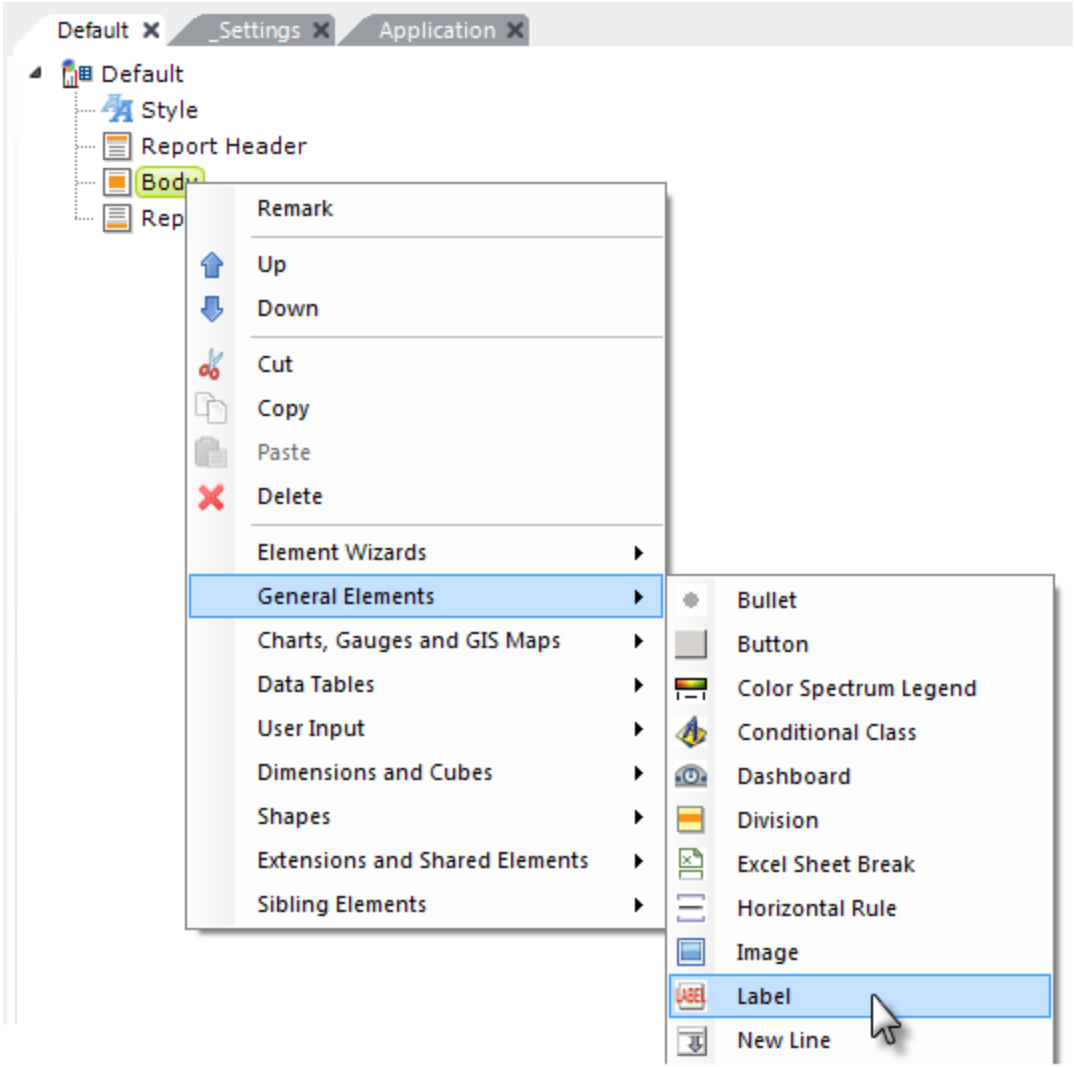
7. Next, enter a caption for the application, and click **Next**.



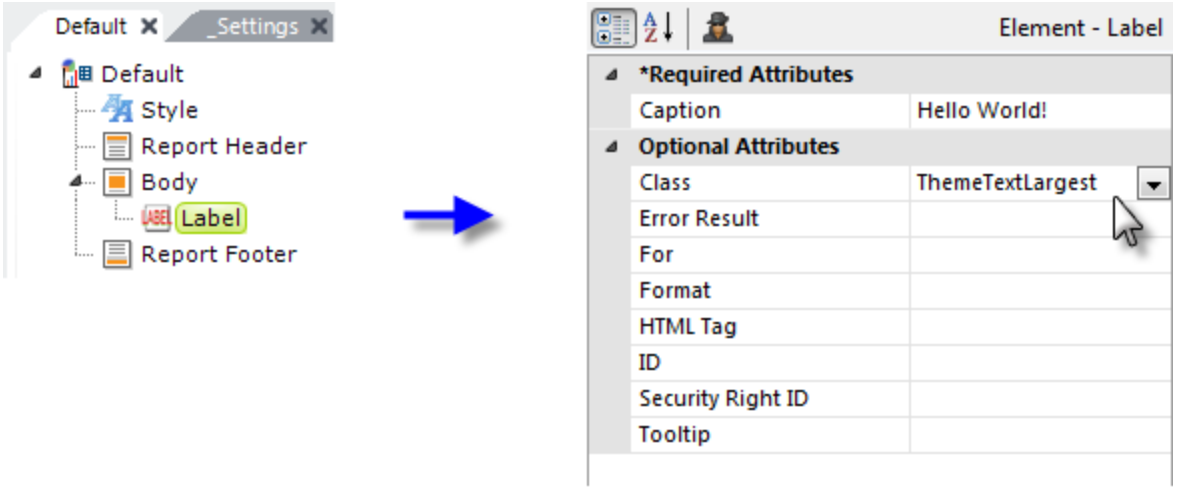
8. When the wizard prompts you to create a Data Visualization, click **Cancel**.



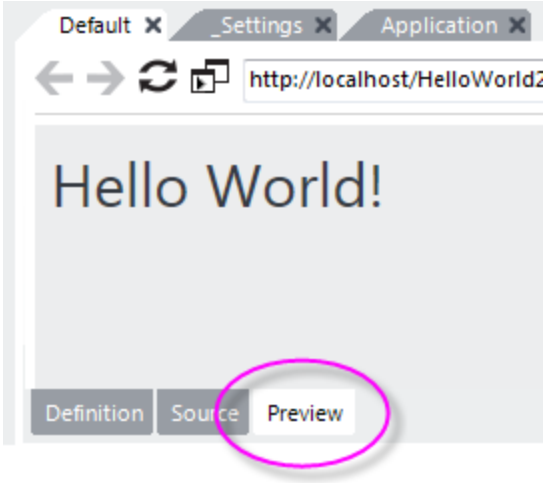
9. In Studio's **Application Panel** (at the upper-left), find and double-click the **Default** report definition. It will open in the center of the screen, in the Workspace Panel.



10. In the Default definition, *right-click* the **Body** element and navigate the pop-up menus as shown above, finally clicking on the Label menu item. A **Label** element will be added to the definition.



11. Click the **Label** element and, in the **Attributes Panel**, enter the text "Hello World!" in the **Caption** attribute, as shown above. Then, in the Class attribute, select *ThemeTextLargest* from its pull-down list of style class options.



12. Finally, click the **Preview** tab at the bottom of the Workspace Panel to save your definition and run the Default report. Your "Hello World" message should be displayed, as shown above.

Congratulations on completing your first Logi report!

Possible Errors

Did you receive an error when you tried to preview the report? Here's how to proceed: *Error: Please specify a URL in "_Settings/Path.AppPath"*
It appears the wizard failed to fill-in the proper value for the Application Path. Open the _Settings definition, and select the **Path** element. Fill-in the value for the **Application Path** attribute with:

```
http://localhost/HelloWorld(.NET)
http://localhost:8080/HelloWorld(Java)
```

Save the definition and try previewing again.

Debug Reports

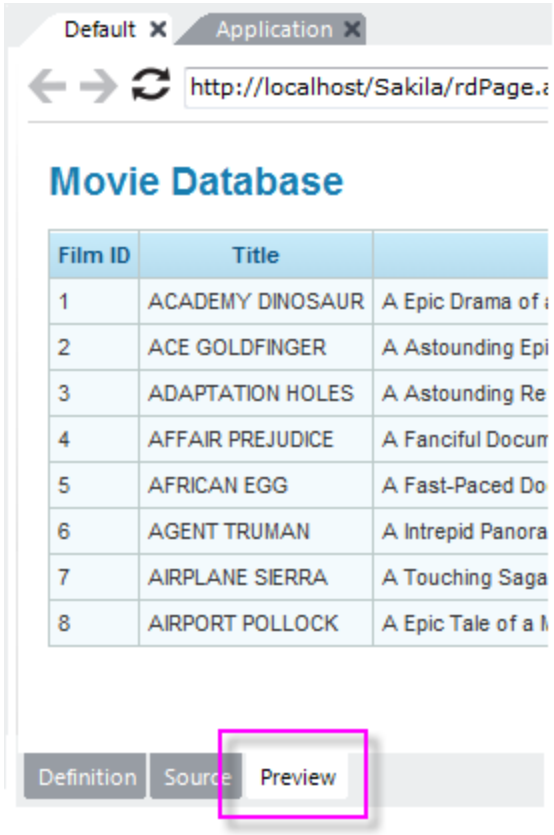
This topic introduces developers to debugging techniques for use with Logi Studio.

The following topics discuss the tools and techniques available in Studio to help you understand what's happening within your application, especially if things don't work as expected:

- Previewing and Browsing
- Using Comments, Remarks, and Application Notes
- Using the Test Parameters Panel
- Using the Debugging Features
- Using the Element Seeker
- Logging Errors
- Debugging Discovery v3.0

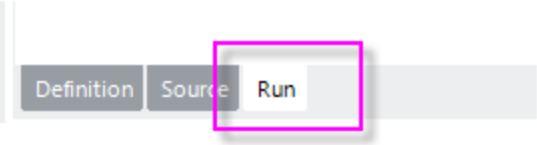
Previewing and Browsing

A Logi application may have one or more report definition files and, while actually browsing a report shows you the big picture, doing so can be cumbersome if you have multiple definitions. Studio's **Preview** tab provides the developer with a fast way to view the report definition currently being edited in the Workspace Panel, without changing the Default Report attribute in the _Settings definition.



To preview a **Report** definition:

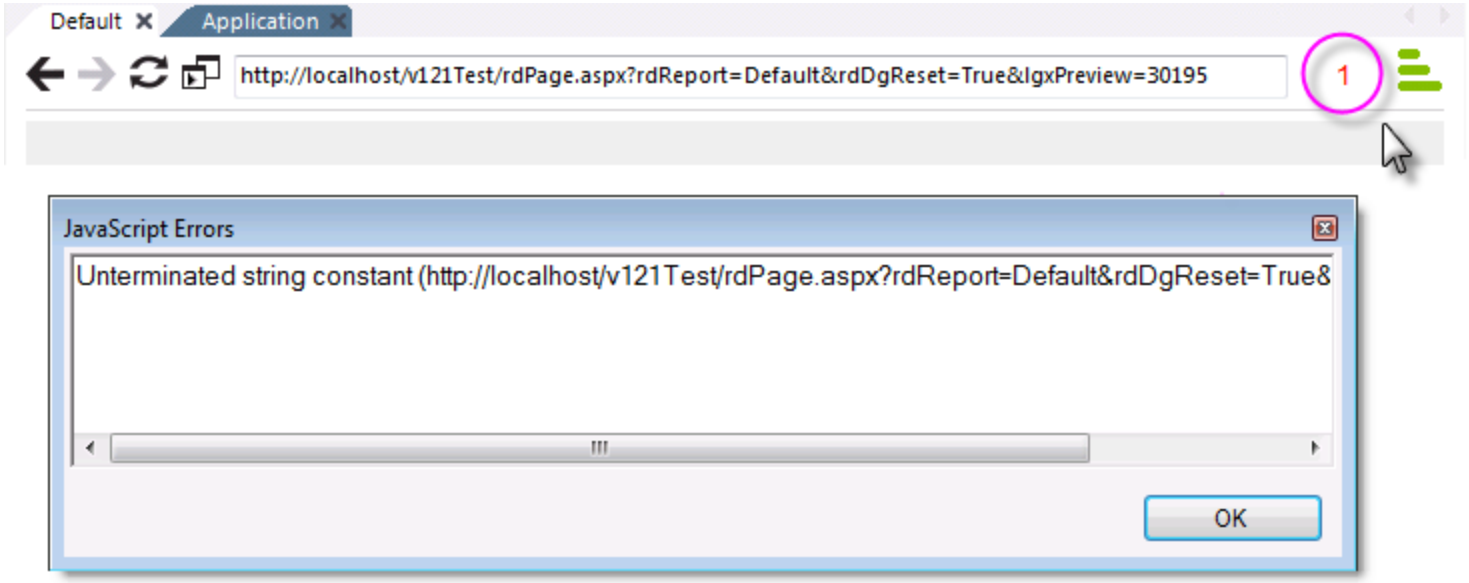
1. In the Application Panel, select and open the report definition to preview.
2. Click the **Preview**button at the bottom of the Workspace Panel, as shown above.
3. Browse and interact with the report definition using the toolbar at the top of the Preview panel.



Logi Info developers can also test their **Process** and **Data** definitions in a similar way:

1. In the Application Panel, select and open the process definition to test.
2. In the Test Parameters Panel, set values for any parameters the process task requires.
3. At the bottom of the Workspace Panel, select the individual process task to test in the task list.
4. Click the **Run** button at the bottom of the Workspace Panel, as shown above.

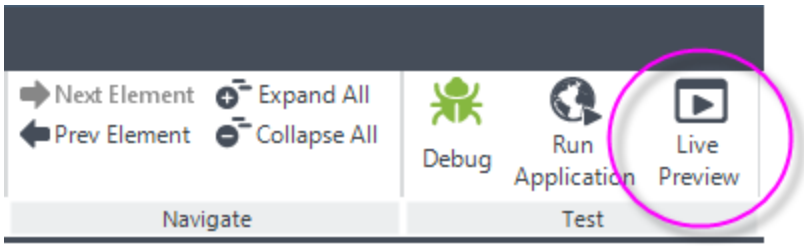
When using Preview, any JavaScript errors thrown will cause a count of the errors to appear next to the preview URL:



Click the number, as shown above, to display a dialog box containing the JavaScript error message. In earlier versions of Studio, the dialog box will simply be displayed automatically, interrupting execution.

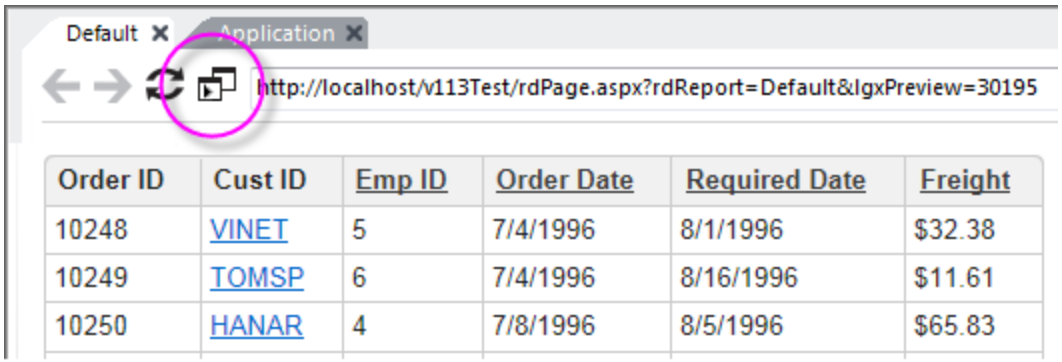
Using Live Preview

The **Live Preview** feature allows you preview your work in a separate, "live" desktop window.



The Main Menu's **Live Preview**item is shown above. The live preview will be displayed in a separate window and any changes made to the definition will immediately be reflected in that window. You can resize and relocate the new preview window as desired; you can even drag it onto a separate display, if you have one.

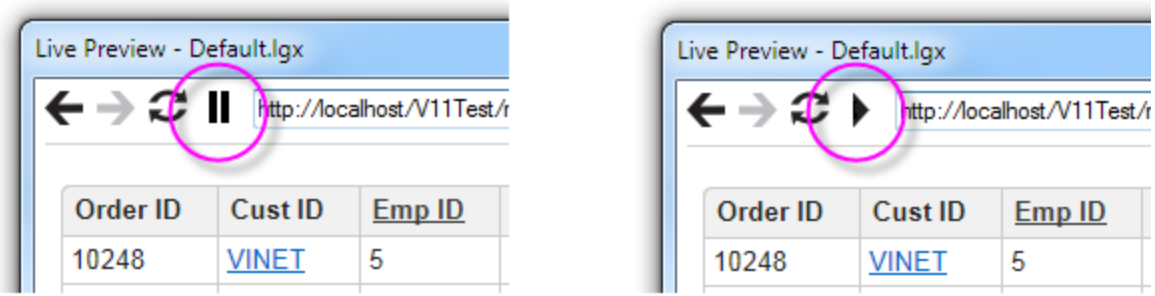
You can also switch to Live Preview from *regular* Preview mode:



To start Live Preview, first preview the report definition, using the Preview tab, and then click the Live Preview icon, circled above.

The preview will be displayed in a separate window, and Studio will switch back to its Definition tab. You can resize and relocate the new preview window as desired; you can even drag it onto a separate display, if you have one.

Any subsequent changes you make to the definition code will immediately be reflected in the Live Preview window.

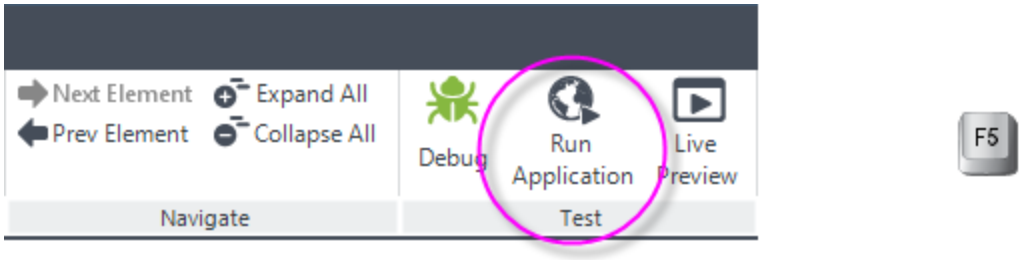


If you don't want definition changes to be updated immediately in the Live Preview window, you can pause and restart them, using the Pause and Play icons in the Live Preview window, circled above.

To exit Live Preview, just click the Live Preview window's Close (X) icon.

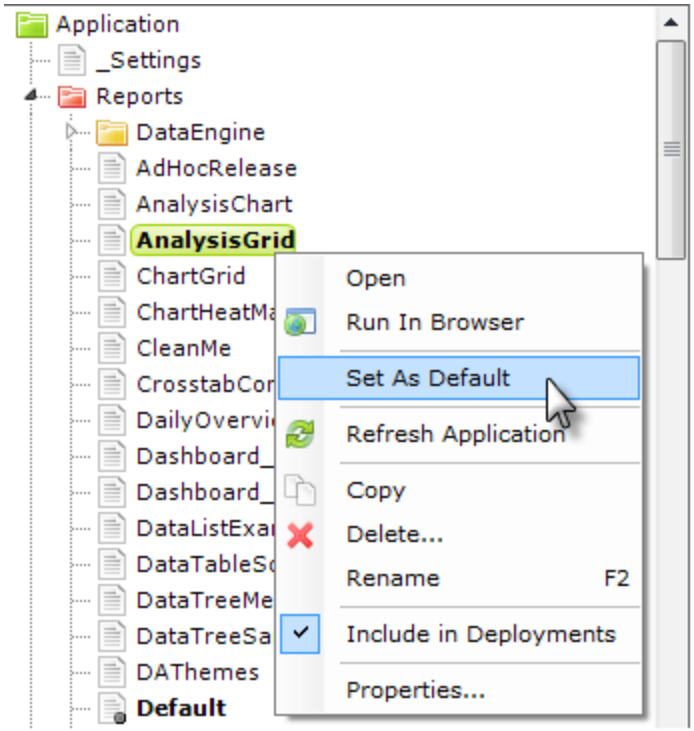
Browsing the Default Report Definition

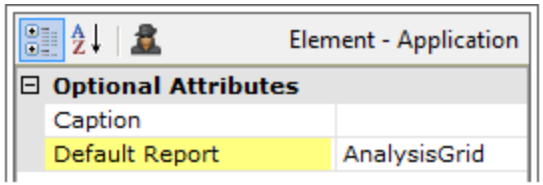
In a Logi application, one of the report definitions is designated (in the _Setting definition) as the "default" report - the page that will be shown if no definition is specified in the URL. If you don't change it, this is usually the **Default** report definition.



You can browse the default report by clicking the **Run Application** menu item, shown above, in Studio's Main Menu, or by pressing the **F5** key. Your computer's default browser (the one associated with .htm/.html file extensions) will be used for this.

You can change the "default" report designation using two methods:





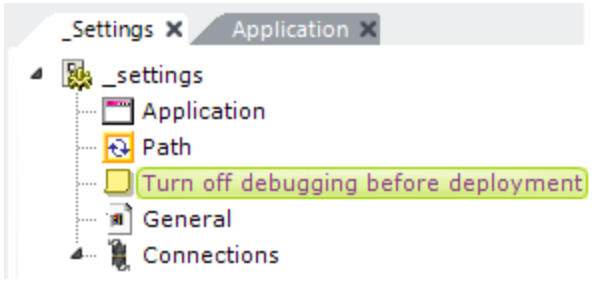
_Settings definition, Application element

As shown above, left, in the Application panel, select the desired report definition, right-click it, and select **Set As Default** from its context menu. This sets the **Application** element's **Default Report**attribute, shown above, right, in the _Settings definition. You can also manually edit this attribute value directly in the _Settings definition.

Using the Run Application menu item will save all open, unsaved definitions before displaying the report.

Using Comments, Remarks, and Application Notes

As an application grows in size and complexity, it's often helpful for developers to leave in-line comments and descriptive notes within the definitions. The **Note** element holds these comments and can be placed anywhere in a report definition; it's ignored when the report is executed.



In the above example, Note elements have been placed in the _Settings definition to provide instructions for other developers. Placing the Notes in different levels of the element tree establishes context for the reader, as shown by the final Note element above. Blue is the default note text color, but you can change this using Studio's Tools → Options menu

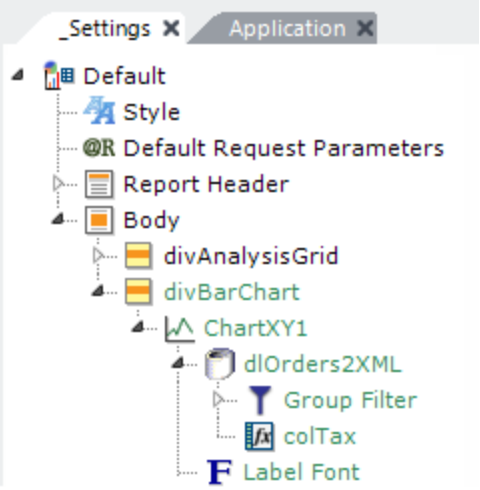
To add a comment:

1. Add the **Note** element to the definition, like any other, wherever it makes sense to you.
2. Select the newly added Note element to define its attributes.
3. Double-click the Note attribute to open the Zoom window, type your comment, and click **OK**.
4. To prevent Notes from appearing in the generated HTML page, remark them (see below).

Remarking Elements

During development, it may be helpful to temporarily "comment out" certain elements. The developer can *remark* a single element or an entire hierarchy of sub-elements in order to temporarily hide them from execution. This can be an important debugging tool as it allows you to selectively prevent execution of parts of the application.

A common troubleshooting technique is to remark a definition repeatedly, successively reducing it down to fewer and fewer basic parts, in order to isolate the source of a difficult-to-find error.



In the example above, the first **Divisions** in the element tree is active. The second Division element has been **remarked** and appears in a green font; it is therefore inactive and will not be rendered in the final HTML output. Green is the default color but you can change this using Studio's Tools→ Options menu.

Remarked elements do not affect the appearance or functionality of the application but remain present in the definition file. At runtime, the Logi Server Engine ignores remarked elements and they are not rendered into the HTML page output.

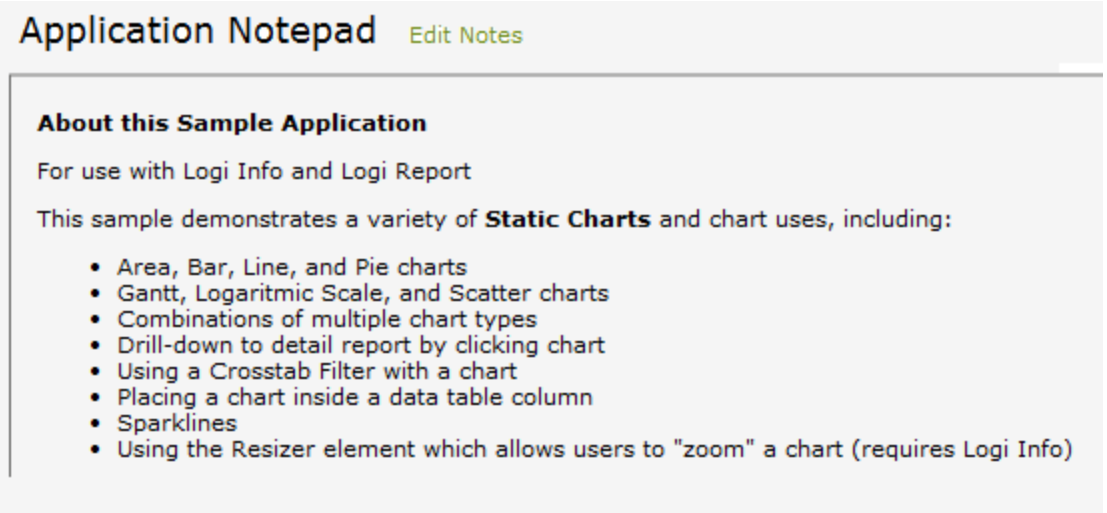
To remark an element:

1. Select the element to remark from the definition file.
2. Right-click it and choose **Remark** from the pop-up menu.

Because elements are structured in a hierarchical tree, if a **parent** element is remarked, all of its **child** elements in the tree are also remarked.

Writing Application Notes

It's often useful to write detailed **notes** about topics that affect the entire application, such as database modifications or session variable names, and the Note element isn't useful for this purpose. Instead there's an **Application Notepad** for each Logi Studio application. To access it, select to the **Application** tab in the Workspace panel.



To write Application Notes:

1. Click **Edit Notes** to begin writing the application notes.
2. Type the notes in the provided textbox.
3. Click **Save Notes** to commit the changes.

The following formatting options are available when writing application notes:

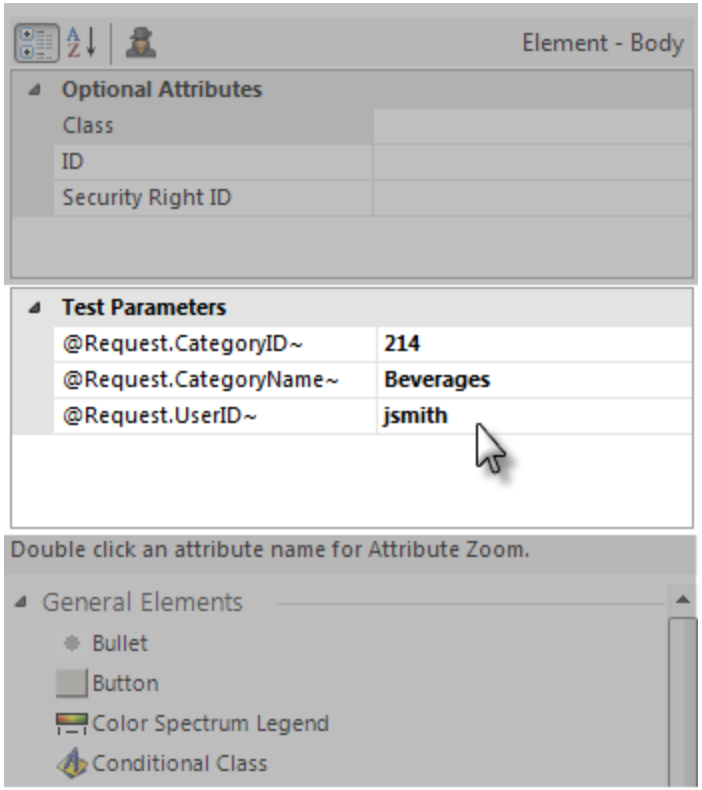
- Bold
- Italic
- Underline
- Bulleted List
- Numbered List

- Decrease Indent
- Increase Indent
- Insert Link

The Application Notes are stored in a separate .xml data file in the _Definitions folder under your application folder.

Using the Test Parameters Panel

One of Studio's least well-known panels is the **Test Parameters** panel. Studio makes this panel visible when the current definition expects to receive and use parameters. It provides a way for developers to supply request variable values for their definitions when previewing them in Studio, and it makes experimenting with different values very easy.

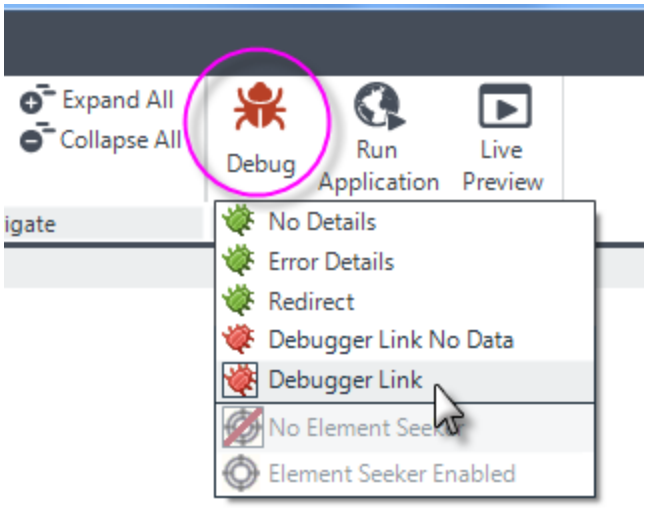


The example above shows this panel in use. Studio will automatically populate the left column with the tokens for any request variables expected by the currently selected definition in the Workspace. You can enter values for those tokens, and they'll be fed to the definition when you Preview it in the Workspace panel, or right-click it in the Application panel and select "Run in Browser".

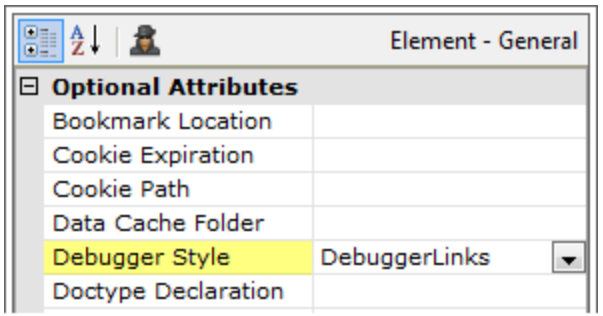
 The token values in Test Parameters *are not* fed to the application when you click the **Run Application** menu item or press the **F5** key.

Using the Debugging Features

Logi Studio provides **debugging** features for an application, but they have to be enabled to work. They are *not* enabled by default.

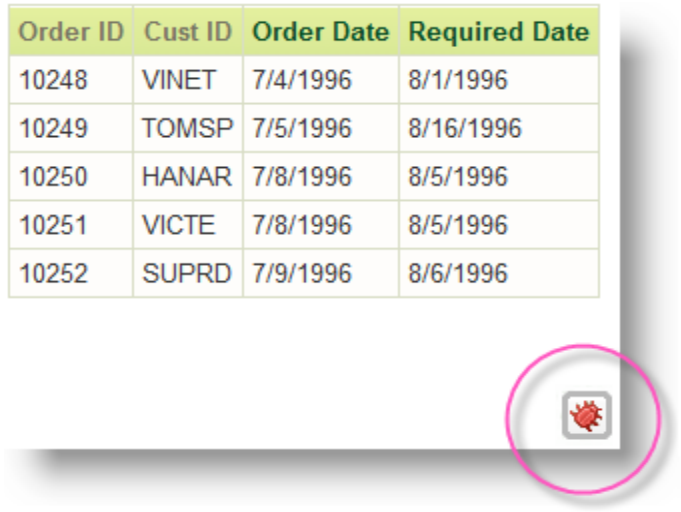


You can turn on debugging for all your report pages by clicking the **Debug** menu item, shown above, in Studio's main menu in the Home tab and selecting *Debugger Link* from the options.



_Settings definition, General element

Selecting a debugger option in the toolbar sets an attribute value in the **General** element, in the _Settings definition, as shown above. This value can also be set manually.



Order ID	Cust ID	Order Date	Required Date
10248	VINET	7/4/1996	8/1/1996
10249	TOMSP	7/5/1996	8/16/1996
10250	HANAR	7/8/1996	8/5/1996
10251	VICTE	7/8/1996	8/5/1996
10252	SUPRD	7/9/1996	8/6/1996

Now, when run, each report page will include a debugger icon that "floats" in the lower right-hand corner of the browser window, as shown above. Click this link to display the **Debugger Trace Report** (discussed below). Any charts on the page will have their own separate debugger links. You can right-click the icon to open the debug page, open it in a new tab (if displayed in a browser), or in a new window, etc.

Each chart will have its own debug icon or link, as well, and a separate debug trace page.

The other debugging modes include:

- *Debugger Link No Data* - Produces the same report as *Debugger Link*, but does not include links to, nor cache, the XML data retrieved. For more discussion of this option, see below.
- *Error Details* - Appends detailed error at the bottom of a page only when an error occurs.
- *No Details* - Prevents any debugger information from appearing when there's an error, so the user cannot see sensitive system security information. Use this option for system deployment.
- *Redirect* - Redirects the browser when an error occurs to a custom error page, at the URL you must specify in the **Redirect Error URL** attribute.

The default value is *No Details*.

For information about logging the debugger output, see "Logging Errors" on page 91.

Using *Debugger Link No Data*

Why is *Debugger Link No Data* useful? When you debug using the standard *Debugger Link* setting, debug data files are generated for *all* of the datalayer elements inside your report. Creating these files requires a significant amount of processing overhead and your reports will run slower. How much slower depends on how much data there is, but expect ~2x slower for each datalayer element.

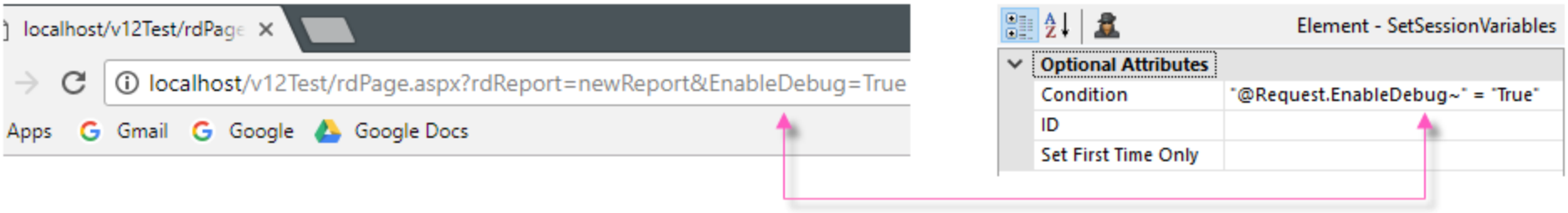
However, if you wish to debug for performance reasons and want to inspect the timings of different events, you can use the *Debugger Link No Data* setting, which creates no debug data files and incurs no processing overhead, to produce a more accurate execution time profile.

Turning on Debugging with Session Variables

You can also turn on debugging for *all* users by setting Session variables. To do this, use the **Set Session Variables** element, and set these two variables:

```
rdDebugSession="True"
rdDebuggerStyle="DebuggerLinks" (can be any of the debugging modes listed earlier)
```

Debugging will be turned on for the duration of a user's session. Because the Set Session Variables element has a **Condition** attribute, you can make debugging dynamic, for example, by adding a parameter and value to the query string used to call a report, then evaluating its @Request token in the Condition attribute:



Enabling Debugging with Security Rights

You may wish to enable debugging just for *specific* users, without enabling it for all users. To do this:

1. Logi Security must be enabled, and users must have security rights assigned to them.
2. In the _Settings definition's **General** element, specify the appropriate security rights as a comma-separated list in the **Debugger Security Right ID** attribute value.
3. Set the General element's **Debugger Style** to *No Details* or leave it blank to ensure debugging is not enabled for all users.
4. Run the application, and pass `rdDebugSession=True` to the page in the query string.

Debugging will remain enabled for users with the specified security rights for the duration of their session.

```
<Label Caption="Enable Debugging" SecurityRightID="Admin">
 <Action Type="Report">
 <Target Type="Report" />
 <LinkParams rdDebugSession="True" />
 </Action>
</Label>
```

The example code shown above will create a link that will be visible only to users with the "Admin" security right ID and, when clicked, will enable debugging for them (assuming Debugger Security Right ID has also been configured with that security right ID).

The Debugger Trace Report

When the Debugger Style has been set to *Debugger Link* or *Debugger Link No Data*, the **Debugger Trace Report** can be accessed in two ways: using the Debug icon or link discussed earlier, or from a link that appears on a runtime error page. The trace report appears either in the Preview panel or in your browser, depending on which method you were using to view your report.

Logi Debugger Trace Report

Event	Event Detail	Value	Time
Debug	Started	2018/09/28 10:27:22.7 AM	.000
Server			.002
	Product Edition	Logi Info Studio and Server	.003
	Server Engine Version	12.6.051-T	.005
	File Version Number	12.6.51.73	.009
	.NET CLR Version	4.0.30319.42000	.011
	Operating System Version	Microsoft Windows NT 10.0.17134.0	.013
	Process Identity	IIS APPPOOL\.NET v4.5	.015
	Process Mode	64-bit	.016
	Machine Name	LGXMCDEV00874	.018
	IP Addresses	::1, 192.168.195.5	.020
Browser	User Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:62.0) Gecko/20100101 Firefox/62.0	.021
Request Tokens			.023
	@Request.rdReport~	Default	.024
	@Request.inpCounty	US	.024
	@Request.dnID~	482231	.025
Response Builder	Start		.028
Requested URL		http://localhost/v12Test/rdPage.aspx?rdReport=Default	.029
Application Security	Enabled	False	.032
Hybrid Data Engine	MemoryStreamLimit	10 MB	.033
Determine Definition			.034
	Requested Definition	Default	.035
Get DefaultRequestParams	Start		.040
	showColumn	False	.043
	inpEmployeeID	2843	.045
	inpCountry	USA	.045

Get DefaultRequestParams	Finish		.046
Get LocalData	Start	localData_Blah	.047
Get DataLayer.SQL	ID = dlEmployeeData		.047
	Connection ID	connNW	.048
	Connection Type	SqlServer	.049
	Source (3)	SELECT EmployeeID FROM Employees	.050
	Process SQL	Open connection	.051
	Process SQL	Send SQL command to data source	.052
	Process SQL	Received response	.053
	Process SQL	Connection closed	.059
	Processed Row Count	9	.059
	Processed Data	View Memory Stream Data (395 bytes)	.060
	Finalize DataLayers		.062
	Data Engine	Data processing completed.	.063
Get LocalData	Finish		.065
Report Show Modes			.066
Apply Theme	Theme Name	Signal	.067
	Applied Modifier	View Modifier Log	.079
	Done	View Modified Definition	.082
Get DataLayer.SQL	ID = dlOrderSQL		.095
	Compute Data Operation Plan	View Data Operation Plan	.096
	Data Operation Group 1 of 1	View Group Details (1 Items)	.097
	Connection ID	connNW	.098
	Connection Type	SqlServer	.099
	Source (4)	Select * FROM Orders Inner Join [Order Details] On Orders.OrderID = [Order Details].OrderID Inner Join Employees On Employees.EmployeeID = Orders.EmployeeID WHERE Orders.EmployeeID = 6	.100

	Process SQL	Open connection	.102
	Process SQL	Send SQL command to data source	.103
	Processing SQL	Requesting first row	.145
	Processing SQL	Retrieved first row	.146
	Scripting	6 = 1	.147
	Process SQL	Connection closed	.153
	Process SQL	Database server took 0.016 seconds to process the query	.154
	Processed Row Count	10	.155
	Processed Data	View Memory Stream Data (300,481 bytes)	.157
	Finalize DataLayers		.160
Apply Theme	Theme Name	Signal	.164
	Applied Modifier	View Modifier Log	.175
	Done	View Modified Definition	.184
Show XSL	XSL Data	View XSL	.195
Load XSL	Success	True	.198
Get XML DataLayers			.199
	Finalize DataLayers		.201
	Data Engine	Data processing completed.	.202
Generated DataLayers	File stream	View Data	.203
	DataLayer Summary XML	View Data	.209
Show ResizableColumns-Modified XSL	XSL Data	View ResizableColumns XSL	.211
HTML Token Replacement	Start	HTML Size: 3,163	.219
	Finish	HTML Size: 3,233	.224
Report	Paging Method	NoPaging	.226
Debug	Completed	2018/09/28 10:27:23.0 AM	.228

* Generating this debugger information increases the overall elapsed time.

Download ZIPed Debugger Trace

As shown above, the **Debugger Trace Report**, includes step-by-step events, values, and time stamps for the report run. Tracing application execution is very useful for troubleshooting, especially for data retrieval issues. Here are some of the items you'll find in the trace report:

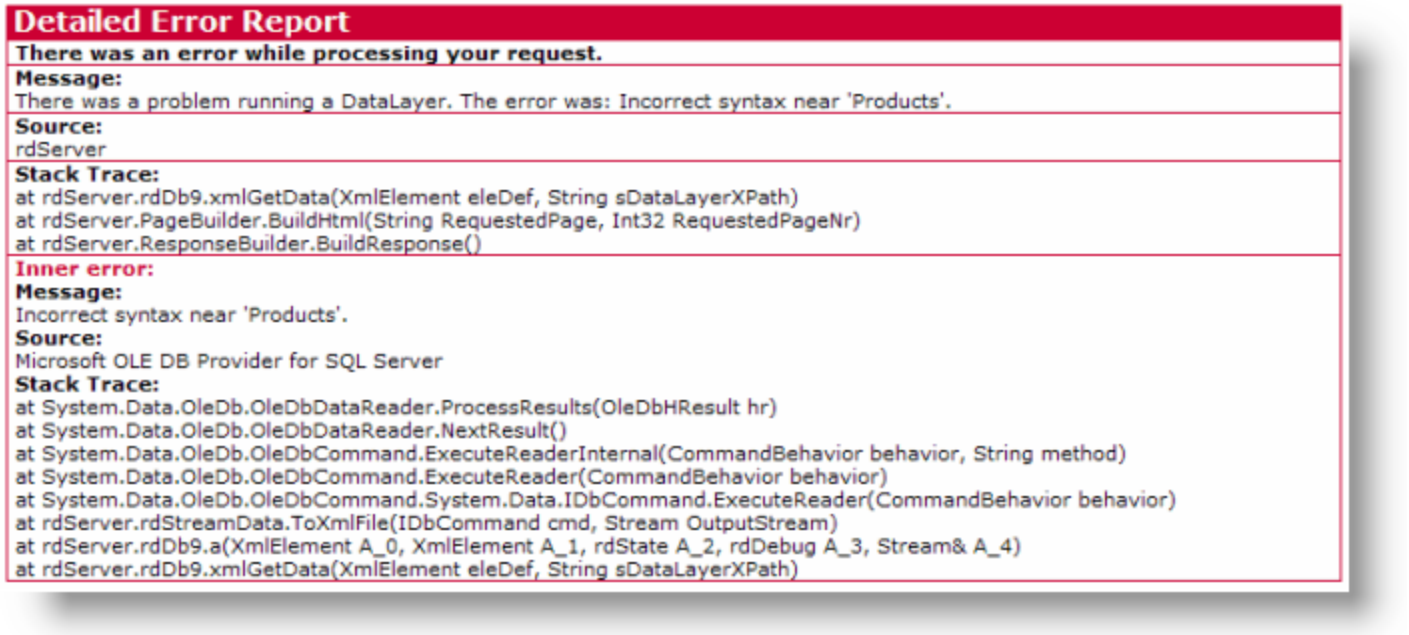
1. Valuable information about Logi Info, the system environment, and the requesting browser.
2. The **Request Tokens** for query string variables being passed to the report, which can be helpful in determining what is being passed from other reports, including from user input.
3. Local Data datalayer activity, such as a SQL query issued to a database, is shown. This is especially useful if a query includes tokens being resolved at runtime. SQL queries shown here can be copied and pasted into Studio's **Query Builder** or other tools in order to run and test them independently. Data retrieval and manipulation is optimized under one or more "data operation plans" to reduce data cache reads and writes. Plan information appears in this section of the debug report.
4. Other datalayer activity, including complex SQL queries with resolved tokens, can be also be viewed.
5. If the Debugging Style selected was *Debugger Link*, then after retrieval and subsequent processing with condition filters, joins, calculated columns, grouping, etc., the data in the datalayer can be viewed, either as a neatly-formatted table or as the stored raw XML.
6. The Time column, with its integrated chart, provides valuable performance information for each step. This allows you to pinpoint which operations are taking the most time.
7. This link at the bottom of the page will package and compress the debug report into a single .zip file, should you need to provide it to Logi Tech Support for diagnosis.

Logi Debugger Trace Report

Event	Event Detail	Value	Time
Debug	Started	2018/09/28 11:39:23.2 AM	.000
Server			.000
	Product Edition	Logi Info Studio and Server	.015
	Server Engine Version	12.6.051-T	.015
	File Version Number	12.6.51.73	.015
	Machine Name	LGXDEV0000	.015
	IP Addresses	192.168.195.5, ::0.0.0.1	.031
Browser	User Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:62.0) Gecko/20100101 Firefox/62.0	.031
Request Tokens			.031
Response Builder	Start		.031
Requested URL		http://localhost:8080/HelloWorld/rdPage.aspx	.031
Java Settings			.047
	Java Version	1.8.0_181	.047
	Java VM name	Java HotSpot(TM) 64-Bit Server VM	.047
	java.awt.headless	true	.047
	java.home	C:\Program Files\Java\jre1.8.0_181	.047
	Free VM Memory	423,888,160	.062
	Total VM Memory	718,798,848	.062
	Max VM Memory	1,193,279,488	.062
	Code Cache Memory	17,887,552	.062
	Metaspace Memory	76,166,648	.062
	Eden Space Memory	72,199,168	.078
	Old Generation Memory	222,711,520	.078
Application Security	Enabled	False	.078
Hybrid Data Engine	MemoryStreamLimit	10 MB	.094
Determine Definition			.094
	Requested Definition	Default	.094
	Read Definition from Cache	False	.125

8. If you're working with a Logi Java application, the Trace Report will also include a special section near the top that presents information about your **Java configuration**.

The debug report also contains more than the information highlighted here, so examine it carefully.



Detailed Error Report

There was an error while processing your request.

Message:
There was a problem running a DataLayer. The error was: Incorrect syntax near 'Products'.

Source:
rdServer

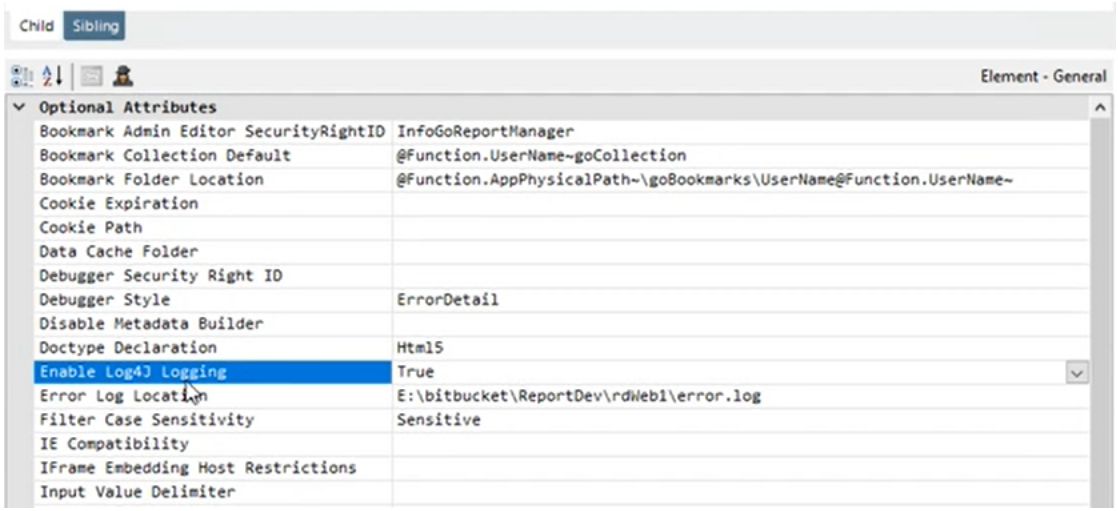
Stack Trace:
at rdServer.rdDb9.xmlGetData(XmlElement eleDef, String sDataLayerXPath)
at rdServer.PageBuilder.BuildHtml(String RequestedPage, Int32 RequestedPageNr)
at rdServer.ResponseBuilder.BuildResponse()

Inner error:
Message:
Incorrect syntax near 'Products'.
Source:
Microsoft OLE DB Provider for SQL Server
Stack Trace:
at System.Data.OleDb.OleDbDataReader.ProcessResults(OleDbHResult hr)
at System.Data.OleDb.OleDbDataReader.NextResult()
at System.Data.OleDb.OleDbCommand.ExecuteReaderInternal(CommandBehavior behavior, String method)
at System.Data.OleDb.OleDbCommand.ExecuteReader(CommandBehavior behavior)
at System.Data.OleDb.OleDbCommand.System.Data.IDbCommand.ExecuteReader(CommandBehavior behavior)
at rdServer.rdStreamData.ToXmlFile(IDbCommand cmd, Stream OutputStream)
at rdServer.rdDb9.a(XmlElement A_0, XmlElement A_1, rdState A_2, rdDebug A_3, Stream& A_4)
at rdServer.rdDb9.xmlGetData(XmlElement eleDef, String sDataLayerXPath)

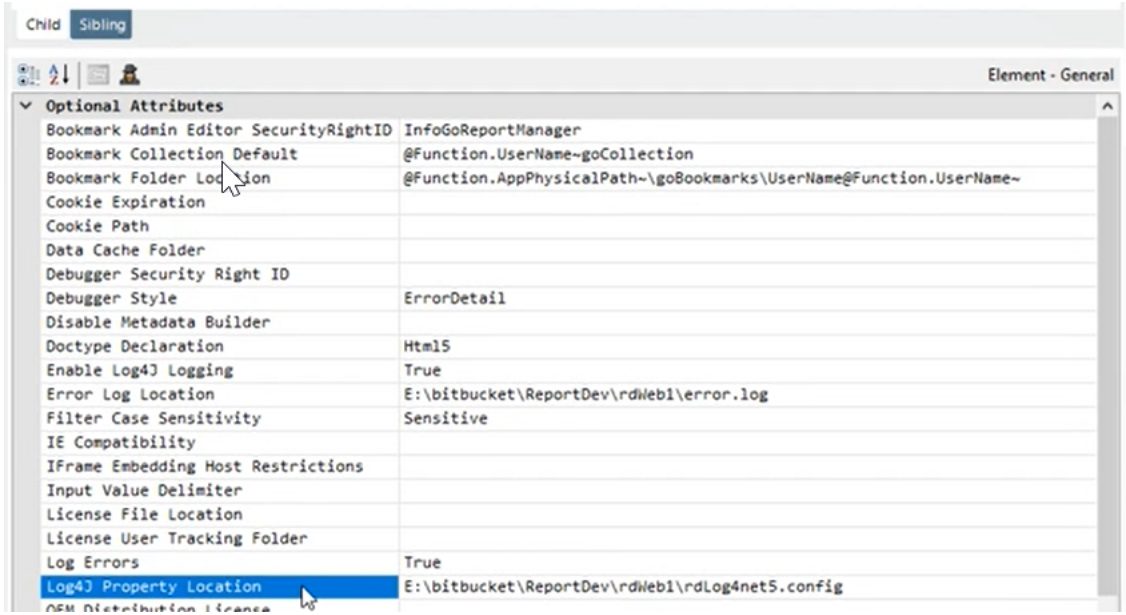
In addition, if an error has occurred, a **Detailed Error Report**, shown above, including the stack trace will also appear, often with a more detailed error message to aid in debugging.

Developers can integrate modern logging framework using Log4Net and Log4J. You must update to Log4j v2.15 to mitigate this 0 day vulnerability. For more information about this vulnerability, refer to this statement.

To pull runtime error and debug information into log files directly, set the attribute "Enable Log4J Logging" to "True":



Then, enter a file location in the "Log4J Property Location" attribute, as shown below:



Below is an example of a Log4J properties file:

```
#app logger option
log4j.rootLogger=INFO, file

# file
log4j.appender.file=org.apache.log4j.DailyRollingFileAppender
#log4j.appender.file.File=c:\\logs\\SSMJava12_6.log
log4j.appender.file.File=${catalina.base}/logs/<APP NAME>.log
log4j.appender.DatePattern='-'yyyy-MM-dd
#log4j.appender.file.MaxFileSize=10MB
#log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n


# Console - not referenced for rootLogger, so logs will not be displayed in console
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
```

Developers can configure the application to log different logging levels through the property file.

Log4J	Logi Info
DEBUG	Debugger No Data
INFO	Event Logging
WARN	Warning

Log4J	Logi Info
ERROR	Error Details
OFF	Disabled Logging

Configuring the Debugger for Java Implementation

For Java implementations there is a separate XML file that needs to be configured in _Settings General:

```
rdDebuggerStyle="NoDetail"
EnableLog4JLogging="True"
Log4jPropertyFile="/usr/share/tomcat/webapps/InfoGo128/rdLog4j2.config.xml"
```

Sample sets configuration status = "debug", which puts superfluous configuration messages in catalina.out with Tomcat; WARN is a better level for implementation.

Please refer to the example below:

```
<?xml version="1.0" encoding="UTF-8"?>
        <Configuration status="WARN">
        <Appenders>
       <File name="RollingFile" fileName="/usr/share/tomcat/webapps/InfoGo128/logs/rdLog4j.log">
        <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m %ex%n</Pattern>
        </PatternLayout>
```

```
</File>
</Appenders>
<Loggers>
<Root level="debug">
<AppenderRef ref="RollingFile"/>
</Root>
</Loggers>
</Configuration>
```

Debugging Scripts

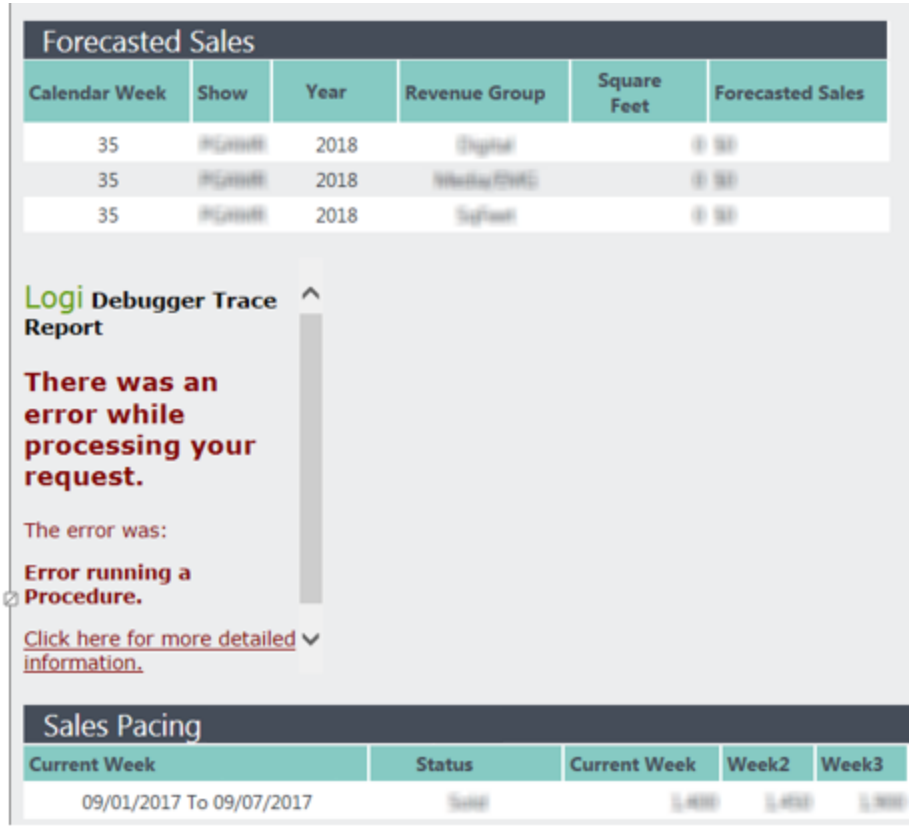
You can also cause debugging to extend to scripts:

Formula Calculations	Start: rdFormula		.035
	Script File	View Script File	.040
	Finish	HTML Size: 985	.047

To enable this, configure the **General** element's **Script Source Debugger Style** attribute. Its *None* value is the default, or you can set it to *OnError*, in which case the View Script File link shown above appears in the Debugger Trace Report when an error occurs. This link displays the script as generated when the report was run. If the script is a simple, single line of script, the actual error may appear here instead of a link. A third value option, *Always*, causes the link to appear in the trace report at all times. *The routine use of "Always" is discouraged as it will incur a significant performance hit.*

Debugging Subreports

If you're working with subreports and a subreport encounters an error, the error message and debug link may appear in a very small iFrame on the parent report page, like this:



It's nearly impossible to view and navigate around the debugger trace report after clicking the link. What to do?

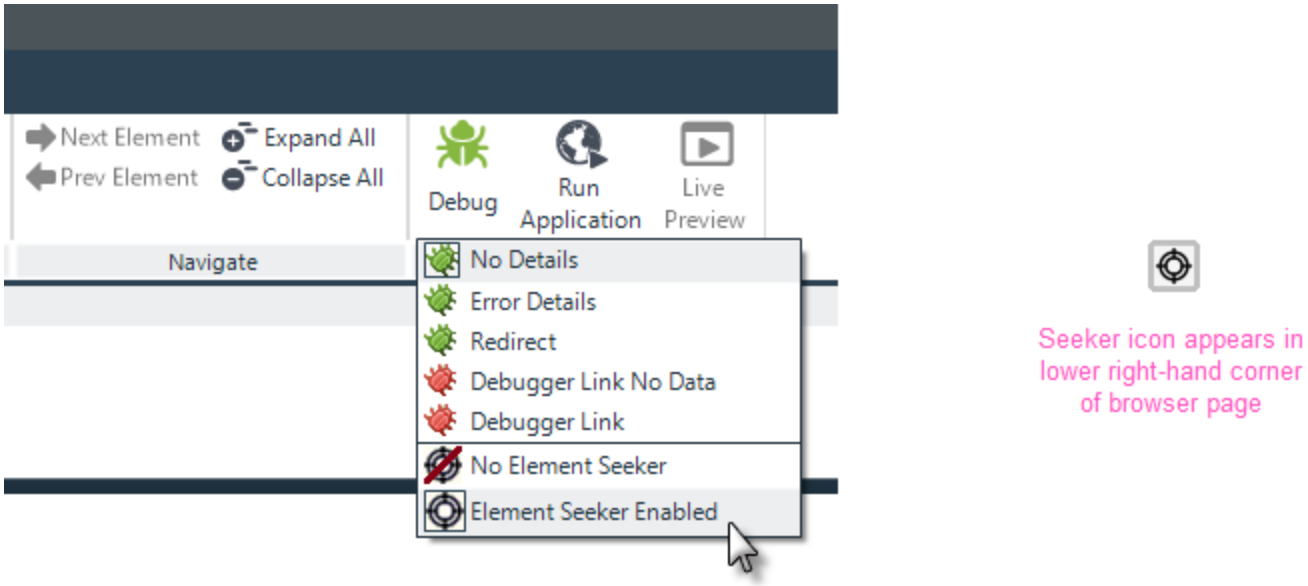
The simplest solution is to temporarily change the SubReport element's SubReport Mode attribute value to *Embedded*. The error message and debug trace page will then expand to the full width of the parent report. Change it back to its original value when you're done debugging.

If, for some reason, you can't change the SubReport Mode, then you need to open the error message or the debug trace page at full-size in a new browser tab or window. Unfortunately, different browsers have different requirements for this, as follows:

- **Firefox**: Right-click the error message and select the "This Frame" and then "Open Frame in a New Tab" or "Open Frame in a New Window" option.
- **Chrome**: Chrome removed its "Open Frame in a New Tab" option some years ago, but you can re-enable it with the free Open Frame extension from the Chrome Store. Once it's installed, right-click the error message and select the "Open frame" and then "Open frame in new tab" or "Open frame in new window" option. If you don't want to install the extension, follow the instructions below for IE11.
- **Microsoft IE11**: Right-click the "Click here for more detailed information" link and select "Inspect element". The Developer Tools panel will open with the error message source code in it. Scroll up in it to the tag that begins with `<body rddbugurl=` and double-click the long value that starts with `rdDownload/` to select it. Right-click the selected value and click "Copy". Manually open a new browser tab and type in the URL for your Logi application, stopping just before `rdPage.aspx`. Paste the copied long value in after the slash and press Enter.
- **Microsoft Edge**: Use the same instructions as above for IE11. When you right-click the "Click here for more..." link, you will see options to open a new tab or window but they won't do anything, so carry on with the "Inspect element" option and other instructions above.

Using the Element Seeker

Logi Studio includes a feature, the **Element Seeker**, that lets you quickly locate an element in a definition file by selecting it in your preview or browser window. To use it, you must be working in Logi Studio v12.2 SP5+ and the Logi Server Engine version of your Logi application also has to be v12.2 SP5+. In addition, if you're using the Internet Explorer browser, you must be using IE 9 or later. Here's how it works:



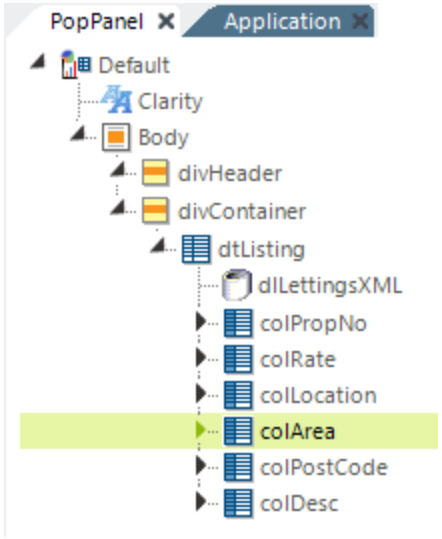
Seeker icon appears in lower right-hand corner of browser page

Enable the Element Seeker in Studio's Debug menu, as shown above. When you preview or browse a report page, you'll see the Seeker icon in the lower right-hand corner.

London Flats to Let

Prop #	Weekly	Location	Area	Post Code	Description
10047	£575	Sinclair Road	Brook Green	W14	A wonderful spacious lower ground two bedroomed flat with large reception good sized rooms and garden, located on the popular Sinclair Road.
10048	£550	Goldhawk Road	Stamford Brook	W6	Situated in a luxury development, this fabulous two bedroomed apartment offers spacious accommodation finished to the highest possible standards with beautiful Amtico wood floor and secure parking.
10049	£525	Blythe Road	Brook Green	W14	Smart two bedroomed flat, benefitting from recent refurbishment and boasting a large reception room, spacious bedrooms and great access to the amenities of High Street Kensington.
10050	£500	Kings Street	Ravenscourt Park	W6	Offering spacious accomodation throughout and benefiting from a delightful south-facing roof terrace, this two bedroomed split-level apartment boasts excellent neutral presentation, private entrance and garage

When you click the Seeker icon, it will turn green and enter selection mode. As you move your mouse cursor around the page, different regions will be framed in green, as shown above. Click a framed region...

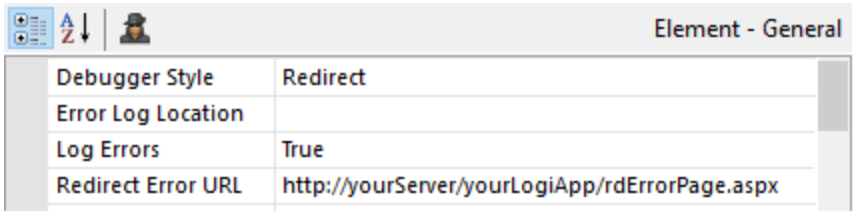


... and Logi Studio will open the correct report definition and select the element responsible for the chosen region, as shown above.

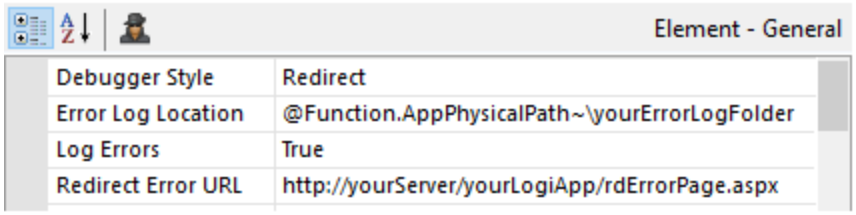
To turn *off* the Seeker's selection mode, you'll need to refresh your preview or browser page.

Logging Errors

If errors are intermittent or difficult to reproduce, then logging the errors may be a good strategy for observing them.



Full debug details written to default folder: rdErrorLog



Full debug details written to custom folder

Error logging with full details is enabled in the application's _Settings definition, as shown above, by setting the **General** element's debugging and logging attributes. Note the forward- and backward-slashes that differentiate a file path and a URL.

If the **Error Log Location** attribute value is left blank, the Debugger Trace Report will be written to the `<application folder->\rdErrorLog`folder, which will be created automatically.

If you only wish to log limited information, you can set the Debugger Style attribute to *ErrorDetail* or *NoDetail.* The log pages will then only include the level of detail provided by the corresponding error messages.

You can also redirect the error logging to a folder of your choice by providing an Error Log Location attribute value. This must be a fully-qualified folder path and you need to manually create the named folder if it does not already exist. Tokens can be used here as part of the path, such as:

```
@Function.AppPhysicalPath~\yourErrorLogFolder
```

Keep in mind that the account, which may be an application pool, used to run web applications requires full File Access rights to the target folder. If the folder does not exist, then the application folder must have these rights so that the target folder, when created, inherits them.

To use a custom location in a web farm environment, specify a network folder and change the application's identity to a network account.

For more information about identifying and debugging common Log4Net problems, see this article.

Debugging Discovery v3.0

The services that support Discovery v3.0, **Logi Application Service** and **Logi Data Service**, have their own logs which can be useful for debugging. Assuming a default installation location, the logs are located in:

```
C:\LogiAnalytics\Discovery\platform\logs
```

There are a variety of log files but focus on the two for the services: `logiApplicationService.yyyymmdd.log`and `logiDataService.log`. The files are created using Log4j and can be easily viewed with Notepad++ or many other free log viewers. You must update to Log4j v2.15 to mitigate this 0 day vulnerability. For more information about this vulnerability, refer to this statement.

Debugging can be turned on and off in this settings file:

```
C:\LogiAnalytics\Discovery\platform\settings\logiApplicationService.json
```

By default, all information, warning, and error messages are logged:

```
      {
      "disableSchemaValidation": false,
      "logiApplicationService": {
      "description": "Platform
 logiApplicationService Configuration",
"logLevel": "debug",
      "corsOriginWhiteList": "*",
      "accessTokenGracePeriod": 120,
      "system": {
```

```
      "pollingIntervalSeconds":
15,
      "pollingTimeoutMinutes":
-1
      },
```

In the file, change the "logLevel" value from "error" to "debug", as shown above, to enable debug logging. You'll need to stop and restart the two services after editing the settings file.

```
      Status 201 [DEBUG] - 2018-01-11T08:07:05-0500 - Response Received
(521bbef0-f6d0-11e7-86ba-97facc95ecaa)
```

An example log debug entry is shown above.

Configure InfoGo for Developers

InfoGo is a complete Logi application distributed with the Self-Service Reporting Module. It provides a complete self-service reporting experience for users.

The following topics discuss how to configure the application after it's been installed:

About the InfoGo Application

Introductory information about the **Self-Service Reporting Module** (SSRM) and directions for installing it can be found in *Install the Self-Service Reporting Module*. If you haven't read it already, we recommend you do so before proceeding.

The SSRM includes a complete Logi application, **InfoGo**. It allows users to create and share visualizations, Dashboards, and reports at runtime, and can be used as is or as a template for your own applications.

- Unlike the Sample Applications on DevNet, InfoGo is distributed complete with Logi Engine files - these include required code and you should *not* downgrade this application to an earlier engine version or it may cease to work.
- InfoGo is an extremely advanced example of Logi Info programming. It uses highly customized code and specialized techniques that may not work if copied into other Logi applications. While you can customize some portions of it, pulling it apart and reusing code in other applications, as you might with a regular Logi Sample Application, may not work well.

Deployment Strategies

You *can* use the InfoGo application right where it was installed, configuring and modifying it right in its installation location. The default installation location is `C:\Program Files\Logi Analytics\InfoGo`. However, this approach requires extreme vigilance when applying Logi upgrades and is therefore dangerous.

 Our Best Practices recommendation is that you *not* run the program from there. Instead, leave those files *untouched* and use them as your "reference model", a pristine set of files that can be copied whenever needed to new applications in other locations

(such as `C:\inetpub\wwwroot`) for customization, registration, and use with your web server. As with any Logi app, you can deploy instances using the Application Deployment tool in Logi Studio.

Java Application

For example, if you want to deploy InfoGo as a Java-based application:

1. Install the SSRM, including the InfoGo application, as usual.
2. Use Logi Studio to create a new Java application.
3. Copy all the folders and their contents from the installed InfoGo application root folder to the new Java application root folder, except `bin` and `rdTemplate`.
4. Copy `InfoGoReportManagementPlugin.jar` from the applications _SupportFile folder to the new Java application's `WEB-INF\lib` folder.
5. Ensure that the copied folders have inherited the file access permissions of the Java application root folder.

Configure the Java application as described in this topic, and make any customizations you desire.

.NET Application

If you want to deploy InfoGo as a .NET-based application:

1. Install the SSRM, including the InfoGo application, as usual.
2. Use Logi Studio to create a new .NET application.

3. Copy all the folders and their contents from the installed InfoGo application root folder to the new .NET application root folder, except `bin` and `rdTemplate`.

4. Give full control to IIS_IUSRS on the .NET application root web folder (and its subfolders).

5. Ensure that the copied folders have inherited the file access permissions of the .NET application root folder.

Configure the .NET application as described in this topic, and make any customizations you desire.

Upgrade Considerations

The 12.5+ versions of InfoGo presents a re-designed side-bar menu and includes more "administrative user" features. These include the new Data Manager tool (an embedded instance of our Web Metadata Builder) which allows designated users to manage connections and metadata at runtime. There are several new standard security Rights, including `InfoGoReportManager` which provides control over a "global main page" and a "global menu". There are also a number of other new security Rights and constants you may need to grant and set.

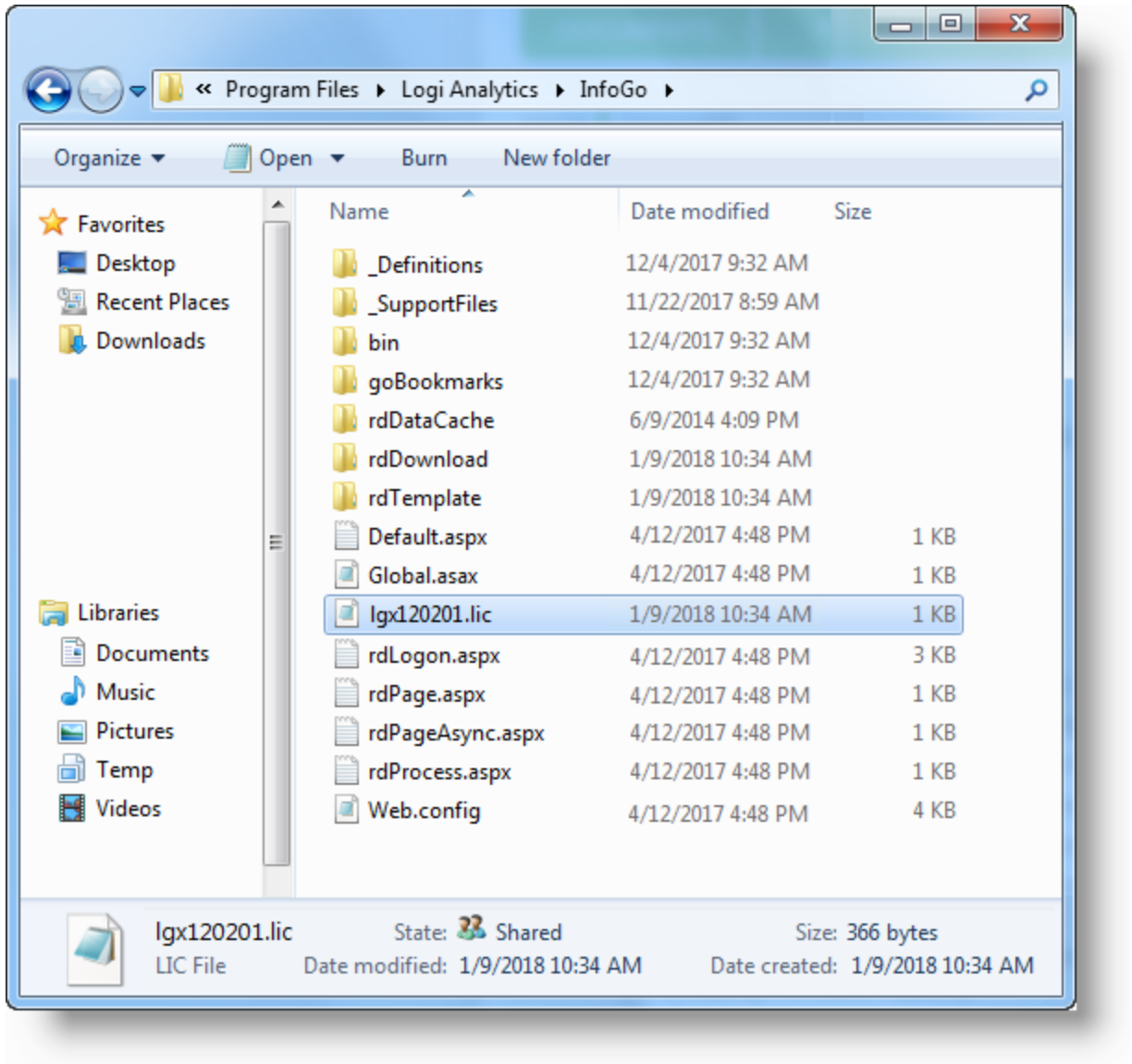
If you're upgrading InfoGo to this Logi Info version, and want to take advantage of in-place editing of Dashboard panel content, you'll need to manually add the `goDashboardPanelEditing` constant. See the "Configuring InfoGo Constants" on page 119 topic for more information.

In addition, the deployment strategy you've chosen has ramifications when you install SSRM updates. When installing an update on top of an existing version in the standard installation location, the installer program will *not* overwrite InfoGo definitions you've modified. So, for example, your customizations won't ever be overwritten. However, if you also have other instances of InfoGo, you'll have to copy updated files to them and then *you* are responsible for deciding what gets overwritten.

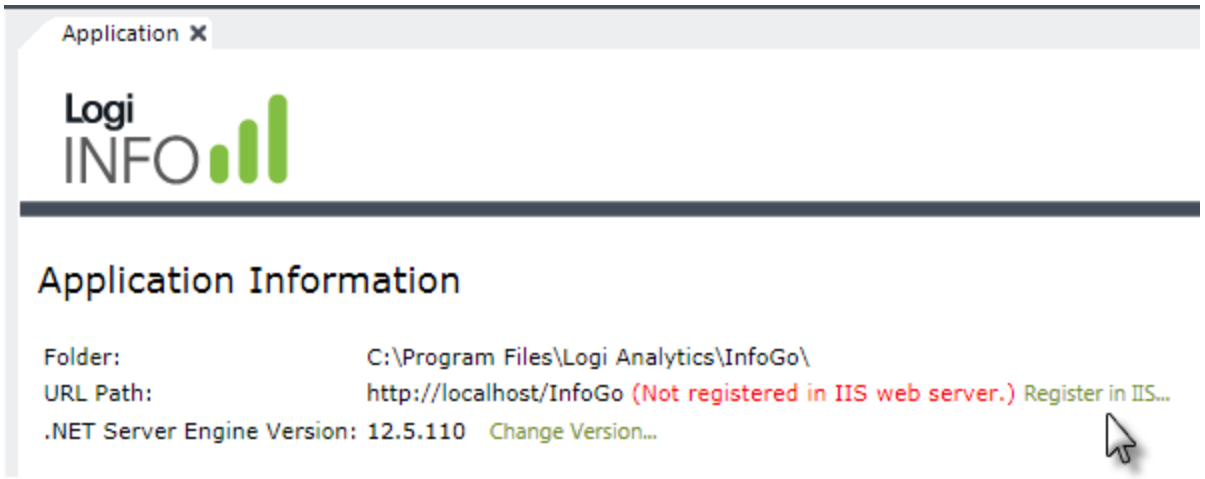
SSRM 12.7 uses a new method of storing gallery items (visualizations) as Bookmarks and an upgrade to this version requires migrating existing items. For more information about using the included migration utility, see *Upgrading the Self-Service Reporting Module*.

Completing the Installation

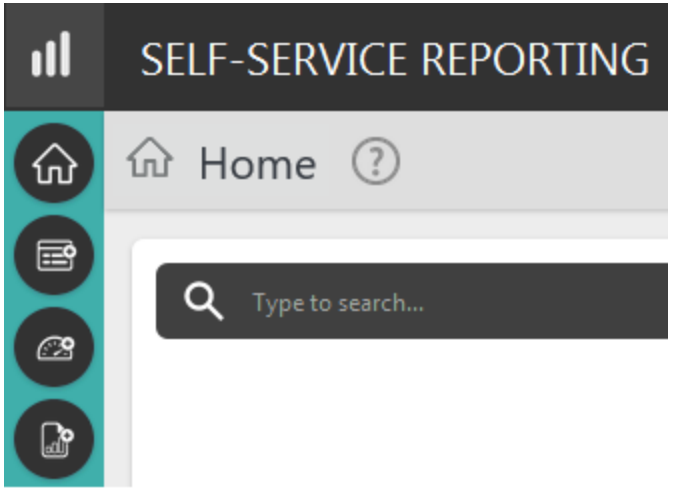
After the SSRM installer has finished, you need to do several things before you can use the InfoGo application.



1. Copy your Logi Info v12 license file, from any existing Logi Info v12 application folder or your Logi Info installation folder (typically `C:\Program Files\LogiXML IES Dev\LogiStudio`) and paste it into the InfoGo application folder, as shown above.



2. Launch Logi Studio and open the InfoGo application by using **Open Application**, then navigating to and selecting the `C:\Program Files\Logi Analytics\InfoGo` folder. Register it with the IIS server, using the link shown above.



3. Now you can test the application installation by previewing its "goHome" report definition in Studio or running the application in your browser. It should look similar to the example above.

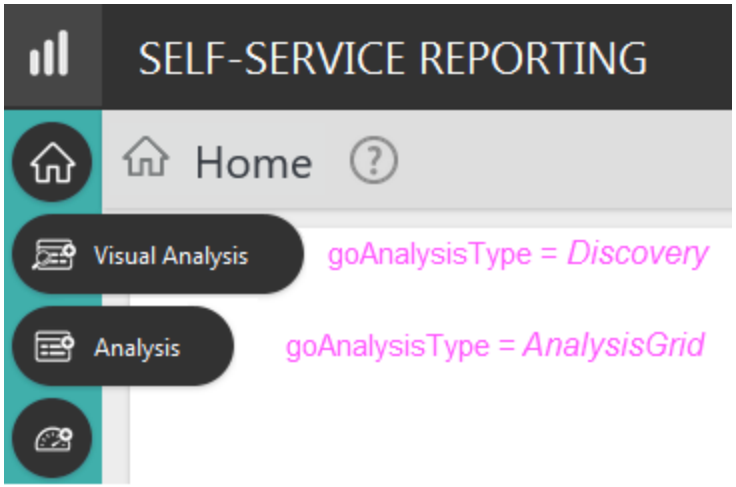
Your installation is complete **but...** you *must* specify a datasource and metadata, as discussed in "Integrating the Discovery Module" on the next page, before it can be used.

Integrating the Discovery Module

The Logi Discovery Module (DM) is a separate, recommended add-on module that enables special elements in Logi Info and provides access to the **Thinkspace**, a highly-interactive analysis tool instead of, or in addition to, the Analysis Grid.

Installation information for the DM is available in *Installing the Discovery Module - Windows* and *Installing the Discovery Module - Linux* and is not repeated here. The remainder of this topic describes the configuration steps needed to enable the DM in InfoGo.

If the DM has been installed, a special definition, goDiscovery, will be available in the InfoGo report definitions.



Depending on how the `goAnalysisType` constant is configured (see below) different analysis tool options will be shown in the menu for use when creating a new analysis. Both options, as shown above, can be displayed if the constant is set to *User-Selectable.*

Connecting to Data

A successful configuration of the Discovery Module requires *two* additional datasource connections, as described in the Connecting to Data section of "Connecting to the Logi Application Service" on page 160. Be sure to add and configure these elements, and to create metadata for the Connection element.

 In the goDiscovery definition, you must set the Thinkspace element's **Logi Application Service ID** attribute to match the ID of the Connection.Logi Application Service element you added in the _Settings definition.

Special DM Constants

In order to enable the DM, add the following constants and values to the _Settings definition:

Constant	Description
goAnalysisType	Specifies the analysis tool options that will appear on the menu. Possible values and effects include: *AnalysisGrid* (or if left blank) - The Analysis Grid will be used, and the goAnalysisGrid definition will be shown when this option is selected. *Discovery* - (Assumes the DM is installed) The Thinkspace will be used, and the DM's goDiscovery definition will be shown when this option is selected. *UserSelectable* - Both options will be displayed, allowing the user to choose their analysis tool.
goMetadataIDsAnalysisGrid	Specifies a comma-separated list of one or more IDs of Metadata elements, defined for con-

Constant	Description
	nections in the _Settings definition, to be used with the Analysis Grid. If left blank, *all* Metadata element IDs from _Settings will be used. This constant is used in the Active Query Builder in the goAnalysisGrid definition.
goMetadataIDsDiscovery	Specifies a comma-separated list of one or more IDs of Metadata elements, defined for connections in the _Settings definition, to be used with the DM. If left blank, *all* Metadata element IDs from _Settings will be used. This constant is used in the Active Query Builder in the goDiscovery definition

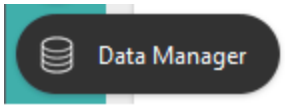
Specifying the Application Metadata

InfoGo takes advantage of two special elements, **Metadata** and **Active Query Builder**, that are installed with it to enable extended functionality in Studio and in the Analysis Grid super-element.

Developers can use the **Web Metadata Builder** to specify the data that will be available for use at runtime by creating a metadata file. Step-by-step instructions for creating or editing a metadata file can be found in *Use the Web Metadata Builder*.

Data connections and metadata can also be managed inside InfoGo at runtime. To enable this feature, the **Security** element's **Metadata Admin Security Right IDs** attribute should have a value of *InfoGoDataManager*.

Then give designated users the `InfoGoDataManager` security Right and they'll see a special option in the side-bar menu:



This option launches an embedded instance of the Web Metadata Builder and allows data connection and metadata creation and management right inside InfoGo, at runtime.

Access to the Web Metadata Builder tool can be disabled for *all* users by setting the **General** element's **Disable Metadata Builder** attribute equal to *True*, in the _Settings definition. To grant more selective access to the tool, use the Security element's Metadata Admin Security Right IDs attribute discussed earlier.

Metadata created externally can also be retrieved from a REST-style web service by specifying the service's URL in the Metadata element's **Metadata Url**attribute. When the MetaData Url is specified, the Web MetaData Builder works in read-only mode, allowing the user to view everything, but editing permissions are withheld.

If you want users to be able to select different metadata files at runtime, be sure to enumerate them as a comma-separated list in the `goMetadataIDsAnalysisGrid` and/or `goMetadataIDsDiscovery` constants. See the constants table on the next page for more information.

If the metadata changes, such as the removal of a column, after Analyses have been created, a warning about the missing column will be displayed, instead of the expected visualization.

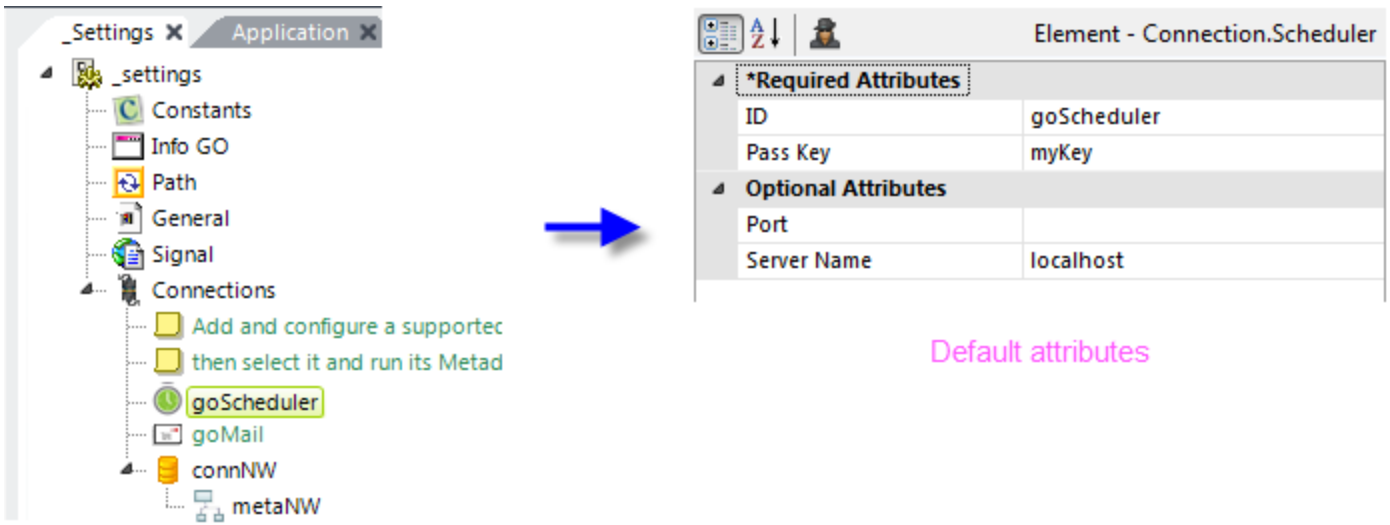
After building your metadata file, return to this topic and continue with "Configuring Scheduler and SMTP Services" on the next page.

Configuring Scheduler and SMTP Services

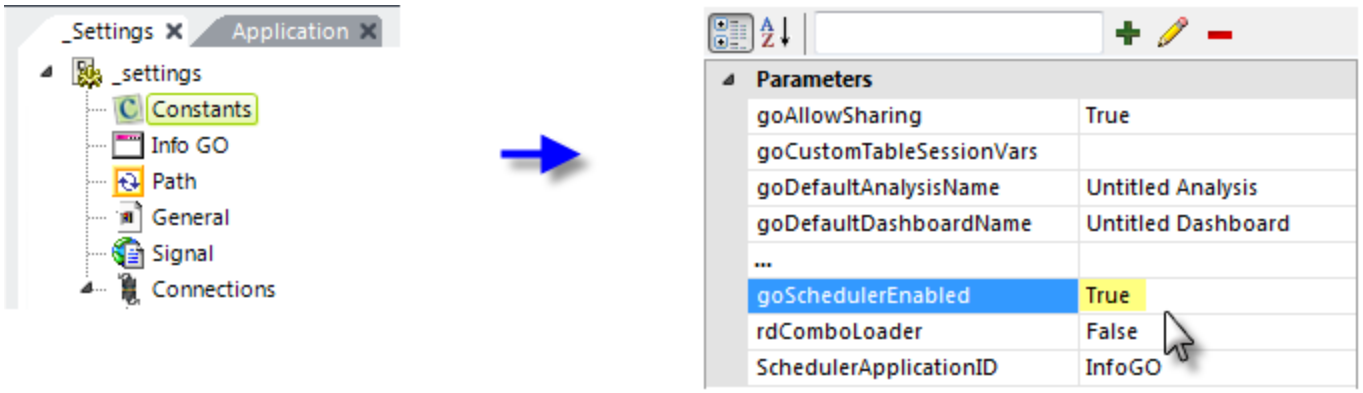
As an optional feature, InfoGo allows you to schedule report production and delivery via email.

The scheduling feature is dependent on the **Logi Scheduler**, a service that must be installed with Logi Info. If the Scheduler is *not* installed, you can re-run the Logi Info installer at any time and install it.

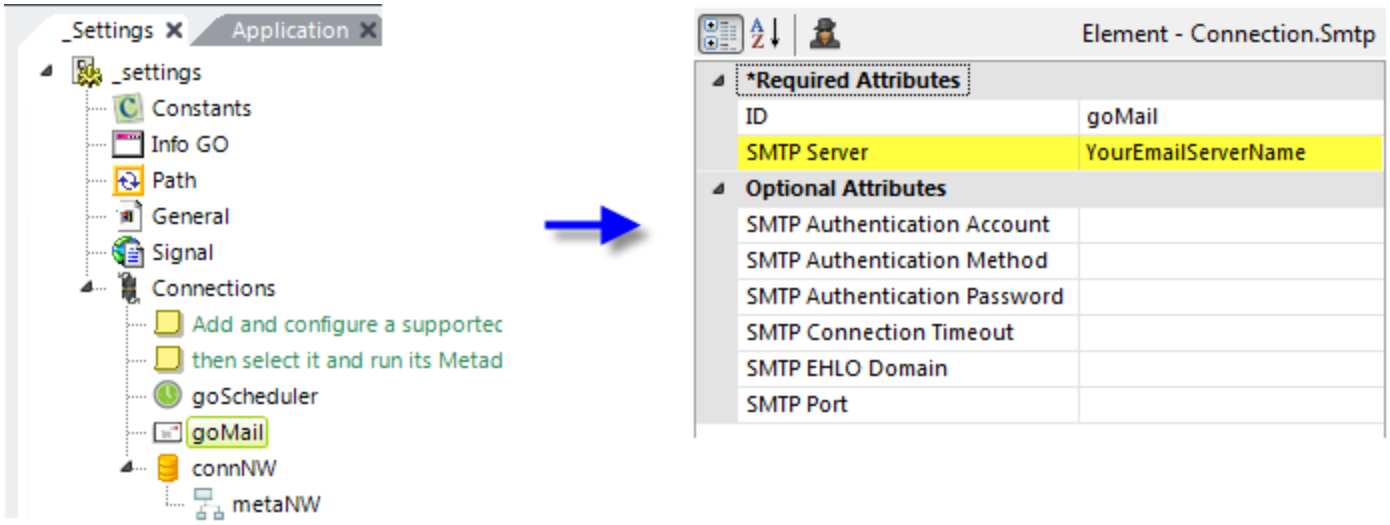
Use the following steps to configure these features:



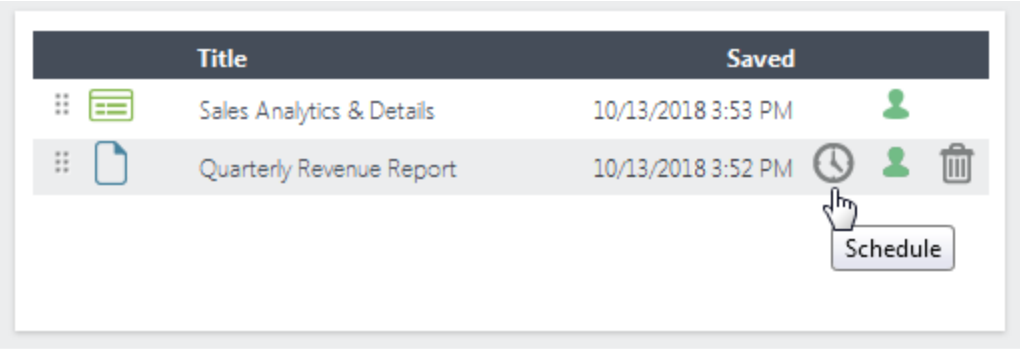
1. Configure the Scheduler: In the InfoGo application's _Settings definition, uncomment the "goScheduler" element, as shown above. Its default attributes will work in most scenarios. For more detailed information about other configurations, see *Using Logi Scheduler*.



2. Enable the Scheduler icons: In the _Settings definition, set the "goSchedulerEnabled" constant to *True*, as shown above. This controls the appearance of the Scheduler icons for items in the Home page.



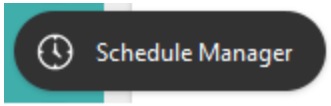
3. Uncomment the "goMail" element and provide an email server name, as shown above.



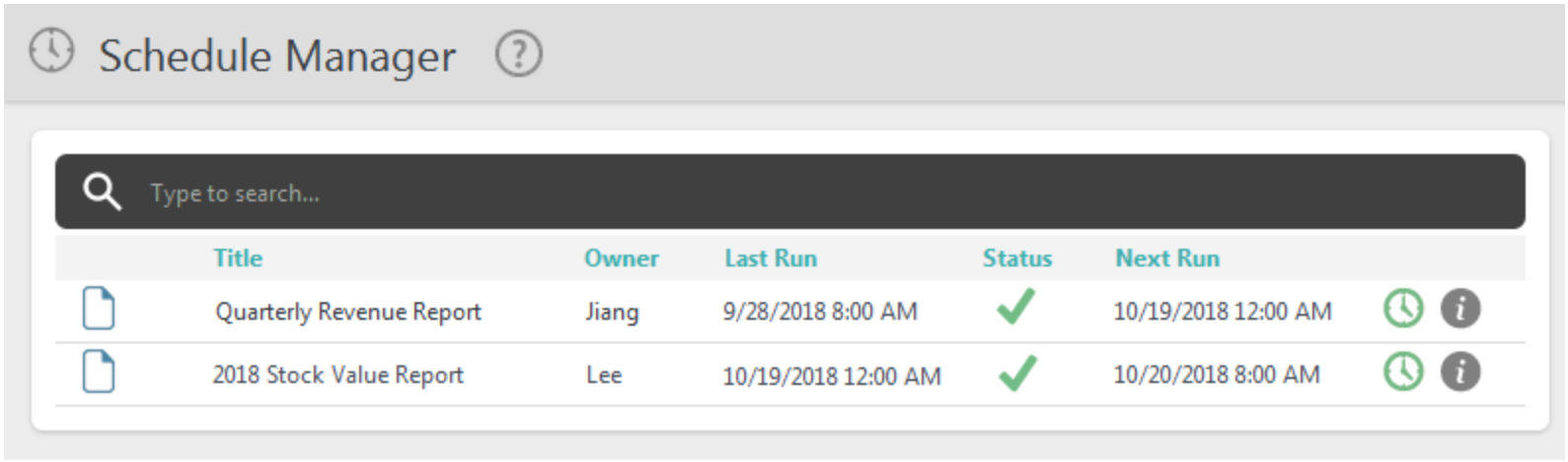
Once scheduling has been configured and enabled, a scheduling icon will appear for each **Report** item in the Home page resource list, as shown above. Users can click this icon at runtime to schedule report production and delivery.

Managing Scheduled Reports

Users with the `InfoGoScheduleManager` security right will see a special option in the side-bar menu:

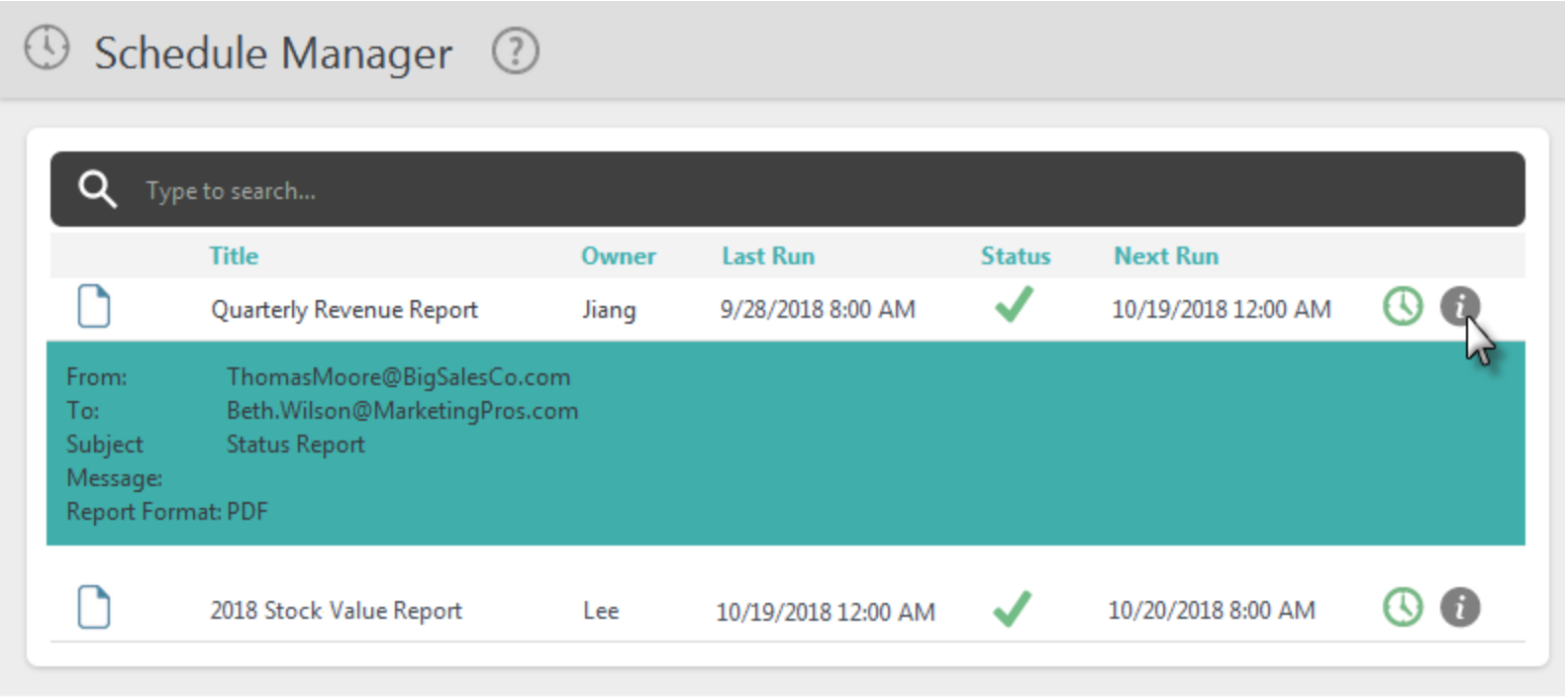


This option launches an embedded instance of the Schedule Manager (the goScheduleManager report definition) and allows management of scheduled events, right inside InfoGo at runtime.



This report definition provides an overview of all scheduled reports, across all users, allowing an administrator to manage the schedules.

Clicking the **information** icon for any entry to toggle a display of its details:

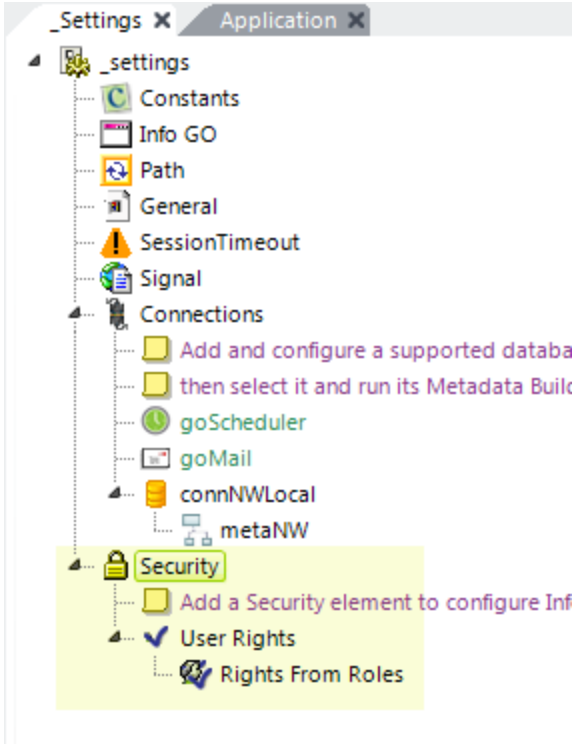


Clicking the **clock** icon for any entry will open the Schedule panel so the details of any scheduled report can be edited. For more information, see *Reporting Scheduling Details*.

Configuring Security

As an optional feature, Logi Security can be used with InfoGo to authenticate users and to control access to application features and data. For information about Logi Security in general, see *Logi Security Scenarios* .

If you do not enable Logi Security then *all* of the features usually controlled by security, including the Data Manager, *will be available to all users*. You can, however, "hide" a feature by hiding its side-bar menu item (by remarking its elements in the InfoGo.goShared definition, under the sharedGoBox shared element) and by renaming the related definition file (to prevent direct browsing).



As shown above, to use Logi Security, add a **Security** element to the _Settings definition, along with whatever child elements are needed for your security scenario and configure them accordingly.

Access to features and data is controlled using the **Security Right ID**, which is an attribute of many elements. For example, the visibility of the Analysis Grid element in a report can be controlled by setting its Security Right ID attribute. You can use this approach to control the availability of many of the elements used in InfoGo. In general, you can assign arbitrary string values for security Rights but these standard Rights have specific purposes:

Security Right	Purpose
InfoGoDataManager	Enables the Data Manager page, allowing the creation and management of connections and metadata from inside InfoGo, and displays an option in the menu. You'll also need to add the default `rdMetadataAdmin` right, or your own custom right, as discussed in *Using the Web Metadata Builder*.
InfoGoReportManager	Enables the ability to select a Global Main Page and to create and share a Global Menu. Also allows editing of shared items.
InfoGoScheduleManager	Enables the Schedule Manager page, allowing the creation and management of scheduled reports from inside InfoGo, and displays an option in the menu.
InfoGoThemeManager	Enables the Theme Editor page, allowing the management of a customized theme from inside InfoGo, and displays an option in the menu.

InfoGo includes a standard logon page, which you can reference in the Security element's Logon Page and Logon Fail Page attributes. You can customize this page which, if the default installation location was used, is

```
C:\Program Files\Logi Analytics\InfoGo\rdLogon.aspx
```

You cannot edit `.aspx` pages in Logi Studio, so you'll need to use some other editing tool, like Notepad.

If you plan to implement sharing of bookmarks, see the security discussion in "Setting Up Sharing" on page 131.

There are two special opportunities to implement **data security** in the InfoGo application. Assuming you have Logi Security enabled:

Table Properties

Tables

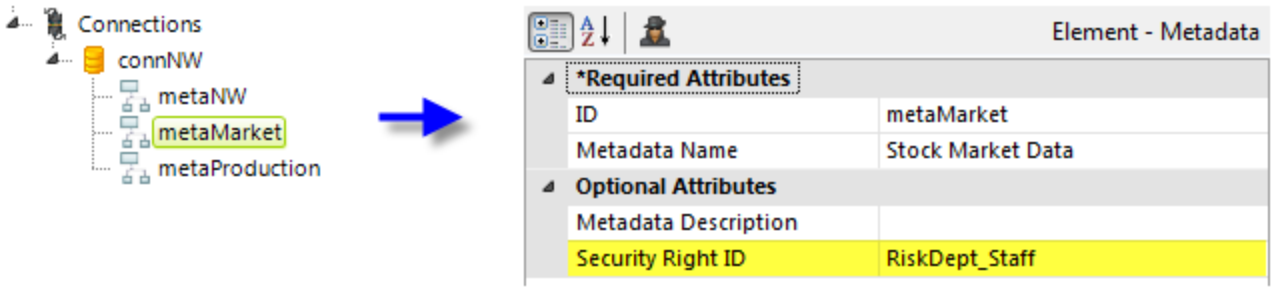
Visible	Name	Friendly Name	Security Right IDs
☑	Categories	Categories	
☑	Customers	Customers	
☑	Employees	Employees	HRStaff, Managers

Column Properties

Columns

Column Name	Friendly Name	Data Type	Format	Alignment	Security Right IDs
ProductID	Product Id	Number	General Number	Right	
ProductName	Product Name	Text	---	---	
SupplierID	Supplier Id	Number	General Number	Right	Executives

1. When using the Web Metadata Builder, you can assign Security Right ID values to tables and to columns, as shown above. These tables and columns will only appear in InfoGo's Analysis page's Data tab if the logged-in user has the appropriate security Rights.



2. The Metadata element has its own Security Right ID attribute. In the _Settings definition fragment shown, there are three Metadata elements in use. Normally, all three would appear in the Analysis page's Data tab, in the Source selector. However, if the metaMarket element is configured with a Security Right ID, as shown above, then the logged-in user will only see it in the Source selector if he or she has the appropriate security Rights.

Configuring InfoGo Constants

The application's _Settings definition includes several InfoGo constants you may want to configure. These constants are:

Constant	Description
ChartAutoZoom	Specifies whether chart zoom is enabled/disabled for Bar Charts, Line Charts, Curve Lines, and Scatter Plots. Default value: *False*.
ChartCanvasAutoDrillTo	Specifies whether or not Self-Service users can drill into data on the Dashboard. Default value: *True*, Drill To feature enabled. To disable, set value to *False*.
goAllowBatchBookmarkDuplication	Specifies whether or not Self-Service users can batch duplicate SSRM bookmarks in their original folder. Default value: *False*. To enable, set value to *True*.
goAllowPauseDataRetrieval	Specifies whether or not to display an extra button on the Analysis tool which, when clicked, temporarily halts data retrieval from the server. This feature is useful when data retrieval is slow and the user prefers to make several changes to the Analysis without having to wait for data after each change. With data retrieval paused, the user can make any desired changes, then click the **Resume** button to resume retrieval. Default value: *False*. When set to *True*, you can also automatically apply Pause Data Retrieval at the application level using the constant `rdAgDataRetrievalPaused`.
goAllowSharing	Specifies whether bookmark folder sharing is enabled. Default value: *False*
goAnalysisType	(Used only when DM is installed) Specifies the analysis tool options that will

Constant	Description
	appear on the menu. Possible values and effects include: *AnalysisGrid* (or if left blank) - The Analysis Grid will be used, and the goAnalysisGrid definition will be shown when this option is selected. *Discovery* - The Thinkspace will be used, and the DM's goDiscovery definition will be shown when this option is selected. *UserSelectable* - Both options will be displayed, allowing the user to choose their analysis tool.
goBatchSelection	Specifies when changes made to chart and table options in analyses will be applied. When *True*, users can make several changes, then click an OK button to apply them. When *False*, changes are applied immediately as each change is made. The batch approach may make for a smoother user experience, especially with large data sets. Default value (or if no such constant exists): *False*
goBookmarkShowItemCount	Specifies whether folders will show item count. Default value: *False*. When *True*, a folder count will display for each folder listed on the SSRM Home page (My Items, My Visualizations, Global Menu, and Shared with Me).
goCustomTableSessionVars	Specifies an optional comma-separated list of session variable names to be saved when adding new bookmarks. Do not enter tokens themselves, just the variable names.  These variables will be saved as static "resolved" values in the bookmark's cus-

Constant	Description
	tom SQL query, rather than being saved as tokens.
goDashboardFilters	Specifies whether Dashboard panels with Analysis Filters, and/or those generated from Analysis Grid analyses that use the Active Query Builder element, will have panel and global filters created for them automatically. Default value: *False.*
goDashboardFiltersFromChartDisabled	When the goDashboardFilters constant is set to *True*, this constant can be used to prevent global filters from being added when Dashboard panel charts are clicked.
goDashboardFiltersOnDemand	Specifies whether the server prepares filter information and user interface components on demand. Default value: *False*. To reduce load time when opening your Dashboard, set this constant to *True*.
goDashboardPanelEditing	Specifies whether editing of visualizations in Dashboard panels at runtime is enabled. When *True*, the Edit option will appear in a panel's gear icon menu (assuming the visualization data source is a candidate for editing). See "Enabling Panel Content Editing" on page 129 for more information.
goDefaultAnalysisName	Specifies the default title that will be used for each new Analysis created. Default value: *Untitled Analysis.*
goDefaultBookmarkCollection	This constant is used by the global features. It's created, and its value is set, by the application. Do not alter it.
goDefaultBookmarkID	This constant is used by the global features. It's created, and its value is set, by

Constant	Description
	the application. Do not alter it.
goDefaultBookmarkReportName	This constant is used by the global features. It's created, and its value is set, by the application. Do not alter it.
goDefaultBookmarkUserName	This constant is used by the global features. It's created, and its value is set, by the application. Do not alter it.
goDefaultDashboardName	Specifies the default title that will be used for each new Dashboard created. Default value: *Untitled Dashboard.*
goDefaultExcludeDetailRows	Specifies whether detail rows will be shown when rows are grouped within the table when using the Analysis Grid. In the table's configuration area, a check box allows excluding of detail rows. Setting this attribute to *True*causes detail rows to be initially excluded (hidden). In some cases, especially when working with very large databases, performance may be better without detail rows. Default value: *False*.
goDefaultHideFunctionNames	When using the Analysis Grid and aggregations in the table, specifies whether the names of the aggregating functions are displayed with the aggregated values in the table summary row.
goDefaultReportName	Specifies the default title that will be used for each new Report created. Default value: *Untitled Report.*

Constant	Description
goDefaultSharedBookmarkID	This constant is used by the global features. It's created, and its value is set, by the application. Do not alter it.
goHideColumnSelection	Specifies whether the column selection panel will be shown in the Analysis page's Data tab after a table is selected. When set to *False*, all columns are automatically selected. Default value: *True.*
goHomeName	Specifies the title of the Home page. Default value: *Home.*
goMetadataIDsAnalysisGrid	Specifies a comma-separated list of one or more IDs of Metadata elements, defined for connections in the _Settings definition, to be used with the Analysis Grid. If left blank, *all* Metadata element IDs from _Settings will be used. This constant is used in the Active Query Builder in the goAnalysisGrid definition.
goMetadataIDsDiscovery	(Used only when DM is installed) Specifies a comma-separated list of one or more IDs of Metadata elements, defined for connections in the _Settings definition, to be used with the DM. If left blank, *all* Metadata element IDs from _Settings will be used. This constant is used in the Active Query Builder in the goDiscovery definition.
goNotificationsEnabled	Controls whether the Notification icon is enabled. When enabled, shared/unshared messages will be generated and recorded. Default value: *True*, disable by setting to *False.*
goReportStarterFile	Specifies a file to be used as the initial Save File for the Report Author in cases

Constant	Description
	when the Save File does not exist yet. For example, when the Save File is different for each user, the Save File Starter file can include a set of default tabs and panels. Example: `@Function.AppPhysicalPath~\SaveFile\InitialReport.xml` Create a Save File Starter file by setting a Save File value, then opening and working with the Report Author to create the desired default set of tabs and panels, then moving and/or renaming the Save File into the Save File Starter location.
goSchedulerEnabled	Controls whether the Schedule icon appears in the Home page list of reports. Default value: *False.*
goSchedulerSessionVars	Specifies a comma-separated list of Session variables to be passed by the Logi Scheduler to a scheduled task during execution.
goSharePermissionEnable	Specifies whether users can use interactive permissions. Default value: *False*. To enable sharing permissions, set this constant to *True*.
goThemeEditorEnabled	Specifies whether custom theme editing is enabled and visible in the Admin menu. Default value: *False*, enable by setting to *True*. See "Customizing Appearance" on page 145.
rdActiveSqlAggregateJoinExclusion	Controls automatic join exclusion for aggregates in ActiveSQL. Default value: *False*, aggregates will be calculated after all joins have occurred. To exclude joins

Constant	Description
	and prevent inflation, set this constant to *True*.
v23.1 rdAGChartGroupDrillthrough	Controls Group Drillthrough capabilities in the Analysis Grid. Default value: *False*. To enable interaction between grouped data columns and rows via drillthrough reports, set this constant to *True*.
rdAgDataRetrievalPaused	Controls the initial status of the Analysis Grid's "Pause Data Retrieval" button. Default value: *False*, data pause is controlled on an individual basis. To automatically pause data retrieval, set this constant to *True*.
rdCalculationsIncludeNulls	By default, columns with Null values are excluded by default when making aggregations. If you want them to be included, create this constant and set it to *True*.
rdDashboardMaxPanelCount	Limits the number of panels that can appear in a Dashboard. If there is no constant, or its value is empty, this function is disabled.
rdDashboardMaxPanelsPerTab	Limits the number of panels that can appear in a Dashboard tab. If there is no constant, or its value is empty, this function is disabled.
rdDisableMetaDataColumnCheck	Controls the ability to disable checking the metadata at runtime. Setting this constant to *True* allows access to bookmarks that reference columns not defined in the metadata source file. Default value: *False.*  When enabled, bookmarks only use the metadata as it existed when it was created, with the exception of the Data tab because that is where the metadata,

Constant	Description
	tables, and columns are selected or changed, and the metadata stored in the bookmark only contains the tables and columns that have already been selected.  When enabled, if you edit the metadata and add or remove Security Right IDs at runtime, those Security Right IDs will be ignored for existing bookmarks.
rdDisableURLMetadataCache	Controls the metadata caching mechanism so that visualizations using the same metadata can be verified by a single shared and cached metadata. Default value: *False*. Setting this constant to *True* closes the caching mechanism.
rdExportDataPreference	Specifies the export preference. Options include: Empty (default), Rapid, and Formatted. Set this global constant to *Rapid* to enable rapid Excel and CSV exports.
v23.1 rdLogFillLevelMessage	Controls the grouping of similar Events in the Debugger Trace Report. Default value: empty or *False*. To group similar Events, set this constant to *True*.
rdHideDataTableHeaderIfNoData	Controls the ability to hide the Data Table/Crosstab Table header when there is no data to show. Default value: *False*. To hide the header when there is no data, set this constant to *True*.
v23.1 rdPdfMultiTableRepeatHeader	Controls whether PDF exports with numerous tables include the table header on each page of the export. Default value: Empty (False). When set to *True*, the PDF displays the table headers on each page of the export.
rdOutputHtmlDivTagDefault	Specifies the default rendering results when the Output HTML DIV Tag of the Divi-

Constant	Description
	sion tag is not set explicitly. Default value: *Empty* (False). When set to *True*, the default rendering result is `<div>`.
rdIncludeFilterInfoInExport	Controls the ability to display filter information in your Excel, PDF, or CSV exports. Default value: *False*. To display filters in your export, set this constant to *True*.
rdIncludeReportTitleInExport	Controls the ability to display the report title in your Excel, PDF, or CSV exports. Default value: *False*. To display the report title in your export, set this constant to *True*.
rdNoDataCaption	Specifies the message that displays in the Data Table/Crosstab Table when there is no data to display. Default value: *None*, no message displays. Enter a custom message as the value for this constant, or, enter "Default" and users will receive the message "No data to display".
rdRapidExportIncludeGroups	Controls whether Group or Summary information is included when exporting via Rapid Excel or Rapid CSV. Default value: *False*. To display Group or Summary information in the Excel/CSV result, set this constant to *True*.
v23.1 rdShowExportDataPreference	Controls end-users' ability to specify an Export Data Preference (Formatted or Rapid) when scheduling an Analysis Grid report. Default value: *False*.
v23.1 rdUpdatePanelTitleWithChartCaption	Controls the ability to update the Dashboard Panel Title automatically according to edits that affect the Visualization Title. Default value: *True*. If you do not want your Dashboard Panel Title to update automatically when editing a visualization,

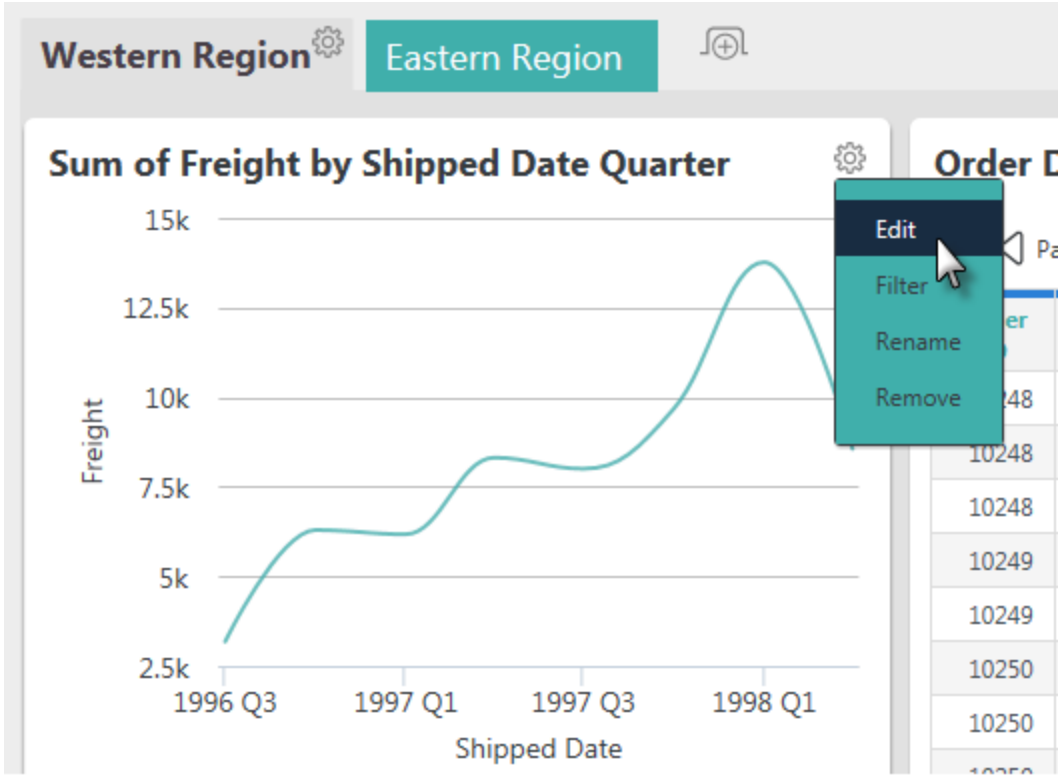
Constant	Description
	set this constant to *False*.
SchedulerApplicationID	Specifies the ID that will be used for this task in the Logi Scheduler. Default value: *InfoGo.*
SchedulerErrorEmail	Specifies the admin email address to send an automatic error notification when a Scheduler Report errors. If there is no constant, or its value is empty, this function is disabled. Multiple addresses are supported, separated by ";".

Other constants, unrelated to InfoGo, may also exist.

Enabling Panel Content Editing

If the `goDashboardPanelEditing` constant has been set to *True*, users can edit the visualizations in Dashboard panels at runtime and save their changes. The **Visualization Editing** element, a child of the Dashboard element in the goDashboard definition, provides the functionality.

Restrictions: Visualization editing is available for Dashboard panels containing charts, gauges, and tables created using Analysis Grid-type analyses, and that Analysis Grid must have been configured to work with Active Query Builder and Active SQL.



When enabled, an *Edit* option is added to a panel's gear icon menu, as shown above. Editing allows the user to adjust visualization options including appearance and column selection. Changes may be saved back to the Dashboard, or not.

Panel content in a Dashboard can come from a variety of sources so, even when enabled, some panels may have the option and some may not.

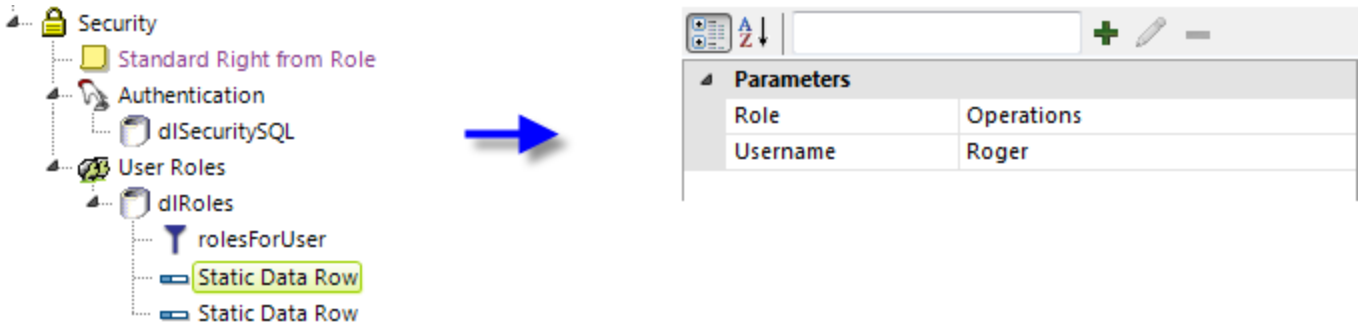
The Visualization Editing element has attributes that can be used to customize the controls presented to the user when editing the visualization. It also supports the use of a "Template Modifier Files" on page 426 to customize the Editing pop-up panel and these template files are used:

For Tables:	For Charts:	For Crosstab Tables:
• rdAgTemplate.lgx • rdDashboardEditAgTable.lgx	• rdAcTemplate.lgx • rdDashboardEditAc.lgx	• rdAxTemplate.lgx • rdDashboardEditAx.lgx

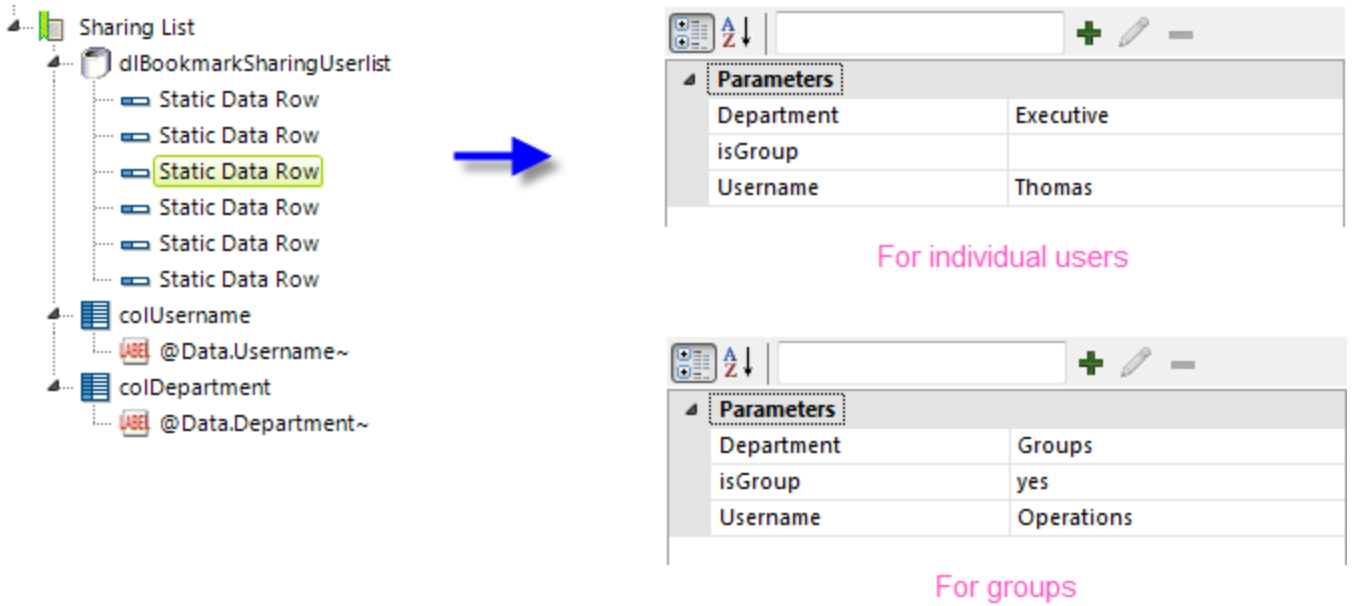
Setting Up Sharing

If the `goAllowSharing` constant described earlier has been set to *True*, users can choose at runtime to share their work with other users and/or user groups. To support this, you need to do two things:

First, if you want to share with groups of users, the security configuration must, of course, identify groups and the users that belong to them.



The example above shows a simple arrangement of security elements that uses a static datalayer to illustrate the assignment of roles, or "group membership", to users. It's through these elements, whether static or from other sources such as Active Directory, that a user is understood to "belong" to a group. We can see above that user "Roger" belongs to the group "Operations".

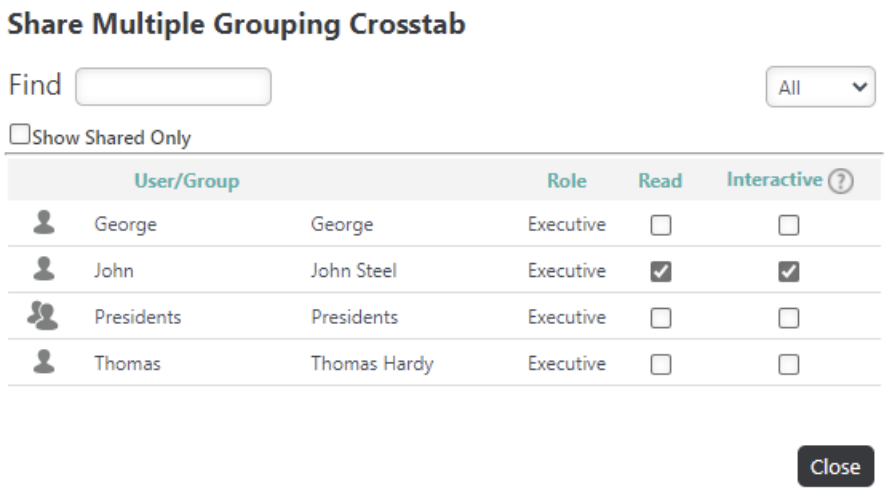


Second, you must configure the **Sharing List** element in the goBookmarkSharingUserlist definition, shown above. Out of the box, it includes an example based on a static datalayer. For sharing with individual users, that datalayer provides *username* and *department* data values and displays those values in two data table columns that appear in the user interface.

The `SharingCollectionDisplayColumn` property can be used to search for users by their first/last name. The default value is empty. Users can set the property value in `goCustomizations.goBookmarkSharingUserlist.lgx`. Once shared, the display name is saved in the bookmark and reflected in the Shared With list.

For sharing with groups, you need to include an additional column for the Sharing List data, as shown above, that's a "this user name is really a group name" flag. When this column value is blank, the username column will be understood to be the name of an individual user; when the column has any other value, such as "yes", the username column will be understood to be a group

name. The name of the group name flag column is entered into the Sharing List element's **Group Identifier Column** attribute. In the example above, that would be "isGroup".



In the InfoGo application, a user will see the pop-up box shown above when they want to share bookmarks. The configuration of the Sharing List element directly affects the box contents.

A drop-down list allows you to filter the list of those with whom a user can share items. The list can be filtered to include People or Groups, or both.

As a content creator, you can now manage access for content shared with other users at the bookmark level. Access types include "Read" or "Interactive". If you are sharing a report, selecting the "Interactive" permission allows the user you are sharing the report with to schedule the report. Granting an “Interactive” permission when sharing a Dashboard or Analysis allows the user to drill-to and apply filters to the shared content; however, they will not be able to save these changes to the original content, as

this can only be done through specific security roles. Note that you must select at least one access type to share content. Selecting the "Interactive" permission automatically selects the "Read" check box, as well. You can edit these permissions by checking/unchecking the corresponding check boxes. Deselecting the "Read" check box will automatically uncheck the "Interactive" check box (if it was selected), and in this case, would unshare the content entirely. Schedules launched by the shared user will be removed if their permission changes to "Read". Likewise, if the report or folder is unshared with that user, schedules launches by the shared user will be removed.

As an administrator, you have the ability to disable/enable the sharing permission feature using a constant. To enable sharing, update the constant `goSharePermissionEnable` to "True". By default, this constant is set to "False".

Element - Constants

goBookmarkShowItemCount	True
goCustomTableSessionVars	
goDashboardFilters	True
goDashboardFiltersFromChartsDisabled	False
goDashboardFiltersOnDemand	True
goDashboardPanelEditing	True
goDefaultAnalysisName	Untitled Analysis
goDefaultDashboardName	Untitled Dashboard
goDefaultExcludeDetailRows	False
goDefaultHideFunctionNames	False
goDefaultReportName	Untitled Report
goHideColumnSelection	False
goHomeName	Home
goMetadataIDsAnalysisGrid	
goMetadataIDsDiscovery	
goNotificationsEnabled	True
goReportStarterFile	
goSchedulerEnabled	True
goSharedMessagesEnabled	True
goSharePermissionEnable	True
goThemeEditorEnabled	False
rdAgDisableMetadataColumnCheck	True

- If you change the constant to "False" after someone has shared a schedule or bookmark using the Interactive permission, the permission will change back to Read-Only.
- The "Interactive" permission is different from edit, described below, as it does not support saving changes back to original bookmark.

For more information, see *Sharing Your Work*.

Editing Shared Items

By default, items that are shared with other users are presented in "read-only" mode to those other users. They can make a copy of them, but they cannot edit the original, shared items. However, other users who have been assigned the `InfoGoReportManager` security Right, *can* edit the original shared items.

Bookmark Storage in a Database

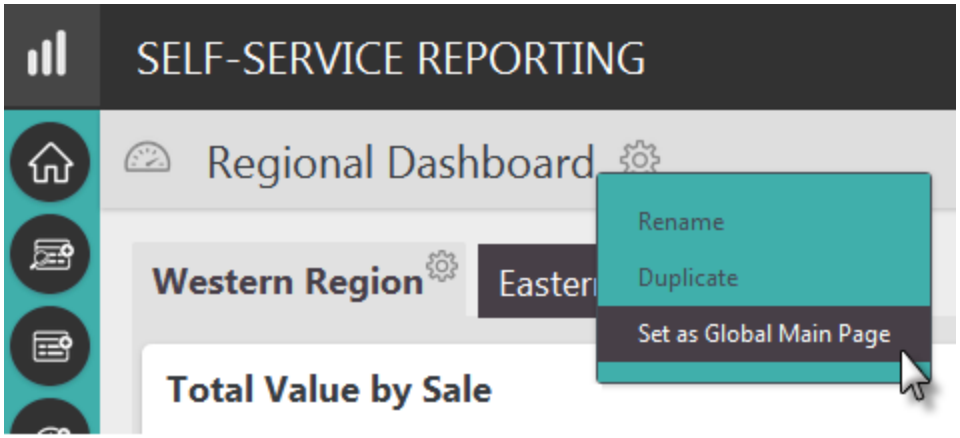
Bookmark, Gallery, and Save files are typically stored as XML files in the web server file system. However, this may not be useful or practical in some InfoGo deployments, so it's possible to store them instead in a SQL database. Logi Studio includes a migration tool and a special **File To Database Mapping** element that make it easy to transfer existing bookmark files into a database, and then use them from there.

For more information about migrating these files, see *Storing Bookmark, Gallery, and Save Files in a Database*.

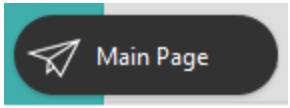
Designating a Global Main Page

InfoGo can be configured to present a "global main page", which will be the starting page all users will see (rather than the Home page) when they first run the application. To set a global main page, the user must have the `InfoGoReportManager` security Right.

If you're upgrading from a previous version and preserved your _Settings file, to make this work you'll need to add a Startup Process element (Process Def File = *InfoGo.goManageReports*, Task = *RedirectToDefault*) to it.



When a user with the appropriate security Right navigates to an InfoGo Dashboard or report and clicks the gear icon, he'll see the **Set as Global Main Page** option shown above. Once the main page is set, the same option becomes **Unset as Global Main Page** to reverse the process, if desired.



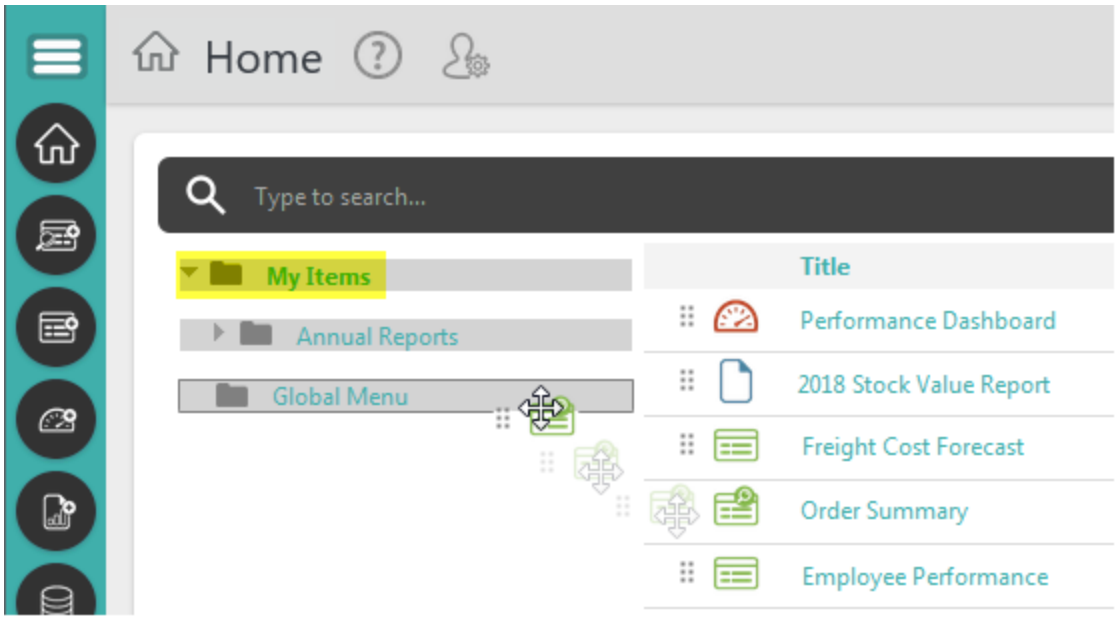
Once a main page has been set, the Main Page option will appear at the top of the menu, as shown above. Users may need to refresh the page or close and restart their browser in order to see it.

If the Dashboard or report set as the global main page has been, or is subsequently, shared with others, they too will see it as their starting page (they do not need a specific security Right to see it).

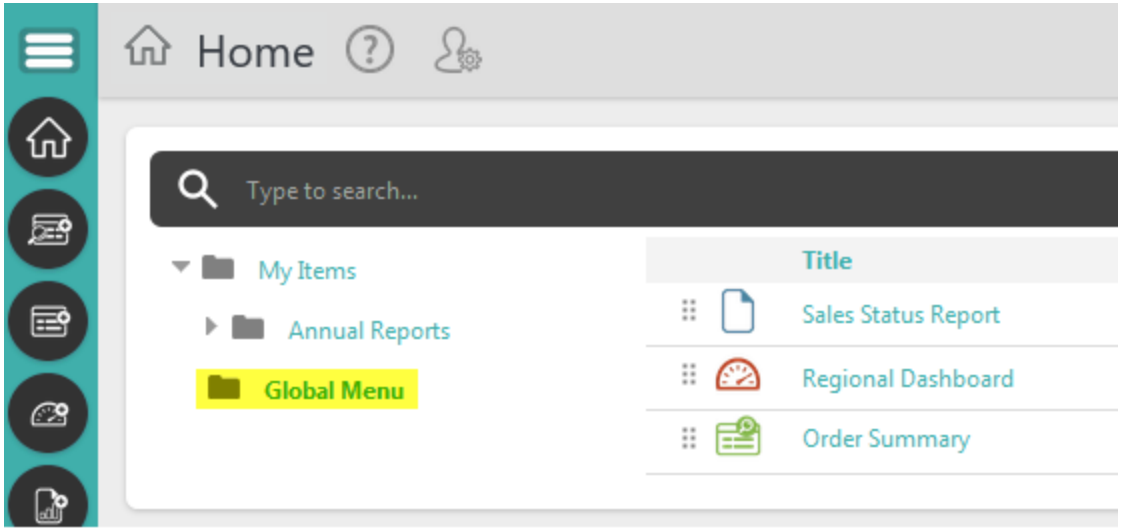
Creating a Global Menu

InfoGo can be configured to present a "global menu", which is a menu of links to selected Dashboards, reports, and analyses, and is available to all users. To create and manage a global menu, a user must have the `InfoGoReportManager` security Right.

If you're upgrading from a previous version and preserved your _Settings file, to make this work you'll need to add a Startup Process element (Process Def File = *InfoGo.goManageReports*, Task = *RedirectToDefault*) to it.

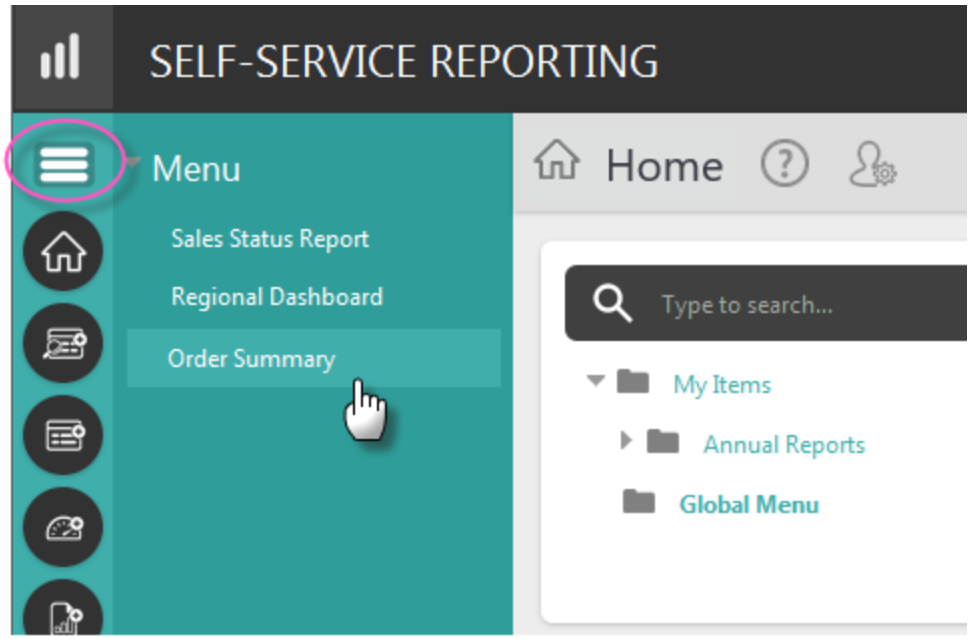


A user with that security Right will see a "Global Menu" folder in their folder list and can drag resources into, just like any other shared folder, as shown above.



If the user selects the Global Menu folder, he'll see its contents and can manage them as usual.

Remember that items in this folder are *not* copies of, or links to, items in the My Items folder. To remove them from the global menu *without deleting them*, drag them back to the My Items (or another) folder.



Once items have been dragged into the Global Menu folder and the browser has been refreshed, a "hamburger" icon will be available at the top of the side-bar menu. When *any* user clicks it, a menu of links to the items in the Global Menu folder will slide out from the left, as shown above. Click again to close it.

Other users do not need the security Right to use the menu, but the Global Menu folder must be *shared* with them. Resources in the menu are automatically available to other users without the need to explicitly share them, but they cannot be modified beyond simple resizing of charts.

As a user drags items into the Global Menu folder, menu items are added to the "shared menu" Shared Element by a plug-in. Customizing the menu for additional folders and/or levels would require rewriting that plug-in and is probably not a realistic goal at this time.

Working with Gallery Files

InfoGo now stores visualizations as bookmarks, not in Gallery or Extra Gallery files. Information about migration of existing gallery items into bookmarks can be found in *Upgrading the Self-Service Reporting Module*.

Users working with InfoGo versions prior to 12.7 can save the charts and other visualizations they create in a "Visual Gallery". By default this is each user's personal collection and, when they subsequently create Dashboards and Reports, they choose the panels and visualizations to include from this private gallery. The visual gallery data is stored in a personal gallery file based on the user's name and InfoGo is configured by default to access this file.

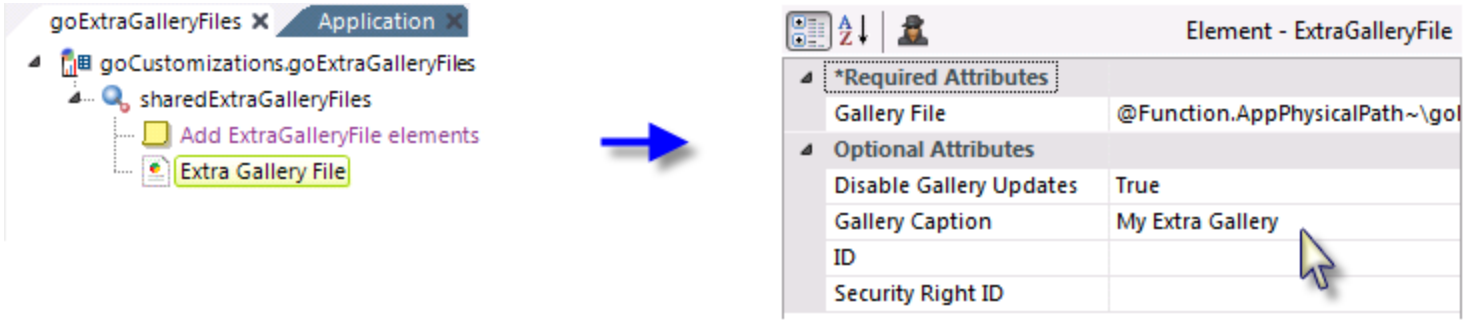
It's also possible to store Gallery and Save files in a SQL database. Logi Studio includes a migration tool and a special **File To Database Mapping** element that make it easy to transfer existing bookmark files into a database, and then use them from there. For more information about migrating these files, see *Storing Bookmark, Gallery, and Save Files in a Database*.

Extra Gallery Files

The **Extra Gallery File** element, a child of the Dashboard and Report Author elements, allows you to specify additional gallery files. When multiple gallery files have been specified and the user is adding panels or visuals to a Dashboard or Report, he'll be able to select them from different visual galleries.

Extra Gallery File elements are added and configured in the goExtraGalleryFiles definition, in the goCustomizations folder in Studio.

How are extra gallery files useful? Imagine a scenario in which a gallery file of pre-configured "standard" visualizations is created and then shared with all users as an extra gallery file. They could then build Dashboards and reports using visualizations from this standard gallery and/or from their personal gallery.



The Extra Gallery File element, shown above, requires a fully-qualified file path and name, with .xml extension, to the file. Other optional attributes allow you to specify whether users can modify the visuals in the gallery (Disable Gallery Update) and specify the name that will appear in the list of gallery choices (Gallery Caption). The element also has a Security Right ID attribute, so access to gallery files can be controlled using security Rights.

Extra gallery files to be shared are typically created by a developer who temporarily manipulates the gallery file path and name in InfoGo manually, then creates and saves the desired visuals. That file path and name are specified in the goAnalysisGrid definition, in the **Custom Dashboard Panels** element's **Dashboard Save File** attribute value. Once the extra gallery has been created, the original file path and name are restored. The file path and name of the extra gallery file are then used in the Extra Gallery File element.

There's also another interesting way to share gallery visualizations: the Extra Gallery File element's **Gallery File** attribute can be configured to point to a regular Logi report definition (.lgx) file. If that definition file includes a Dashboard element with child **Panel** elements containing visualizations, the application will locate those panels in the definition and also make *them* available in the Visual Gallery.

Saving Other Visualizations

As delivered, InfoGo allows users to save visualizations made with its analysis tools to the visual gallery. However, as a developer, you may wish to add elements or definitions to the application which include coded visualizations *not* created with these tools, and you might want to save them to the gallery, too. Here are the steps to do this:

1. Ensure that you have an **Extra Gallery File** element configured in the `goCustomizations.goExtraGalleryFiles` definition. You can add an additional Extra Gallery File element for this purpose, if you need to. Its Gallery File attribute should point to the Dashboard Save File used by the target Dashboard.
2. In the definition with your visualization, add a Link, Button, or Image element with an **Action.Add Dashboard Panel** element beneath it. This will be the element clicked to save the visualization to the gallery.
3. In the Action element's **Add Panel Content Element ID** attribute, specify the element ID of the visualization to be saved.
4. In its **Dashboard Save File** attribute, specify the value from the Extra Gallery File element's **Gallery File** attribute.
5. In its **Image** attribute, specify name of a thumbnail image of your visualization, less than 200 pixels wide.

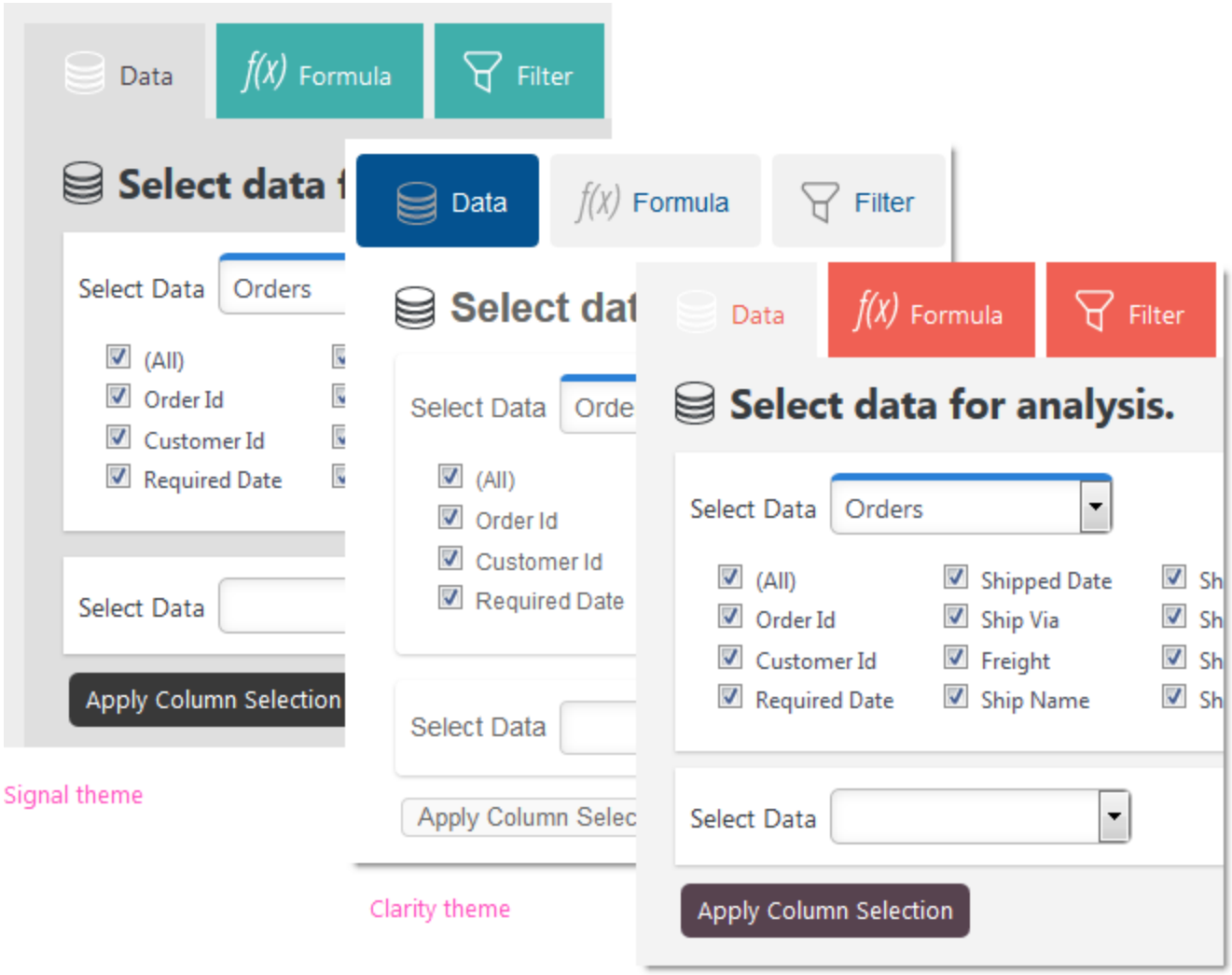
When the report is run and the Save link, image, or button is clicked, the user will prompted for a title and descriptive text and the visualization will be saved to the gallery. From there it will be available for use in a Dashboard or report like any other visualization.

 Visualizations in a *different* Logi application cannot be saved in this way.

More information about the Action.Add Dashboard Panel element is available in "Logi Info Dashboard " on page 351.

Customizing Appearance

You can change the basic look of the application by changing its theme, which is set in the _Settings definition.



Signal theme

Clarity theme

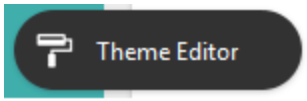
Arizona theme

The default theme is *Signal* but, as shown above, you can use other themes. More information about themes can be found in *Working with Themes*.

Editing Themes

Custom themes can be created using the **Theme Editor**. This is most easily done by the developer, using the Theme Editor wizard in Logi Studio. When a custom theme is used in the InfoGo application, administrative users can be allowed edit it at runtime. For more information about creating a custom theme, see *Using the Theme Editor*.

To enable an administrative user to edit a custom theme while using the InfoGo application, the `goThemeEditorEnabled` constant must be set to *True* and the user must have the `InfoGoThemeManager` security Right. Once this is done, the Theme Editor will appear as an option on the menu:



This option launches an embedded instance of the Theme Editor and allows modification of an existing custom theme right inside InfoGo at runtime.

The embedded Theme Editor will *not* edit standard built-in themes, like Signal. It will only edit "custom" themes, which are copies of standard themes.

A custom theme can be created using the Theme Editor wizard in Logi Studio, or it can be created manually:

1. Copy a standard theme folder from: `C:\Program Files\Logi Analytics\InfoGo\rdTemplate\rdTheme` to `C:\Program Files\Logi Analytics\InfoGo\_Themes` and rename the folder with your custom theme name (you may

need to create the `_Themes` folder).

2. In InfoGo's _Settings definition, set the **Global Style** element's **Theme** attribute to your custom theme name.
3. The embedded Theme Editor will now be able to edit the custom theme.

Any theme customizations are not user-specific; they will affect the appearance of InfoGo for *all* users.

Customizing InfoGo Files

As an optional feature, InfoGo can be customized or "branded" in several ways to make it look like "your" application.

All of the following definitions appear in the goCustomizations folder, in Logi Studio's Application panel:

Reports	Customization Opportunity
goBookmarkSharingUserlist	Used to specify an optional list of user names, with other information, that will be available for selection when sharing folders with users and groups.
goEmailAddressList	Used provide a list of email addresses, allowing the user to make address selections, in addition to entering them.
goExtraGalleryFiles	Used to specify customized Extra Gallery File elements in order to include globally-shared custom gallery items.
goWebFooter	Used to customize copyright statement and other footer information.
goWebHeader	Used to customize logo, application title, and user icons. Includes icon for sidebar menu.

Support Files	Customization Opportunity
Custom.css	Add your own custom CSS classes here; be sure to include this file using a Style or Global

Support Files	Customization Opportunity
	Style element.
Custom.html	Include custom HTML for insertion within the HTML `<Head>` tags. Among other things, this is where you can specify the location of your Favicon file.
Custom.js	Include custom JavaScript for use within the application.
dmfAnlysisGrid.xml	Definition Modifier code for the goAnalysisGrid definition. A reference to this already exists in the definition.
dmfDashboard.xml	Definition Modifier code for the goDashboard definition. A reference to this is already exists in the definition.
dmfHome.xml	Definition Modifier code for the goHome definition. A reference to this is already exists in the definition.
dfmMaster.xml	Definition Modifier code for the goMaster definition. A reference to this is already exists in the definition.
dmfReport.xml	Definition Modifier code for the goReport definition. A reference to this is already exists in the definition.
dmfSchedulerManager.xml	Definition Modifier code for the goScheduleManager definition. A reference to this is already exists in the definition.

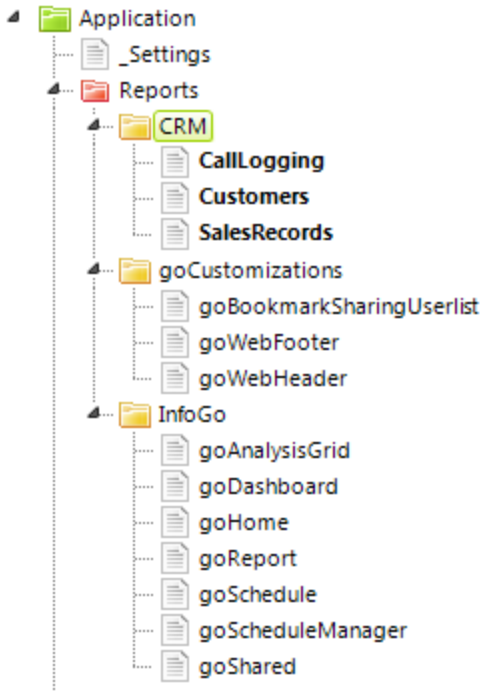
Support Files	Customization Opportunity
tmfBookmarkOrganizer.xml	Template Modifier code for the goHome definition. A reference to this is already exists in the definition.

Embedding InfoGo

The InfoGo application can be embedded into other Logi applications or into non-Logi HTML pages.

Using InfoGo as the Base Application

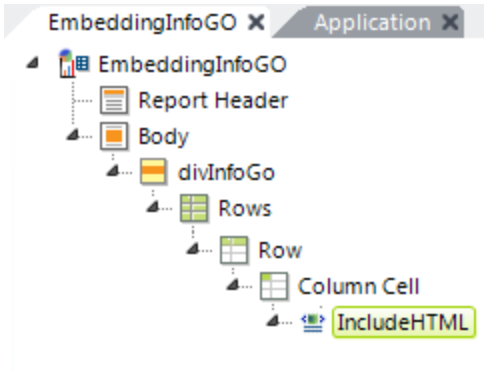
One of the simplest ways to embed InfoGo is to use it as the base application, then add your own definitions and supporting files to it. This lets you "build out" the application with your own code and refer to InfoGo definitions directly.



In the example above, Studio's **Application Panel** shows an additional folder ("CRM") and definitions that have been added to InfoGo in order to use this approach. It has the advantage of a single code base, does not use an iFrame, and the security scheme in InfoGo can be used by the rest of the application.

Embedding in an iFrame without SecureKey

The following example shows one way to embed InfoGo into another Logi application using an iFrame, but without using Logi Secure Key.



As shown above, an **Include HTML** element is added to the report definition. Width control is exercised by placing it inside a container, like an HTML table Column Cell or a Division, whose width can be set. Here's the HTML code to include:

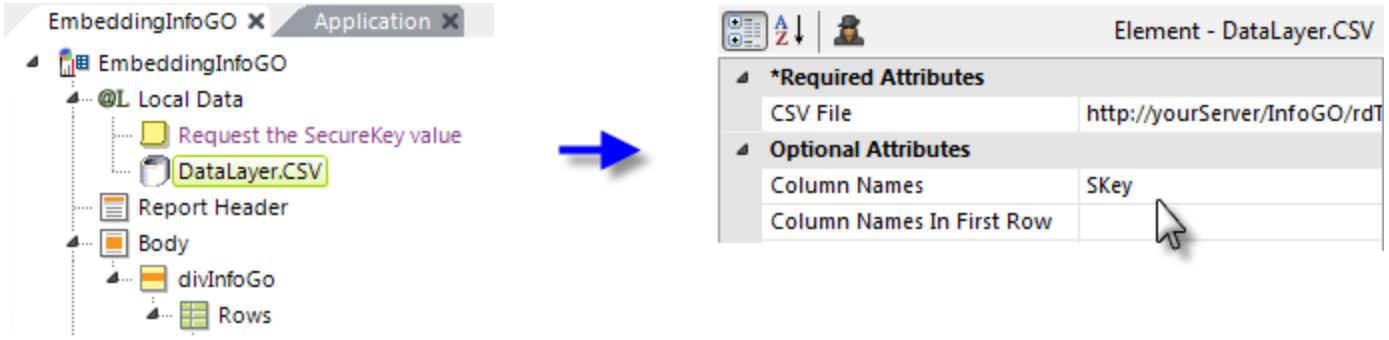
```
<script language="javascript" type="text/javascript">
function resizeIframe(obj) {
 obj.style.height = obj.contentWindow.document.body.scrollHeight + 'px';
 }
```

```
 </script>

<iframe src='http://yourServer/InfoGo' style="width: 100%; min-height: 1200px;
 frameborder: false;" frameborder='0' onload='javascript:resizeIframe(this);' "
 scrolling="no">
</iframe>
```

The same code can be used to embed InfoGo into a non-Logi application.

Embedding in an iFrame with SecureKey

If you want to embed InfoGo using an iFrame and make use of Logi SecureKey authentication, you first have to get the SecureKey value from the InfoGo application.



One way to do this is shown above. Under a Local Data element, **DataLayer.CSV** is used to retrieve the key value. This datalayer type is used because it can use a URL to retrieve the data. Its **CSV File** attribute value will be similar to:

```
http://y-
ourServer-
/In-
foGo/rdTem-
plate/rdGetSecureKey.aspx?UserName=@Function.UserName~&UserID=@Function.UserID~&Roles=@Function.UserRoles~
```

That retrieves one row into Local Data, with a column named `SKey`, containing the SecureKey value. Then, using the same embedded script from the previous example:

```
<script language="javascript" type="text/javascript">
function resizeIframe(obj) {
 obj.style.height = obj.contentWindow.document.body.scrollHeight + 'px';
 }
 </script>

<iframe src='http://yourServer/InfoGo/?rdSecureKey=@Local.SKey~' style="width: 100%;
 min-height: 1200px; frameborder: false;" frameborder='0'
 onload='javascript:resizeIframe(this);' "scrolling="no">
</iframe>
```

Add the `rdSecureKey` parameter and Local Data token, shown in blue above, to the iFrame source URL to complete the authentication process when embedding the InfoGo application.

Develop with the Discovery Module 3.2

The **Discovery Module** (DM) is an add-on module for Logi Info, that enables special elements in Logi Info and provides an embedded discovery experience that works directly with relational data sources.

The following topics are intended for *Logi developers* and discusses how to configure an application after the DM has been installed:

- Connecting to the Logi Application Service
- Using The Thinkspace Element
- Configuring Thinkspace Bookmarks
- Configuring Add-to-Dashboard or -Gallery
- Embedding Visualizations

About the Discovery Module

 Advanced features discussed here work with Logi Info v12.5 and later. Earlier and later Info versions may not support them; consult the Release Notes for specific details.

Introductory information about the DM and directions for installing it can be found in our installation topics for *Installing the Discovery Module - Windows* and *Installing the Discovery Module - Linux*. If you haven't read it already, we recommend you do so before proceeding.

Once installed, the DM makes special services and elements available in Logi Studio, controlled by various licensing combinations. These include Logi Platform Services and the **Thinkspace** element, which allows end-users to work with its interface to get a rich,

highly-interactive analysis experience. Best-fit visualizations are automatically suggested by a built-in "recommendation engine" and data is organized using intelligent technology that clusters data to make it more easily consumable by users.

Important Restrictions

Please note these important restrictions on the use of the DM 3.1+:

- Client browsers: Chrome 26+, IE 11+, Edge 40+, Firefox 20+, and Safari 7+.
- Server OS: 64-bit versions of Windows Server 2012, Windows 7, or Linux (Java 1.8 capable)
- Server: 4GB RAM minimum, 8GB RAM recommended (more if RAM also used for a VM)
- Server: 4GB hard disk space available
- Logi Info: v12.5, 64-bit version (there is no 32-bit DM version available)

The supported databases are:

- AWS Redshift
- Infobright
- JDBC database providers
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL
- Snowflake
- Vertica

Logi Info .NET applications that include the DM must run using an IIS Application Pool configured for *Integrated* pipeline mode. *Classic* pipeline mode Application Pools will *not* work.

Special Application Services

The DM uses two special application services on the web server:

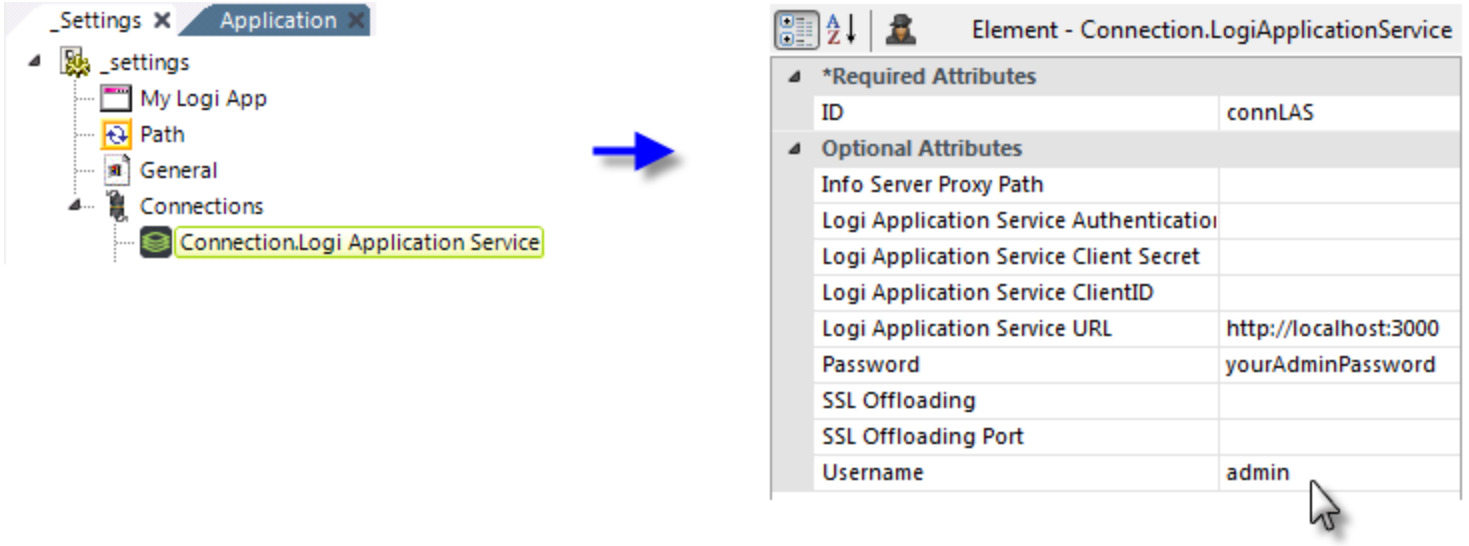
- An embedded, runtime environment for server-side web applications (node.js)
- An embedded, Java-based relational database (H2)

Node.js and H2 are extremely compact, have a small installation footprint, and do not install a lot of extra tools or management utilities. They're provided as transparent services that require little or no management for use with the DM.

Two Windows Services or Linux daemons are created during the installation: *Logi Application Service* and *Logi Data Service*.

Connecting to the Logi Application Service

The DM installs and uses the *Logi Application Service* and your Logi application uses a special Connection element to connect to this service:



The **Connection.Logi Application Service** element, in your _Settings definition, is used for this purpose, as shown above. This element will only be present in Studio *after* the Discovery Module is installed.

The element's attributes are:

Attribute	Description
ID	(Required) Specifies an element identifier, unique within the application.

Attribute	Description
Logi Application Service URL	(Required) Specifies the URL for the Logi Application Service. A server name or IP address may be used and a port value is required. For example: `http://192.168.999.999:3000` `http://localhost:3000`If you want to use a different port number, you'll need to also adjust the port value in the service settings to match it. See the *Customizing Service Settings* for installing the Discovery Module for Windows, or *Customizing Service Settings* for installing the Discovery Module for Linux.
Info Server Proxy Path	The Logi Info Engine usually determines the path to call the Logi Application Service based on the current request context. However, in certain cases based on server configurations, you may have to explicitly specify the local Logi Info server path. The format for this value can be: `http://localhost:port/applicationName` or `http://machineName:port/applicationName`
Logi Application Service Authentication Type	Specifies the authentication method to be used to access the Logi Application Service. *Standard* authentication (the default) uses the values of the Username and Password attributes to connect to the service. *TrustedAccess* authentication uses the values of the Client ID and Client Secret attributes, along with the current Logi Security UserName, to request an accesstoken. This access token is then used for all subsquent interactions. If this value is selected, the Username and Password attribute values are ignored. See the section below for details.
Logi Application Service Client Secret	Specifies the Client Secret value returned when registering an application client with the Logi Application Service and used when the Logi Application Service Authentication Type = *TrustedAccess*. Administrative users of the Dataview Authoring tool can create a Client Secret using the Client Credentials menu item beneath their avatar. See the section below for details.

Attribute	Description
Logi Application Service Client ID	Specifies an arbitrary Client ID value used when registering an application client with the Logi Application Service and used when the Logi Application Service Authentication Type = *TrustedAccess*. See the section below for details.
Password	Specifies the password needed to connect to the Logi Data Service, which was established when Logi Platform Services was installed. Ignored if the Logi Application Service Authentication Type = *TrustedAccess*.
SSL Offloading	Set this value to *True* if your Logi Info application sits behind a load-balancer with SSL Offloading enabled. It will build a call to the Logi Application Service using the HTTPS protocol, even if the Info application itself does not have SSL enabled.
SSL Offloading Port	If a load balancer is used and it redirects to an info application which is on a different port number, use this attribute to specify the port number. This value will used when building the URL to call the Logi Application Service.
Username	Specifies the Username needed to connect to the Logi Data Service, which was established when Logi Platform Services was installed. This is typically *admin.* Ignored if the Logi Application Service Authentication Type = *TrustedAccess*.

Using Trusted Access Authentication

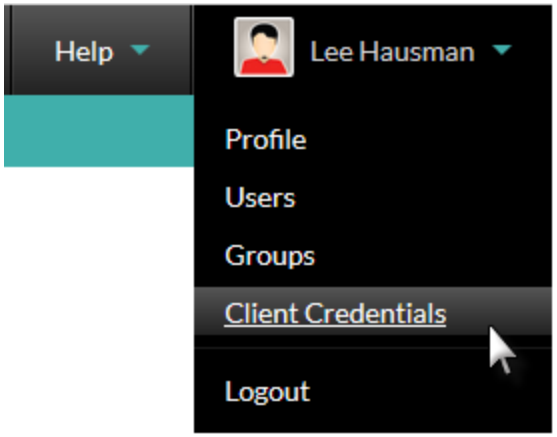
The example above shows the connection to the Logi Application Service (LAS) secured with Standard authentication, using a username + password scheme. However, **TrustedAccess** authentication, is also available. It's a security scheme that doesn't expose security credentials as plain-text in the _Settings definition and provides more secure communications.

TrustedAccess authentication uses the values of the Connection element's **LAS Client ID** and **LAS Client Secret** attributes, along with the current Logi Security user name, roles, and rights to request an access token. This access token is then used internally for all subsquent interactions with the service.

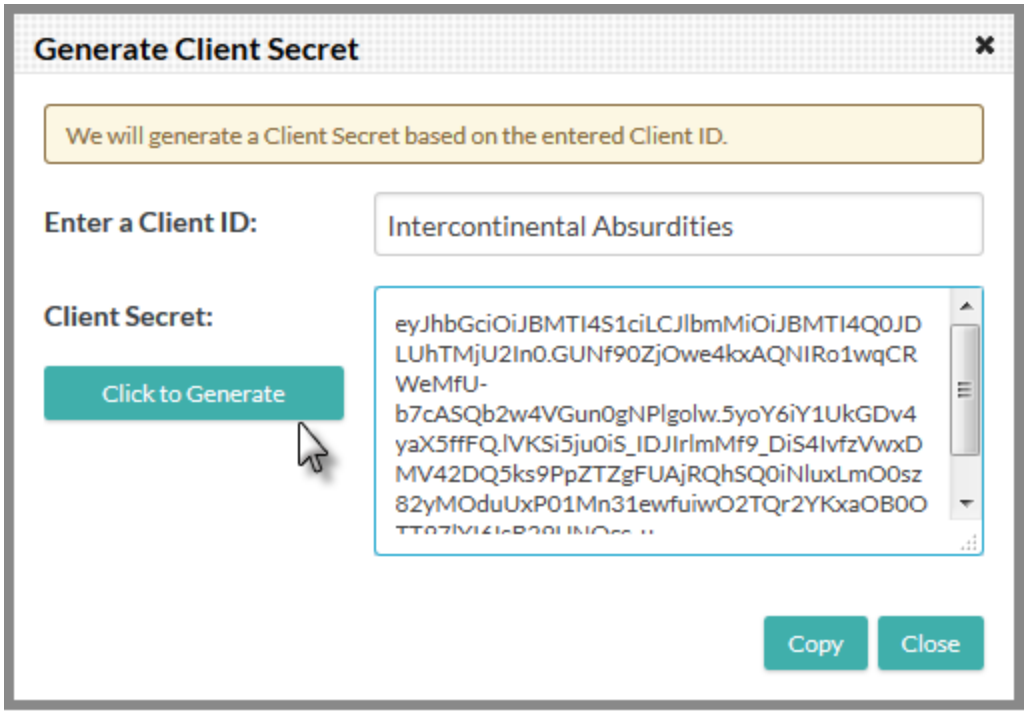
In order to use TrustedAccess, you must have a **Security** element, properly configured and enabled in your _Settings definition, to get the current user name, roles, and rights, which are then provided to the LAS. Any **Authentication Source** setting can used, and providing a value for the **Development Username** attribute is also a valid way of temporarily providing a user name.

Next you need to generate a value for the Connection element's LAS Client Secret attribute using the Dataview Authoring tool in Logi Studio. To do this:

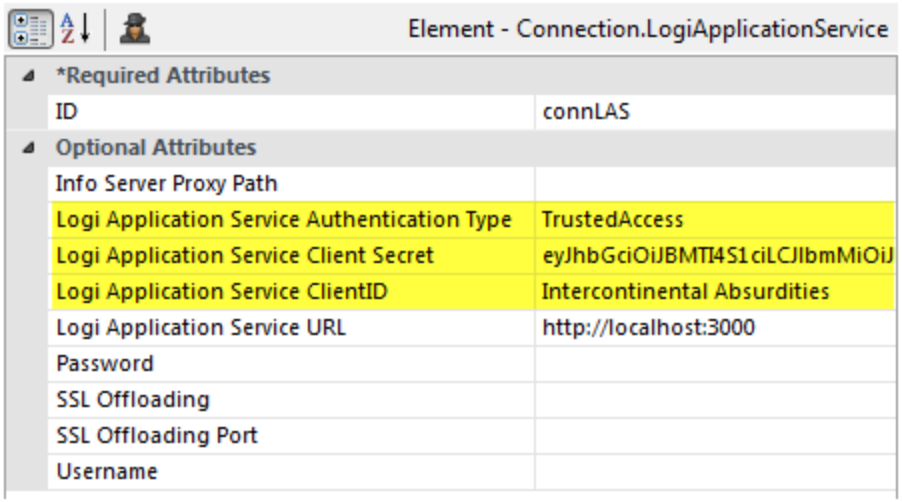
1. Ensure that the Connection element's **LAS Authentication Type** is blank or set to *Standard*.
2. Launch the Dataview Authoring tool from the Studio Tools menu tab.
3. Log into the tool with an administrator account. This is typically *admin* and the password that was established when Logi Platform Services was installed.



4. Pull-down the submenu under your username in the menu bar, as shown above, and select *Client Credentials*.



5. The Generate Client Secret dialog box, shown above, will appear. Enter an arbitrary string for the Client ID and click the **Generate** button. The Client Secret code will be generated and displayed. Click **Copy** to copy the code to your Clipboard. Click **Close** and logout of the Dataview Authoring tool.



6. Back in your _Settings definition, set the Connection element's **LAS Authentication Type** to *TrustedAccess*, as shown above.
7. Double-click the **LAS Client Secret** attribute name to open its Zoom window. Paste the Client Secret you copied in Step 5 into the window and click **OK** to close it.
8. Enter the Client ID you used in Step 5 in the **LAS Client ID** attribute value.
9. Save your definition. Any values in the Username and Password attributes will now be ignored.

Your Trusted Access configuration is now ready for use.

Connecting to Data

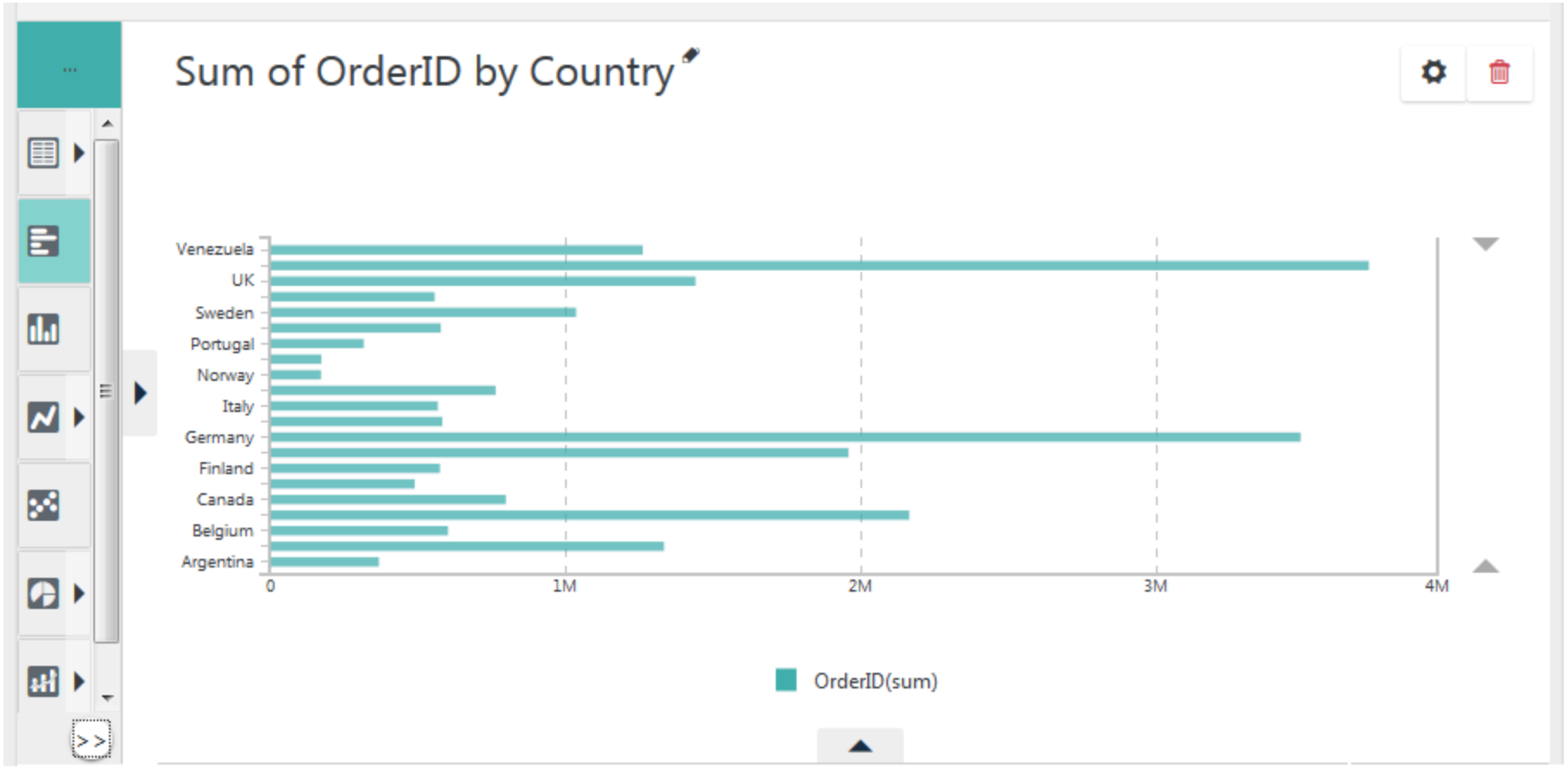
When using Logi Platform Services techology features such as *Dataviews* and *SuperWidgets* connections to the data are-configured in the Dataviews.

Using the Thinkspace Element

Logi Platform Services features provide a seamless visualization and Dashboard creation capability and is highly recommended. However, as a developer, you can also implement similar functionality using elements such as the Thinkspace.

The Thinkspace element has a user-interface, a built-in "recommendation engine", and automatic data grouping capabilities that help users discover insights in their data. The data is defined by a query in a child datalayer. It allows users to:

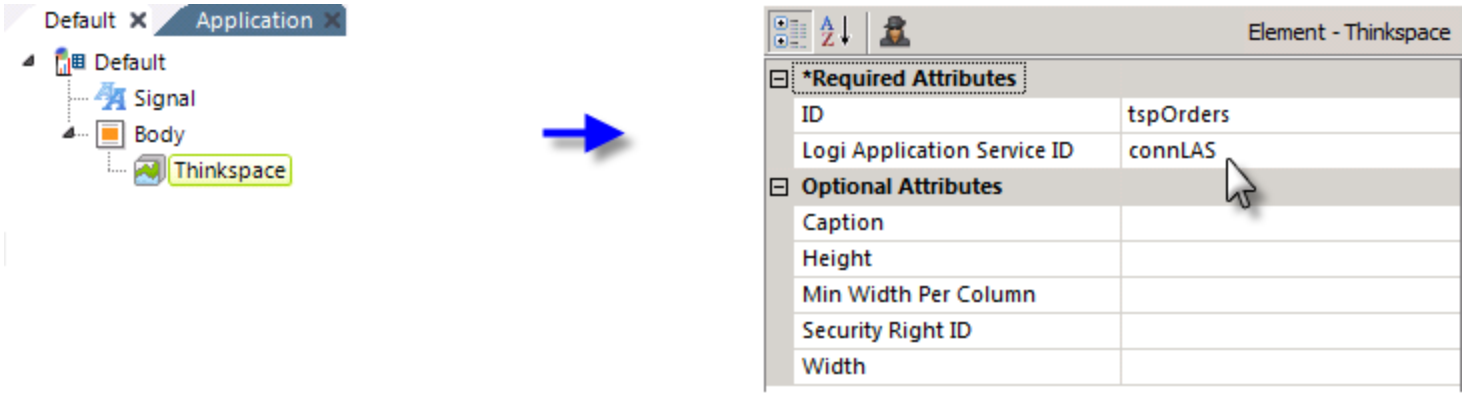
- Filter and aggregate their data in a table.
- Create charts with drag and drop simplicity. The recommendation engine automatically creates a best-fit chart while also allowing users to select their own visualizations.
- Benefit from binning capabilities that group data in easy-to-understand bins, enhancing comprehension of the data set.
- Add visualizations created in the Thinkspace to a Dashboard or visual gallery for further use and sharing.



An example of the Thinkspace in use is shown above. The element offers a rich user-interface with a lot of functionality, and requires very little configuration from developers (in fact, it has very few configuration options). Earlier versions may look slightly different.

Technically, it's a not a "super-element" but it has a lot of the same qualities.

The settings and changes a user makes while working with the Thinkspace are *not* automatically saved between sessions. It's possible to build the application so that users have opportunities to save their data and this is discussed in later sections.



In the example shown above, the **Thinkspace** element has been added to a report definition. It's configured by giving it a unique element ID and specifying the ID of the Connection.Logi Application Service element added to your _Settings definition earlier.

Attributes

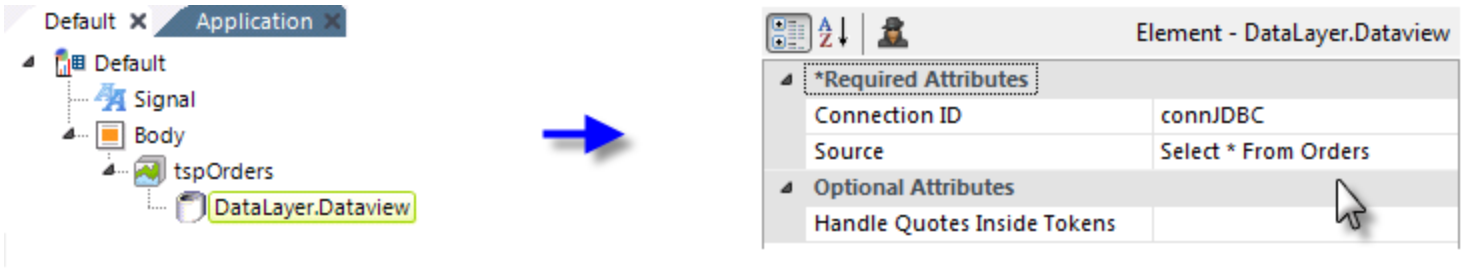
The Thinkspace element has the following attributes:

Attribute	Description
ID	(Required) Specifies an element identifier, unique within the definition.
Logi Application Service	(Required) The ID of a Connection.Logi Application Service element, in the _Settings definition.

Attribute	Description
ID	
Caption	Specifies a Caption to be displayed for the element.
Height	Specifies the height of the element, in pixels. If left blank, the element will be automatically sized to the height of its content.
Min Width Per Column	Specifies a minimum width, in pixels, for the data columns displayed.
Security Right ID	If entered, controls access to this element via Logi security. Supply the ID of a Right defined in the application's settings/security section. Only users that have a Role referenced in the Right will be able to see the element. (Be careful - when the Right is not defined in the settings, the element is visible.) Multiple Right IDs, separated by commas may be entered. In this case, the user will see the element if he has access to any one of the Rights.
Width	Specifies the width of the element, in pixels. If left blank, the element will be automatically sized to the width of its content.

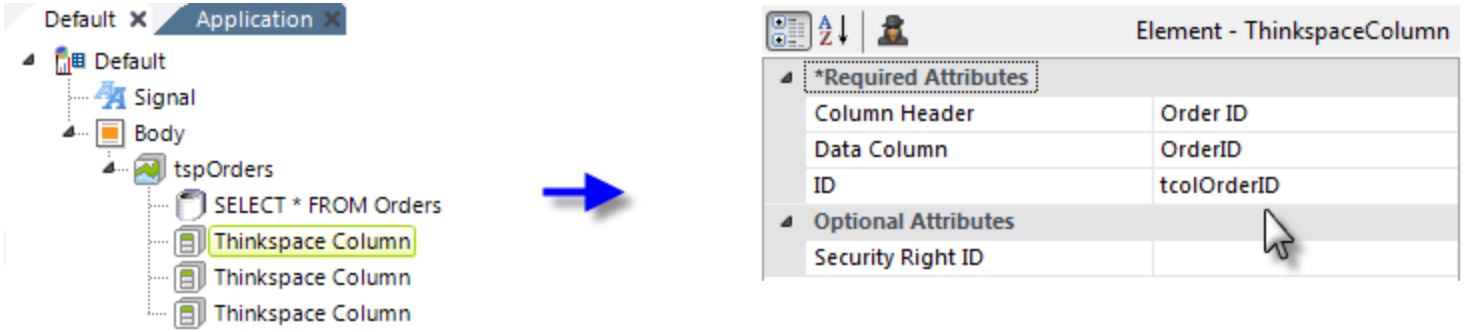
Retrieving Data for the Thinkspace

To retrieve data for the Thinkspace, you use a special datalayer and the Connection.JDBC element discussed earlier:



Add a **DataLayer.Dataview** element beneath the Thinkspace element, as shown above. Configure it to use the Connection.JDBC element from the _Settings definition and provide a valid SQL query for it.

Once the data is retrieved, *all* columns in the datalayer will be displayed in the Thinkspace. The column header text displayed and the initial column order will match that found in the datalayer.

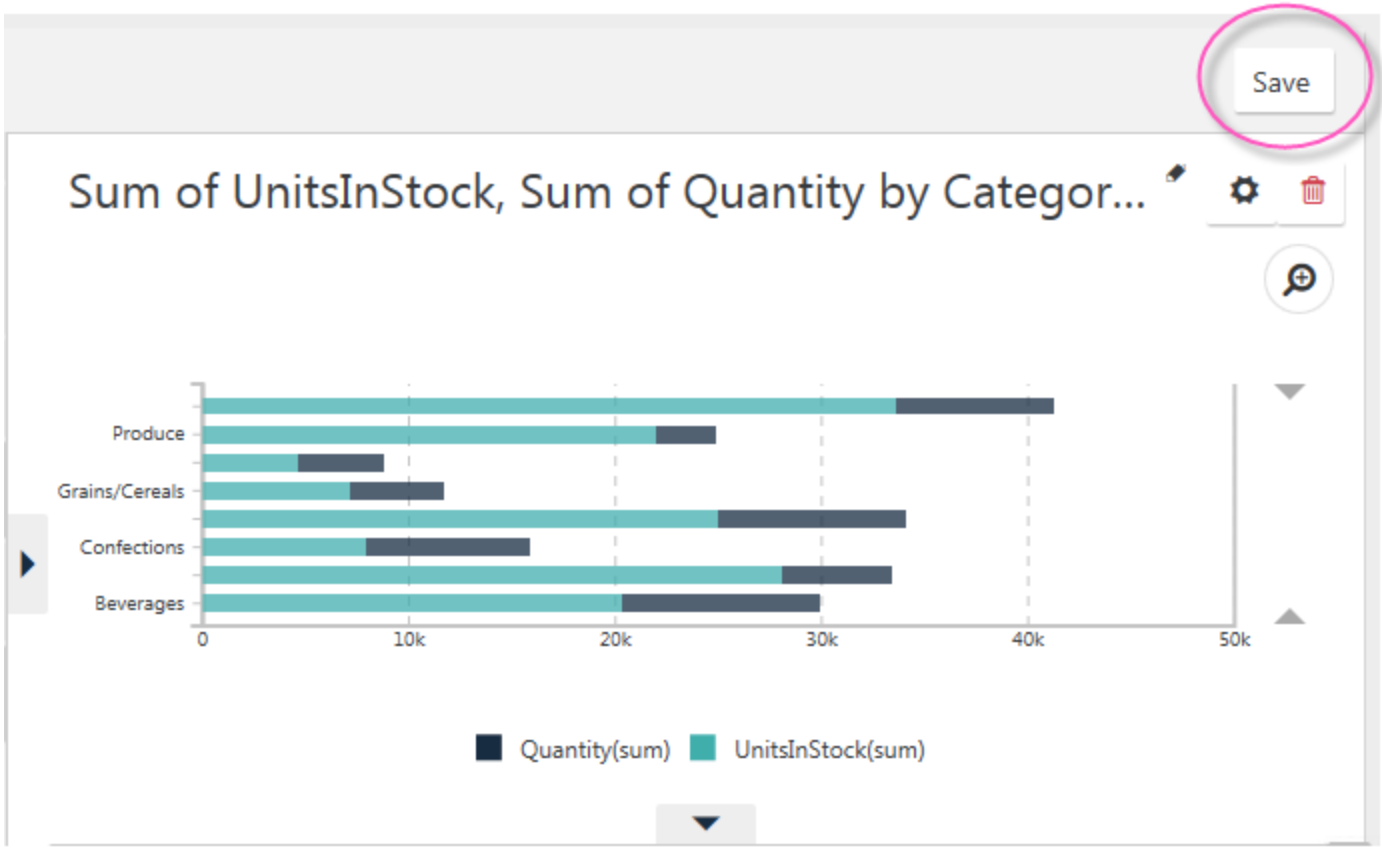


If you want control over the column order, header text, and security restrictions, you can use **Thinkspace Column** elements, as shown above. Add one for every column you want displayed and set their attributes as desired. Other columns not represented by these elements will not be displayed.

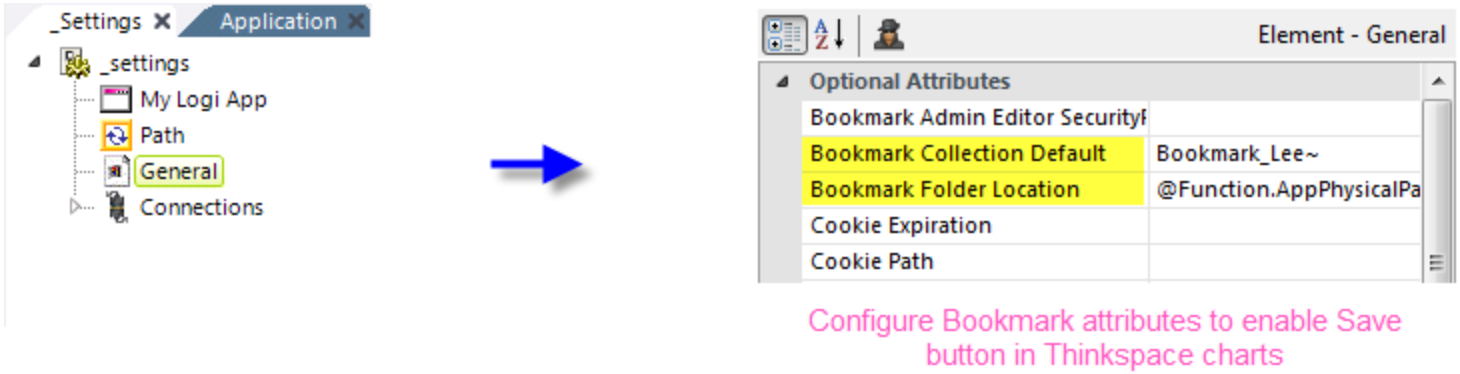
Configuring Thinkspace Bookmarks

Users may want to save the Thinkspace visualizations they've created, for re-use in later sessions. This can be done using *Bookmarks*.

A bookmark is a mechanism for storing and re-using request parameters, user input data, and configuration choices made at runtime. More information about bookmarks are available in *Bookmarks*.



When properly configured, Thinkspace charts will display a **Save** button, as shown above. When users click it, a "snapshot" of their runtime settings is saved in a bookmark. The snapshot can be retrieved in a later session to reproduce their settings and visualization.

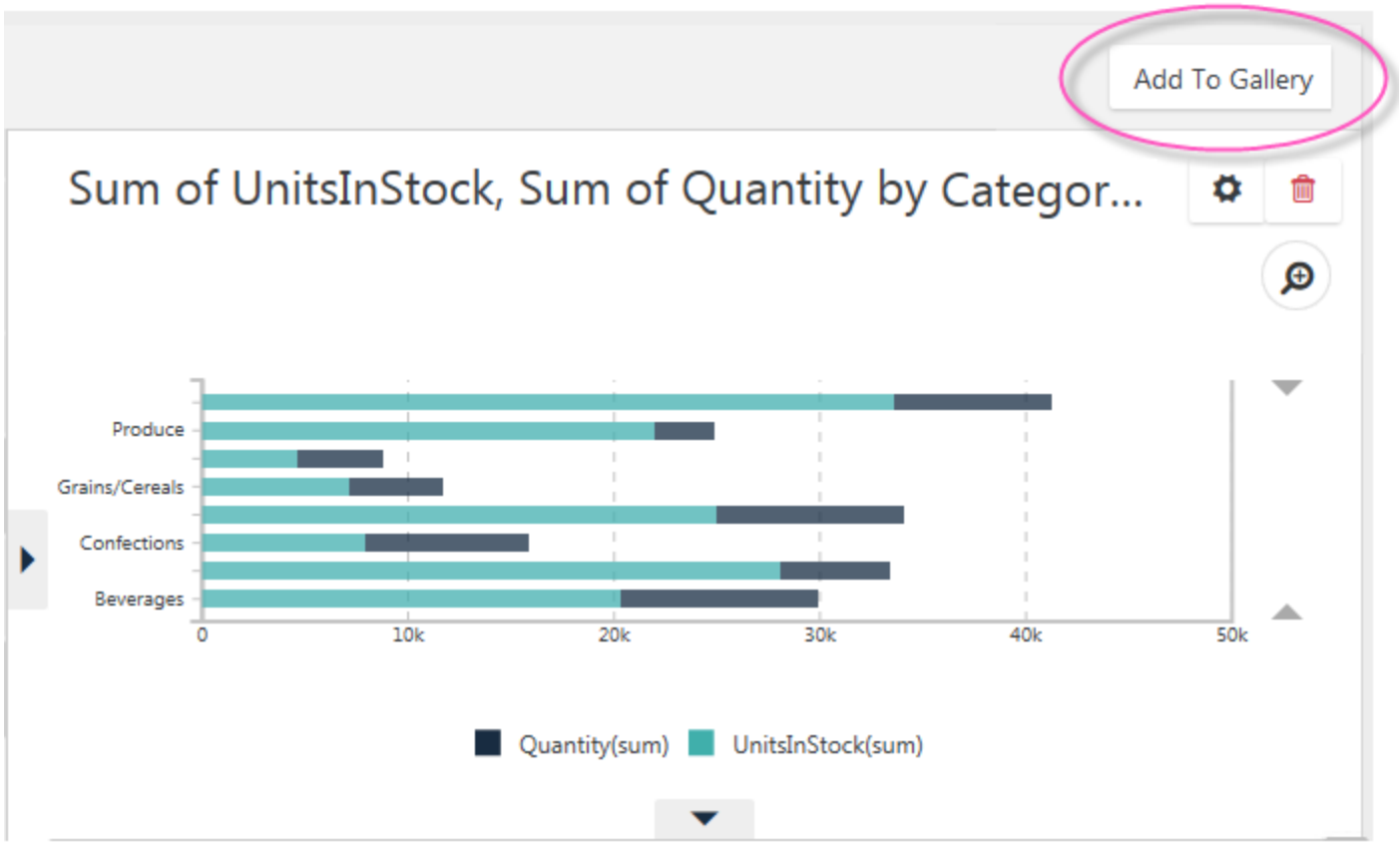


Configure Bookmark attributes to enable Save button in Thinkspace charts

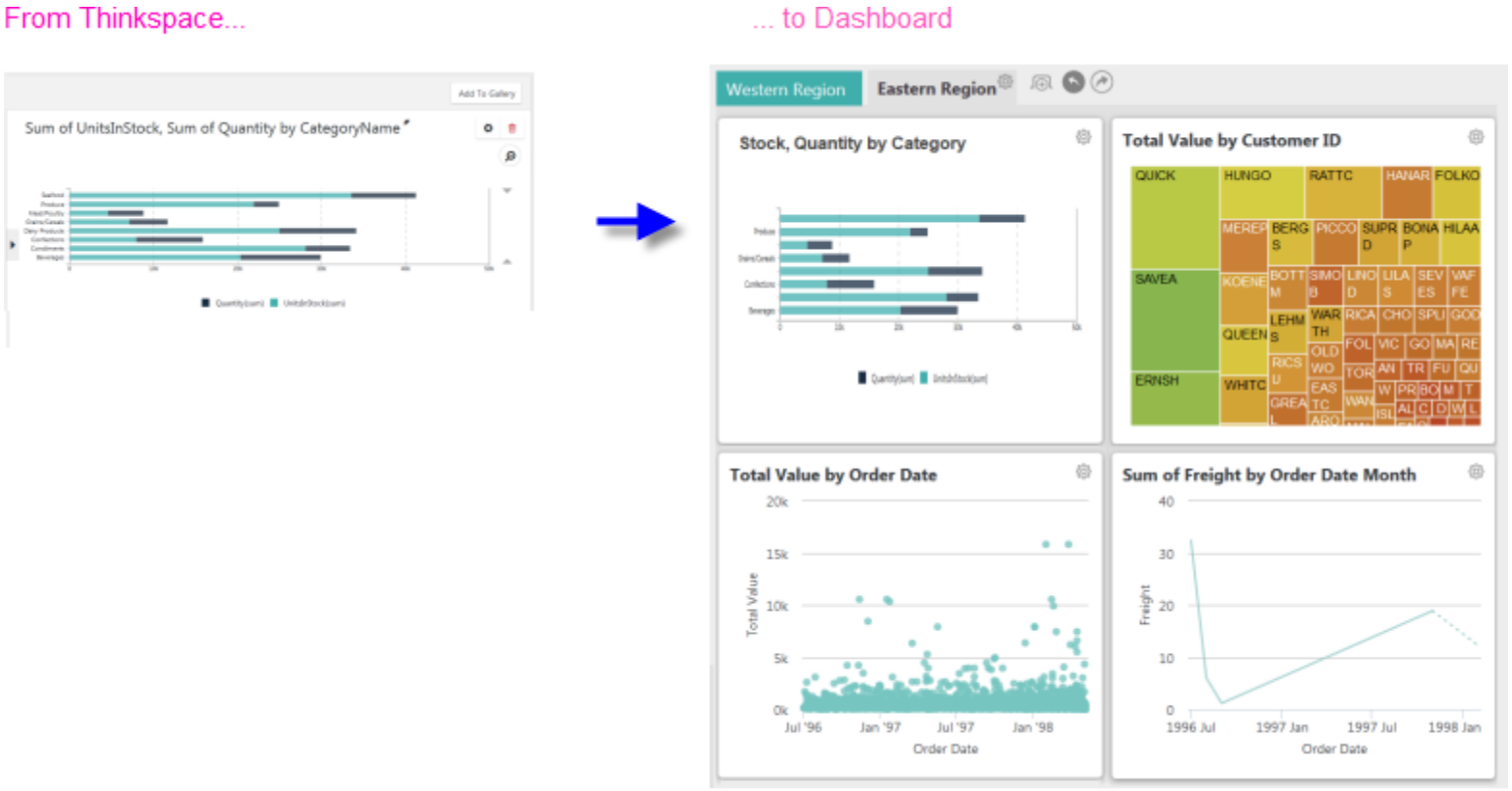
To enable the button, in the _Settings definition's **General** element, shown above, set the **Bookmark Collection Default** and **Bookmark Folder Location** attributes. Bookmark collections are often user-specific and the @Function.UserName~ token can be used here to achieve this. The location attribute requires a fully-qualified path to a specific folder within the application folder.

Configuring Add-to-Dashboard or -Gallery

The Thinkspace allows users to generate a visualization, then add it as a new panel in an existing Dashboard or as a resource in a visualization gallery in another report. The "Logi Info Dashboard " on page 351 and *Report Author* elements can make use of visualization galleries.



When properly configured, the Thinkspace visualization area will display a button like the one shown above. Its tooltip caption can be configured; by default it's "Add to Gallery".

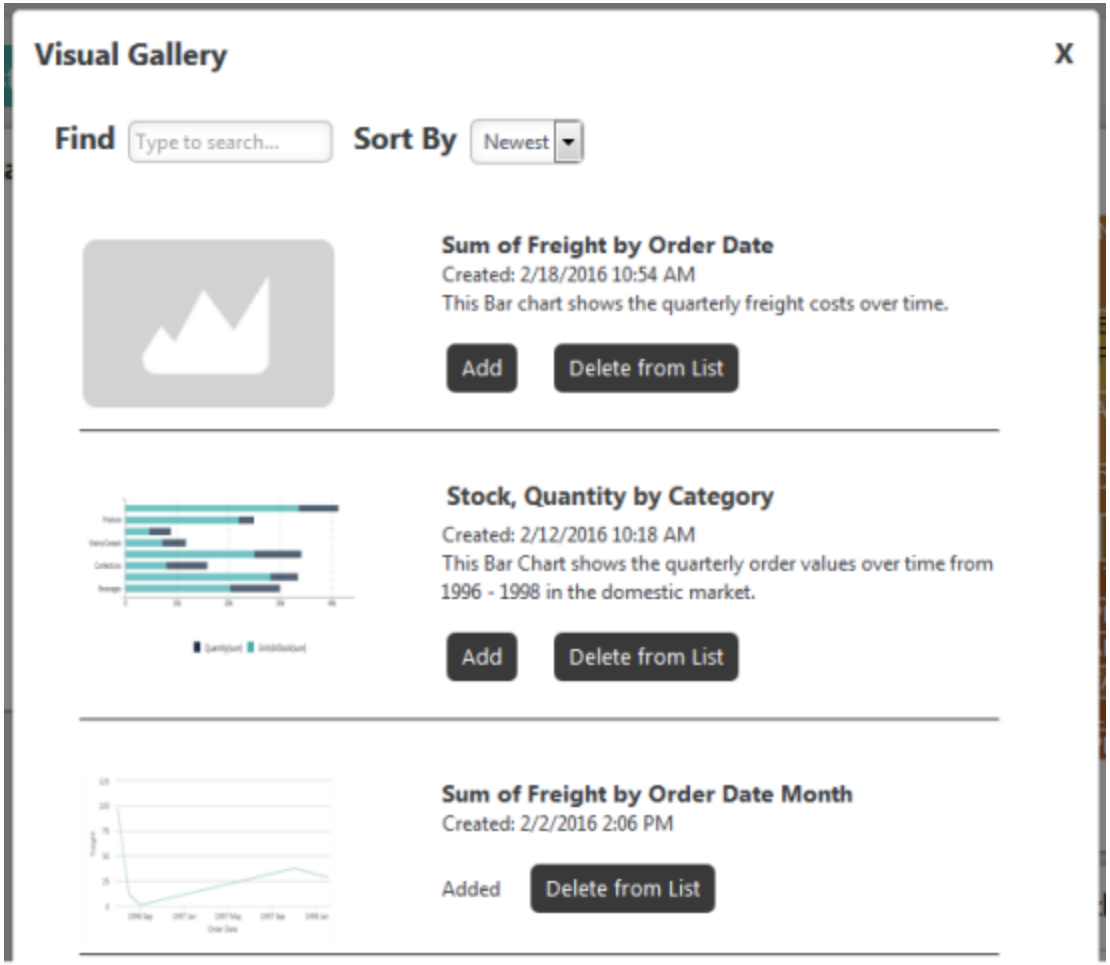


When configured for use with a Dashboard "Save File" and the button is clicked, the visualization is added immediately as a *new panel* in the Dashboard, as shown above.

The Thinkspace writes the underlying XML for its visualization into the Dashboard's configuration file. In the Dashboard, the chart is inserted at the top of Column 1; if Dashboard tabs are being used, the insertion occurs in the currently active tab.

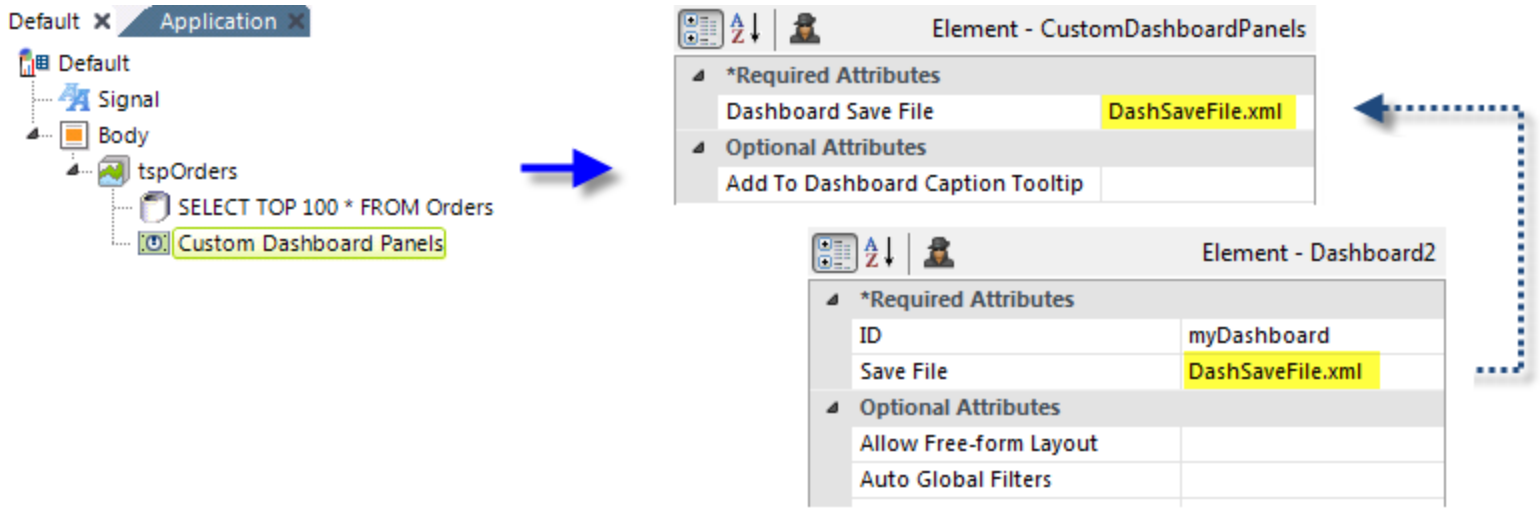
Or, if so configured, the Thinkspace can write the underlying XML for its visualization into a Dashboard or Report Author's "Gallery File". The visualization is then available for use but no immediate insertion occurs.

Just before the visualization is saved, the user will be prompted for the **Panel Title** (with the chart title from the Thinkspace provided as a suggestion) and an optional description for display.

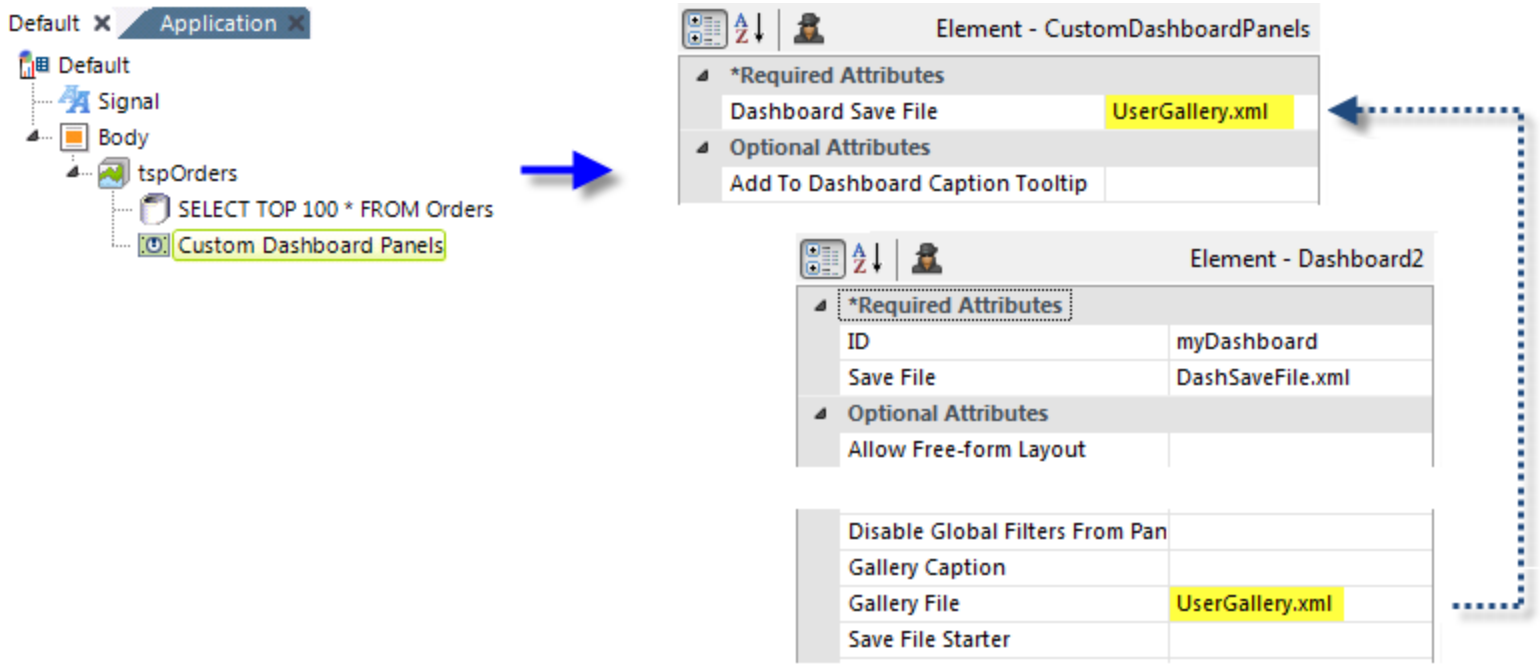


The new visualization thereafter appears in the Dashboard Configuration Page or Visual Gallery, as shown above, just like any other resource, complete with a thumbnail image. The visualization can be removed from the visible Dashboard panels and from the configuration page or gallery entirely, using the usual controls.

The user can insert multiple visualizations into a Dashboard or Gallery using this technique.



In the report definition, the developer makes this functionality available by adding a **Custom Dashboard Panels** element, shown above, beneath a Thinkspace element. For use with a Dashboard Save File, its **Dashboard Save File** attribute value should be set to match the target Dashboard's **Save File** attribute value, as shown above.



For use with a Dashboard or Report Author's visualization gallery, the attributes should be configured as shown above.

The Save File and Gallery File attribute example values shown above have been shortened for visual clarity. However, a *fully-qualified path and file name*, with .xml extension, is required, so a realistic attribute value would be something like:

```
@Function.AppPhysicalPath~\DashFiles\DashbdSave.xml
@Function.AppPhysicalPath~\Galleries\@Function.UserName~_UserGallery.xml
```

Ensure that the account that runs the Logi application has **Write** permissions to the "DashFiles" or "Galleries" folder. For experimentation purposes, you can use the rdDownload folder, for which appropriate permissions already exist (but its contents are periodically deleted, so don't use it for anything but experimentation).

If Logi Security is being used, Dashboard Save Files and Gallery Files can be personalized using the `@Function.UserName~`token.

The Custom Dashboard Panel element also has an attribute, **Add Caption**, which sets the caption for the button used to save the visualization. If left blank, the default caption is "Add to Gallery".

For more information about Dashboards, see "Logi Info Dashboard " on page 351.

Embedding Visualizations

Visualizations can be embedded into Logi applications and into non-Logi app web pages. As we've seen in previous examples, Logi Platform Services can be used directly in a Logi application. The **Thinkspace** element is available as a child of **Body**, **Division**, and **Column Cell** elements but cannot be used in Data Table Columns or Dashboards. Here are two other embedding scenarios:

Embedding in Logi Apps Using an iFrame

You can embed a Logi definition containing a Thinkspace element in another definition in the *same* Logi application, or in a *different* Logi application, using the **SubReport** (Include Frame) element, without any special requirements.

Embedding in HTML Pages using the Embedded Reports API

Logi Platform Services can be used with visualizations embedded into non-Logi app web pages using the Logi **Embedded Reports API**. Here's a very simple example of the HTML code required:

```
<html>
<head><title>DM Embed Test</title></head>
<body>
 <p>This is my embedded Discovery Module test</p>
 <div id="div1" data-applicationUrl="http://localhost/myLogiApp" data-report-
t="Default"style="width:900px; height:900px;"></div>
 <script src-
c="http://localhost/myLogiApp/rdTemplate/rdEmbedApi/rdEmbed.js"type="text/Javascript"></script>
```

```
</body>
</html>
```

See "Embedded Reports API" on page 466 for information about using it.

Embedding SuperWidgets

SuperWidgets can be embedded in Logi and other applications using JavaScript and the *Logi Platform Services API*.

Chart Tutorial Wizard

This topic is for new Logi developers. It guides you through the process of building a Logi report that presents data from your SQL database in a chart. It uses Studio's wizards to create the report.

This topic contains the following sections:

Requirements

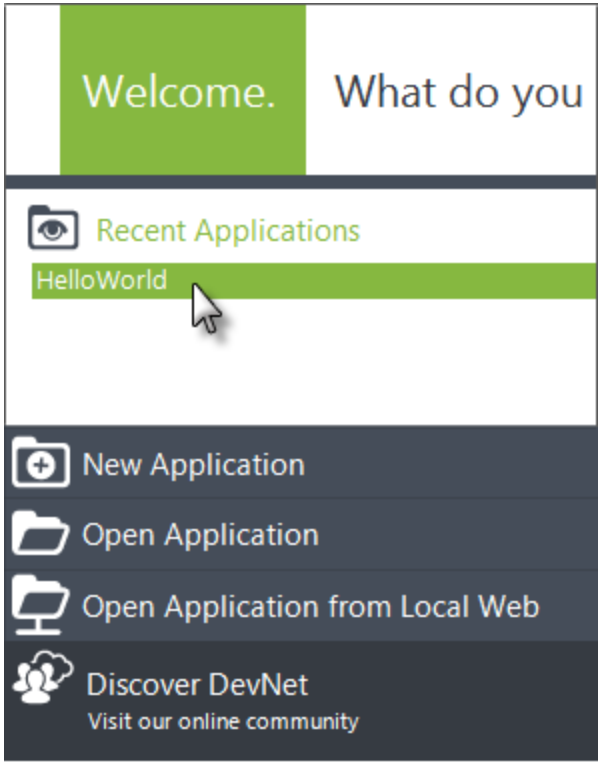
This tutorial assumes that you have:

- Microsoft IIS web server and .NET Framework 4.x installed, - OR -
- A supported Java web server and Oracle JDK or OpenJDK 8, 11, 12, 13, or 14 installed;
   Oracle has changed its Java usage policies - see *Java Usage Policy*for important information.
- An un-expired trial license or a regular or OEM license file installed in `C:\Program Files\LogiXML IES Dev\LogiStudio;`
- A working connection to a SQL database server;
- Credentials for accessing a table on the database server.

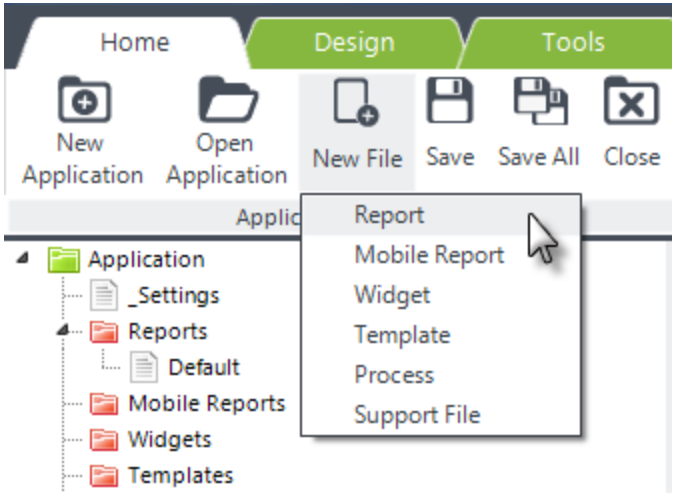
Tutorial

This tutorial builds on the *Hello World!* and *Data Table - Wizard* tutorials. Please complete those tutorials before proceeding with this one. We also offer a more comprehensive tutorial, "Data Table Tutorial - Manual" on page 194, which does not use Studio Wizards and provides more "behind-the-scenes" insight into report creation.

1. Begin by launching **Logi Studio**, using its desktop icon or Start → All Programs → Logi Info → Studio.



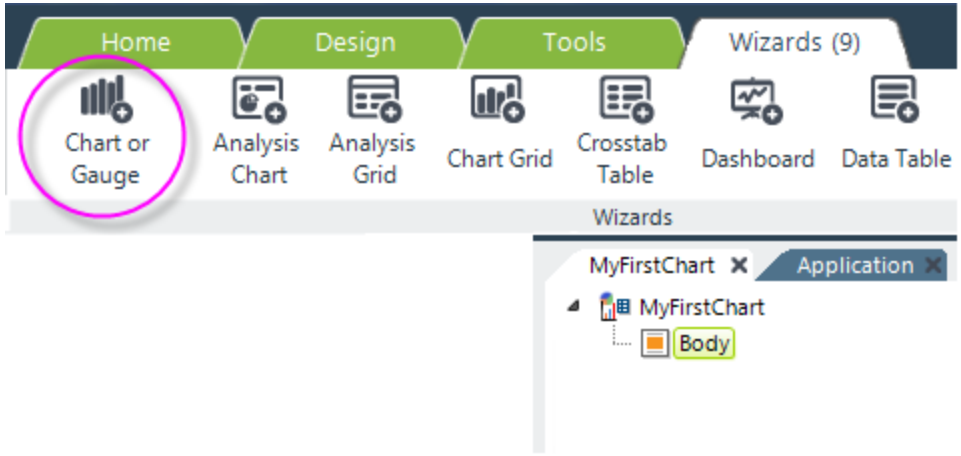
2. In Logi Studio, close the Getting Started dialog box and re-open the **HelloWorld** application from the Welcome Panel, as shown above.



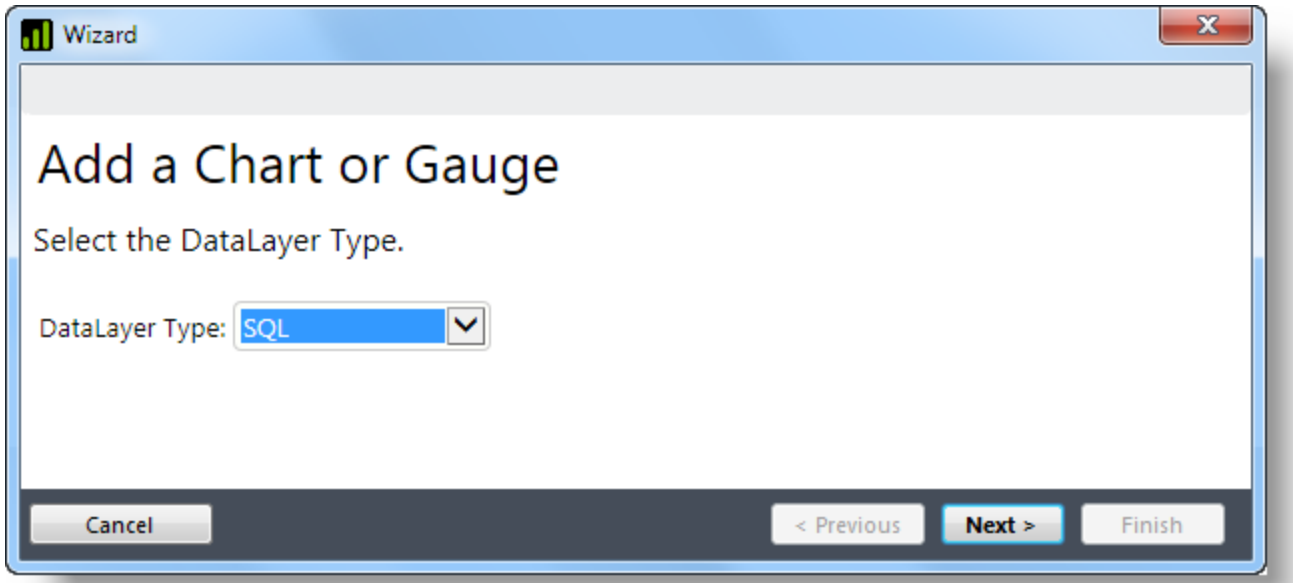
3. Click the **New File** menu item, and select *Report* from its drop-down list of options, as shown above.

Studio will add a new report definition, named "newReport" in the Application Panel, and put it into rename mode. Just start typing a new name - such as "**MyFirstChart**" - and press Enter to rename it.

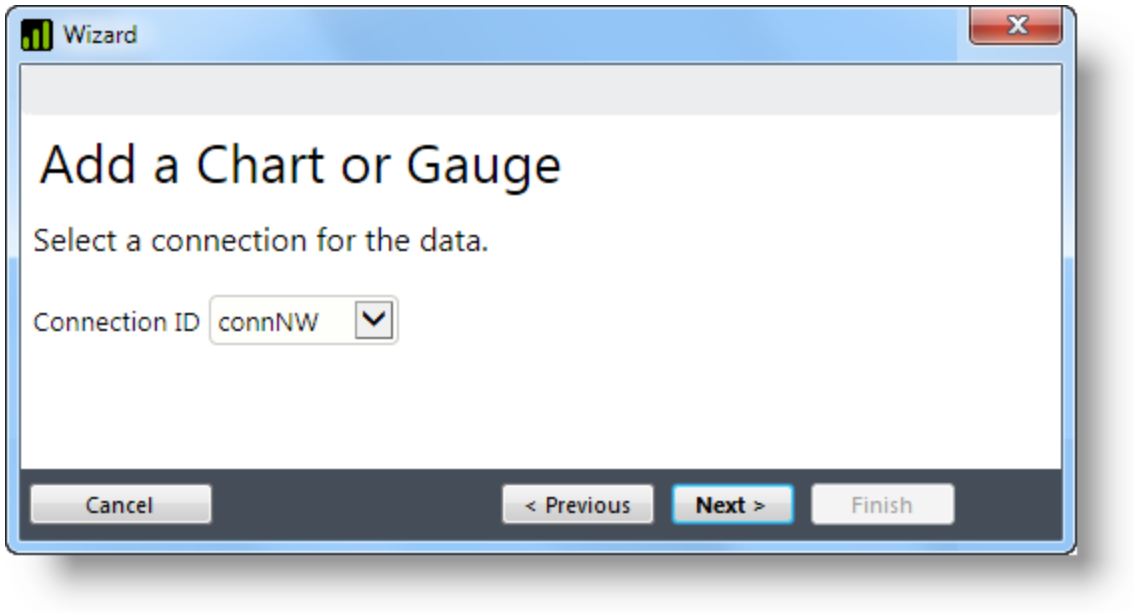
If something happens, like you click elsewhere, and it exits rename mode, just select the newReport definition and press F2, or right-click and select *Rename*, to rename it.



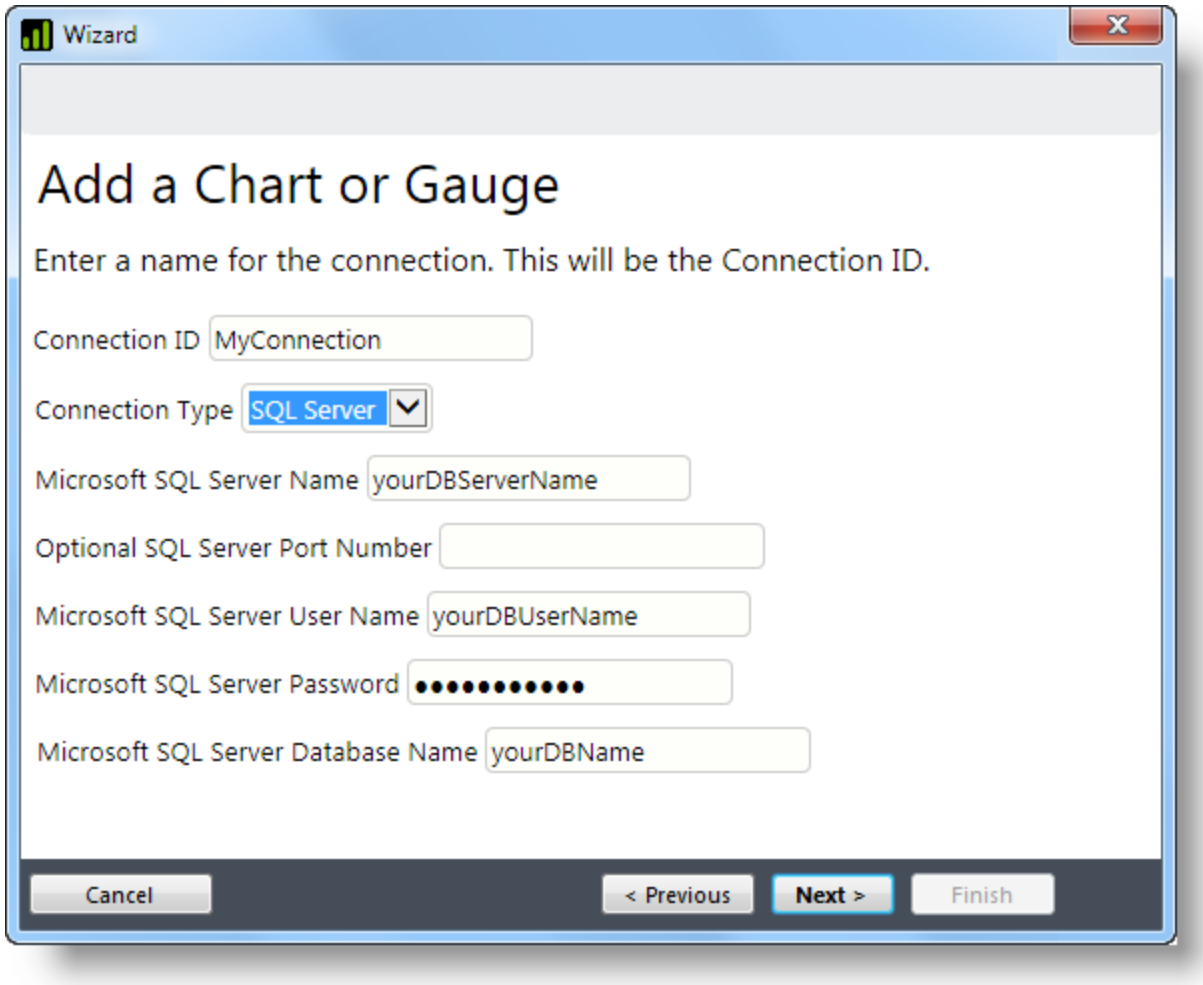
4. You probably noticed that the new definition was opened in the Workspace editor. Select its **Body** element and the **Wizards** tab will appear in the main menu. In the Wizards tab, click the **Chart or Gauge** menu item to start the Add a Chart or Gauge wizard.



5. The wizard will first ask you to specify the datalayer type. Datalayers retrieve data for use in reports. Use the default *SQL* datalayer type and click **Next**.



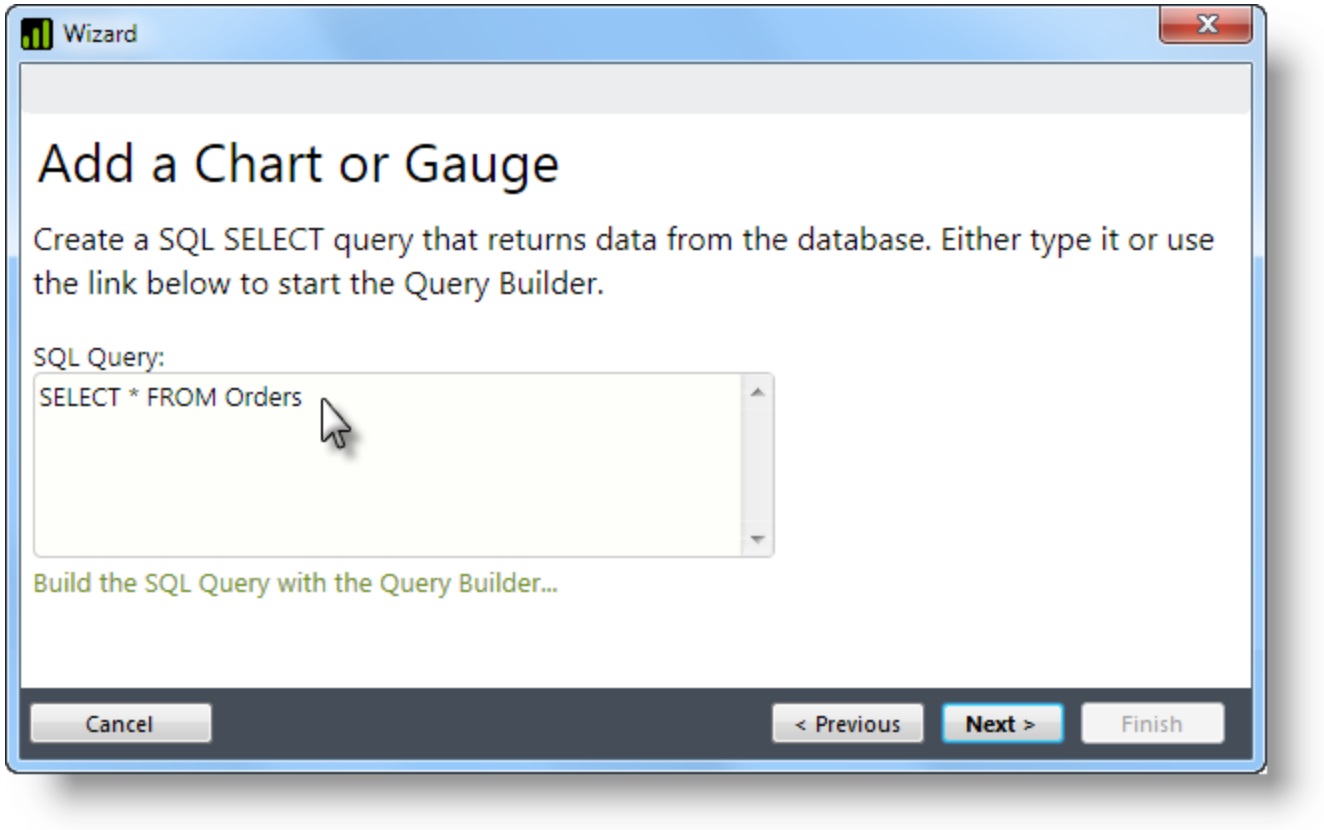
6. Data for this report is coming from a database server, so you need to establish a **connection** to it. Select the connection you created in the previous tutorial (or create a new one) and click **Next**.



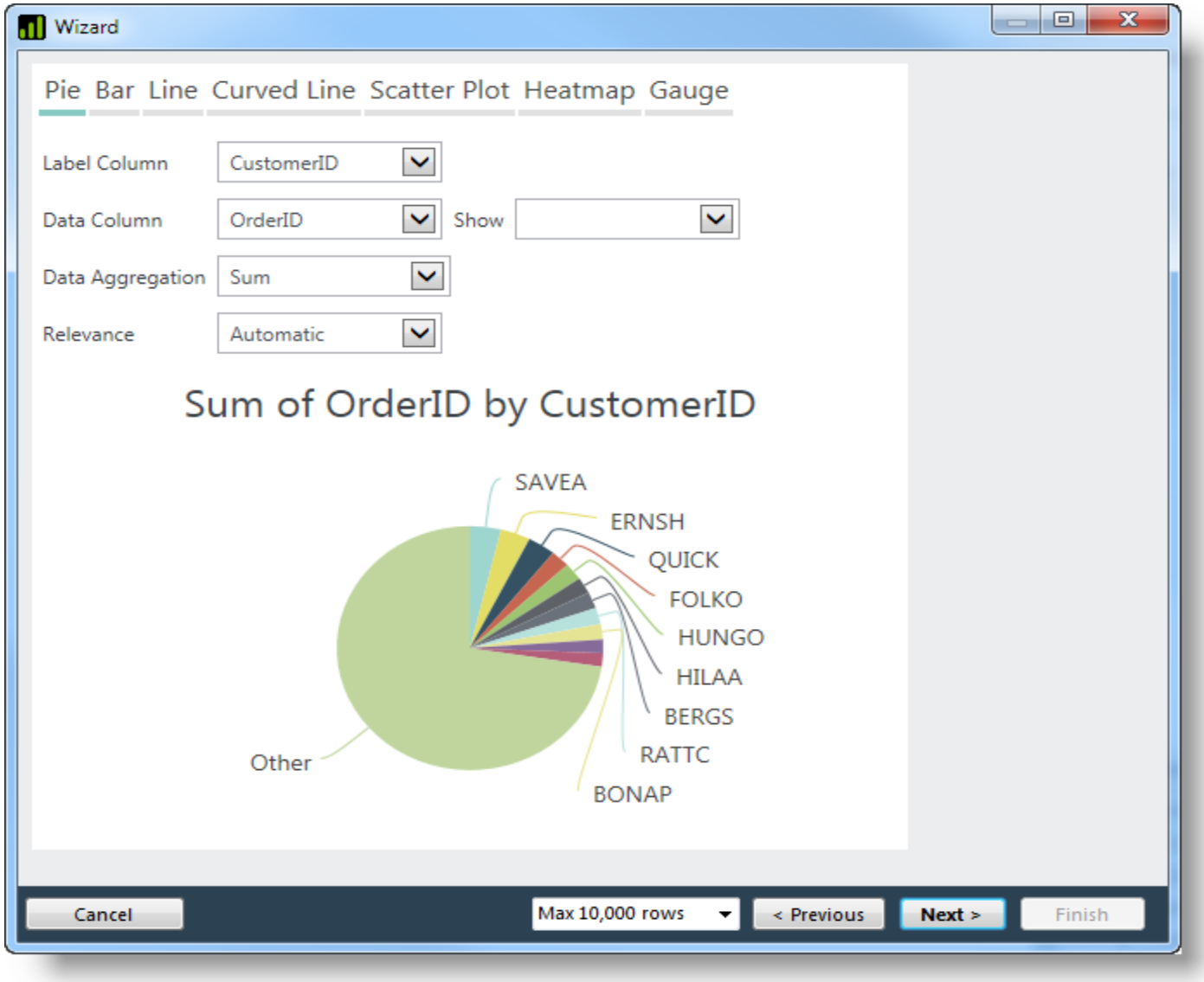
7. If you chose to create a new connection, the wizard will prompt you, in separate dialog boxes, for all of the information shown above. To save space, we've grouped them into *one* wizard image. You'll provide information as necessary and click **Next** to proceed each time.

   The prompts shown above are for Microsoft SQL Server, but they'll vary depending on the Connection Type you select.

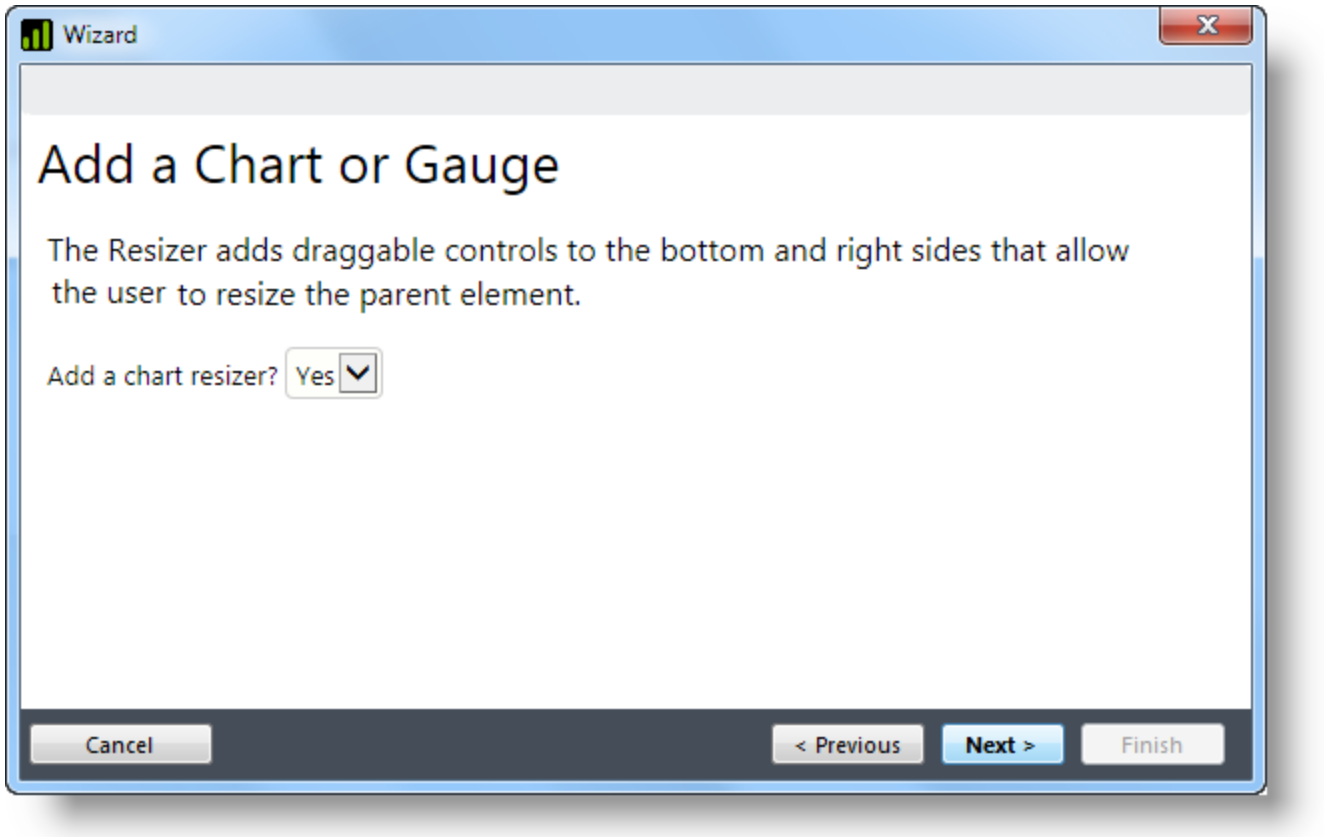
After you provide the requested information, the wizard will test the connection and present its status. If there's a problem, you can click **Previous** to go back and review or change your entries. Click **Next** again as needed to re-test the connection.



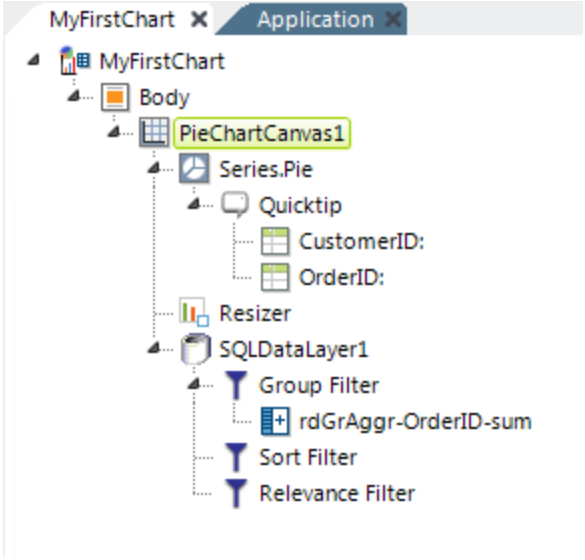
8. Once the connection is successful, you're ready to enter a SQL query. Skip using the SQL Query Builder this time and just type-in a simple query, like the one shown above, that will return some columns from your database. The click **Next** to continue.


Wizard
Pie Bar Line Curved Line Scatter Plot Heatmap Gauge
Label Column
CustomerID
Data Column
OrderID
Show
Data Aggregation
Sum
Relevance
Automatic
Sum of OrderID by CustomerID
SAVEA
ERNSH
QUICK
FOLKO
HUNGO
HILAA
BERGS
RATTC
BONAP
Other
Cancel
Max 10,000 rows
< Previous
Next >
Finish

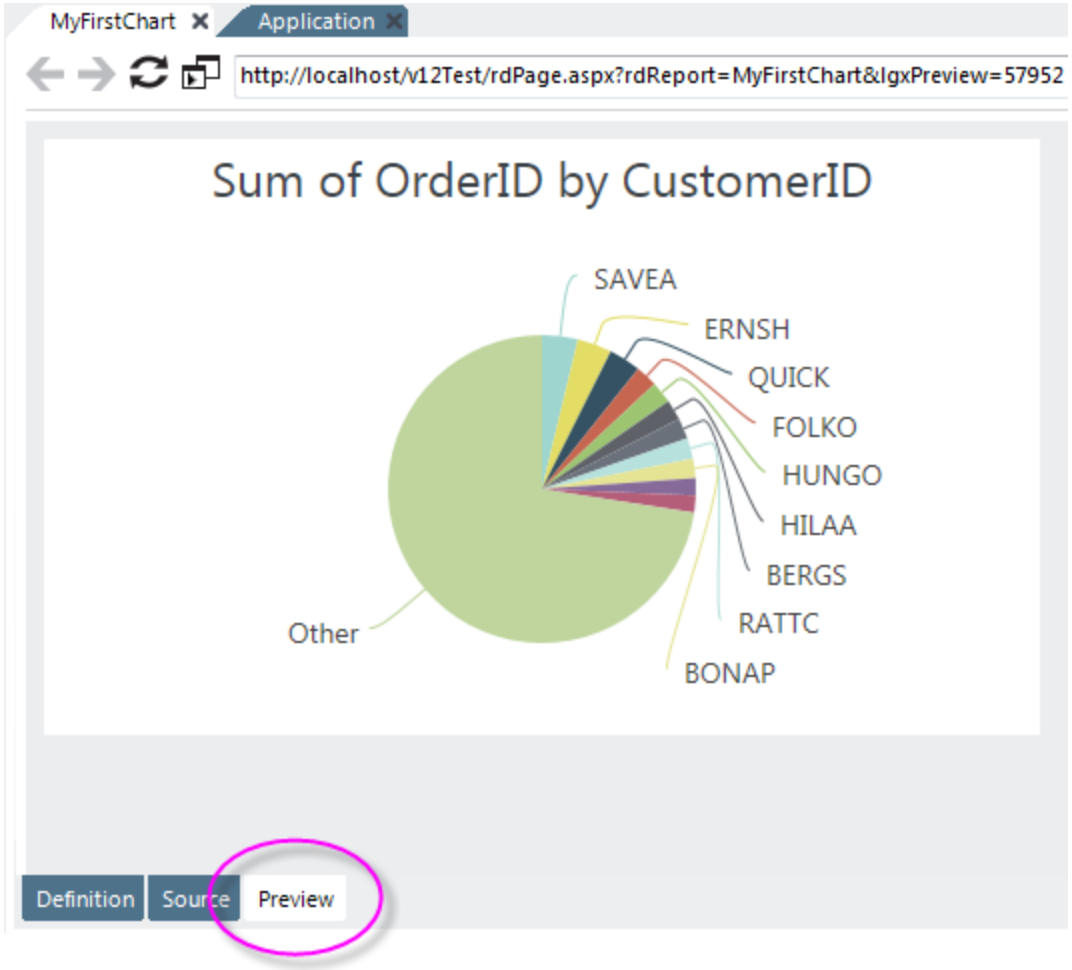
9. As shown above, the next dialog box displays a chart preview. Different chart type options are listed across the top of the box and will vary depending on your data. Click them to try out some different charts! You can also change the data columns that are used in the chart and other supporting configuration options. Once you have the preview chart the way you want it, click **Next** to continue.



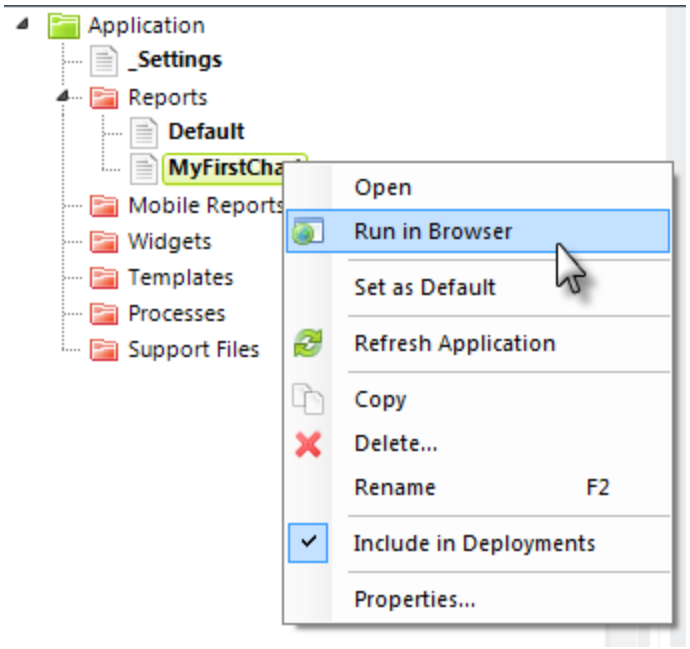
10. The wizard will prompt you to add Resizer controls to the chart. These allow the user to resize the chart at runtime by dragging "resizer handles". Click **Next** to continue.



11. The wizard will insert all of the properly configured elements necessary for the chart you created into your definition, as shown above, and then exit.



12. Click the **Preview** tab at the bottom of the Workspace editor to save your work and preview the report in Studio. You should see something like the example shown above.



If you'd like to view it in your default browser, select and right-click the report definition, as shown above, and select *Run in Browser* from its context menu.

Well done! We recommend that you take some time to examine the elements the wizard added to your report and _Settings definitions. Congratulations on completing this tutorial.

Data Table Tutorial - Manual

This topic is for new Logi developers. It guides you through the process of building a Logi report that presents data from your SQL database in a table. It uses mostly manual methods, rather than Studio's wizards, to create the report.

This tutorial builds on the "Hello World! Tutorial" on page 41. Please complete that tutorial before proceeding with this one.

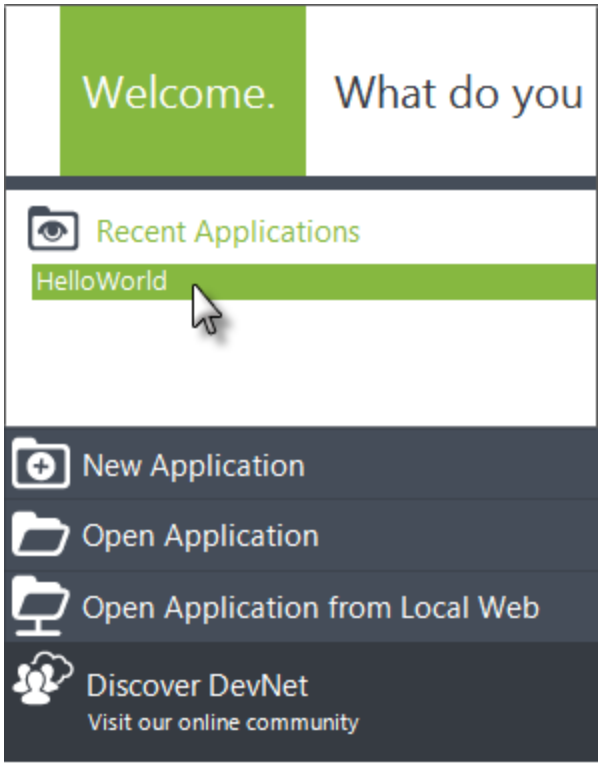
The following topics provide a detailed, "behind the scenes" look at the process of building a Logi report:

- Create a New Report Definition
- Add a Database Server Connection
- Add a Data Table
- Preview and Run Your Report
- Change the Theme

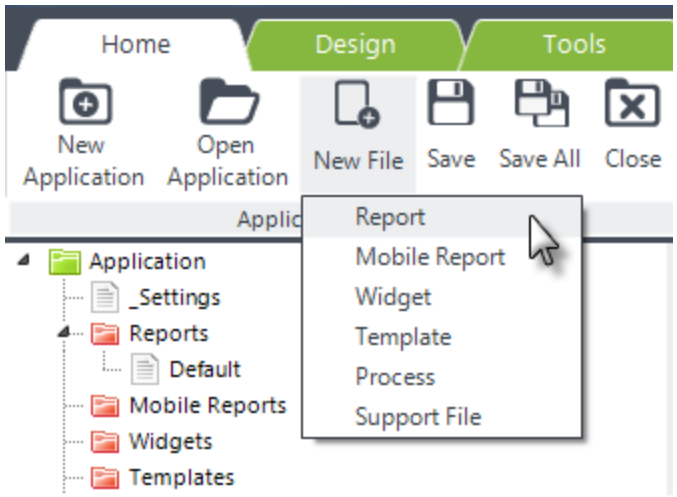
Create a New Report Definition

This topic demonstrates how to create a new report definition in Logi Studio.

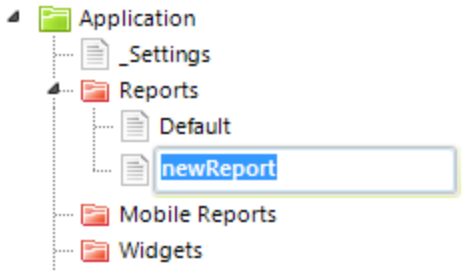
1. Begin by launching **Logi Studio**, using its desktop icon or Start → All Programs → Logi Info → Studio.

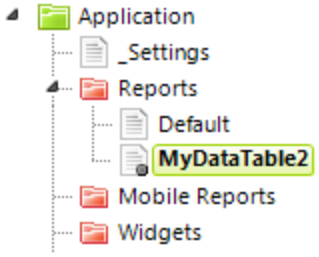


2. In Logi Studio, close the Getting Started dialog box and re-open the **HelloWorld** application from the Welcome Panel, as shown above.



3. Click the **New File** menu item, and select *Report* from its drop-down list of options, as shown above.





Studio will add a new report definition, named "newReport" in the Application Panel, and put it into rename mode. Just start typing a new name - such as "**MyDataTable2**" - and press Enter to rename it.

If something happens, like you click elsewhere, and it exits rename mode, just select the newReport definition and press F2, or right-click and select *Rename*, to rename it.

You probably noticed that the new definition was opened in the Workspace editor. It contains five key elements by default:

- The **Root** element of the report. Its ID is the report definition name, in this example, "MyDataTable2", and
- A **Body** element, a container for displaying the main report content (tables, charts, controls, etc.)

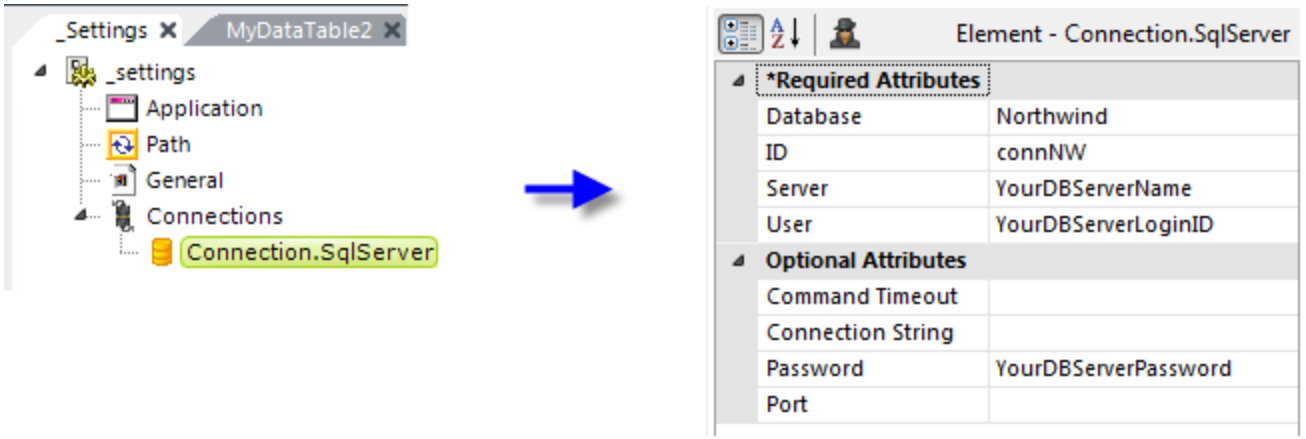
Any report that displays data in a table also requires five more types of elements, which we'll work with next:

- A **Connection** to the database or other data source.
- A **Data Table** element, which defines the data presentation (rows and columns).
- A **DataLayer** element, to retrieve and cache the data.
- One or more **Data Table Column** elements, to organize the data.
- A **Label** element in each column to format and display the actual data.

Your next step is to create a connection to your database server.

Add a Database Server Connection

The **Connection** element comes in a variety of flavors, depending on the nature of the connection to be made. The best method of connecting is to use a *vendor-specific* connection element, such as Connection.Oracle, or Connection.MySQL, when you can. There are also generic connection elements. The following example uses **Connection.SQLServer** to connect to Microsoft SQL Server. If you're using a different database server, substitute the appropriate elements and attributes in the following discussion.



Here's how to add a Connection element:

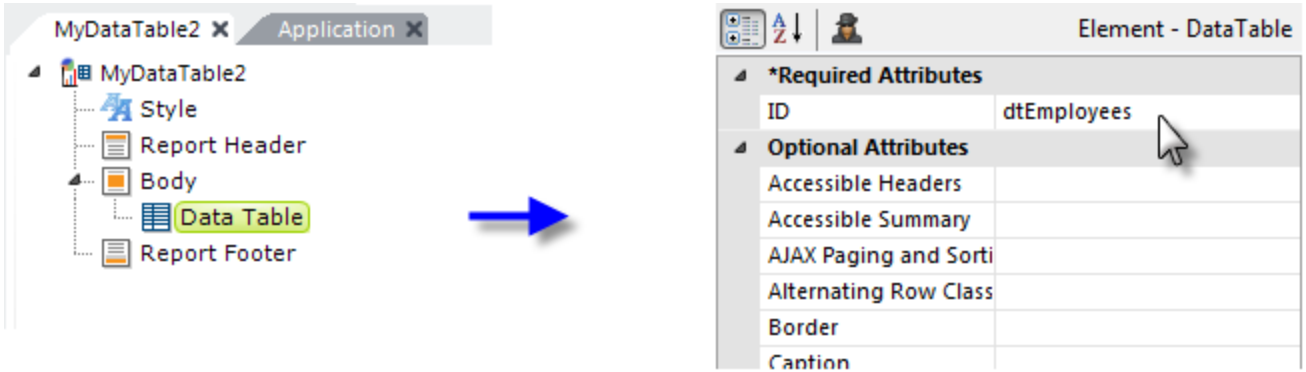
1. In the Application Panel, double-click the **_Settings** definition to open it in the Workspace editor. You can see it opens in a different tab than the new definition you added earlier.
2. Select the **Connections** element. In the Element Toolbox panel, double-click the connection element appropriate for your database server to add it. This is the basic technique for adding elements: select a *parent* element and double-click a *child* element which is then added beneath it.

3. When the newly-added connection element is selected, its properties or "attributes" will appear in the Attributes Panel. Find its **ID** attribute and set it to a unique value. The example uses *connNW* because we're going to connect to the Northwind database.
4. In the Attributes Panel, provide the other attribute values shown above, using values appropriate for your database and server.
5. Save your _Settings definition by clicking the **Save** item in the main menu
6. Close the _Settings definition by clicking the "X" icon in its editor tab.

The connection is now configured and ready for use.

Add a Data Table

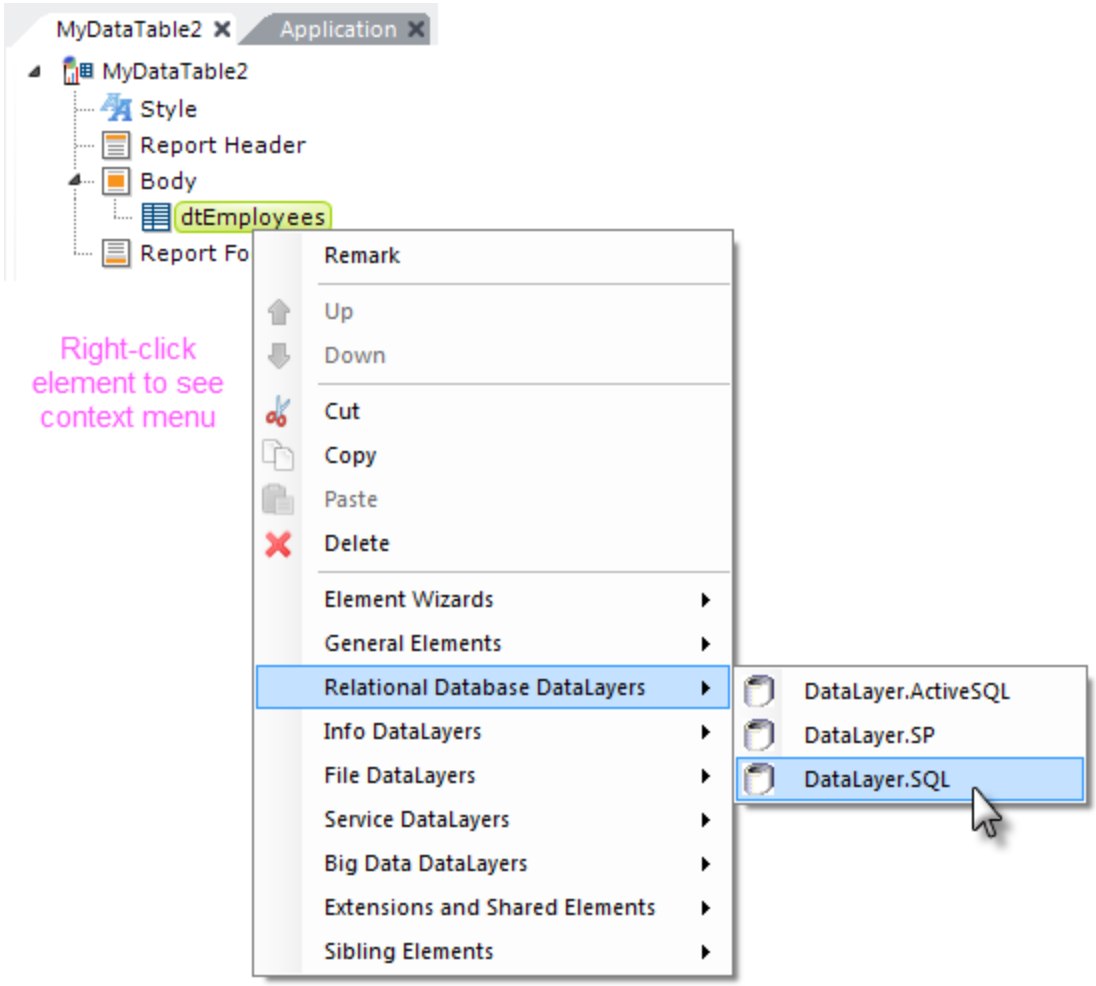
The **Data Table** element is used to display data in a tabular format of rows and columns. The following examples use a table, *Employees*, in the Northwind database. If you're using a different database and table, adjust the names and selections as appropriate.



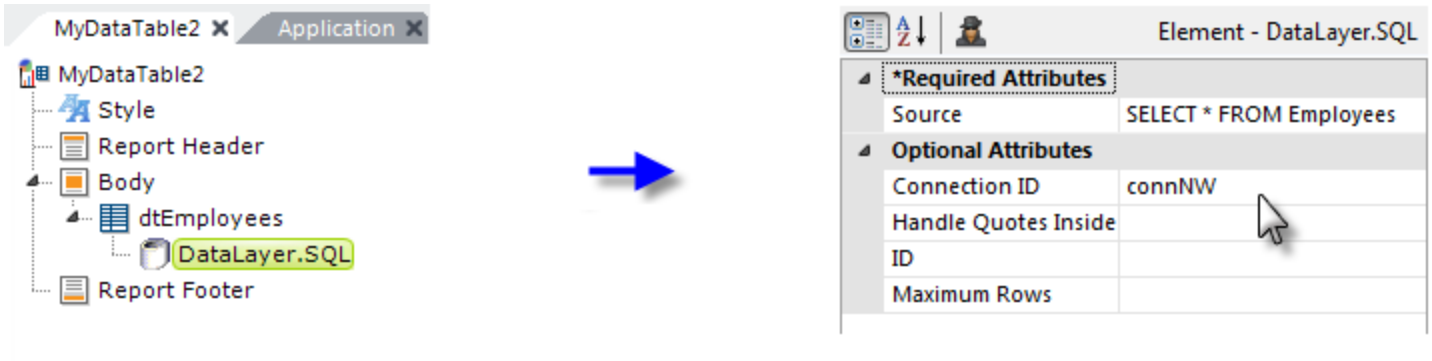
Here's how to add a Data Table element:

1. If it's not already selected, select the **MyDataTable2** definition tab in the Workspace editor.
2. In the definition, select the **Body** element.
3. In the Element Toolbox, expand the Data Tables section (if necessary) and double-click the **Data Table** element to add it to the report as a child of the Body element, as shown above.
4. With the newly added element selected, in the Attributes panel, set its required **ID** attribute.

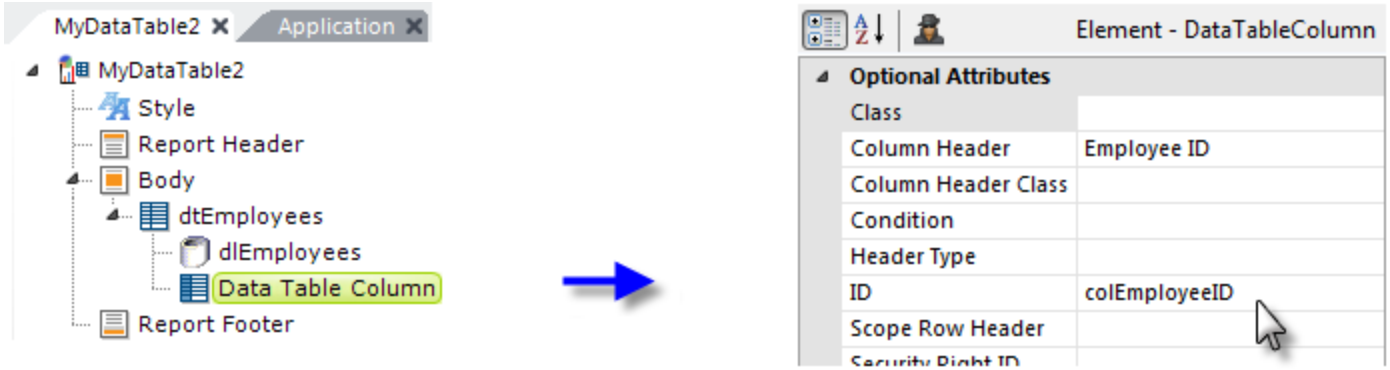
   To add the next element, let's use an alternate method, which you may find you prefer:



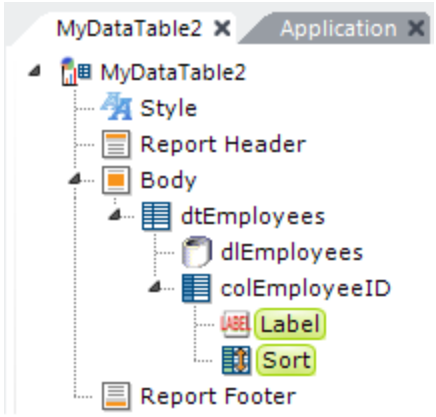
5. Select the newly-added **Data Table** element ("dtEmployees"), and right-click it.
6. In the context menu that appears, select **Relational Database DataLayers**.
7. In the second context menu, click the **DataLayer.SQL** element to add it as a child of the Data Table element.



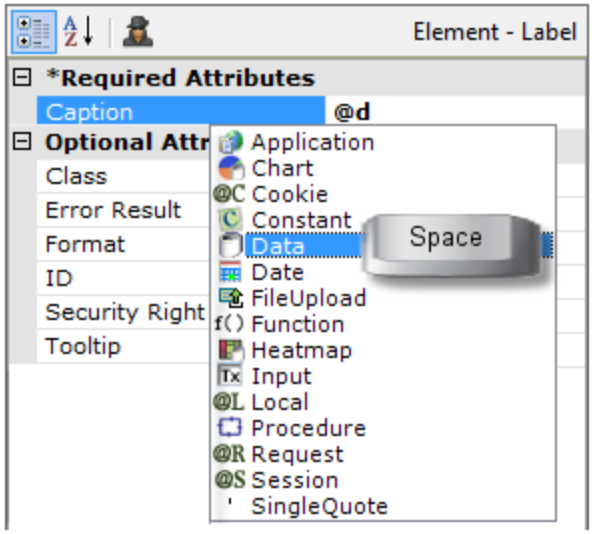
8. Select the newly-added datalayer and set its attributes as shown above. The **Connection ID** can be selected from a pull-down list of choices. The **Source** attribute value is, obviously, the actual SQL query used to get data from the database and, if you're using a different database, you'll need to alter it.

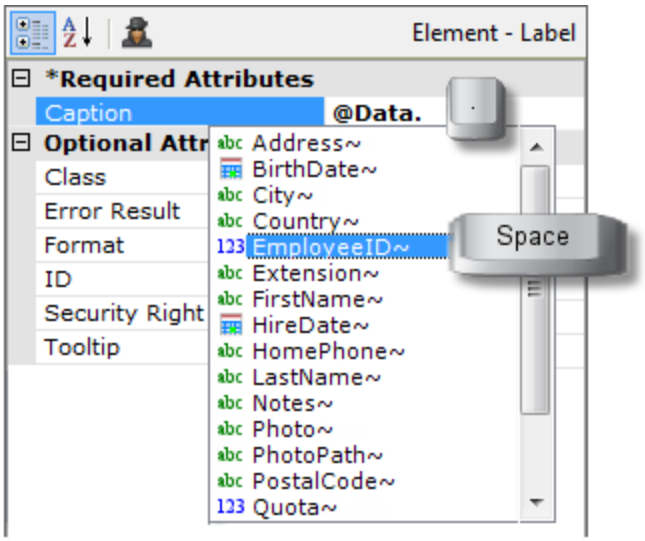


9. Select the **Data Table** element ("dtEmployees") again and below it add a **Data Table Column** element, as shown above. Set its **ID** attribute to *colEmployeeID* and its **Column Header** attribute to *Employee ID*.



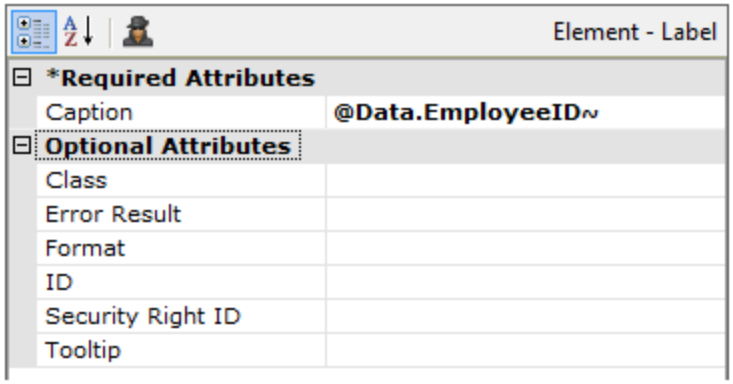
10. Select the newly-added Data Table Column element and, below it, add **Label** and **Sort** child elements, as shown above.





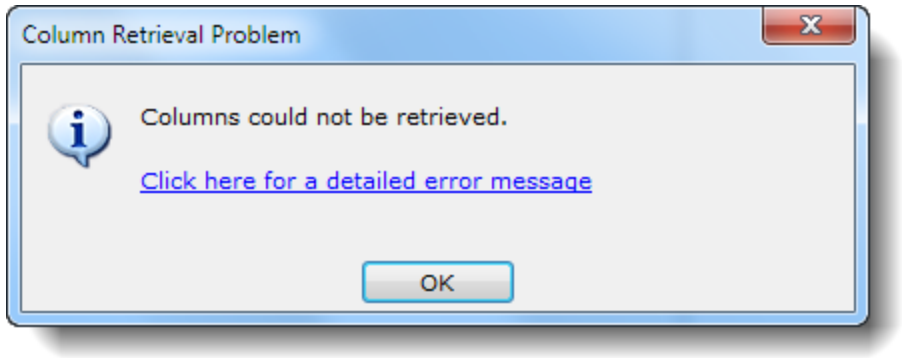
Now we'll use Studio's powerful "Token Completion" feature to fill-in the data value to be displayed for this column.

11. Select the **Label** element and in its **Caption** attribute, type @d . Studio will immediately display a drop-down list of "token" types. A token is a value that is evaluated at runtime and is used in Logi apps to represent data, session variables, cookies, and more. In our case, we want to use a **Data** token. Studio has anticipated our need and selected the Data token type in the drop-down list. Press the Space Bar on your keyboard to select it and Studio will enter the token type in the Caption attribute.
12. Now press the Period key and Studio will attempt to retrieve a list of the data columns returned by our SQL query. Use your arrow keys or the mouse to select *EmployeeID~,* as shown above left, and, again, press the Space Bar.

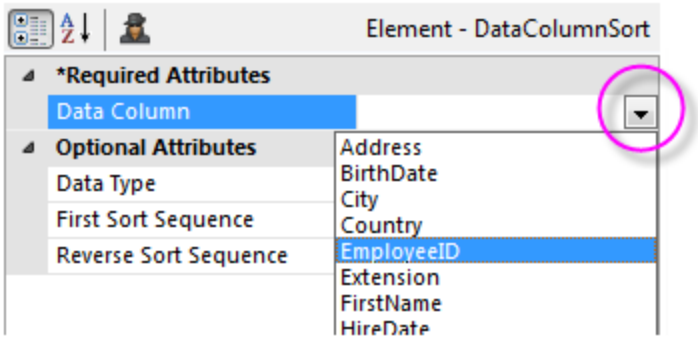


13. Studio enters the column name in the Caption attribute, as shown above, and adds the trailing tilde ("~") character which is the required terminator for tokens. Here's another important fact about tokens: they're *case-sensitive*, so you must take care to spell them, and related identifiers such as column names, correctly.

    Once you do this a few times, you'll find the Token Completion feature to be really valuable: it speeds things up and helps you avoid error-causing typos.



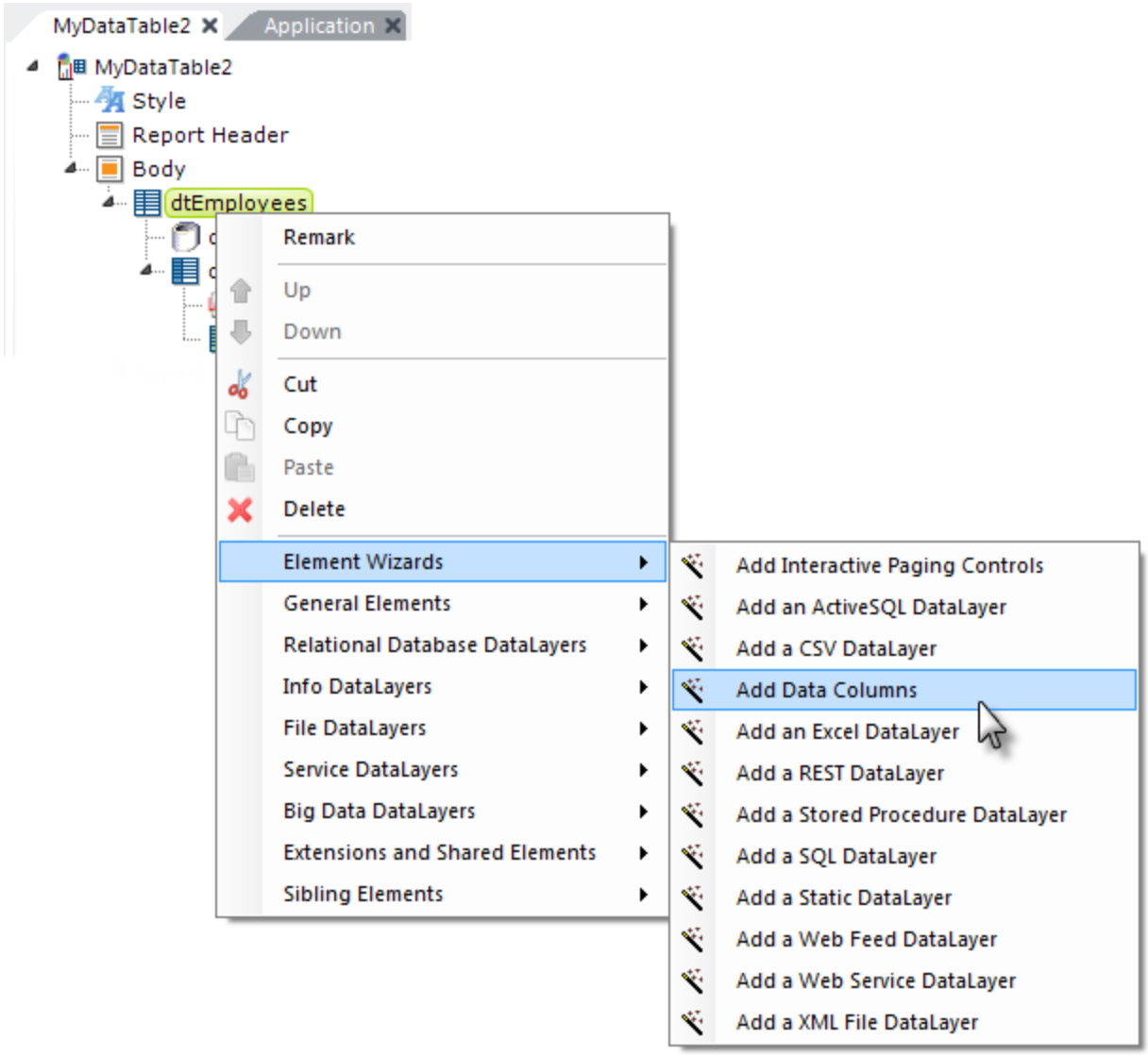
Did Studio return an error when it tried to retrieve the table column names? If so, you most likely have a problem with the configuration of your database connection or your SQL syntax. The link in the error message may provide a useful hint. Fix any problems and retry.



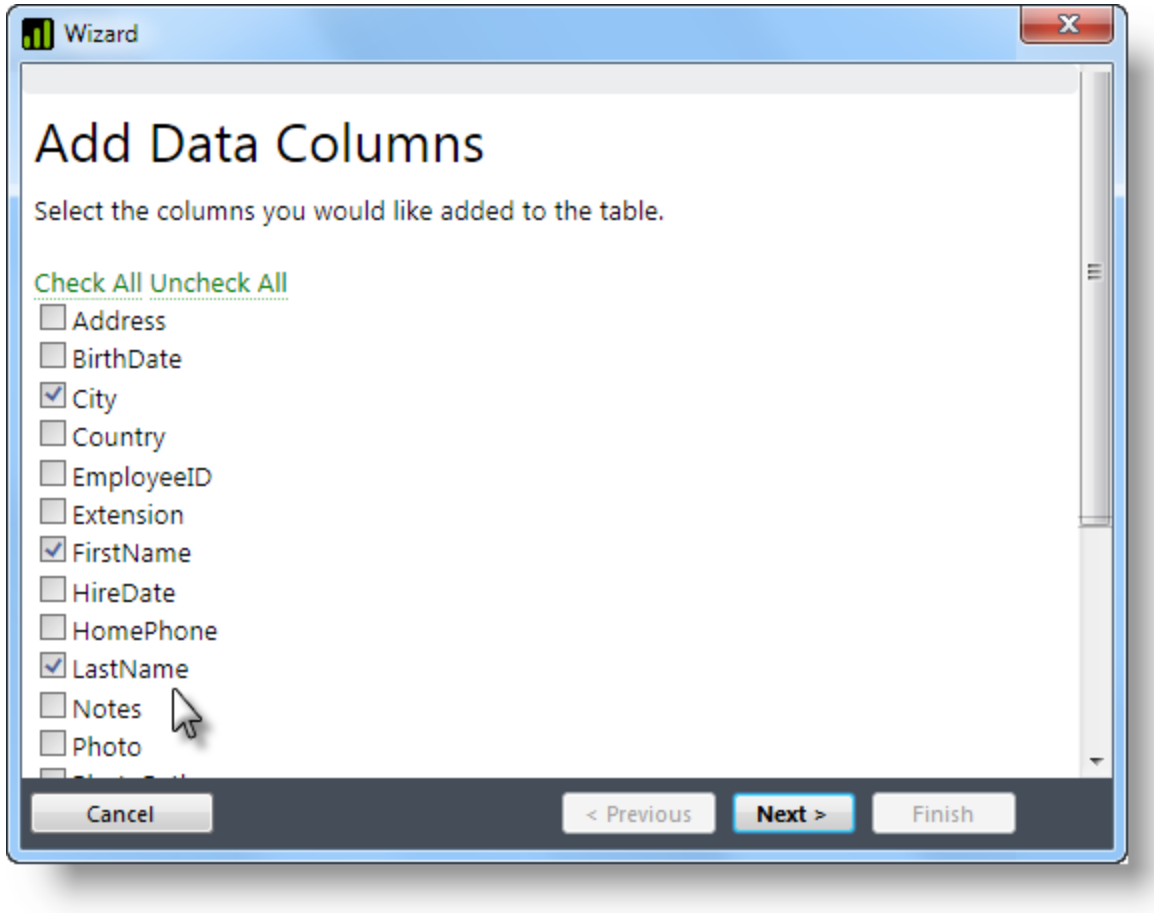
14. The **Sort** element causes the column header text in the table to become a link that will sort the data based on the column clicked. Select it and set its **Data Column** attribute to the column name: *EmployeeID*.

If an attribute *name* includes the world "Column" then the attribute *value* must be the actual column name, not a Data token. The column name is case-sensitive and must be spelled correctly. You can click the button at the end of the attribute value input box to see a drop-down list of the column names.

This grouping of elements: Data Table Column, Label, and (optionally) Sort needs to be repeated for each column that you want to display. Now that you know what's involved, to save a little time, let's use a wizard to provide the other columns:



15. Right-click the Data Table element ("dtEmployees"). Select **Element Wizards** from its context menu.
16. In the second context menu, click **Add Data Columns**, as shown above.



17. When the wizard appears, as shown above, click the *Uncheck All* link, then check a few columns. Then click **Next** and watch in the Workspace editor as the wizard adds elements for you. Click **Finish** when it's done.

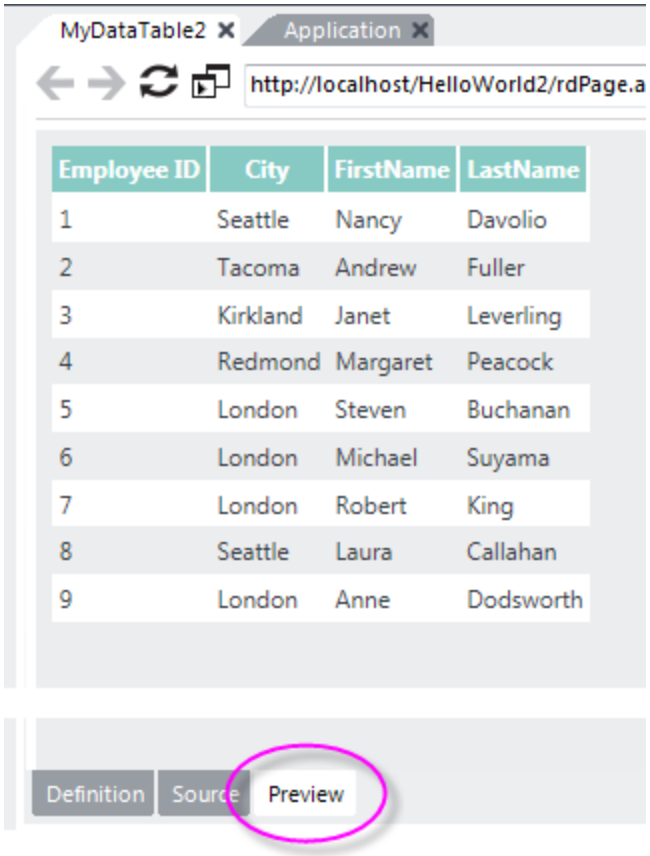
    Click on some of the elements the wizard added and you can see that it has configured their attributes for you, too.

Now that all of the columns have been added, your Data Table is complete and you're ready to see the results.

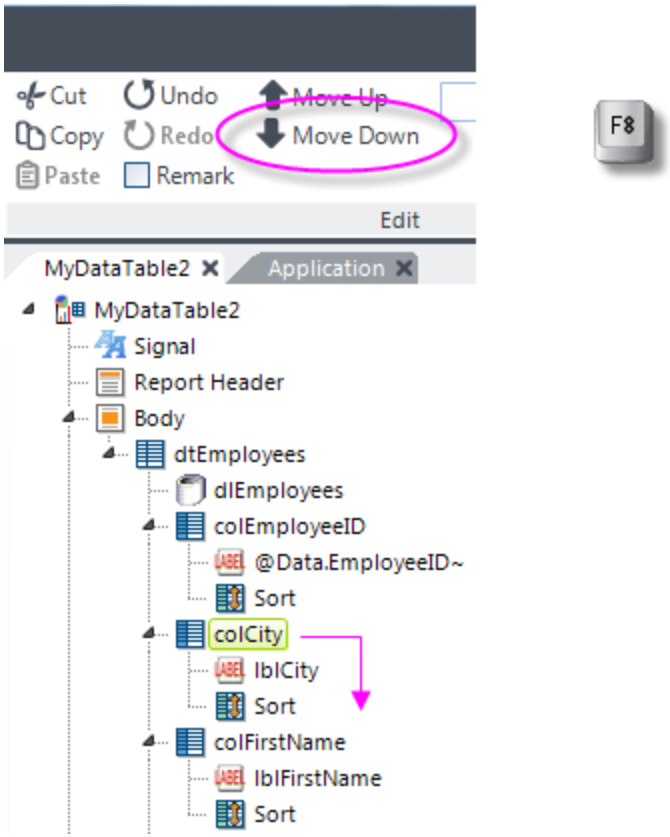
Preview and Run the Report

This topic demonstrates how to preview and run the report in Logi Info.

You can preview the report right inside Studio:

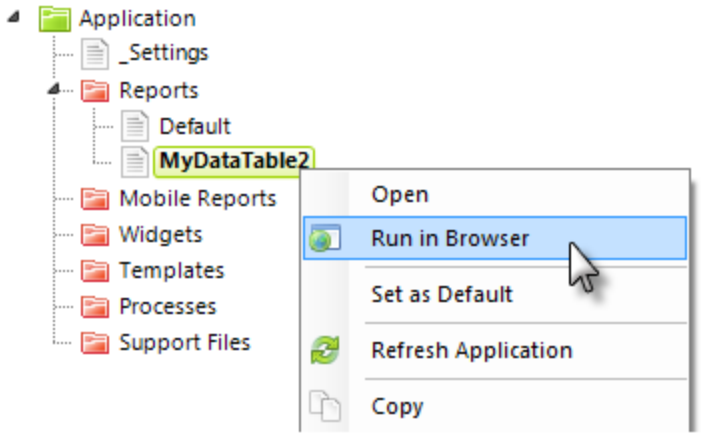


In the Workspace panel, click the **Preview** tab at the bottom. Your report should appear in the panel. Any changes to definitions (but not to support files, such as style sheets) are *automatically saved* when **Preview** is clicked. Click the column headers to see the sorting feature in action. Let's clean up a few things before we move on. Let's rearrange the column order and set the column header titles.



1. Click the **Definition** tab to return to the definition editor.
2. In the MyDataTable2 definition, select the Data Table Column for the City data ("colCity").

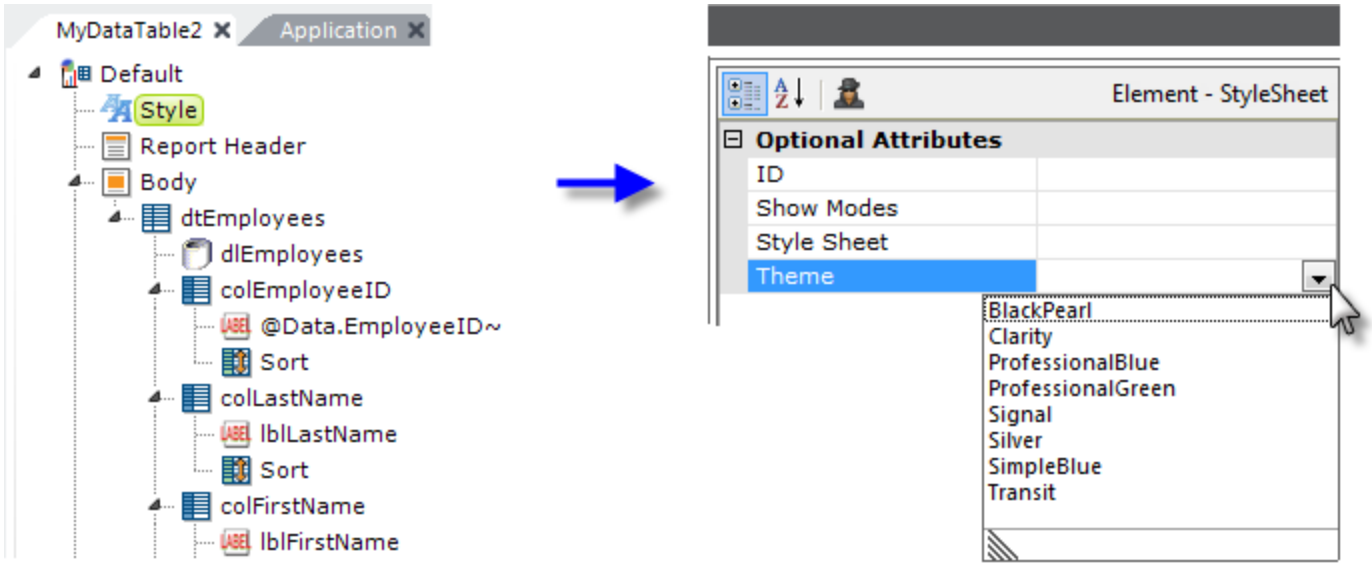
3. Use the **Move Down** item in the main menu, or the **F8** key, to move the selected element down below the other Data Table column elements. Notice that its child elements (Label and Sort) move with it.
4. Repeat this process with the First Name ("colFirstName") Data Table column element so we wind up with a column order of EmployeeID, LastName, FirstName, and City. Preview the results to check the order.
5. Next, select each Data Table Column element in turn, and insert a space into its **Column Header** attribute value, separating the two words found there, so "FirstName" becomes "First Name", for example. Then preview the results again.



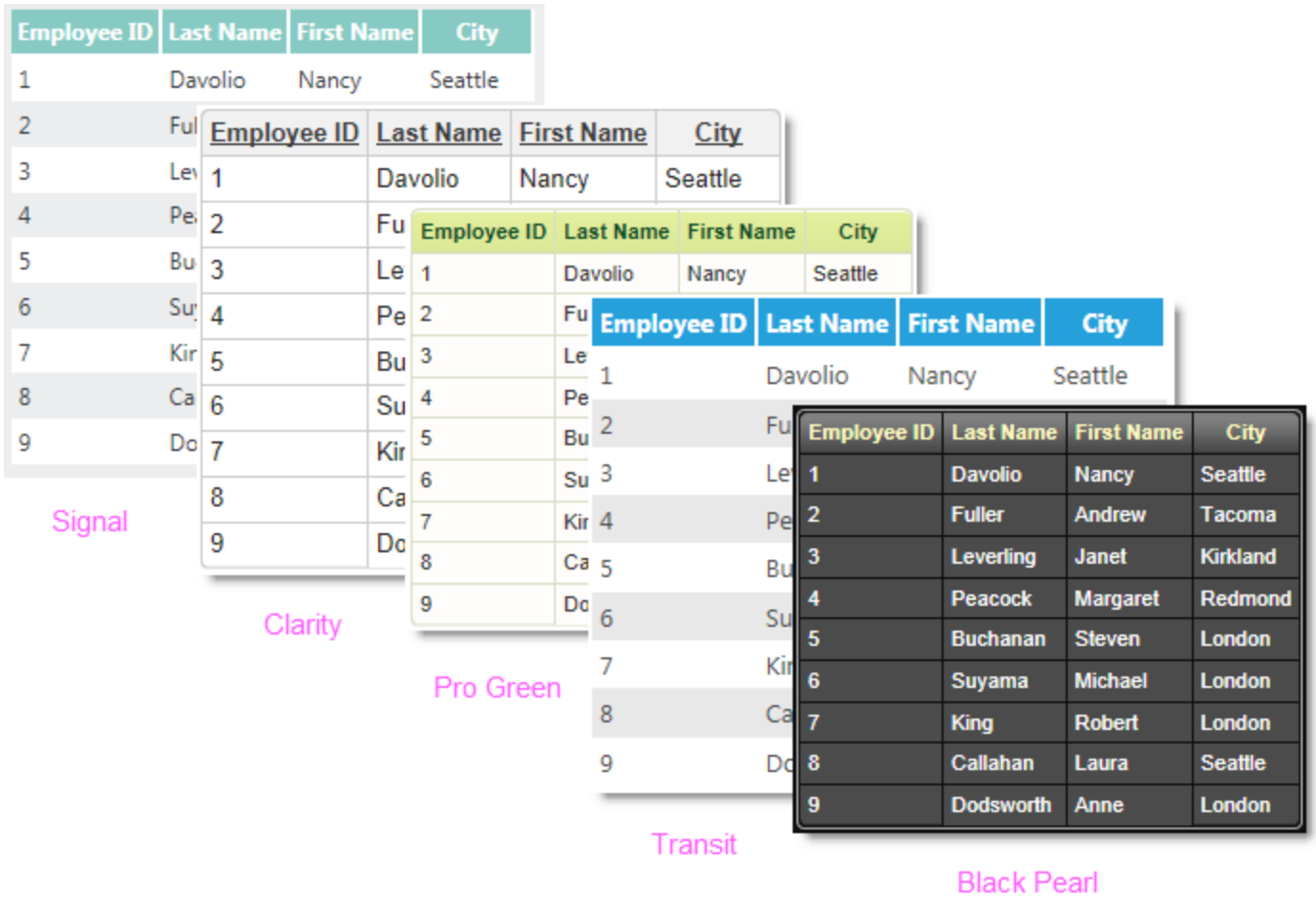
You can also view the results in your browser by right-clicking the **MyDataTable2** definition in the Application Panel and selecting *Run in Browser* from its context menu.

Change the Theme

Your Data Table report is using the *Signal* theme, which was configured in the Hello World application's _Settings definitionby default when it was created. However, you can experiment with other themes by assigning them to this report definition.



In your definition, select the report root element and add a **Style** element at the top of your definition, as shown above, Then select it and, using its drop-down list of choices, set the **Theme** attribute to something other than *Signal.*



Now preview the report and see what you get. Try a few other themes to see how they differ.

Themes are pre-configured settings and scripts that impart a specific look to the definition. They can be applied, as you did here,

at the report definition level or, for the entire application, in the _Settings definition. Finally, here are a few things to explore on your own:

- Examine the **Label** elements to ensure you understand how an @Data token relates to a column of data.
- Select any **Data Table Column** element, right-click it, and select **Remark**.

The element and all of its child elements now appear in a different color in the definition and, when you preview the report, that column is no longer visible.

Congratulations on completing this Data Table Tutorial.

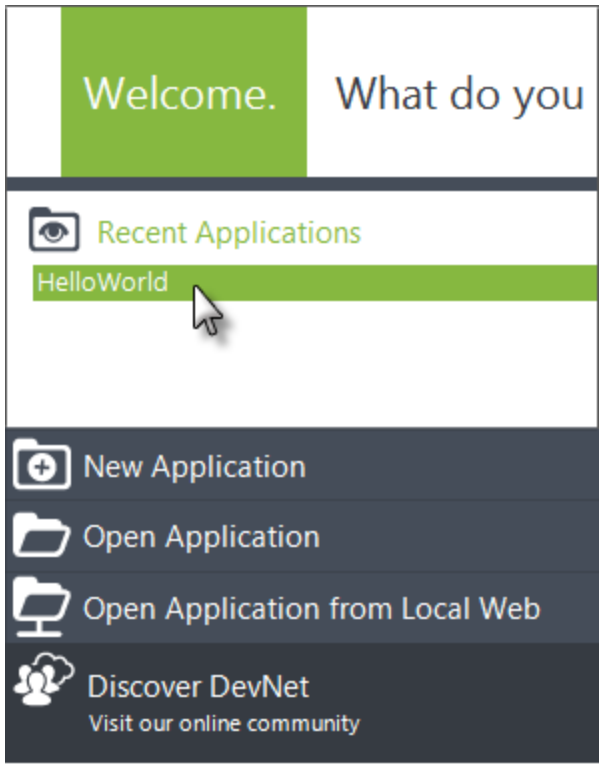
Data Table Tutorial - Wizard

This topic is for new Logi developers. It guides you through the process of building a Logi report that presents data from your SQL database in a table. It uses Studio's wizards to create the report.

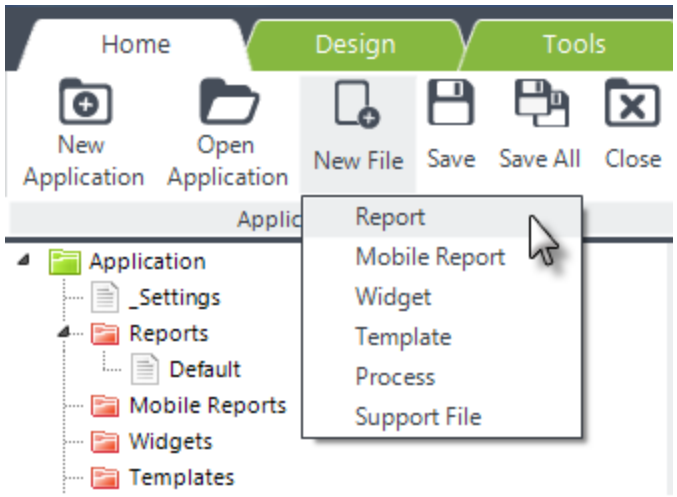
Requirements: This tutorial assumes that you have:

- Microsoft IIS web server and .NET Framework 4.x installed, - OR -
- A supported Java web server and Oracle JDK or OpenJDK 8, 11, 12, 13 (depending on your setup)
  Oracle has changed its Java usage policies - see Java Usage Policyfor important information.
- And an un-expired trial license, or a regular or OEM license file, installed in the *<Logi Info installation folder>*\LogiStudio folder, which defaults to:
  `C:\Program Files\LogiXML IES Dev\LogiStudio;`
- A working connection to a SQL database server;
- Credentials for accessing a table on the database server.

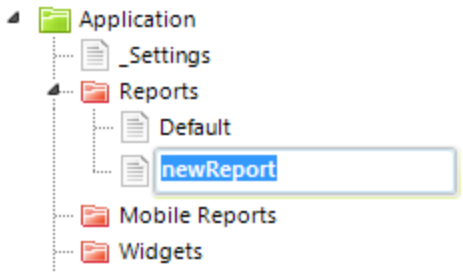
This tutorial builds on the "Hello World! Tutorial" on page 41. Please complete that tutorial before proceeding with this one. We also offer a more comprehensive tutorial, "Data Table Tutorial - Manual" on page 194, which does not use Studio Wizards and provides more "behind-the-scenes" insight into report creation. Begin by launching **Logi Studio**, using its desktop icon or Start → All Programs → Logi Info → Studio.

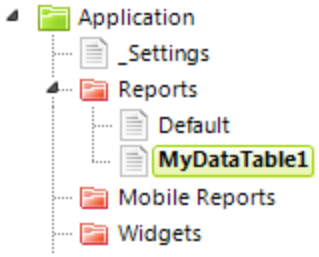


1. In Logi Studio, close the Getting Started dialog box and re-open the **HelloWorld** application from the Welcome Panel, as shown above.



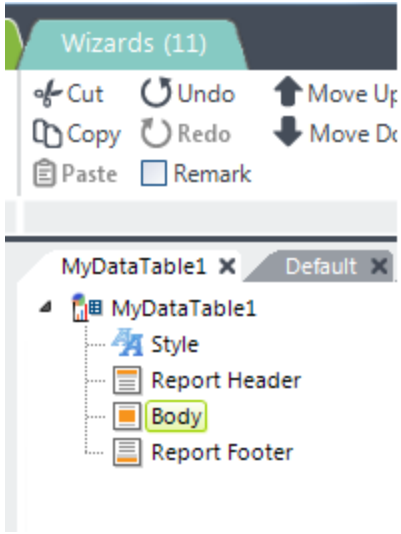
2. Click the **New File** menu item, and select *Report* from its drop-down list of options, as shown above.

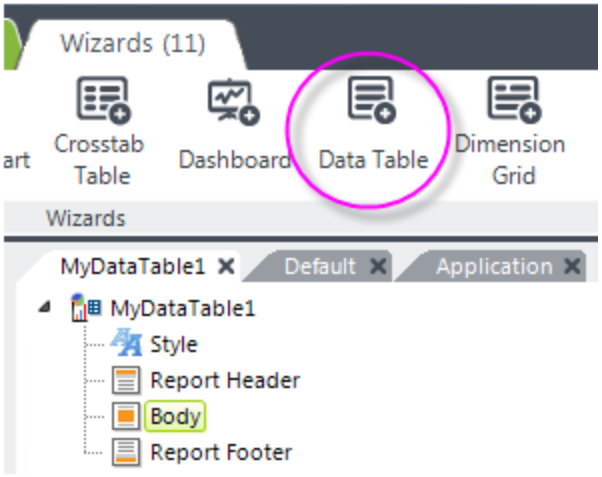




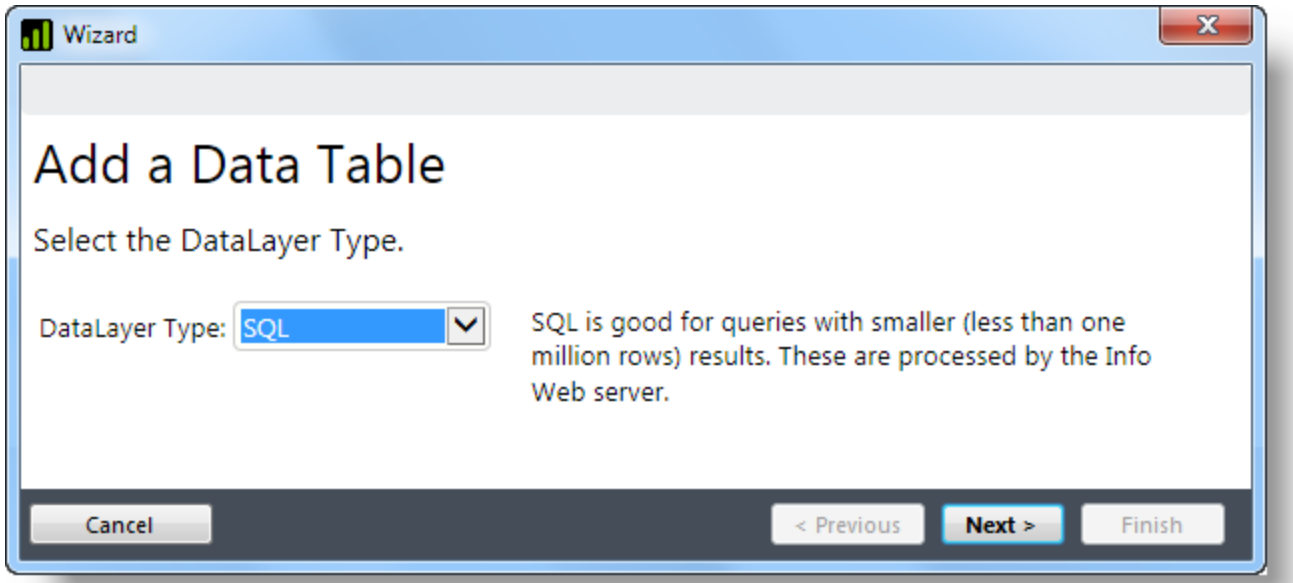
Studio will add a new report definition, named "newReport" in the Application Panel, and put it into rename mode. Just start typing a new name - such as "**MyDataTable1**" - and press Enter to rename it.

If something happens, like you click elsewhere, and it exits rename mode, just select the newReport definition and press F2, or right-click and select *Rename*, to rename it.

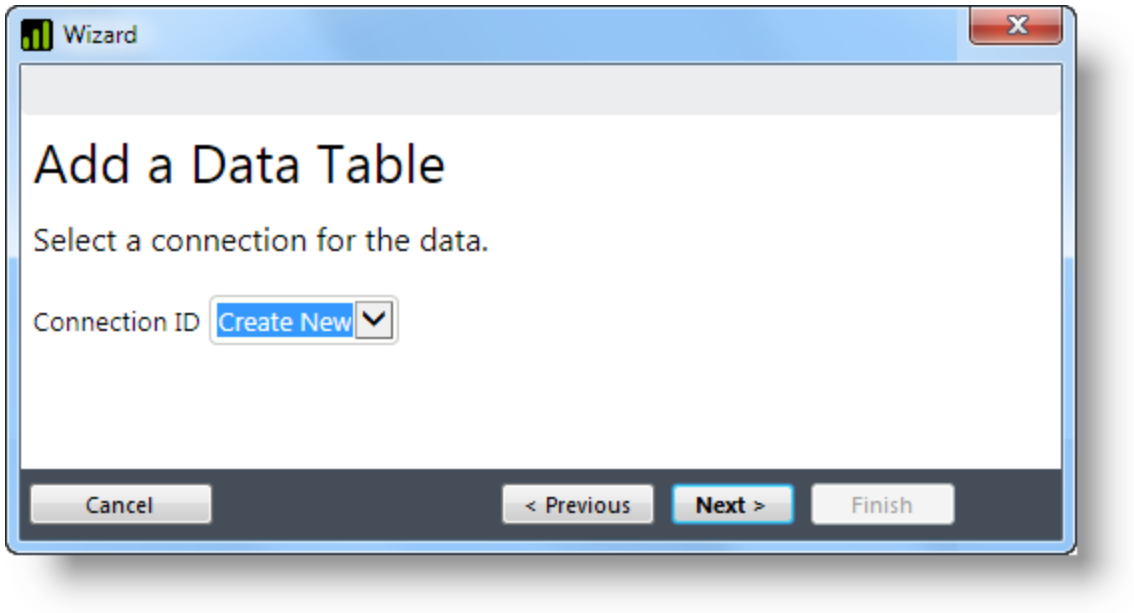




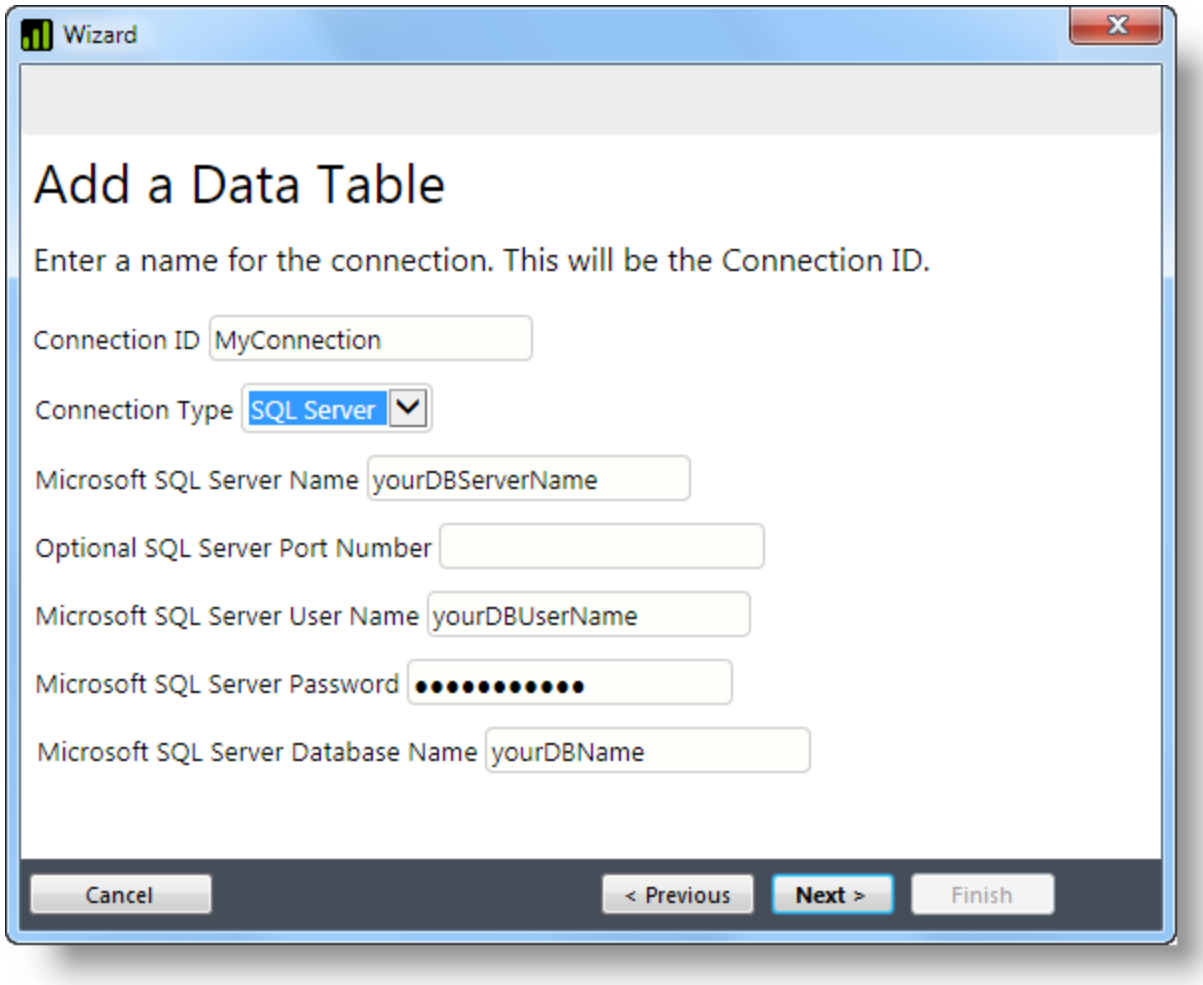
3. You probably noticed that the new definition was opened in the Workspace editor. Select its **Body** element and the **Wizards** tab will appear in the main menu. In the Wizards tab, click the **Data Table** menu item to start the Add a Data Table wizard.



4. The wizard will first ask you to specify the datalayer type. Datalayers retrieve data for use in reports. Use the default *SQL* datalayer type and click **Next**.



5. Data for this report is coming from a database server, so you need to establish a **connection** to it. Select *Create New* to create a new connection and click **Next**.

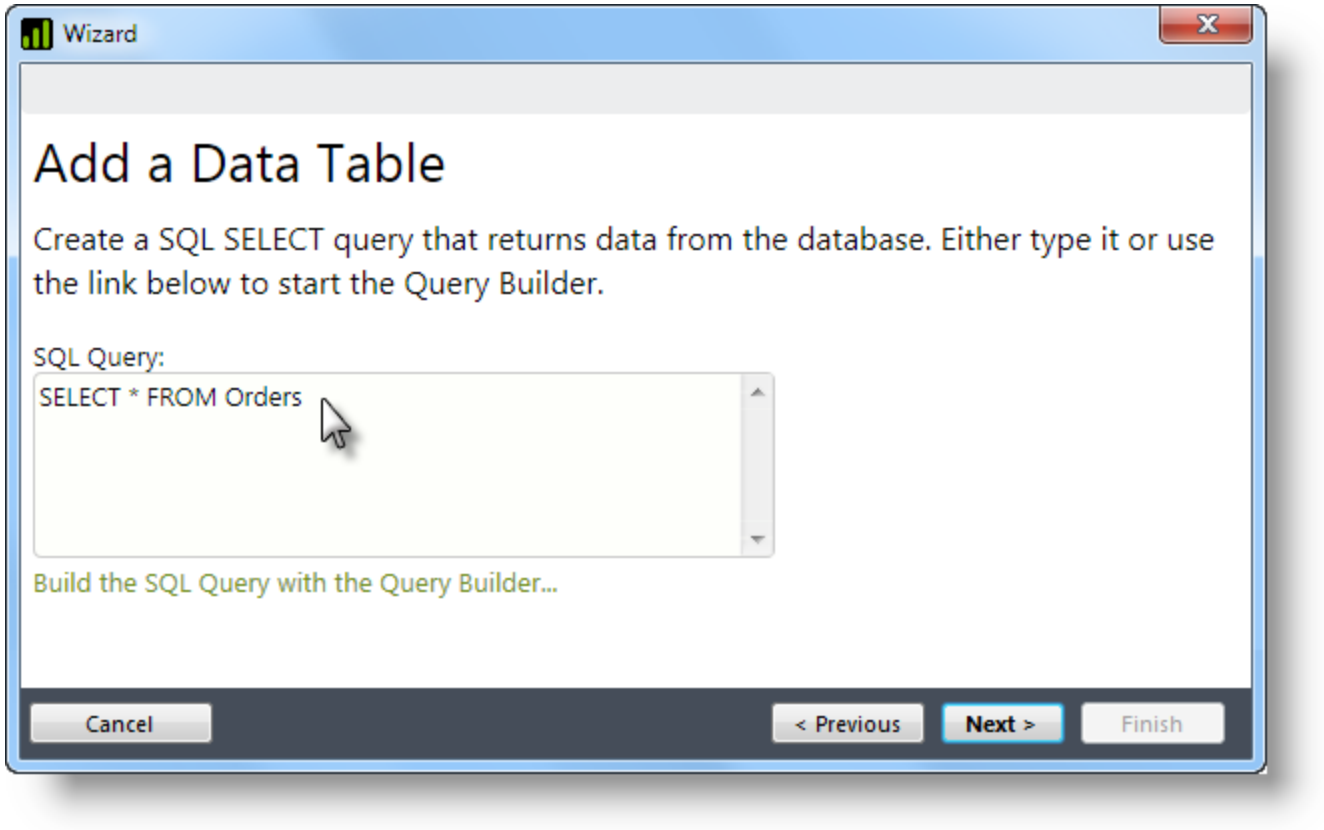


6. The wizard will prompt you, in separate dialog boxes, for all of the information shown above. To save space, we've grouped them into *one* wizard image. You'll provide information as necessary and click **Next** to proceed each time.

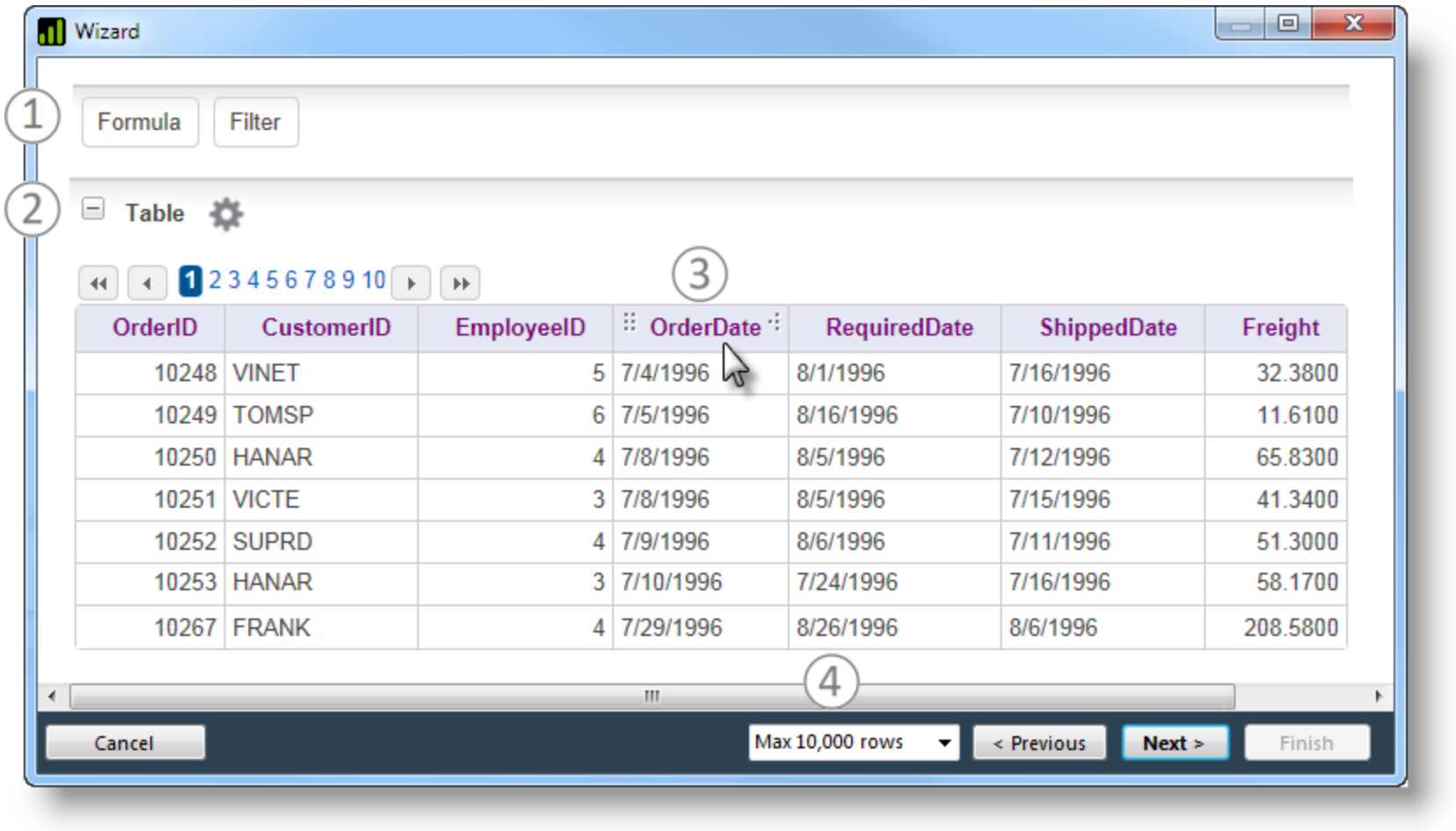
   The prompts shown above are for Microsoft SQL Server, but they'll vary depending on the Connection Type you select.

   After you provide the requested information, the wizard will test the connection and present its status. If there's a problem,

you can click **Previous** to go back and review or change your entries. Click **Next** again as needed to re-test the connection.



7. Once the connection is successful, you're ready to enter a SQL query. Skip using the SQL Query Builder this time and just type-in a simple query, like the one shown above, that will return some columns from your database. The click **Next** to continue.



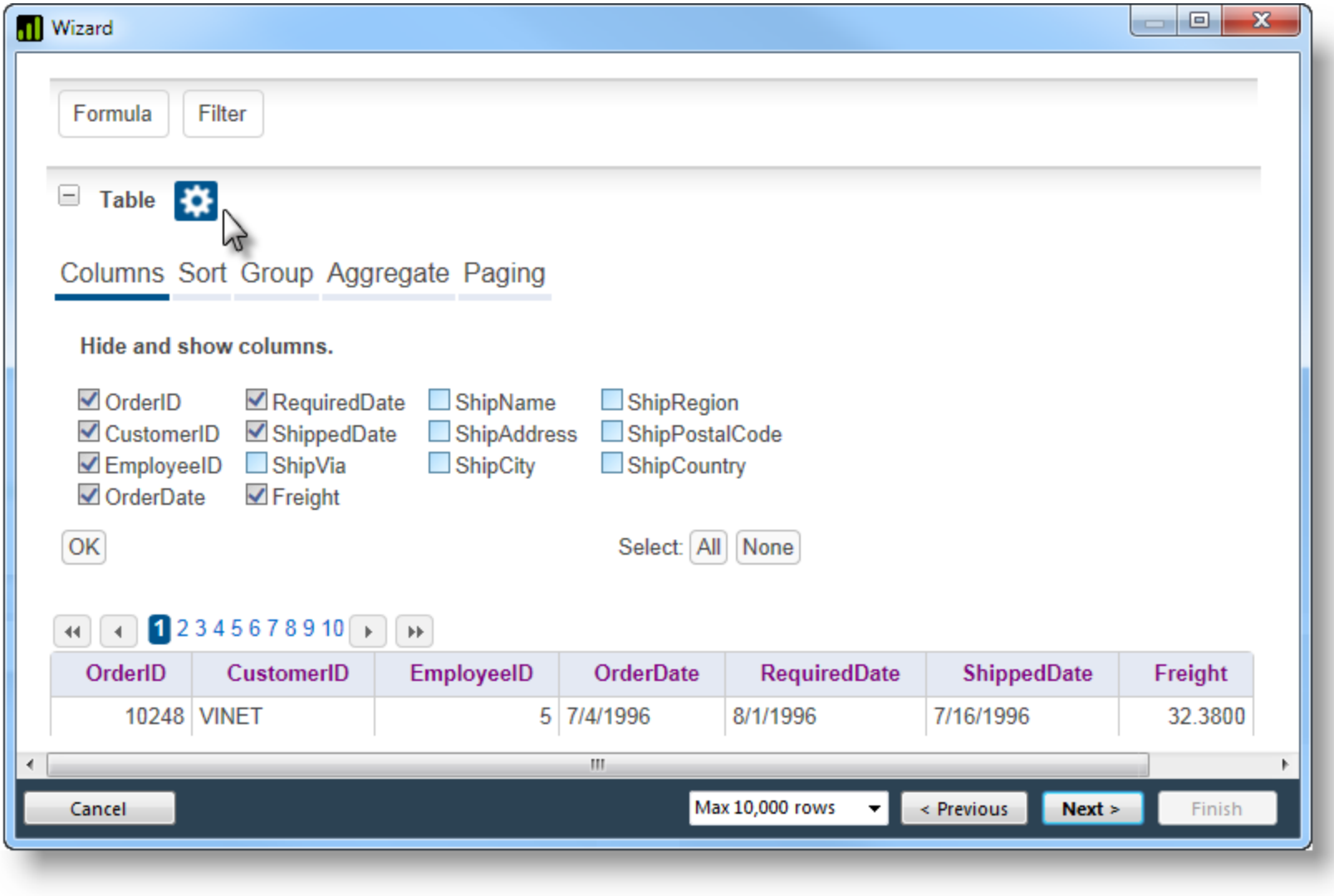
8. The wizard will display an example Data Table in a dialog box, where you can make adjustments to it, if desired. If the table is just as you want it, click **Next** to exit the wizard. If not, you can continue making adjustments, using the features identified above:

   1- Click these buttons to add calculated or "Formula" columns, and to Filter the data based on column values;
   2- Click the Gear icon to display the table's Configuration Panel (discussed below);

3- Hover your cursor over a table column header and use the drag handles that appear to change the column width or rearrange column order. Click the header text to see a configuration menu for that column (discussed below);
4- Select the maximum number of rows to be retrieved.

Table Configuration Panel

If you click the Gear icon, the table's Configuration Panel will be displayed:



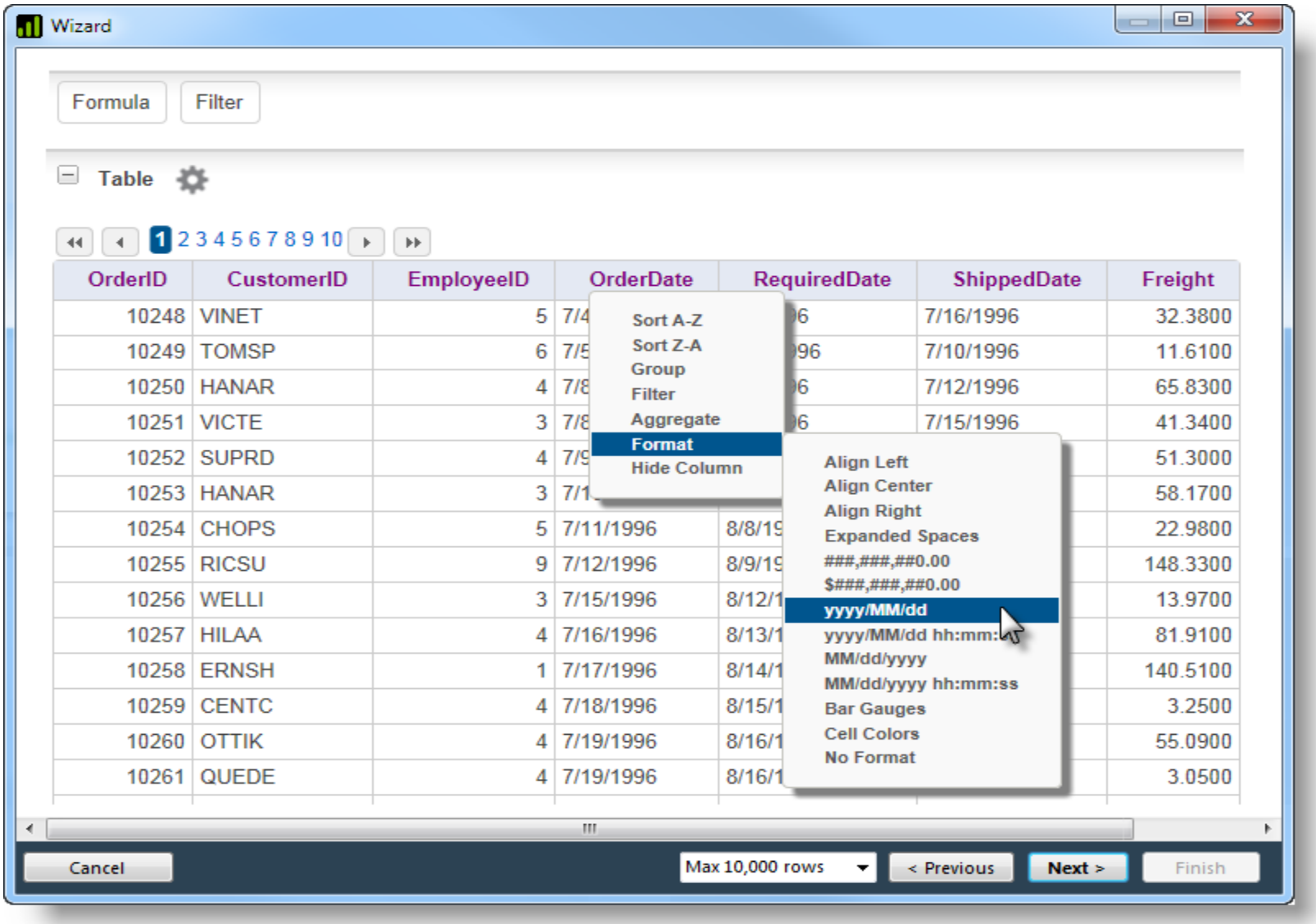
As shown above, the Configuration Panel includes these controls:

- **Columns** - Select the columns that will be displayed (displayed vs. retrieved in the data).
- **Sort** - Specify the sort order of the data after it's been retrieved.
- **Group** - Specify the grouping of the data after it's been retrieved.
- **Aggregate** - Add new columns for aggregations including *Sum, Average, Standard Deviation, Count, Distinct Count, Minimum,*and *Maximum*.
- **Paging** - Specify whether the table will use paging and, if so, how many rows will be shown per page.

Click the **gear** icon again to hide the Configuration Panel.

Column Configuration Menu

If you click a column's header text, a pop-up Configuration Menu for that column will be displayed:

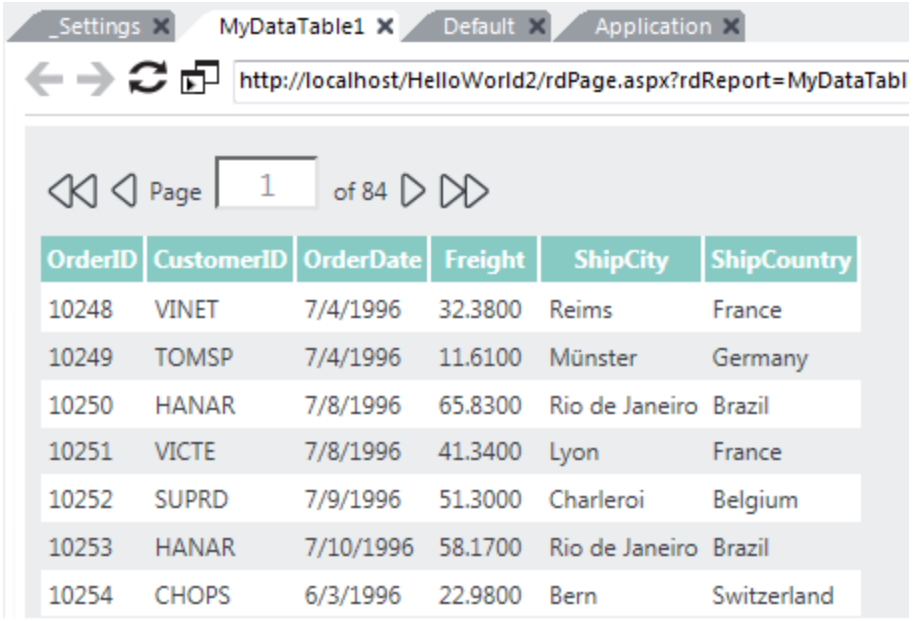


Options on that menu allow you to do some of the same thing you can do in the Configuration Panel but other options, like Format, work specifically on the selected column. Changes in the example table will take affect immediately. The wizard

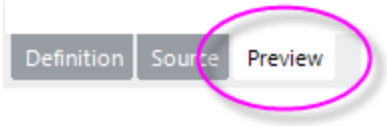
automatically gives each column interactive sorting capabilities.

9. When you're done, click **Next** and **Finish** and the Wizard will close. You may have noticed that the wizard was adding all of the elements for the Data Table to your definition as you went along.

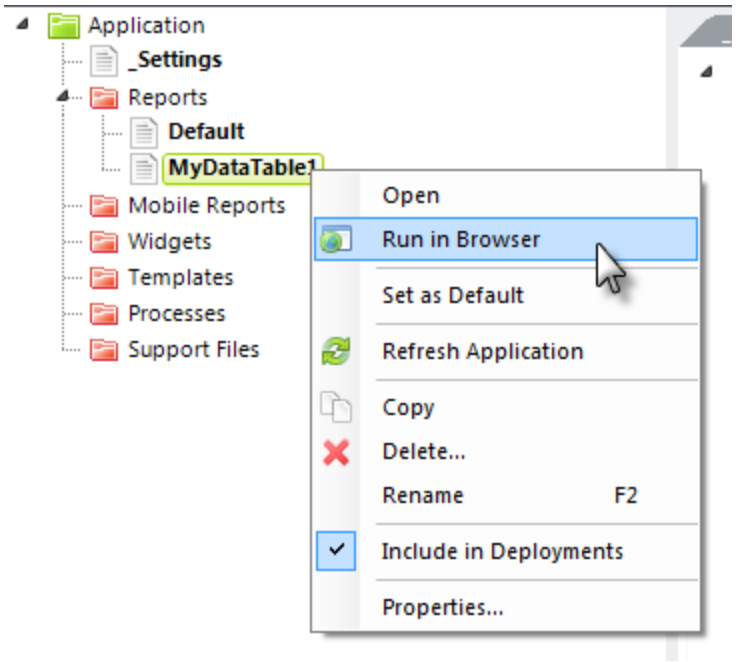
   Only the elements for the Data Table itself will be added. The wizard's configuration controls that you, as a developer, have been working with are *not* part of the Data Table and do not appear at runtime.



OrderID	CustomerID	OrderDate	Freight	ShipCity	ShipCountry
10248	VINET	7/4/1996	32.3800	Reims	France
10249	TOMSP	7/4/1996	11.6100	Münster	Germany
10250	HANAR	7/8/1996	65.8300	Rio de Janeiro	Brazil
10251	VICTE	7/8/1996	41.3400	Lyon	France
10252	SUPRD	7/9/1996	51.3000	Charleroi	Belgium
10253	HANAR	7/10/1996	58.1700	Rio de Janeiro	Brazil
10254	CHOPS	6/3/1996	22.9800	Bern	Switzerland



10. Click the **Preview** tab at the bottom of the Workspace editor to save your work and preview the report in Studio. It should look something like the example shown above.



If you'd like to view it in your default browser, select and right-click the report definition, as shown above, and select *Run in Browser* from its context menu.

Well done! We recommend that you take some time to examine the elements the wizard added to your report and _Settings definitions. In the report definition, in particular, notice the **Label** element added beneath each **Data Column** element. Look at its attributes and notice the **@Data** "token" used to represent the data from the datalayer. You can **rearrange** the order of the table

columns by selecting the **Data Table Column** elements and moving them up or down in the element tree in the report definition. You can drag them (along with their child elements) to a new position, or use the Move Up / Move Down items in the Home tab of Studio's main menu. Move a few and preview the report again to see the results. Congratulations on completing this tutorial.

Google Map Tutorial

This topic introduces the requirements for building a sample application using the Logi Info Google Map element. You are highly encouraged to read *Google Maps*first, in order to get a basic understanding of the topic before proceeding.

The following topics introduce the steps for using the Google Map element:

- Creating the Basic Application
- Configuring the Web Service Connection
- Creating a Basic Google Map
- Adding Interactive Features
- Geocoding Data

Requirements Checklist

Before starting this tutorial you should be sure that:

- You're using Logi Info **v12** or later.
- You've reviewed the Google Maps API Terms of Serviceand have a Google Maps API Premiere license if you need one.
- You've downloaded these sample data filesand have unzipped them to a temporary location. The files are:

```
HD_Dealers.xml, HD_BarShield.gif, BikeShops.xml, BikeShopIcon.gif
```

- You can connect to the Internet (outbound and inbound) from your development web server.
- You're familiar with the general steps for creating a new Logi Info application and for accessing databases to retrieve data.

If any of the above items are incomplete or missing, please take care of them before proceeding.

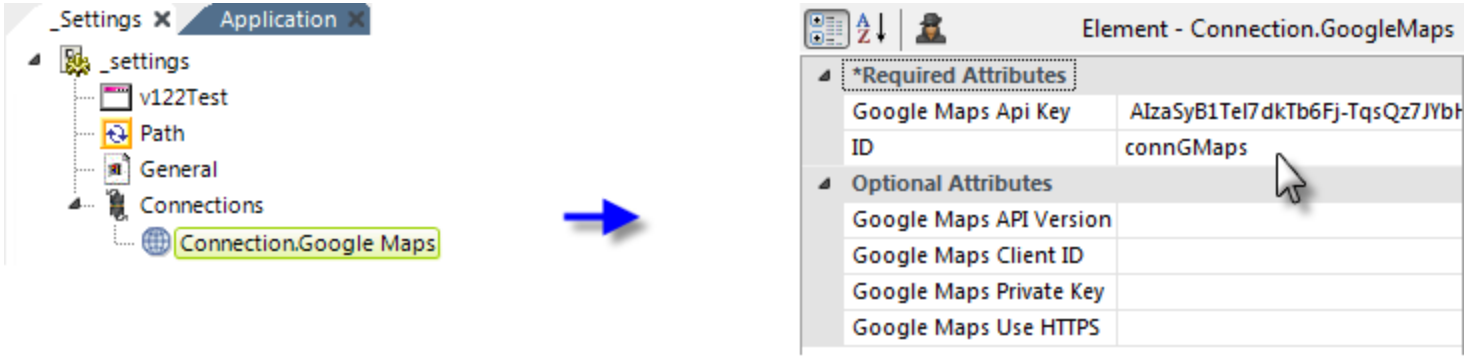
Creating the Basic Application

Let's get started by building your Google Map application and placing the sample data appropriately:

1. In Logi Studio, **create** and **register** a new application. Test it to ensure that it works.
2. Extract all four files from the sample data .zip file downloaded in the previous section and save them in the _SupportFiles folder under your Logi application folder.

Configuring the Web Service Connection

The web service connection requires the **Connection.Google Maps** element:



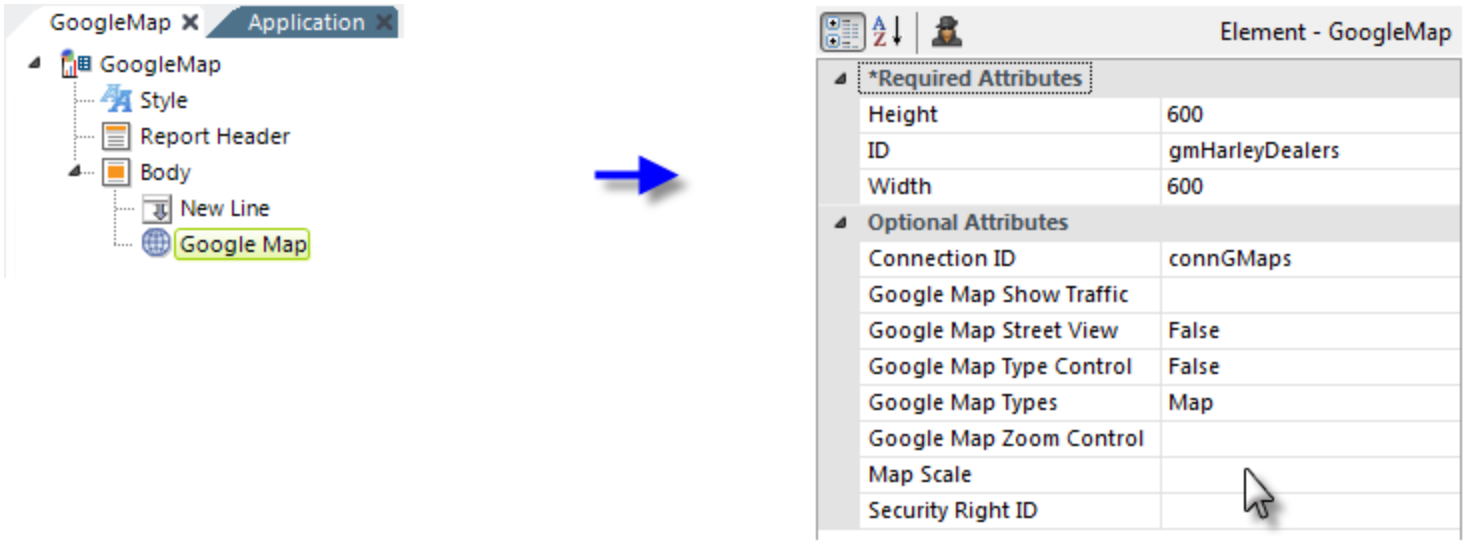
For detailed information about getting an API Key and configuring this element, see *Google Connections*.

Logi Info now includes the **Connection.Nominatim** element, for connections to a free, public, OpenStreetMap (OSM) Nominatim server for geographic data. More information is available in *Datasource Connections*.

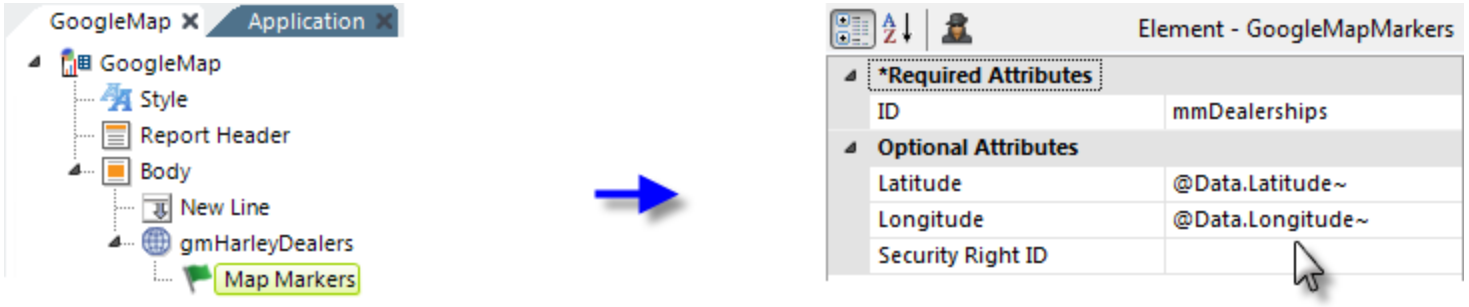
We don't need any other connections for this tutorial: you're going to create sample Google Maps using data from two different XML data files, which do not require connections.

Creating a Basic Google Map

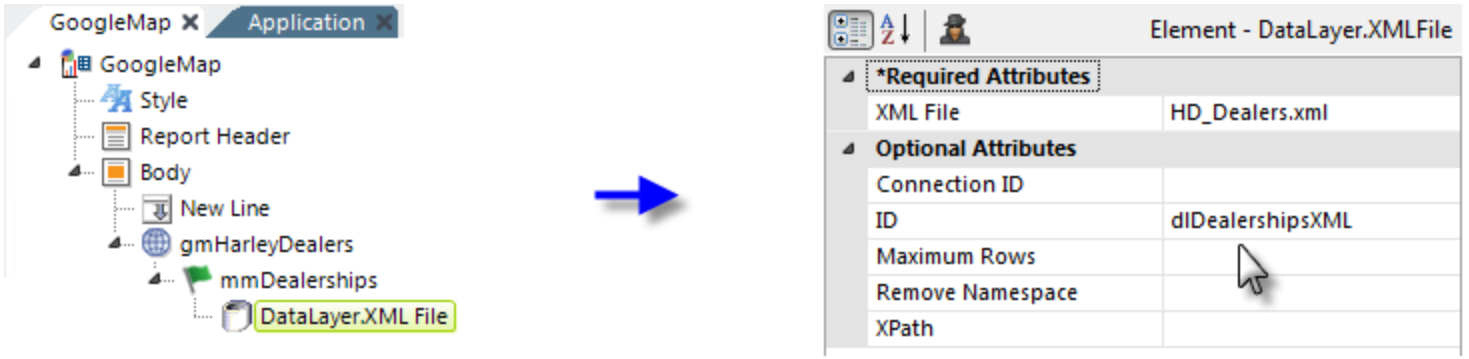
Now, with just a few quick steps, you'll produce a functional Google Map. The map will show you the locations of **Harley-Davidson Motorcycle Dealerships** in the state of Illinois:



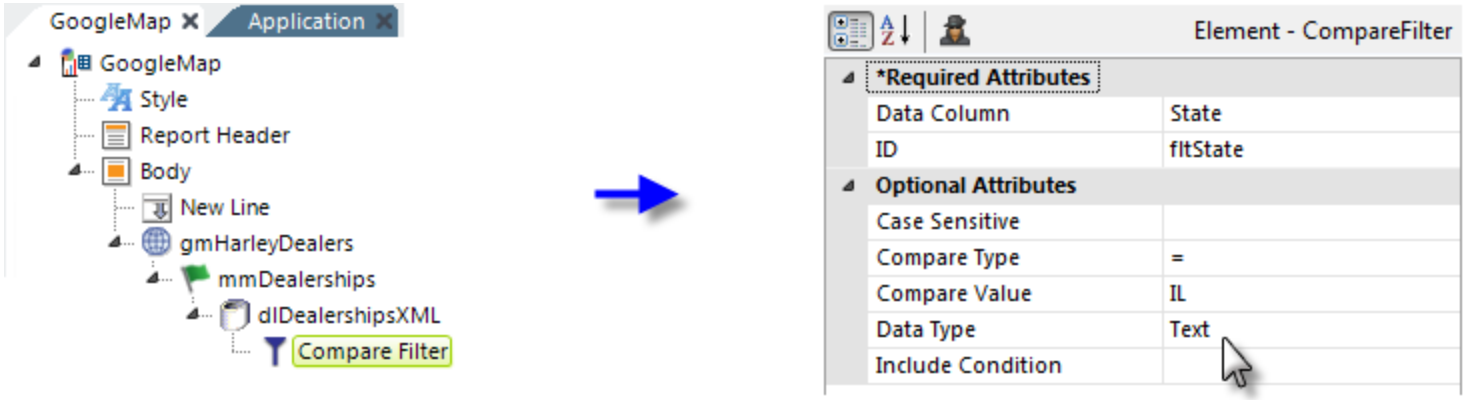
1. In your report definition, add a **Google Map** element and configure its attributes, as shown above.



2. Add a **Map Markers** element below the Google Map element and configure its attributes as shown above. The sample database you're working with includes columns named "Latitude" and "Longitude", which contain *positioning data*. Later on, you'll discover how to get that data from Google if you don't have it.

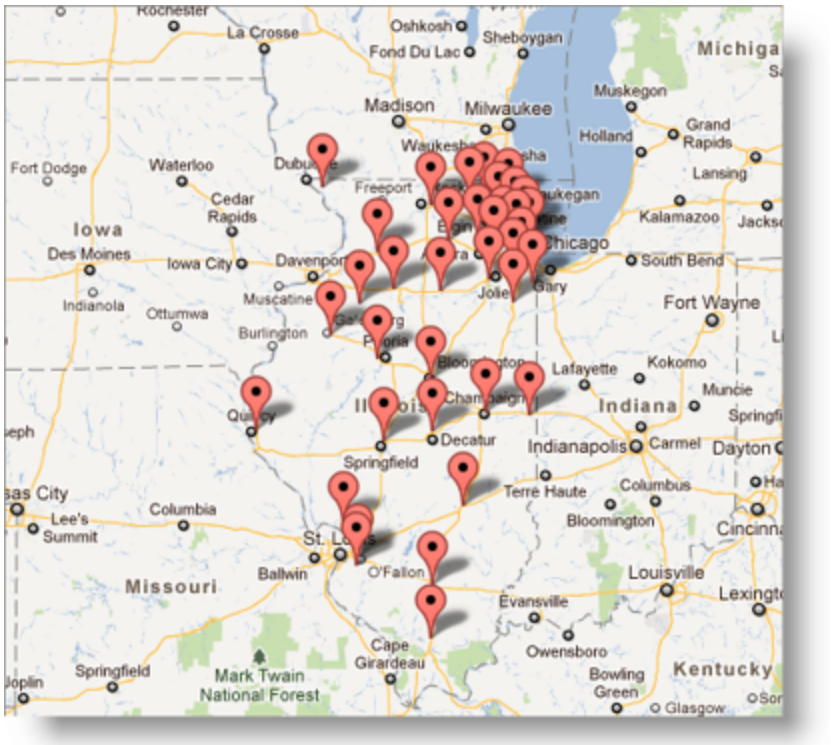


3. Add a **DataLayer.XML** element below the Google Map Marker and configure its attributes as shown above. The XML data file should be in the drop-down list of choices for this attribute.



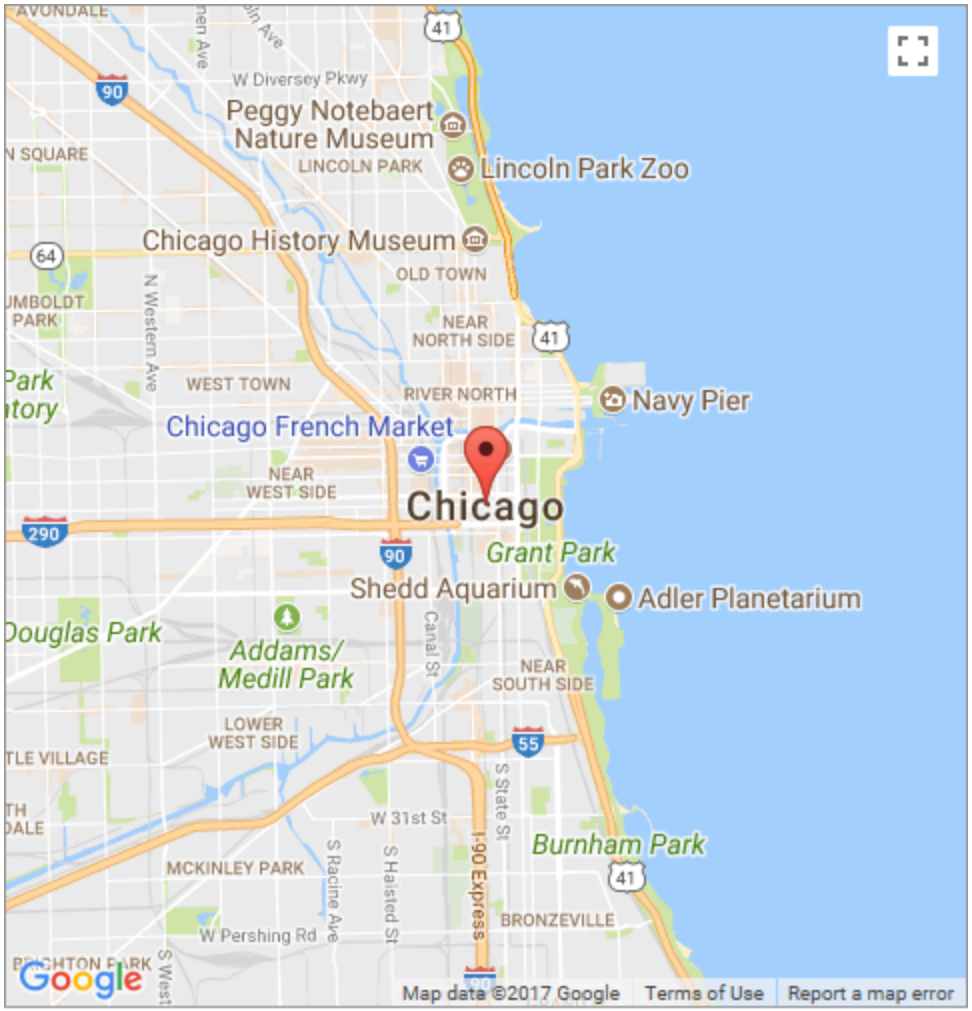
4. Finally, add a **Compare Filter** element beneath the datalayer, as shown above, and set its attributes as shown, to filter out all records except those where the State column equals "IL", the postal abbreviation for Illinois.

Now you're ready to create your first Google Map. Save and browse your application. The results should look something like this:



At this point, take some time to play with this map:

- Change the Compare Filter element's attributes to get data for a different state or for a city. Try Data Column = City, Compare Value = Chicago, for example, and notice how the map scales to the data.



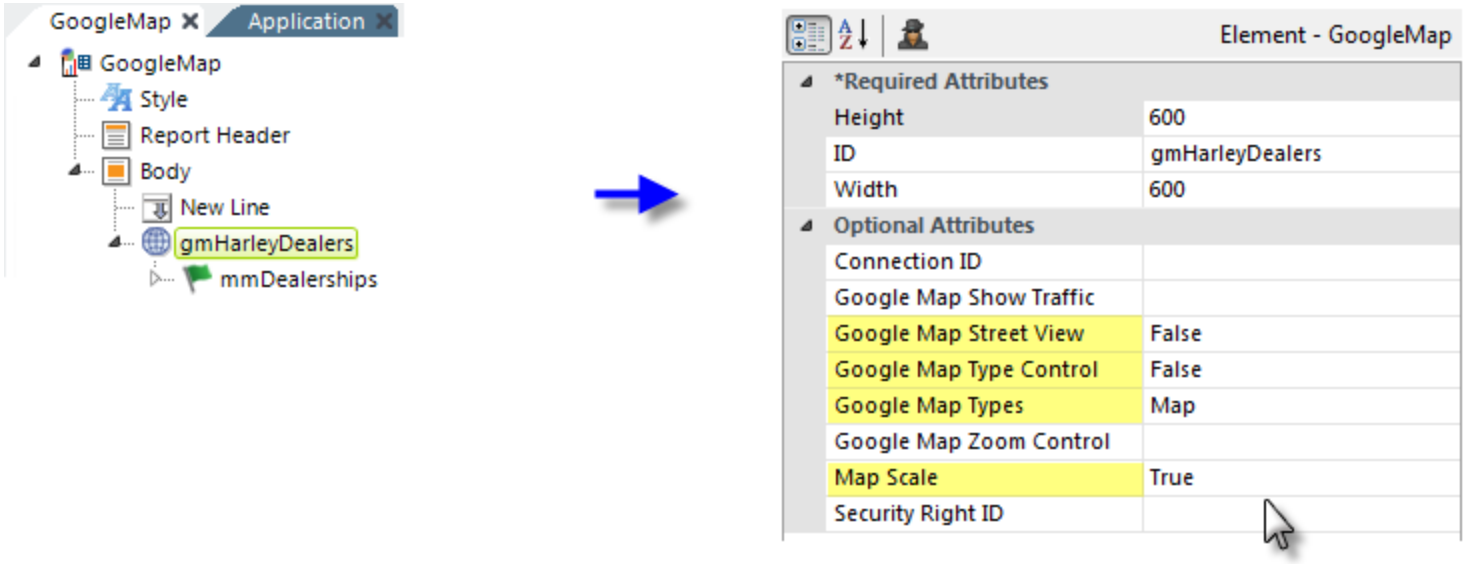
- Your cursor is a "hand" when it's over the map, meaning that you can *pan the map* by dragging the cursor. Give it a try!
- Change your Google Map element's **Height** and **Width** attributes and see what happens to the *map scale*.

Adding Interactive Features

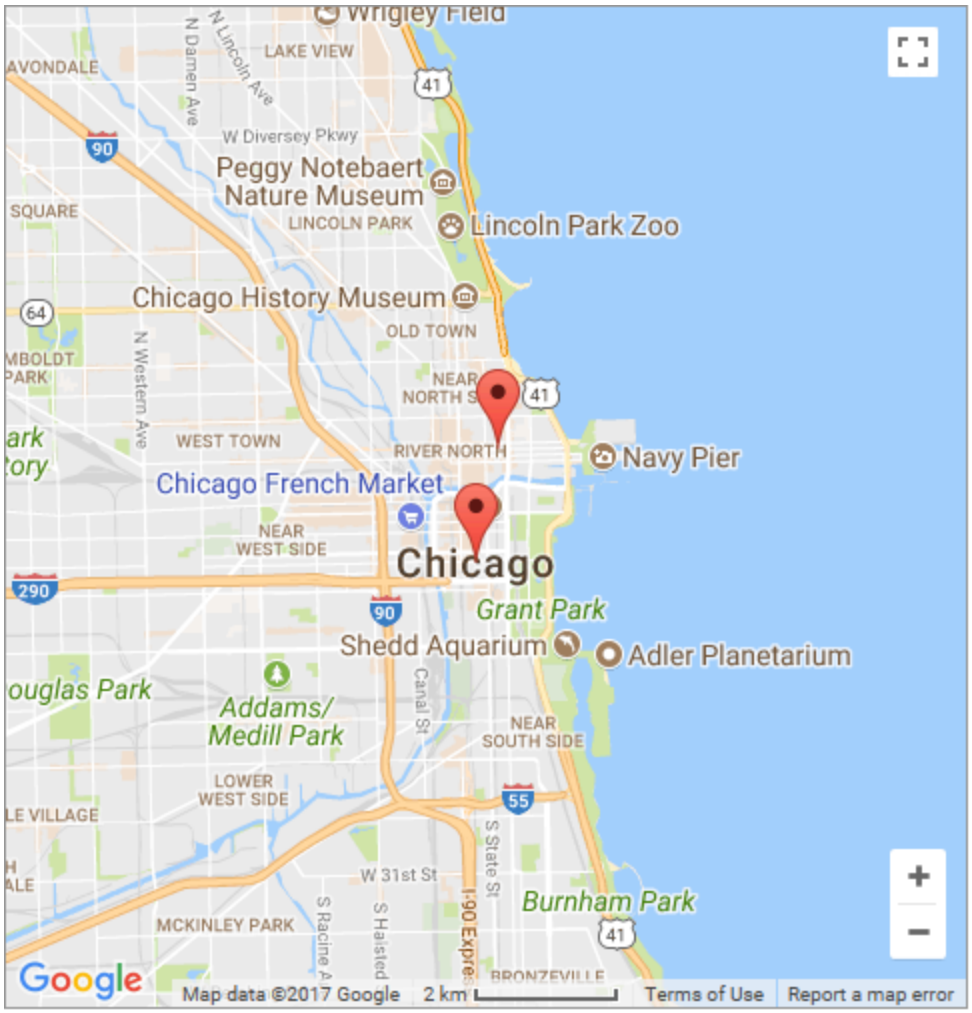
There are several ways to customize your map to make it more interactive. First, ensure that you change the Compare Filter element's attributes to get data for Chicago only (Data Column = City, Compare Value = Chicago).

Put Controls on the Map

You can make your map more interactive by adding user controls to it:



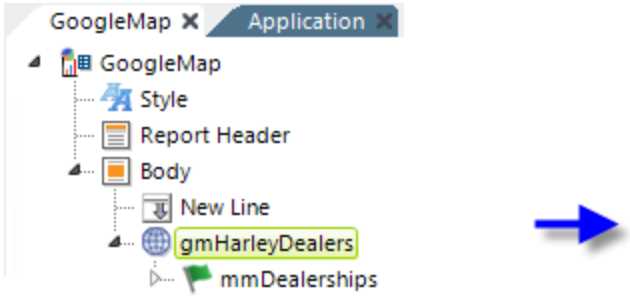
Revisit your **Google Map** element and set its **Google Map Street View**, **Google Map Type Control**, **Google Map Types**, and **Map Scale** attributes, as shown above.

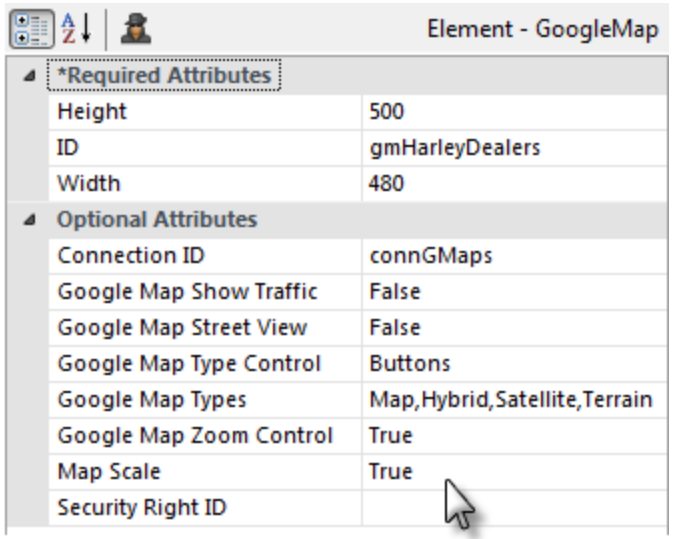


Save and browse your application. The map should now include **Zoom** controls and a **Scale legend**, as shown above. Try out the Zoom controls - they're actually a lot of fun to use!

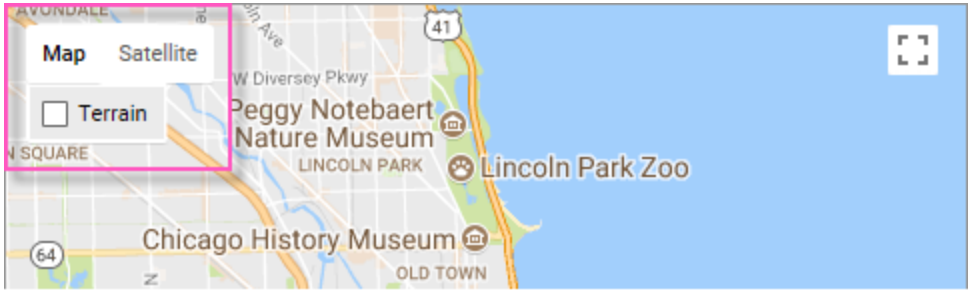
Switch Between Map Types

The Google Map has other map types available including: a *terrain* map, a *satellite* image, and a *hybrid* combination of the satellite image overlaid with the street map. These can be selected at runtime by the user if you add the proper controls:





Go back to your Google Map element again and set its attribute as shown above.

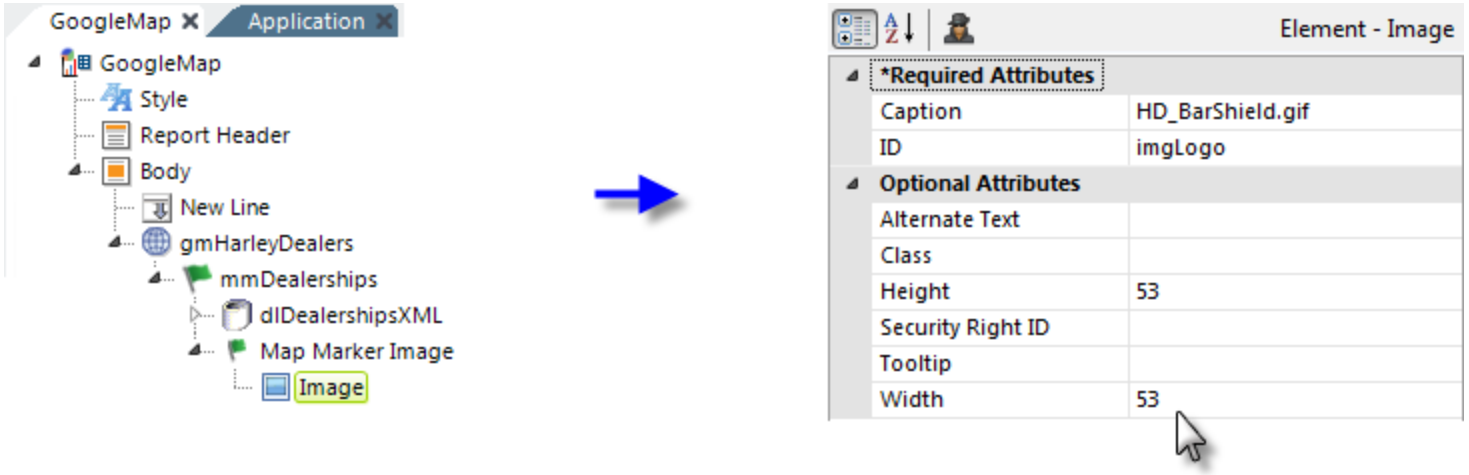


Save and browse your application. The map should now include the **buttons** shown above, which allow the user to switch between map types. Click the buttons and see what happens.

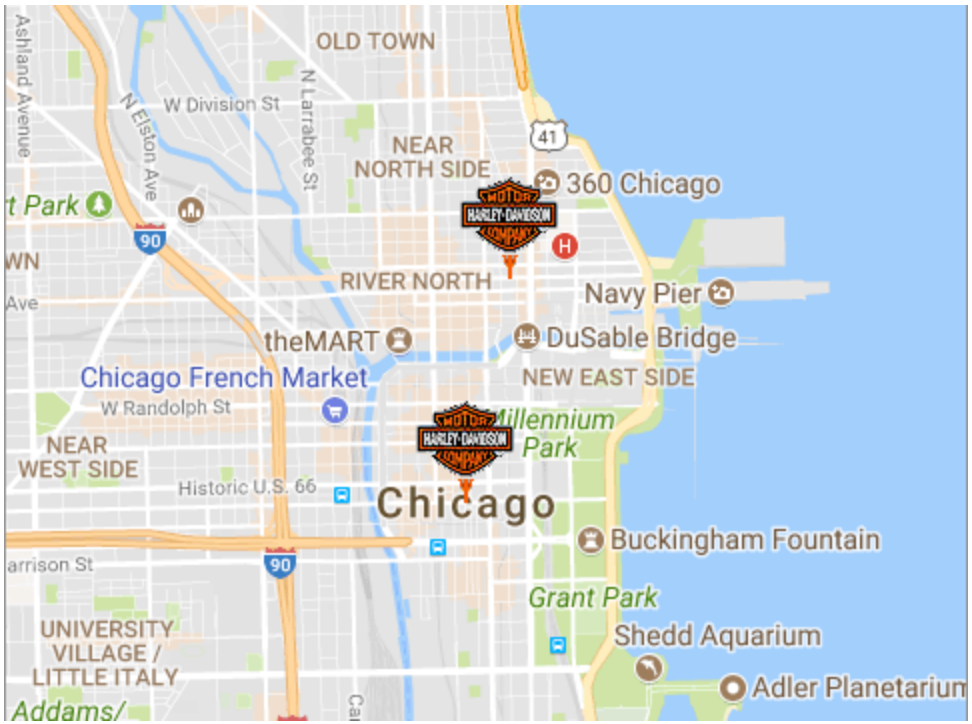
Some of the map types are selected using check boxes that appear when you click a button.

Display a Custom Map Marker

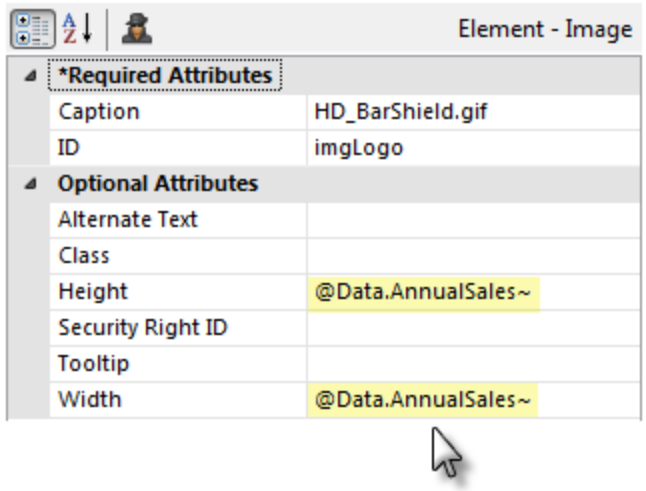
The map of Harley-Davidson dealerships we created earlier can be enhanced by **changing the graphic** used to indicate **locations** on the map:



1. In your definition, beneath the Google Map Markers element ("mmDealerships"), add a **Map Marker Image** element. This element has no attributes.
2. Beneath it, add an **Image** element and set its attributes as shown. The image is from the data .zip file you downloaded earlier. A **Height** and **Width** attribute value is required when using an Image element in a Google Map.



Save and browse your application, and see that the stock map marker has been replaced by the custom image, as shown above. Naturally, you have to exercise some caution when designing the size of alternate marker images so that you don't completely hide the map behind them.



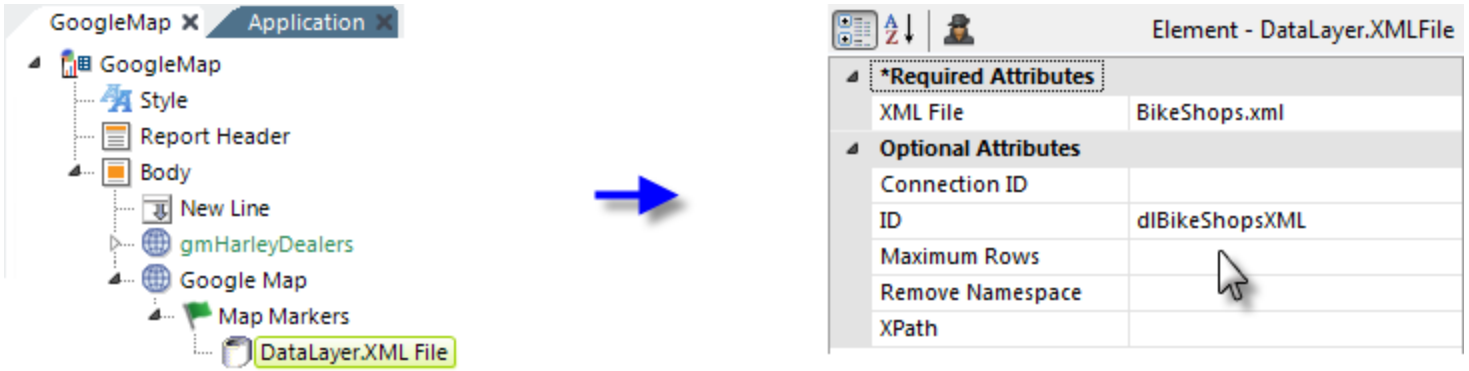
Speaking of size, you can cause the marker image itself to convey *relative data* by setting its **Height** and **Width** attributes to some value from your database. In the example above, the token for a sales revenue data column (just an example here, it's not in the sample data) is used in the attribute values. The images on your map will then appear larger or smaller based on the data, allowing easy visual comparison.

Geocoding Data

So far, you've been working with a sample database table that includes *address information* and *positioning data* (latitude and longitude).

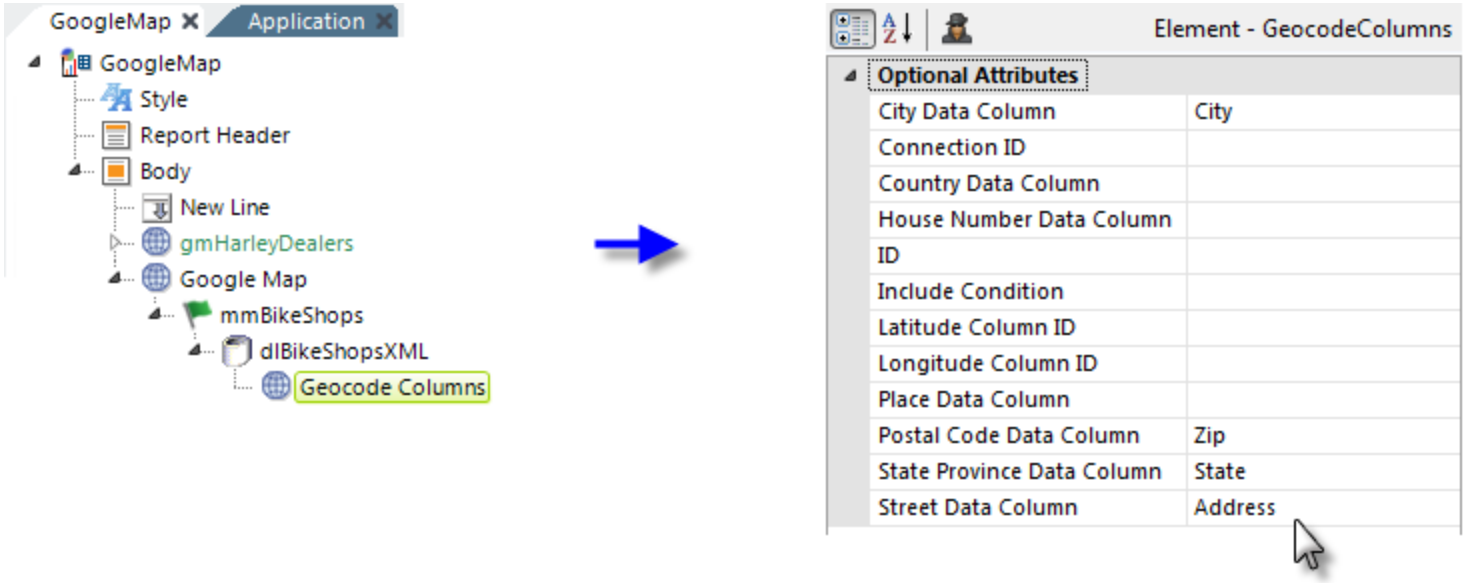
But what if you're working with a database that doesn't have that positioning data? Most location-oriented databases only have address information such as street address, city, state, and zip code. Well, if you have address information or place names, you can use a Google Map "helper" web service to get the positioning data. Even partial address data, such as zip codes alone, can result in useful positioning data.

The way this works is: you send your address information to the helper service and it will return it with the positioning data (if it can find it) appended. Practically, what happens is that your datalayer gets two columns added to it, with the default names "Longitude" and "Latitude". This process is called "geocoding". The following exercise will show you how it's done.



1. In Logi Studio, go to your **_Settings** definition and select the **General** element. Set the value for the **Debugger Style** attribute to *DebuggerLinks*.

2. In your main report definition, *collapse* and *remark* your "gmHarleyDealers" Google Map element and add a new **Google Map** element to your definition, as shown above. Set its **ID** attribute to *gmBikeShops* and set its other attributes to match those of the remarked Google Map element.
3. Beneath the new Google Map element, add a **Map Markers** element, as shown above. Set its **ID** attribute to *mmBikeShops* and set its other attributes to match those of the remarked Map Markers element.
4. Beneath that element, add a **DataLayer.XML** element and set its attributes as shown above. It will use one of the .xml data files from the .zip file you downloaded earlier.

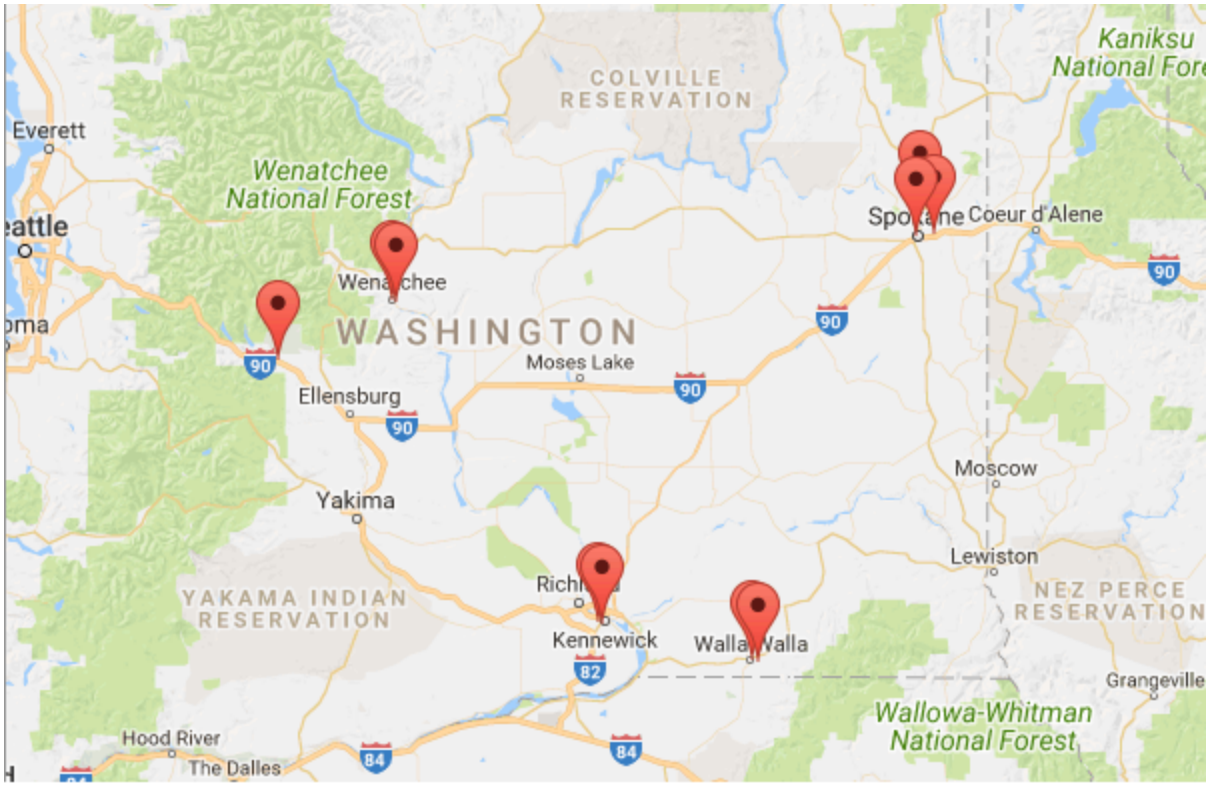


5. Add a **Geocode Columns** element beneath the datalayer and set its attributes as shown above. If you're using a Connection.Google Maps element, enter its ID in the **Connection ID** attribute.

   Also, notice that there are attributes for the **Latitude** and **LongitudeColumnIDs**; these default to "Latitude" and

"Longitude", which also happen to be the names of the columns that the Google service appends to your data. Therefore, you can leave them blank for this exercise.

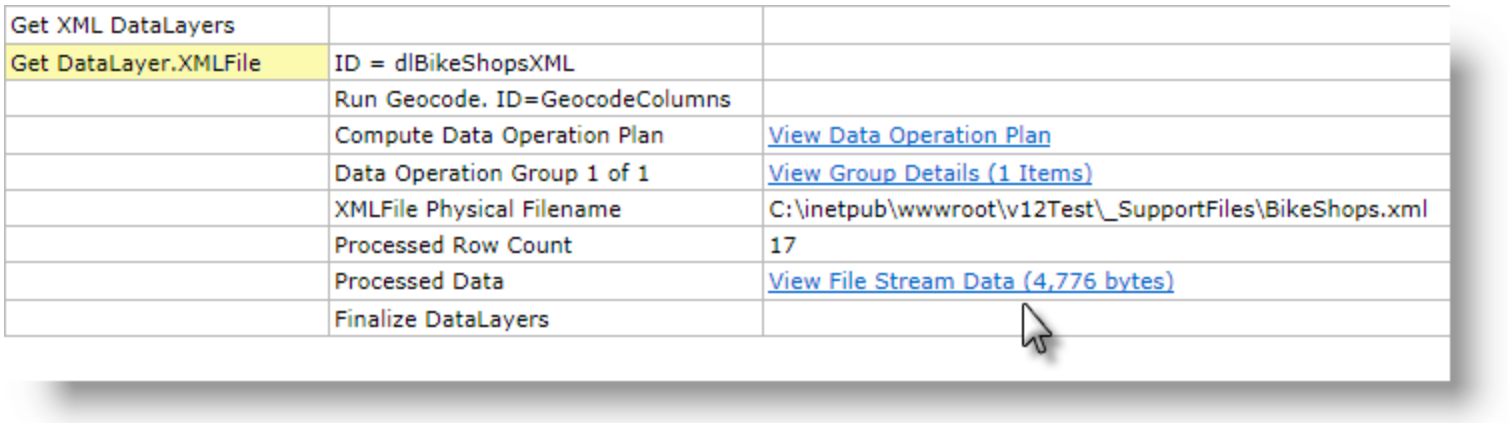
There's also a **Place Data Column** attribute. This is used if you don't have address data but instead have the names of points-of-interest, such as "Washington Monument" or "Brooklyn Bridge".



Save your changes and browse or preview the application. Your map of bicycle shops in Washington State should look like the image shown above.

Examine the Geocoded Data

Let's look at what the geocoding process actually did. Click the Debug icon in the bottom-right corner of your map page. Your map page will be replaced with the **Debugger Trace** page.



Get XML DataLayers		
Get DataLayer.XMLFile	ID = dlBikeShopsXML	
	Run Geocode. ID=GeocodeColumns	
	Compute Data Operation Plan	View Data Operation Plan
	Data Operation Group 1 of 1	View Group Details (1 Items)
	XMLFile Physical Filename	C:\inetpub\wwwroot\v12Test_SupportFiles\BikeShops.xml
	Processed Row Count	17
	Processed Data	View File Stream Data (4,776 bytes)
	Finalize DataLayers	

Look down the page and, in the Event column, find the **Get DataLayer.XML** entry, as shown above. A little further down, in the right-hand column, look for and click the *View File Stream Data* link, as shown above (the number of bytes may vary).

Logi **Debugger Data View** View raw XML

mmBikeShops **Rows: 17**

Longitude	Latitude	Employees	BMX	Cruiser	Freestyle	Comfort	Mountain	URL	Phone
-120.3106440	47.4192930	20	18	12	14	31	25	www.fullcircle.com	(509) 66
-120.3115870	47.4103480	40	14	24	12	14	36	www.bikedoc.com	(509) 66
-120.3309460	47.3550830	34	23	22	11	16	28	www.valleybikes.com	(509) 99
-120.2944902	47.4157567	32	20	31	17	18	14	www.skisite.com	(509) 88
-120.6560340	47.5976625	25	6	32	13	27	22	www.dasradhaus.com	(509) 54
-120.9327130	47.1944760	55	24	21	9	34	12	www.cleelumroslyn.org	(509) 67
-117.4329018	47.6565053	26	21	19	13	16	31	www.twowheeltransit.com	(509) 74
-117.4115728	47.6732905	38	11	16	16	12	45	www.wheelsport.com	(509) 32
-117.4356909	47.7175264	75	26	14	18	32	10	www.bicyclebutler.com	(509) 32
-117.3290820	47.6685820	22	33	18	21	12	16	www.hairygary.com	(509) 53

You should see the data in a table that was retrieved into your datalayer from the data file. Notice that **two new columns,** *Latitude* and *Longitude*, with positioning data have been added to each record by the Google Map service.

** WARNING **	Address could not be geocoded. Too many queries or you are over your daily quota. Google GeoCode Status: OVER_QUERY_LIMIT	, 10503 N Division St, Spokane, WA, , 99218

Return to the Debugger Trace page again and look down the Event column for ** WARNING ** messages. You can see here what happens in the trace when an address *can't be geocoded*. This maybe because of something extraneous in the data or you may have reached your allowed Google Map query limit.

Geocoding is a slick way of getting positioning data based on your address information. However, keep the following in mind:

- *Each data record* submitted for geocoding is considered a "transaction" and counts against your daily allotment as described in the terms of your license with Google.
- Geocoding takes *time*; the more records to be processed, the longer a delay the user faces so, for example, geocoding a huge database of addresses in real-time is not recommended.
- *Errors* in your address data not only take geocoding time but the failed attempts still count as transactions, so it's in your interest to have your address data be as *clean* and *accurate* as possible before geocoding.
- The Google service also "throttles" geocoding requests to keep itself from being overwhelmed and generally returns an "Error 620" if too many requests are received too quickly. The threshold is described in the terms of your license with Google.

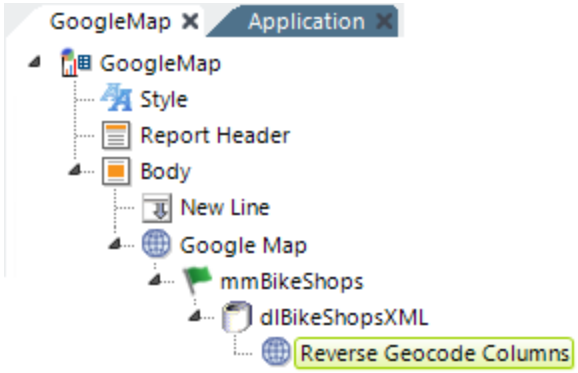
Logi Info now includes the **Connection.Nominatim** element, for connections to a free, public, OpenStreetMap (OSM) Nominatim server for geographic data. More information is available in *Datasource Connections*.

The Include Condition Attribute

The Geocode Columns element has an **Include Condition** attribute. If the value of this attribute is left blank or contains a formula that evaluates to *True*, the element is utilized with the map. If the value evaluates to *False*, the element is ignored. This feature allows developers to dynamically switch between different sets of geocoding data.

Reverse Geocoding

Suppose you have the opposite location data situation from the one discussed above: you have the positioning data but not the address information. A typical use-case would be a mobile device app that "knows" its own GPS coordinates and needs to display the address.



The **Reverse Geocode Columns** element allows you to feed positioning data to the Google web service and retrieve address information. The web service will attempt to append and return data for the following columns into the datalayer: StreetNumber, StreetName, Locality, Sublocality, AreaLevel1, AreaLevel2, AreaLevel3, Country, Latitude, Longitude, and PostalCode.

Logi Google Map API

The Logi Info **Google Map** element allows developers to include the geo-mapping features of the Google Maps web service in their reports and provides an opportunity to combine it with their own data to produce hybrid maps. A JavaScript-based API is available, providing developers with an opportunity to handle low-level events and customize their maps.

The following topics discuss the Logi Google Map API:

About the Logi Google Map API

The Logi Google Map API is a JavaScript-based API that lets you handle Google Map-related events and objects. It depends on other JavaScript objects that are automatically created by the standard Logi Engine code. These objects are clearly identified in the example provided. The Logi Google Map API extends Logi's Google Map element, letting developer dynamically configure map features and markers. It's based on the Google Maps JavaScript API v3. The Logi Google Map API is contained in `rdTemplate\rdGoogleMap\rdGmap.js`, which is automatically included in your Logi application when you use the Google Map element. You *do not* need to include this script file independently.

You can include the JavaScript code you write that uses Logi Google Map API calls using the standard elements for that purpose, such as **Include Script** or **Include Script File**. If you use the latter, it must be a child of the Body or Report Footer elements, not the Report (root) or Report Header elements.

Events and Objects

The API provides the following events and objects:

Y

This object is an underlying Logi application script object, automatically created. It's used in your code to identify the next two events.

rdCreate

This event fires *before* the Google Map object is created. It has three properties:

- **id** - Gets the ID of the Google Map object about to be created.
- **mapOptions** - Sets the Google Map's MapOptions object properties, ranging from background color to zoom control options. A complete list of these is available from Google on their specifications page.
- **container** - Gets the ID of the internal Division created as a container for the Google Map object.

The mapOptions property is the most useful as it allows you to configure the map in a variety of ways before it's created:

var mapDiv = Y.one('#gmap');
mapDiv.on('rdCreate', function(e) {
e.mapOptions["scrollwheel"] = false;
}); The example code shown above refers to a Google Map element with an element ID of "gmap". The code configures the map to disable the mouse scroll wheel (used to zoom in and out) when it's created.

rdCreated

This event fires *after* the Google Map object is created. It has four properties:

- **map** - Gets a reference to the newly-created Google Map object.
- **id** - Gets the ID of the newly-created Google Map object.
- **mapOptions** - Gets the Google Map's MapOptions object properties, ranging from background color to zoom control options. A complete list of these is available from Google on their specifications page.
- **container** - Gets the ID of the internal division created as a container for the Google Map object.

This event and its properties allow you to customize the map after it's been created:

```
var mapDiv = Y.one('#gmap');
mapDiv.on('rdCreated', function(e) {
var latLng = new google.maps.LatLng(38.924622, -77.216779);
var marker = new google.maps.Marker({
position: latLng,
map: e.map
});
```

}); The example code shown above refers to a Google Map element with an element ID of "gmap". It adds a Map Marker at a specific location (Logi World HQ) after the map is created, using the **LatLng** and **Marker** classes from the Google Map API.

```
var latLng = new google.maps.LatLng(@Request.LatHQ~, @Request.LngHQ~);
```

Of course, the code can be made dynamic through the use of tokens, as shown above.

rdGetMapObject()

This Logi function returns a reference to the specified Google Map object and can be used to customize an existing Google Map based on events related to the Logi page, such as a user-initiated mouse click. Imagine a button or link on a page, with Action.JavaScript as its child with this code:

```
var myMap = rdGetGMapObject('gmap');
var randomNumber = Math.random();
var latLng = new google.maps.LatLng(38.924622 + randomNumber, -77.216779 - randomNumber);
var marker = new google.maps.Marker({
position: latLng,
map: myMap
});
```

The example code shown above refers to a Google Map element with an element ID of "gmap". It adds a Map Marker at a random location, using the **LatLng** and **Marker** classes from the Google Map API, when the button or link is clicked.

Example Definition

Here's a complete report definition that includes all of the previous code examples and some embellishments. Download it from DevNet and when it opens in your browser, save it, with an .lgx file extension, to the _Definitions_Reports folder in one of your Logi applications. The application version must be Logi Info v12.0.036 SP2 or later for it to work. After you save the file, in Studio you may need to right-click the Reports folder in the Application panel and select **Refresh Application** in order to see the new definition.

ESRI ArcGIS API

The following topics discuss the integration of ESRI ArcGIS map visualizations into Logi applications, using the ArcGIS JavaScript API:

- Requirements Checklist
- Adding a Simple ArcGIS Map
- Adding a Polygon Overlay
- Embedding Logi Visualizations

About ArcGIS

ESRI ArcGIS is a geographic information system for working with maps and geographic information, similar to Google Maps. When integrated into a Logi application, ArcGIS maps can driven by data from the wide range of datasources supported by Logi Info and contained within the rich Logi interactive framework. Integration is achieved using ESRI's ArcGIS API for JavaScript and Logi script-related elements. Developers intent on doing this integration should be very experienced with JavaScript, JSON, and Logi Info development techniques and should review the ESRI API documentation before proceeding. There are multiple techniques that can be used to include ArcGIS maps in a Logi application and several examples are shown below. The examples use Asynchronous Module Definition and Modern Dojo.

Licensing

The following information is provided as a courtesy by Logi Analytics and may change without notice. It's your responsibility as a developer to understand and comply with the ESRI ArcGIS licensing requirements. Free use of the ArcGIS

API is allowed for development and evaluations, for non-commercial external use, for educational purposes (teaching only), for NGOs and non-profit businesses, and when used in conjunction with other ArcGIS licenses and services. Otherwise, commercial use on a publicly-accessible web site requires you to purchase an ArcGIS license. For complete information about licensing, see the ArcGIS for Developers - Terms of Use page.

Requirements Checklist

Before starting to integrate ArcGIS maps into a Logi application, be sure that:

- You're using Logi Info v12 or later
- You have a proper ArcGIS license, if necessary
- You can connect to the Internet from your development server
- You've examined ESRI's ArcGIS API for JavaScript documentation
- You're familiar with the general steps for creating a new Logi application and using JavaScript

If any of the above items are incomplete or missing, please take care of them before proceeding.

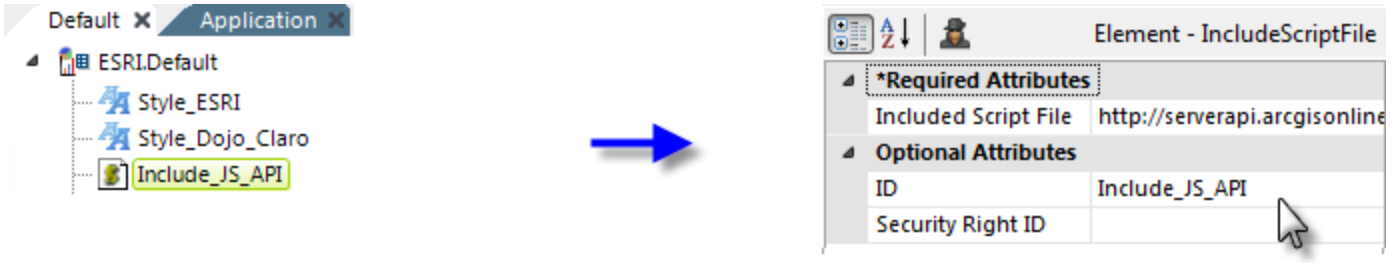
Adding a Simple ArcGIS Map

The following assumes an existing Logi application, with a report definition ready to have ArcGIS maps added to it.

We'll begin by adding two **Style** elements to your report definition, as shown above, and configuring them to access style sheets available from ArcGIS, using API v3.4. The style sheet URLs are:

```
http://serverapi.arcgisonline.com/jsapi/arcgis/3.4/js/esri/css/esri.css
http://serverapi.arcgisonline.com/jsapi/arcgis/3.4/js/dojo/dijit/themes/claro/claro.css
```

The second style sheet is for one of four Dojo Dijit themes available: *claro*, *tundra*, *soria* and *nihilo*. Feel free to try out different themes.

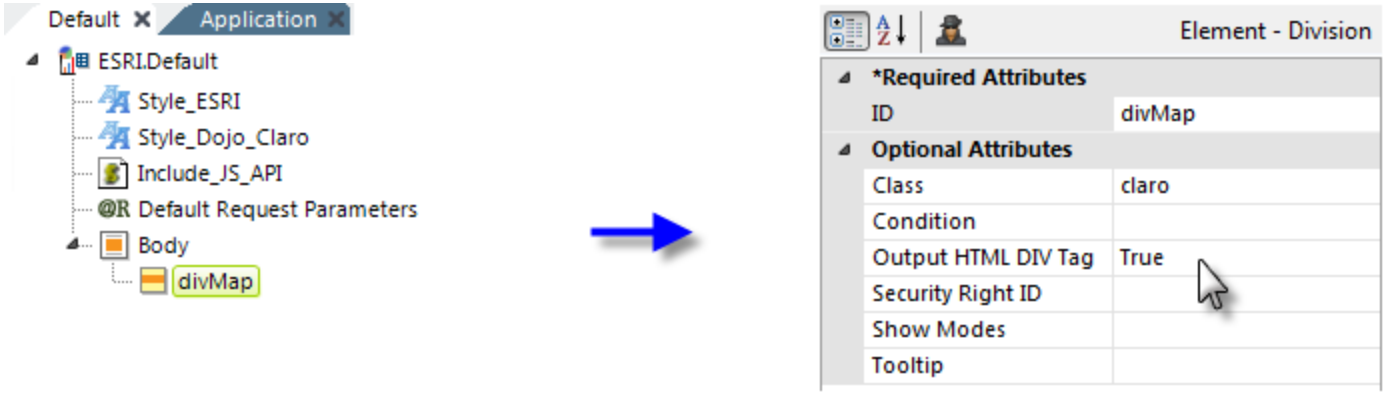


Next, we need to include the ArcGIS API library, using an **Include Script File** element, as shown above. The complete URL for the v3.4 API is:

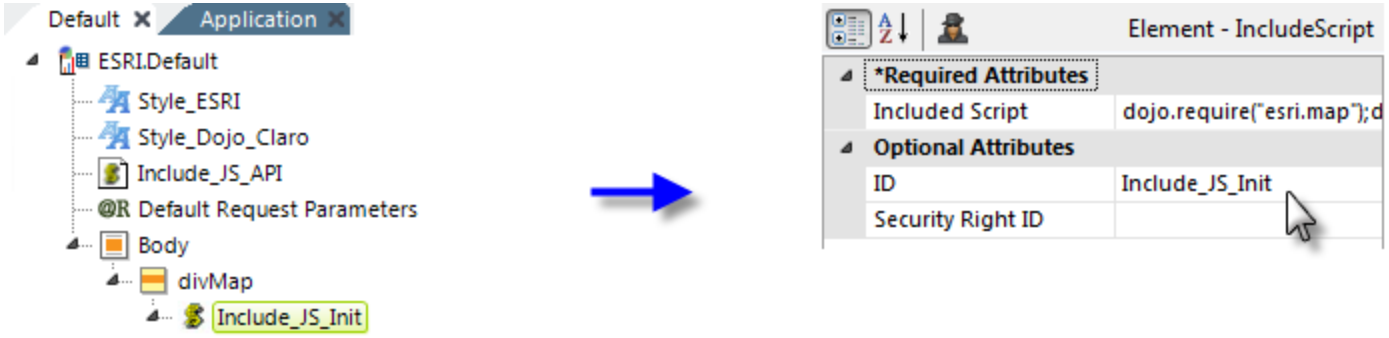
```
http://serverapi.arcgisonline.com/jsapi/arcgis/3.4/
```

This URL doesn't include a filename with a .js extension as you might expect.

Next, we add a **Default Request Parameters** element to provide some default values to control the initial map appearance and specify the icon image that will be used to pinpoint locations on the map. The names of the parameters are arbitrary and will be used later in our JavaScript code.



Now, we add a **Division** element, as shown above, which will be the ArcGIS map container. Its attributes are set to give it a unique ID, to set its Class to one from the Dojo themes, and to ensure that it's rendered as a DIV tag. You can control the size of the map by using CSS to set the height and width of this container element.



Next, we add an **Include Script** element with JavaScript code, using the API, to create and initialize the map. Here's that code:

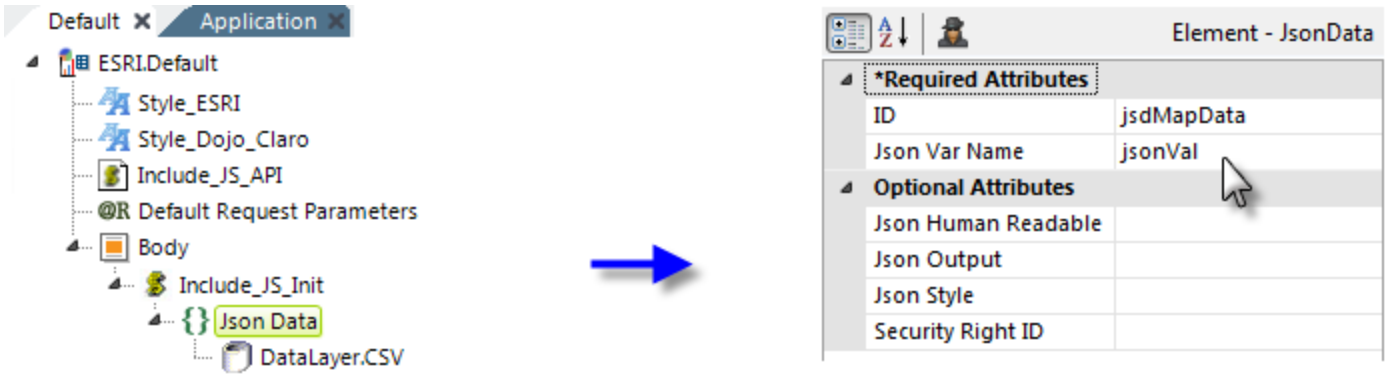
```
dojo.require("esri.map");
dojo.require("esri.dijit.Popup");


 var map;


function init() {
map = new esri.Map("divMap",{
basemap: "topo",
center: [ @Request.longitude~ , @Request.lattitude~ ], //longitude, latitude
zoom: @Request.zoom~,
infoWindow: new esri.dijit.Popup(null, dojo.create("div"))
```

```
});
}
dojo.ready(init);
```

Note these items in the code:

- The "divMap" identifier of our map container Division element is used in the Map constructor.
- The API offers several basemap types, including: *topo*, *streets*, *satellite*, *hybrid*, *national-geographic*, and more.
- @Request tokens, for the default request parameters, are used to center the map and set its initial zoom factor.

If we preview the definition, we'll see the map shown above. The zoom controls and ESRI logo can be controlled using different API options. Now let's add some data-driven points to the map:



In this example, we'll access data in a .csv data file using **Json Data** and **DataLayer.CSV** elements, as shown above. The .csv file data looks like this:

dataTitle,dataDescription,latitude,longitude
Nichols Hills Wellhead,Details of Well #1,35.534741,-97.528243
Del City Wellhead,Details of Well #2,35.434741,-97.428243
Draper Lake Wellhead,Details of Well #3,35.334741,-97.328243

The first row in the .csv file provides the data column names we'll use in our code. Notice that HTML is included right in the data, to provide links. We could, of course, get the data from any of the many other datasources supported by Logi Info. In order to use the JSON data retrieved from the file, we need to add some new JavaScript code in our Include Script element, as follows:

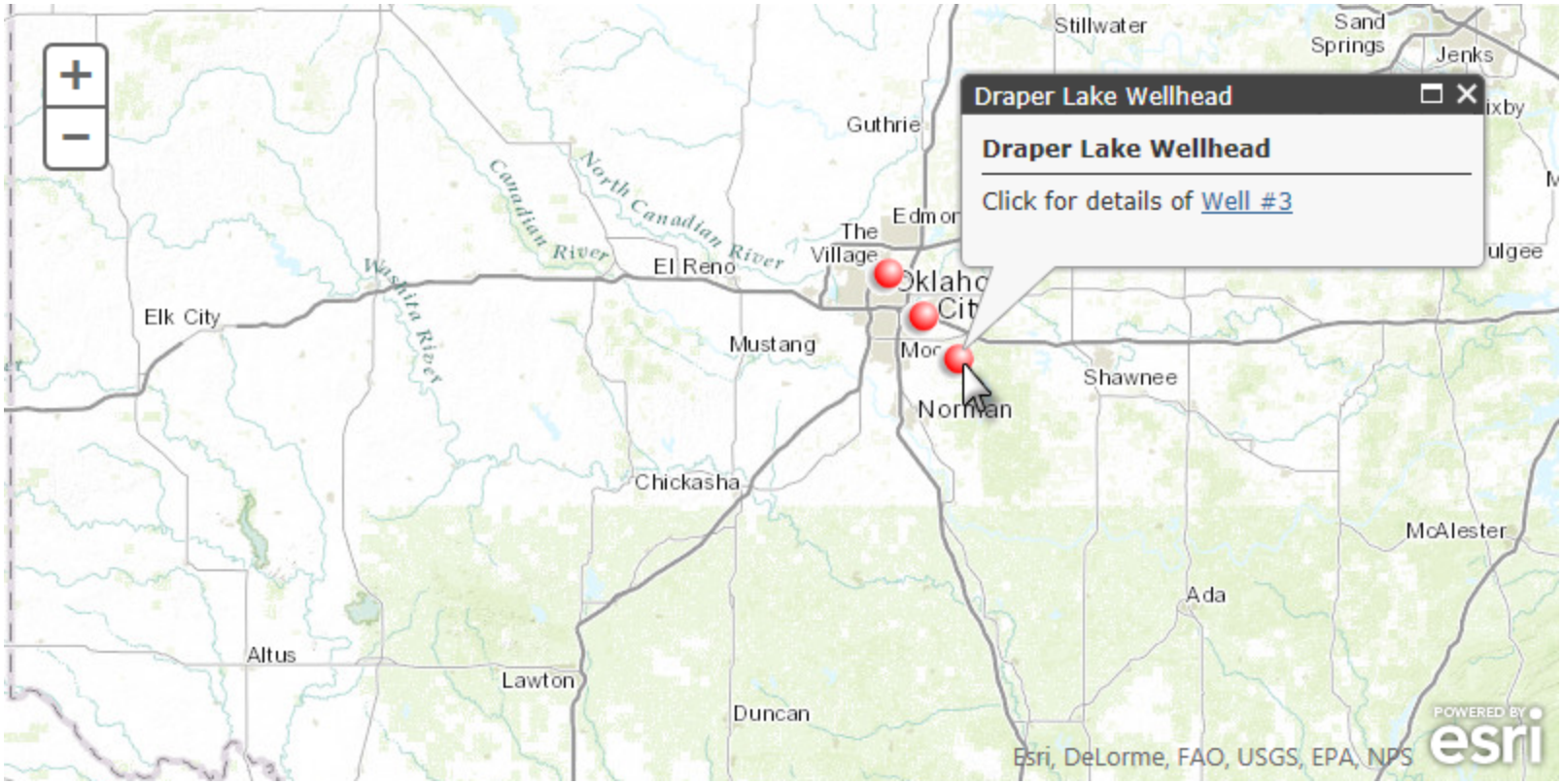
```
dojo.require("esri.map");
dojo.require("esri.dijit.Popup");

 var map;

function init() {
```

```
map = new esri.Map("divMap",{
basemap: "topo",
center: [ @Request.longitude~ , @Request.lattitude~ ], //longitude, latitude
zoom: @Request.zoom~,
infoWindow: new esri.dijit.Popup(null, dojo.create("div"))
});
dojo.connect(map,"onLoad", addWells);
}
function addWells(data){
var symbol = new esri.symbol.PictureMarkerSymbol("@Request.iconFile~", 24, 24);
var template = new esri.dijit.PopupTemplate({
title: "{dataTitle}",
description:"{dataDescription}"
});
for (var w in jsonVal) {
var loc = new esri.geometry.Point(jsonVal[w].longitude, jsonVal[w].latitude);
map.graphics.add(new esri.Graphic(loc, symbol, jsonVal[w], template));
}
}
dojo.ready(init);
```

New code is shown above with a white background. Note these items in the code:

- The @Request token for the default request parameter for the map marker icon image is used.
- Data column names "dataTitle" and "dataDescription" are used to insert the data where needed for the pop-up that appears when a map marker is clicked.

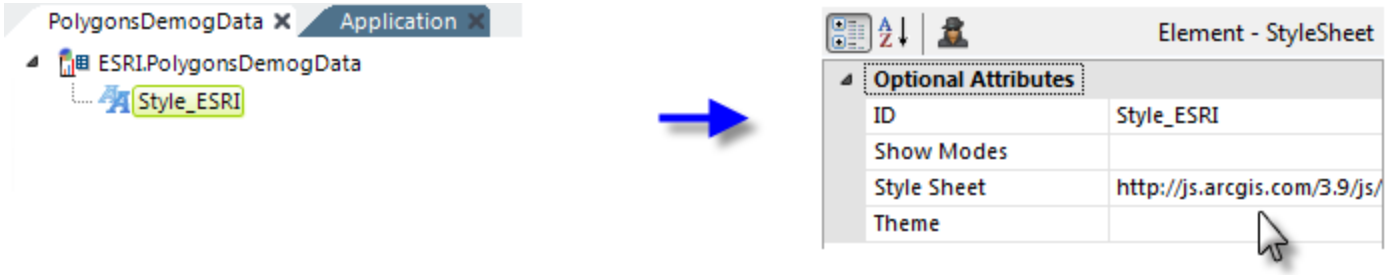
- Latitude and longitude data for positioning the map markers is inserted using the array of values, *jsonVal*, identified in the Json Data element.



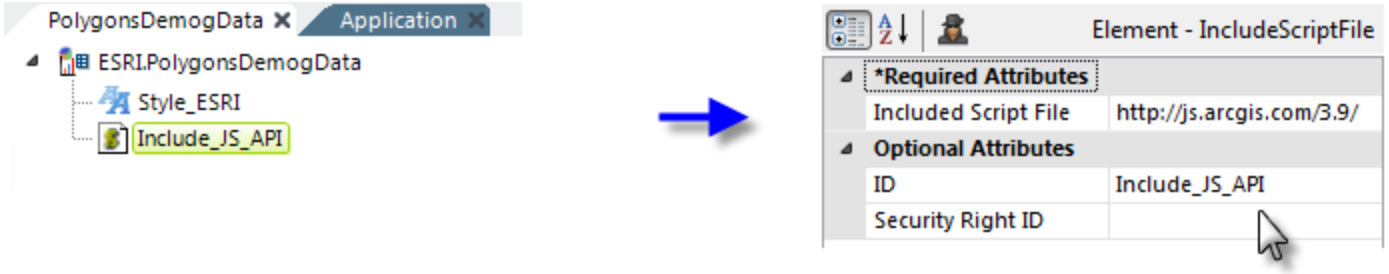
Now when the map is displayed (and we zoom in a couple of clicks), the three map markers from our data file appear. If a marker is clicked, as shown above, a pop-up panel provides the data for that location. You may need to drag the map southward to see the complete pop-up panel. The pop-up panel can be styled and customized using the API.

Adding a Polygon Overlay

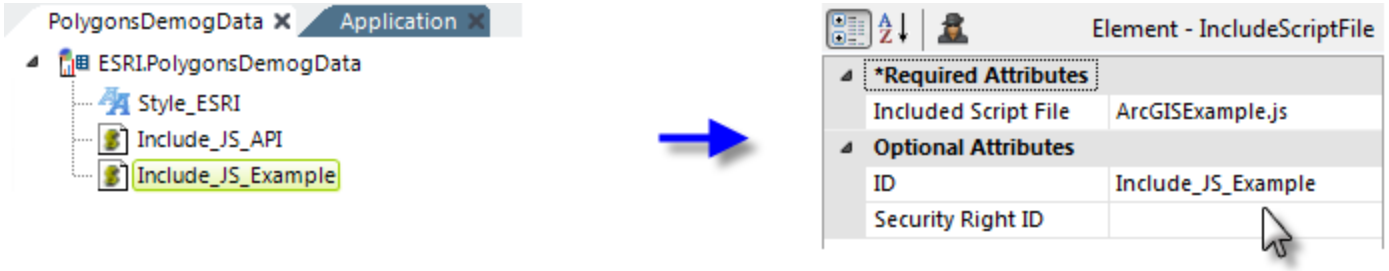
In this example, we'll draw a state map and add an overlay layer of polygons, representing the state's counties and reflecting their population size. This will be driven by U.S. Census data from an online source and we'll use the JavaScript from the ArcGIS Sample web site to render it, using API v3.9.



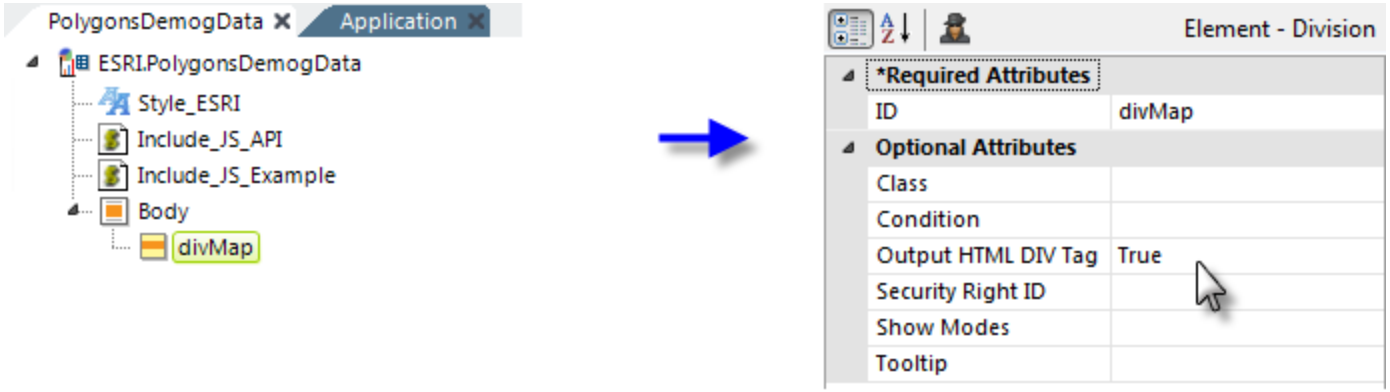
Once again, we'll begin by adding a Style element, as shown above, to a new definition. The style sheet for this example is: `http://js.arcgis.com/3.9/js/esri/css/esri.css`



Next, we need to include the ArcGIS API library, using an **Include Script File** element, as shown above. The complete URL for this is:`http://js.arcgis.com/3.9/`



Next, we add the JavaScript code to create and initialize the map. The code has been taken from the example at https://developers.arcgis.com/javascript/jssamples/renderer_class_breaks.html and saved into a file, `ArcGISExample.js`, in the application's _SupportFiles folder.



And, once again, we add a **Division** element, as shown above, which will be the ArcGIS map container. Its attributes are set to give it a unique ID and to ensure that it's rendered as a DIV tag. You can control the size of the map by using CSS to set the height and width of this container element. Let's have a look at the JavaScript in the `ArcGISExample.js` file:

```
var map;
```

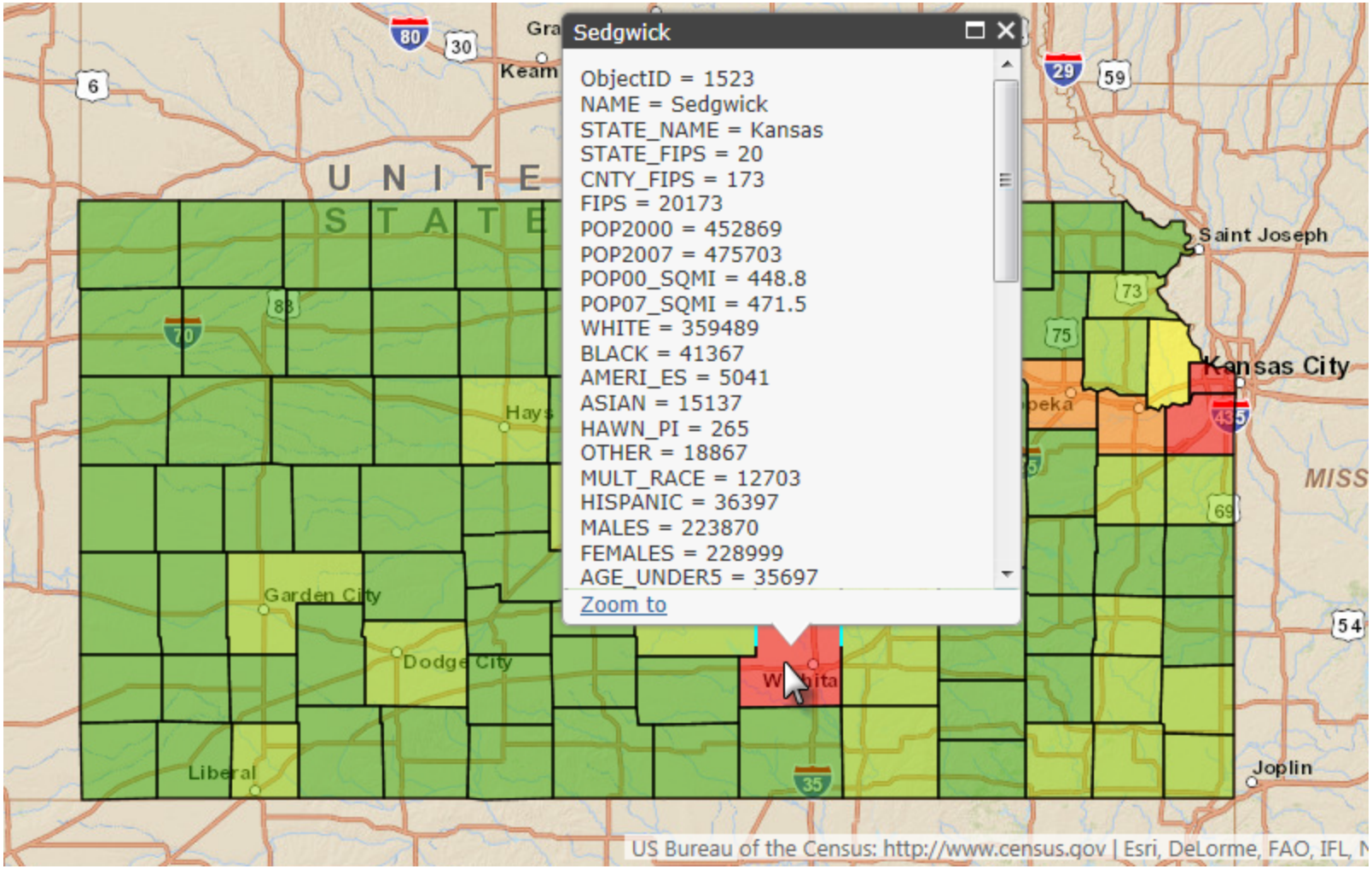
```
require([
"esri/map", "esri/layers/FeatureLayer",
"esri/InfoTemplate", "esri/symbols/SimpleFillSymbol",
"esri/renderers/ClassBreaksRenderer",
"esri/Color", "dojo/dom-style", "dojo/domReady!"
], function(
Map, FeatureLayer,
InfoTemplate, SimpleFillSymbol,
ClassBreaksRenderer,
Color, domStyle
) {

map = new Map("divMap", {
basemap: "streets",
center: [-98.215, 38.382],
zoom: 7,
slider: false
});
var symbol = new SimpleFillSymbol();
symbol.setColor(new Color([150, 150, 150, 0.5]));

// Add five breaks to the renderer
// If you have ESRI's ArcMap available, this can be a good way to determine break values
// You can also copy the RGB values from the color schemes ArcMap applies, or use colors
// from a site like www.colorbrewer.org
//
```

```
// alternatively, ArcGIS Server's generate renderer task could be used
var renderer = new ClassBreaksRenderer(symbol, "POP07_SQMI");
renderer.addBreak(0, 25, new SimpleFillSymbol().setColor(new Color([56, 168, 0, 0.5])));
renderer.addBreak(25, 75, new SimpleFillSymbol().setColor(new Color([139, 209, 0, 0.5])));
renderer.addBreak(75, 175, new SimpleFillSymbol().setColor(new Color([255, 255, 0, 0.5])));
renderer.addBreak(175, 400, new SimpleFillSymbol().setColor(new Color([255, 128, 0, 0.5])));
renderer.addBreak(400, Infinity, new SimpleFillSymbol().setColor(new Color([255, 0, 0, 0.5])));

var infoTemplate = new InfoTemplate("${NAME}", "${*}");
var featureLayer = new FeatureLayer("http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/Demographics/ESRI_
Census_USA/MapServer/3", {
mode: FeatureLayer.MODE_SNAPSHOT,
outFields: ["*"],
infoTemplate: infoTemplate
});

featureLayer.setDefinitionExpression("STATE_NAME = 'Kansas'");
featureLayer.setRenderer(renderer);
map.addLayer(featureLayer);
});
```

Note these items in the code:

- The "divMap" identifier of our map container Division element is used in the Map constructor.
- In this example, values for the map center point and zoom factor are hard-coded, but they could come from tokens, as in the earlier example.

- Renderer "breaks" relate population data to colors in the polygon overlay (the "FeatureLayer").
- Data for the overlay comes from an ArcGIS online sample source.



Now when the map is displayed the counties are shown in a polygon overlay and the polygons are colored based on their pop-

ulation density. If a polygon is clicked, as shown above, a pop-up panel provides the data for that county. You may need to drag the map southward to see the complete pop-up panel. The pop-up panel can be styled and customized using the API.

Embedding Logi Visualizations

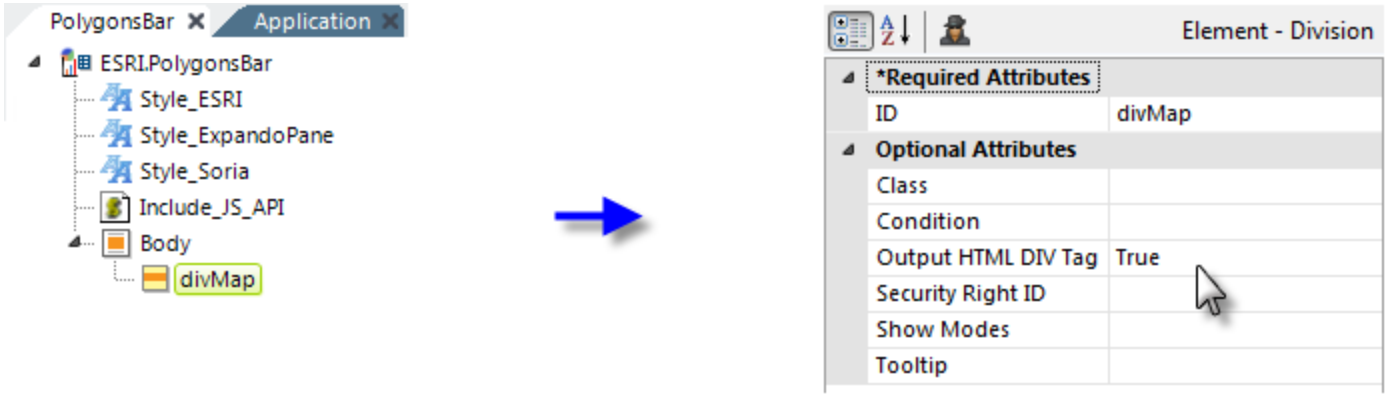
In this final example, we'll draw a state map and add an overlay layer of polygons, representing the state's counties, as before. This time, however, when a county is clicked we'll display a pop-up panel that contains a Logi visualization (a bar chart) displaying the county's population size in two different years. This will be driven again by U.S. Census data from an online source and we'll use the JavaScript from the ArcGIS Sample web site to render it, using API v3.9.

Once again, we'll begin by adding three **Style** elements, as shown above, to a new definition. The style sheets for this example are:

```
http://js.arcgis.com/3.9/js/esri/css/esri.css
http://js.arcgis.com/3.9/js/dojo/dojox/layout/resources/ExpandoPane.css
http://js.arcgis.com/3.9/js/dojo/dijit/themes/soria/soria.css
```

The third style sheet is for one of four Dojo Dijit themes available: *claro*, *tundra*, *soria* and *nihilo*. Feel free to try out different themes.

Next, we need to include the ArcGIS API library, using an **Include Script File** element, as shown above. The complete URL for the v3.9 API is: `http://serverapi.arcgisonline.com/jsapi/arcgis/3.9/`



And, once again, we add a **Division** element, as shown above, which will be the ArcGIS map container. Its attributes are set to give it a unique ID and to ensure that it's rendered as a DIV tag. You can control the size of the map by using CSS to set the height and width of this container element.

Next, using an **Include Script** element, we add the JavaScript code to create and initialize the map. Our code looks like this:

```
var map;

require([
"esri/map",
"esri/InfoTemplate",
"esri/layers/FeatureLayer",
"esri/renderers/SimpleRenderer",
"esri/symbols/SimpleFillSymbol",
"esri/symbols/SimpleLineSymbol",
"dojo/dom",
"dojo/number",
"dojo/on",
"dojo/parser",
"esri/Color",
"dijit/layout/BorderContainer",
"dijit/layout/ContentPane",
```

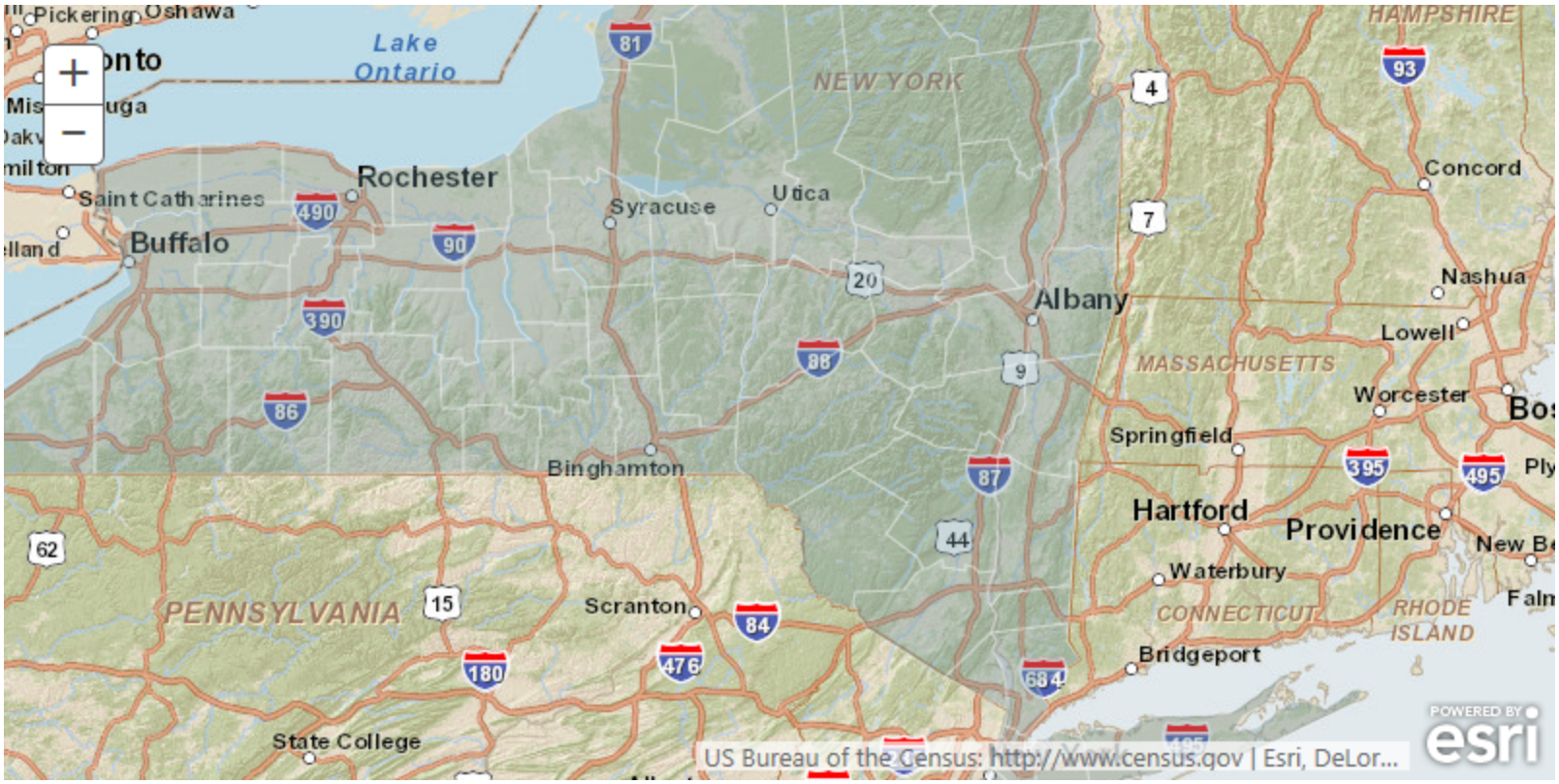
```
"dojox/layout/ExpandoPane",
"dojo/domReady!"
],

function (
Map, InfoTemplate, FeatureLayer, SimpleRenderer, SimpleFillSymbol,
SimpleLineSymbol, dom, number, on, parser, Color
) {

parser.parse();

map = new Map("divMap", {
basemap: "streets",
center: [-75.18, 42.34],
zoom: 7
});

var infoTemplate = new InfoTemplate();
infoTemplate.setTitle("County Census Data");
infoTemplate.setContent("<iframe src-
c=\"http://@Function.ServerName~/@Function.AppVirtualPath~?rdReport=CountyChart&County=${NAME}, ${STATE_
NAME}&pop2000=${POP2000}&pop2007=${POP2007}\" frameborder=0 width=250 height=290></iframe>");

var counties = new FeatureLayer("http://sampleserver1.arcgisonline.com/ArcGIS/rest///services/Demographics/ESRI_Census_
USA/MapServer/3", {
mode: FeatureLayer.MODE_SNAPSHOT,
```

```
infoTemplate: infoTemplate,
outFields: [
"NAME", ,"STATE_NAME", "POP2000", "POP2007", "POP00_SQMI",
"POP07_SQMI"
]
});

counties.setDefinitionExpression("STATE_NAME = 'New York'");

//apply a renderer
var symbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
new Color([255, 255, 255, 0.35]), 1),
new Color([109, 146, 155, 0.35]));
counties.setRenderer(new SimpleRenderer(symbol));

map.addLayer(counties);

});
```

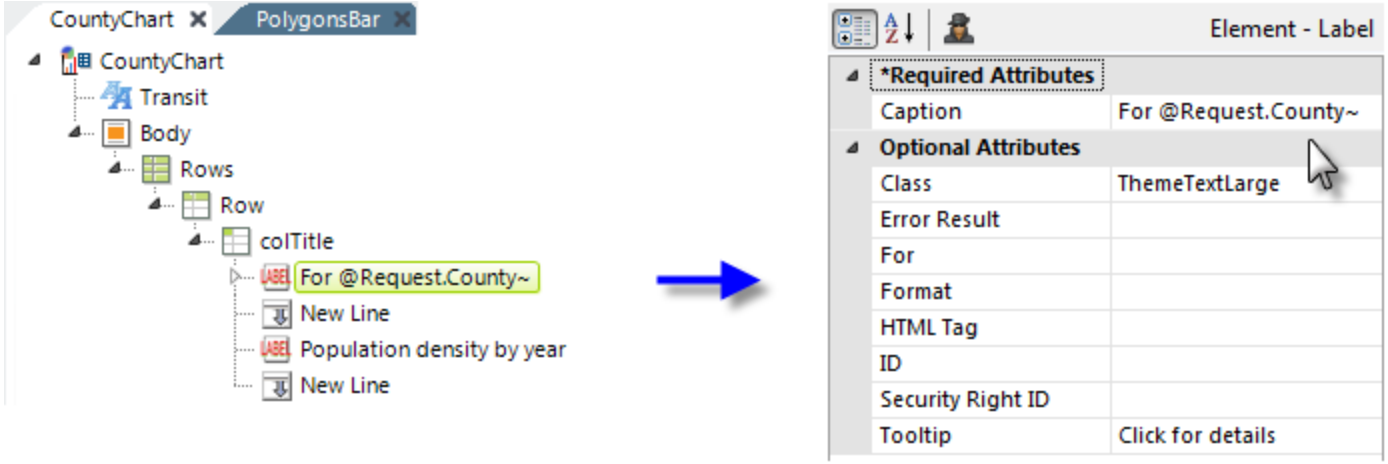
Note these items in the code:

- The "divMap" identifier of our map container Division element is used in the Map constructor.
- In this example, values for the map center point and zoom factor are hard-coded, but they could come from tokens, as in the earlier example.

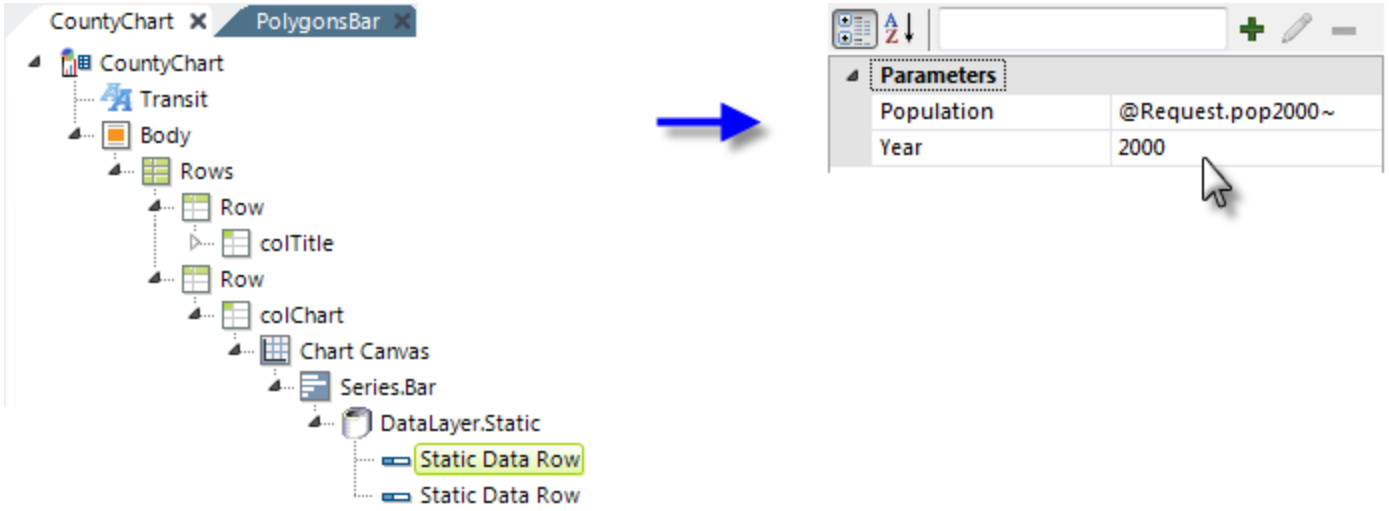
- An "InfoTemplate" is created for the pop-up panel that appears in response to clicks on the polygons. It includes the HTML to embed a different Logi report in an iFrame.
- The polygon overlay (the "FeatureLayer") reads data from an ArcGIS online sample source. Data retrieved is passed to the Logi report in the Info Template as a request variables in iFrame URL.
- A "SimpleFillSymbol" object is used to change a polygon's outline color when it's clicked.



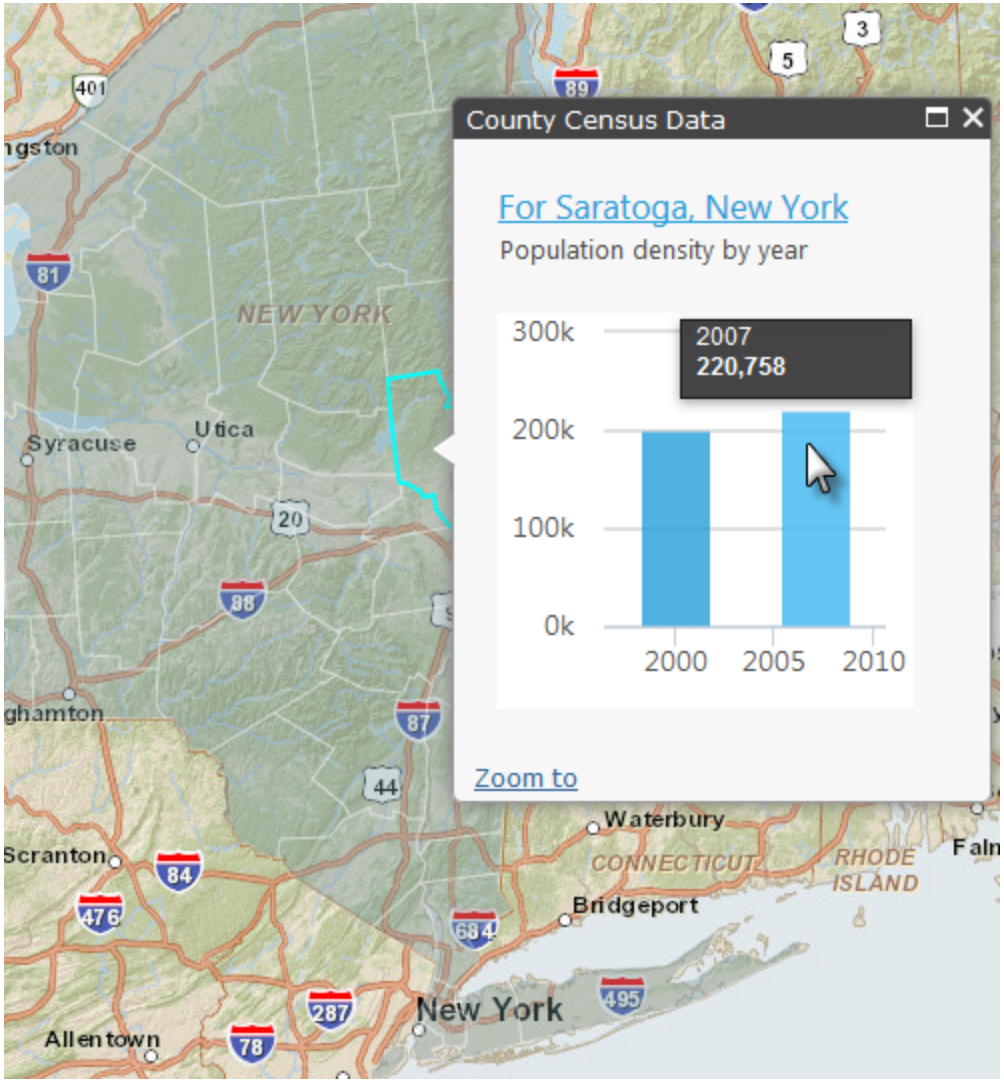
If we preview the definition, we'll see the map shown above. The zoom controls and ESRI logo can be controlled using different API options. Now let's add the separate Logi report definition that's going to appear in an iFrame in the pop-up panel.



Create a new definition, give it the name "CountyChart" (which matches the iFrame source URL in the JavaScript code above) and add some element for style, structure, and information, as shown above. The **Label** element shown above uses one of the request variables passed in from the iFrame URL and it also has a Action.Link element (not shown) which uses the same request token to create a link to search for the county in Google.



And, finally, we add a set of elements for a Chart Canvas Chart, with a Bar series, as shown above. We have a fixed number (two) years of data, so we can use a DataLayer.Static element to receive the request variable data passed in the iFrame URL and place it in a datalayer to be used by the chart.



Now when the map is displayed the counties are shown in a polygon overlay, as before, and if a polygon is clicked, as shown above, a pop-up panel provides the data for that county, as a chart. The title is shown as a live link and all of the Chart Canvas

Chart features, including hover-highlighting and Quicktips are present. We've seen how easy it is include ArcGIS maps into a Logi application, using static and online data. You can, of course, use any of the datasources Logi Info can access to produce JSON data to be consumed by a map.

The Analysis Grid for Developers

The Analysis Grid is a dynamic analysis tool that allows users to manipulate data, create charts, change table layouts, and much more, at runtime. Like all super-elements, the Analysis Grid has its own built-in user interface and functionality.

The following topics introduce Logi Info *developers* to one of the "super-elements", the Analysis Grid:

- Analysis Grid Element Family
- Analysis Grid Attributes
- Retrieving and Using Data
- Special Formatting
- Controlling Column Availability
- Changing Column Order and Width
- Setting Filtering Pop-up Values
- Using a More Info Row
- Controlling Feature Access
- Setting Pagination Options
- Exporting Data
- Configuring Add-to-Dashboard
- Configuring for Report Author Galleries
- Saving and Loading Analysis Grids
- Special Query String Parameters
- Customizing Analysis Grid Appearance

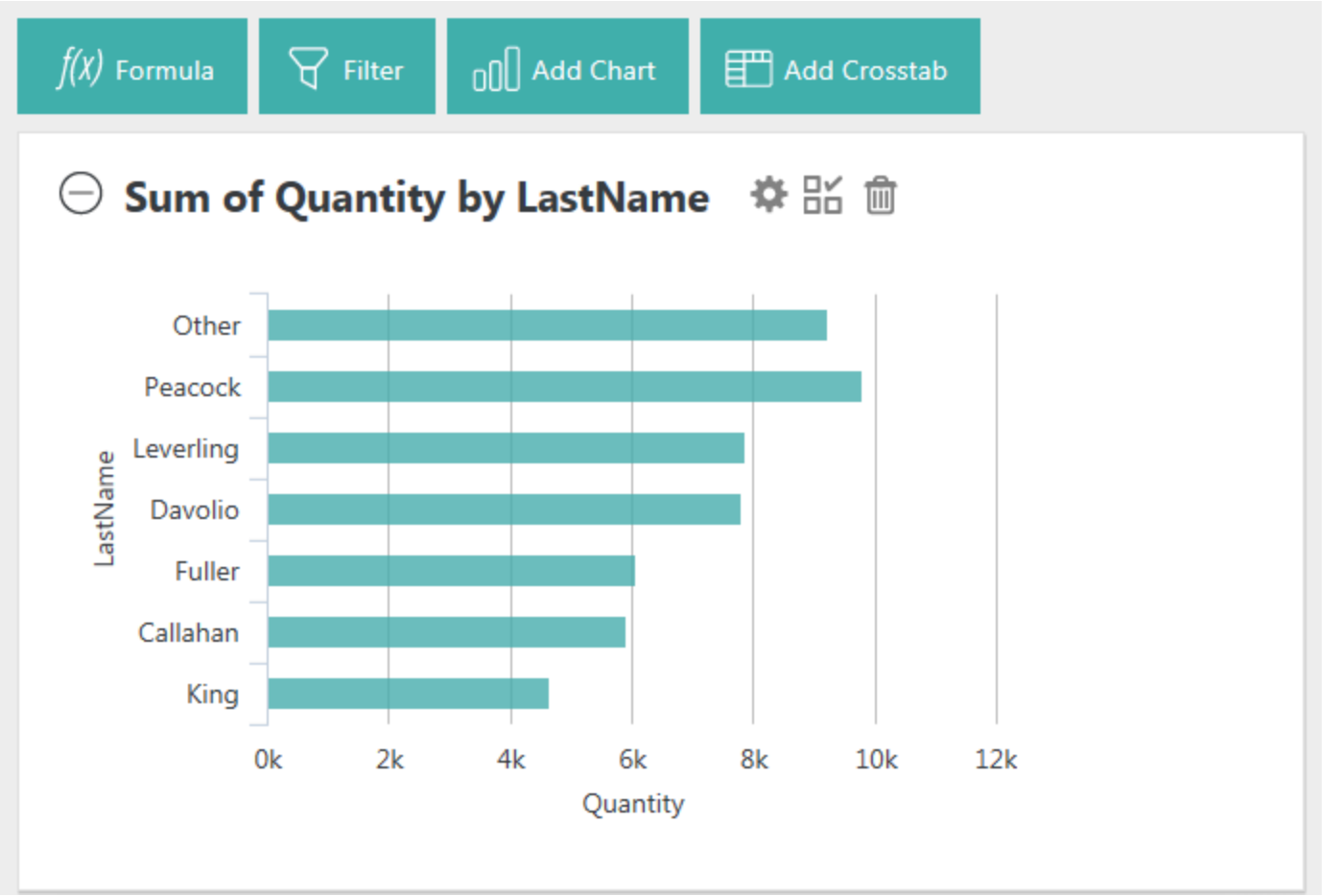
For information about *using* an Analysis Grid, intended for *end-users*, see *The Analysis Grid for End Users*.

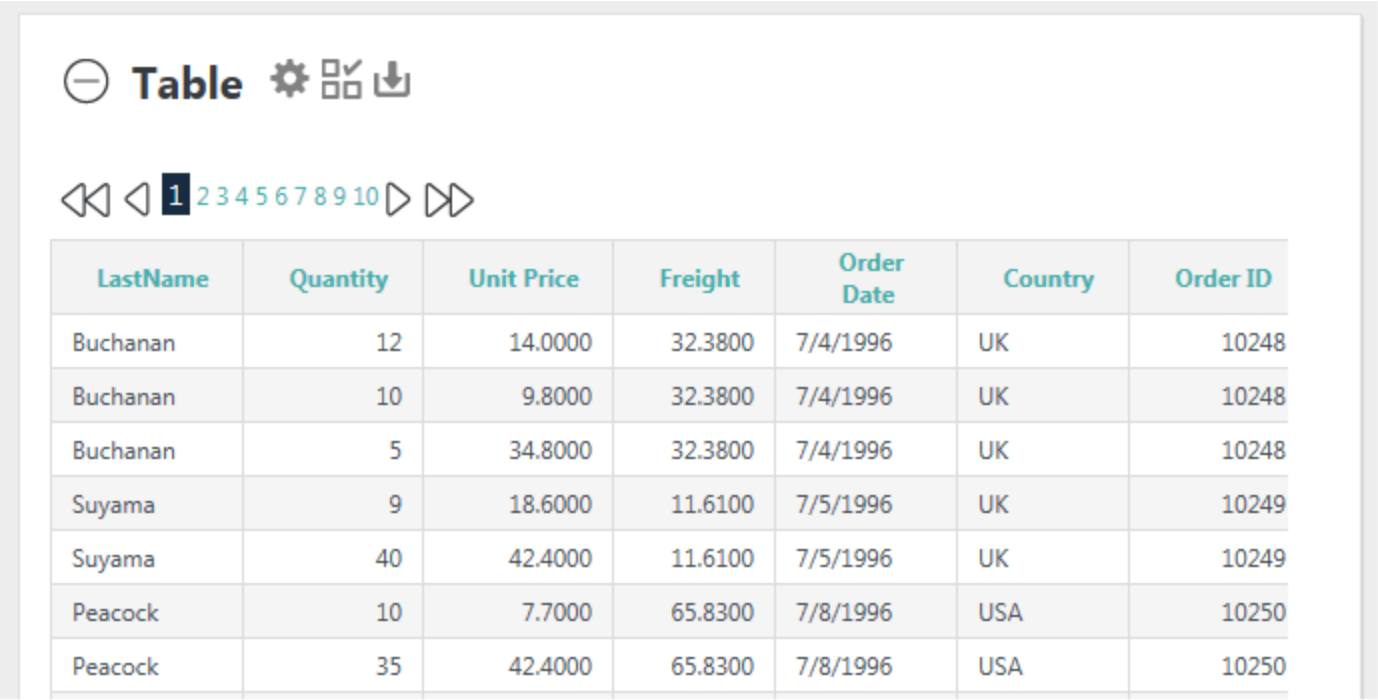
About the Analysis Grid

The Analysis Grid element provides a complete package of data analysis capabilities in a single element, letting developers provide their users with a lot of functionality, while saving development time and effort.

The Analysis Grid UI consists of separate panels for controls, tables, and visualizations. At runtime, users can manipulate the controls, creating data analyses and visualizations on the fly. Logi Info provides developers complete control over the content and layout of Analysis Grids, and the ability to restrict the UI controls available to users.

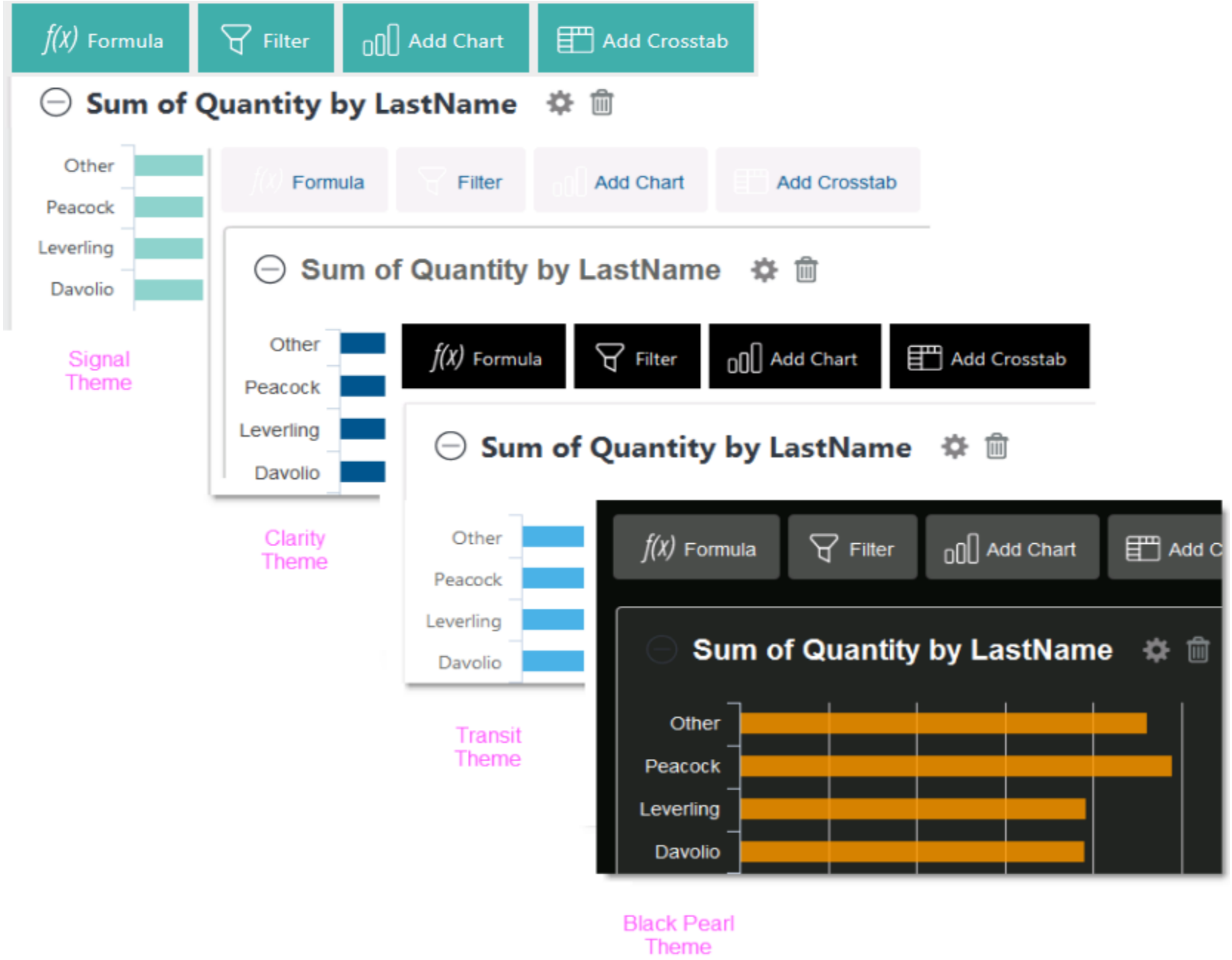
Here's a quick overview of the features and controls found in an Analysis Grid:


Formula
Filter
Add Chart
Add Crosstab
Sum of Quantity by LastName
Other
Peacock
Leverling
Davolio
Fuller
Callahan
King
LastName
0k
2k
4k
6k
8k
10k
12k
Quantity

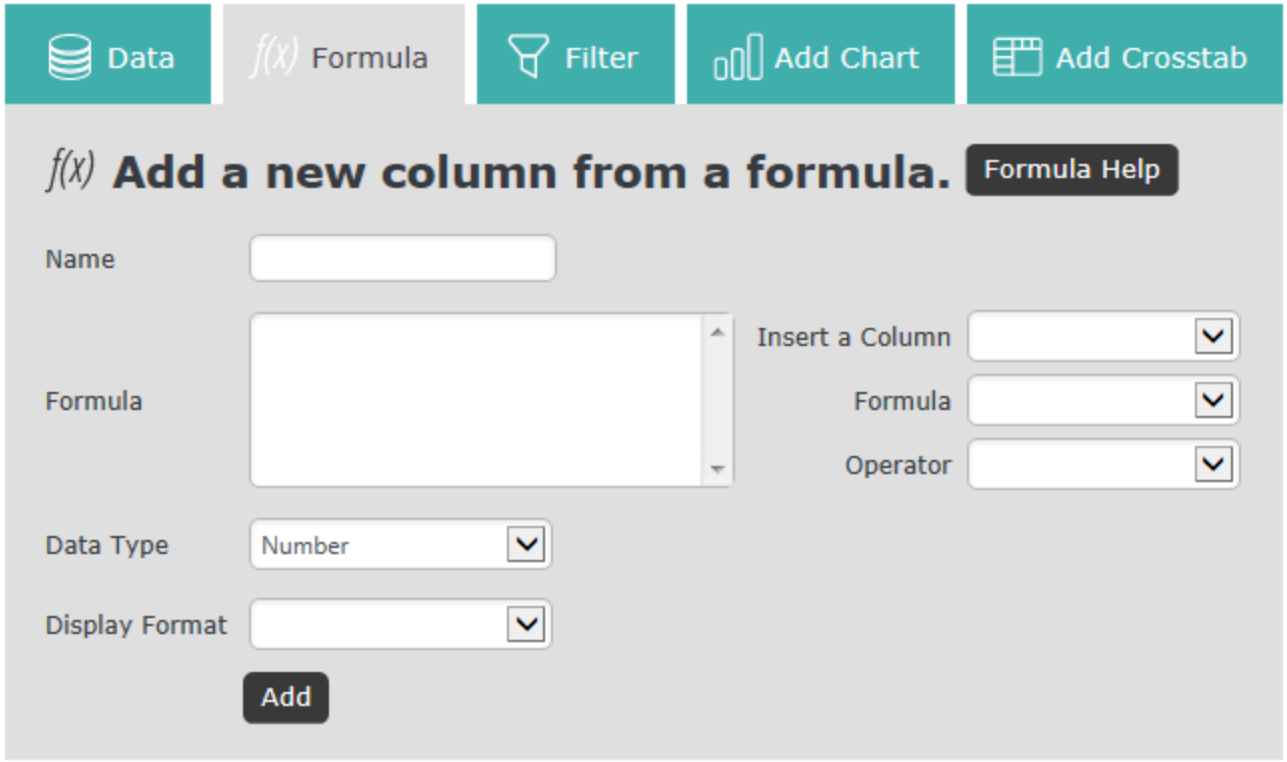


LastName	Quantity	Unit Price	Freight	Order Date	Country	Order ID
Buchanan	12	14.0000	32.3800	7/4/1996	UK	10248
Buchanan	10	9.8000	32.3800	7/4/1996	UK	10248
Buchanan	5	34.8000	32.3800	7/4/1996	UK	10248
Suyama	9	18.6000	11.6100	7/5/1996	UK	10249
Suyama	40	42.4000	11.6100	7/5/1996	UK	10249
Peacock	10	7.7000	65.8300	7/8/1996	USA	10250
Peacock	35	42.4000	65.8300	7/8/1996	USA	10250

In the example Analysis Grid shown above, both a table and a chart are displayed. The *Signal* theme is being used.

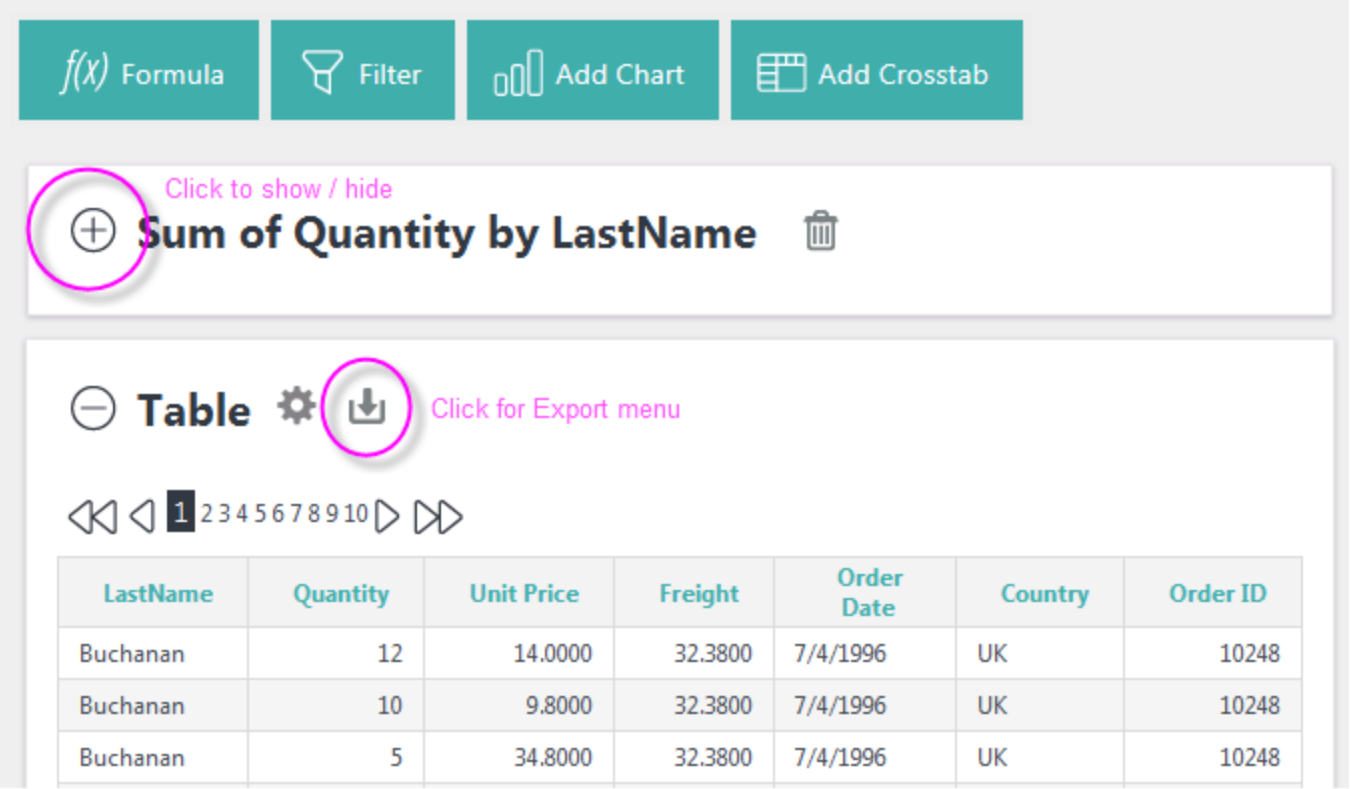

Formula
Filter
Add Chart
Add Crosstab
Sum of Quantity by LastName
Other
Peacock
Leverling
Davolio
Signal Theme
Clarity Theme
Transit Theme
Black Pearl Theme

As you can see in the examples above, the Analysis Grid can look quite different depending on the theme being used.



The Analysis Grid consists of a set of panels. The **Controls Panel**, at the top, includes a set of tabs or buttons that allow the user to manipulate the data and visualizations at runtime. The visibility of the tabs or buttons is configured by the developer and they can be individually shown or hidden.

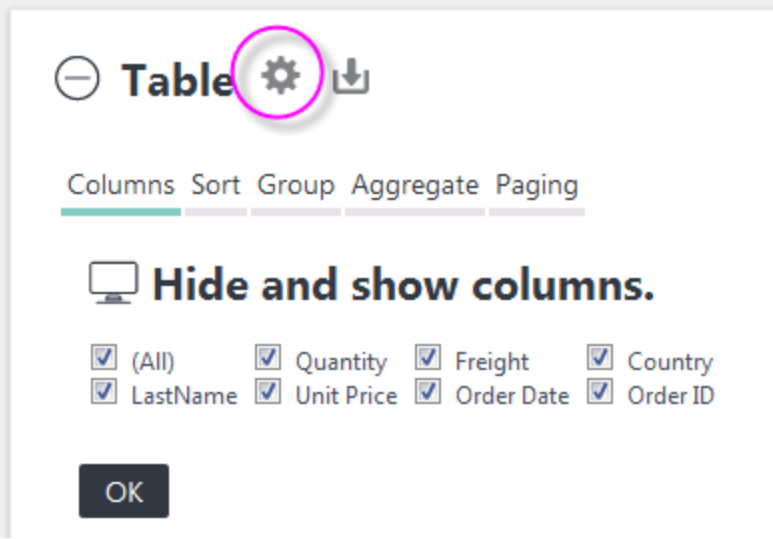
When a tab or button is clicked, a configuration panel for that feature appears, as shown above. Clicking it again hides the panel.



LastName	Quantity	Unit Price	Freight	Order Date	Country	Order ID
Buchanan	12	14.0000	32.3800	7/4/1996	UK	10248
Buchanan	10	9.8000	32.3800	7/4/1996	UK	10248
Buchanan	5	34.8000	32.3800	7/4/1996	UK	10248

Visualization Panels contain either a Data Table, Crosstab Table, or chart. Panels can be collapsed or expanded using the "+" and "-" icons. Panels can also be rearranged by dragging their title bar area to a new position.

Panels containing tables may display an icon for exporting them to Excel, CSV, or PDF. The availability of different types of exports is under developer control.



Clicking the **gear** icon will display a configuration area for the specific visualization. The Table configuration area, shown above, lets users control a variety of table features.

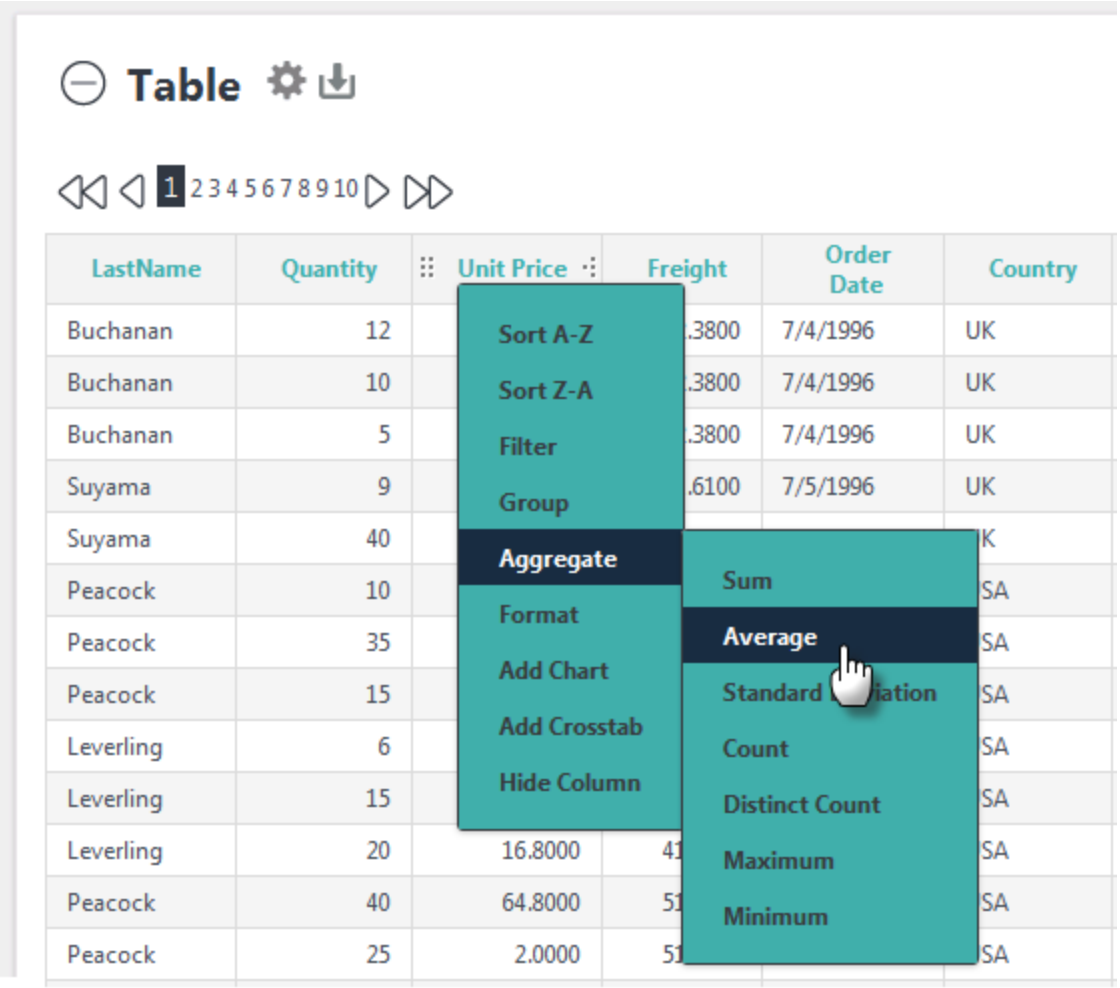


Table column *headers* contain a variety of features. They can be *dragged* to rearrange the column order or to adjust column widths.

When the column header text is clicked, drop-down menus provide sorting, filter, formatting, grouping, and other features, as shown above. The availability of each feature can be restricted during development.

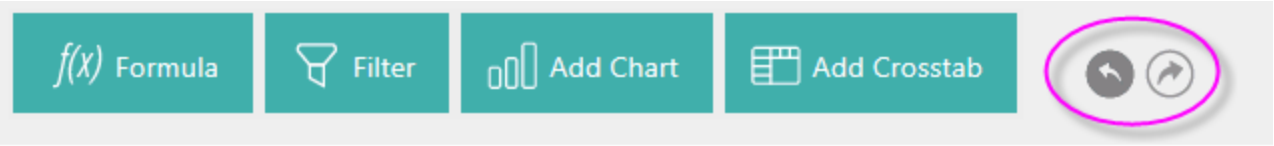
SubReport Restriction

Do not use a **SubReport** element configured for *Embedded* mode to embed report definitions containing an Analysis Grid. Internal component naming and session variable conflicts will arise and the super-element will not work correctly. You must use *IncludeFrame* mode instead.

Retaining User Settings

With all its configuration possibilities, it may be important for users to be able to **save** their Analysis Grid **settings** between sessions. Traditionally, this has been done using a Process Task, saving user settings on the web server and making them available during the next session (this is discussed in more detail in "Analysis Grid Developers - Saving and Loading Analysis Grids" on page 342).

The **Auto Bookmark** element will automatically save all changes made by the user in bookmarks, which can be used later to reproduce the report with settings intact. For more information, see *Bookmarks*.



When the Auto Bookmark element is used, Undo and Redo icons, circled above, will appear. These allow users to undo and redo their recent actions.

Passing Parameters

Do not send critical information to a page for use with an Analysis Grid via request parameters such as LinkParams. When an Analysis Grid's configuration is changed by the user at runtime and the element refreshes itself, request parameters may become unreliable. We recommend that any information that needs to be passed to a page for use with an Analysis Grid be sent using Session variables.

Refreshing using AJAX

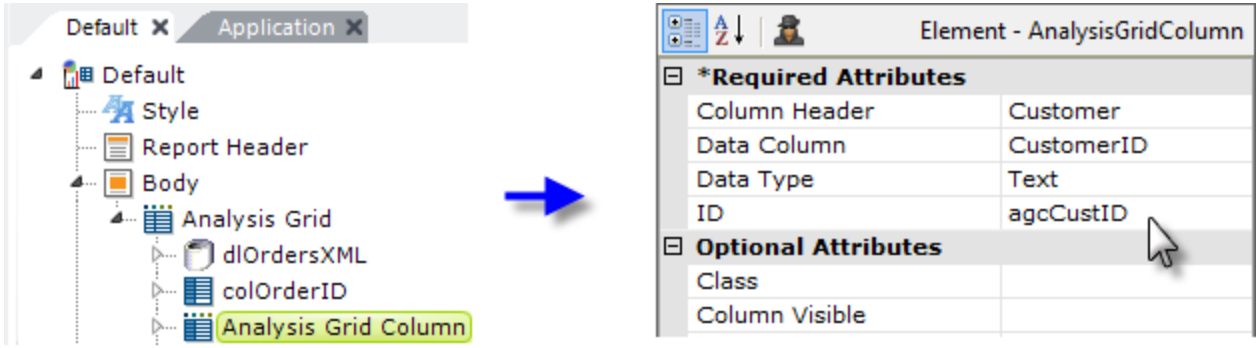
Action.Refresh Element uses AJAX technology to refresh portions of a report page, which produces a smooth update. However, we *do not* recommend that you use this element to refresh a **Division** which contains a "super-element", such as an **Analysis Grid**. Because the super-elements incorporate AJAX themselves, unexpected results can occur when AJAX calls are "nested" in this scenario.

Analysis Grid Developers - Analysis Grid Element Family

You can always manually add an Analysis Grid (AG) element to your definition. However, Logi Studio includes an "Analysis Grid Wizard" that can make the process easier. Select the Body or similar container element in the element tree, click the main menu's Wizards tab, and click the Analysis Grid item. Then follow the prompts.

- An **Analysis Grid** element. Only one may be used per report definition and it should not be used within Dashboards or Tabbed Panels.
- A **DataLayer** or **Active Query Builder** element, to retrieve the data used in tables and charts.
- One or more **Analysis Grid Column** elements, to display data that *will* be available for analysis.
- One or more **Data Table Column** elements, to display data that *willnot* be available for further analysis.
- An optional **Crosstab Comparison** element which displays column value differences in Crosstab Tables.

The example above shows a typical set of Analysis Grid elements. A few other special purpose child elements are also available.



Why two kinds of table columns? **Analysis Grid Column** elements are used to display data and work with the grid's analysis features. However, your ability to customize them is limited. **Data Table Column** elements, on the other hand, are also available to

display data *and* they allow customization, but their data can't be used for analysis. Using the two types of column elements together in an AG is common practice.

To add an **Analysis Grid Column**:

1. As shown above, select the parent **Analysis Grid** element and add an **Analysis Grid Column** element beneath it.
2. Set the **Column Header** and **ID** attributes. The Column Header value is the text that appears above each column.
3. Set the **Data Column** attribute to the name of a column returned by the data layer. Do not use tokens when the attribute name includes the word "column"; just provide the actual column name.
4. Set the **Data Type** attribute to Text, Number, or Date. Tokens can be used here to set the data type.

Dynamic Column Visibility

When a regular Data Table Column element is used in an Analysis Grid, its visibility can be controlled dynamically by setting its **Condition** or **Show Modes** attributes. However, the Analysis Grid Column element doesn't have those attributes, so its visibility can't be controlled dynamically. Its visibility can only be controlled with Security Rights when using Logi Security.

Analysis Grid Developers - Analysis Grid Attributes

The Analysis Grid element has the following attributes:

Attribute	Description
ID	(Required) Specifies a unique element ID.
AJAX Paging and Sorting	Enables AJAX-style table paging and sorting. When the user moves to another table page, or sorts a column, only the table is updated. This method prevents the web page from flickering or losing the scroll position. This alters the behavior of the browser's "Back" button because the page history does not get updated with AJAX Paging. Default value: *False*.
Allow Pause Data Retrieval	Specifies whether or not to display an extra button on the Analysis Grid which, when clicked, temporarily halts data retrieval from the server. This feature is useful when data retrieval is slow and the user prefers to make several changes to the Analysis Grid without having to wait for data after each change. With data retrieval paused, the user can make all changes, then click the **Resume** button to resume retrieval. Default value: *False*. When set to *True*, you can also automatically apply Pause Data Retrieval at the application level using the constant `rdAgDataRetrievalPaused`.
Allow Zoom Control	Specifies whether chart zoom is enabled/disabled for Bar Charts, Line Charts, Curve Lines, and Scatter Plots. Default value: *False*.
Alternating Row Class	Specifies a CSS class to be used for every other row of the table. If you're using a Theme, the class *ThemeAlternatingRow* can be used.
Batch Selection	Specifies whether Analysis Grid charts and tables are updated as the user makes individual interface

Attribute	Description
	choices. When *True*, allows the user to select a number of options and then click an OK button to show the selections. This can improve performance by reducing the number of server requests. Users may need to click the OK button when the page is first displayed in order to get charts to render (even if no changes are made). Default value: *False*.
Caption	Specifies the title that will appear above the Analysis Grid.
Class	Specifies the CSS class to be used by the element. When set, this class will also be used by all child elements that don't have their own class. Generally not used if a Theme is being used.
Current Page Class	When paging is being used for a table, specifies the CSS class to be used by the "current page number" indicator.
Disable Replace Confirmation	Specifies whether pop-up confirmation prompts will appear when replacing Formula Columns, Filters, and Sort Orders. Set to *True* to disable the prompts. Default value: *False*.
Disable Show All Rows	Specifies whether the "Show all rows" option will appear in the Paging controls at runtime. If your users work with very large datasets, you may not want them to be able to use this option. Set to *True* to hide the option. Default value: *False*.
Draggable Panels	Specifies whether the panels that visualizations appear in can be rearranged at runtime by dragging them. Default value: *True*.
Exclude Details	When rows are grouped within the Analysis Grid table, specifies whether detail rows are shown. In the

Attribute	Description
Rows Default	table's configuration area, a check box allows excluding detail rows. Setting this attribute to *True* causes detail rows to be initially excluded (hidden). In some cases, especially when working with very large data-bases, performance may be better without detail rows. Default value: *False*.
Filter Case Sensitivity	Specifies if runtime Filters will be case-sensitive or case-insensitive. If set to *Insensitive*, values are com-pared without regard to upper and lower case. When set to *Sensitive*, case must match exactly. Default value: *blank*. In this case, the "Starts With" and "Contains" comparisons are case-insensitive, and the other comparisons are case-sensitive. This attribute is ignored when using DataLayer.ActiveSQL with a SQL Server database and filters are always treated as *Insensitive*.
Hide Aggreg-ates	Specifies whether the Aggregate feature will be hidden from view in the Table configuration and all column header drop-down options. Default value: *False*.
Hide Charts	Specifies whether the Chart tab or button in the Controls panel and all Table column header drop-down options will be hidden from view. Default value: *False*.
Hide Crosstabs	Specifies whether the Crosstab tab or button in the Controls panel and all Table column header drop-down options will be hidden from view. Default value: *False*.
Hide Exports	Specifies whether or not the Export icon will be hidden from view. The default value is *False*.
Hide Filters	Specifies whether the Filter tab or button in the Controls panel and all Table column header drop-down options will be hidden from view. Default value: *False*.

Attribute	Description
Hide For-matting	Specifies whether the Format option in all Table column drop-down options lists will be hidden from view. Default value: *False*.
Hide Formulas	Specifies whether the Formula tab or button in the Controls panel will be hidden from view. Default value: *False*.
Hide Function Names Default	When using aggregations, specifies whether the names of the aggregating functions are displayed with the aggregated values in the table summary row.
Hide Grouping	Specifies whether the Group tab or button in the Controls panel and all Table column header drop-down options will be hidden from view. Default value: *False*.
Hide Header If No Data	Controls the ability to hide the Data Table/Crosstab Table header when there is no data to show. Default value: *False*. To hide the header when there is no data, set this constant to *True*.
Hide Layout	Specifies whether the Columns feature in the Table configuration area will be hidden from view. Default value: *False*.
Hide Menu	Specifies whether the Controls panel itself will be hidden from view. Default value: *False*.
Hide Paging	Specifies whether the Paging tab or button in the Table configuration area will be hidden from view. Default value: *False*.
Hide Sort Order	Specifies whether the Sort feature in the Table configuration area and al Table column header drop-down options will be hidden from view. Default value: *False*.

Attribute	Description
Max Rows Export	Specifies a limit for the number of rows that will be exported and whether exporting is enabled. When working with very large data sets, exports may not be practical. If standard datalayers are being used, exports will only be enabled if the number of rows in the Analysis Grid's table does not exceed this limit. Default value: *100,000* rows. If the special value -1 is entered, the default maximum limit is removed, allowing *allrows* to be exported. If DataLayer.ActiveSQL or the Active Query Builder is being used, exports will not be disabled because the row count cannot always be known.
No Data Caption	Specifies the message that displays in the Data Table/Crosstab Table when there is no data to display. Default value: *None*, no message displays. Enter a custom message as the value for this constant, or, enter 'Default' and users will receive the message "No data to display".
Numbered Page Count	Specifies how many number links will appear as paging controls when the Show Page Number attribute is set to *Numbered*. Default value: *10*.
Page Row Count	Specifies how many rows of data will be shown in each "page" of the table.
Saved Analysis Grid Folder	Specifies the name of a folder where users' Analysis Grid options can be saved using the Procedure.Save Analysis Grid element. See "Analysis Grid Developers - Customizing Analysis Grid Appearance" on page 348.
Security Right	When using Logi Security, specifies one or more Security Right IDs, separated by commas. Users with these

Attribute	Description
ID	Security Right IDs will be able to view the Analysis Grid, users without them will not.
Show Page Number	Specifies whether, and in what format, paging controls will appear. Options include: *True* - Displays page number in the format "Page 1 of N". Users can navigate pages by clicking icons or by typing in a page number. *Numbered* - (default) Displays a series of page number links in the format "1 2 3 N". The number of links is set using the Numbered Page Count attribute. The Current Page Class attribute can be used to emphasize the current page number. *False* - No paging controls will be shown.
Sort Arrows	Specifies whether sort direction indicators (arrows) are displayed when the user clicks a column heading to sort the table. Default value: *False*.
Template Modifier File	Specifies the name, with file extension, of a "Template Modifier Files" on page 426 to be applied to the Analysis Grid at runtime. This is an XML file that can be used to change the underlying template used to create the Analysis Grid. Template Modifier files can be in any folder accessible to the application; if a folder isn't specified, the _SupportFiles folder is assumed.
Width	Specifies the width of the Analysis Grid, in units set in the Width Scale attribute.
Width Scale	Specifies the width units of the Width attribute, either *px* for pixels or *%* for percent.

Analysis Grid Developers - Retrieving and Using Data

The Analysis Grid works with any standard datalayer to retrieve data, which is then cached *in session*. This allows changes the user makes using the AG's user interface to change the displayed data very rapidly.

Resetting Cached Data

If, after initially retrieving and viewing data in the AG, you then change the data retrieved (for instance, by changing your SQL query) and you do not end your session by completely closing and re-opening your browser, you will see no data differences, due to the session-based caching.

To reset the UI selections or the data without ending the session, you need to refresh the page and pass the special query string parameters, `rdAgReset` and/or `rdAgRefreshData` with a *True* value. These are discussed in "Analysis Grid Developers - Special Query String Parameters" on page 346, and here's an example of one of them in use:

```
yourLogiApp/rdPage.aspx?rdReport=yourAGReport&rdAgReset=True
```

This can be achieved in several ways, including using Action.Report with Link Parameters.

In many use-cases, this cached data approach provides adequate performance. However, if you're working with very large data sets, you may be able to take advantage of the following two special data retrieval elements.

Crosstabs and Calculated Columns

If you're working with standard dalayers, you can use *The Calculated Column* elements beneath them to add columns to the datalayer. These columns will then be available to the end-user in the Analysis Grid.

To prevent creation of unmanageable tables, *numeric* columns (calculated or otherwise) are not available for use in a Crosstab Table as the Header Values Column, Label Values Column, or the Secondary Label Column.

DataLayer.ActiveSQL

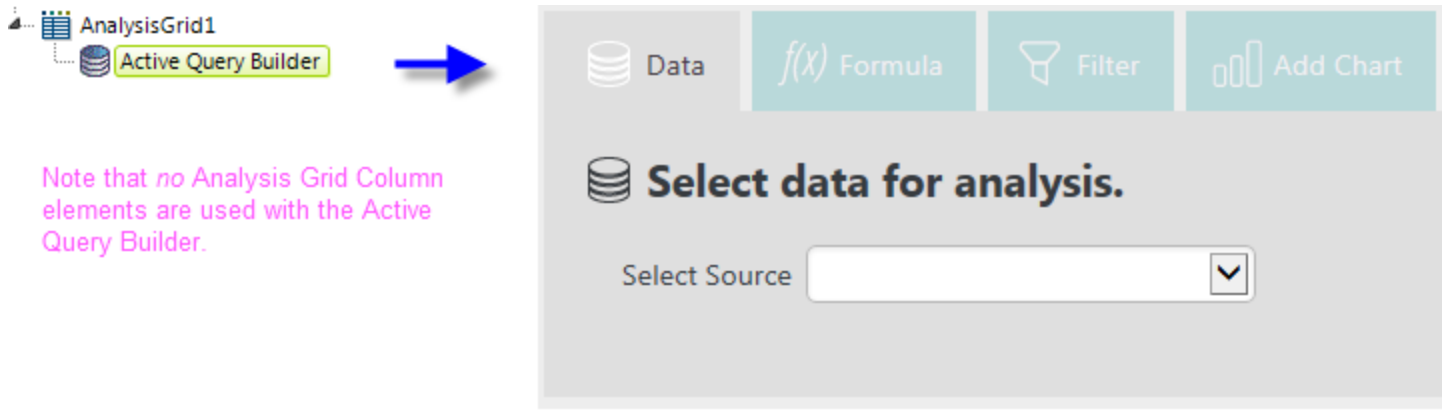
This special-purpose datalayer, *DataLayer.ActiveSQL*, can provide better performance for large data sets. Follow the link and read its documentation for important database restrictions.

The datalayer provides the basic SQL query needed to retrieve data but doesn't initially retrieve and cache *all* of the result set, like a standard datalayer does. Instead, in response to runtime manipulations of the Analysis Grid interface, it dynamically modifies its own SQL query and retrieves the minimum required amount of data with each request. This increases the number and frequency of SQL queries the database server must handle (and, generally, isn't a burden on it) but improves performance for the user.

Active Query Builder

Typically, when the Analysis Grid is configured with a SQL datalayer, it retrieves data based on a SQL query hard-coded by the developer. The query determines the dataset that will be available to users for analysis.

The **Active Query Builder** element, available with the Self-Service Reporting Module, extends our ActiveSQL technology and is subject to the same database restrictions as DataLayer.ActiveSQL, see *Add-on Modules*. It provides a different approach for Analysis Grid queries: when used instead of a datalayer, it allows *users* to have more choice about what data to work with, by giving them a way to interactively select tables, views, columns, and joins, at runtime.



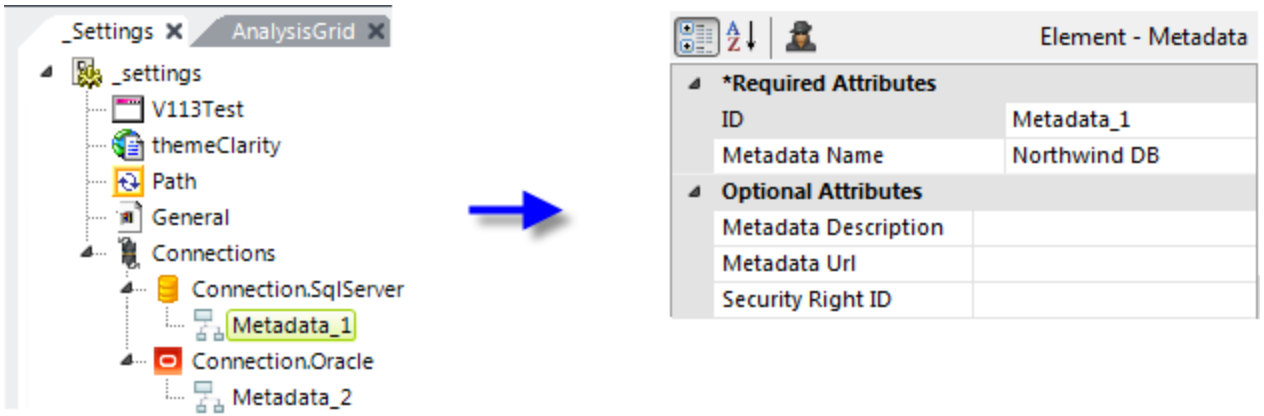
When added as a child of an Analysis Grid, the Active Query Builder element causes a special "Data" tab or button to appear in the Analysis Grid, as shown above. In it, users can select the tables, views, columns, and joins they want to work with. Like DataLayer.ActiveSQL, the Active Query Builder creates SQL queries on-the-fly based on Analysis Grid UI manipulations.

The developer, however, still ultimately controls which tables, views, columns, and joins are available in this tab. This is done by building a "metadata file", associated with a Connection element, which enumerates all of the database objects that will be available to users for selection in the Analysis Grid.

The Active Query Builder element's attributes are:

Attribute	Description
ActiveSql Buffer Size	Specifies the maximum number of rows to buffer when reading from the data source. The buffer is used when paging tables so that the data source need not be queried for every new page of rows. The default value is *5,000* rows.

Attribute	Description
	The default value is *200* rows.
Hide Column Selection	Specifies which Active Query Builder user interface to display. When set to *True*, a simplified interface is shown that allows users to select tables, but not columns (all columns are automatically selected). The default value, *False*, allows table and column selection.
Metadata IDs	Specifies the IDs of one or more Metadata elements that have been defined beneath Connection elements. Multiple IDs can be entered in a comma-separated list and, when more than one Metadata ID is specified, the interface offers users a choice of data sources. The default, blank value will use all Metadata elements that are included.
Template Modifier File	Specifies the name, with file extension, of "Template Modifier Files" on page 426 to be applied to the Active Query Builder at runtime. This is an XML file that can be used to change the underlying template used to create the Active Query Builder. Template Modifier files can be in any folder accessible to the application; if a folder isn't specified, the _SupportFiles folder is assumed.



The **Metadata** element provides a tool, the Metadata Builder Wizard. This tool is used during development to create the metadata file. The Metadata Builder Wizard has been deprecated and replaced with a browser-based tool, the **Web Metadata Builder**, which is described in *Use the Web Metadata Builder*.

Multiple metadata files can be associated with a single Connection element. You can add Metadata elements and build the files by selecting the Connection element and running the wizard. Or, metadata can be retrieved instead from a REST-style web service by specifying the service's URL in the Metadata element's **Metadata Url** attribute. When the MetaData Url is specified, the Web MetaData Builder works in read-only mode, allowing the user to view everything, but editing permissions are withheld.

Unless otherwise configured, the **Active Query Builder** will automatically include the metadata from all existing Metadata elements in the Data tab. If this is not desired, its **Metadata IDs** attribute can be used to list specific Metadata elements (and their files) to be used.

Analysis Grid Column elements are *not* used with the Active Query Builder. You'll see an error if you include these elements.

The Data tab panel normally includes check boxes so that users can individually select columns for analysis. However, these can be hidden by setting the Active Query Builder element's **Hide Column Selection** attribute to *True*, in which case *all* columns will be available for use.

Aggregations

Users can create aggregations of column values and display them in summary rows, using either the Aggregate option at the top of the Table configuration area, or by clicking a table column header and selecting Aggregate:


Table
Columns Sort Group Aggregate Paging
1 2 3 4 5 6 7 8 9 10
"Top" results positioning
Customer ID
Order ID
Employee ID
Order Date
Ship Country
Unit Price
Quantity
Sum: 51317
Sort A-Z
Sort Z-A
Filter
Aggregate
Format
Add Chart
Add Crosstab
Hide Column
Sum
Average
Standard Deviation
Count
Distinct Count
Maximum
Minimum
VINET 5 7/4/1996 France 14 12
VINET 5 7/4/1996 France 9.8 10
VINET 34.8 5
TOMSP 18.6 9
TOMSP 42.4 40
HANAR 7.7 10
HANAR 42.4 35
HANAR 10250 16.8 15
16.8 6
15.6 15
3 7/8/1996 France 16.8 20
Click column header and select options to aggregate or remove existing aggregations
"Bottom" results positioning
Distinct Count: 830
Sum: 51317

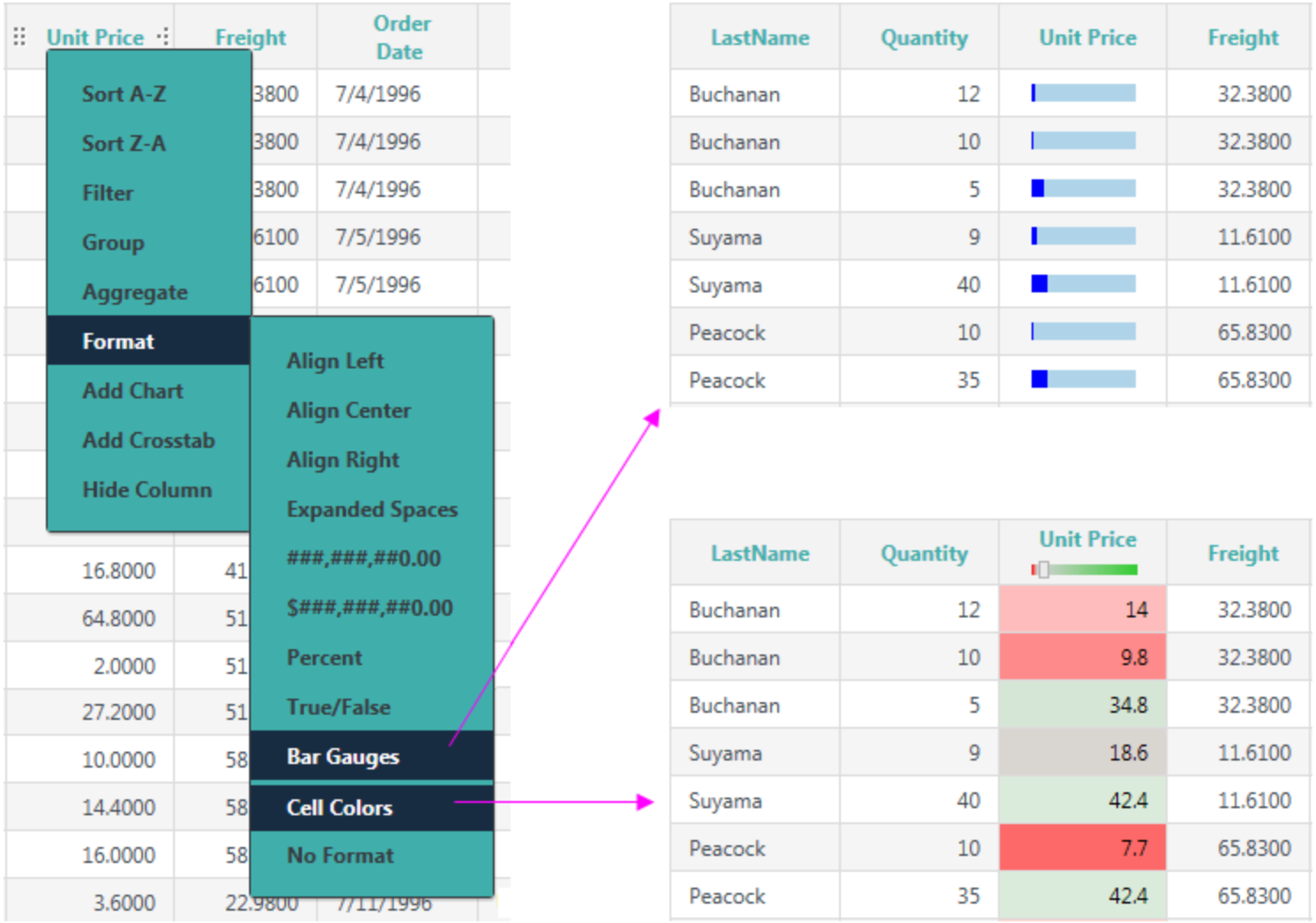
When creating aggregations, columns with Null values are excluded by default. If you want them to be included, create the constant `rdCalculationsIncludeNulls` and set it to *True*.

The format of aggregated values will match that of their table column.

Aggregated values, by default, are displayed with a label that's the name of the aggregating function. In the example above, they're "Sum:" and "Distinct Count:". These labels can be hidden by setting the Analysis Grid element's **Hide Function Names Default** attribute to *True*.

Analysis Grid Developers - Special Formatting

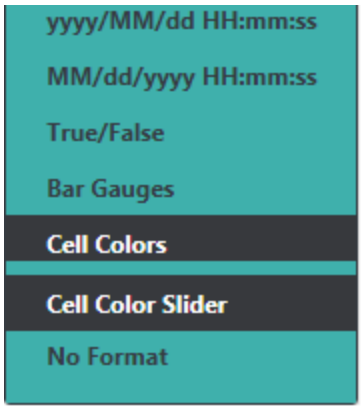
Data shown in an Analysis Grid Column element can be formatted by the developer using the Analysis Grid Column element's **Format** attribute or, if the attribute is left blank, at runtime by the user.



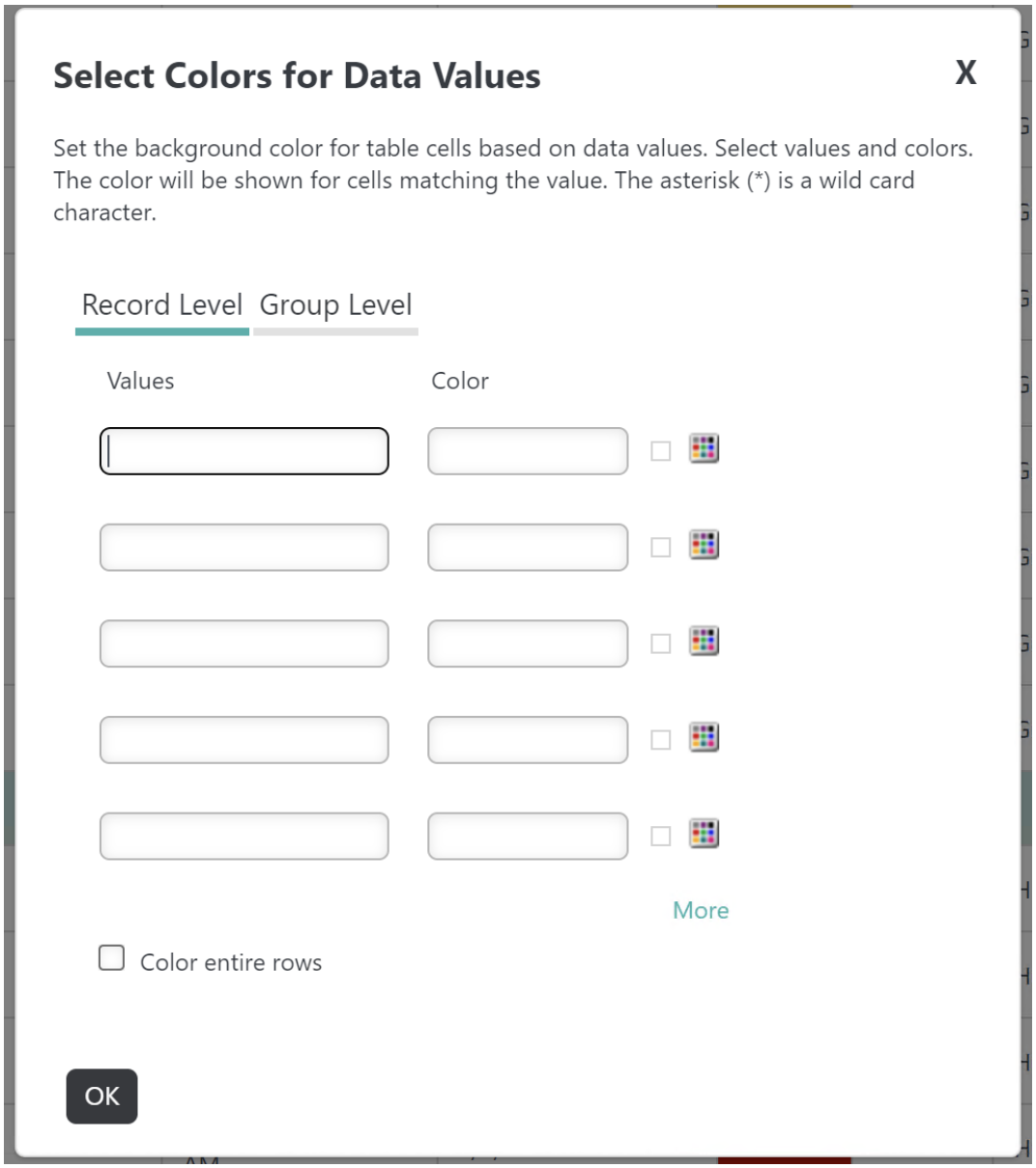
The attribute value options in Studio are *Bars* and *BackgroundColorSlider*. The corresponding runtime menu options, shown above, are *Bar Gauges* and *Cell Colors.*

Selecting the *Bar Gauges* option will produce a horizontal bar gauge within the column, as shown above. The actual data value will be automatically assigned to the element's **Tooltip** attribute. Selecting the *Cell Colors* option will display the column value using an imbedded Cell Color Slider, as shown above.

The *Cell Colors* feature has been changed to allow the user to apply conditional colors at runtime:



Its previous functionality is now represented by a new Format menu item, *Cell Color Slider*, as shown above.


Select Colors for Data Values
X
Set the background color for table cells based on data values. Select values and colors. The color will be shown for cells matching the value. The asterisk (*) is a wild card character.
Record Level
Group Level
Values
Color
More
Color entire rows
OK

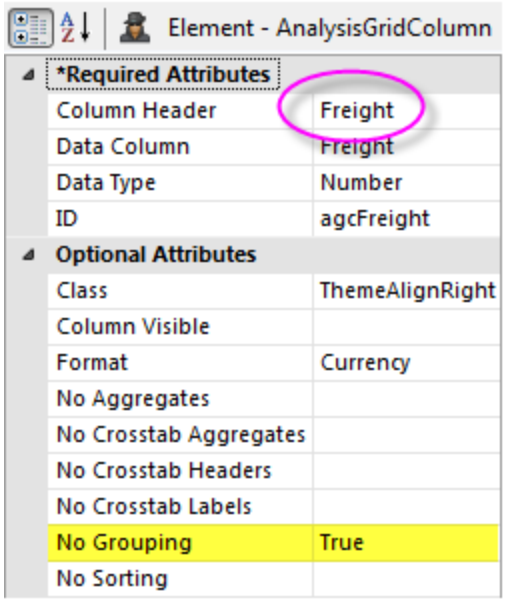
The menu's *Cell Colors* item now displays the pop-up dialog box shown above, allowing the user to select data value ranges and cell background colors. If the Analysis Grid utilizes aggregates at the group level, users can also format conditional colors by selecting the **Group Level** tab.

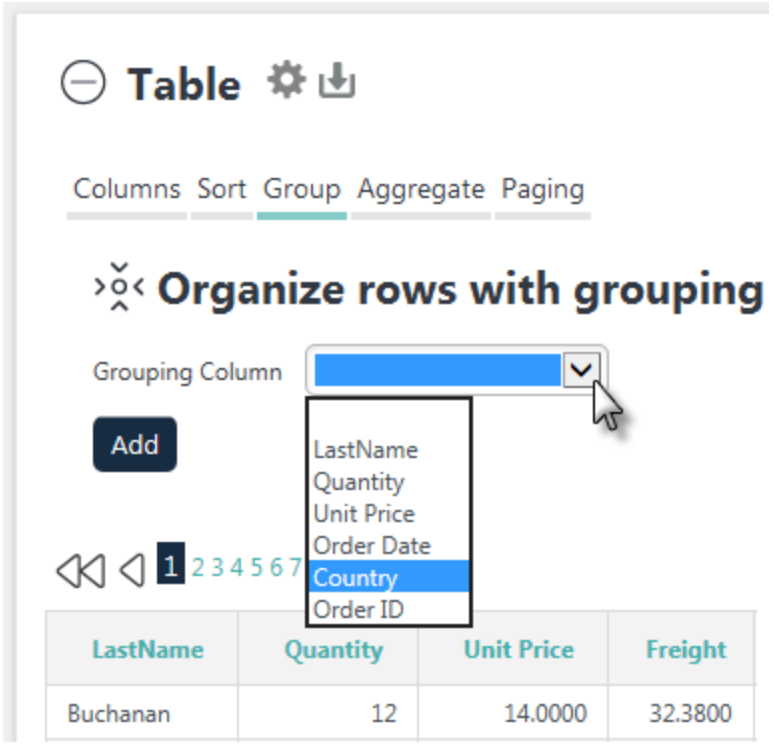
If you use the Analysis Grid Column element's Format attribute to change the *appearance* of data, for example, to cause a *1* or *0* data value to be displayed as *Yes* or *No*, users who attempt to create custom formulas at runtime using that column may run into difficulties. They may not realize that the data is really, as in the example, a number instead of text. In this case, user education about the actual underlying data values is the key to smooth operations.

For more information about the available data formats, see *Format Data*.

Analysis Grid Developers - Controlling Column Availability

Each **Analysis Grid Column** element has attributes that allow you to control whether or not that column can be used with a specific feature. For example, you may decide that some columns should be available for Sorting, while others should not.





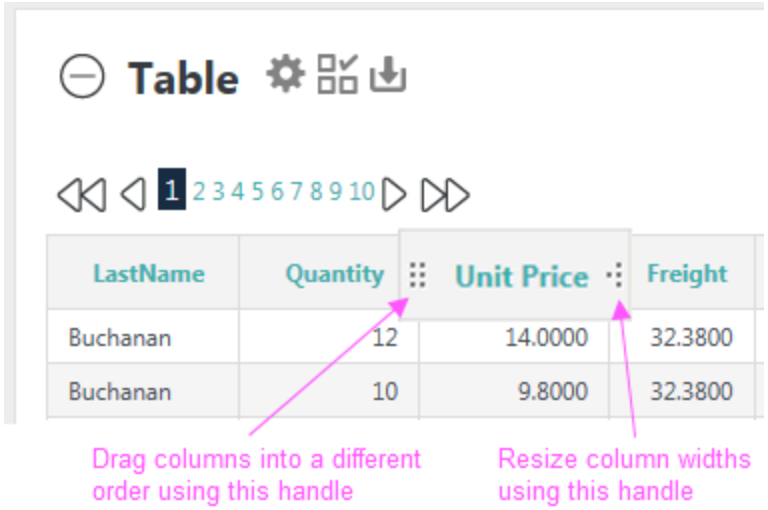
Columns available for grouping *do not* include Freight

For example, assume that an Analysis Grid Column that displays freight charges is *not* a good candidate for grouping. Setting this column's **No Grouping** attribute to *True* removes this column from the list of available columns in the Grouping feature, as shown above.

Developers can improve the overall usability of the Analysis Grid by determining in advance which columns are good candidates for aggregation, grouping, sorting, etc. and configuring the Analysis Grid columns accordingly.

Analysis Grid Developers - Changing Column Order and Width

The top-to-bottom order of Data Table Column and Analysis Grid Column elements in your report definition dictates the initial left-to-right order of columns in the Analysis Grid table.

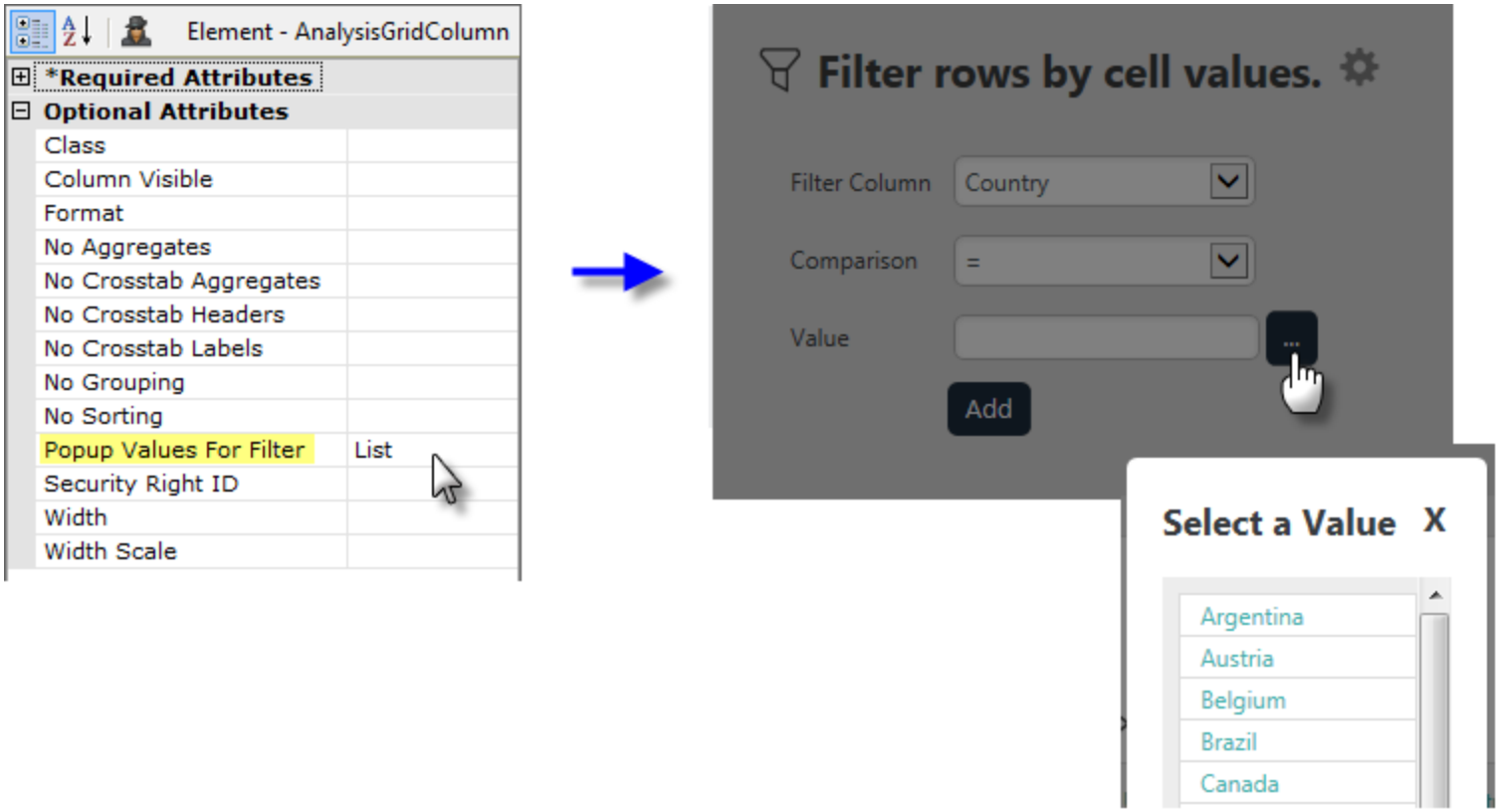


Users can rearrange the order, and change the widths, of table columns using two "drag" handles that appear when you hover your mouse cursor over a column header, as shown above.

In the Data Table element, set the "Draggable Columns" attribute to "True" to enable column dragging. If this attribute is set to "True", the column order will be remembered for future use.

Analysis Grid Developers - Setting Filtering Pop-up Values

Analysis Grid Column elements have a **Pop-up Values for Filter** attribute, which can be set to *List* or *Calendar*.



The example above shows what happens when it's to *List*. When this column is selected in the Filter panel, users can either type in a value or click a browse button. Clicking the button will cause a modal pop-up option list to be displayed, and the value for the filter can then be chosen from the list.

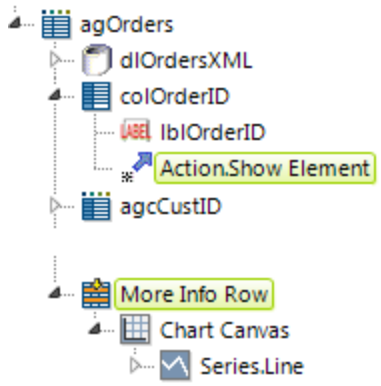
If the attribute is set to Calendar, then a pop-up calendar will appear. Obviously, this attribute value should only be used for date-type columns.

Analysis Grid Developers - Using a More Info Row

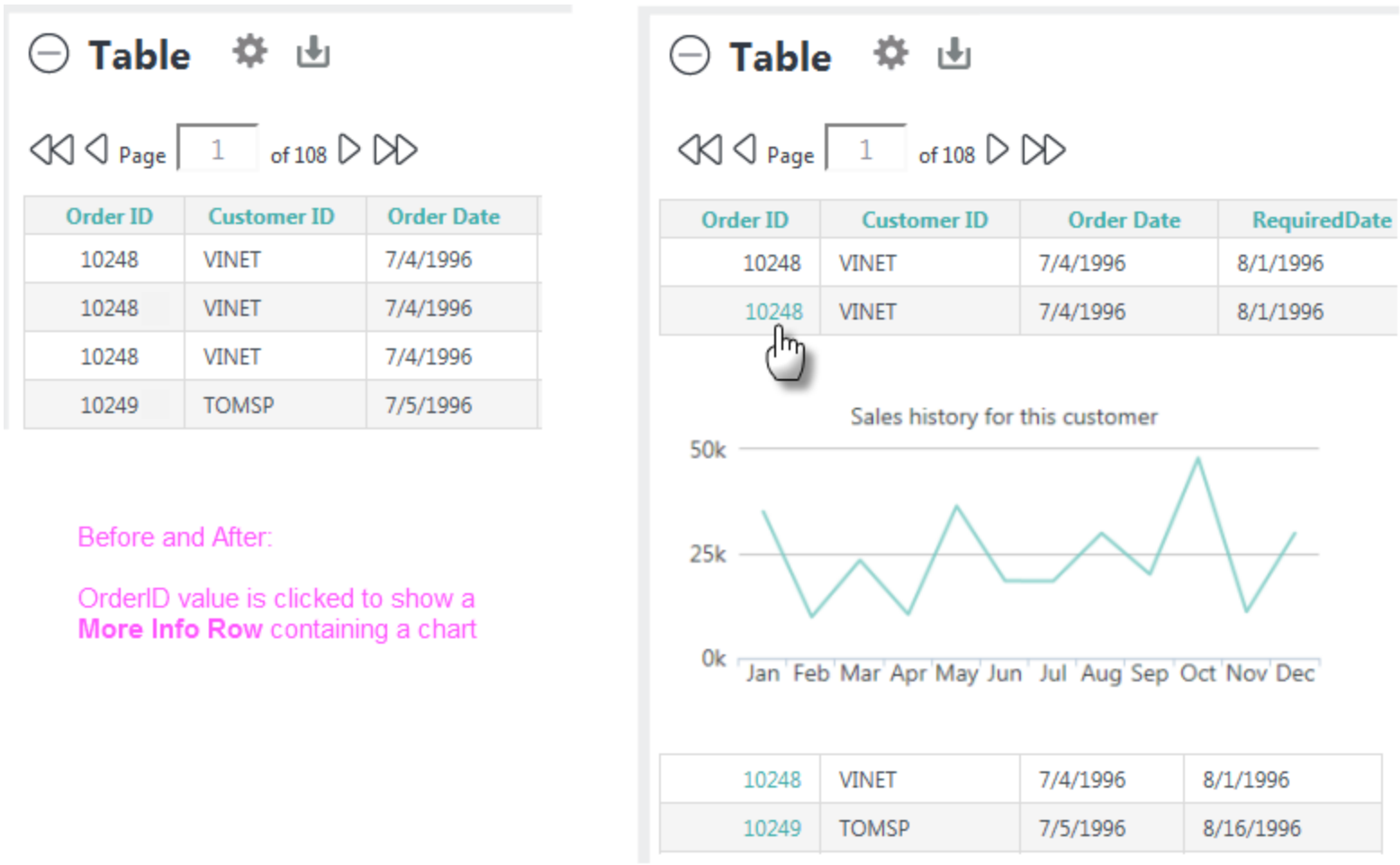
The **More Info Row** element can be used to show additional data "inside" a table or Crosstab Table row. The row expands vertically to accommodate rows of additional data or other visualizations.

In order to use a More Info Row element with an Analysis Grid, you must include a regular Data Table Column element as one of your table columns. This is because More Info Row elements are shown/hidden using an Action.Show Element element, which is not a valid child of Analysis Grid Column elements.

Use this Data Table Column to display values, such as invoice, product, or ID numbers, that won't be used for further analysis in the Analysis Grid.



In the example above, Action.Show Element has been added beneath the Data Table Column element "colOrderID". A More Info Row element has also been added, with a Chart Canvas chart beneath it.



As shown above, clicking the column's value at runtime will then show/hide the More Info Row and whatever it contains.

For more information about the general use of More Info Rows, see *Data Table Rows*.

Analysis Grid Developers - Controlling Feature Access

The Analysis Grid and Analysis Grid Column elements include attributes that allow developers to control **access** to features and data, and **user interaction** with the element's controls. The following table lists the elements and the attributes that can be used to control feature access.

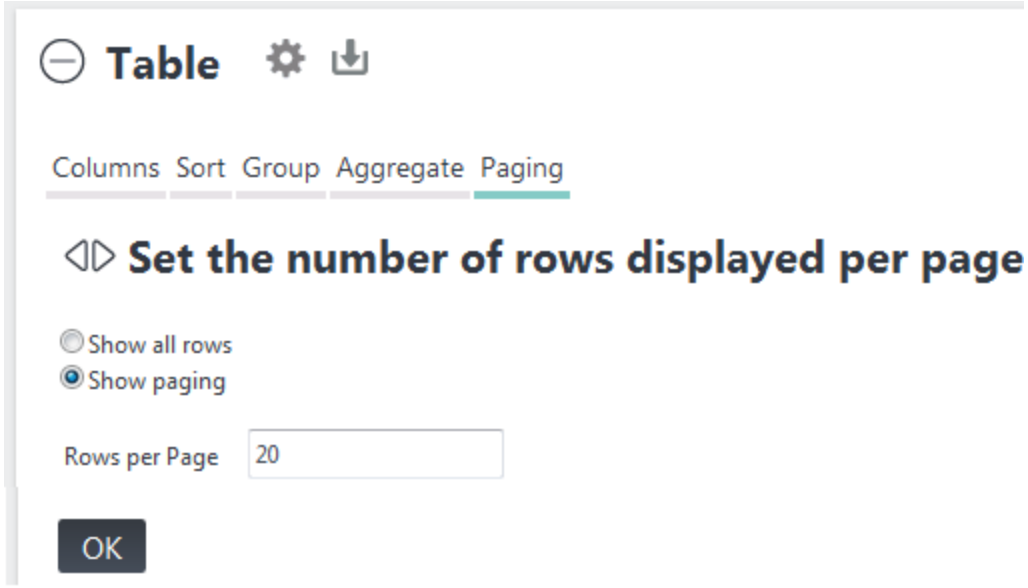
Element	Attribute	Description
Analysis Grid	Allow Data Pause Retrieval	Specifies whether or not to display an extra button on the Analysis Grid which, when clicked, temporarily halts data retrieval from the server. Default value is *False*. When set to *True*, you can automatically apply Pause Data Retrieval at the application level using the constant `rdAgDataRetrievalPaused`.
Analysis Grid	Allow Zoom Control	Specifies whether chart zoom is enabled/disabled for Bar Charts, Line Charts, Curve Lines, and Scatter Plots. Default value is *False*.
Analysis Grid	Hide Aggregates Hide Charts Hide Crosstabs Hide Filters Hide Formatting Hide Formulas	Specifies whether feature tabs or buttons will be hidden in the menu panel and all Table column header drop-down options lists. Default value is *False*.

Element	Attribute	Description
	Hide Grouping Hide Layout Hide Menu Hide Paging Hide Sort Order	
Analysis Grid	Hide Exports	Specifies whether the Export icon will be hidden in the Data Table panel. Default value is *False*.
Analysis Grid	Hide Function Names Default	Specifies whether a label describing the aggregation in Group Summary Rows and Aggregates is hidden. These include Count, Sum, and Average. Default value is *False*.
Analysis Grid	Hide Header If No Data	Specifies whether the Data Table/Crosstab Table header is hidden when there is no data to show. Default value is *False*. To hide the header when there is no data, set this constant to *True*.
Analysis Grid	Security Right ID	If Logi Security is implemented, specifies the security Right required to view this element. Default value is *no restriction*.
Analysis Grid	Draggable Panels	Specifies whether the user can re-arrange the vertical order of the major table and chart panels. The new panel arrangement is "remembered" only for the user's current session. Default value is *True*.

Element	Attribute	Description
Analysis Grid Column	Column Visible	Specifies whether this column is initially visible. Default value is *True*.
Analysis Grid Column	No Aggregates No Crosstab Aggregates No Crosstab Headers No Crosstab Labels No Filtering No Formatting No Grouping No Sorting	Specifies whether this column is included in the drop-down list of columns in the control panel for the feature. Also, for some options, controls whether the option appears in this column's header drop-down option list. Default value is *False*.
Analysis Grid Column	Security Right ID	If Logi Security is implemented, specifies the security Right required to view this column. Default value is *no restriction*.

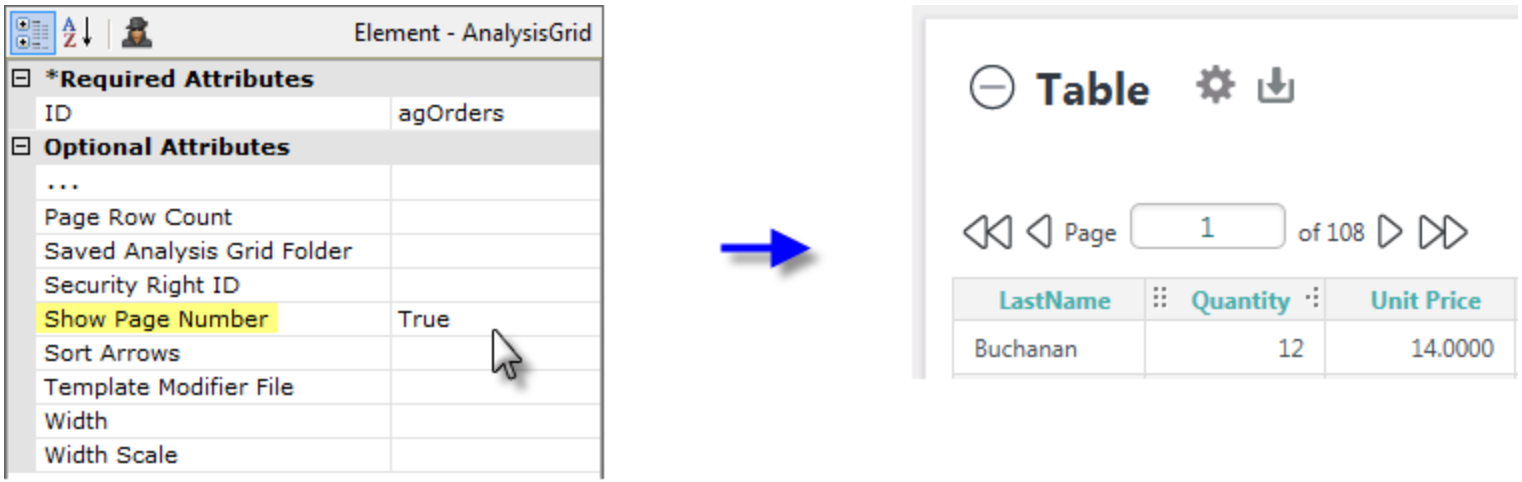
Analysis Grid Developers - Set Pagination Options

This feature allows users to determine how many **rows of data** will appear in the Data Table:

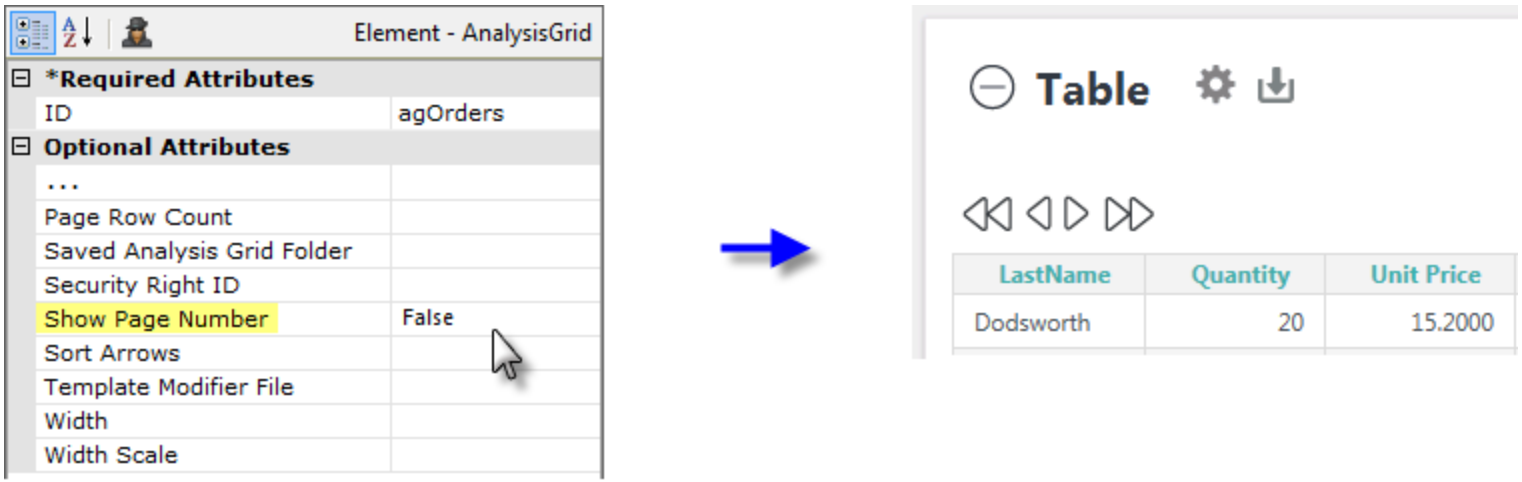


Selecting the paging option **Show all rows** will display *all* of the data at once in the Data Table; **Show paging**will display a subset of rows and also include the paging controls. Developers may not want to allow users to select all rows as it may result in a lengthy delay while data is retrieved. You have the option to hide this radio button from users by setting the Analysis Grid element's **Disable Show All Rows** attribute to *True*.

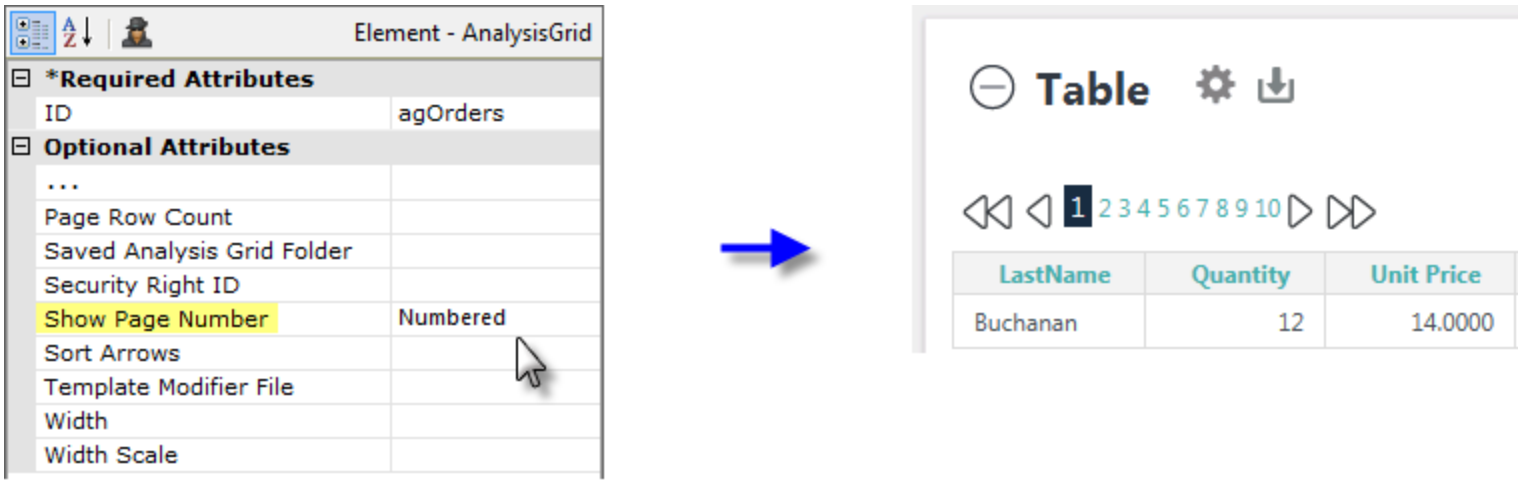
When Show Paging is selected, one of three styles of **paging controls** is displayed, depending on how the developer configures the Analysis Grid element in the report definition:



If the Analysis Grid element's **Show Page Number** attribute is set to *True*, the navigation controls, with page number display and input, as shown above, will appear.



If the Show Page Number attribute is set to *False*, the navigation controls will appear, as shown above, but without a page number.

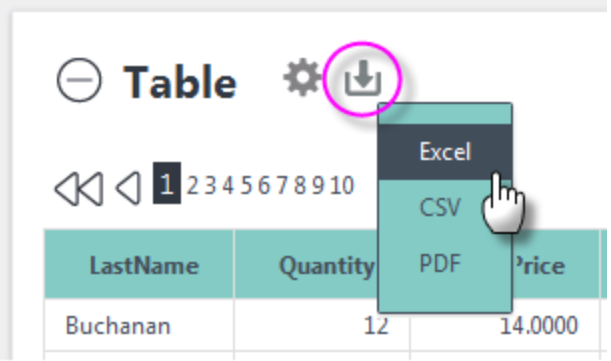


And, finally, if the Show Page Number attribute is set to *Numbered*, or is left blank, the navigation icons will appear, as shown above, along with a range of page numbers. In addition, two other attributes will become active: **Numbered Page Count**, which specifies how many numbers will be shown at a time (default = *10*), and **Current Page Class**, which allows the number for the current page to be styled (color, size, background, etc.) differently from the others.

The look of the navigation controls will vary depending on which Theme is applied.

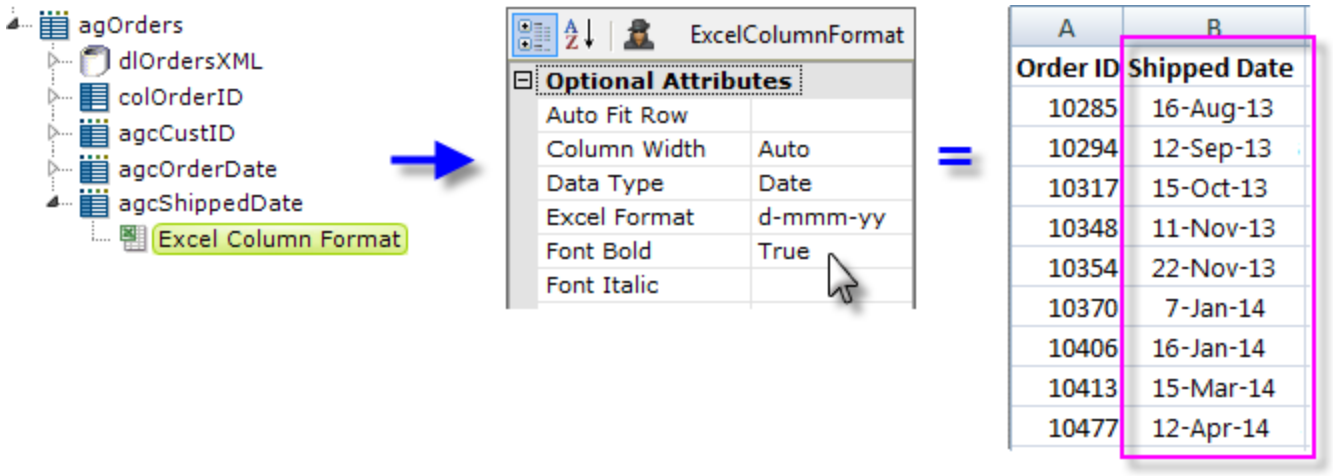
Analysis Grid Developers - Exporting Data

The Analysis Grid can be configured by the developer to include options to export table data to three formats: **Excel**, **CSV**, and **PDF:**



The application can also be configured to export the report title and filter information. By default, the report title and filter information are not included with the export. To enable this functionality, set the global constants `rdIncludeReportTitleInExport` and `rdIncludeFilterInfoInExport`to **True**.

In all cases, the exported file is written to a temporary file, which is then opened in the browser or related application for display, manipulation, printing, or storage.



Developers can optimize Excel exports by adding an **Excel Column Format** element to each grid column, as shown above. This provides developers with full control over the *layout* and *appearance* of exported data in the worksheet. The element's **Data Type** attribute is especially useful for ensuring that data is correctly formatted when it arrives in Excel. There are no export-specific elements for the other export formats.

Export Threshold

When working with very large data sets, exports may not be practical. Developers can control this using the Analysis Grid element's **Max Rows Export** attribute specifies the maximum number of data rows that can be exported.

When the number of rows in the Analysis Grid's main table is equal to or less than this value, exports will be enabled; if the number of rows exceeds this value, exports are disabled. This allows developers to prevent users from running export requests that will take an unacceptable amount of time. The default value for this attribute is *100,000* rows.

If the special value -1 is entered, the default maximum limit is removed, allowing *allrows* to be exported.

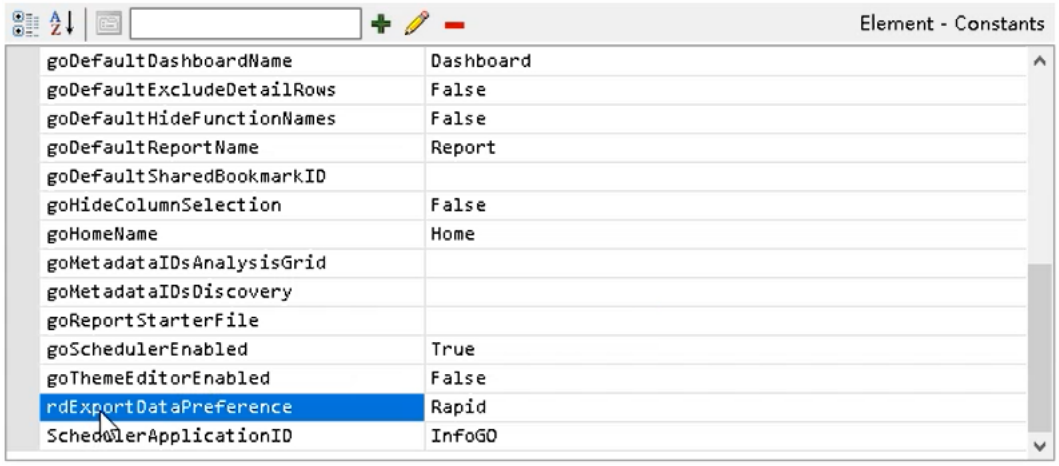
At runtime, users can reduce the number of rows in the Analysis Grid's main table, and thus enable exports, by adding one or more Analysis Grid filters.

Rapid Export Options

Excel and CSV exports can also be configured for Rapid Export, which provides a faster mechanism for exporting large datasets. The Rapid Export feature honors basic formatting (bold, font, italics, etc), but it does not apply custom styling or group aggregation.

You cannot export a report with more than one Data Table.

In Studio, set the attribute **Export Data Preference** (Formatted, Empty {default}, or Rapid) on the Target.NativeExcel and Target.CSV elements. This preference can also be set on a global application level by setting a restricted constant called rdExportDataPreference to **Rapid**, like below:



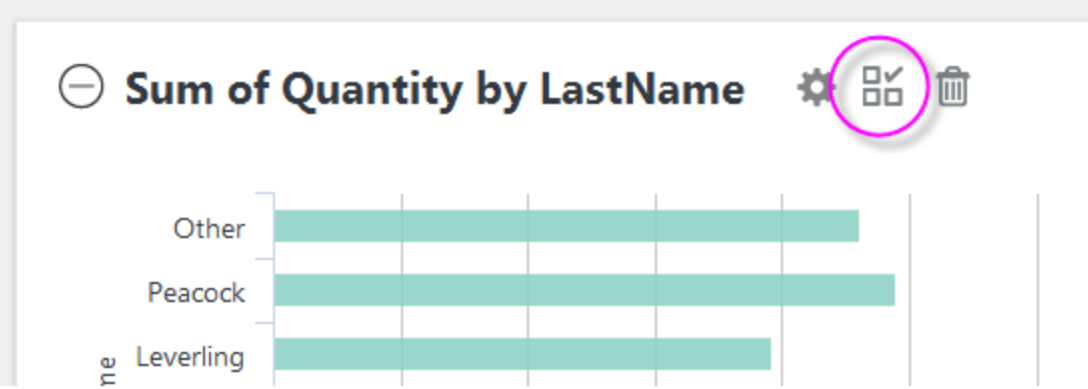
The default value for the above constant is empty.

v23.1 An additional constant, `rdShowExportDataPreference`, controls end-users' ability to specify an Export Data Preference (Formatted or Rapid) when scheduling an Analysis Grid report. By default, this constant is set to *False*.

For more information, see *Rapid Excel Export* and *Rapid CSV Export*.

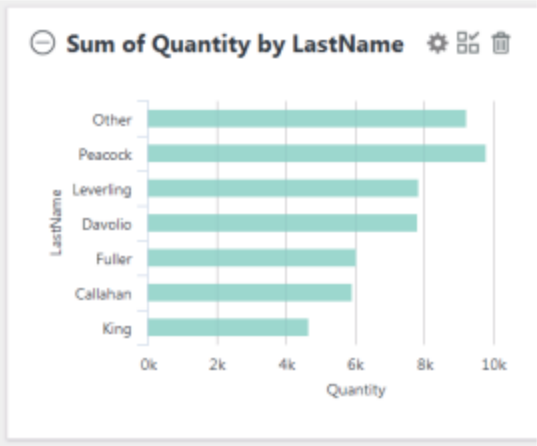
Analysis Grid Developers - Configuring Add-to-Dashboard

The Analysis Grid allows users to generate a Data Table or chart, then add it as a new panel in an existing Dashboard in another report. This is the "Add-to-Dashboard" feature.



When properly configured, Analysis Grid charts will display an **Add to Dashboard** icon, shown circled above.

From Analysis Grid...

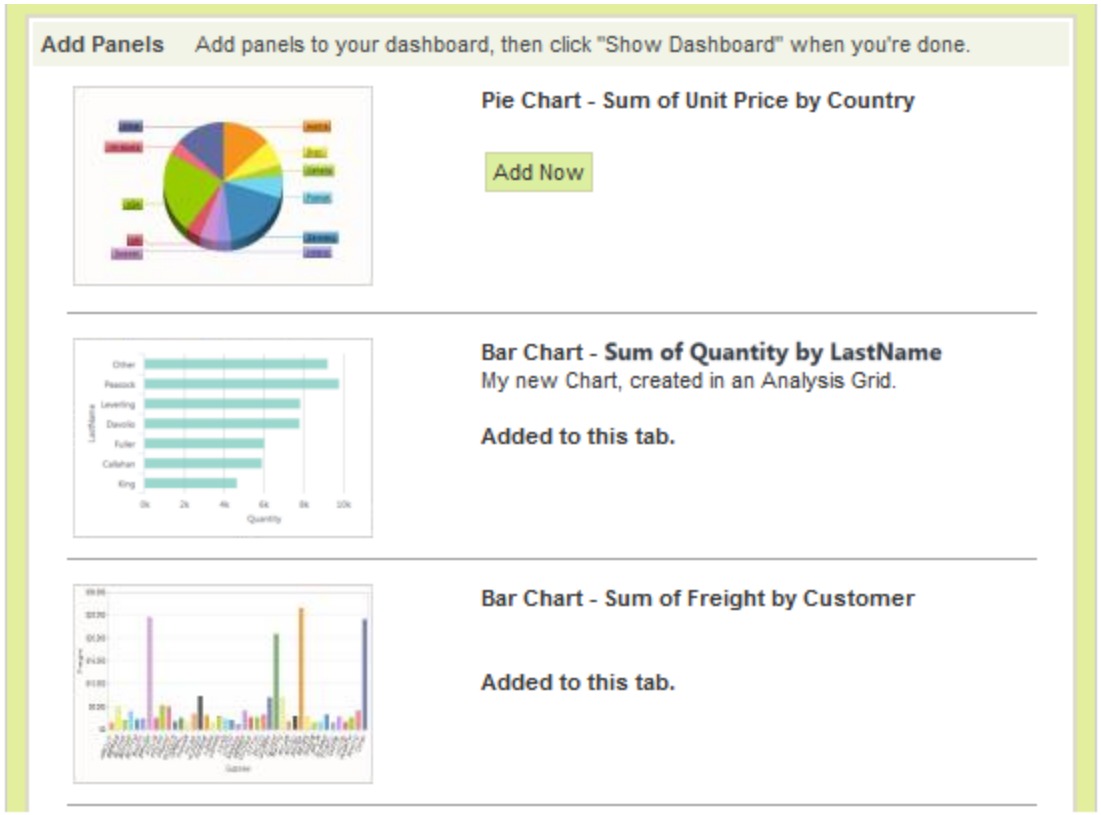


... to Dashboard

When the icon is clicked, the chart, with its resizing control, is added as a **new panel** in the Dashboard, as shown above. When the resizing control is used to resize the chart, the widths of the Dashboard columns and other panels will adjust dynamically.

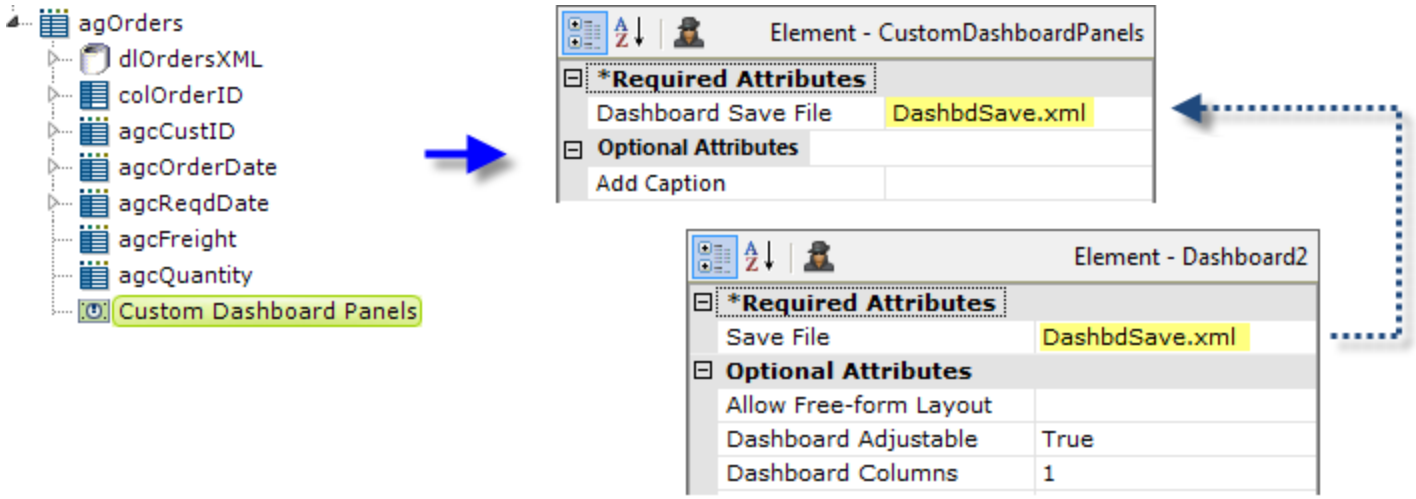
Behind the scenes, when the icon is clicked, the Analysis Grid writes the underlying XML for its chart into the Dashboard's configuration file (its "Save File"). In the Dashboard, the chart is inserted at the top of Column 1; if Dashboard tabs are being used, the insertion occurs in the currently active tab.

Just before panel insertion, the user will be prompted for the **Panel Title** (with the chart title from the Analysis Grid provided as a suggestion) and a description for display on the Configuration Page:



The new chart panel thereafter appears in the Dashboard Configuration Page, as shown above, just like any other panel, complete with a thumbnail image. The new panel can be removed from the visible Dashboard panels and from the configuration page entirely, using the usual controls.

The user can insert multiple charts into a Dashboard using this technique.



In the report definition, the developer makes this functionality available by adding a **Custom Dashboard Panels** element, shown above, beneath an Analysis Grid element. Its **Dashboard Save File** attribute value should be set to match the target Dashboard's **Save File** attribute value.

The Save File attribute values shown above have been shortened for visual clarity. However, a fully-qualified path and file name, with .xml extension, is required, so a realistic attribute value would be something like:

```
@Function.AppPhysicalPath~\DashFiles\DashbdSave.xml
```

and the account the Logi application runs under would be given **Write** permissions to the "DashFiles" folder. For experimentation purposes, you can use the rdDownload folder, for which appropriate permissions already exist (but its contents are periodically deleted, so don't use it for anything but experimentation).

Dashboard Save Files are often named using a token, such as `@Function.UserName~`, in order to create personalized Dashboard configurations and, in that case, the Custom Dashboard Panel attribute would use the same token.

The Custom Dashboard Panel element has a new attribute, **Add Caption**, which sets the caption for the button used to save the Data Table or chart. If left blank, the caption is "Add to Dashboard". For more information about Dashboards, see "Logi Info Dashboard " on page 351.

Charts that utilize the Zoom feature can also be added to the Dashboard for future use. To enable, set the attribute **Allow Zoom Control**to "True".

Analysis Grid Developers - Configuring for Report Author Galleries

The *Report Author* element, available in the Self-Service Reporting Module (SSRM) add-on, v12+, provides users with the ability to construct reports using a drag-n-drop interface. Visualizations for use with the Report Author are created using the Analysis Grid's "Add to Dashboard" feature.

To do this, the Custom Dashboard Panels element is used with an Analysis Grid, just as described in "Analysis Grid Developers - Configuring Add-to-Dashboard" on page 336. However, for this purpose, its **Dashboard Save File** attribute value does *not* point to an existing Dashboard Save file. Instead, it identifies a separate Save file that will be used later by the Report Author as its "Gallery" file.

If Logi Security has been and will be used, the current user's name will be available in the @Function.UserName~ token. This can be embedded into this attribute to provide separate Gallery files for individual users. For example, the attribute value might be:

```
@Function.AppPhysicalPath~\GalleryFiles\@Function.UserName~.xml
```

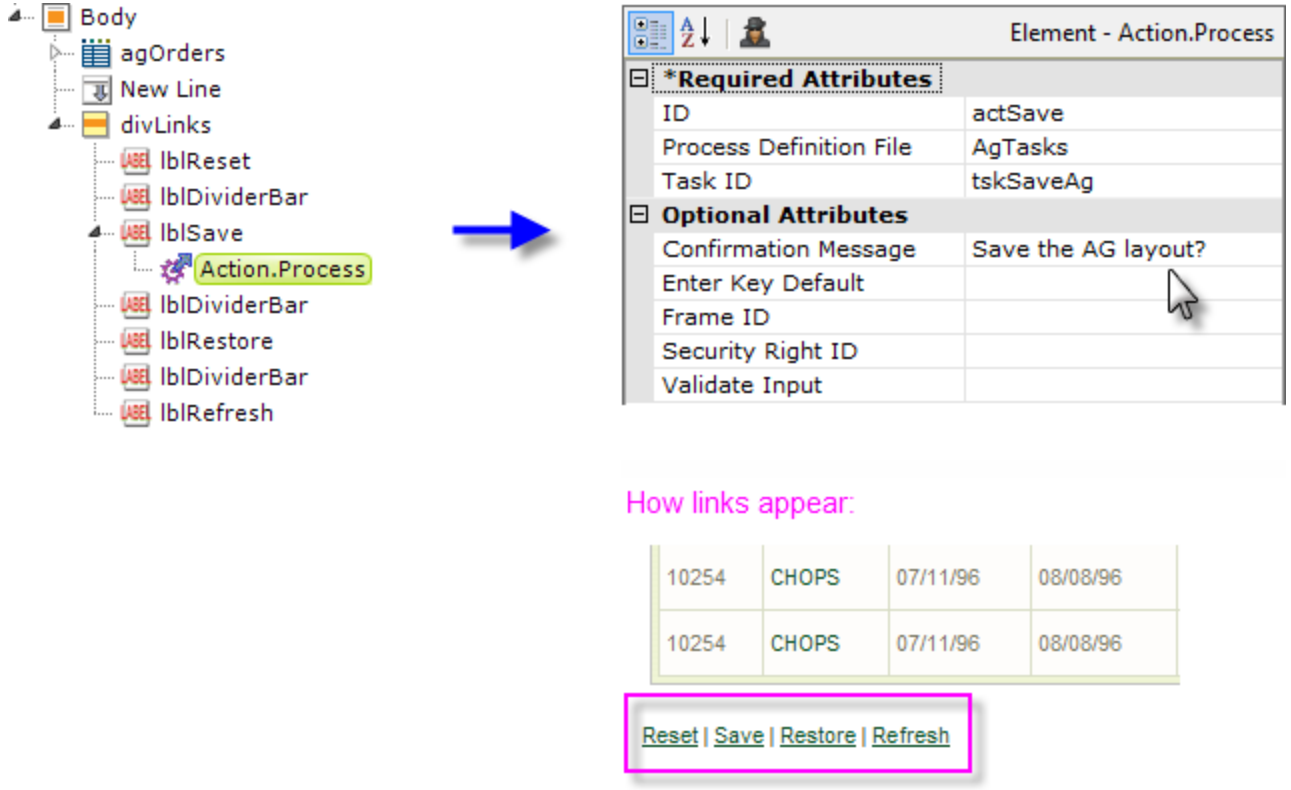
which might translate to:

```
C:\inetpub\wwwroot\MyReportApp\GalleryFiles\BGates.xml
```

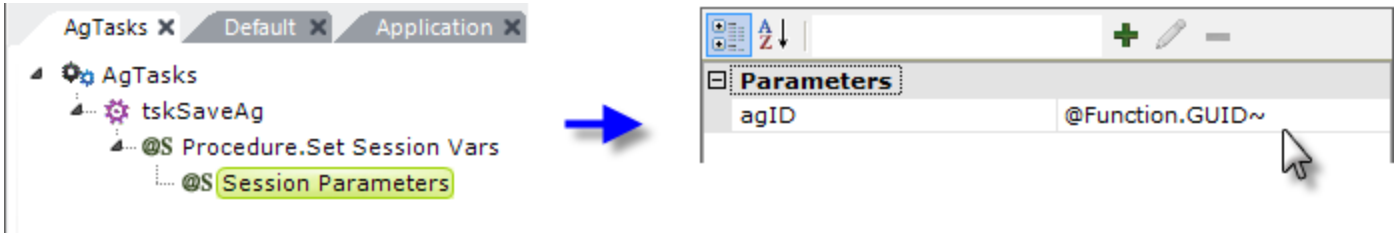
This file can be created, renamed, and copied as necessary to make it available to report definitions using the Report Author element.

Analysis Grid Developers - Saving and Loading Analysis Grids

It may be important for users to be able to **save** their Analysis Grid configurations between sessions. Their configurations can be saved on the web server and made available to them during their next session. Developers can implement this behavior with a simple **Process** definition. The following examples illustrate how to save and manipulate grid settings.

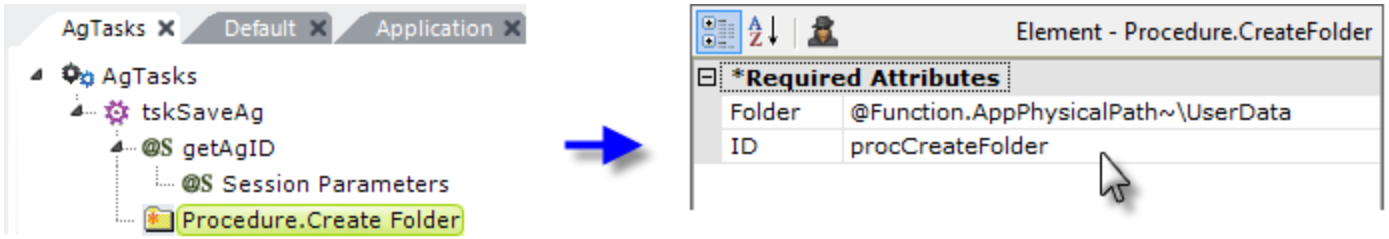


The original Analysis Grid example definition is shown above with a set of **Label** and **Action** elements added, beneath their own Division. The **actSave** element attributes point to a Process definition task, **tskSaveAg**.

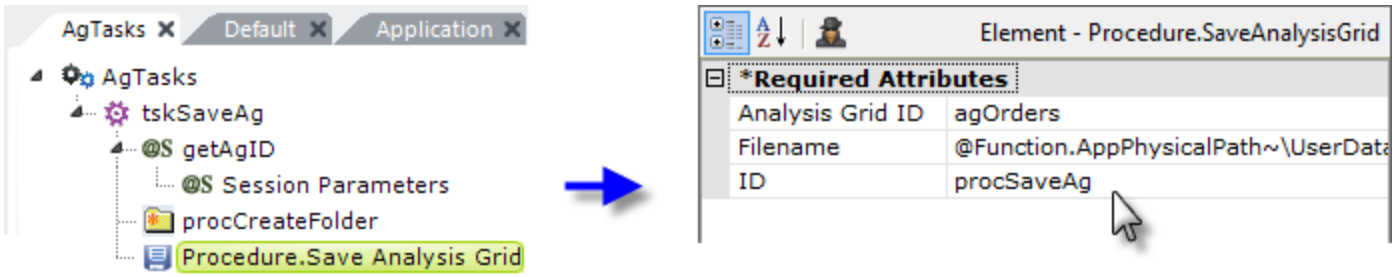


To create the task:

1. In a Process definition, add a **Task** element ("tskSaveAg").
2. Below it, add a **Procedure.Set Session Vars** element ("getAgID").
3. Below that, add a **Session Parameters** element and set its attributes as shown above. The GUID token provides a random 48-character number for use as a unique identifier. The session parameter name is **case-sensitive**!



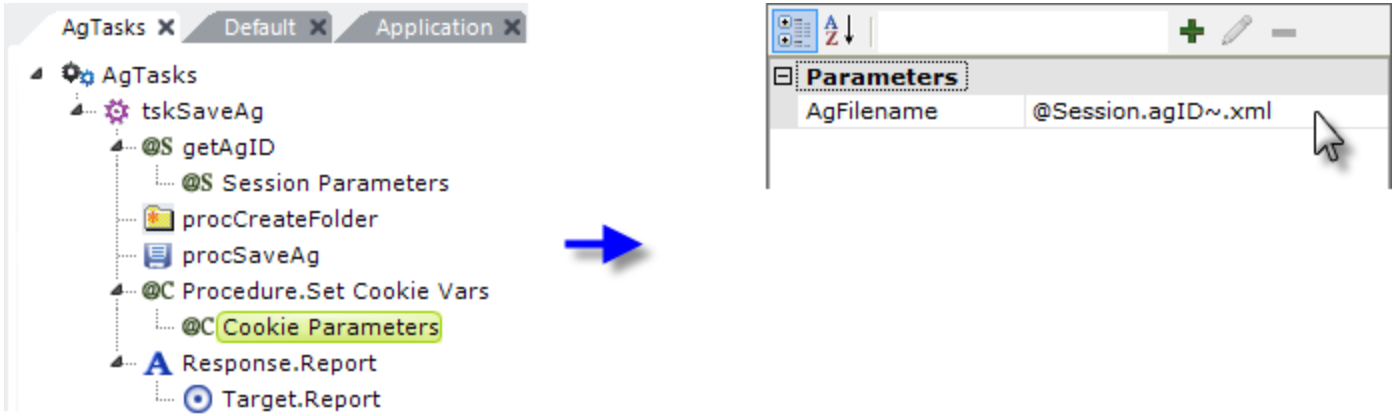
4. Next, add to the task a **Procedure.Create Folder** element and set its attributes as shown above.



5. Add a **Procedure.Save Analysis Grid** element to the task and set its attributes as shown above. The full **Filename** value is:

```
@Function.AppPhysicalPath~\UserData\@Session.AgID~.xml
```

The session variable created from the GUID token earlier is now being used as the file name; use of a session variable allowed us to "pass" the value through the task from an earlier procedure to this one.



6. Add a **Procedure.Set Cookie Vars** element to the task. Below it, add a **Cookie Parameters** element and set its attributes as shown above. This stores the name of the Analysis Grid settings file on the user's computer; the cookie will be used in future sessions to open the find and restore the settings.
7. Add a **Response.Report** and **Target.Report** element to the task to return processing to the report definition containing the Analysis Grid.

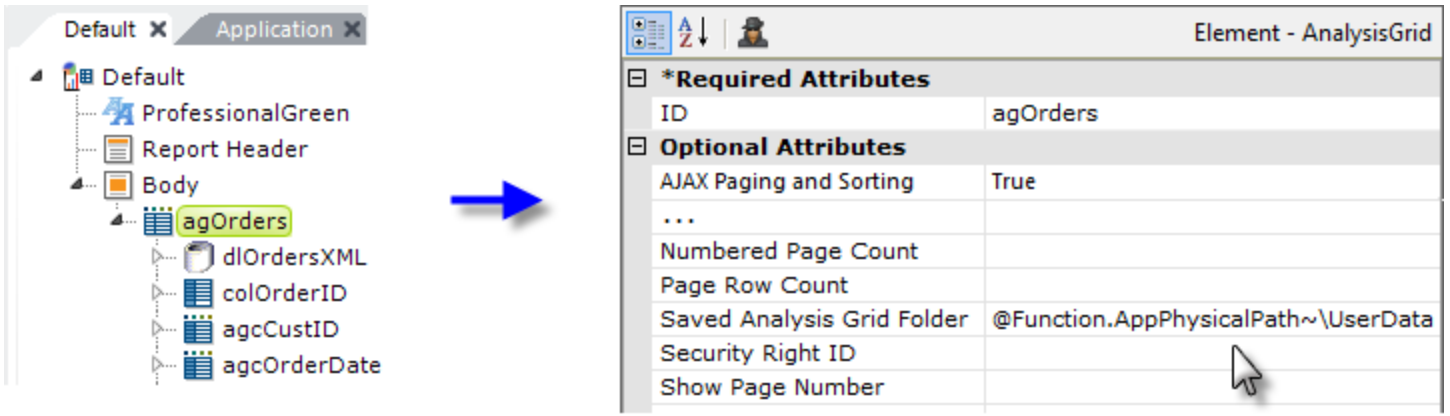
Cookies are useful when the user is unknown to the server and session information is unimportant. If authentication is integral to an application, developers can use strategies such as tying a saved Analysis Grid filename to a user ID and storing this information in a database. Developers can also create a list of saved Analysis Grids that users can access from any workstation.

Analysis Grid Developers - Special Query String Parameters

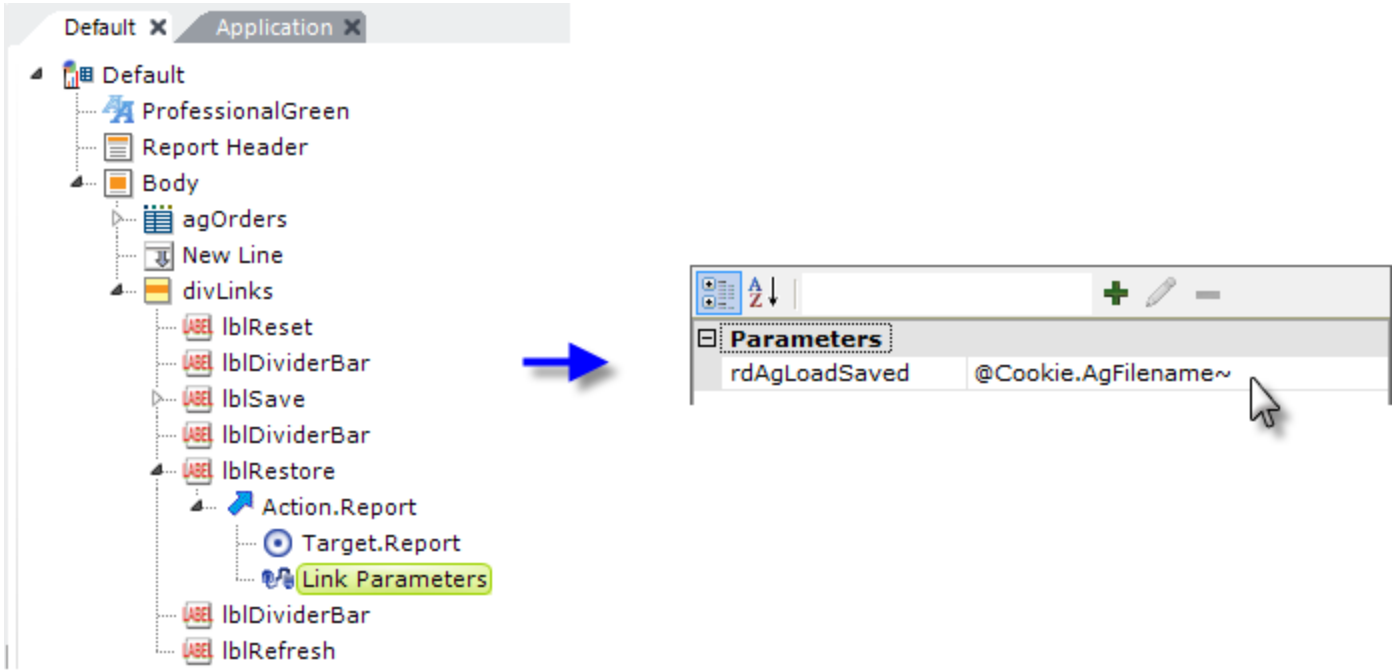
Logi Studio provides developers with three special query string parameters to manipulate Analysis Grids:

- **rdAgReset -** Set to **True** to revert the Analysis Grid back to its original, unmodified state
- **rdAgLoadSaved -** Set to the name of an XML file (with .xml extension) that contains saved Analysis Grid settings
- **rdAgRefreshData -** Set to **True** to refresh the Analysis Grid with current data from the data store, while preserving user configurations.

This example shows how to restore saved Analysis Grid settings using two of these parameters:



First, select the Analysis Grid element, as shown above, and set its **Saved Analysis Grid Folder** attribute to the path to the folder where we earlier saved the grid configurations.



Next, examine the "lblRestore" element in the example shown above. It has a **Target.Report** child element set to the current report ("Default" - *not* "Current Report") and a **Link Parameters** element that defines one of the special parameters as shown. When the Restore link is clicked, the report will be called and the parameter passed to it in the query string.

When this happens, the Analysis Grid settings will be set to those in the saved settings file (whose name was saved as a cookie on the user's computer in the previous example). The datalayer will then be run again to get fresh data, *without resetting* any of the configurations the user has made with the Analysis Grid controls.

A similar technique can be used to create "reset" functionality, using the **rdAgReset** parameter, which *does* clear any user settings and returns the Analysis Grid to its default state.

Analysis Grid Developers - Customizing Analysis Grid Appearance

Analysis Grid appearance can be changed most easily by applying a theme to the report definition. Most of the screen shots in this topic were taken with the *Signal* theme applied.

You can create your own custom theme, based on a standard theme, *Using the Theme Editor* tool.

Changing Appearance Using Style Classes

Analysis Grid appearance can also be customized using **style classes** and Logi Studio provides the following standard style sheet for all Analysis Grids:

```
<YourAppFolder>\rdTemplate\rdAnalysisGrid\rdAg10Style.css
```

Developers can override classes in this style sheet by *copying* them into their own application style sheet and modifying them there.

- Do not make changes in the standard style sheet in the rdTemplate folder!
- If you're using a Theme, it will override some of the Analysis Grid classes.

Changing Appearance Using Template Modifiers

The Analysis Grid element uses a "template file" to define certain element properties that are not otherwise available as attributes to the developer for modification. These include language- and culture-specific **Caption** attributes that you may want to change

for locale-based reasons (or you may simply want to change the captions to better suit your application).

The element's **Template Modifier File** attribute identifies a custom XML file developers can create containing elements that will override the same elements in the template file.

For example, the Analysis Grid template file:

```
<yourAppFolder>\rdTemplate\rdAnalysisGrid\rdAg10Template.lgx
```

contains several Label elements. One of them has an ID of "lblInstructCols"; this controls the "help text" that is displayed when the Layout icon is clicked. This text can be modified by changing the Caption associated with that Label element. To change the text from its default "Reorder and hide columns" to "Rearrange and hide columns", create your own XML file, identify it in the Template Modifier File attribute, and add this code to it:

```
<?xml version="1.0" encoding="UTF-8"?>
<TemplateModifier>
 <SetAttribute ID="lblInstructCols" Caption="Rearrange and hide columns" />
</TemplateModifier>
```

You can set the attributes for any number of elements in this file; examine the rdAgTemplate.lgx file to learn the ID and Caption attributes available. The template modifier file can be in any folder accessible to the web application; if a fully-qualified file path is not provided in the Template Modifier File attribute value, then the application expects it to be in your applications's _SupportFiles folder.

For more detailed information about template modifier files, see "Template Modifier Files" on page 426.

You can also format your chart's label and caption directly in the user interface. For more information, see Analysis Grid - Editing Chart Labels and Captions.

Logi Info Dashboard

Dashboards provide an organized arrangement of visualizations in tabs and panels. Like all super-elements, they have a built-in user interface and specialized functionality.

The following topics introduce Logi Info *developers* to one of the "super-elements", the **Dashboard**:

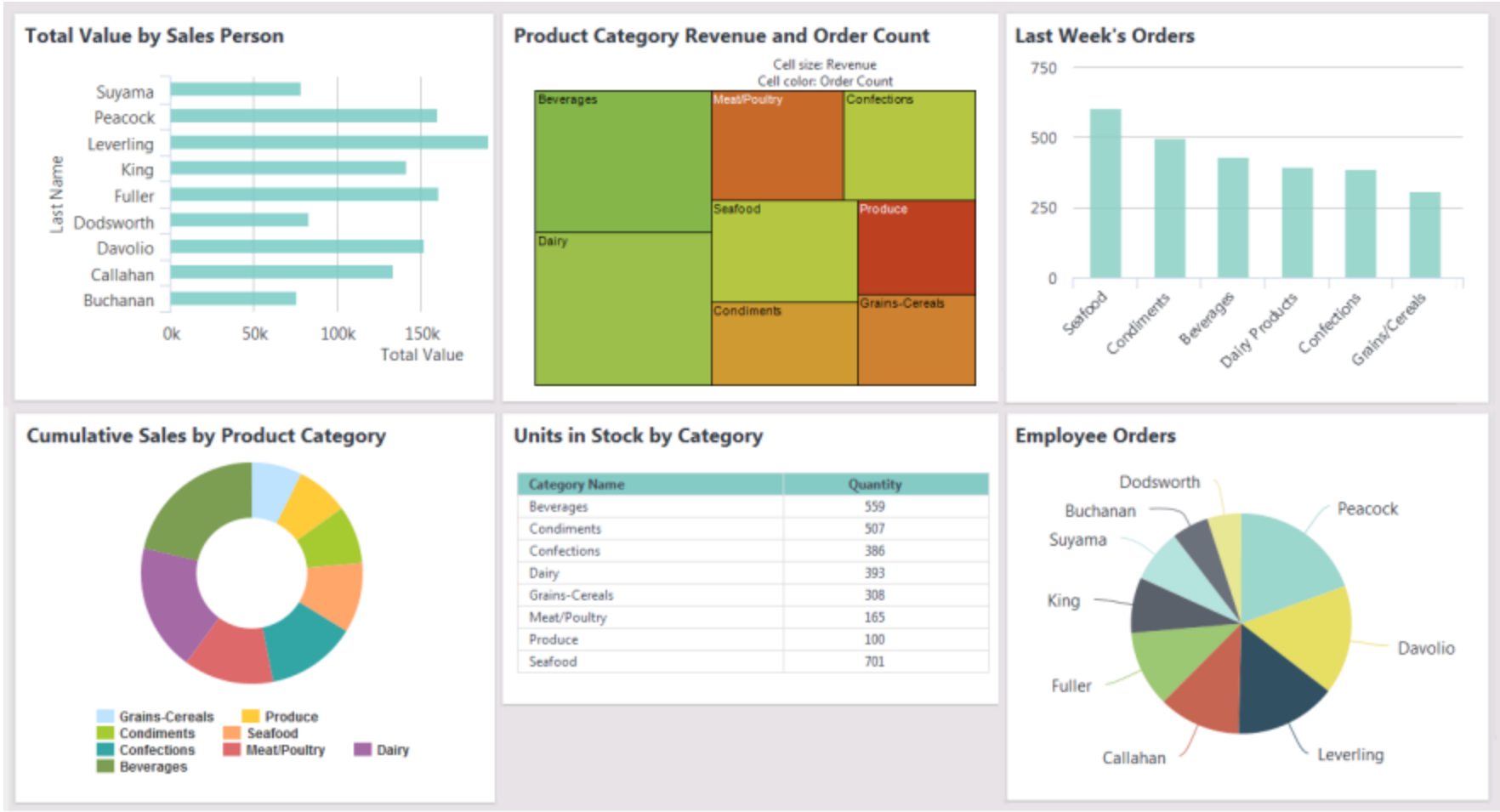
- Using the Dashboard Element
- Using the Panel and Panel Content Elements
- Working with Dashboard Tabs
- Saving Dashboard Settings
- Creating a Default Panel Arrangement
- Using Analysis Filters with Dashboards
- Using the Panel Parameters Element
- Using the Refresh Element Timer
- Using Action.Add Dashboard Panel
- Customizing Dashboard Appearance

Information about *using* a Dashboard, intended for *end-users*, is available in *Dashboard for End Users*.

About the Dashboard

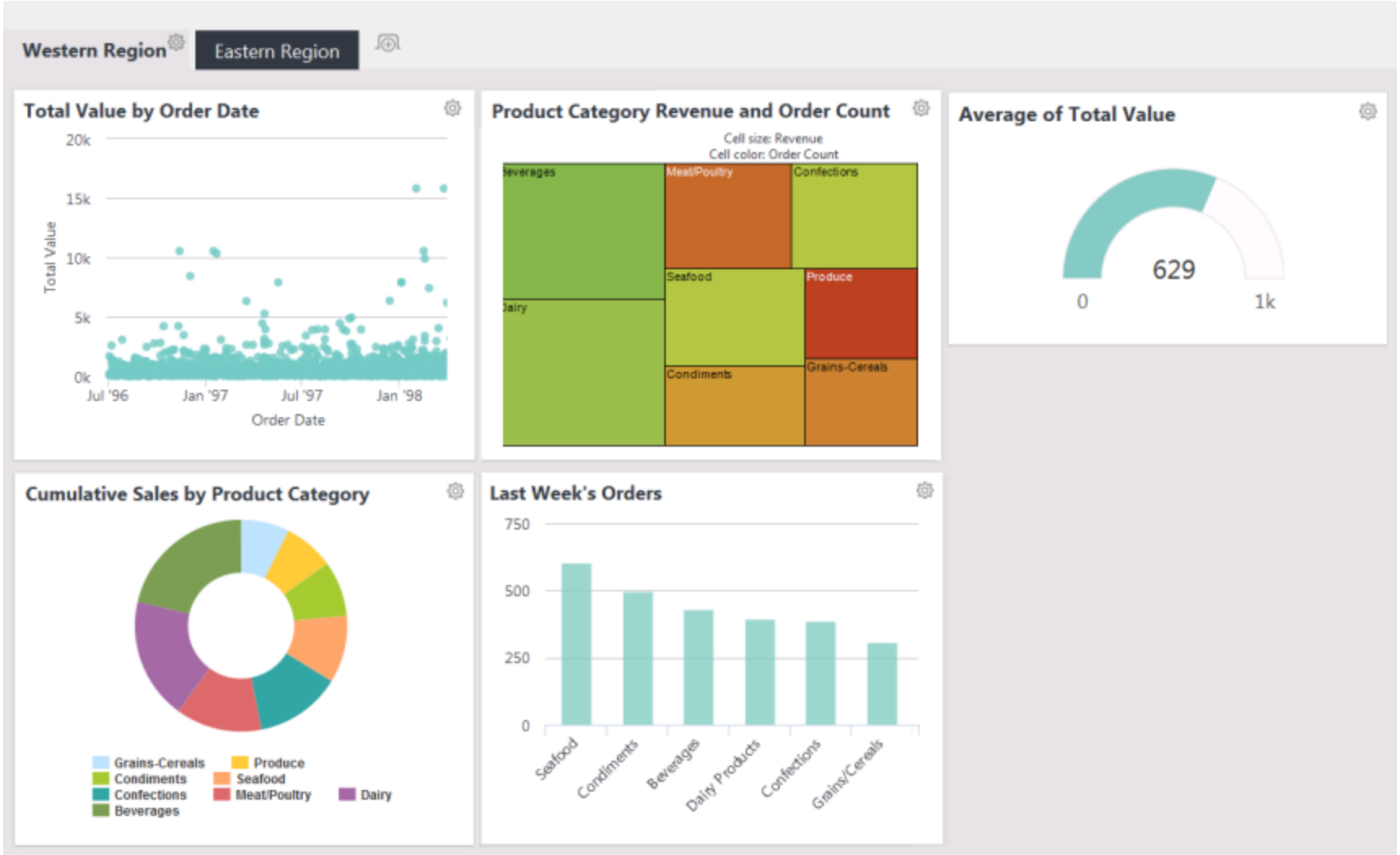
A Logi "Dashboard" is a collection of *Dashboard panels*, each containing a data visualization (table, chart, gauge, etc.). A Dashboard can be configured to allow end-users to customize it at runtime by adding, removing, or rearranging its panels. A Dashboard can also be configured to have tabbed pages, so that different collections of Dashboard panels can be viewed. Dashboard panel content is interactive and end-users can be allowed to vary the data criteria for them at runtime. Links within the panels

and visualizations can be used to drill-down and drill-through to supporting data. Dashboards present a very flexible way of presenting a range of information that's easy to take in with a single glance.



Category Name	Quantity
Beverages	559
Condiments	507
Confections	386
Dairy	393
Grains-Cereals	308
Meat/Poultry	165
Produce	100
Seafood	701

A simple Dashboard, with no user customization features, styled using the standard *Signal* theme, is shown above.

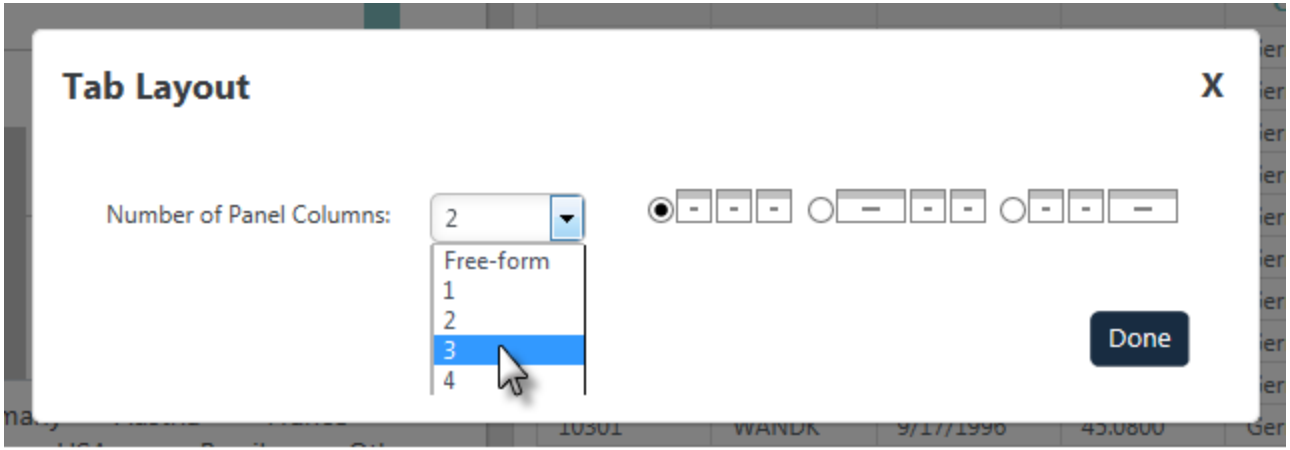


In the example above, we see a more complex Dashboard which includes tabs and has end-user customization enabled. Dashboards are created using the **Dashboard** element, and its child elements, and they provide attributes that control customization

Dashboard panels. There's also a mechanism for saving runtime customizations between sessions, on an individual-user basis.

Dashboard Layout

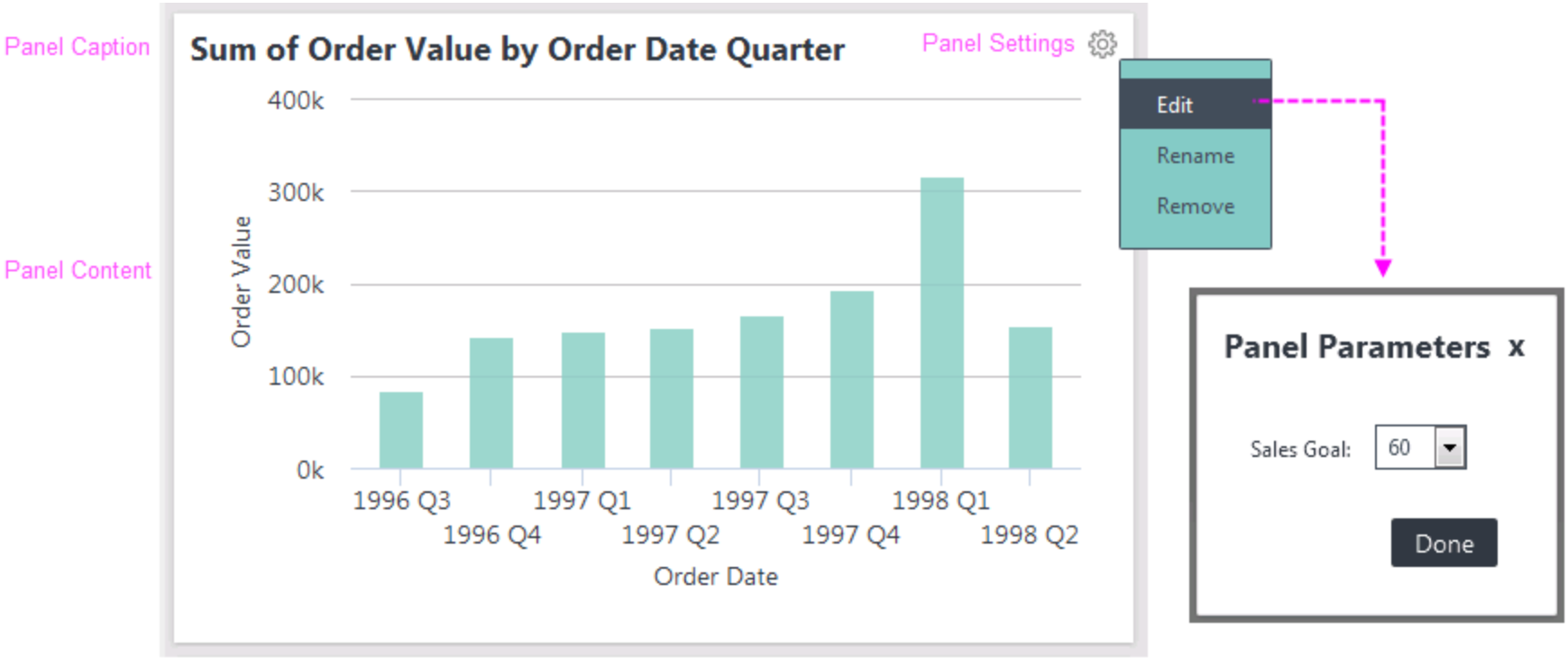
A Dashboard or tab's layout usually consists of one or more *columns*and, if the Dashboard is configured to be adjustable, users can drag-and-drop the panels into the columns and the arrangement of their choice.



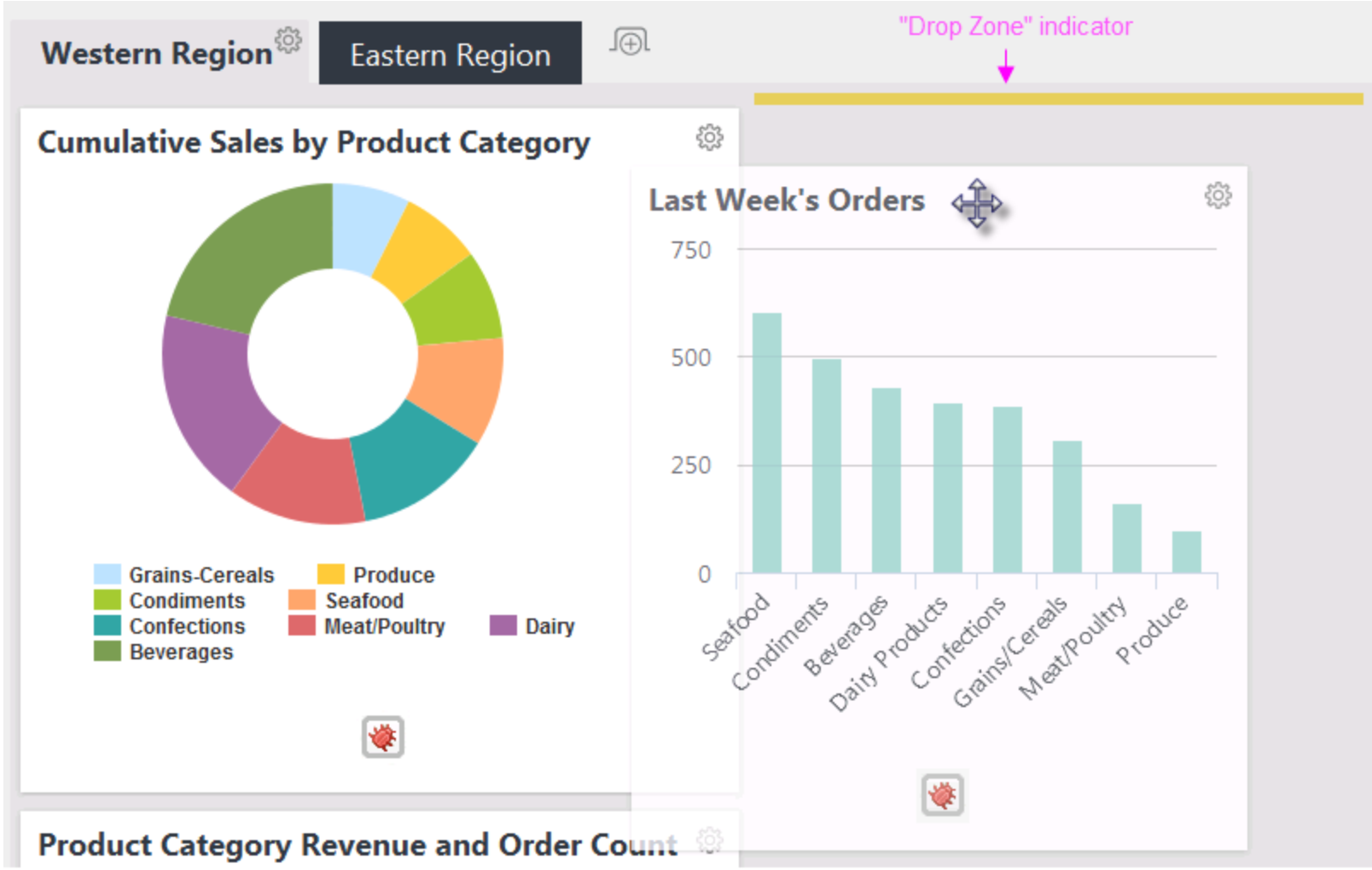
If the Dashboard is adjustable, then using the Dashboard or tab's gear icon and Settings menu, users can change the number of columns in a tab or Dashboard, as shown above. Depending on your application's configuration, a "Free-form Layout" option may also available. If it's enabled, users can resize and rearrange Dashboard panels without regard to any predetermined columns.

Dashboard Panels

Dashboard panels are the basic building blocks of the Dashboard and can be added programmatically or dynamically at runtime.



As shown above, panels have a **Caption** and **content**, and may be configurable by users at runtime in other ways. If the Dashboard is adjustable, a panel can have its caption changed or be removed from its Dashboard via its Settings menu. If the panel has been created with parameters, users can adjust the parameter values, affecting the panel content. In the definition, the **Panel** element creates a Dashboard panel. Its **Panel Content** and **Panel Parameters** child elements are used to create and configure the panel.



At runtime, users can drag a panel to a new location by clicking on its caption bar, as shown above (note the cursor change). As the panel is dragged into a different column or position, a yellow "drop zone" indicator appears to help snap the panel into its new location. In a columnar layout, panels will automatically resize based on the width of the column they're dropped into, and the

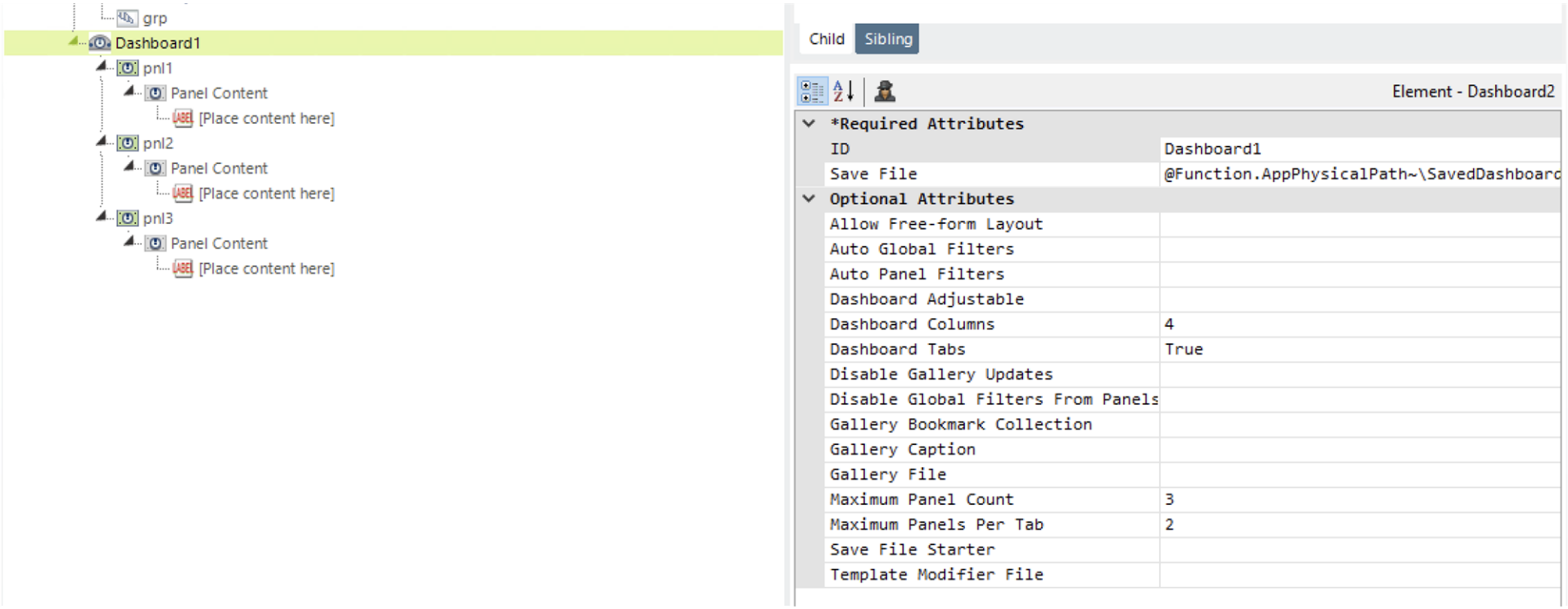
column's width is determined by the *widest panel* it contains. If the contents of a panel are widened, then the panel and its column will widen, too. *This is a change from panel behavior in previous versions!* In a columnar layout, panels will automatically resize based on the width of the column they're dropped into. Column width is a *fixed width*, determined by the layout chosen and screen width. Panel content will be automatically scaled up to the maximum width that fits in the panel. In support of this change, charts in panels in columnar layouts no longer have horizontal resizing handles.

The Dashboard Wizard

The "Create a Dashboard" wizard is one of Logi Studio's collection of wizards. This wizard assists developers in creating Dashboards by populating the report definition with the necessary Dashboard-related elements. More information about this wizard is available in our "Dashboard Wizard for Developers" on page 393.

Using the Dashboard Element

Only one Dashboard element is allowed per report definition. Four child elements, discussed in other topics, are used with the Dashboard element. *We do not recommend that you use this element within an embedded sub-report*.



Let's examine the Dashboard element's attributes:

Attribute	Description
Allow Free-form Layout	Specifies whether Free-form Layout mode is enabled. This layout mode allows Dashboard panels to be placed wherever desired by the user, instead of fitting into a columnar arrangement. When disabled, users

Attribute	Description
	Dashboard Columns attribute must be *0*. Default: *False.*
Auto Global Filters	Specifies whether Dashboard panels with Analysis Filters, and/or those generated from an Analysis Grid that uses the Active Query Builder element, will have global filters created automatically. Default: *False.*
Auto Panel Filters	Specifies whether Dashboard panels generated from an Analysis Grid that uses the Active Query Builder element will have panel filters created automatically, based on its metadata. Default: *False.*
Dashboard Adjustable	Determines if the user is allowed to add, remove and manage tabs and panels. Set this attribute to *False* for a static Dashboard that users cannot change. When set to *True,* the user's Dashboard changes are saved to the location specified in the Save File attribute. 💡 When set to *False*, the user cannot save changes, but the Save File is still used to control the Dashboard appearance. Therefore, set this to *True* during the development process so that you, the developer, can make desired changes. Then, if a static Dashboard is desired, set it back to *False*. Finally, be sure to deploy the actual .xml Save file with the reporting application to the production server. Default: *True*
Dashboard Columns	Specifies the number of vertical columns (1- 8) the Dashboard should be divided into. Columns are automatically sized to fit the available space. When Allow Free-form Layout is *True*, this attribute must be *0*. Default: *3.*
Dashboard Tabs	When *True*, allows the user to put Dashboard panels into different tabs. The user can create, rename, reposition and remove tabs. The initial tab is created automatically.
Disable Gallery Updates	Specifies whether users can delete panels at runtime from the gallery specified below in the Gallery File attribute. Default: *True*.

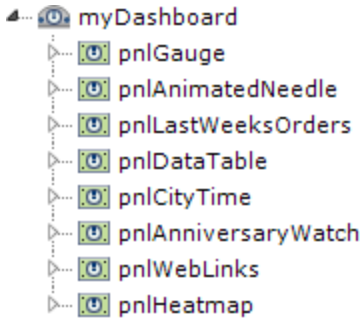
Attribute	Description
Disable Global Filters from Panels	When Auto Global Filters attribute set to *True*, the user is able to click certain types of visualizations, such as charts, to add or update a global filter. Setting the attributes to *True* will prevent those clicks from adding Dashboard global filters.
Gallery Book-mark Collection	Specifies a separate file/location to store the gallery items as bookmarks. The default (and recommended) value is the default bookmark collection of the application.
Gallery Caption	Specifies a caption for the gallery specified below in the Gallery File attribute. This caption appears in the list of galleries in the Add Panels pop-up panel.
Gallery File	Specifies the fully-qualified file path and name of an XML file where charts and tables created by users at runtime using an Analysis Grid can be accessed for use in the Dashboard. This file is independent of the Dashboard's Save File, allowing the visualizations to be shared and used by multiple Dashboards. For example, the attribute value might be: `@Function.AppPhysicalPath~\GalleryFiles\Gallery.xml`  When a Gallery File is specified here, *do not* add Panel child elements under the Dashboard element. A Gallery File is created by using an Analysis Grid with a Custom Dashboard Panels element. The latter element's Dashboard Save File attribute is set to the name of an XML file that becomes the Gallery File. This file will be created and updated as the user adds visualizations in the Analysis Grid. Gallery files can also be stored in a database, see *Storing Bookmark, Gallery, and Save Files in a Database*.
ID	(Required) Specifies a unique identifier for this element.
Maximum Panel Count	Specifies the number of panels that can display in a Dashboard. If there is no constant, or its value is empty, this function is disabled.

Attribute	Description
Maximum Panels Per Tab	Specifies the number of panels per Dashboard tab. If there is no constant, or its value is empty, this function is disabled.
Save File Starter	Specifies a file to be used as the initial Save File in cases when the Save File does not exist yet. For example, when the Save File is different for each user, the Save File Starter file can include a set of default tabs and panels. See "Creating a Default Panel Arrangement " on page 373 for more information.
Template Modifier File	The name of the template modifier file, if any. See "Customizing Dashboard Appearance " on page 389 for more information about template modifiers. The template modifier file can be in any folder accessible to the web application; if a fully-qualified file path is not provided in this attribute value, then the application expects it to be is your project's _SupportFiles folder.

Using the Panel and Panel Content Elements

The Dashboard **Panel** element is simply a required container for the other child elements and multiple panel elements are generally used to create a Dashboard.

If specifying a Gallery File in the Dashboard element attributes *do not* use Panel elements. See the Attributes table in "Using the Dashboard Element " on page 358 for more information.



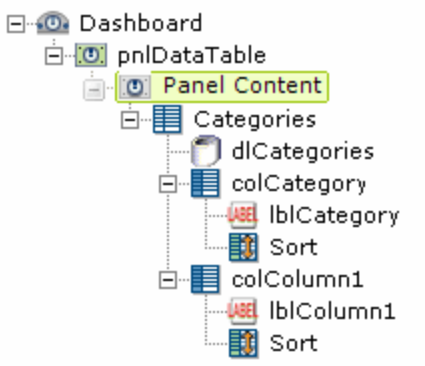
The presence of Panel elements means simply that the panels are available, but they will not be displayed in the Dashboard unless configured through the Dashboard configuration page to be visible. The Panel element has these attributes:

Attribute	Description
Caption	(Required) Specifies the text displayed in the panel's title bar, and used as a title in the Dashboard configuration page.
ID	(Required) Specifies a unique identifier for this element.

Attribute	Description
Allow Caption Rename	Specifies if the user is allowed to change the Dashboard panel caption. When *True*, the "Rename" item appears in the panel Settings menu. When selected, the caption can be edited in place then saved by pressing Enter or clicking outside of the text box. The default value is *True*.
Image	Specifies the file name of a thumbnail image that will appear on the left side of the Add Panels panel. For the best appearance, images should be less than 200 pixels wide. The image name can come from an image in the _SupportFiles folder or from a relative URL within the same Logi application.
Minimum Height	When the Dashboard has been configured for Free-form Layout, specifies the minimum height, in pixels, to which a panel can be resized. The default value, if left blank, is *100* pixels.
Minimum Width	When the Dashboard has been configured for Free-form Layout, specifies the minimum width, in pixels, to which a panel can be resized. The default value, if left blank, is *100* pixels.
Multiple Instances	Specifies whether multiple instances of a panel can be added to a Dashboard. When set to *True*, the Dashboard panel can be added multiple times to the same Dashboard or tab. This is especially useful for panels that have input parameters because the user can show different versions of the same panel with different parameters in the same Dashboard or tab.
Panel Description	Specifies the descriptive text for this panel that appears in the Add Panels panel.
Security Right ID	If entered, controls access to this element via Logi security. Supply the ID of a Right defined in the application's settings/security section. Only users that have a Role referenced in the Right will be able to see the element. (Be careful - when the Right is not defined in the settings, the element is visible.) Multiple Right IDs,

Attribute	Description
	separated by commas may be entered. In this case, the user will see the element if he has access to any one of the Rights.
Tooltip	Text that appears in a small balloon when the user hovers his mouse pointer over the panel's Title Bar.

The **Panel Content** element is the first child element of a Dashboard panel. This element is a container for the actual content to be displayed. Its only attribute, **Height**, is optional and used to set a fixed panel height. When left blank, the height is resized automatically to fit the contents. If a fixed height is set, a vertical scroll-bar appears when necessary if the content's height is greater than the fixed height specified.



As shown above, left, content can be **included as a sub-report**, or be included directly in the Panel Content element or, as shown above, right.

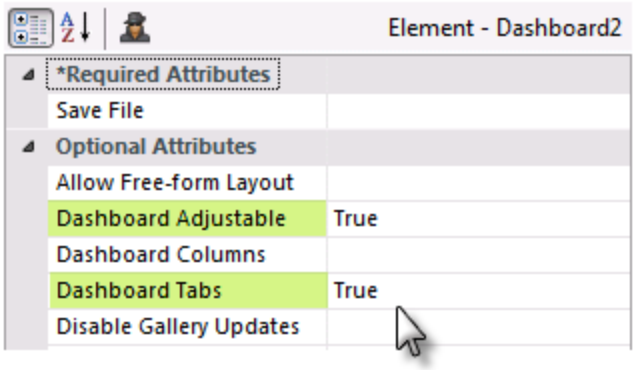
Inclusion of content **as a sub-report**, especially if there are many panels, is a *very desirable approach*: it keeps the main definition containing the Dashboard from becoming overly long and complex and allows panel content to be developed and tested independently before being integrated into the Dashboard. In addition, if you plan to *export* panel content, we highly recommend that you use sub-reports.

Dashboard panels are processed in separate threads at the web server, resulting in improved performance, especially for Dashboards with large numbers of panels. Each Panel Content element is an individual reporting environment and, as such, can have its own Default Request Parameters element, Local Data element, etc. If necessary, panels can be identified with the token @Function.InstanceID~. The **Load Panels Plugin Call** element allows developers to use a plug-in to access the underlying definition code for a Dashboard panel.

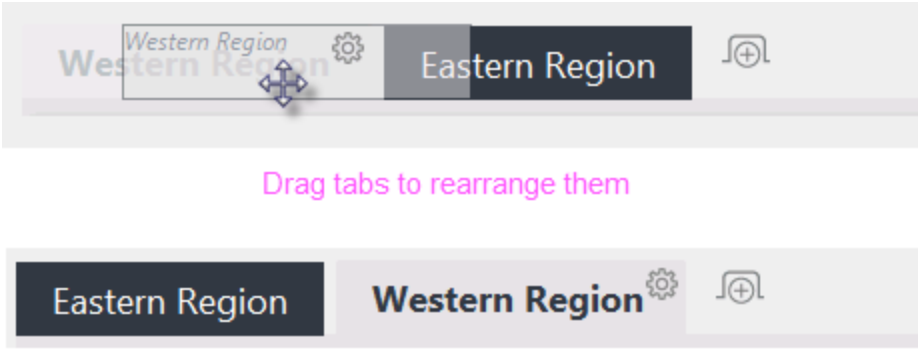
Working with Dashboard Tabs

A Dashboard can consist of a single collection of panels (a base page) or a series of tabbed pages, each with its own collection of panels. If a Dashboard has been configured to be *adjustable* and *to have tabs,* then users can add, remove, and rearrange the tabs at runtime.

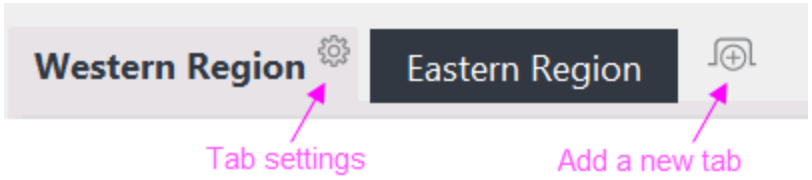
In the example below, the Dashboard element's attributes have been set to make it adjustable and to have tabs:



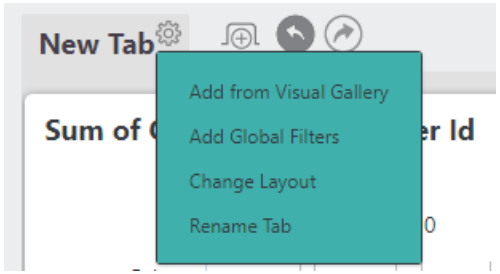
A Dashboard with tabs will have one tab created by default. In the example shown below, an additional Dashboard tab has been added. Tabs can be dragged with the mouse to rearrange them:



Adjustable Dashboards display this gear icon on the selected tab; click the **icon** to configure their settings. To add a new tab to the Dashboard, click the **plus** icon:



A tab's Settings pop-up menu, shown above, appears when its gear icon is clicked and allows you to add panels to the tab, change the number of columns being used in the tab, to rename the tab, and to remove the tab.



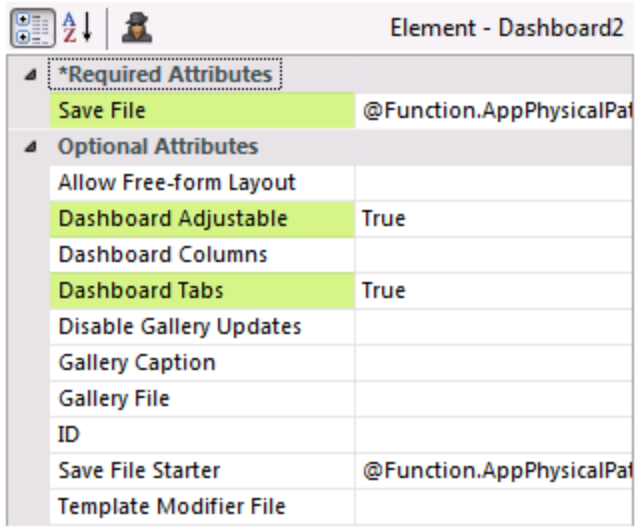
Saving Dashboard Settings

Given that users can potentially customize a Dashboard at runtime, it's useful to be able to save those customizations for future use. There are several ways to do this.

This topic contains the following sections:

Save Files

The **Save File** is an XML file that stores a record of the Dashboard changes users make at runtime. Its data is associated internally with IDs for the Dashboard and panels used to create it.



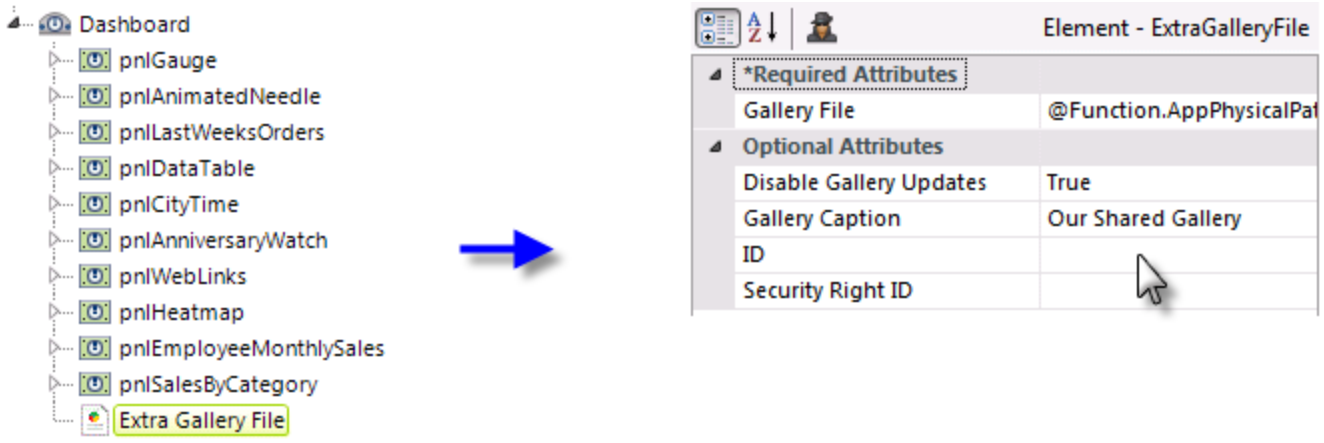
The Dashboard element's **Save File** attribute specifies a fully-qualified path and file name, as shown above, for the Save File. Tokens may be used in the path and file name. This creates the opportunity for reusing an individual user's configuration for their next session through the use of Logi Security, or a GUID-and-cookie scheme, tying the user name to the Safe File. For example:

```
C:\inetpub\wwwroot\myLogiApp\SavedDash\DashConfig.xml
@Function.AppPhysicalPath~\SavedDash\DashConfig.xml
@Function.AppPhysicalPath~\SavedDash\@Function.UserName~.xml
```

In the examples shown above, a special folder "SavedDash" has been created by the developer in the application folder to hold Dashboard Save Files and is shown as part of the Save File path. (Extra spaces in the examples are just there for reading clarity).

Gallery Files

A **Gallery File** contains the same data as a Save File and its attribute accepts the same values and tokens. However, unlike a Save File, it's not associated internally with a specific Dashboard and its panels. Instead, it's created using an Analysis Grid and can be copied, moved, and shared between Logi applications and users, providing an independent collection of visualizations as a resource. The **Extra Gallery File** element, a child of the Dashboard element, allows you to specify additional gallery files. When multiple gallery files have been specified and the user is adding panels to a Dashboard, he'll be able to select them from different visual galleries. The general Logi Info development rule that elements requiring an element ID be given *unique* IDs is very important when dealing with Extra Gallery Files. For example, if a Dashboard and a gallery file both refer to Dashboard panels that have the same element ID, an "duplicate panel" error will occur. How are extra gallery files useful? Imagine a scenario in which a gallery file of pre-configured "standard" panels is created and then shared with all users as an extra gallery file. They could then build Dashboards using panels from this standard gallery and/or from their personal gallery.



The Extra Gallery File element, shown above, requires a fully-qualified file path and name, with .xml extension, to the file. Other optional attributes allow you to specify whether users can modify the panels in the gallery (Disable Gallery Update) and specify

the name that will appear in the list of gallery choices (Gallery Caption) that will appear in the Add Panels pop-up panel. The element also has a Security Right ID attribute, so access to gallery files can be controlled using security rights. Multiple Extra Gallery File elements may be used.

Using a Report Definition as an Extra Gallery File

There's also another interesting way to share gallery visualizations: the Extra Gallery File element's **Gallery File** attribute can be configured to point to a regular Logi report definition (.lgx) file. If that definition file includes a Dashboard element with child **Panel** elements containing visualizations, the application will locate those panels in the definition and also make *them* available in the Visual Gallery.

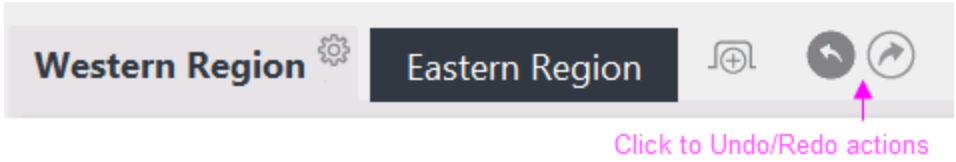
Do not use definitions with Panels that get their data using *linked datalayers* as Extra Gallery Files. Linked datalayers used to create the content of Panel elements in this scenario are not recognized and the content will not be available. If possible, replace the linked datalayers with regular datalayers in order to make definitions work as Extra Gallery Files.

Bookmarks

A Bookmark for a report page that includes a Dashboard, when created manually using **Action.Add Bookmark**, stores Request and Session variables and a *reference* to the Dashboard's Save File. It does not *contain* the Dashboard configuration data itself. For more information, see *Bookmarks*.

Manual bookmarks can be created independently of updates to the Dashboard configuration (and, therefore, updates to the Save File). So running a bookmark for a page containing a Dashboard may not necessarily reproduce the Dashboard configuration that existed at the time the bookmark was created. Automatic Bookmarks (see *Adding Automatic Bookmarks*), created using the **Auto Bookmark** element, on the other hand, will automatically create a bookmark *and* the bookmark equivalent of a Save File

whenever the Dashboard configuration is changed, giving you the best of both worlds. Do not specify a Save File value when using this child element.

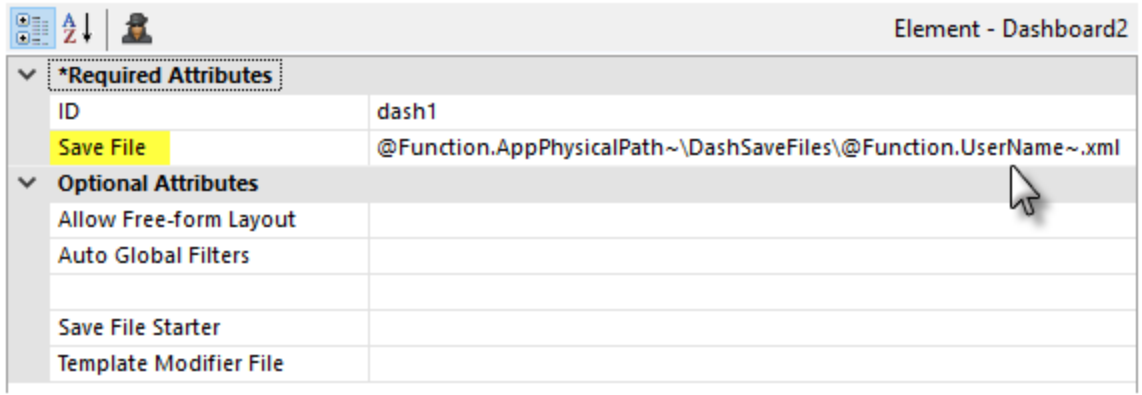


When Automatic Bookmarks are in use, Undo and Redo icons will appear beside the tabs, as shown above. You can also store your Bookmarks, Save files, and Gallery files in a SQL database. See *Bookmarks* for more information.

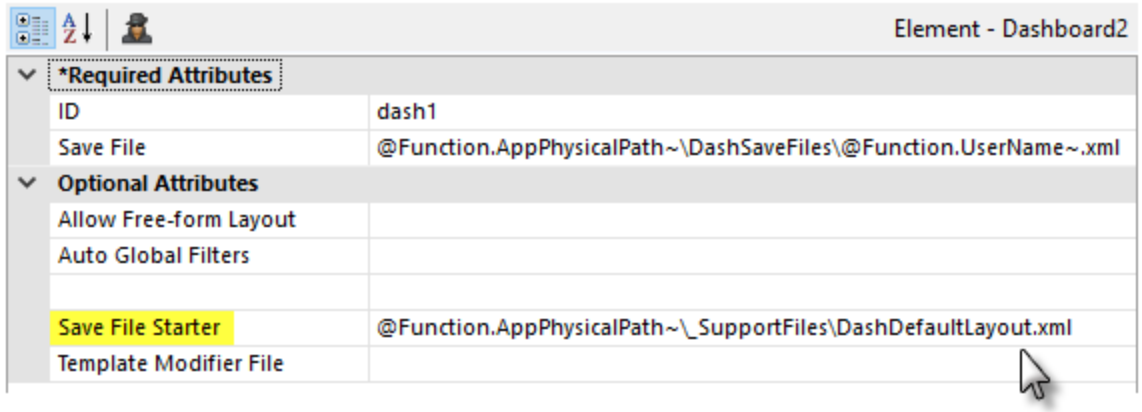
Creating a Default Panel Arrangement

You may want all of your users to start using the Dashboard with the exact same, pre-defined tab and panel arrangement. A default tab and panel arrangement can be saved for later use by working with the Dashboard element attributes **Save File** and **Save File Starter**. As mentioned earlier, the "Save" file specified in the **Save File** attribute value is created automatically the first time the user changes the Dashboard configuration. It records any configuration changes a user makes to the Dashboard, such as which panels are included, panel arrangement, and tabs. When the user returns to use the Dashboard in a subsequent session, this file is used to automatically configure the Dashboard. As mentioned earlier, this file can be identified on a per-user basis, providing each user with their own personal arrangement every time they use the Dashboard. If a file is specified in the **Save FileStarter** attribute, and no Save File has been created yet (because this is the first time the user is using the Dashboard), then the Save File Starter file will provide the initial Dashboard configuration. This allows each user to begin with a standard initial Dashboard configuration. Follow these steps to create a default panel and tab arrangement:

1. Create your Dashboard and panels in Logi Studio, as usual.



2. Configure the Dashboard element's **Save File** attribute with a value that will create a separate file for each user, as discussed earlier and shown above. The example above demonstrates the use of tokens in the value and assumes Logi Security is in use.
3. Run the application and, in your browser, create and arrange the tabs, panels, and panel parameters (if any) desired for your default arrangement. Close your browser.
4. In the file system, find the Save file that was just created during your session, and copy it to the `_SupportFiles` folder. Rename the copy using an arbitrary but meaningful name, such as `DashDefaultLayout.xml`.



5. Back in Studio, configure the Dashboard element's **Save File Starter** attribute with the complete path and file name of your renamed Save file copy, as shown above.

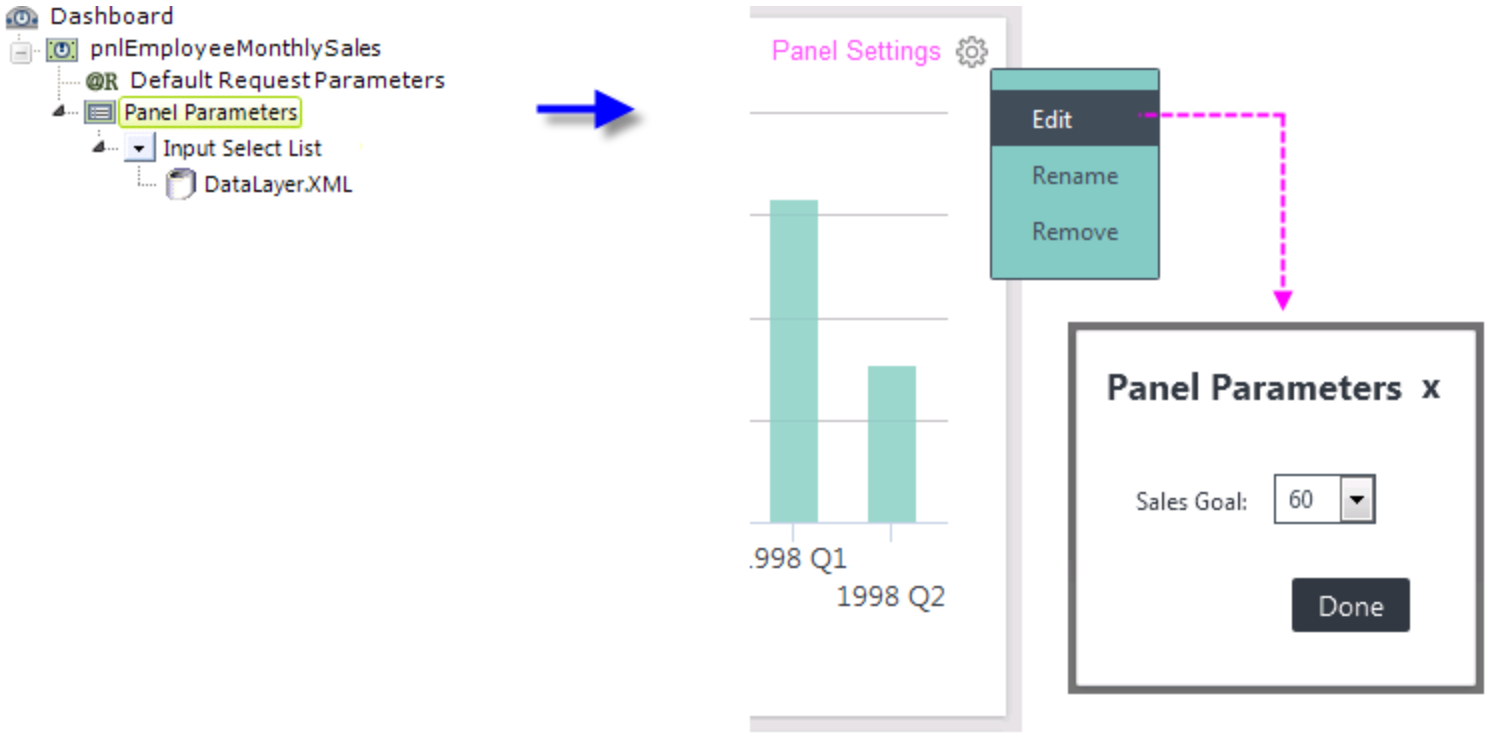
All first-time users will now see the default Dashboard arrangement, and any customizations they're allowed to make will be saved for their next session.

Using Analysis Filters with Dashboards

The **Analysis Filter** super-element can be used with Dashboard panels to provide individual panel or global data filtering. Global filtering allows all visualizations in a Dashboard tab that use similar data to be updated based on a single filter the user can adjust at runtime. Two related Dashboard element attributes, **Auto Global Filters** and **Auto Panel Filters**, are available. They're used when Dashboard panels are included that were saved from an Analysis Grid that used the Active Query Builder. Filters can be created based on the metadata for the panel data. Detailed information about this element, including how to implement it in Dashboard panels, is available in *The Analysis Filter*.

Using the Panel Parameters Element

It's often desirable to let users change Dashboard report criteria at runtime. This can be done using user input elements that are included within the panel *content* definition, or by using the **Panel Parameters** element, which allows the data entry feature to be de-coupled from the panel content. In this way, user input elements are placed into a special pop-up panel. This approach is useful as it prevents user input controls from permanently taking up valuable screen real estate.



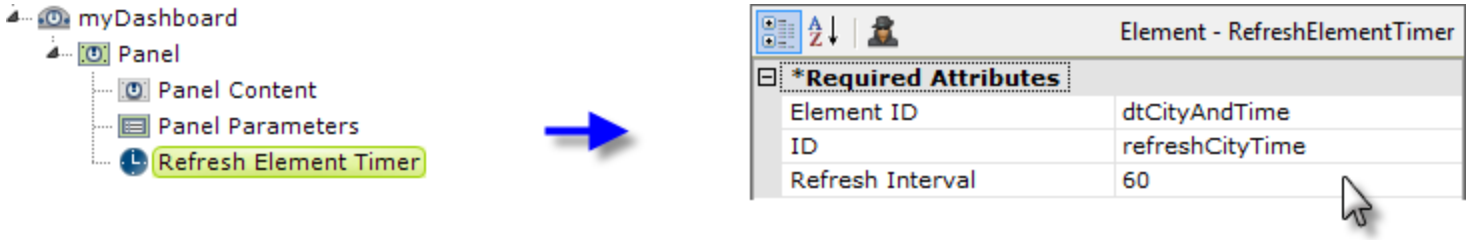
The images above illustrate how the Panel Parameters element is used in a report definition and how it looks on the screen. Notice the gear icon in the upper right hand corner of the panel images:

- Click the gear icon to display the Settings menu**.**
- Click the **Edit** item to get the Panel Parameters panel to pop-up.
- Select or change the user input controls as desired. Click **Done** to close the panel, write the user input values to the Save File (discussed earlier) and refresh the panel's content using the new values.
- Or, click **X** to close the panel. If the Panel Parameter element's **Refresh For Cancel** attribute is *True*, the Dashboard panel will be refreshed using the original values (i.e. no change); if the attribute is *False* (the default), the new values are used.

The next time the application is run, the user input values are retrieved from the Save File and presented to the Panel as @Request variables. This allows the content definition to use them, for example, by embedding @Request.*variable*~ tokens in a DataLayer.SQL statement. If you're using a **sub-report** for your content, you will need to pass these Request variables to it, using **Link Parameters**.

Using the Refresh Element Timer

Dashboards are often used to display **time-sensitive** or **transactional** data and it's very useful to be able to update the Dashboard panel contents periodically and automatically. This can be accomplished using the **Refresh Element Timer** element. This element uses AJAX techniques to automatically refresh one or more elements in a panel's content using an **a**djustable time interval.



The attributes for this element (all required) include:

- **Element ID** - Specifies the ID of the element that you want to periodically refresh. Multiple IDs, separated by commas, can be entered but the **scope** of each timer element is **limited** to its own panel. Multiple elements can also be refreshed by placing them under a Division element and specifying the Division element ID here. *Do not* specify elements that are children of Data Table elements (datalayers, columns, etc.)
- **ID** - Specifies a unique element ID for this element.
- **Refresh Interval** - Specifies the amount of time, **in seconds**, between refreshes of the element specified in Element ID. *Use caution* when specifying a very short interval if a large number of users will access this Dashboard: database server performance could be seriously impacted.

When there's a Data Table in the report being refreshed and it has **Interactive Paging** or **Sorting** elements, set the Data Table element's **AjaxPaging** attribute to *True*. The Refresh Element Timer element works by **re-running** the panel's report with just

those elements that are to be refreshed. The report runs using request parameters from 1) Input elements on the report, 2) LinkParams and 3) URL request parameters when Request Forwarding = *True*. When debugging has been turned on for an application, you can view **debug information** generated for each Refresh Element Timer request. These appear as additional links at the bottom of the Debug Trace page. For more information, see "Debug Reports" on page 56.

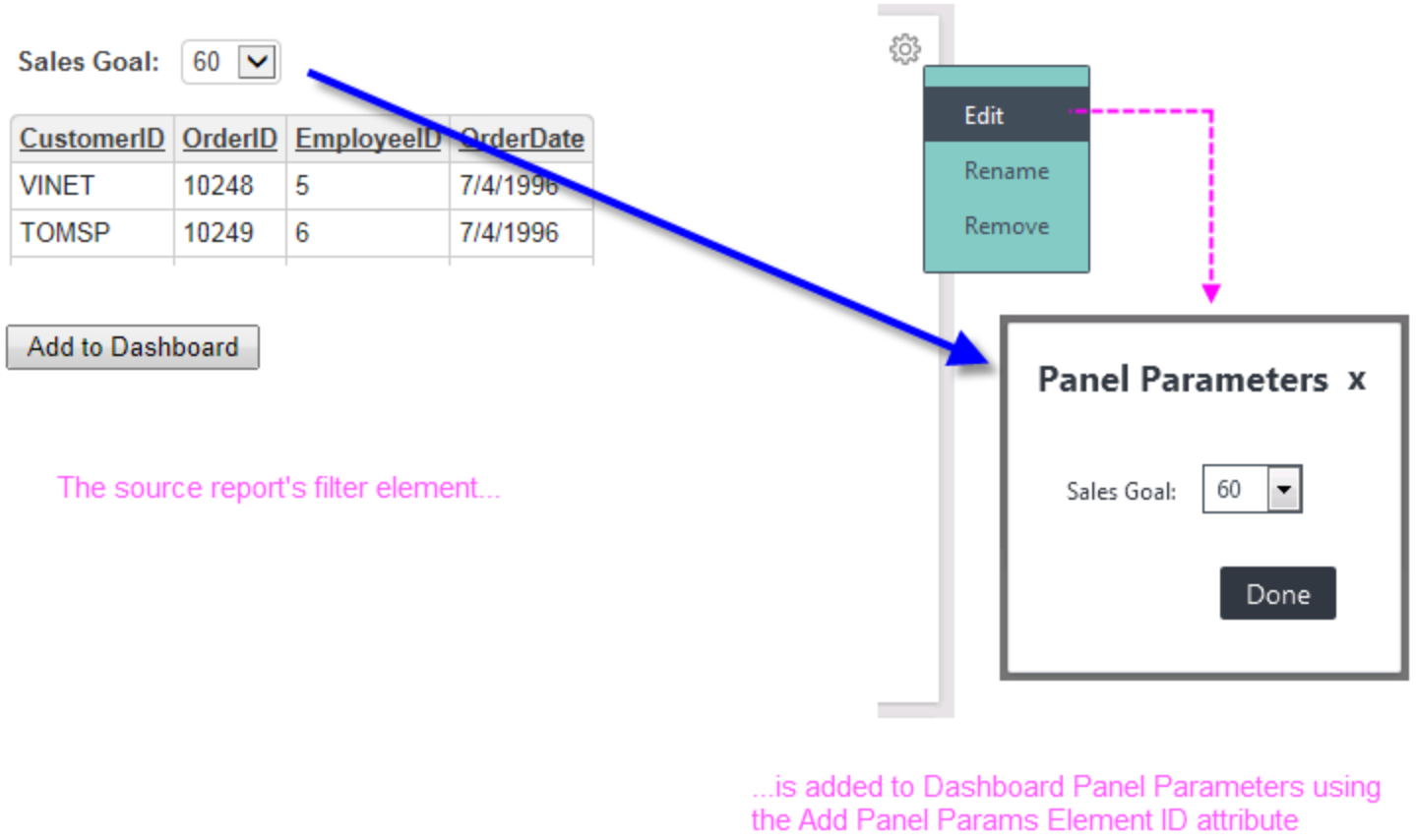
Using Action.Add Dashboard Panel

The **Action.Add Dashboard Panel** element, used in other definitions with elements like Data Tables and Analysis Grids, adds specified content to a new Dashboard panel which is added to a specified Dashboard. A bookmark for the content can optionally be created by adding a **Bookmark Linkback** element beneath this action element. Essentially the element works by allowing the developer to specify all of the necessary content in the current report that will be copied into the new Dashboard panel. This includes the properties of the panel itself and any relevant parameters. On execution at runtime, the report definition containing the specified Dashboard, and its supporting files, are edited to include the new Dashboard panel and its content.

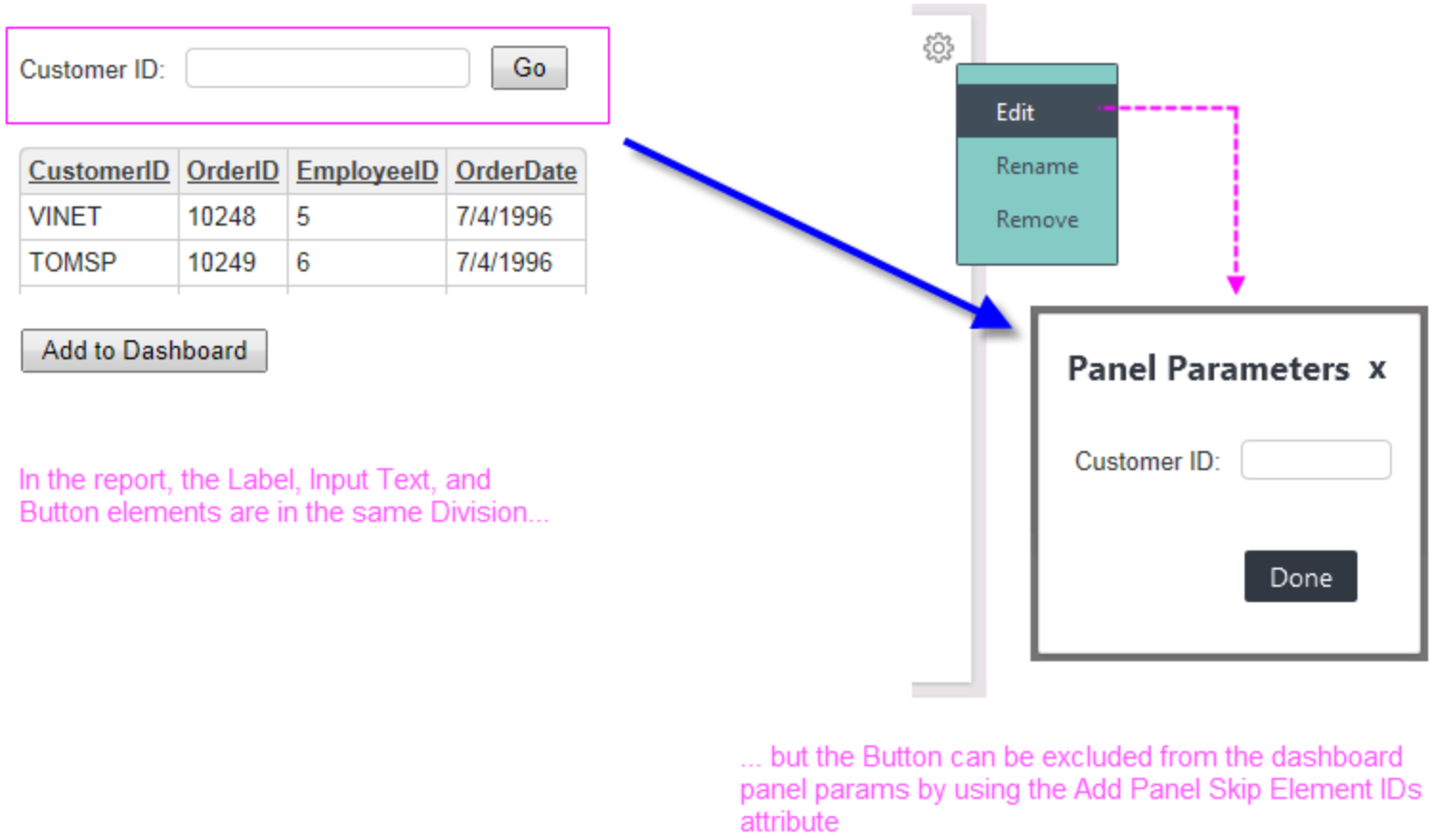
At runtime, when a user clicks on this element's parent element (Button, Label, Image, Event Handler, or Popup Option), a pop-up panel, shown above, is displayed soliciting a title and a description for the new Dashboard Panel, which is used in the Dashboard Configuration Page. Optional attributes allow default information, and a thumbnail image, to be specified by the developer.

Parameters

Reports often include input elements that allow users to filter the displayed data at runtime.



Action.Add Dashboard Panel includes a mechanism for copying these elements into the new Dashboard panel as Panel Parameters, as shown above. The **Add Panel Params Element ID** attribute allows you to specify the IDs of the elements to be added as parameters.



You can specify the ID of a **Division** or other container element and *all* of its contents will be added to the panel parameters, with the exception of any elements specified in the **Add Panels Skip Element IDs** attribute. This allows you to exclude, for example, a Submit button that's in a Div, along with Input elements, in the current report but which won't be needed in the Dashboard parameters panel (which has its own Done button).

Token Handling

In anticipation of the need at runtime to "pass along" some tokens (i.e. don't resolve them now) associated with the new content into the new Dashboard panel (where they will be resolved when the Dashboard definition is run), a special token handling scheme has been provided:

- In general, @Function tokens will *not* be resolved, but will be passed along with the content into the panel for resolution when the panel is viewed. This was done in anticipation that these tokens will most likely be used to provide date/time- or user-related information.
- @Local tokens, *specified by the developer* in the **Add Panel Local Data IDs** attribute, will *not* be resolved and will be passed along. Multiple tokens can be entered in a comma-separated list. Tokens not listed will be resolved before the panel is created.
- @Request tokens, *specified by the developer* in the **Add Panel Request IDs** attribute, will *not* be resolved and will be passed along. Multiple tokens can be entered in a comma-separated list. Tokens not listed will be resolved before the panel is created.
- @Session tokens, *specified by the developer* in the **Add Panel Session IDs** attribute, will *not* be resolved and will be passed along. Multiple tokens can be entered in a comma-separated list. Tokens not listed will be resolved before the panel is created.

Security

If a value is entered in the **Add Panel Security Right ID** attribute, access to the resulting Dashboard panel access can be controlled with Logi Security. This value should be ID of a security Right defined in the application's _Settings definition, under the Security element. This value is written into the panel's Security Right ID attribute and, with Logi Security enabled, only users with that Right will be able to see the new Dashboard panel. Note these two special conditions:

- When user rights come from **Right From Role** elements under the Security → User Rights elements, and this element's Security Right ID *does not* match any of the Right IDs defined in the Settings definition, then the user *does* have access.
- But when rights are instead derived from **Rights From DataLayer** or **Rights From Roles** elements, and the user *does not* have a matching Right, then the user *does not* have access.

Multiple Right IDs can be entered in a comma-separated list. The user will see the Dashboard panel if he has any one of the Rights.

Attributes

The **Action.Add Dashboard Panel** element has the following attributes:

Attribute	Description
Add Panel Content Element ID	(Required) Specifies the ID of the element that will be added to the new Dashboard panel. Only one ID is valid, multiple IDs are *not* allowed. However, you can specify the ID of a **Division** or similar container element and *all* of its contents will be added to the new panel.
Dashboard Save File	(Required) Specifies the Dashboard that will become the parent of the new Dashboard panel. Provide a fully-qualified web server file path and file name, with .xml extension, for the target Dashboard's existing Save File. Hint: This can be copied directly from the Dashboard's Save File attribute.
ID	(Required) Specifies a unique element identifier.
Add Panel Button Caption	Specifies the text inside the button, in the popup panel, that triggers the addition of the target content as a new panel in the specified Dashboard. Default: *Add to Dashboard*

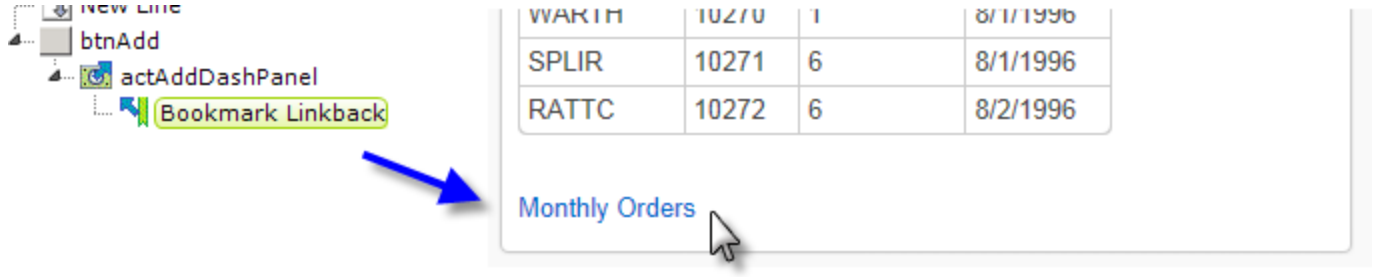
Attribute	Description
Add Panel Content Height	Specifies the height of the new Dashboard panel content, in pixels.
Add Panel Description	Specifies the new panel's default description, which will appear in the Dashboard Configuration Page. Users can change this text prior to executing the process.
Add Panel Local IDs	Specifies @Local token IDs whose values should be passed into the Dashboard panel. Token IDs listed here are normally saved in the Panel in a Local Data element with a static datalayer. In this way, when the Dashboard is run, the @Local tokens get replaced with the same values from when the panel was added. To have the Local Data datalayer re-run when the Dashboard is run, set the Add Panel Local Data IDs attribute to the ID of the Local Data element. In this way, the @Local values can change over time. @Local IDs listed will remain in the panel to be resolved by the Local Data defined either in the Dashboard or copied to the Dashboard panel via the Add Panel Local Data IDs attribute. Multiple token IDs can be entered as a comma-separated list.
Add Panel Local Data IDs	Specifies the element IDs of Local Data elements to be copied into the Dashboard panel. The Local Data datalayers will be run when the Dashboard is run, replacing @Local tokens. Multiple Local Data IDs can be entered in a comma-separated list.
Add Panel Params Ele-	Specifies the ID of the element in the current definition that will be added to the new panel's child Dashboard

Attribute	Description
ment ID	Panel Parameters element. This is used to include Input elements and supporting data in the panel, accessible using the panel's Edit button. This can be the ID of a Division or other container element, causing all of its child elements to be included. Some of those child elements can be excluded by specifying their IDs individually in the Add Panel Skip Element IDs attribute.
Add Panel Popup Caption	Specifies the title text shown in the popup panel displayed when the user clicks the Action.Add to Dashboard Panel element's parent element.
Add Panel Request IDs	Specifies the IDs of @Request tokens that will be passed along, unresolved, to the new panel. These tokens will be resolved in the new panel's Default Request Params or Dashboard Panel Parameters. @Request tokens not listed here will be resolved when the panel gets saved into the Dashboard. Multiple token IDs can be entered as a comma-separated list.
Add Panel Security Right ID	Specifies the security Rights a user must have in order to access the new Dashboard panel. Multiple Right IDs can be entered as a comma-separated list. Please see the *Security* section above for special conditions.
Add Panel Session IDs	Specifies the @Session token IDs that will be passed along, unresolved, to the new panel. @Session tokens not listed here will be resolved when the panel gets saved into the Dashboard. Multiple token IDs can be entered as a comma-separated list.
Add Panel Skip Element IDs	Specifies the ID of one or more child elements of the element named in Add Panel Params Element ID that will be excluded from being added to the new Dashboard Panel Parameters. Multiple element IDs can be entered as a comma-separated list.

Attribute	Description
Add Panel Title	Specifies the new Dashboard panel's default caption. Users can change this text prior to executing the process.
Image	Specifies the file name of an image that will appear on the left side of the Add Panels section of the Dashboard Configuration page. For the best appearance, images should be less than 200 pixels wide. The file name can be selected from a list of the images in the _SupportFiles folder or be entered manually as a relative URL to a file within the *same* Logi application. URLs to external sites or applications are *not* valid.
Multiple Instances	Specifies whether or not the new Dashboard panel can be added multiple times to the same Dashboard or Dashboard tab. Default: *False*
Security Right ID	Specifies the security Rights a user must have in order to use this Action element. Users without Rights that match Rights specified here will not be able to execute the action. Multiple Right IDs can be entered as a comma-separated list.

With Bookmark Linkback

The optional **Bookmark Linkback** element can be added as a child of Action.Add Dashboard Panel:



As shown above, once you include the element and configure a few attributes, it will automatically add a link to the new Dashboard panel which will run the target content with the request variables that were used in the original report. You must provide the name for the **Bookmark Collection** (usually associated with a user) and provide a **Caption**, which is the text that appears as the bookmark link in the new Dashboard panel. You can optionally decide to "run" the bookmark in a new window or selected window. For more information about this feature, see *Bookmarks*.

Customizing Dashboard Appearance

Dashboard appearance can be changed most easily by applying a theme to the report definition. Most of the screen shots in this document were taken with the *Signal* theme applied. You can create your own custom theme, based on a standard theme, using the tool.

This topic contains the following sections:

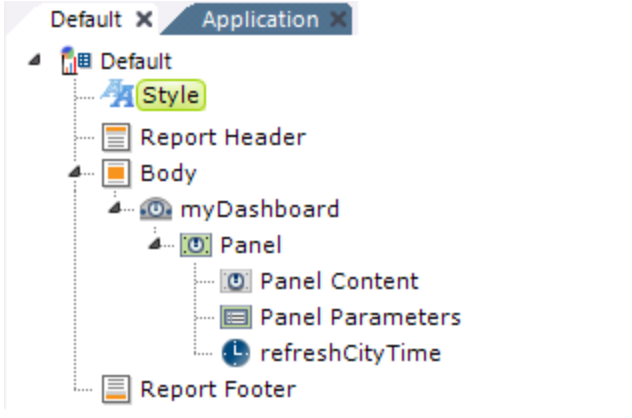
Changing Appearance Using Style Classes

Dashboard appearance can also be customized using **style classes** and Logi Studio provides the following standard style sheet for all Dashboards:

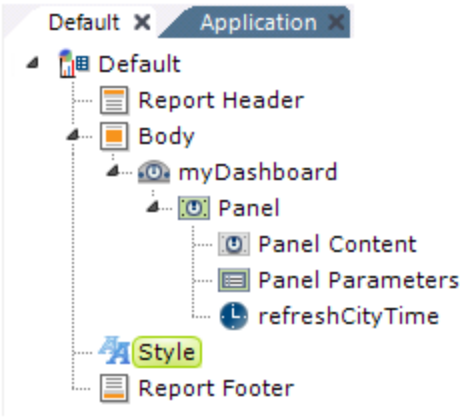
```
<yourAppFolder>\rdTemplate\rdDashboard\rdDashboard2.css
```

Developers can override classes in this style sheet by *copying* them into their own application style sheet and modifying them there.


Do not make changes in the standard style sheet in the rdTemplate folder! If you're using a Theme, it will override some of the Dashboard classes.



Custom style classes may not override defaults



Custom style classes *will* override defaults

As shown above, to ensure that style classes cascade correctly, in your report definition your Style element should be *lower* in the element tree than the Dashboard element (the Style Sheet element has to be processed *after* the Dashboard element in order to override the default classes). Here are some examples of specific classes, with their default values, that can be overridden by adding them to your project style sheet and then setting additional elements:

Change font size of all panel captions:

```
.rdDashboardTitleCaption {
 font-size: 14pt;
}
```

Change pop-up parameters panel caption:

```
.rdPopupPanelTitleCaption {
 font-size: 14pt;
}
```

Change color of the pop-up parameters panel "Done" button text:

```
.rdPopupPanelTitleCaption {
color: Red;
}
```

Many of the class names needed to override Dashboard styling can be discovered by using your browser's development tools (press F12 in many browsers).

Changing Appearance Using Template Modifiers

The Dashboard element uses a "template file" to define certain element properties that are not otherwise available as attributes to the developer for modification. These include language- and culture-specific Caption attributes that you may want to change for locale-based reasons (or you may simply want to change the captions to better suit your application). The Dashboard element's **Template Modifier File** attribute identifies a custom XML file developers can create containing elements that will override the same elements in the template file. For example, the Dashboard template file:

```
<yourAppFolder>\rdTemplate\rdDashboard\rdDashboard2Template.lgx
```

contains several Label elements. One of them has an ID = "lblAddPanelsTitle"; this controls the title in the Add Panels selection list (shown earlier). The title for that selection list can be modified by changing the Caption associated with that Label element. To change the title from its default "Add Panels" to "Add New Dashboard Panels", create your own XML file, identify it in the Template Modifier File attribute, and add this code to it:

```
<TemplateModifier>
 <SetAttribute ID="lblAddPanelsTitle" Caption="Add New Dashboard Panels" />
</TemplateModifier>
```

You can set the attributes for any number of elements in this file; examine the rdDashboard2Template.lgx file to learn the ID and Caption attributes available. The template modifier file can be in any folder accessible to the web application; if a fully-qualified file path is not provided in the Template Modifier File attribute value, then the application expects it to be is your project's _SupportFiles folder. More detailed information about template modifier files can be found in "Template Modifier Files" on page 426.

Dashboard Wizard for Developers

This topic introduces Logi Info *developers* to the **Dashboard Wizard** in Logi Studio. Dashboards provide an excellent method of presenting business intelligence in a discrete but condensed manner and Studio's wizard makes creating them easy.

The following sections are covered in this topic:

About Dashboards

A "Dashboard" is a **collection of panels** containing Logi reports, which in turn contain table, charts, images, etc. At runtime, the user can customize the Dashboard by rearranging these panels on the browser page, by showing or hiding them, and even by changing their contents using adjustable reporting criteria.

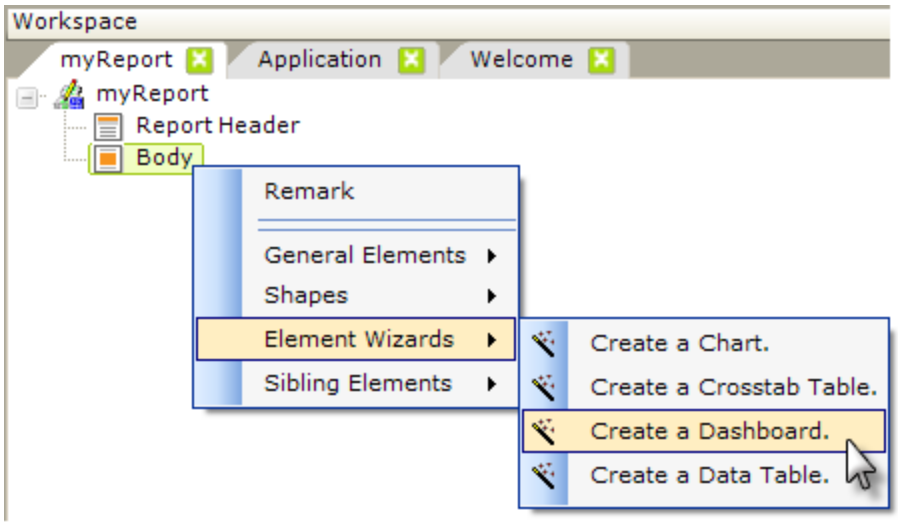
Collections of panels can also be arranged in **tabbed panels** within the Dashboard, allowing grouping. The reports displayed in Dashboard panels are regular Logi reports, sized to fit the panel space.

As shown above, Dashboards are frequently used to give users a broad view of **a variety of information** at once. The data displayed within the panels can be configured, as in any Logi report, to link to other reports, providing "drill-down" functionality.

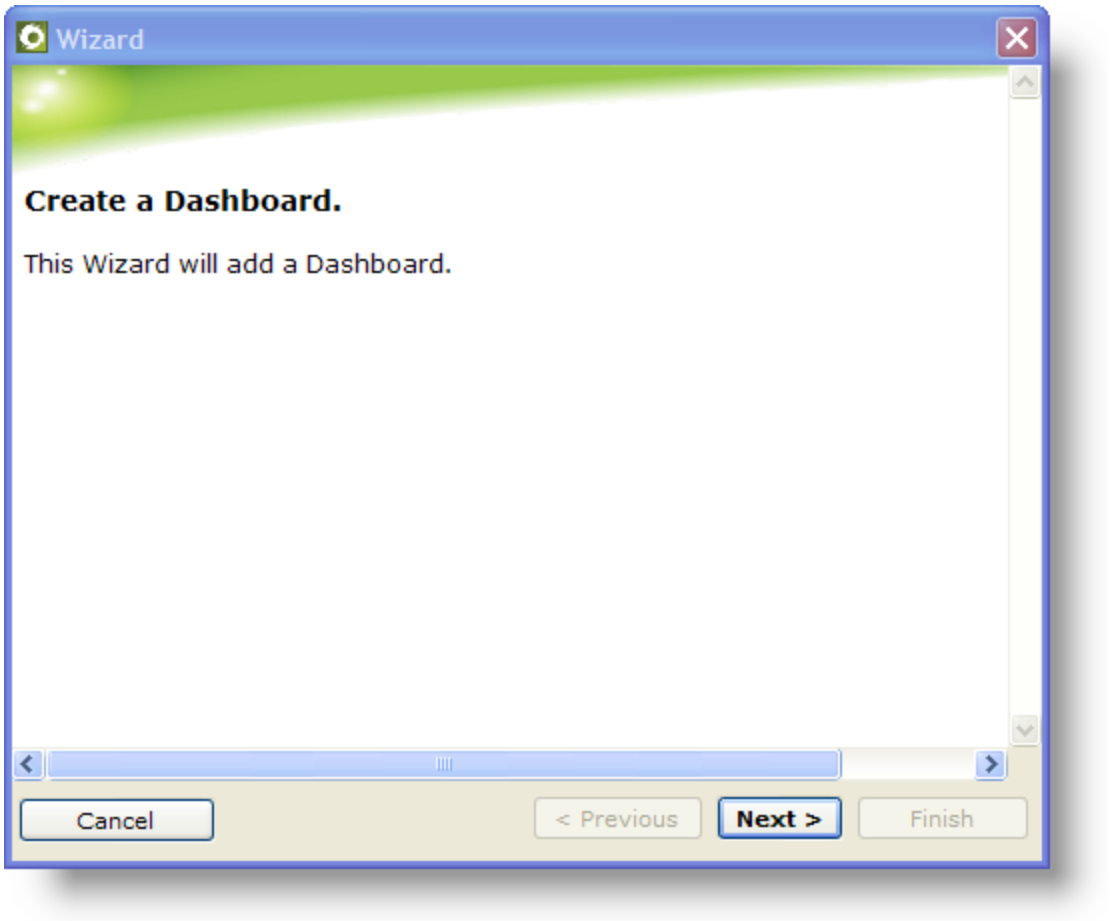
For more detailed information about the Logi Dashboard element, see "Logi Info Dashboard " on page 351.

The Dashboard Wizard

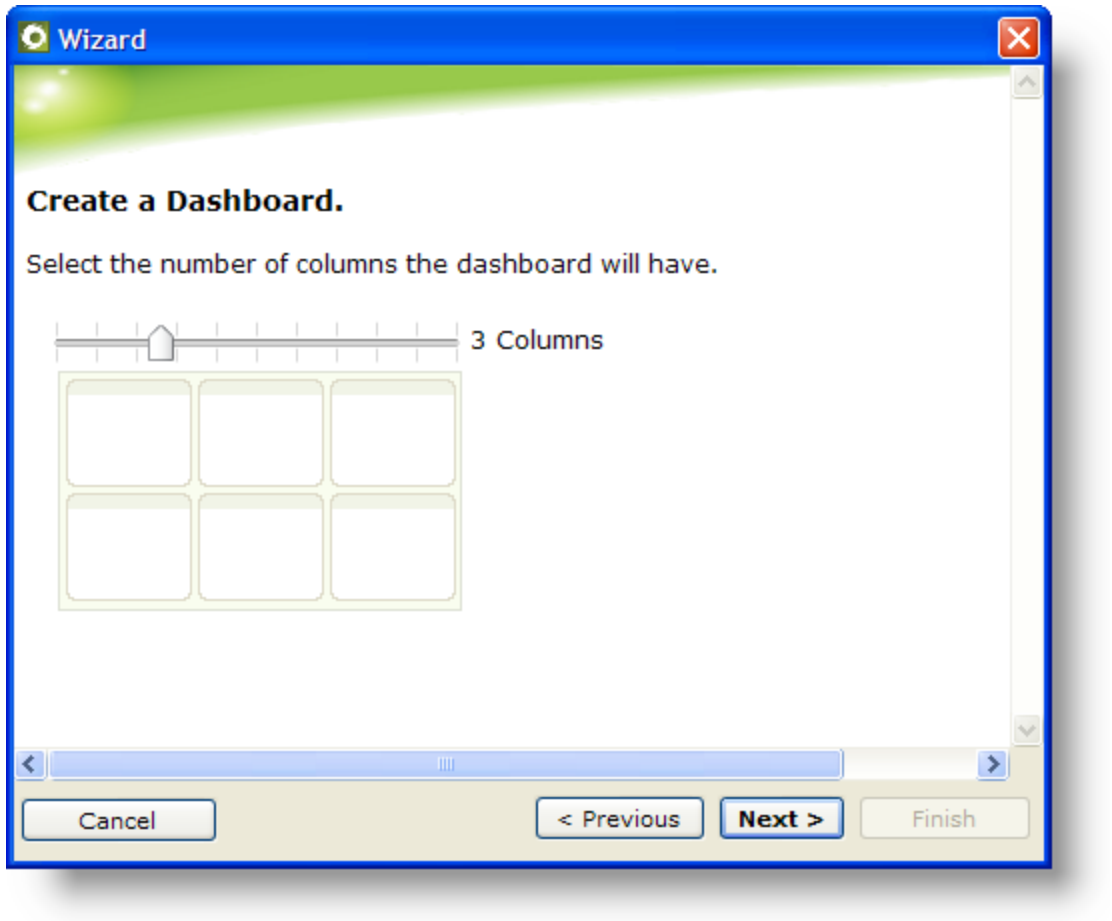
Logi Studio includes a wizard to assist developers in the creation of Dashboards:



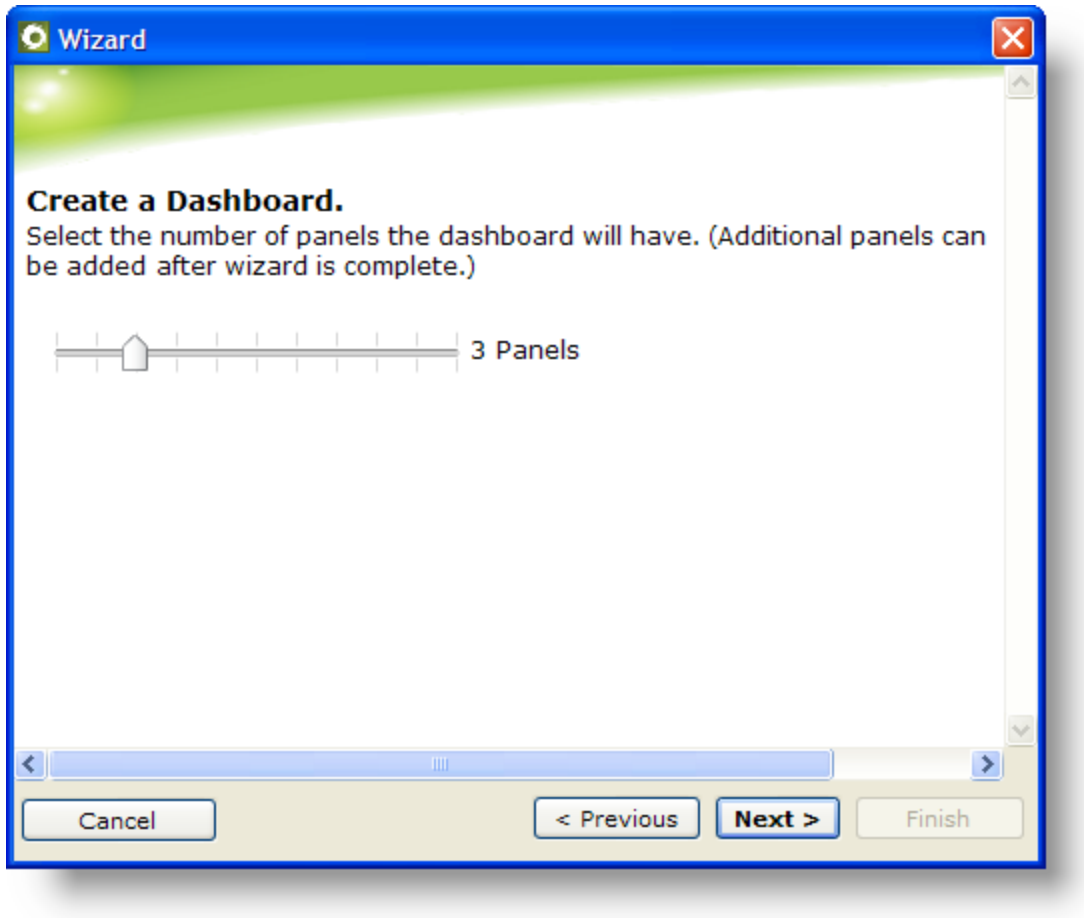
The wizard can be started by right-clicking any valid Dashboard parent element and selecting it from the context menu, as shown above. The option is also available in the Wizards folder of the Element Toolbox.



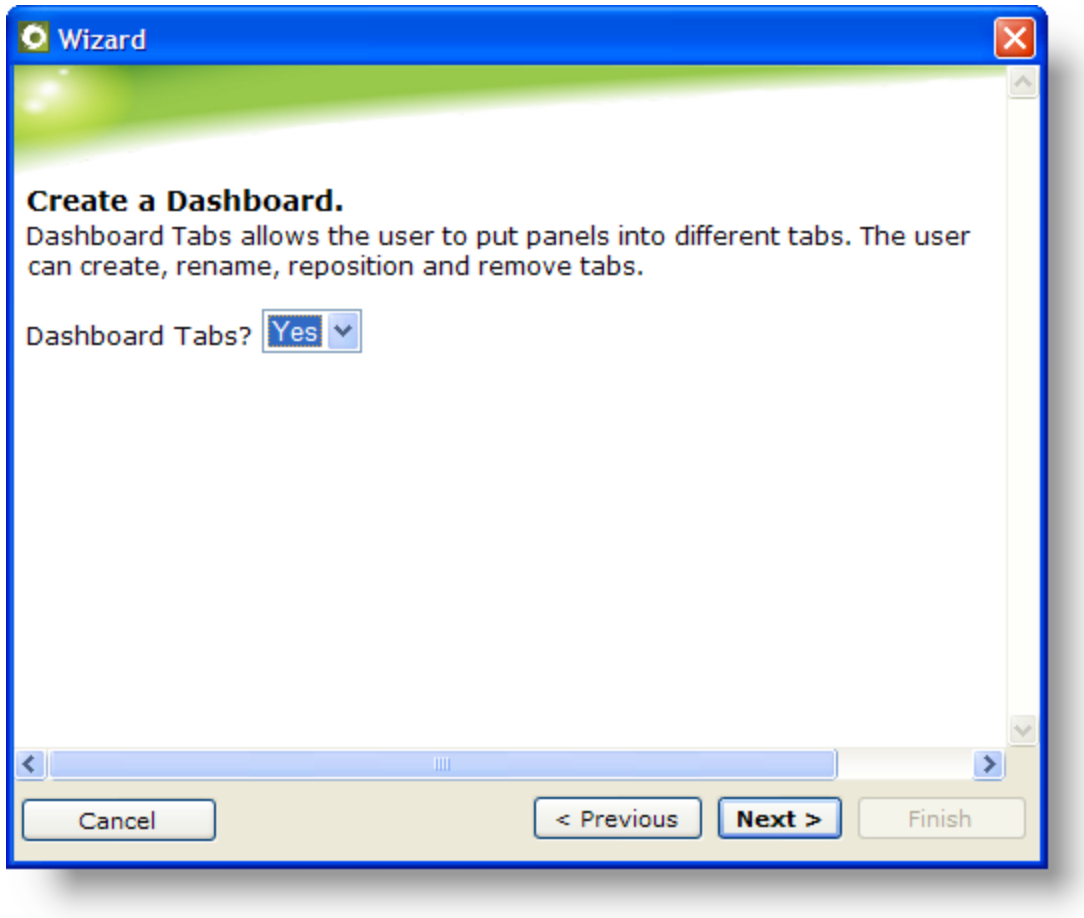
1. When the wizard is started, it will display an introductory dialog box, shown above. You can click **Previous** at any time to return to the previous dialog box. Click **Next** to continue.



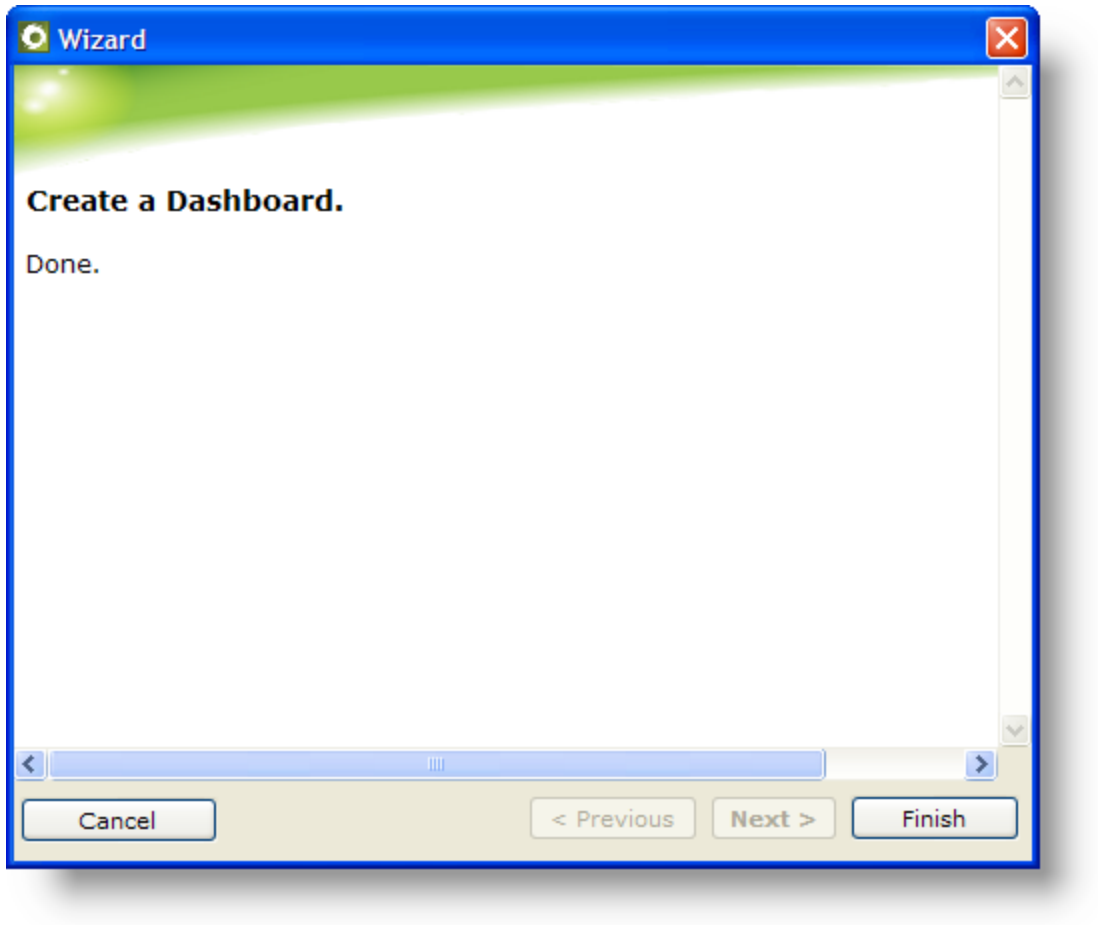
2. The wizard will prompt you to select the number of columns to include in the Dashboard. Move the slider to increase or decrease the number of columns. Click **Next** to continue.



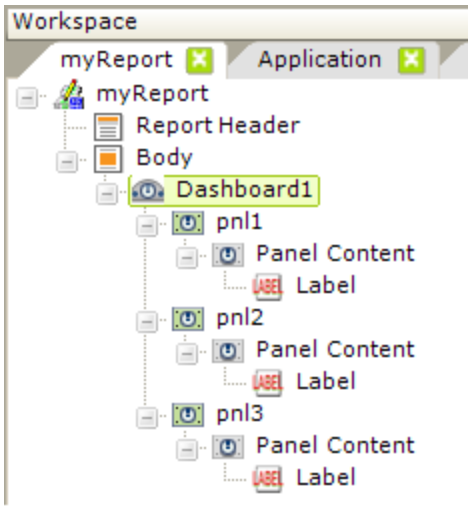
3. The wizard will prompt you to select the number of Dashboard panels to include in the Dashboard initially. You may add additional panels later, when the wizard completes, if necessary. Move the slider to increase or decrease the number of panels. Click **Next** to continue.



4. The wizard will prompt you to select the indicate whether or not the Dashboard will include tabs. If you allow it, users can add and manage the tabs at runtime. Select *Yes* or *No* and click **Next** to continue.



5. The wizard will insert all of the elements necessary to create the Dashboard and then finally display the dialog box shown above. Click **Finish** to complete the process.



Your definition will now include all the elements, similar to those shown above, inserted by the wizard. At this point, you may begin to add elements for the content of the individual Dashboard panels.

Event Logging

Event logging can be valuable for operational auditing and is also an important tool for troubleshooting Logi applications.

The following topics introduce developers to the use of Logi Info's Event Logging elements and custom logging techniques:

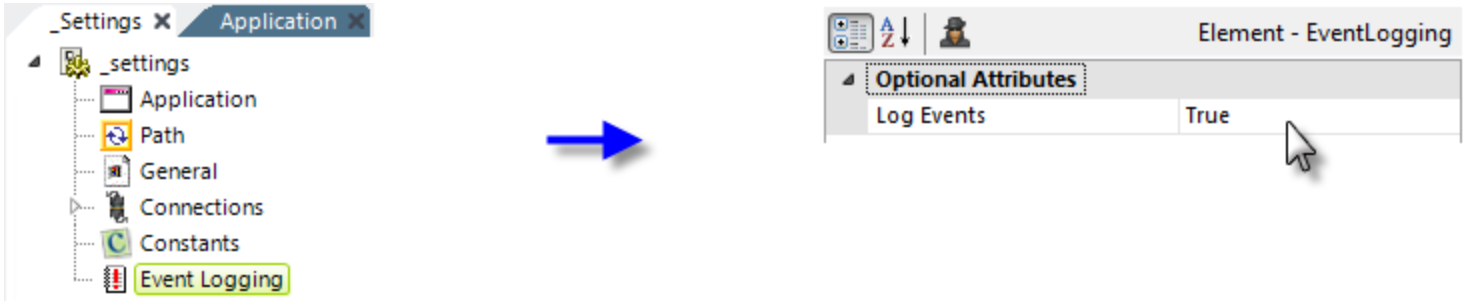
- Adding Standard Event Handling to Your Application
- Standard Events
- Custom Logging

About Event Logging

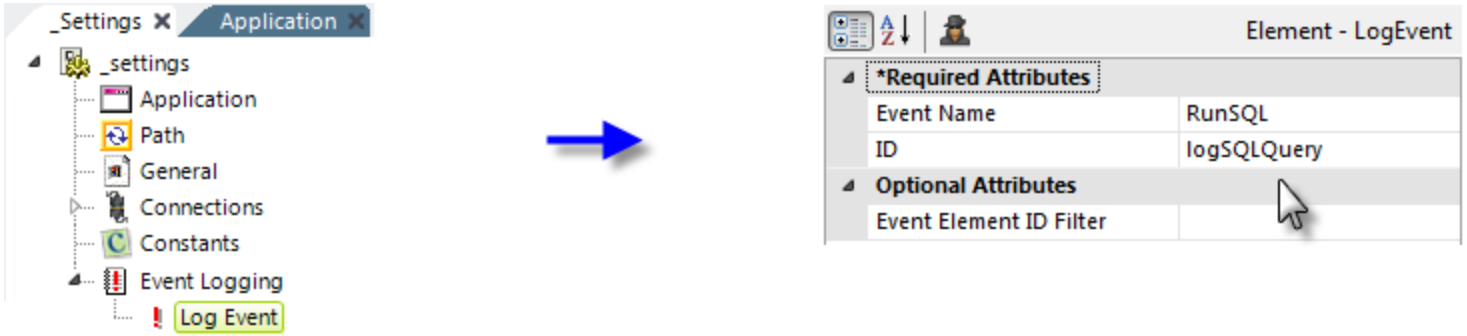
Logi Info includes built-in support for handling standard*events*, which are internal notifications. Standard eventsare triggered or "fired" when specific actions occur. In the context of a Logi application, you as a developer can define event handling, using Action.Process elements, so that processing occurs when standard events are fired. To do this, you associate an **Action.Process** element with an event and specify a Process definition task. When the event fires, program flow will be directed to that task and specific parameters will be automatically passed to it. These parameters communicate *event-specific information* to the task, which can be designed to log the event by writing it to a database table, displaying it in a report, or taking other task actions. You can also code your own *custom* logging, whichdoesn't use standard events but is instead based on operational flow, as discussed in "Custom Logging" on page 411.

Adding Standard Event Handling to Your Application

Logging standard events requires that you include appropriate elements in your application definitions:

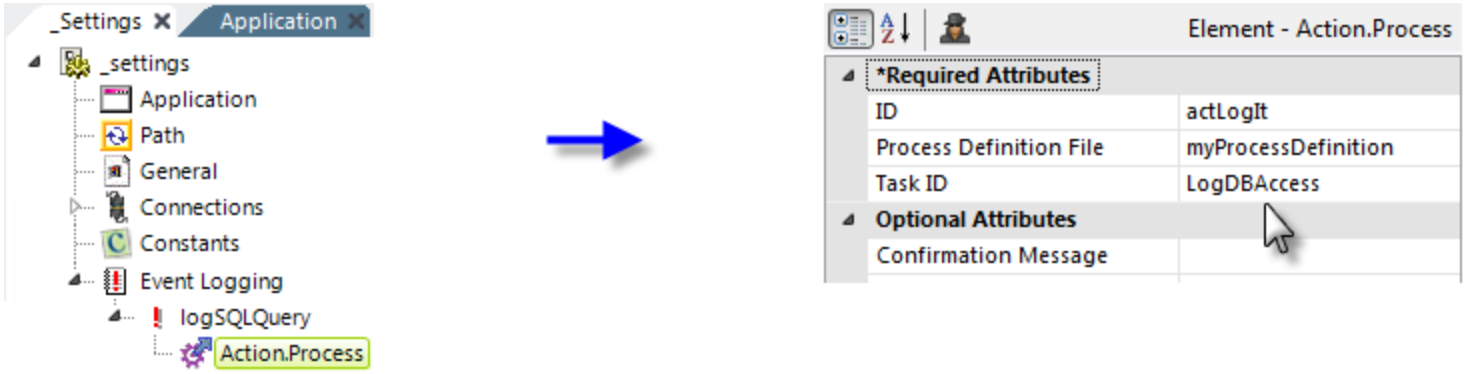


1. Open your application's **_Settings** definition in Studio and add an **Event Logging** element, as shown above. Set its **Log Events** attribute to *True*. This attribute can be used to toggle event logging on and off. Tokens *cannot* be used in this attribute value.

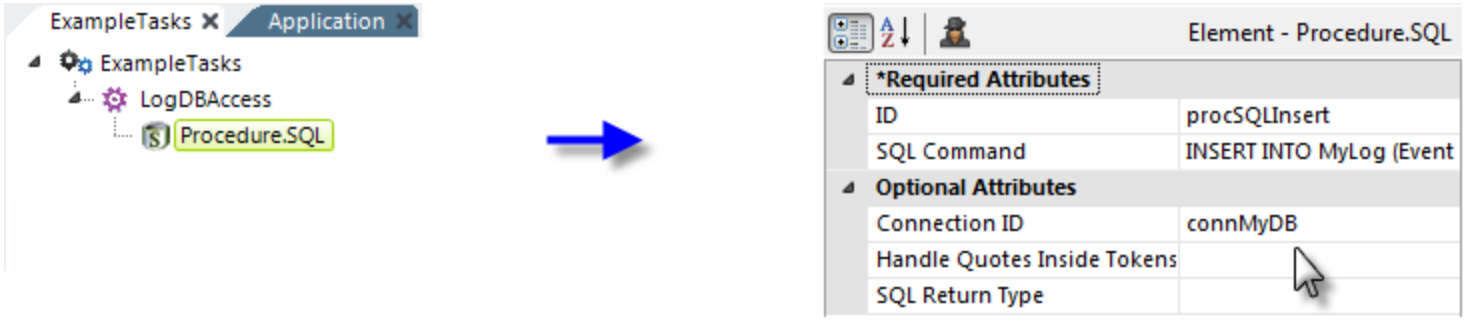


2. Below the Event Logging element, add a **Log Event** element and set its **Event Name** attribute to the type of event to be logged, as shown above. The Event Name attribute value has a drop-down list of the valid events, which are discussed in detail later in this topic. Multiple Log Event elements can be added in order to provide logging of different types of events.

   The optional **Event Element ID Filter** attribute may be used to "zero-in" on the behavior of a specific element. You identify the element, by its ID, as the only element for which events of this type should be handled. Events of this type generated by other elements will then be ignored.



3. Add a standard **Action.Process** element below each Log Event element, as shown above. The **Process Definition File** and **TaskID** attributes identify the definition file and task, respectively, that will be called when the event fires.



4. The final step is to create a process **task** to handle the information provided when the event fires. A sample task is shown above.

   This is an example of an SQL statement that adds the logging information to a database. Notice the **tokens** used in the VALUES clause:

   ```
   INSERT INTO myLog (EventName, ElementID, EventTime, SPName)
   VALUES ('@Request.EventName~', '@Request.ElementID~', @Request.EventTime~, '@Request.SP~')
   ```

   The records generated by such a statement might look like the following example:

EventName	ElementID	EventTime	SPName
RunSP	dlUserProfiles	2007-06-08 09:33:14	spInsertUser
RunSP	dlUserProfiles	2007-06-08 09:34:21	spInsertUser
RunSP	dlUserProfiles	2007-06-08 09:34:42	spUpdateUser
RunSP	dlUserProfiles	2007-06-08 09:34:56	spInsertUser

   The @Request parameters that are automatically passed when the event fires **vary** depending on the **event type**; they are discussed in the following sections.

Standard Events

The following tables identify the standard events and the Request parameters that are generated for them:

SessionStart Event

The SessionStart event fires when the web server *session starts* for the current user. The parameters that are automatically passed by an associated Action element include:

@Request Parameter	Description
EventName	SessionStart
EventTime	The event's timestamp, in the format: `yyyy-MM-dd HH:mm:ss` Example: `2007-04-25 17:49:21`
EventTimePrecise	The event's timestamp, including milliseconds, in the format: `yyyy-MM-ddTHH:mm:ss.msZ` Example:`2007-04-25T17:49:21.8106969Z`

BuildReport Event

The BuildReport event fires when the Report ID *is determined* and ends *after* the final output HTML is created. The parameters that are automatically passed by an associated Action element include:

@Request Parameter	Description
EventName	BuildReport

@Request Parameter	Description
ElementID	Report
EventTime	The event's timestamp, in the format: `yyyy-MM-dd HH:mm:ss` Example: `2007-04-25 17:49:21`
EventTimePrecise	The event's timestamp, including milliseconds, in the format: `yyyy-MM-ddTHH:mm:ss.msZ` Example:`2007-04-25T17:49:21.8106969Z`
EventDuration	The duration of the event, in milliseconds. It captures the time it takes to produce the report on the server but does not include the time for the network to transmit the report to the client, nor the time for the client workstation to load the report.
ReportID	The report definition name. Example: `Default` During processing the Logi Engine may append additional identifying characters to the Report ID value. If you intend to write this information to a database, we recommend that you define the column for this value in the schema as at least `varchar(100)`.

AuthenticateUser Event

Logi Security must be in use in order for the AuthenticateUser event to occur. In that case, the event fires when a user's Rights and Roles are determined. The firing of the event is therefore dependent on the setting of the **Security** element's **CacheRights** attribute. If the attribute is set to *False* or left blank, the event will fire with every request; if it's set to *Session*, the event will only fire at the *beginning* of the session. The parameters that are automatically passed by an associated Action element include:

@Request Parameter	Description
EventName	AuthenticateUser
ElementID	Security
EventTime	The event's timestamp, in the format: `yyyy-MM-dd HH:mm:ss` Example: `2007-04-25 17:49:21`
EventTimePrecise	The event's timestamp, including milliseconds, in the format: `yyyy-MM-ddTHH:mm:ss.msZ` Example:`2007-04-25T17:49:21.8106969Z`

RunSQL Event

The RunSQL event fires when a **DataLayer.SQL** or **Procedure.SQL** element runs. The parameters that are automatically passed by an associated Action element include:

@Request Parameter	Description
EventName	RunSQL
ElementID	The ID of the DataLayer.SQL or Procedure.SQL element that triggered the event.
EventTime	The event's timestamp, in the format: `yyyy-MM-dd HH:mm:ss` Example: `2007-04-25 17:49:21`
EventTimePrecise	The event's timestamp, including milliseconds, in the format: `yyyy-MM-ddTHH:mm:ss.msZ`

@Request Parameter	Description
	Example:`2007-04-25T17:49:21.8106969Z`
EventDuration	The duration of the event, in milliseconds.
RowCount	The number of rows returned by the SQL query *(only available when DataLayer.SQL runs).*
SQL	The SQL command that was executed.

RunSP Event

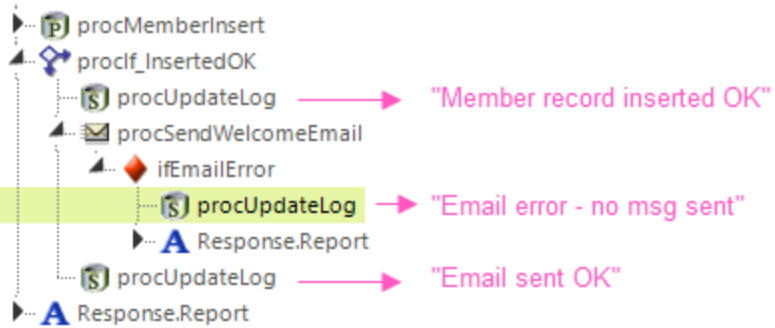
The RunSP event fires when a **DataLayer.SP** or **Procedure.SP** element runs. The parameters that are automatically passed by an associated Action element include:

@Request Parameter	Description
EventName	RunSP
ElementID	The ID of the DataLayer.SP or Procedure.SP element that triggered the event.
EventTime	The event's timestamp, in the format: `yyyy-MM-dd HH:mm:ss` Example: `2007-04-25 17:49:21`
EventTimePrecise	The event's timestamp, including milliseconds, in the format: `yyyy-MM-ddTHH:mm:ss.msZ` Example:`2007-04-25T17:49:21.8106969Z`
EventDuration	The duration of the event, in milliseconds.

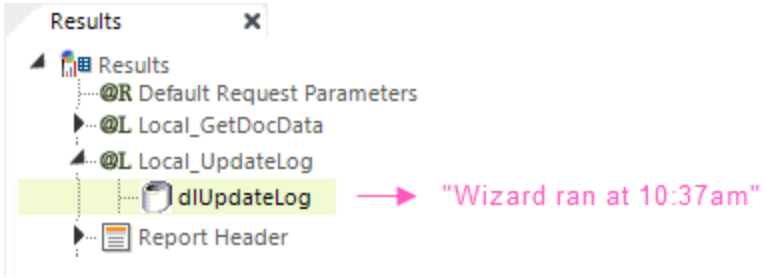
@Request Parameter	Description
RowCount	The number of rows returned by the stored procedure *(only available when DataLayer.SP runs).*
SP Parameter IDs	An @Request token for each input SP Parameter element used. Example: `@Request.myParam1~,@Request.myParam2~, etc.`
SP	The name of the stored procedure that was executed.

Custom Logging

You may wish to write your own custom logging for use in addition to, or instead of, the standard event handlers. The actual writing of the data to a log is done in the same way as discussed above for the standard events: with a SQL statement that writes out the particulars. However, rather than use the Event Logging element and standard events, you can have the SQL statement execute as part of the regular application flow. Keep in mind that because you're not using standard events, there will be no automatically-generated Request parameters. It's up to you to see that you have all the right data, Request and Session tokens, etc. that you need for your log at the time when the entry will be written. The @Function.DateTime~ token, for example, is very useful in this situation. Let's look at two example scenarios. First, imagine that your Logi application uses a Process task to add a new member record and send a confirming email, and you want a record of when that message was sent:



To do this, you can simply add a Procedure.SQL element after your **Procedure.Send Mail** element, as shown above. The SQL statement writes to the log table and can make use of some of the tokens used to personalize the email message. You can even go so far as to add *two* Procedure.SQL elements, one under an **If Error** element that writes to the log if there's a problem sending the message, and one that writes to it if the message was successfully sent. In our second scenario, imagine a report definition that acts as a "wizard", collecting input element selections, and then calls another report definition that displays some kind of results based on the parameters passed to it. You'd like to keep track of usage of the wizard in a log.



To make this log entry, simply add a **Local Data** element with a child **DataLayer.SQL** element, as shown above, in the "results" definition. When it loads, the datalayer will run and the log entry will be inserted. You can also use the Local Data element's **Condition** attribute to control this action dynamically.

Definition Modifier Files

Definition Modifier Files (DMFs) allow you to modify any of the elements and attributes in a report definition, based on conditional criteria, at runtime.

The following topics discuss the use of these files:

About Definition Modifier Files

A DMF is an XML file that contains instructions to modify a definition file's elements and their attributes at runtime. It's a separate file that's processed *before* the Logi Server Engine starts to render a report definition. At that time, elements may be conditionally inserted or removed, and attributes may be set or unset.

This makes it possible to set culture- or customer-specific values based, for example, on locale or security roles, at run-time. This means that each user can potentially receive a different report because the definition used to generate their report has been customized for them, on the fly.

DMFs are similar to "Template Modifier Files" on page 426(which only affect super-elements and themes) and they're also conceptually similar to a plug-in that executes on the "LoadDefinition" event. In this context, you might think of them as the "poor man's plug-in": they can provide similar functionality but don't require additional development tools (Visual Studio, etc.) to write and compile a plug-in .dll or .jar file.

Here are two examples of how DMFs might be used.

- Language Translation: you can use **XPath** notation in a DMF to find and replace specific strings of text anywhere in a report definition, in order to translate them into another language on the fly. You could even have a different DMF for each language supported and tokenize the DMF name in your report definition so that you can support multiple languages on the fly.
- Insert Additional Elements: you can use a DMF to add custom reporting content (headers, footers, and data) based, for example, on the user type, company, or other criteria.

DMFs can even insert nested DMFs.

When debugging links are turned on and DMFs are used, special entries will appear in the Debugger Trace report:

Apply Theme	Theme Name	Blues
	Applied Modifier	View Modifier Log
	Done	View Modified Definition
	Load Modifier File for element: DefinitionModifierFile ID:	DMF_HideHeaderText.xml
	Applied Modifier	View Modifier Log
	Done	View Modified Definition

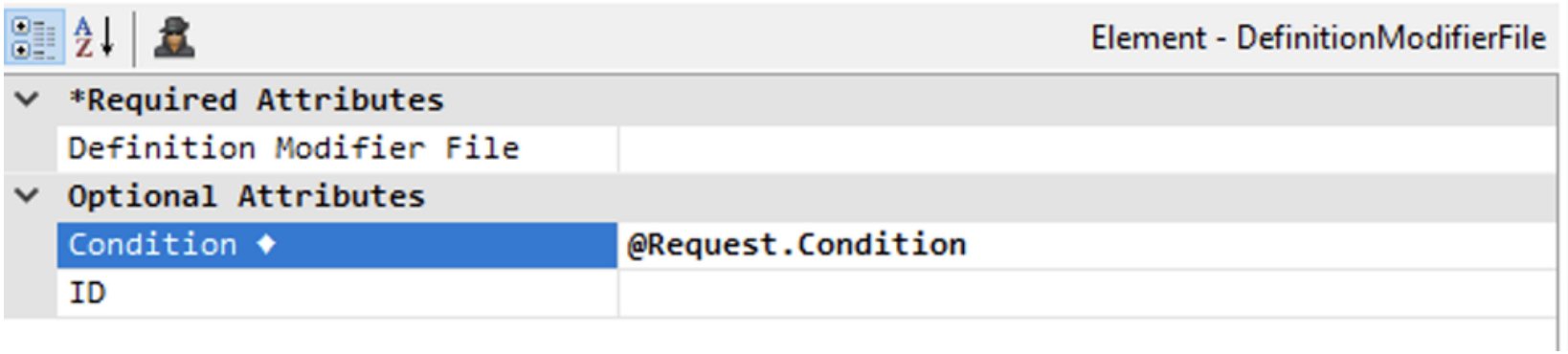
```
Modifier File: DMF_HideHeaderText.xml

        XPath = //DataTableColumn[@Header='Order ID']
                SetAttribute
                        Header =

        XPath = //DataTableColumn[@Header='Cust ID']
                SetAttribute
                        Header =
```

As shown above, these entries will follow those for the DMF applied by a Logi Theme. Click them to view the DMF log and the source code of the modified definition. For more information, see "Debug Reports" on page 56.

The Definition Modifier File Element

The Definition Modifier File element is a child of the report definition's root element. The presence of a DMF element tells the Logi Server Engine to process the file it identifies. Its **Definition Modifier File** attribute value is the name of the actual XML file, with .xml extension. The file can be in any folder accessible to the Logi application but when a fully-qualified file path is not specified, the assumed location is the _SupportFiles folder.

The Definition Modifier File (DMF) element now includes a **Condition** attribute that enables loading of the DMF. By default, this attribute it set to "True".



- If this attribute is not set, set to an empty string, or set to "1", it is considered "True".
- You can use tokens to provide the DMF filename (as shown above), enabling you to specify different DMFs (or no DMF) for different situations, users, etc.

The Definition Modifier File

Although the proper terminology for an XML object is "element", the term "tag" will be used in the following discussion in order to avoid confusion between Logi elements and XML elements.

A DMF is an XML file that contains special tags that temporarily customize a report definition, in memory, just as the Logi Server Engine begins to render it. This "virtual customization" does not alter the report definition saved in the file system. These XML files can be edited right in Studio's Workspace Panel. Here's an example of a DMF:

```
<DefinitionModifier>

          <!-- 1. setting attributes based on existing
 content -->
          <SetAttribute
 XPath="//DataTableColumn[@Header='Order ID']"
 Header="Ordre" />
          <SetAttribute
 XPath="//DataTableColumn[@Header='Order Date']"
 Header="Date d'Ordre" />
          <SetAttribute
 XPath="//DataTableColumn[@Header='Customer ID']"
 Header="ID de Client" />
          <SetAttribute
 XPath="//DataTableColumn[@Header='Freight']"
 Header="Fret" />

          <!-- 2. adding additional elements into the report
```

```
-->
         <AppendChildElement
XPath="/Report/ReportFooter">
         <NewElement>
         <Division
ID="divExports">
         <LineBreak
LineCount="1" />
         <Label
ID="lblExportExcel" Caption="Export Excel">
         <Action
Type="NativeExcel" ID="exportExcel">
         <Target
Type="NativeExcel" />
         </Action>
         </Label>
         <Spaces
Size="3" />
         <Label
ID="lblExportPDF" Caption="Export PDF">
         <Action
Type="PDF" ID="exportPDF">
         <Target
Type="PDF" />
         </Action>
```

```
          </Label>
          <Spaces
Size="3" />
          <Label
ID="lblExportWord" Caption="Export Word">
          <Action
Type="NativeWord" ID="exportWord">
          <Target
Type="NativeWord" />
          </Action>
          </Label>
          </Division>
          </NewElement>
          </AppendChildElement>
          </DefinitionModifier>
```

The example starts and ends with <DefinitionModifier> tags, but they're not strictly required; any tags that properly open and close the XML are acceptable.

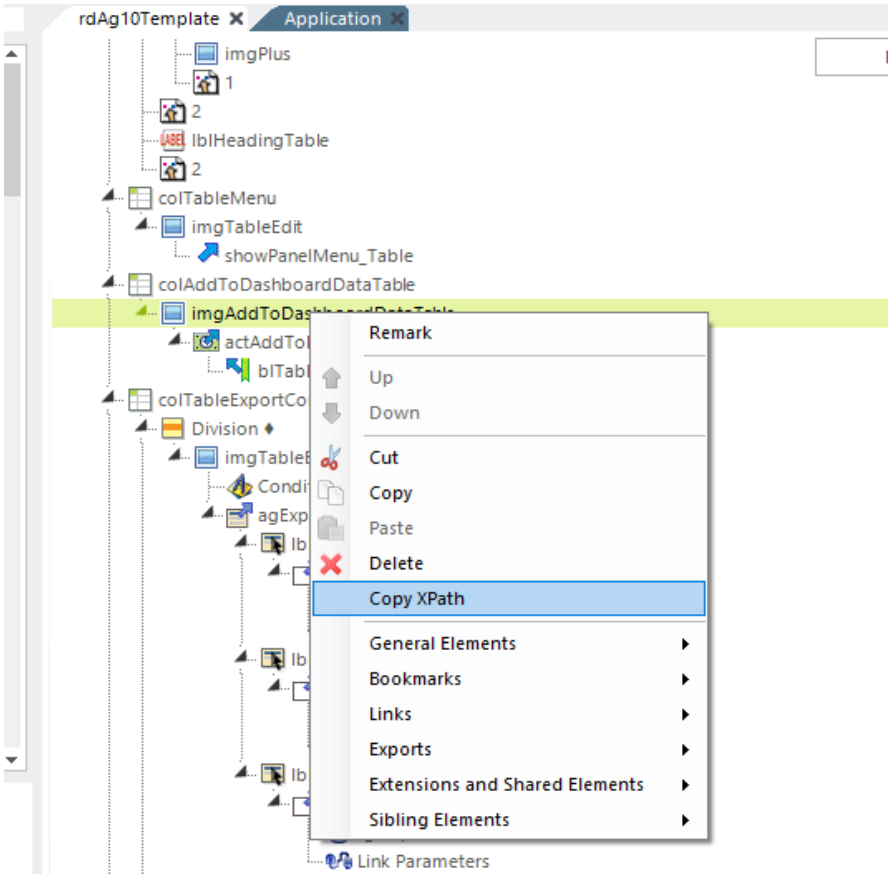
Comments can be embedded into the XML using the HTML <!-- ... --> comment syntax.

If a Security Right ID attribute is being set, the element is re-evaluated after the operation for security purposes.

XPath or Element ID Notation

In the first section of the example, a special **<SetAttribute>** tag is used to find and replace the desired text. **XPath notation** is used here to identify the element and attribute values to be replaced. The result is that all of the English text in the table column headers will be replaced with French text.

In Studio, you can find the XPath of an element in a report or template by right-clicking on the element and selecting **Copy XPath**:



Selecting this option will copy the direct XPath of that element, the tag name, and ID attribute (if available) to the Clipboard, which can then be copied into a TMF.

See this site for more information about XPath syntax.

In the second section, the **<AppendChildElement>** tag is used to insert elements and set their attributes. One of the beauties of this tag is that you can list multiple elements beneath it as a group, rather than having to use a tag per element, as you might expect.

Also in this section, to illustrate an alternative method, the actual Logi **element ID** is used to identify the new elements and their attributes, rather than XPath. Either method, XPath or Element ID notation, can be used. The result of this second section is that a set of Export links are added to report footer.

List of Special XML Tags

This table provides a complete list of the special XML tags available for use in DMFs:

XML Tag	Description
AppendChildElement	Appends new child element as the *last* element beneath the element identified in its XML "ID" attribute. See section below about use of NewElement tag.
InsertAfterElement	Inserts new element *after* the element identified in its XML "ID" attribute. See section below about use of NewElement tag.
InsertBeforeElement	Inserts new element *before* the element identified in its XML "ID" attribute. See section below about use of NewElement tag.

XML Tag	Description
PrependChildElement	Adds new child element as the *first* element beneath the element identified in its XML "ID" attribute. See section below about use of NewElement tag.
Remark	"Comments" the XML tag it encloses, so it has no effect.
RemoveElement	Removes the element identified in its XML "ID" attribute.
SetAttribute	Sets the attribute value identified for the element identified in its XML "ID" attribute, over-writing any value already there.
SetAttributeWhenEmpty	Sets the attribute value identified for the element identified in its XML "ID" attribute, but only if that attribute does not have a value already.
SetAttributeWithInsert	Inserts an additional attribute value *before* any existing value identified for the element identified in its XML "ID" attribute. This order is important, as it allows existing values related to CSS to "win" any conflicts by being last in the order.
UpdateOrAppendChildElement	If the child element does not exist, appends it as the *last* element beneath the element identified in its XML "ID" attribute. If the child element exists, updates its attributes. *See section below about use of NewElement tag.*

If elements affected by these operations have their Security Right ID attributes set, those attribute values will be evaluated *before* the modification is executed. This allows security rights to control which modifications are applied. Additional examples of these tags in use can be found in "Template Modifier Files" on page 426.

Using the <NewElement> Tag

Tags that insert or add a new element use the <NewElement> child tag set to enclose the element to be inserted:

```
      <InsertAfterElement>
      <NewElement>
      <Division></Division>
      </NewElement>
      </InsertAfterElement>
```

You may only insert one element per <NewElement> tag set, as shown above.

```
      <InsertAfterElement>
      <NewElement>
      <Division></Division>
<Division></Division> <--- will be
 ignored
      </NewElement>
      </InsertAfterElement>
```

You may *not* insert multiple elements per <NewElement> tag, as shown above.

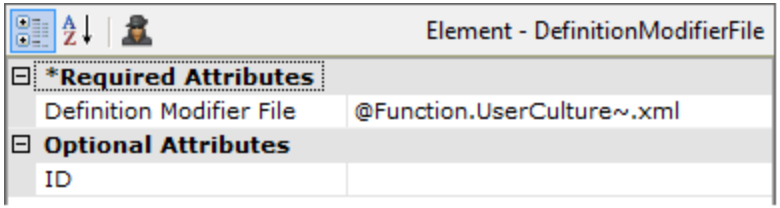
Instead use multiple <InsertAfterElement> (or similar) tags,

```
<InsertAfterElement>
<NewElement>
<Division>
<Division></Division>
<Division></Division>
</Division>
</NewElement>
</InsertAfterElement>
```

or wrap the multiple elements in a single top-level container element, as shown above.

Using Tokens in the DMF

DMFs can provide dynamic customizations through the use of tokens.



First, the actual file name used in the Definition Modifier File attribute can be a **token**. You could, for example, have a different DMF for each language or culture and use the token that signifies the user's browser Culture setting, as shown above, to select the correct file.

```
<SetAttribute XPath="//DataTableColumn[@Header='Freight']" Header="@Constant.myItalianWord~" />
```

Second, you can use tokens within the DMF file itself. For example, the line above replaces the word "Freight" with the value of a constant.

Template Modifier Files

Template Modifier Files (TMFs) allow Logi Info developers to modify the "super-elements", such as the Analysis Grid, which behave like mini-applications, have a UI, and let users control visualizations at runtime. In addition, Logi Info developers can use TMFs to create themes.

The following topics discuss the use of these files:

- Setting Hidden Underlying Element Attributes
- Using XPath Notation
- Modifying Selected Chart Captions
- Manipulating Existing Underlying Elements
- Inserting and Removing Underlying Elements
- List of Special XML Tags

For more information about Themes, see *Working with Themes*.

Looking for information about Excel, Word, or PDF templates? Template Modifier Files are *not* related to them; see *Form-based Reporting* for information about those date-related templates.

About Template Modifier Files

Super-elements are a complex combination of standard elements, JavaScript, and style classes. The following super-elements can be customized using a Template Modifier File (TMF):

- Analysis Chart
- Analysis Filter

- Analysis Grid
- Chart Grid
- Dashboard
- Dimension Grid
- OLAP Grid
- Schedule
- Report Author

Super-elements are rendered at runtime using a special definition file and standard Logi Info elements. A TMF is a separate definition file that's processed *after* a super-element's definition file, providing an opportunity to alter the super-element at runtime, without editing the fundamental super-element blueprint. UI customizations can be placed in a TMF to alter the appearance and behavior of the super-element.

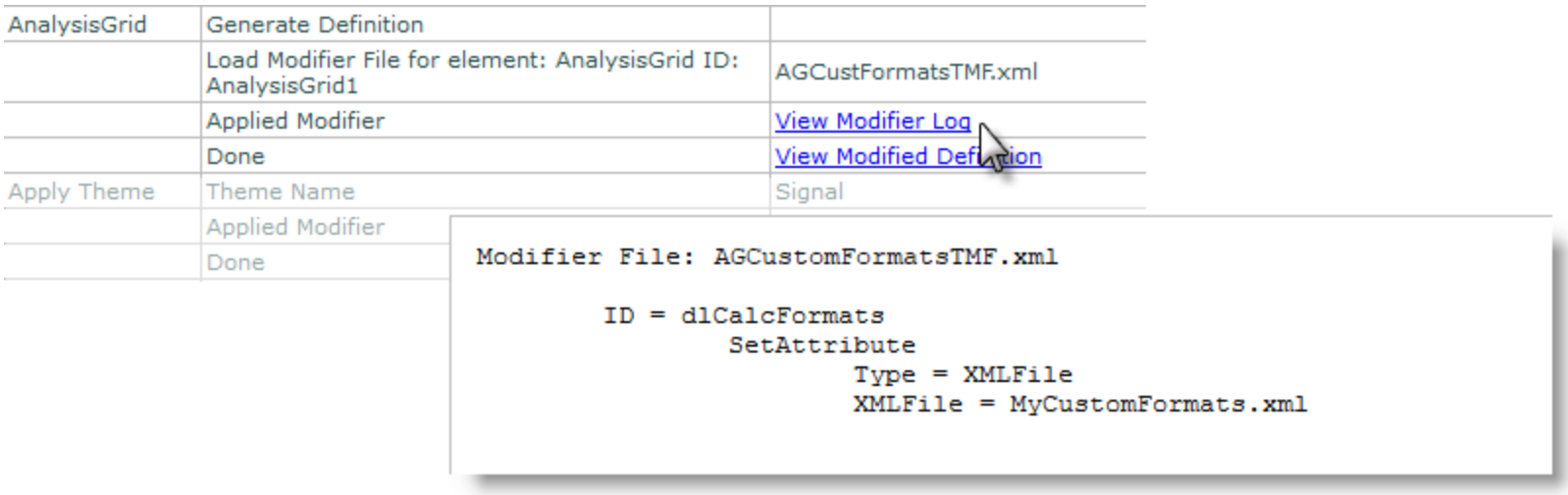
Examples of common customizations using TMFs include language- and culture-specific text changes for internationalization, and the hiding of some of controls, such as **Export** buttons.

A TMF is an XML file that contains special tags for customizing the super-element. It can be stored in any folder accessible to the web application and it's specified in a super-element's **Template Modifier File** attribute. If a fully-qualified file path is not provided in the attribute value, then the application expects it to be in your application's _SupportFiles folder.

Tokens can be used in the Template Modifier File attribute to provide the TMF filename. This allows you to specify different TMFs (or no TMF) for different situations and/or different users.

Template modifier files are also a primary component of Logi **Themes** technology. The power of the TMF is quite obvious in this scenario, as it effectively "scripts" the output of the designated report to adopt the desired appearance.

When debugging links are turned on and TMFs are used, special entries will appear in the Debugger Trace report (see):



As shown above, these entries will be in the AnalysisGrid section. Click them to view the TMF log and the source code of the modified definition.

<Column> Element vs. <Column> Tag

Although the proper terminology for an XML object is "element", the term "tag" will be used in the following discussion in order to avoid confusion between super-elements, underlying elements, and XML elements.

Setting Hidden Underlying Element Attributes

Some aspects of the appearance and behavior of a super-element can be configured using its *exposed* attributes (those that appear in Studio in the Attributes Panel). Further customization can occur by using a TMF to change its *hidden* attributes: the attributes of the underlying elements that make up the super-element.

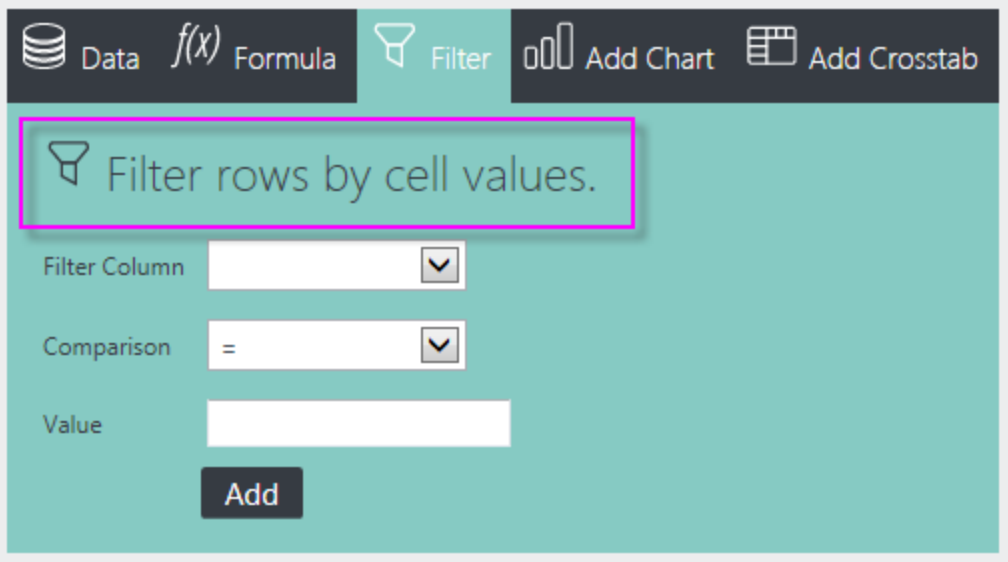
The key to configuring hidden attributes is knowing the **ID** of the underlying elements.

The IDs of these underlying elements can be found by examining the special definition file that defines each super-element. These files are located in your application's `rdTemplate` folder. Here are a few of them:

Super-Element	Template File
Analysis Chart	rdTemplate\rdAnalysisChart\rdAcTemplate.lgx
Analysis Filter	rdTemplate\rdAnalysisFilter\rdAfTemplate.lgx
Analysis Grid	rdTemplate\rdAnalysisGrid\rdAg10Template.lgx
Dashboard	rdTemplate\rdDashboard\rdDashboard2Template.lgx
Dimension Grid	rdTemplate\rdOlapGrid\rdDgTemplate.lgx
OLAP Grid	rdTemplate\rdOlapGrid\rdOgTemplate.lgx
Schedule	rdTemplate\rdSchedule\rdScheduleTemplate.lgx
Report Author	rdTemplate\ReportAuthorTemplate.lgx

Never edit these files directly; only examine them to determine element IDs!

Here's a simple example of how TMFs are used. Suppose we want to change the Layout "descriptive text" that appears in an Analysis Grid.



When the **Filter** tab is clicked and the Filter configuration panel appears in an Analysis Grid, as shown above, the highlighted descriptive text is displayed.

To change this text to **French**, we first examine the Analysis Grid's definition file, `rdAg10Template.lgx`, with Notepad or a suitable text editor:

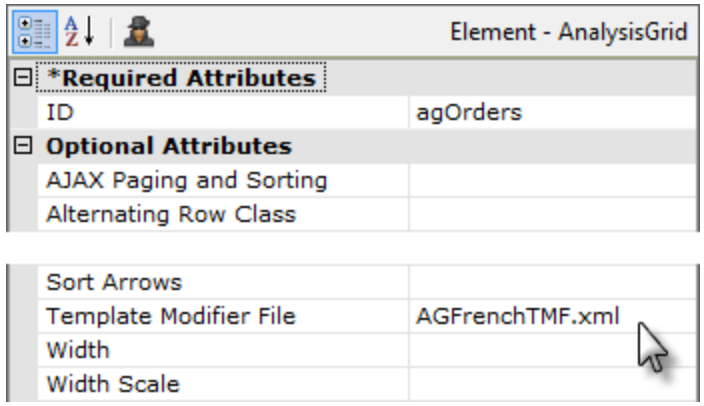
```
<Label ID="lblFilterDescription" Class="rdAgContentHeading" Caption="Filter rows by cell values." />
```

If we search the file for "filter rows", we'll find the code shown above. It contains a **Label** element, with an **ID** attribute of "lblFilterDescription", that sets the text we want to change. Note the ID of this element for use in our TMF.

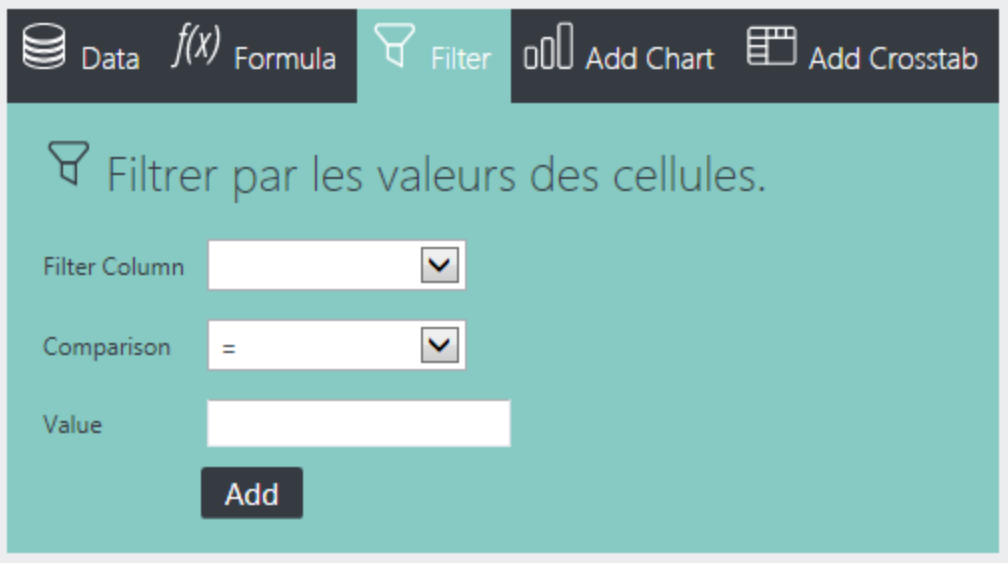
```
<TemplateModifier>
 <SetAttribute ID="lblFilterDescription" Caption="Filtrer par les valeurs des cellules." />
 </TemplateModifier>
```

To cause the change, we create a TMF, as shown above, which we'll give the arbitrary filename `AGFrenchTMF.xml`.

This is a standard XML file, which starts and ends with the `<TemplateModifier>` tag. It uses a special TMF tag `<SetAttribute>` to make the change and the element to be changed is identified using its **ID** from the Analysis Grid's template definition file ("lblLayoutDescription"). The attribute to be changed is identified as the **Caption** and is given its new French text value.



Next we save the TMF file to the application's _SupportFiles folder and, finally, enter its filename as the Analysis Grid's **Template Modifier File** attribute value, as shown above.

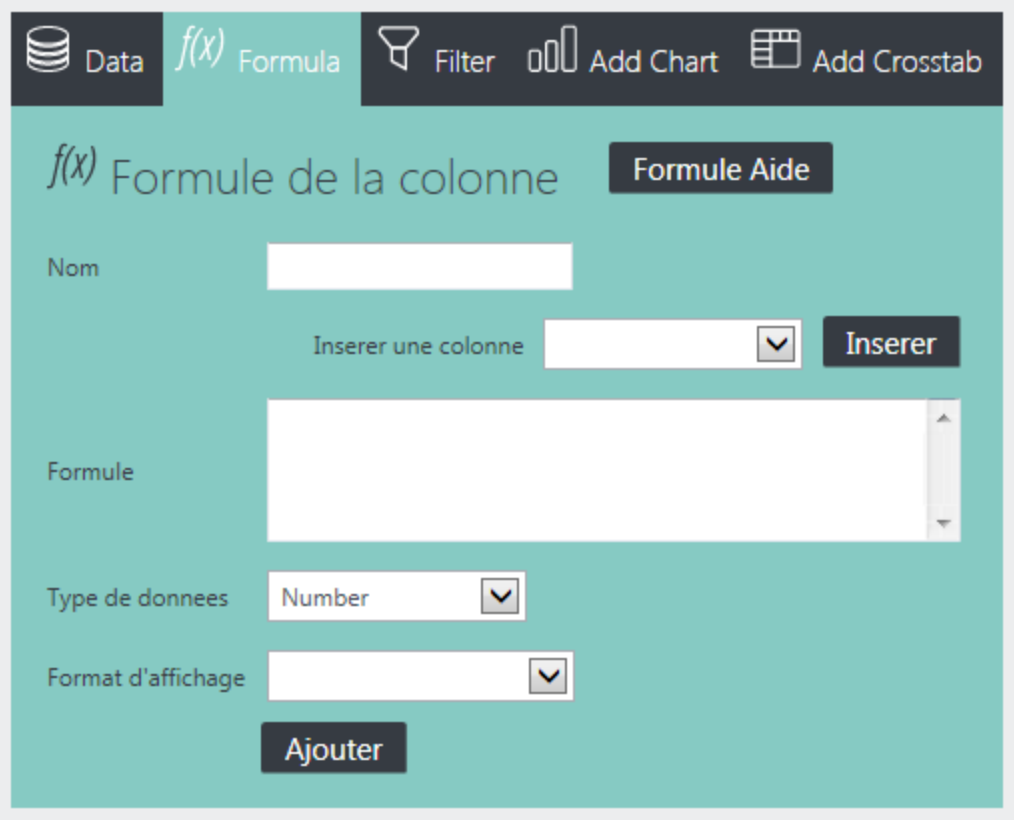


And the resulting change, shown above, can be seen when the report is run again.
Now that you've seen the basic concept in action, here's a more complicated TMF example:

```
<ExampleTMF>
<!-- These SetAttributes will change the Formula Column Panel Text -->
<SetAttribute ID="lblFormulaColumnDescription" Caption="Formule de la colonne" />
<SetAttribute ID="lblCalcHelp" Caption="Formule Aide" />
 <SetAttribute ID="lblCalcName" Caption="Nom" />
<SetAttribute ID="lblCalcInsertDataColumn" Caption="Inserer une colonne" />
<SetAttribute ID="lblCalcInsertDataColumnNow" Caption="Inserer" />
<SetAttribute ID="lblCalcFormula" Caption="Formule" />
```

```
<SetAttribute ID="lblCalcDataType" Caption="Type de donnees" />
<SetAttribute ID="lblCalcDisplayFormat" Caption="Format d'affichage" />
<SetAttribute ID="lblCalcOk" Caption="Ajouter" />
</ExampleTMF>
```

Actual `<TemplateModifier>` tags are not required; any tags that properly open and close the XML are acceptable. Also note that **comments** can be embedded into the XML using the HTML `<!-- ... -->` comment syntax.



The resulting changes are shown above. This example changed the section description, field captions, and button captions to French.

If a Security Right ID attribute is being set, the element is re-evaluated after the operation for security purposes.

For information about the Input Hidden element, see Input Hidden.

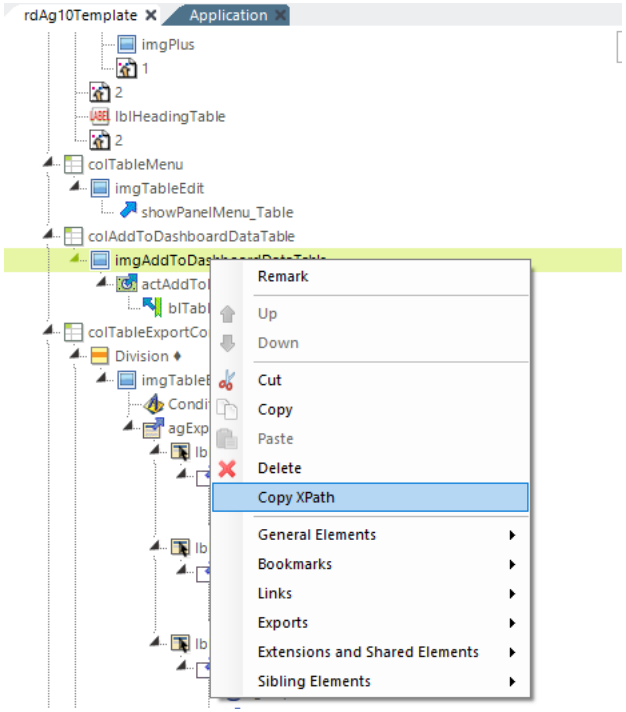
Using XPath Notation

The TMF examples shown so far have used element **IDs** to identify elements whose attributes are to be changed. However, XPath notation can be used instead, if preferred.

```
<TemplateModifier>
 <SetAttribute XPath="//Label[@ID='lblFilterDescription']" Caption="Filtrer par les valeurs des cellules." />
 </TemplateModifier>
```

The example shown above uses an earlier TMF example with XPath notation, instead of an element ID. XPath can be very useful when you want to identify multiple elements with similar names or types.

In Studio, you can find the XPath of an element in a report or template by right-clicking on the element and selecting **Copy XPath**:



Selecting this option will copy the direct XPath of that element, the tag name, and ID attribute (if available) to the Clipboard, which can then be copied into a TMF.

See this external site for more information about XPath syntax.

Modifying Selected Chart Captions

When you create charts using the **Analysis Chart** or **Analysis Grid** super-elements, the chart caption is created automatically and appears in a format similar to:

Sum of OrderID by OrderDate

You may wish to translate this caption and/or reorder its parts, and this can be done using a TMF. Here's an example of the relevant code in the Analysis Chart template file (`rdTemplate/rdAnalysisChart/rdAcTemplate.lgx`):

```
<ChartCanvas ChartCaption="{AggrName} of {DataColumnName} by {LabelColumnName}" CaptionDated="
{AggrName} of {DataColumnName} by {LabelColumnName}" ChartHeight="300" ChartWidth="500" Border-
Color="#7A7A7A" BorderRadius="4" ID="ChartPie" rdUnderSuperElement="True">
```

Note the construction of the default **ChartCaption** and **CaptionDated** attribute values, highlighted above. The items within the curly braces are internal chart variables and, using a TMF, you can substitute a different value that includes them, as follows:

```
<SetAttribute XPath="//ChartCanvas" ChartCaption="{AggrName} de {DataColumnName} par {LabelColum-
nName}" />
```

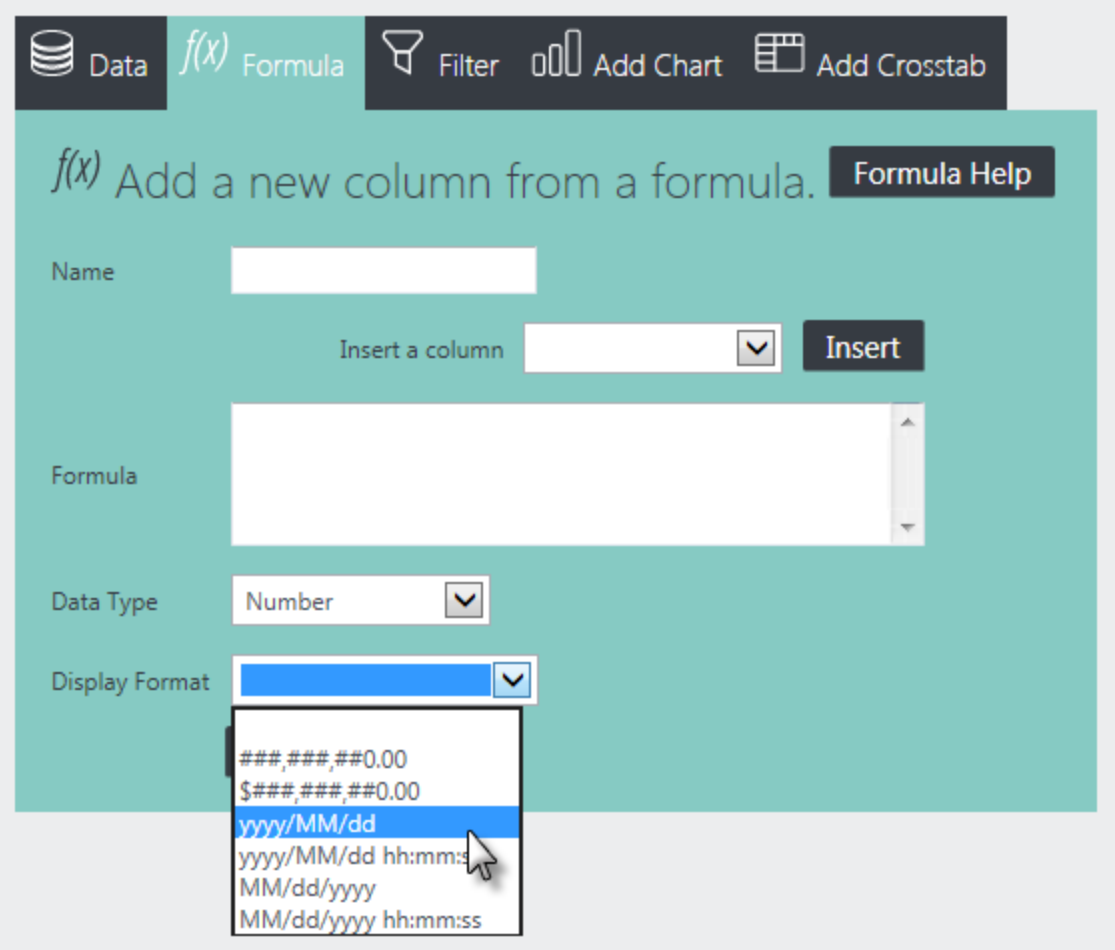
The example TMF code shown above translates the English connecting words "of" and "by" to their French equivalents. You can change the value in any way you desire, using the { } chart variables or removing them altogether.

```
<SetAttribute XPath="//ChartCanvas" ChartCaption="{DataColumnName} by {LabelColumnName}: {AggrName}"
/>
```

You can also rearrange the value parts, as shown above. The resulting caption would appear as "OrderID by OrderDate: Sum".

Manipulating Existing Underlying Elements

Some of the super-elements include static data sets, such as **Format** options, that are presented in the user interface as drop-down selection lists. TMFs can also be used to alter this data.



For example, in the Analysis Grid's **Formula** tab, the user can select from a list of six pre-defined display format options, shown above. Let's see how we can use a TMF to change the format options.

```
<Column ID="col2Row5Calc">
 <InputSelectList OptionValueColumn="Format" ID="rdAgCalcFormats" IncludeBlank="True" OptionCap-
tionColumn="FormatName" DefaultValue="@Request.rdAgCalcFormats~">
 <DataLayer Type="Static" ID="dlCalcFormats">
 <StaticDataRow Format="###,###,##0.00" FormatName="###,###,##0.00" />
 <StaticDataRow Format="$###,###,##0.00" FormatName="$###,###,##0.00" />
 <StaticDataRow Format="yyyy/MM/dd" FormatName="yyyy/MM/dd" />
 <StaticDataRow Format="yyyy/MM/dd hh:mm:ss" FormatName="yyyy/MM/dd hh:mm:ss" />
 <StaticDataRow Format="MM/dd/yyyy" FormatName="MM/dd/yyyy" />
 <StaticDataRow Format="MM/dd/yyyy hh:mm:ss" FormatName="MM/dd/yyyy hh:mm:ss" />
 </DataLayer>
 </InputSelectList>
</Column>
```

If we look at Analysis Grid's template definition file again, we find the above code that shows a **DataLayer.Static** element and child **Static Data Row** elements are used by default to create the six standard format options. In order to change that list, we need to do two things:

1. Provide an XML data file that contains the data for our custom option list, and
2. Override the existing datalayer, replacing it with a **DataLayer.XML** element that will read our custom option list file.

Here's the XML data file with our nine custom format options:

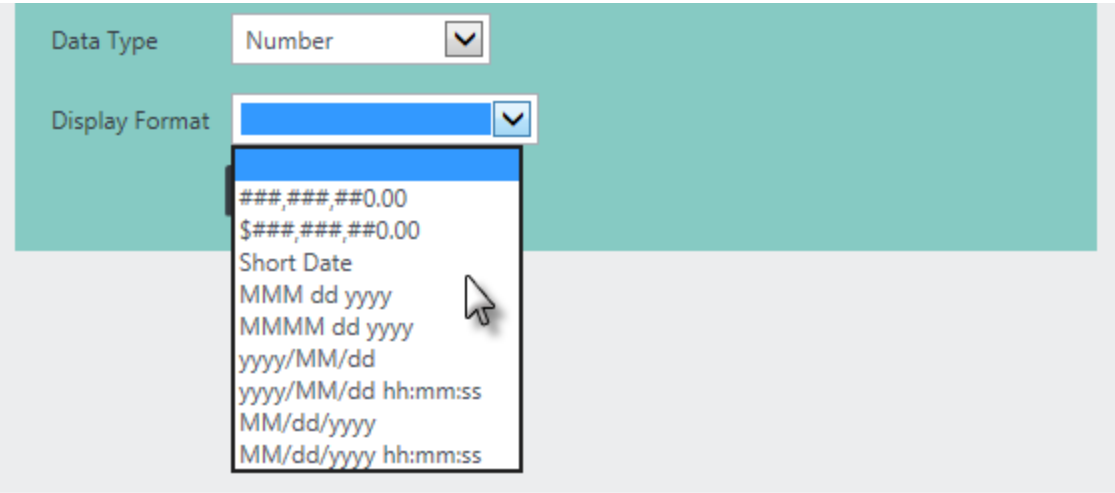
```
<MyCustomFormats>
<Option Format="###,###,##0.00" FormatName="###,###,##0.00"/>
<Option Format="$###,###,##0.00" FormatName="$###,###,##0.00"/>
<Option Format="Short Date" FormatName="Short Date"/>
<Option Format="MMM dd yyyy" FormatName="MMM dd yyyy"/>
<Option Format="MMMM dd yyyy" FormatName="MMMM dd yyyy"/>
<Option Format="yyyy/MM/dd" FormatName="yyyy/MM/dd"/>
<Option Format="yyyy/MM/dd hh:mm:ss" FormatName="yyyy/MM/dd hh:mm:ss"/>
<Option Format="MM/dd/yyyy" FormatName="MM/dd/yyyy"/>
<Option Format="MM/dd/yyyy hh:mm:ss" FormatName="MM/dd/yyyy hh:mm:ss"/>
</MyCustomFormats>
```

The column names ("Format" and "FormatName") in the XML data file *mustmatch* those used in the original Static Data Row elements in the Analysis Grid template definition file. Our custom format options file is stored in the application's _SupportFiles folder as `MyCustomFormats.xml`

```
<TemplateModifier>
<SetAttribute XPath="//Column[@ID='col2Row5Calc']/InputSelectList/DataLayer" Type="XMLFile"
 XMLFile="MyCustomFormats.xml" />
 </TemplateModifier>
```

Finally, we need to create the TMF shown above. It uses XPath notation to change the type of the existing template DataLayer.Static element to a DataLayer.XML element and specifies MyCustomFormats.xml, our custom options file, as the datasource.

"Type" and "XML File/URL" are standard attributes of the DataLayer.XML element. When this file has been created and saved to _SupportFiles, all we need to do is set the Analysis Grid's Template Modifier File attribute to our TMF filename and preview it.



The resulting Display Format option list is shown above and it now contains our nine custom format options. This example demonstrates that TMFs are not limited to mere text changes, but can also be used to manipulate existing elements in more interesting ways.

Working with Underlying Datalayers

In many cases, the text that appears in super-element interfaces is drawn from internal static datalayers. Suppose you needed to change specific text that comes from one of these datalayers, for example, into a different language. To do this you would:

1. Provide an XML data file, in _SupportFiles, that includes the translated text.
2. Use a TMF to change the relevant internal datalayer element type from "Static" to "XML File" and to point it at your XML data

file.

3. In the TMF, change the internal Label element that displays the text to use @Data tokens.

There is one complication here: tokens inside the TMF itself are evaluated prior to being used in the modification. This is because you might have @Local, @Request, or other tokens that need to be evaluated in this file prior to being used. However, that won't achieve the "later" evaluation that we need to put the token values in the user interface. To get around this, you need to "escape" the @Data token used in your TMF for Step #3 above like this:

```
@@Data.dummy~Data.columnName~
```

This isa "token-within-a-token" construction, where `@Data.dummy~` is an arbitrary, dummy token and `Data.columnName` is the @Data token and column name from the XML data file. When the report runs and the TMF is evaluated, the dummy token will be evaluated first, to an empty string, leaving the real @Data token to be recognized and evaluated later. The odd construction above has the result of masking the fact that `@Data.columnName~` *is* a token during the first evaluation.

Inserting and Removing Underlying Elements

You can also use TMFs to insert, remove, and remark the internal elements that make up a super-element. This is accomplished using these XML tags:

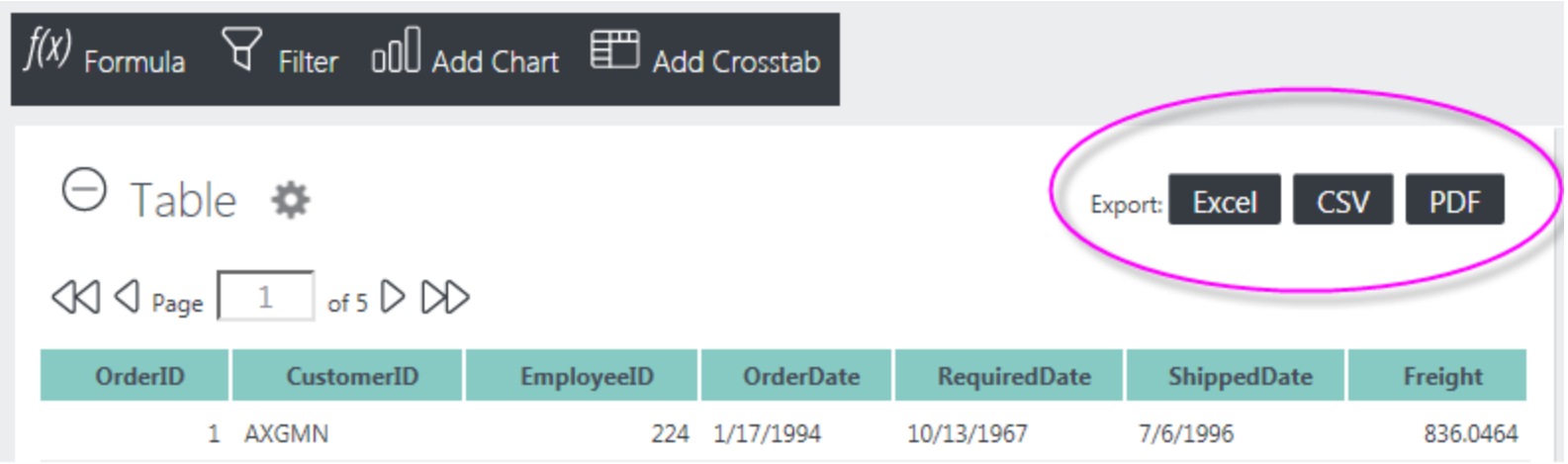
- <PrependChildElement>
- <AppendChildElement >
- <InsertBeforeElement >
- <InsertAfterElement >
- <Remark >
- <RemoveElement >

The first four tags use a child tag, `<NewElement>`, to delimit the new element to be affected. The `<Prepend-` and `<Append-` tags make the new element the first or last child element in a group. See the section below about using the `<NewElement>` tag.

TMFs can insert "Definition Modifier Files" on page 413.

If elements affected by these operations have their Security Right ID attributes set, those attribute values will be evaluated *before* the modification is executed. This allows security rights to control which modifications are applied.

Here's a practical example of some of these elements in action:



An Analysis Grid can display three **Export** buttons, as shown above, and its **Hide Exports** attribute can be used to hide all three of them from users. However, if you use the attribute to hide the buttons, it's an "All-or-None" situation: developers can't choose to show a subset of the export buttons. A solution for those who wish to show a subset is to use a TMF. In the following example, we'll remove the **CSV** export button and add a **Print** button that mimics the PDF export functionality. One important thing to know when looking at the Analysis Grid's definition file to get the appropriate element IDs is that the export "buttons" are really **Label** elements made to look like buttons using CSS. Assuming the elements have been found and identified, the TMF can be written as follows:

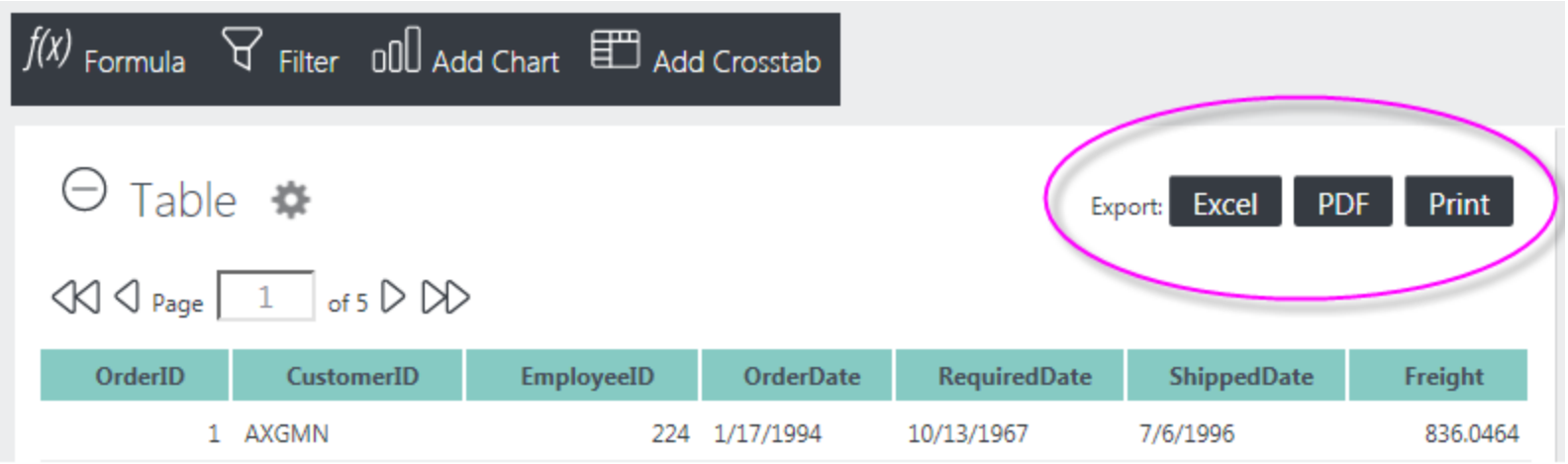
```
<TemplateModifier>
<RemoveElement ID="lblExportCsv" />
<InsertAfterElement ID="lblExportPdf">
<NewElement>
<Label Caption="Print" ID="lblPrintBtn" Class="rdAgCommand" />
</NewElement>
</InsertAfterElement>
```

```
<AppendChildElement ID="lblPrintBtn">
<NewElement>
<Action Type="PDF" ID="actExportPrint" />
</NewElement>
</AppendChildElement>
<AppendChildElement ID="actExportPrint">
<NewElement>
<Target Type="PDF" Report="CurrentReport" ID="tgtExportPrint"/>
</NewElement>
</AppendChildElement>
<InsertAfterElement ID="tgtExportPrint">
<NewElement>
<LinkParams rdAgNoDefCache="True" rdExportTableID="dtAnalysisGrid"/>
</NewElement>
</InsertAfterElement>
</TemplateModifier>
```

The first thing this TMF does is remove the CSV button ("lblExportCsv") using the `<RemoveElement>` tag.

Then three elements are appended, using the `<InsertAfterElement>` and `<AppendChildElement>` tags, creating the new Print button, its child **Action** element, and that element's child **Target** element. Where they are appended is specified in the tag's ID attribute. Note the use of the `<NewElement>` tag to delimit each element being added.

Finally, `<InsertAfterElement>` is used to add a LinkParams element at the same element hierarchy level as the Target element.



The resulting interface change is shown above. It's possible to write the first part of the TMF in a more compact way, taking advantage of tag nesting:

```
<TemplateModifier>
<RemoveElement ID="lblExportCsv" />

<InsertAfterElement ID="lblExportPdf">
<NewElement>
<Label Caption="Print" ID="lblPrintBtn" Class="rdAgCommand">
 <Action Type="PDF" ID="actExportPrint" >
 <Target Type="PDF" Report="CurrentReport" ID="tgtExportPrint"/>
 </Action>
</Label>
</NewElement>
```

```
</InsertAfterElement>
</TemplateModifier>
```

As shown above, only one `<InsertAfterElement>` tag is necessary, as long as the opening and closing tags for the nested elements are arranged correctly.

```
<TemplateModifier>
<Remark>
<RemoveElement ID="lblExportCsv" />
</Remark>

<InsertAfterElement ID="lblExportPdf">
<NewElement>
<Label Caption="Print" ID="lblPrintBtn" Class="rdAgCommand">
 <Action Type="PDF" ID="actExportPrint" >
 <Target Type="PDF" Report="CurrentReport" ID="tgtExportPrint"/>
 </Action>
</Label>
</NewElement>
</InsertAfterElement>

<InsertAfterElement ID="tgtExportPrint">
<NewElement>
<LinkParams rdAgNoDefCache="True" rdExportTableID="dtAnalysisGrid"/>
</NewElement>
```

```
</InsertAfterElement>
</TemplateModifier>
```

And, in the last variation, shown above, rather than removing tags, the `<Remark>` tag allows you to temporarily "comment out" portions of the TMFs, as shown above.

Using the <NewElement> Tag

Tags that insert or add a new element use the `<NewElement>` child tag set to enclose the element to be inserted:

```
<InsertAfterElement>
 <NewElement>
<Division></Division>
</NewElement>
</InsertAfterElement>
```

You may only insert one element per `<NewElement>` tag set, as shown above.

```
<InsertAfterElement>
 <NewElement>
<Division></Division>
<Division></Division> <--- will be ignored
</NewElement>
</InsertAfterElement>
```

You may *not* insert multiple elements per `<NewElement>` tag, as shown above.
Instead use multiple `<InsertAfterElement>` (or similar) tags,

```
<InsertAfterElement>
 <NewElement>
<Division>
<Division></Division>
<Division></Division>
</Division>
</NewElement>
</InsertAfterElement>
```

or wrap the multiple elements in a single top-level container element, as shown above.

List of Special XML Tags

Here's a complete list of the special XML tags available for use in TMFs:

XML Tag	Description
AppendChildElement	Appends new child element as the *last* element beneath the element identified in its XML "ID" or "XPath" attribute. See the Using the <NewElement> Tag section in "Inserting and Removing Underlying Elements" on page 444.
CutElement	Used as a child of the Insert tags, this tag deletes the element identified by its XML "ID" or "XPath" attribute and makes it available for insertion. Used primarily to "move" elements within the definition.
CycleAttributeValues	Used primarily with Themes, allows assignment of values to elements such as charts that have multiple layers. Elements to be modified are identified in its XML "CycleOnChildElement" attribute; uses child tag CycleValue.
CycleValue	Used to provide values for CycleAttributeValues tag.
InsertAfterElement	Inserts new element *after* the element identified in its XML "ID" or "XPath" attribute. See the Using the <NewElement> Tag section in "Inserting and Removing Underlying Elements" on page 444.
InsertBeforeElement	Inserts new element *before* the element identified in its XML "ID" or "XPath" attribute. See the Using the <NewElement> Tag section in "Inserting and Removing Underlying Elements" on page 444.

XML Tag	Description
PrependChildElement	Adds new child element as the *first* element beneath the element identified in its XML "ID" or "XPath" attribute. See the Using the <NewElement> Tag section in "Inserting and Removing Underlying Elements" on page 444.
Remark	"Comments" the XML tag it encloses, so the tag has no effect.
RemoveAttribute	Removes the attribute identified using its element XML "ID" or "XPath" and "AttributeName" attributes.
RemoveElement	Removes the element identified in its XML "ID" or "XPath" attribute.
SetAttribute	Sets the attribute value identified for the element identified in its XML "ID" or "XPath" attribute, overwriting any value already there.
SetAttributeWhenEmpty	Sets the attribute value identified for the element identified in its XML "ID" or "XPath" attribute, but only if that attribute does not have a value already.
SetAttributeWithInsert	Inserts an additional attribute value *before* any existing value identified for the element identified in its XML "ID" or "XPath" attribute. This order is important, as it allows existing values related to CSS to "win" any conflicts by being last in the order.
UpdateOrAppendChildElement	If the child element does not exist, appends it as the *last* element beneath the element identified in its XML "ID" or "XPath" attribute. If the child element exists, updates its attributes. See the Using the <NewElement> Tag section in "Inserting and Removing Underlying Elements" on page 444.

Upload Files to the Web Server

Logi applications include the ability to allow users to upload files for storage on the web server. Logi Info provides elements specifically for this purpose and this topic provides guidance in using them.

The following topics discuss how to upload files to the web server:

- Identifying Files to Upload
- Controlling Maximum File Size
- Validating File Size Before Upload
- Saving Uploaded Files
- Qualifying Uploaded Files

About Uploading Files

Logi Info provides a mechanism for uploading files from a user's computer to the web server, using the RFC 1867 Form-based File Upload in HTML protocol. This protocol was published in 1995 yet continues to function cleanly. With it, files are transmitted *individually*, without encryption, and *immediately* (there is no deferred transmission).

The upload process first completes the file transmission then makes information about the file (type, size, etc.) available. At that point, developers can use code to test the uploaded file and delete it, if desired, as a kind of "retroactive filtering".

Upload Considerations

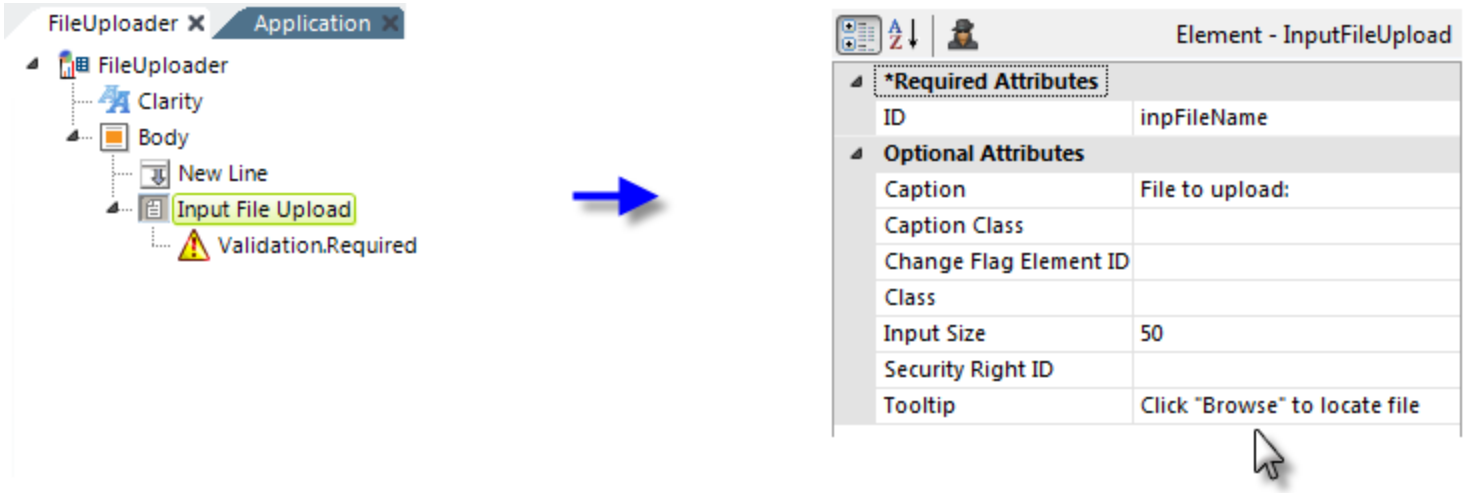
The process of uploading a file to your web server consists of two steps: first, at the browser, the user has to *identify* a file on his file system to be uploaded, and then, second, at the web server, the uploaded file has to be *named* and *saved*. Logi Info provides elements that make this all fairly easy to implement. However, developers first need to consider these issues:

- **What file types will you allow to be uploaded?** For security or virus-protection reasons, you may want to limit uploads to non-executable file types, prohibiting those such as .exe, .com, .bat, and perhaps even macro-capable Microsoft Office types such as .docx and .xlsx. It's good security practice to identify a small set of acceptable file types and refuse all others.
- **Should you restrict the size of uploaded files?** For most developers, this is a question of available storage space. The default maximum file size that can be uploaded under .NET is 4MB (although this can be increased if necessary - see below). You may also want to consider a limit on the *number* of files each user can upload.

  Another critical issue is that the actual upload transmission is not very efficient and large uploads can take a considerable amount of time. The process doesn't provide any feedback that would allow you to display a progress bar, either, so your users will have no way of knowing what's going on. Finally, if the upload is too large or the session times out, your application will throw an exception that's hard to handle gracefully.|
- **How will the uploaded files be stored and identified?** Uploaded files are first stored as temporary files in your application's *rdDownload* folder and are subsequently renamed and relocated. Developers need to plan for storage locations on the web server and for a method of providing unique names to the files (the Logi @Function.GUID~ token can be very helpful for this purpose). You could, for example, use a database table to hold information about the uploaded files such as their original names, sizes, and GUID-based identifiers.
- **How will the uploaded file be managed?** Assuming you're not going to just store uploaded files forever, you'll probably need to develop a way of managing them. Here again, a database table can be useful for identifying uploaded files for deletion.
- **What About Virus Protection?** If your web server's anti-virus software conducts "on access" scans of all files, any uploaded files stored on the web server should be examined automatically. If not, you'll want to consider how you can screen uploaded files for viruses.

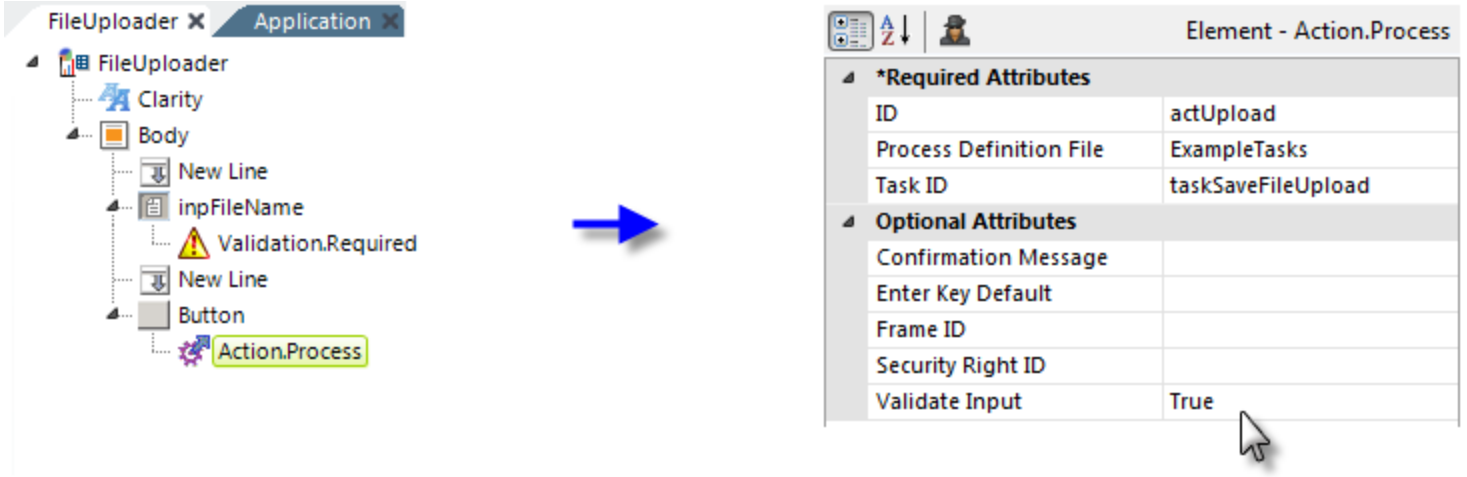
Identifying Files to Upload

The upload process begins with the user *identifying* the file to upload. This can be done very easily using the **Input File Upload** element in a report definition. This element displays both an input text box and a "Browse" button that allows users to browse their local file system.



1. As shown above, add an **Input File Upload** element to a report and set its attributes as desired.
2. Add a **Validation.Required** element beneath it to ensure that the filename value is not empty.

The example above shows what the **Input File Upload** element looks when displayed. Clicking the Browse... button opens a file explorer window, allowing the user to find and select a local file. The fully-qualified file name (drive letter, complete path, and full file name) is entered into the text box. It's not possible to enter this information by typing. Only *one* file at a time can be uploaded per Input File Upload element.



3. Add **New Line** and **Button** elements, as shown above, and beneath the Button, add an **Action.Process** child element which calls the appropriate Process task. Note that its **Validate Input** attribute is set to *True* to cause the Validation.Required element to check to see if a file name was entered.

When the user clicks the button and submits the page, the upload will take place immediately and the uploaded file will be stored as a temporary file in your application's *rdDownload* folder on the web server.

At the same time, the Process task specified will be called; its role is discussed in "Saving Uploaded Files" on page 461.

Controlling Maximum File Size

The default maximum upload size, as determined by the .NET Framework, is 4MB. To increase this, to a maximum of 2GB, you need to edit the `web.config` file in your application folder. In the `<system.web>` section, add this:

```
<httpRuntimemaxRequestLength="nnnn" executionTimeout = "ssss"/>
```

where: *maxRequestLength* specifies the limit for the input stream buffering threshold, in KB. This limit can be used to prevent denial of service attacks that are caused, for example, by users posting large files to the server. The default value is 4096 (4MB). To enable large file uploads you need to change the value of this attribute to the largest allowed combined file size for your application. If someone selects and uploads files with total size larger than maxRequestLength, this will result in a "Page not found" error (which is the default error of the Framework), and *executionTimeout* specifies the maximum number of seconds that a request is allowed to execute before being automatically shut down by .NET. The value of this setting is ignored in Debug mode. The default in the .NET 4.x Framework is 110 seconds. To enable large file uploads, which can take large periods of time, increase the value of this property.

Validate File Size Before Upload

If you have concerns about the potential size of uploaded files, you may find it prudent to check a file's size *before* it's uploaded. This can be easily done with some JavaScript and a small set of Logi elements.

Setting Size Limit Constants

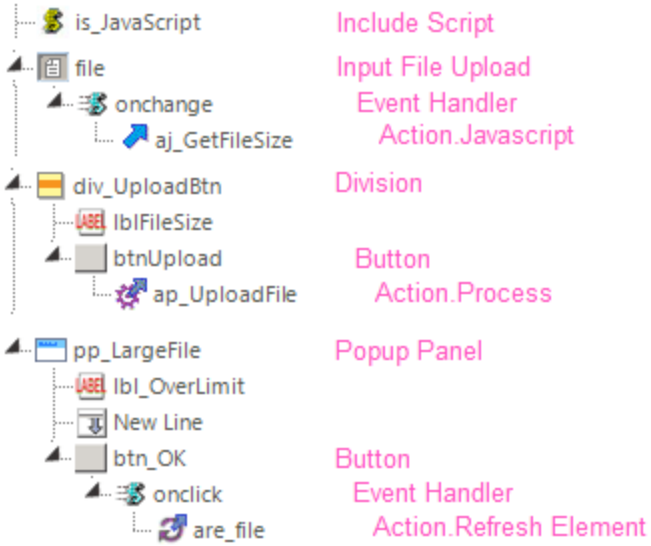
To set a file size limit, you first need to know what the system or environment limit is. For a Logi .NET application, the default maximum upload size is governed by the .NET Framework limit, which is 4MB. This may have been changed by the system administrator, to allow larger files or smaller files. You can find this value in the web.config file in your application folder or in the application's parent web.config file under the <system.web> section, as explained in "Controlling Maximum File Size" on the previous page.

Once the existing limits are known, you'll need to add two Constants in the _Settings definition: `setFileSizeLimitKB=25600` `setFileSizeLimitMB=25`

The examples shown above set a limit of 25MB. Use the ratio 1MB = 1,024KB to determine the values.

Setting Up the Elements

An example arrangement of elements is shown below:



The **Include Script** element "is_JavaScript" contains the JavaScript function that checks the file size and updates the other elements based on the file size; the code is available for download here.

The script uses the element IDs shown in the example above; if you use different IDs, edit the script to match.

The standard **Input File Upload** element "file" is used to select the desired file for upload.

The **Action.Javascript** element "aj_GetFileSize" calls the function using this JavaScript attribute value: *GetFileSize()*.

The **Division** element "div_UploadBtn" uses a Condition expression to show or hide the Upload button so the user has a visual clue that the selected file is ready for upload. Its Class attribute is set to *ThemeHidden* (or a similar `display:none` class) so that it, and the Upload button, are initially hidden from view.

The optional **Popup Panel** element "pp_LargeFile" is being used here to give the user feedback if the file exceeds the size limit.

What the JavaScript Does

The zipped files you can download using the link above includes both a file with the JavaScript code and a Word document that provides a detailed explanation of how the code works. Briefly, when a file is selected for upload by browsing, the code examines it and compares its size against the tokens for the two size limit constants set earlier. If within the limit, the code displays the file size and makes the Upload button visible. If beyond the limit, the Upload button remains hidden and a modal pop-up panel warns the user that the file is too large.

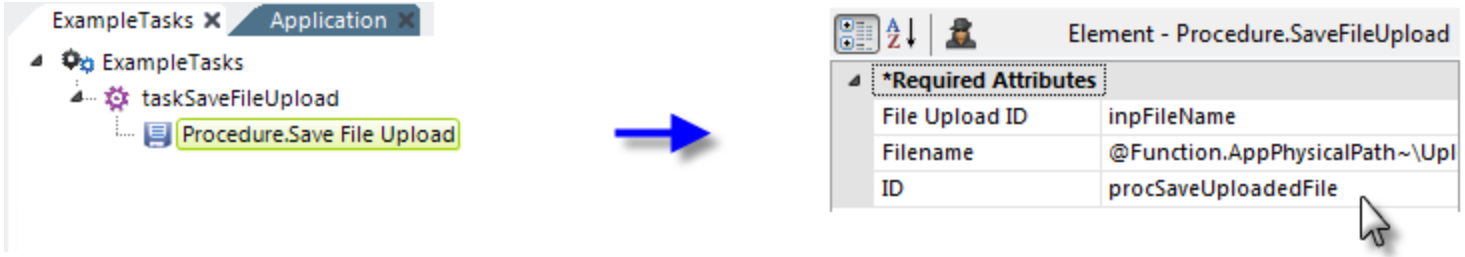
Saving Uploaded Files

After the physical upload completes, a Process task is used to save the temporary file to the final file and determine what has been uploaded. Several useful special tokens become available in the task to facilitate the process:

Token	Description
@FileUpload.UploadFileName~	Contains the file name, with extension but stripped of any path info, of the original file. Only available for use in attributes of the Procedure.Save Uploaded File element.  The RFC 1867 protocol used to upload files *does not* include access to the full file *path* of the original file, only its file *name*.
@FileUpload.UploadFileExtension~	Contains the file extension of the original file, including the period. This is case-sensitive and only available for use in attributes of the Procedure.Save Uploaded File element.
@Procedure.<*MyProc*>.UploadFileName~	Contains the file name, with extension but stripped of any path info, of the original file. Available throughout the task.
@Procedure.<*MyProc*>.UploadFileExtension~	Contains the file extension of the original file, including the period. This is case-sensitive and available throughout the task.
@Procedure.<*MyProc*>.UploadFileContentType~	Contains the file's MIME or content-type string. Available throughout the task.
@Procedure.<*MyProc*>.UploadFileLength~	Contains the number of bytes uploaded (the file size). Available throughout the task.

where *<MyProc>* is the ID of the task's **Procedure.Save Uploaded File** element.

Now you're ready to create a Process task to save the uploaded file:



1. In a process definition, add a **Task** element. Its **ID** should be the same as the value entered in the Task ID attribute of the Action.Process element in the report definition in "Identifying Files to Upload" on page 455.
2. Add a **Procedure.Save File Upload** element to the task.
3. Set its **File Upload ID** attribute to the ID of the Input File Upload element in the report definition.
4. Set its **Filename** attribute to the fully-qualified file name you want to save the temporary upload file to, beginning with a physical drive letter on the web server. Several tokens can be used together to provide a value for the this attribute, for example:

   ```
   @Function.AppPhysicalPath~\UploadedFiles\@FileUpload.UploadFileName~
   ```

   The value shown above will move the temporary upload file to a folder called "UploadedFiles" in your application folder and rename it to the file's original name. The target folder ("UploadedFiles") must exist *in advance*; the element will not create it. If this folder is outside of your application folder, the account the web server uses to run your application (ASPNET,

NETWORK SERVICE, or Application Pool) has to be granted *Write* permission to that folder.

4. To handle errors, add an **If Error** element beneath the Save Uploaded File element.
5. Add **Response.Report**, **Target.Report**, and **Link Params** elements, as shown, to redirect to the calling report in case of an error. Any error message will be available using this token:

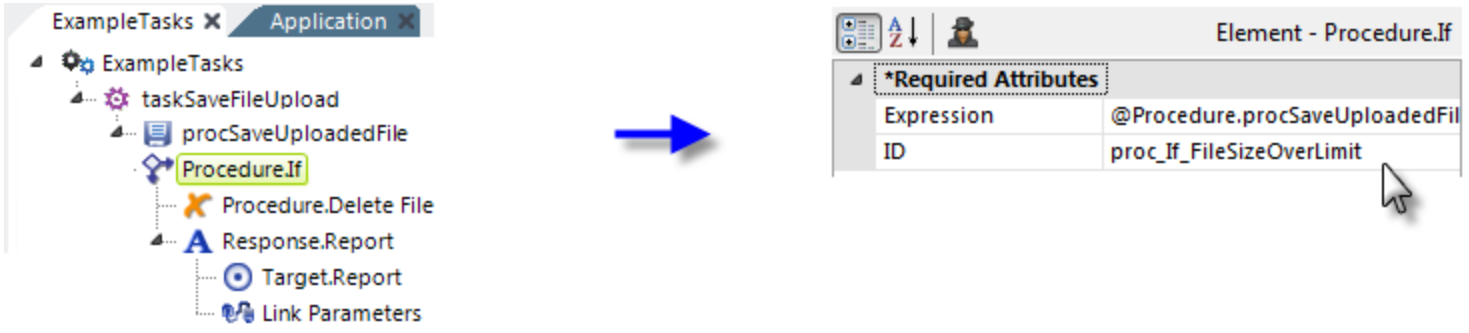
```
@Procedure.procSaveUploadedFile.ErrorMessage~
```

6. Back in the report definition, you can use an @Request token to test for and display any error message returned.

If the report is run now, the file will be uploaded and saved to the UploadedFiles folder. There is a default 4MB file size limit, discussed in more detail below.

Qualifying Uploaded Files

Once an upload has occurred and the temporary file has been saved, we have an opportunity to *qualify* the uploaded file and delete it if necessary, using criteria such as file size or file type. This "post-upload" filtering isn't the most desirable arrangement but it's all that's available under the RFC 1867 protocol.

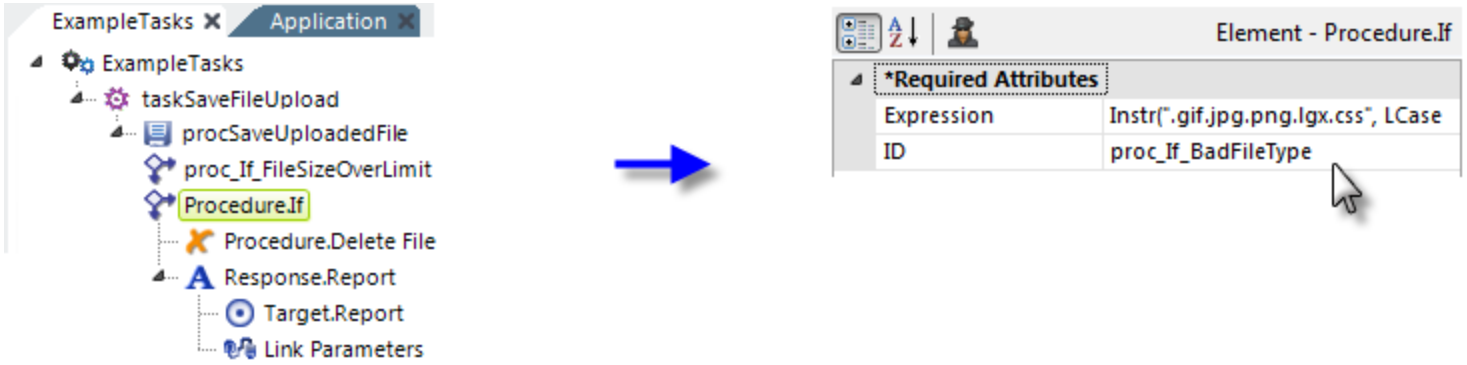


In the example above, we've added additional elements to qualify the uploaded file. A **Procedure.If** element is used to test the *size* of the uploaded file; if it's larger than 1MB, then it's deleted. The complete **Expression** attribute value used is:

```
@Procedure.procSaveUploadedFile.UploadFileLength~ > 1024000
```

Set the **Procedure.Delete File** element's **Filename** attribute to the same value as the Procedure.Save File Upload element's Filename attribute.

As before, you should also include Response, Target, and Link Parameters elements to redirect the browser to the original report and display a meaningful error message.



In the example above, more elements have been added to delete the file if the uploaded file's *extension* isn't one of five approved types. The complete **Expression** attribute value is:

```
Instr(".gif.jpg.png.lgx.css", LCase("@Procedure.procSaveUploadedFile.UploadFileExtension~")) = 0
```

In this example, the intrinsic **Instr** function is used to determine if the uploaded file's extension is in the string of concatenated approved file types.

The procedure token's file extension value is *case-sensitive*, which is why the **LCase** function is also used.

As before, you should also include Response, Target, and Link Parameters elements to redirect the browser to the original report and display a meaningful error message.

In conclusion, file uploading is best done under very controlled circumstances and you should be prepared for the likelihood that some uploads will fail.

Embedded Reports API

The **Embedded Reports API** provides developers with easy and agile methods for embedding Logi report components into other, non-Logi web pages.

The following topics provide information about the API:

- Including the API
- Embedding Static Reports using Markup
- Embedding Dynamic Reports using JavaScript
- Accessing Embedded Reports on Different Domains
- Preventing Session Timeout

 The Windows version of the **Safari** browser does not currently work with the Embed API.

Including the API

The Embedded Reports API, which is a JavaScript file, is:

```
<yourLogiApp>/rdTemplate/rdEmbedApi/rdEmbed.js
```

To use it, include a link to it at the **end** of your custom HTML page, just above the `</body>` tag:

```
1. <body>
2. ...
3. <script src="https://<yourLogiApp>/rdTemplate/rdEmbedApi/rdEmbed.js" type="text/Javascript">
4. </script>
5. </body>
```

Next, create a `div` element to contain the embedded Logi report, and give it a unique `id`attribute.

```
1. <div id="div1" />
```

Then decide which of the two following embedding approaches, described in the following sections, you want to use: **Markup** or **JavaScript**.

The Embedded Reports API works well with all modern browsers, but some features (such as auto-sizing and accessing embedded reports) will not work with older browser, such as IE7.

Embedding Static Reports using Markup

Static embedding allows you to embed a Logi report simply by manipulating the markup tags. This is done by adding the appropriate `data-*` attributes to the `div` container you've already created:

```
1. <div id="div1" data-applicationUrl="yourLogiAppURL" data-report="yourReportName" data-autoSizing="all"></div>
```

The attributes are:

Attribute	Description
data-applicationUrl	Required The URL of your Logi Info application.
data-report	Required The Logi report definition name.
data-autoSizing	Required Specifies automatic resizing of the container div to show the entire report or include scroll bars. Valid values include:`"none", "height", "width", "all"`.
data-heightOffset	Embedded content frame padding has been increased to 20px. This attribute allows you to override that value, if desired.

Attribute	Description
data-linkParams	A name-value collection of report parameters. Example: `{'Year':'2010', 'CustomerId' : 1}`. You may not use a parameter named "UserName".
data-secureKey	A SecureKey value; required when using SecureKey Authentication.
data-widthOff-set	Embedded content frame padding is 20px. This attribute allows you to override that value, if desired.

Example 1: Simple Static Embedded Report

```
<body>

div id="div1" data-applicationUrl="https://sampleapps.logianalytics.com/logiapp" data-report="EmbeddedReports.DataTable" data-autoSizing="all"></div>

<script src="https://sampleapps.logianalytics.com/logiapp/rdTemplate/rdEmbedApi/rdEmbed.js" type="text/Javascript"></script>

</body>
```

 It's very important that the <script> statement that includes rdEmbed.js be the *last line of code* before the </body> tag in your HTML or else the API will not work.

Example 2: Static Embedded Report with Parameters for use in Query

Style can also be added to the div to give it a border and background color.

```
<div id="div2" data-applicationUrl="https://sampleapps.logianalytics.com/logiapp" data-report="Embed-
dedReports.DataTable data-autoSizing="all" data-linkParams="{'CustomerID' : 'VINET', 'StartDate' : '01/01/1996'}"
class="divFrame" ></div>
```

Example 3: Using a Fixed-Size Layout

By default, embedded reports fill 100% of their div container. To control the report size, you can specify the div element's size with `width` and/or `height` style attributes. If the embedded report's physical size is greater than its div container, the report will automatically show scrollbars. You can hide the scrollbars by using style for the div element: `overflow(-x | -y): none;`.

```
<div id="div3" style="width:350px; height:150px;" data-applicationUrl="https://sampleapps.logianalytics.com/logiapp"
data-report="EmbeddedReports.DataTable" data-linkParams="{'CustomerID' : 'CHOPS'}" ></div>
```

When you specify a size in this way, you can skip the normally required `data-autoSizing` attribute.

Embedding Dynamic Reports using JavaScript

You can also use JavaScript and the Embedded Reporting object from the API to embed reports. This object has three primary methods: **create**, **get**, and **remove**. Remember that JavaScript is case-sensitive! All three methods operate with the following properties:

Property	Type	Description
applicationUrl	string	Required The URL of your Logi Info application.
report	string	Required The Logi report definition name.
autoSizing	string	Required Specifies automatic resizing of the container div to show the entire report or include scroll bars. Valid values include:`"none", "height", "width", "all"`.
heightOffset	number	Embedded content frame padding has been increased to 20px. This property allows you to override that value, if desired.
linkParams	array	A name-value collection of report parameters. Example: `{"Year":"2010", "CustomerId" : 1}`.  You may not use a parameter named "UserName".
secureKey	string	A SecureKey value; required when using SecureKey Authentication.
iframeId	string	Read Only Returns the `id` attribute value of the iFrame element that contains the embedded report.

Property	Type	Description
iframe	object	Read Only Returns the iFrame DOM object that contains the embedded report page

Using the create() and remove() Methods

1. Create a container for the report in the HTML markup:

```
<div id="reportContainer" />
```

2. In your JavaScript code, create array of report options:

```
var options = {};
options.applicationUrl = "https://sampleapps.logianalytics.com/LogiApp";
options.report = "EmbeddedReports.DataTable";
```

```
options.autoSizing = "all";

options.secureKey = "";

options.linkParams = {"CustomerID" : "VINET", "StartDate" : "01/01/2009" };
```

3. Create a report instance:

```
var report = EmbeddedReporting.create('reportContainer', options);
```

4. Remove the report instance:

```
EmbeddedReporting.remove('reportContainer');
```

Here's the JavaScript used in this page, with the onClick events for the two buttons, to create and remove the report:

```
function CreateReport() {
var options = {
applicationUrl : "https://sampleapps.logianalytics.com/LogiApp",
report : "EmbeddedReports.DataTable",
autoSizing : "all",
linkParams : { "CustomerID": "VINET", "StartDate": "01/01/1976" }
};

var report = EmbeddedReporting.create('reportContainer', options);
}
```

```
12. function RemoveReport() {
13. EmbeddedReporting.remove('reportContainer');
14. }
```

Using the get() Method

To manipulate existing reports created with the API, start by getting the report object, using its container div ID:

```
1. var report = EmbeddedReporting.get('containerId')
```

Report object properties can be directly manipulated:

```
1. EmbeddedReporting.get('containerId').report = "SalesByMonth"
```

The report object also has these methods available:

Method	Description
loadReport()	Loads or refreshes an embedded report. Example: `EmbeddedReporting.get('containerId').loadReport()`
setParam(name, value)	Updates or inserts a parameter into the report parameters collection. Example: `EmbeddedReporting.get('containerId').setParam("startDate", "01/01/2012")`
getParam(name)	Returns a parameter value from the report parameters collection. Example: `EmbeddedReporting.get('containerId').getParam("startDate")`
removeParam(name)	Removes a parameter from the report parameters collection. Example: `EmbeddedReporting.get('containerId').removeParam("startDate")`
removeAllParams()	Removes all parameters from the report parameters collection. Example: `EmbeddedReporting.get('containerId').removeAllParams()`

Accessing Embedded Reports on Different Domains

When an embedded report is hosted on a server in a different domain, you can't directly access a its DOM elements and Javascript functions due to iFrame cross-domain access policy restrictions. However, the EmbeddedReporting object provides functionality to get around those restrictions, using these methods:

execEmbeddedFunction(functionName, arg1, arg2, ...argN, callback)

Executes a JavaScript function in an embedded report and returns its execution result.

Our example Logi app contains a report definition named EmbeddedReports.Elements and it includes this JavaScript function:

```
1. function SayHello(firstName, secondName) {
2. var hello = "Hi " + firstName + " " + secondName + "!"
3. return hello;
4. }
```

We can use the execEmbeddedFunction method to remotely execute this function and then we can display its results locally.

1. Embed the report using HTML markup:

```
<div id="div7" class="divFrame" data-applicationUrl="https://sampleapps.logianalytics.com/logiapp" data-report="EmbeddedReports.Elements" data-autoSizing="height"></div>
```

and here's the report:

2. Enter values for arg1 and arg2 on the current HTML page to send to the report's function, and click Execute:

Execute this: arg1: arg2: Callback to:

Here's the supporting JavaScript included in the current HTML page and called when you click Execute:

```
function runEmbeddedFunction() {
var funcName = document.getElementById('inpExecuteThis').value;
var args1 = document.getElementById('inpArg1').value;
```

```
var args2 = document.getElementById('inpArg2').value;

var callback = document.getElementById('inpCallbackTo').value;



var report = EmbeddedReporting.get('div7');

report.execEmbeddedFunction(funcName, args1, args2, callback);

}



function showResults(result) {

alert(JSON.stringify(result, null, 2));

}
```

getElementAttribute(selector, attributeName, callback)

Returns an object with two properties: `elements` and `values`. The elements property contains a list of elements, which matched the search query. The values property contains a list of attribute values. The `selector` argument should represent a valid CSS ID selector string.

1. Embed the report using HTML markup:

```
<div id="div8" class="divFrame" data-applicationUrl="https://sampleapps.logianalytics.com/logiapp" data-report="EmbeddedReports.Elements" data-autoSizing="all" ></div>
```

and here's the report:

2. Enter a CSS selector (#elementID) to specify the element, and an attribute name, and click Execute.

Element selector: Attribute name: Callback To:

Here's the supporting JavaScript included in the current HTML page and called when you click Execute:

```
function GetElementAttribute() {
var elementSelector = document.getElementById('inpSelector').value;
```

```
var attrName = document.getElementById('inpAttrName').value;
var callback = document.getElementById('inpCallbackTo').value;


var report = EmbeddedReporting.get('div8');
report.getElementAttribute(elementSelector, attrName, callback);
}


function showResults(result) {
alert(JSON.stringify(result, null, 2));
}
```

Note that values for **Input Text Area** and **Input Select List** elements cannot be retrieved using this method; you must write a separate function to extract their values and call it with execEmbeddedFunction() instead. **setElementAttribute(selector, attributeName, attributeValue, callback)**

Sets an attribute value of element inside an embedded report. Returns "true" when the element was found and an attribute has been updated successfully. The `selector` argument should represent a valid CSS selector string.

1. We'll use the report we embedded in the previous example.

2. Enter a CSS selector (#elementID), an attribute name and value, and click Execute. Scroll up to see the result.

Selector: Attribute name: Attribute value: Callback To:

Here's the supporting Javascript included in the current HTML page and called when you click Execute:

```
function SetElementAttribute() {
var elementSelector = document.getElementById('inpSelector').value;
var attrName = document.getElementById('inpAttrName').value;
var attrValue = document.getElementById('inpAttrValue').value;
var callback = document.getElementById('inpCallbackTo').value;


var report = EmbeddedReporting.get('div8');
```

```
8.  report.setElementAttribute(elementSelector, attrName, attrValue, callback);
9.  }
10.
11. function showResults(result) {
12. alert(JSON.stringify(result, null, 2));
13. }
```

Note that, if the specified attribute does not exist, this method will create it and then set its value. Values for **Input Text Area** and **Input Select List** elements cannot be set using this method; you must write a separate function to set their values and call it with execEmbeddedFunction() instead.

Preventing Session Timeout

The parent application and its embedded Logi reports may be hosted on separate web servers and, if so, there will be independent session state maintained for them on each host. Session expiration on either host may cause problems. The Embedded Reports API provides functionality for preventing session timeout for parent and embedded reports. To use it, create a minimum web page (.htm, .html, .aspx, etc.) that has no visual elements in the parent application and use the following method to "ping" it (request the page) at specified intervals.

keepSessionsAlive(pingParentPageUrl, interval)

Pings the parent and embedded applications at the defined interval (in milliseconds). The default interval is 60000 (1 minute).

```
function CreateMyEmbeddedApp() {
var options = {
applicationUrl : "https://myAppServer/myLogiAppToEmbed",
 report : "Default",
autoSizing : "height"
};

```

```
var report = EmbeddedReporting.create("divEmbedDiv", options);



EmbeddedReporting.keepSessionsAlive("https://myParentApp/keepAlive.htm", 60000);

}
```

The .keepSessionsAlive method call must be placed *after* the .create method call in your code.

The method pings the page in the parent application and also automatically determines the URL of the embedded Logi application and pings a standard page that's supplied with the Logi Info product (so you *don't* have to create a page to ping in the embedded application).

If you have both the parent and the embedded application on the same server, be sure that you specify the same domain for them. For example, do not use https://localhost/myApp to embed the application and then https://127.0.0.1/myParentApp for the keepSessionsAlive method.

This technique only works with a single embedded Logi application; if you have reports from multiple Logi applications embedded in the same parent application, this method *will not* ping all the Logi applications.

Logi Plug-ins

"Plug-ins" give Logi developers the ability to *programmatically extend* the functionality of their Logi applications.

The following topics discuss the creation and use of Logi plug-ins:

About Plug-ins

Developers can dynamically change the behavior of their Logi applications at runtime through the use of plug-ins. Some of the things that can be done with plug-ins include:

- Modifying report definitions
- Grooming retrieved data by removing extraneous character sets
- Customizing the look and functionality of super-elements, like the Analysis Grid
- Accessing HTTP *Request* and *Session* variables
- Accessing resources such as the web server file system and OS services
- Altering the generated HTML output before it's returned to the browser

Writing a plug-in is accomplished with a development tool like **Visual Studio** or **Eclipse** and requires competence with an appropriate object-oriented language, such as C#, VB.Net, or Java. Developers who are well-acquainted with these languages often appreciate an opportunity to leverage their non-Logi skill sets. A plug-in is included in a Logi application as a .dll (.NET) or .jar (Java) file. The plug-in is loaded as a library along with the application and runs under the application's process on the web server. While a plug-in doesn't necessarily provide any specific performance advantage, it *may* do so, depending on the

circumstances. Specific elements are used in Logi definitions to call the methods in a plug-in, depending on the actions the plug-in is to carry out, and the calls are triggered by specific events. The elements and triggering events are discussed later. Generally, information is passed to the special Logi Analytics object used in plug-ins as a string or an XML document. XML document objects and XPath can then be used to manipulate the information.

Examples on DevNet

Examples of plug-in source code are included in the documents referenced at the end of this topic. In addition, DevNet includes two sample applications, one for .NET and one for Java, that demonstrate *how* plug-ins are called.

Alternatives to Plug-ins

But first, a reality check: developers considering using plug-ins may want to ask themselves if there is a simpler way to achieve the same results. Logi products offer a variety of ways to customize reports and data that *do not* require the use of external languages such as Java or C#. They include:

- Elements that can manipulate the data in datalayers after it's been retrieved, see *Datalayer Introduction*. These includes elements that can add datalayer columns with data based on calculations, columns that specifically manipulate date/time data, columns that format the data, columns containing statistical derivations, repeating elements, and more.
- Template and Definition Modifier File technology, which allows developers to dynamically alter super-element interfaces and report definitions. For more information, see "Template Modifier Files" on page 426 and "Definition Modifier Files" on page 413.
- Themes, the technology often used to apply an *appearance* to an application, uses a Template Modifier File and can also be used to apply dynamic changes to definitions, such as setting default attribute values for globalization, language translation, etc. For more information, see *Working with Themes*.

- JavaScript scripting, both inline and in external files, is also supported, allowing developers to call functions and pass/receive data from them. For more information, see "Scripting" on page 685.
- The ability to integrate external libraries, such as "jQuery" on page 567 and the Bootstrap Framework, to provide extended functionality.
- An "Embedded Reports API" on page 466, a JavaScript library that allows you to embed Logi visualizations in straight HTML pages.

If you're reluctant to tackle a Java or .NET coding project to create a plug-in, you may want to explore these alternatives.

Plug-in Elements and Triggering Events

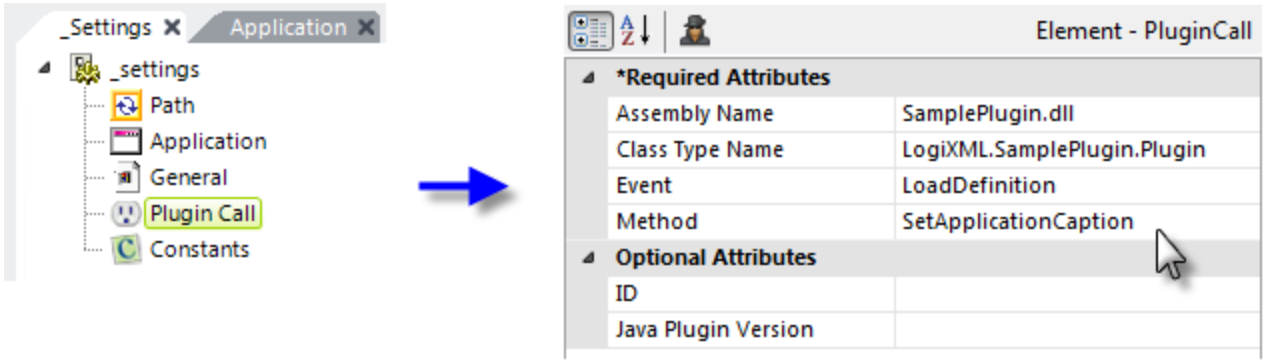
The first decision you must make is *when* to call your plug-in. Special elements are provided for calling plug-ins; you select which one to use based on events associated with them. For example, if your plug-in is designed to modify data, then you want to use the element that calls your plug-in *after* the data has been retrieved. The following table provides details about these special elements and the events they're associated with:

Element	Usage	Event
Plugin Call	Developers select from three events: A. To access the report definition XML in order to add, remove, or change elements before the report is rendered. B. To change the generated HTML, altering the output page code. C. To modify the XML data retrieved by the datalayer.	A. Select the *LoadDefinition* event: the plug-in is called when the definition file is loaded. B. Select the *FinishHtml* event: the plug-in is called just before the HTML response is sent back to the browser, or to a report export processor. C. Select the *FinishData* event: the plug-in is called when data processing finishes.
Data Layer Plugin Call	To modify the XML data retrieved by the DataLayer, with full access to web Request, Application, Session and Response objects.	The plug-in is called right after data is retrieved into the datalayer.
DataLayer.Plugin	To use a custom datalayer that retrieves data from sources that are not directly supported by other Logi datalayer elements.	The plug-in is called when datalayers are normally run, i.e. whenever the report definition is loaded or refreshed.
Generated Element	To access the underlying definition code for	The plug-in is called after its parent element is

Element	Usage	Event
Plugin Call	super-elements and the Crosstab Table.	parsed into its XML code. In the sequence of events on the Debug Trace page, this occurs just after the "View Definition" link is written.
Load Panels Plugin Call	To access the underlying definition code for Dashboard Panels.	The Plug-in is called after the Save File has been loaded, but before processing of the panels.
Panel Plugin Call	To change the definition code of its immediate parent Dashboard Panel. Provides just the panel XML code, not the complete report definition.	The plug-in is called before, or after, the Default Request Parameters and Local Data elements run, depending on the Call element's location in definition.
Procedure.Plugin Call	Runs a plug-in method from a Process definition task.	The plug-in is called when the Process task execution reaches this element.

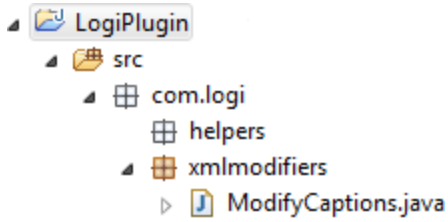
Using a Plug-in to Modify a Report Definition

Plug-in methods are called using one of the elements described in "Plug-in Elements and Triggering Events" on page 491. The element is included in the report, process, or job definition, and its attributes are set to identify the object and method to call. Let's see some examples of these elements in action. **Plugin Call** elements can be added to any definition and give developers the ability to customize the report definition before or after it's processed, or to modify data retrieved into the report's datalayers. The simple example below shows how this element can be used to call a plug-in that modifies the Application element's Caption attribute in the _Settings definition.



The Plugin Call element is a child of the definition's Root element. It has the following unique attributes:

Attribute	Description
Assembly Name	(Required) Specifies the name of the plug-in file. For .NET, provide the filename with its .dll file extension. No path information is necessary if the file is located in the _Plugins sub-folder, otherwise provide a fully-qualified path and name. If the dll is not in the _Plugins folder, you have the ability to call plugins within the applic-

Attribute	Description
	ation directory by setting the path using `@Function.AppPhysicalPath~/` . Other tokens, such as @Request, @Sessions, and @Constant can also be used to call plugins within the application directory by setting the path. For Java, provide the class name, including package, as necessary:  The example above shows the source package for a Java plug-in which has been created as `LogiPlugin.jar` and copied into *`yourLogiApp`*`/WEB-INF/lib`. The Assembly Name attribute value would then be `com.logi.xmlmodifiers.ModifyCaptions`.
Class Type Name	(Required for .NET) Consists of the root namespace + "." + the name of your plug-in class.
Event	(Required) Specifies the event that triggers the calling of this plug-in. Options include: *LoadDefinition* - when the report definition has been loaded, before any Logi engine processing *FinishHtml* - when the HTML response is ready to be sent back to the browser or to a report export processor. *FinishData* - when datalayer processing finishes.
Method	(Required) Specifies the case-sensitive name of the method to be called in your plug-in.
Java Plugin	(Java only) Specifies which version of the underlying plug-in class to use. Select *Version10* here to use the

Attribute	Description
Version	LogiPluginObjects10 class, which has additional methods such as addDebugMessage, replaceTokens and getSettingsDefinition.

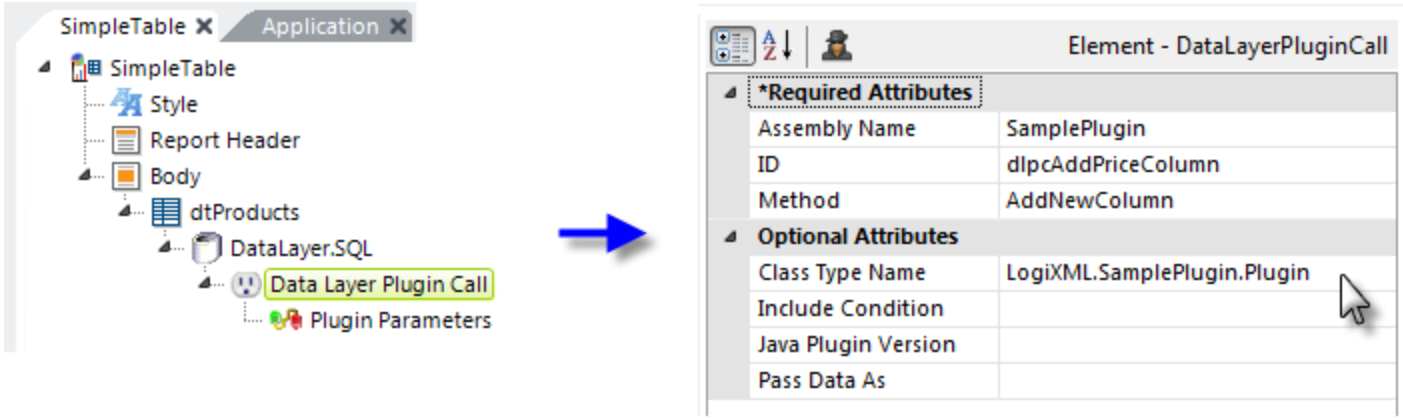
When developing a .NET plug-in in Visual Studio, the root namespace defaults to the project name, but you can set it explicitly in the Projects → Properties page in VS.

Location Dependencies

The class object's *CurrentDefinition* value is that of the report definition containing the Plugin Call element used to call the plug-in. For example, if a Plugin Call element is placed in the Customers report definition, the *CurrentDefinition* field contains the string representation of the Customers report definition. The class's *Request* and *Session* values have application-wide scope and are not affected by the location of the Plugin Call element.

Using a Plug-in to Modify Data

This example shows how to call a plug-in to modify data that has been retrieved into a datalayer and cached as XML. This is done using the **Data Layer Plugin Call** element.



As shown in the example above, the **Data Layer Plugin Call** element is a child of a datalayer element. Some of its attributes are the same as those for the Plugin Call element. Why use this element instead of the Plugin Call element, which can also be used to modify data after retrieval? This element has several additional attributes that Plugin Call does not. One of its unique attributes is **PassDataAs**; this attribute governs how data retrieved by the datalayer is made available to the plug-in:

- When set to *XmlDocument* (the default value), the data will be passed to the plug-in as an in-memory XML document object and will be available in the plug-in code as the *CurrentData* property.
- When set to *FileName*, the data will be identified to the plug-in by its XML data cache file name, which the plug-in can then access using XML stream readers and writers. The class object's *CurrentDataFile* and *ReturnedDataFile* values will be populated with the fully qualified paths and filenames to these files. This option is provided to reduce the amount of memory used when manipulating very large datasets (100,000+ rows).

Another of its attributes is **Include Condition**, which allows you to conditionally enable the plug-in call. You can enter a formula here and the plug-in will only be enabled when it evaluates to *True*. You can also pass parameters to your plug-in using the child **Plugin Parameters** element, shown above. This allows you to dynamically affect the behavior or output of the plug-in at runtime.

Creating the Plug-in

Now that you have an idea of how and when plug-ins are called, you're ready to create one. A plug-in consists of a function wrapper, which instantiates a plug-in class as an object, and exposes its fields or methods. The classes and development techniques used to create a plug-in are described separately in these DevNet topics: "Create .NET Plug-in" on the next page and "Create a Java Plug-in" on page 530.

The .NET and Java classes are functional duplicates, though their method/field names differ. The topics linked above include source code examples for specific purposes.

Create .NET Plug-in

"Plug-ins" give Logi developers the ability to *programmatically extend* the functionality of their Logi applications.

The following topics discuss the development of Logi plug-ins using the .NET framework:


If you haven't done so already, you should read "Logi Plug-ins" on page 488 for important supporting information before proceeding. If you're developing a **Java** plug-in, see "Create a Java Plug-in" on page 530 instead of this one.

.NET Plug-in - Writing Your Plug-in

Plug-ins written for Logi .NET applications are compiled into Dynamic Link Library (.DLL) files. In the code, the **System.Web** and **System.XML.Linq** .NET *namespaces* are referenced for interactions with Logi applications and report definitions. Namespaces are collections of classes with methods for doing things with specific kinds of data or objects. You import or include them in your plug-in project.

For example, System.Web is needed to access *HTTP Request* and *Session* objects, which are often useful in a plug-in.

Logi definitions and data are handled as XML streams during processing by the web server and therefore by your plug-in, too. Much of your code will be concerned with searching, parsing, and modifying this XML. The System.XML.Linq namespace provides useful methods for doing this. If you're familiar with XPath notation, you'll find that to be very helpful as well.

Create your plug-in project in Visual Studio:

1. Create a new Class Library project, using C#, Visual Basic, or other language of your choice.
2. Configure the project to build the .DLL output in the `_Plugins` folder of the Logi application (which you may need to create). The folder's name *must* be "`_Plugins`", beginning with an underscore.
3. Add a reference to the `System.Web` and `System.XML.Linq` .NET namespaces. Both may not be needed, depending on what you do in the plug-in, but you can refine that later.
4. Add a reference to the file *`yourLogiApp`*`/bin/rdPlugin.dll` distributed with your Logi product.

   A "strongly-signed" plug-in can be created by referencing *`yourLogiApp`*`/bin/rdPluginSigned.dll`

   Be sure to use the file found in the `bin` folder of the Logi application that will use this plug-in. It's version-specific so don't refer to it in the `bin` folder of some other Logi application.

5. Write your plug-in code, starting with this prototype:

```
[Visual Basic]
Imports System.Xml.Linq
Imports System.WebPublic Class Plugin
 Public Sub yourMethodName(ByRef yourObjName As rdPlugin.rdServerObjects)
' your method code goes here
End Sub
End Class
[C#]
using System;
using System.Xml.Linq;
using System.Web;
using rdPlugin;

namespace myPlugin
{
 public class Plugin
 {
 public void yourMethodName(ref rdPlugin.rdServerObjects yourObjName)
{
 // your method code goes here
 }
 }
}
```

Add the methods and code you need to get the plug-in to do whatever it's supposed to do.

.NET Plug-in - Example: Change the Application Caption

The following are code examples for a plug-in that alters the caption of the Logi application at runtime:

```
[Visual Basic]
Imports System.Xml.Linq

Public Class Plugin

 Public Sub SetApplicationCaption(ByRef rdObjects As rdPlugin.rdServerObjects)
 Dim xmlSettings As New XmlDocument()
 xmlSettings.LoadXml(rdObjects.CurrentDefinition)
 Dim eleApp As XmlElement = xmlSettings.SelectSingleNode("//Setting/Application")
 eleApp.SetAttribute("Caption", "Greetings from the Sample Plugin! Time: " & Now.ToString)
 rdObjects.CurrentDefinition = xmlSettings.OuterXml
 End Sub

End Class

[C#]
using System;
using System.Xml.Linq;
using rdPlugin;

namespace SamplePlugin
{
 public class Plugin
```

```
 {
 public void setApplicationCaption(ref rdServerObjects rdObjects)
 {
 XmlDocument xmlSettings;
 XmlElement eleApp;
 string currentTime;

 // Load the current report definition into an XmlDocument object
 xmlSettings = new XmlDocument();
 xmlSettings.LoadXml(rdObjects.CurrentDefinition);

 // Locate the Application element and define its Caption attribute
 eleApp = (XmlElement)xmlSettings.SelectSingleNode("//Setting/Application");
 currentTime = DateTime.Now.ToShortTimeString();
 eleApp.SetAttribute("Caption", "Hello from the Sample Plugin! Time: " + currentTime);

 // Save the new report definition
 rdObjects.CurrentDefinition = xmlSettings.OuterXml;
 }
 }
}
```

.NET Plug-in - Example: Modify an SQL Query with a Request Variable

The following are code examples for a plug-in that customizes a SQL query based on a request variable:

```
[Visual Basic]
Imports System.Xml.Linq

Public Class Plugin

 Public Sub SetCustomerQuery(ByRef rdObjects As rdPlugin.rdServerObjects)
 Dim xmlDefinition As New XmlDocument()
 xmlDefinition.LoadXml(rdObjects.CurrentDefinition)
 Dim xpath As String = "//Report/Body/DataTable/DataLayer"
 Dim eleDataLayer As XmlElement = xmlDefinition.SelectSingleNode(xpath)
 If IsNothing(eleDataLayer) Then
 Throw New Exception("The report is missing the DataLayer element.")
End If

 'Use a Request variable to set set the SELECT query.
 Dim sSelect As String
 Select Case rdObjects.Request("Continent")
 Case "NA"
 sSelect = "SELECT * FROM Customers WHERE Country IN('USA','Mexico','Canada')"
 Case "SA"
 sSelect = "SELECT * FROM Customers WHERE Country IN('Argentina','Brazil','Venezuela')"
 Case "EU"
 sSelect = "SELECT * FROM Customers WHERE Country IN('UK','France','Spain')"
```

```
 Case Else
 sSelect = "SELECT * FROM Customers"
 End Select

 eleDataLayer.SetAttribute("Source", sSelect)
 rdObjects.CurrentDefinition = xmlDefinition.OuterXml
 End Sub

End Class
[C#}
using System;
using System.Xml.Linq;
using rdPlugin;

namespace SamplePlugin
{
 public class Plugin
 {
 public void setCustomerQuery(ref rdServerObjects rdObjects)
{
 XmlDocument xmlDefinition;
XmlElement eleDataLayer;
 string sSelect, xpath;

 // Load the current report definition into an XmlDocument object
 xmlDefinition = new XmlDocument();
```

```
xmlDefinition.LoadXml(rdObjects.CurrentDefinition);

 // Locate the DataLayer element in the specified XML tree
 xpath = "//Report/Body/DataTable/DataLayer";
 eleDataLayer = (XmlElement)xmlDefinition.SelectSingleNode(xpath);
if (eleDataLayer == null)
throw new Exception("The report is missing the DataLayer element.");

 // Use a Request variable to set the SELECT query
 switch (rdObjects.Request["Continent"])
 {
 case "NA":
 sSelect = "SELECT * FROM Customers WHERE Country IN('USA','Mexico','Canada')";
 break;
 case "SA":
 sSelect = "SELECT * FROM Customers WHERE Country IN('Argentina','Brazil',
'Venezuela')";
 break;
 case "EU":
 sSelect = "SELECT * FROM Customers WHERE Country IN('UK','France','Spain')";
break;
 default:
 sSelect = "SELECT * FROM Customers";
 break;
 }
```

```
 // Save the SQL query to the report definition
 eleDataLayer.SetAttribute("Source", sSelect);
 rdObjects.CurrentDefinition = xmlDefinition.OuterXml;
 }
 }
}
```

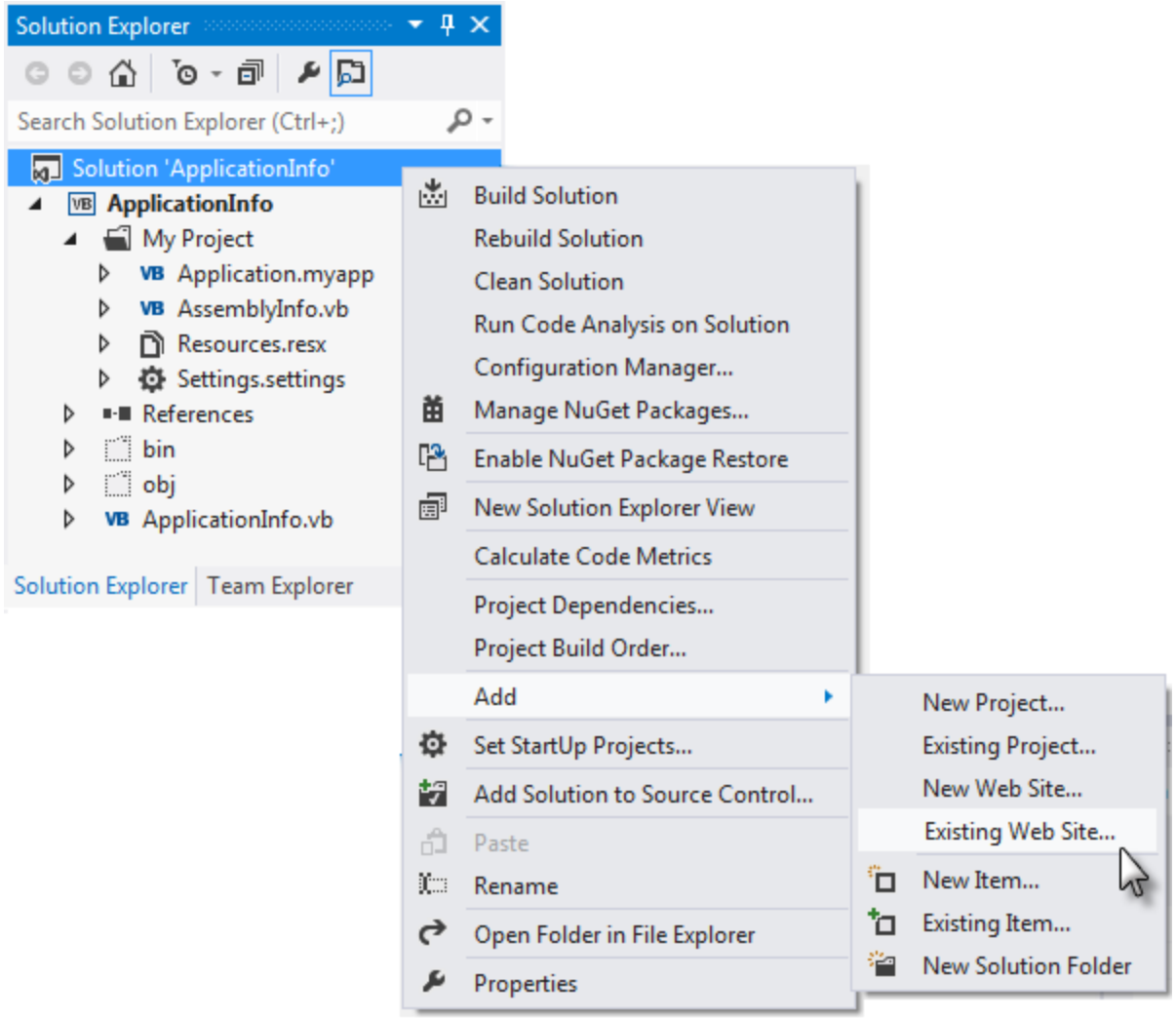
.NET Plug-in - Implementing Your Plug-in

When your plug-in has been written and compiled, add the element(s) you need to call it in your Logi application. The elements used to do this are described in "Logi Plug-ins" on page 488.

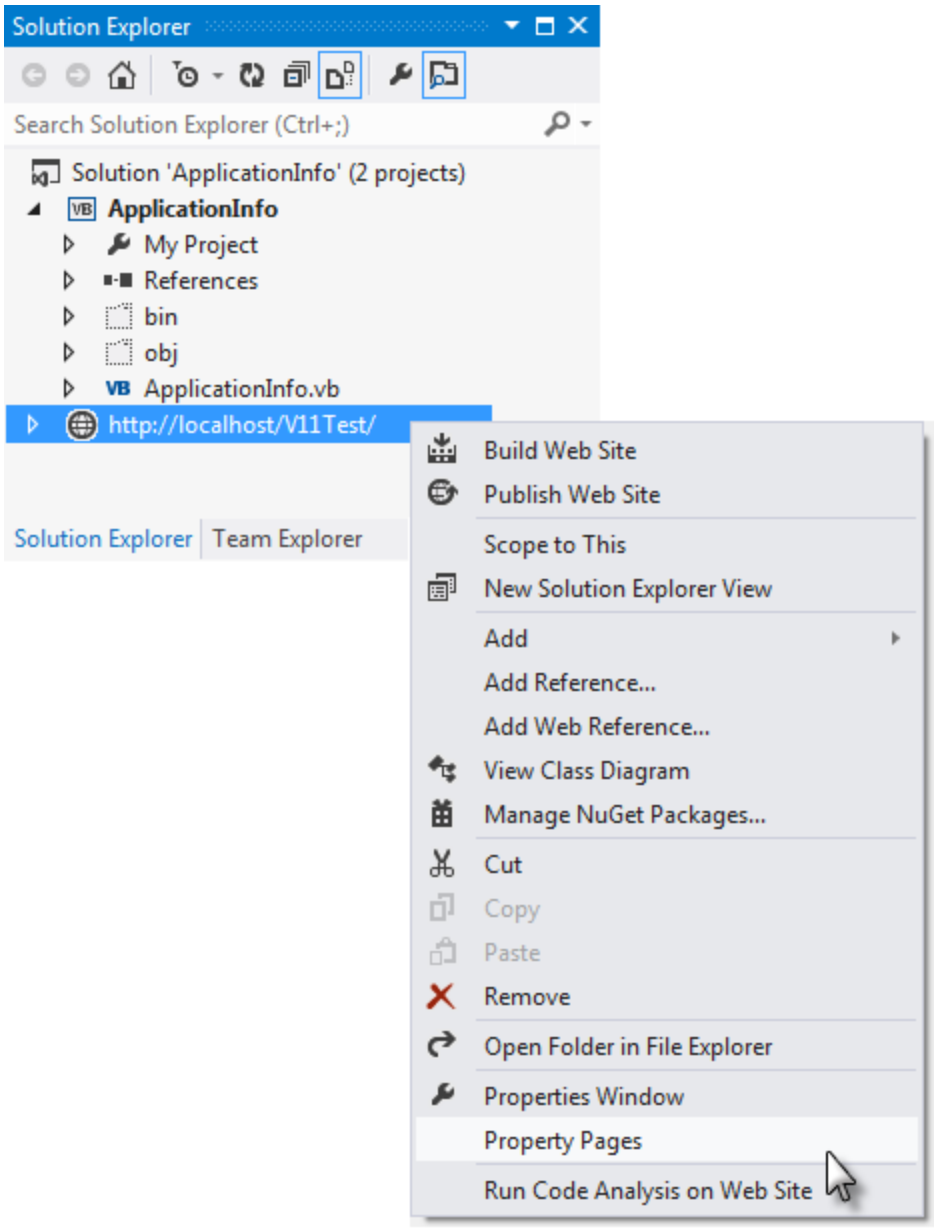

The first time you run your Logi application, IIS will load the plug-in into memory and it will remain cached there. If you make coding changes to the plug-in and want to rebuild it, you'll need to run **stop** and **restartIIS** (run `iisreset.exe`) first, in order to be able to replace the previous build.

.NET Plug-in - Debugging in Visual Studio 2012

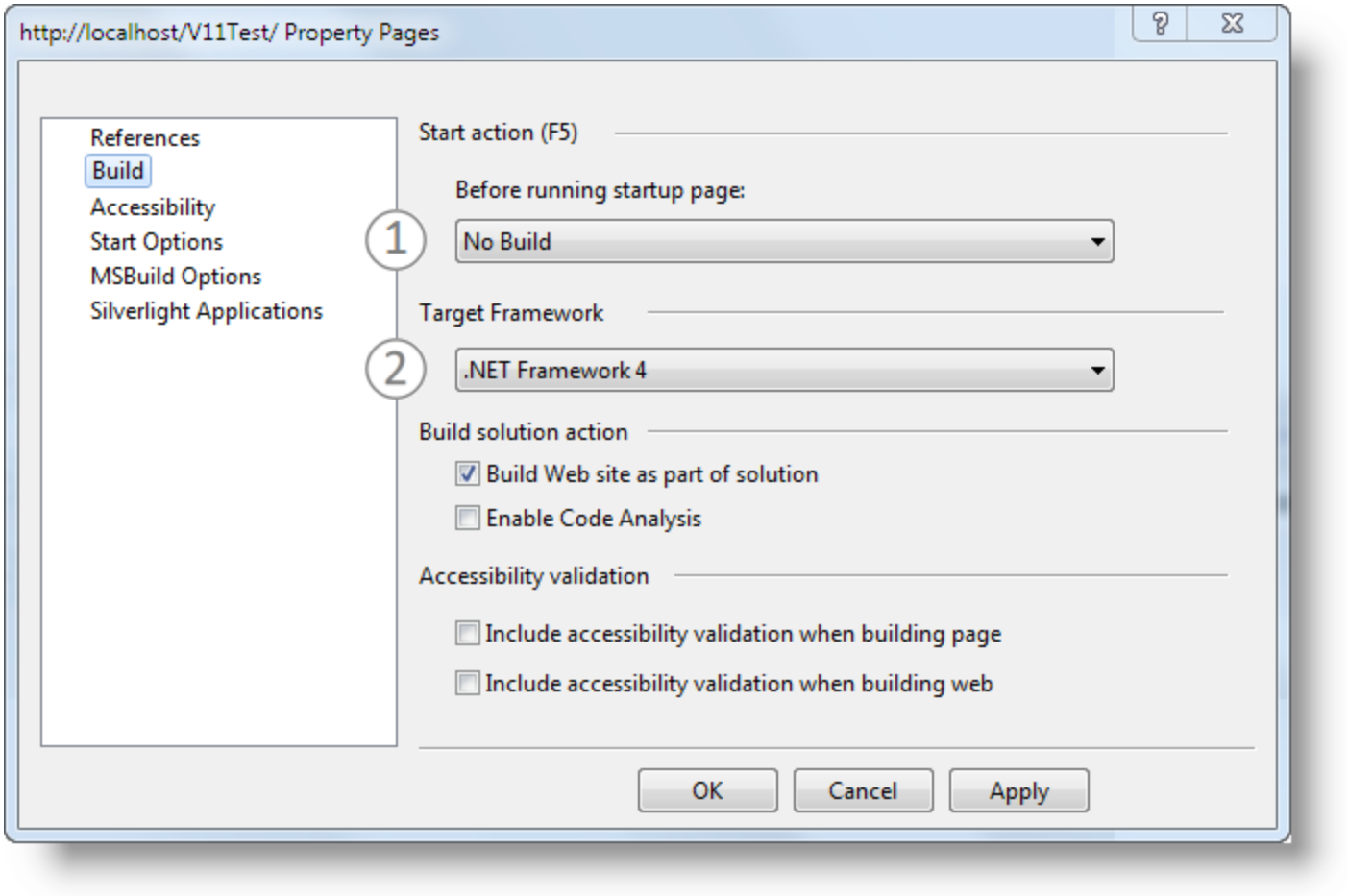
You can debug your plug-in as it runs in your Logi application, using Visual Studio (VS). The examples in this topic show you how it's donein VS 2012. Ensure that you're running VS with a user account that has Administrator privileges.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ApplicationInfo'
ApplicationInfo
My Project
Application.myapp
AssemblyInfo.vb
Resources.resx
Settings.settings
References
bin
obj
ApplicationInfo.vb
Solution Explorer
Team Explorer
Build Solution
Rebuild Solution
Clean Solution
Run Code Analysis on Solution
Configuration Manager...
Manage NuGet Packages...
Enable NuGet Package Restore
New Solution Explorer View
Calculate Code Metrics
Project Dependencies...
Project Build Order...
Add
Set StartUp Projects...
Add Solution to Source Control...
Paste
Rename
Open Folder in File Explorer
Properties
New Project...
Existing Project...
New Web Site...
Existing Web Site...
New Item...
Existing Item...
New Solution Folder

Open your plug-in solution in VS and, in the Solution Explorer, select and right-click the root node, then select **Add** → **Existing Web Site...** Click the **Local IIS** category and select your Logi application web site's virtual directory from the list, adding it to your solution.

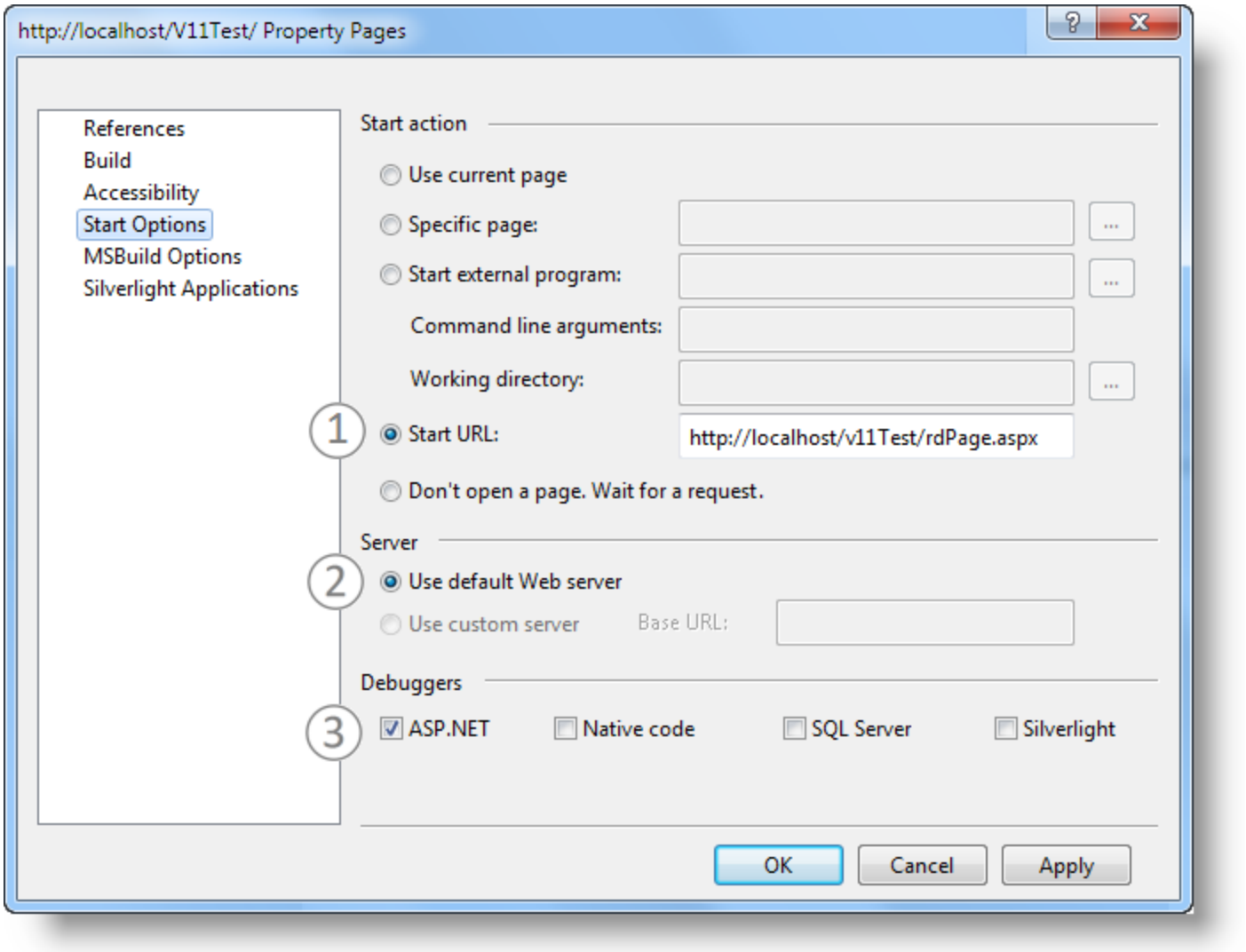

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ApplicationInfo' (2 projects)
ApplicationInfo
My Project
References
bin
obj
ApplicationInfo.vb
http://localhost/V11Test/
Solution Explorer
Team Explorer
Build Web Site
Publish Web Site
Scope to This
New Solution Explorer View
Add
Add Reference...
Add Web Reference...
View Class Diagram
Manage NuGet Packages...
Cut
Copy
Paste
Remove
Open Folder in File Explorer
Properties Window
Property Pages
Run Code Analysis on Web Site

Right-click the Logi application web site in the Solution Explorer and select its **Property Pages** from the pop-up menu, as shown above.



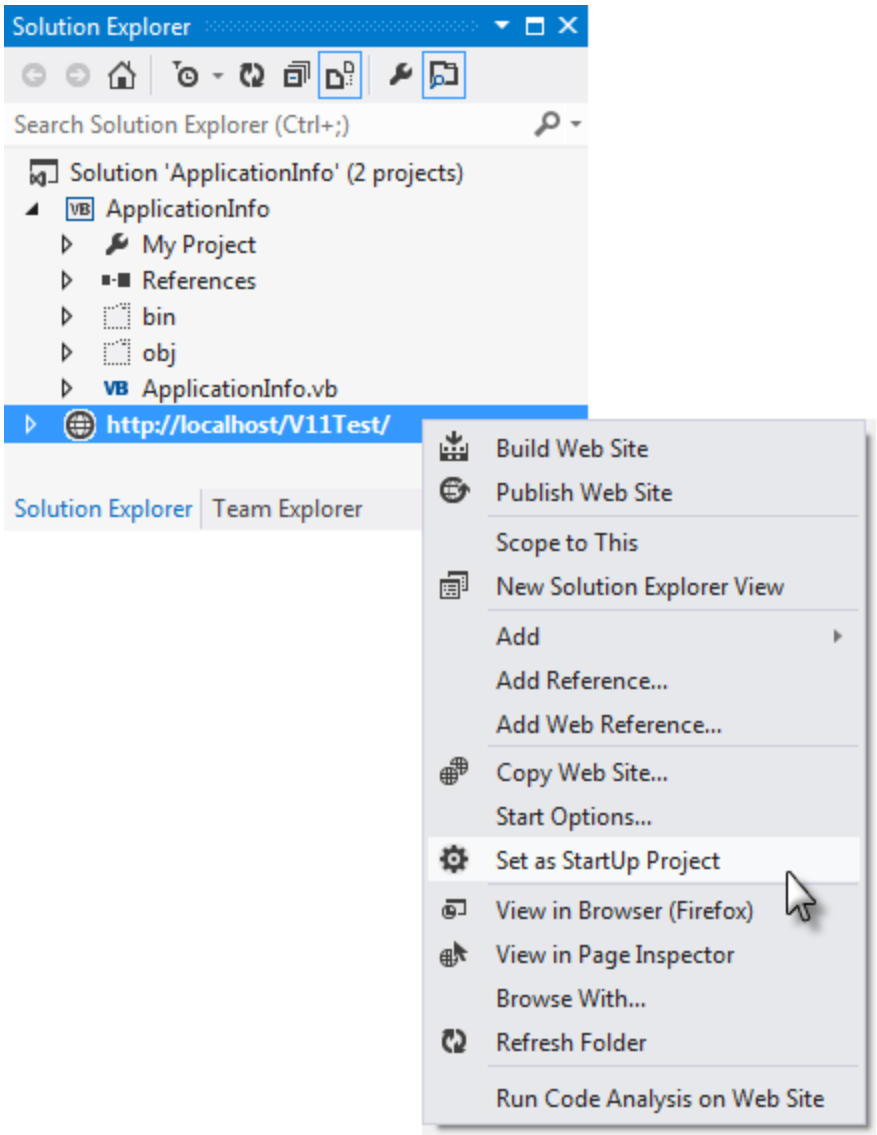
In the Property Pages window, shown above, select the **Build** page in the left menu, and then:

1. Select **No Build** in the list of Start actions.
2. Ensure that the appropriate .NET Framework version for your Logi application is selected.



Next, select the **Start Options** page in the left menu, as shown above, and then:

1. Select the **Start URL** radio button, and enter your Logi application's URL.
2. Ensure that the **Use default Web server** option is selected.
3. Ensure that the **ASP.NET** debugger option is checked. Click **OK** to save all your settings



Finally, in the Solution Explorer, right-click your Logi application web site, and select the **Set as Startup Project** option.

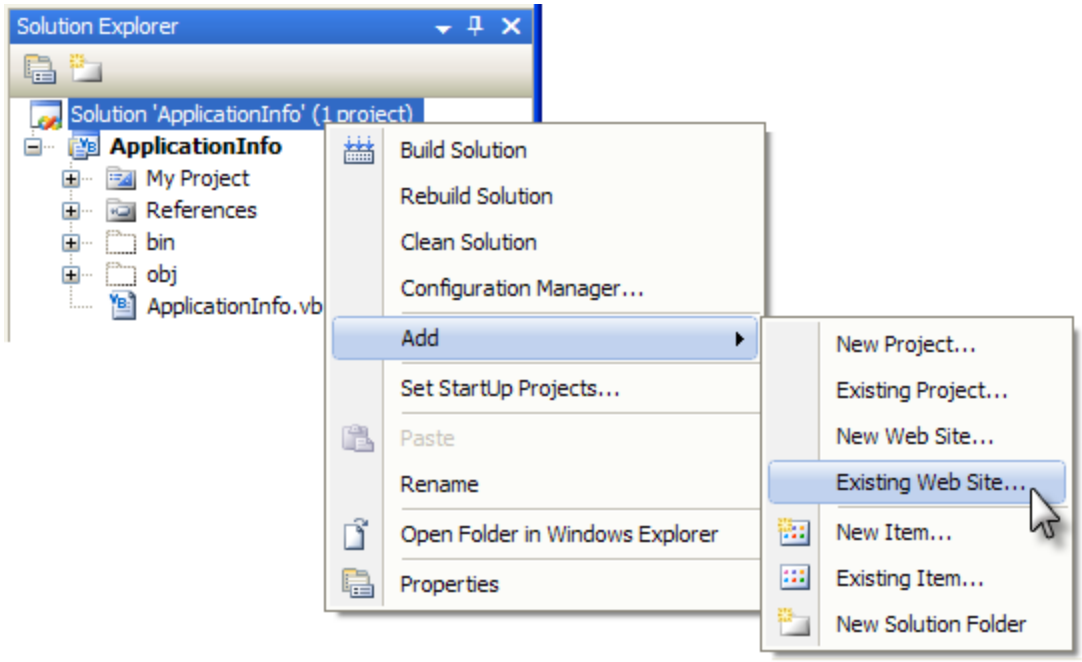
You're now ready to press **F5** (or use the Debug menu) to run your solution and start debugging. The details of setting Watches, Breakpoints and other debugging-related activities are beyond the scope of this topic; refer to your Visual Studio documentation.

If you start debugging and receive the error "...Debugging failed because integrated Windows authentication is not enabled", read this MSDN article for information about configuring the authentication for your Logi app virtual directory.

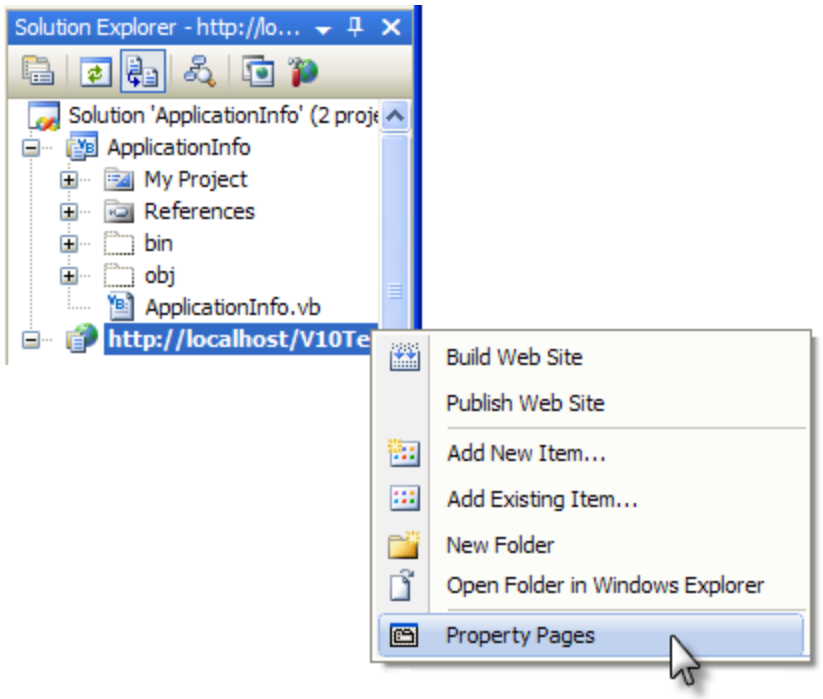
These settings are for a simple debugging example. Depending on the implementation of your Logi application, such as use of a remote web server, you may need to adjust some of the settings shown above.

.NET Plug-in - Debugging in Visual Studio 2008/5

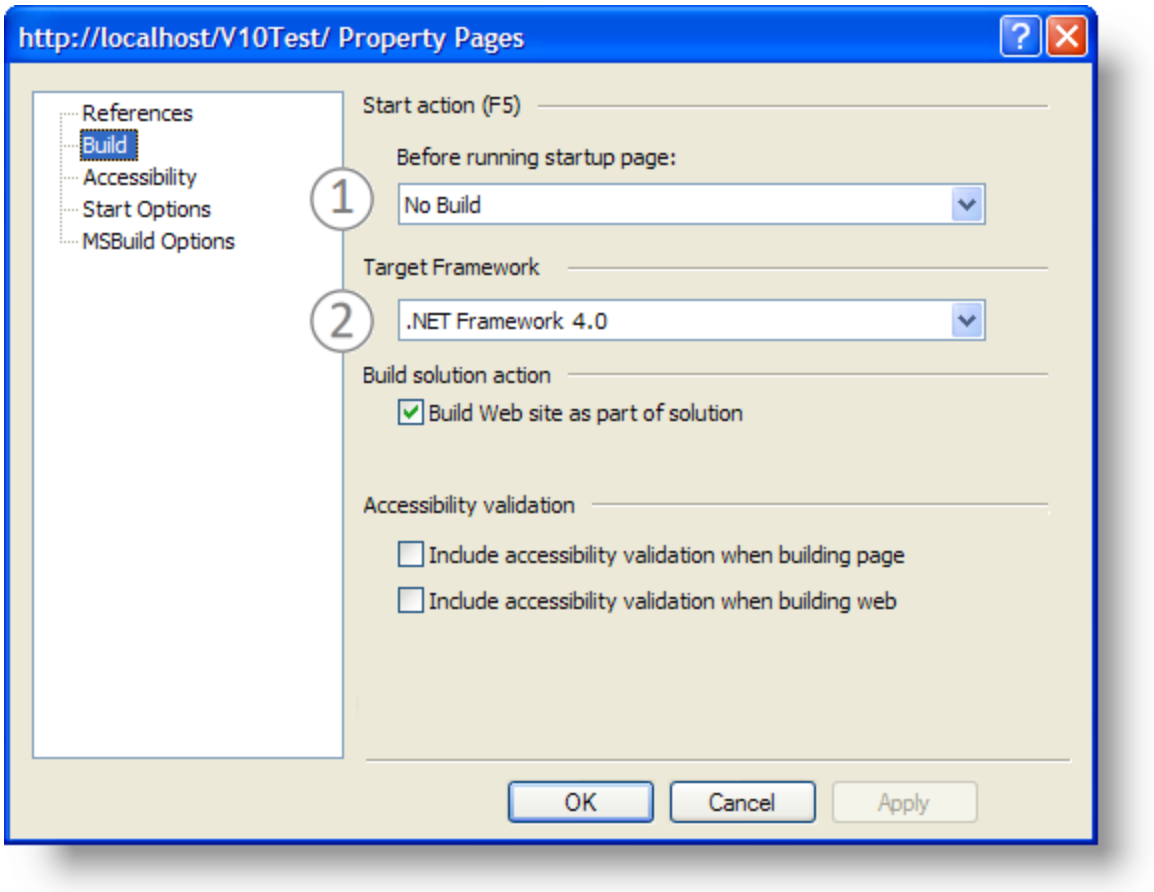
You can debug your plug-in as it runs in your Logi app, using Visual Studio (VS). The technique is the same regardless of VS version, with minor differences as noted below. Ensure that you're running VS with a user account that has Administrator privileges.



Open your plug-in solution in VS and, in the Solution Explorer, select and right-click the root node, then select **Add** → **Existing Web Site...** Browse to and select your Logi application web site (its virtual directory), adding it to your solution.

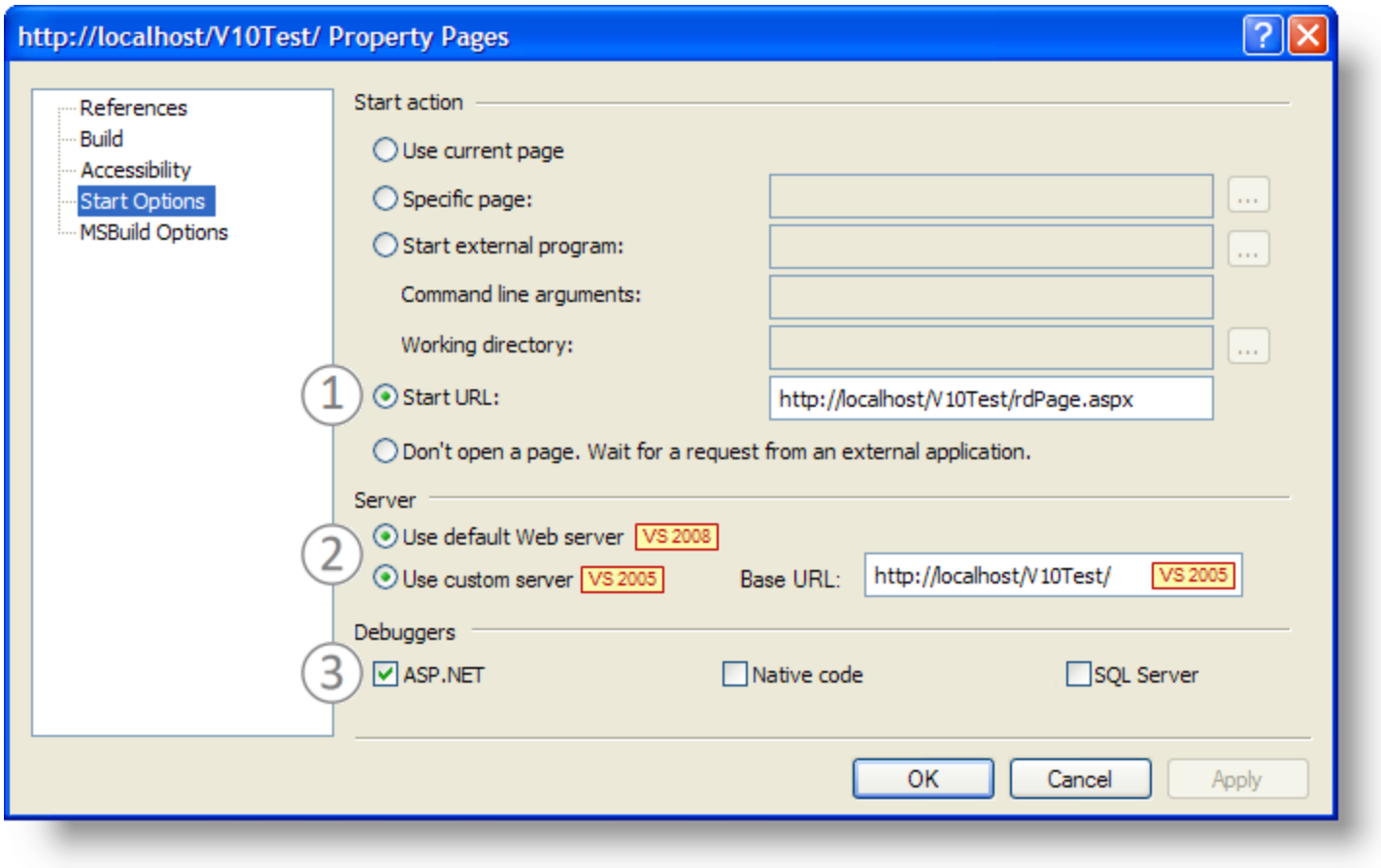


Right-click the Logi application site in the Solution Explorer and select its **Property Pages** from the pop-up menu, as shown above.



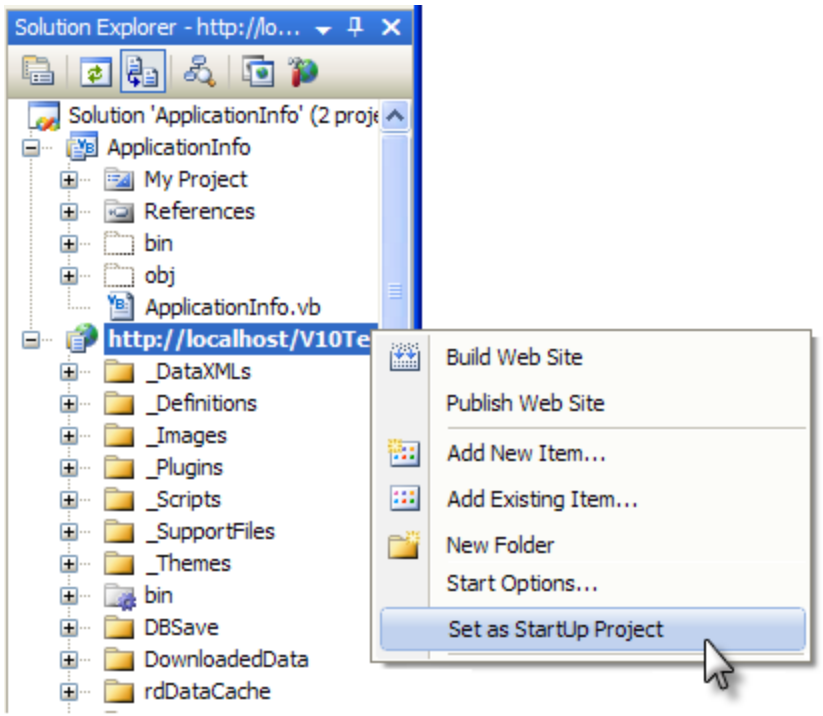
In the Property Pages window, shown above, select the **Build** page in the left menu, and then:

1. Select **No Build** in the list of Start actions.
2. Ensure that the appropriate .NET Framework version for your Logi application is selected.



Next, select the **Start Options** page in the left menu, as shown above, and then:

1. Select the **Start URL** radio button, and enter your Logi application's URL.
2. To run your Logi app using the local IIS server:
   VS 2008: Select the **Use default Web server** radio button.
   VS 2005: Select the **Use custom server** radio button and enter the **Base URL**, as shown above.
3. Check the **ASP.NET** debugger check box; click **OK** to save all your settings



Finally, in the Solution Explorer, right-click your Logi application web site, and select the **Set as Startup Project** option.

You're now ready to press **F5** (or use the Debug menu) to run your solution and start debugging. The details of setting Watches, Breakpoints and other debugging-related activities are beyond the scope of this topic; refer to your Visual Studio documentation.

If you start debugging and receive the error "...Debugging failed because integrated Windows authentication is not enabled", read this MSDN article for information about configuring the authentication for your Logi app virtual directory.

These settings are for a simple debugging example. Depending on the implementation of your Logi application, such as use of a remote web server, you may need to adjust some of the settings shown above.

If you haven't done so already, you should read "Logi Plug-ins" on page 488 for important supporting information before proceeding. If you're developing a **Java** plug-in, see "Create a Java Plug-in" on page 530 instead of this one.

.NET Plug-in - Plug-in Class Properties and Methods

Logi plug-ins for .NET are written in Visual Studio or a similar development tool, using the rdServerObjects class, which resides in the `rdPlugin.dll` assembly and is a member of the `rdPlugin` namespace. You instantiate this class in your code and it provides access to the Logi application and its data. The class object is defined in your code as:

```
[Visual Basic]
public Class rdServerObjects[C#]
public class rdServerObjects
```

Once instantiated, the object provides the following properties and methods for use with your custom code:

- AddDebugMessage()
- CurrentData
- CurrentDataFile
- CurrentDefinition
- CurrentElement
- CurrentHttpContext
- PluginParameters
- ReplaceTokens()
- Request
- ResponseHtml
- ReturnedDataFile
- Session
- SettingsDefinition

Here are the object's methods and parameters:

AddDebugMessage()

This method inserts an entry into the Debugger Trace Report: `[Visual Basic]`

```
Public Sub AddDebugMessage(Optional ByVal CurrentEvent As String,
 Optional ByVal ProgramObject As String,
 Optional ByVal ObjectValue As String,
 Optional ByVal More Info As Object )[C#]
public void AddDebugMessage(string CurrentEvent, string ProgramObject, string ObjectValue, object More Info);
```

Parameters:

`CurrentEvent` - text to be inserted in the Debug Trace page's Event column

`ProgramObject` - text to be inserted in the Debug Trace page's Object column

`ObjectValue` - text to be inserted in the Debug Trace page's Value column

`More Info` - object (XML data or XSL code) that is displayed via link inserted in the Debug Trace page's Value column

The time value for the entry inserted into the Debugger is provided automatically.

CurrentData

Returns an XmlDocument object containing the data in all of the datalayers. `[Visual Basic]`

```
Public CurrentData As System.Xml.XmlDocument[C#]
public System.Xml.XmlDocument CurrentData;
```

CurrentDataFile

Returns a string object containing of the fully-qualified path and filename for the cache file containing the XML data retrieved

by the datalayer. [Visual Basic]

```
Public CurrentDataFile As String [C#]
public string CurrentDataFile;
```

CurrentDefinition

Returns a string object containing the XML source code of the current report definition. [Visual Basic]

```
Public CurrentDefinition As String[C#]
public string CurrentDefinition;
```

CurrentElement

Returns an XmlElement object containing the XML source code of the current element. Only available when using Data Layer Plugin Call and Procedure.Plugin Call elements. [Visual Basic]

```
Public CurrentElement As System.Xml.XmlElement [C#]
public System.Xml.XmlElement CurrentElement;
```

CurrentHttpContext

Returns an XmlDocument object containing all HTTP-specific information, including Application, Session, Server, Request, and Response objects. [Visual Basic]

```
Public CurrentHttpContext As System.Web.XmlDocument[C#]
public System.Web.XmlDocument CurrentHttpContext;
```

PluginParameters

Returns a Hashtable object containing the parameters, if any, passed to the plug-in. [Visual Basic]

```
Public PluginParameters As System.Collections.Hashtable[C#]
```

```
public System.Collections.Hashtable PluginParameters;
```

ReplaceTokens()

Returns a string object containing the `sTokens` parameter with some or all of its tokens replaced with their current values, and with other optional effects applied. `[Visual Basic]`

```
Public Function ReplaceTokens(ByVal sTokens As String,
 Optional ByVal bDuplicateSingleQuotes As Boolean = False,
 Optional ByVal eleDataLayerRow As System.Xml.XmlElement,
 Optional ByVal bDuplicateBracketedQuotes As Boolean = False,
 Optional ByVal bAssumeNonQuotedAsNumeric As Boolean = False,
 Optional ByVal TokenList() As String,
 Optional ByVal bSqlInjectionGuard As Boolean = False) As String[C#]
public string ReplaceTokens(string sTokens,
 bool bDuplicateSingleQuotes,
 XmlElement eleDataLayerRow,
 bool bDuplicateBracketedQuotes,
 bool bAssumeNonQuotedAsNumeric,
 string TokenList,
 bool bSqlInjectionGuard);
```

Parameters:

`sTokens` - string containing Logi tokens, such as @Data.LastName~, to be replaced with current values.

`bDuplicateSingleQuotes` – if *True*, causes any single-quotes in `sTokens` to be doubled in return string.

`eleDataLayerRow` - XmlElement containing the data that will be used to replace the @Data tokens in `sTokens`.

`bDuplicateBracketedQuotes` - if *True*, double-quotes embedded in a token value will be included in return string.

`bAssumeNonQuotedAsNumeric` - if *True*, replacement processing will handle unquoted, numeric token values as numbers.

`TokenList` - comma-separated list of token types, indicating the types to be replaced; other types will be skipped.

`bSqlInjectionGuard` - if *True*, replacement processing checks all token values for potentially dangerous SQL strings.

Request

Returns an HttpRequest object, which can be used to access HTTP Request values sent by the client to a Logi application.

```
[Visual Basic]
Public Request As System.Web.HttpRequest[C#]
public System.Web.HttpRequest Request;
```

ResponseHtml

Returns a String object containing the HTML for the rendered response page; available when the *FinishHtml* event completes.

```
[Visual Basic]
Public ResponseHtml As String[C#]
public String ResponseHtml;
```

ReturnedDataFile

Returns a String object containing the fully-qualified path and filename for the cache file which contains the manipulated XML data after processing. `[Visual Basic]`

```
Public ReturnedDataFile As String [C#]
public string ReturnedDataFile;
```

Session

Returns an HttpSessionState object, which can be used to access Session variable values related to the current Logi application session. `[Visual Basic]`

```
Public Session As System.Web.SessionState.HttpSessionState[C#]
```

```
public System.Web.SessionState.HttpSessionState Session;
```

SettingsDefinition

Returns a read-only XmlDocument object containing the XML of the _Settings definition. [Visual Basic]

```
Public ReadOnly Property SettingsDefinition() As System.Xml.XmlDocument[C#]
public System.Xml.XmlDocument SettingsDefinition;
```

Create a Java Plug-in

"Plug-ins" give Logi Info developers the ability to *programmatically extend* the functionality of their Logi applications.

The following topics discuss the development of Logi plug-ins using Java:

- Writing Your Plug-in
- Example: Change the Application Caption
- Example: Modify an SQL Query with a Request Variable
- Implementing Your Plug-in
- Debugging Example: Eclipse and Tomcat
- Plug-in Class Methods and Properties

If you haven't done so already, you should read "Logi Plug-ins" on page 488 for important supporting information before proceeding. If you're developing a **.NET** plug-in, see "Create .NET Plug-in" on page 499 instead of this one.

Java Plug-in - Writing Your Plug-in

Plug-ins written for Logi applications on Java platforms are included as either standalone `.class` files, in the Logi application's `WEB-INF/classes` folder, or as a class file within a `.jar` file in the Logi application's `WEB-INF/lib` folder.

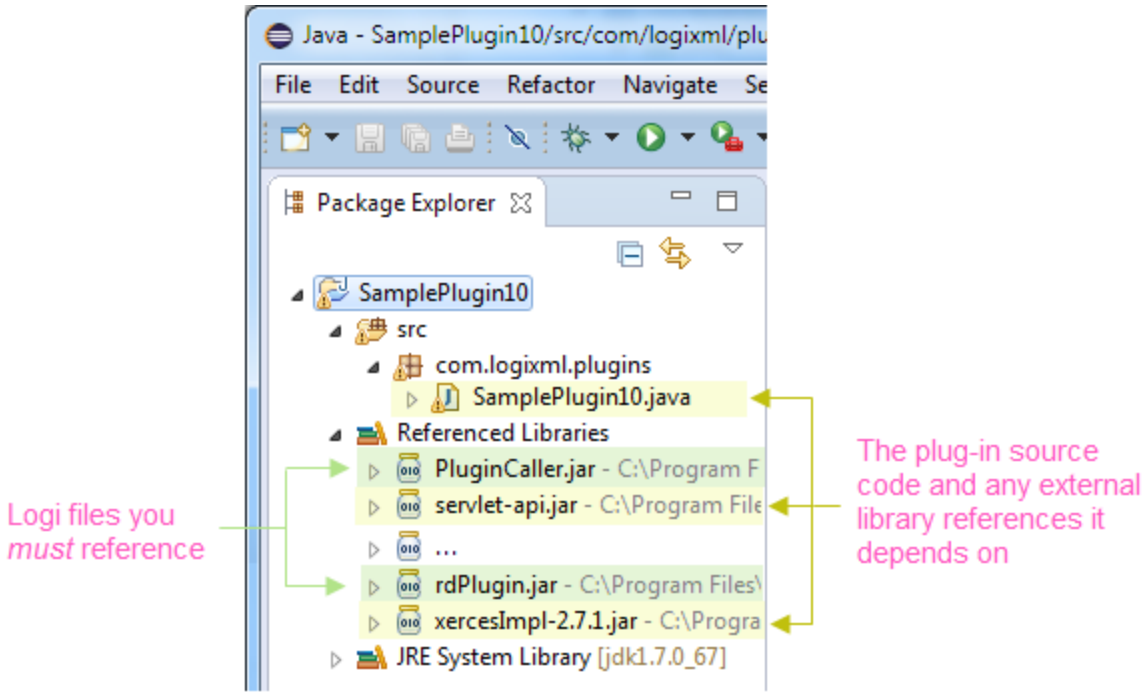
Many copies of the files that define the LogiPluginObjects and LogiPluginObjects10 classes, `LogiPluginObjects.java` and `LogiPluginObjects10.java`, may exist on your hard drive if you're working with multiple Logi applications. To ensure version compatibility, be sure to use the file found in the folders of the Logi application that will use the plug-in you're creating.

Logi definitions and data are handled as XML streams during processing by the web server and therefore by your plug-in, too. Much of your code will be concerned with searching, parsing, and modifying this XML. If you're familiar with XPath notation, you'll find that to be very helpful as well.

Create your plug-in project in Eclipse

Here's an example of the steps needed to create a plug-in using the popular Eclipse IDE:

1. Create a standard Java Project. The Project Name is inconsequential if you're compiling to a single class file. The version of the JDK used in the plug-in project should match the JDK version used to run the Logi application on your web server.



2. Add the necessary source code files and library references to your project. For example, to create our Plug-in (Java) sample application, we added the files shown above. You *must* include these two Logi files:

   `yourLogiApp/WEB-INF/lib/PluginCaller.jar`

   `yourLogiApp/WEB-INF/lib/rdPlugin.jar`

   We also had to include the other references to several .jar files to satisfy the requirements of our sample code.

3. Eclipse requires that the project output be placed within the project folder. Once created, you'll need to copy it to either:

   `yourLogiApp/WEB-INF/classes` (`.class` file) - or -

   `yourLogiApp/WEB-INF/lib` (`.jar` file)

4. Write your plug-in code, starting with this prototype:

```
import org.w3c.dom.*;
import org.w3c.dom.Document;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.util.Hashtable;
import com.logixml.plugins.LogiPluginObjects10;

public class myPlugin
{
 public void yourMethodName(LogiPluginObjects10 rdObjects)
{
/* your method code goes here */
 }
}
```

Add the methods and code you need to get the plug-in to do whatever it's supposed to do.

Java Plug-in - Example: Change the Application Caption

The following are code examples for a plug-in that alters the caption of the Logi application at runtime:

```
import org.w3c.dom.*;
import org.w3c.dom.Document;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.util.Hashtable;
import com.logixml.plugins.LogiPluginObjects10; import java.io.*;
import java.io.File;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.OutputKeys;
import com.sun.net.httpserver.HttpContext;
public class myPlugin
{
 public void SetApplicationCaption(LogiPluginObjects10 rdObjects)
{
 try
{
 DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder);
byte b[] = rdObjects.getCurrentDefinition().getBytes();
java.io.ByteArrayInputStream input = new java.io.ByteArrayInputStream(b);
```

```
Document xmlSettings = docBuilder.parse(input);
NodeList nl = xmlSettings.getElementsByTagName("Application");
if (nl.getLength() > 0)
 {
 Node nodApp = nl.item(0);
Element eleApp = (Element)nodApp;
eleApp.setAttribute("Caption", "Greetings from the Sample Plugin!");
rdObjects.setCurrentDefinition(getOuterXml(xmlSettings));
 }
}
catch (Exception ex)
{
ex.printStackTrace();
System.out.println("SetApplicationCaption Error " + ex.getMessage());
}
 }
}
```

Java Plug-in - Example: Modify a SQL Query with a Request Variable

The following are code examples for a plug-in that customizes a SQL query based on a request variable:

```
import org.w3c.dom.*;
import org.w3c.dom.Document;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.util.Hashtable;
import com.logixml.plugins.LogiPluginObjects10;import java.io.*;
import java.io.File;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.OutputKeys;
import com.sun.net.httpserver.HttpContext;
public class myPlugin
{
 public void SetCustomerQuery(LogiPluginObjects10 rdObjects)
{
try
 {
DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
byte b[] = rdObjects.getCurrentDefinition().getBytes();
java.io.ByteArrayInputStream input = new java.io.ByteArrayInputStream(b);
```

```
Document xmlDefinition = docBuilder.parse(input);
NodeList nl = xmlDefinition.getElementsByTagName("DataLayer");
if (nl.getLength() == 0)
{
throw new Exception("The report is missing the DataLayer element.");
}//Use a Request variable to set set the SELECT query.
int iPos = rdObjects.getRequestParameterNames().indexOf((Object)"Continent");
if (iPos < 0)
 {
 throw new Exception("Continent Parameter name not found in SetCustomerQuery ");
 }String sContinent = (String)rdObjects.getRequestParameterValues().get(iPos);
String sSelect = new String();
if (sContinent == null)
{
sSelect = "SELECT * FROM Customers";
}
else
{
if (sContinent.equals("NA"))
 {
 sSelect = "SELECT * FROM Customers WHERE Country IN('USA','Mexico','Canada')";
 }
else
{
if (sContinent.equals("SA"))
 {
```

```
sSelect = "SELECT * FROM Customers WHERE Country IN ('Argentina','Brazil','Venezuela')";
 }
else
{
if (sContinent.equals("EU"))
 {
 sSelect = "SELECT * FROM Customers WHERE Country IN ('UK','Sweden','France','Spain','Switzerland','Austria',
 'Portugal','Ireland','Belgium','Germany','Finland','Poland','Denmark')";
 }
 else
 {
sSelect = "SELECT * FROM Customers";
}
}
}
}
Node nodApp = nl.item(0);
Element eleDataLayer = (Element)nodApp;
eleDataLayer.setAttribute("Source", sSelect);
rdObjects.setCurrentDefinition(getOuterXml(xmlDefinition));
}
catch (Exception ex)
{
ex.printStackTrace();
System.out.println("SetCustomerQuery Error " + ex.getMessage());
}
```

```
}
```

Java Plug-in - Example: Changing Data in a Datalayer

The following are code examples for a plug-in that converts RTF text in a datalayer to plain text. It creates an in-memory RichTextBox control, then iterates through all records in the datalayer, processing and replacing the contents of a data column.

```
[Visual Basic]
Imports System.Xml.Linq
Imports System.Windows.Forms.RichTextBox

Public Class Plugin

 Public Sub RTF2Text(ByRef rdObjects As rdPlugin.rdServerObjects)
Dim objRTB As New System.Windows.Forms.RichTextBox()
Dim objDoc As New XmlDocument()
Dim objNodeList As XmlNodeList
Dim objNode As XmlNode' load the XML from the datalayer
objDoc = rdObjects.CurrentData

' get a nodelist for all nodes (records) in XML doc
' in XPath syntax "dtTest" equals the name of the Data Table,
 ' and is the node name in the XML document
objNodeList = objDoc.SelectNodes("//dtTest")' loop thru each node, processing the "colText" attribute value,
 ' which is name of column with RTF text.
' put each value into RichTextBox as RTF, then reassign it as plain text
For Each objNode In objNodeList
 objRTB.Rtf = objNode.Attributes.ItemOf("colText").Value
objNode.Attributes.ItemOf("colText").Value = objRTB.Text
```

```
Next' clean up
objNode = Nothing
objNodeList = Nothing
objDoc = Nothing
objRTB = Nothing End Sub

End Class
[C#}
using System;
using System.Xml.Linq;
using rdPlugin;
using System.Windows.Forms;

namespace RTFPlugin
{
 public class Plugin
 {
public void RTF2Text(ref rdServerObjects rdObjects)
{
RichTextBox objRTB;
XmlDocument objDoc;
XmlNodeList objNodeList;
// create RichTextBox
objRTB = new RichTextBox();// load the XML from the datalayer
objDoc = new XmlDocument();
objDoc = rdObjects.CurrentData;// get a nodelist for all nodes (records) in XML doc
//
```

```
 in XPath syntax "dtTest" equals the name of the Data Table,
// and is the node name in the XML document
objNodeList = objDoc.SelectNodes("//dtTest");// loop thru each node, processing the "colText" attribute value,
// which is name of column with RTF text.
// put each value into RichTextBox as RTF, then reassign it as plain text
foreach (XmlNode objNode in objNodeList)
 {
objRTB.Rtf = objNode.Attributes.GetNamedItem("colText").Value;
objNode.Attributes.GetNamedItem("colText").Value = objRTB.Text;
}
}
 }
}
```

Java Plug-in - Implementing Your Plug-in

When your plug-in has been written and compiled, add the element(s) you need to call it in your Logi application. The elements used to do this are described in "Logi Plug-ins" on page 488.


The first time you run your Logi application, your web server may load the plug-in into memory and cache it there. If you make coding changes to the plug-in and want to rebuild it, you may need to **stop** and **restart** your web server first, in order to be able to replace the previous build.

Java Plug-in - Debugging Example: Eclipse and Tomcat

You can debug your plug-in as it runs in your Logi app. Due to the variety of Java tools and web servers available, it's not possible to provide comprehensive debugging guidance here. However, the following example, using the **Eclipse** IDE and the **Apache Tomcat** web server should be helpful.

Generally speaking, Java application servers support a Debug mode, which can be enabled when starting the application server. This creates a debugging server, which is typically available on port 8000. You connect your Eclipse project to this port to start debugging your application.

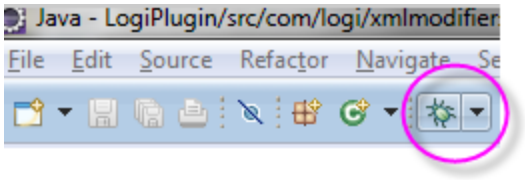
If you're developing and testing on your desktop platform, to enable Debug mode navigate to the `{catalina_home}/bin` folder and execute the following commands:

```
set JPDA_ADDRESS=8000
set JPDA_TRANSPORT=dt_socket
catalina.bat jpda start
```

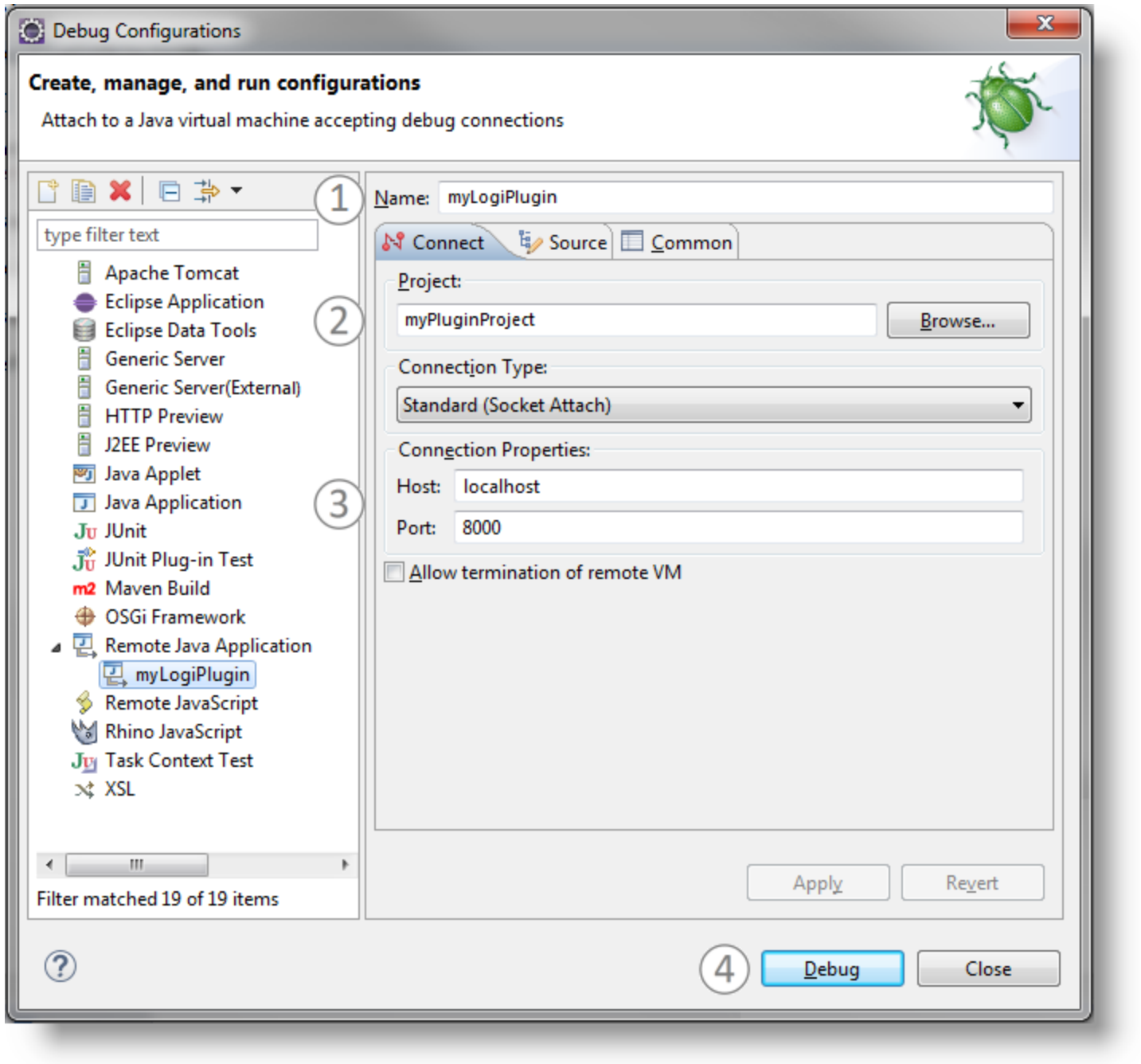
If Tomcat is on a remote machine, you may instead add this to your JAVA_OPTS:

```
-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

Tomcat should start, with its secondary debugging server running. If you have difficulty with this, consult your web server documentation.



Next, in Eclipse, open your plug-in project, and then click the little options arrow beside the Debug icon on the toolbar, as shown above. Select the **Debug Configurations...** option in the pop-up menu that appears.


Debug Configurations
Create, manage, and run configurations
Attach to a Java virtual machine accepting debug connections
type filter text
Apache Tomcat
Eclipse Application
Eclipse Data Tools
Generic Server
Generic Server(External)
HTTP Preview
J2EE Preview
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Maven Build
OSGi Framework
Remote Java Application
myLogiPlugin
Remote JavaScript
Rhino JavaScript
Task Context Test
XSL
Filter matched 19 of 19 items
1
2
3
4
Name: myLogiPlugin
Connect
Source
Common
Project:
myPluginProject
Browse...
Connection Type:
Standard (Socket Attach)
Connection Properties:
Host: localhost
Port: 8000
Allow termination of remote VM
Apply
Revert
Debug
Close

Double-click the **Remote Java Application** item in the left menu to create a new debug profile, as shown above. Then,

1. Provide an arbitrary profile **Name**.
2. Browse for and select your plug-in **Project**.
3. Ensure that the Connection Properties are correct. The **Host** is the base address of your Tomcat server; if it's running on your development machine, then it will be *localhost*. The **Port** is the one configured when starting Tomcat (default: *8000*).
4. Click **Debug** to begin your debugging session.

Eclipse will now connect to the debugging server loaded in Tomcat and is then ready for you to browse your Logi application. Interacting with your Logi app will eventually run your plug-in; proper debugging can begin in Eclipse at that point, with the use of watches, breakpoints, etc. These techniques are beyond the scope of this topic; see your Eclipse User Guide for more information.

To end your debugging session, click the Disconnect icon in the Debug view toolbar, as shown above.

If you haven't done so already, you should read "Logi Plug-ins" on page 488 for important supporting information before proceeding. If you're developing a **.NET** plug-in, see "Create .NET Plug-in" on page 499 instead of this one.

Java Plug-in - Plug-in Class Methods and Properties

Logi plug-ins for Java are written in Eclipse or a similar development tool, using the LogiPluginObjects10 class. The class files are:

yourLogiApp/WEB-INF/classes/com/logixml/plugins/LogiPluginObjects10.java - or -
yourLogiApp/WEB-INF/classes/LogiPluginObjects.java

You instantiate this class in your code and it provides access to the Logi application and its data. The class object is defined in your code as:

```
public class LogiPluginObjects10 (or LogiPluginObjects)
extends java.lang.Object
```

Once instantiated, the object provides the following properties and methods for use with your custom code:

- addDebugMessage
- get/setCurrentData
- get/setCurrentDataFile
- get/setCurrentDefinition
- get/setCurrentElement
- get/setCurrentHttpContext
- get/setPluginParameters
- replaceTokens
- setRequest
- getRequestParameterNames
- getRequestParameterValues

- get/setResponse
- get/setResponseHtml
- get/setReturnedDataFile
- get/setSession
- getSettingsDefinition

Here are the object's methods and parameters:

addDebugMessage

Inserts an entry into the Debugging Trace page.`void addDebugMessage(java.lang.String cur-rentEvent)`

`void addDebugMessage(java.lang.String currentEvent, java.lang.String programObject)`

`void addDebugMessage(java.lang.String currentEvent, java.lang.String programObject, java.lang.String objectValue)`

`void addDebugMessage(java.lang.String currentEvent, java.lang.String programObject, java.lang.String objectValue, java.lang.Object More Info)` **Parameters**:

`currentEvent` - text to the inserted in the Trace page's Event column

`programObject` - text to the inserted in the Trace page's Object column

`objectValue`- text to the inserted in the Trace page's Value column

`More Info` - object, such as XML data or XSL code, that is displayed via link inserted in the Trace page's Value column

The time value for the entry is automatically provided. If an error occurs within the plug-in, a fresh Trace page is generated, so any earlier messages placed with addDebugMessage are lost.

getCurrentData / setCurrentData

Gets/sets an XML Document object containing the data in all of the datalayers. `public org.w3c.-dom.Document getCurrentData()`

`public void setCurrentData(org.w3c.dom.Document curdata)`

getCurrentDataFile / setCurrentDataFile

Gets/sets a string object containing of the fully-qualified path and filename for the cache file containing the XML data retrieved by the datalayer. `public java.lang.String getCurrentDataFile()`

`public void setCurrentDataFile(java.lang.String currentDataFile)`

getCurrentDefinition / setCurrentDefinition

Gets/sets a string object containing the XML source code of the current report definition. `public java.lang.String getCurrentDefinition()`

`public void setCurrentDefinition(java.lang.String curdef)`

getCurrentElement / setCurrentElement

Gets/sets a string object containing the XML source code of the current report definition. `public org.w3c.dom.Element getCurrentElement()`

`public void setCurrentElement(org.w3c.dom.Element ele)`

getCurrentHttpContext / setCurrentHttpContext

Gets/sets an XmlDocument object containing all HTTP-specific information, including Application, Session, Server, Request, and Response objects `public javax.servlet.ServletContext getCurrentHttpContext()`

`public void setCurrentHttpContext(javax.servlet.ServletContext currHttpContext)`

getPluginParameters / setPluginParameters

Gets/sets a Hashtable object containing the parameters, if any, passed to the plug-in. `public java.util.Hashtable getPluginParameters()`

`public void setPluginParameters(java.util.Hashtable plugPar)`

replaceTokens

Returns a string object containing the `sTokens` parameter with some or all of its tokens replaced with their current values, and with other optional effects applied. `replaceTokens(java.lang.String sTokens)`

`replaceTokens(java.lang.String sTokens, boolean bDuplicateSingleQuotes)`

`replaceTokens(java.lang.String sTokens, boolean bDuplicateSingleQuotes, org.w3c.dom.Element eleDataLayerRow)`

`replaceTokens(java.lang.String sTokens, boolean bDuplicateSingleQuotes, org.w3c.dom.Element eleDataLayerRow, boolean bDuplicateBracketedQuotes)`

`replaceTokens(java.lang.String sTokens, boolean bDuplicateSingleQuotes, org.w3c.dom.Element eleDataLayerRow, boolean bDuplicateBracketedQuotes, boolean bAssumeNonQuotedAsNumeric)`

`replaceTokens(java.lang.String sTokens, boolean bDuplicateSingleQuotes, org.w3c.dom.Element eleDataLayerRow, boolean bDuplicateBracketedQuotes, boolean bAssumeNonQuotedAsNumeric,`

`java.lang.String[] tokenList)`

`replaceTokens(java.lang.String sTokens, boolean bDuplicateSingleQuotes, org.w3c.dom.Element eleDataLayerRow, boolean bDuplicateBracketedQuotes, boolean bAssumeNonQuotedAsNumeric, java.lang.String[] tokenList, boolean bSqlInjectionGuard)` **Parameters**:

`sTokens` - string containing Logi tokens, such as @Data.LastName~, to be replaced with current values.

`bDuplicateSingleQuotes` – if *True*, causes any single-quotes in `sTokens`to be doubled in return string.

`eleDataLayerRow` - XmlElement containing the data that will be used to replace the @Data tokens in `sTokens`.

`bDuplicateBracketedQuotes` - if *True*, double-quotes embedded in a token value will be included in return string.

`bAssumeNonQuotedAsNumeric` - if *True*, replacement processing will handle unquoted, numeric token values as numbers.

`TokenList` - comma-separated list of token types, indicating the types to be replaced; other types will be skipped.

`bSqlInjectionGuard` - if *True*, replacement processing checks all token values for potentially dangerous SQL strings.

Throws:

`javax.xml.parsers.TranformerConfigurationException`

`java.lang.Exception`

setRequest

Sets an HttpServletRequest object and passes it as an argument to the servlet's service methods. `public void setRequest(javax.servlet.http.HttpServletRequest req)`

getRequestParameterNames

Gets an array of the HTTP Request names sent by the client to a Logi application. `public java.util.Vector getRequestParameterNames()`

getRequestParameterValues

Gets an array of the HTTP Request values sent by the client to a Logi application. `public java.util.Vector getRequestParameterValues()`

getResponse / setResponse

Gets/sets an HttpServletResponse object containing the rendered HTML response page; available when the *FinishHtml* event completes. `public javax.servlet.http.HttpServletResponse getResponse()`
`public void setResponse(javax.servlet.http.HttpServletResponse res)`

getResponseHtml / setResponseHtml

Gets/sets a String object containing the rendered HTML response page, available when the *FinishHtml* event completes. `public java.lang.String getResponseHtml()`
`public void setResponseHtml(java.lang.String resp)`

getReturnedDataFile / setReturnedDataFile

Gets/sets a String object containing the fully-qualified path and filename for the cache file which contains the manipulated XML data after processing. `public java.lang.String getReturnedDataFile()`

```
public void setReturnedDataFile(java.lang.String returnedDataFile)
```

getSession / setSession

Gets/sets an HttpSession object, which is used to access Session values for the current Logi application session. `public javax.servlet.http.HttpSession getSession()`
`public void setSession(javax.servlet.http.HttpSession ses)`

💡 Most session variables are replicated back and forth between the Logi Engine and the web server with every plug-in call.

getSettingsDefinition

Gets an XML Document object representation of the _Settings definition. `public org.w3c.dom.Document getSettingsDefinition()`

Throws:

`javax.xml.parsers.ParserConfigurationException`
`org.xml.sax.SAXException`
`java.lang.Exception`

Create an RSS Feed

If you report on data that's constantly changing, an **RSS feed** might be the perfect solution for keeping your report consumers informed.

The following topics provide guidance for creating an RSS ("Really Simple Syndication") feed within Logi Info:

- Preparing Your Information
- RSS Feed File Format
- Creating Your Feed File

About RSS Feeds

An RSS feed is an XML text file on your web server that contains a list of headlines or topics, each with a description and link to the complete document or report. To see this in action, take a look a the following RSS feed:

Yahoo! News Top Stories

Users can subscribe to a feed and their browser will periodically check for new content and, optionally, alert them when it's found. Because the "feed file" on your web server is just an XML text file, it's easy to regenerate it periodically without any human intervention.

This can be very useful, for example, if you want to monitor a discussion forum or blog and be alerted when new information is posted. Many Logi Analytics product users monitor our Discussion Forum in this way. Another possible use of feeds is to let users subscribe to reports that are published multiple times per day. The image at the left identifies elements common to all feeds: (1) a *title* or subject, which is also a link to the complete content; (2) a *timestamp* indicating when the content was published; and (3) a brief *description* of the content. The photo is optional. Though RSS is widely used, another feed variety, called **Atom**, also exists. This topic only deals with RSS feeds. Popular browsers such as Internet Explorer, Firefox, and Chrome have a full set of

features for working with feeds. This image has been widely accepted as the standard icon for feeds and appears in the browsers themselves to identify feeds and their features.

What do you need to do to establish your own feed?

1. You need to identify the information that you want to publish by using links in a feed
2. You need to place those links in an **XML file** on your web server, using the correct RSS format and XML elements
3. You need to schedule the periodic generation of your feed file

Logi Info makes this fairly easy to do and each step is discussed below. There are other aspects of RSS feeds, including the use of 3rd-party "feed aggregator" web sites and feed validation that are not typically relevant to reporting with Logi products and are therefore not covered here. You can find out more about these and other RSS topics here.

Preparing Your Information

The first step in the process is to create a new, or identify an existing, Report definition that will produce the information that you want to have appear on your feed page. This could be in an existing Logi application or in a separate application created just for this purpose. At a minimum, the application will need one Report definition and one Process definition with a task.

If you're creating a new Report definition just for this purpose, don't worry about its appearance; no one is going to see it. Don't bother with style sheets, other formatting, or report headers or footers, either.

Your report definition can contain anything from a static set of **Label** elements to a **Data Table** with data from a database. You want to be sure this report will contain the minimum information you'll need to create your feed page: a *title*, a *URL* to the content it describes, a *timestamp*, and a brief *description***.** for each item in the feed list.

Feed pages are usually *time-oriented*; users get them because they want to know the latest information. In addition, the best feed pages provide a sane number of entries - the Top Ten (most recent) stories, for example - rather than a great long list of links. If you're getting the data from a database, you might want to retrieve the data sorted by timestamp and limited to a small number of records.

Once you've built your report, preview it and ensure that all of the desired data is there before proceeding.

RSS Feed File Format

There are several versions of RSS in use; the version discussed here conforms to one of the latest, **RSS 2.0**. The syntax rules of RSS 2.0 are very simple but very strict. There are basically two sections, one that describes the feed itself and one that contains the descriptions of the articles, reports, or items in the feed.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version="2.0">
<channel>
<title>Logi Analytics Support Discussion Forum</title>
<link>https://devnet.logianalytics.com/rdPage.aspx?rdReport=DiscussionForum</link>
<description>Logi Analytics Support Discussion Forum</description>
<item>
<title>Window SizeThreadID=71102&https://devnet.logianalytics.com/rdPage.aspx?rdReport=ForumThreadPosts>link<

>/title<</link>
<description>How can I determine the Window size?></description>
<author>John Doe</author>
<pubDate>Thu, 12 Apr 2007 3:12:01 EDT</pubDate>
</item>
</channel>
</rss version>
```

In the example above, first line in the document is the XML declaration which defines the XML version and the character encoding used in the document.

The next line is the RSS declaration which identifies that this is an RSS document (in this case, RSS version 2.0).

Elements are *case-sensitive*.

The next line contains the required `<channel>` element, which is used to describe the RSS feed. It has three required child elements:

- `<title>` - Defines the title of the channel (e.g. Logi Analytics Support Discussion Forum)
- `<link>` - Defines the hyperlink to the channel (e.g. https://devnet.logianalytics.com/rdPage.aspx?rdReport=Forum)
- `<description>` - Describes the channel (e.g. Logi Analytics Support Discussion Forum)

Each `<channel>` element can have one or more `<item>` elements, each of which defines an article or report in the RSS feed. The `<item>` element has three required child elements:

- `<title>` - Defines the title of the item (e.g. "Window Size")
- `<link>` - Defines the hyperlink to the item (e.g. https://devnet.logianalytics.com/rdPage.aspx?rdReport=Posts&ID=71102)

- `<description>` - Describes the item (e.g. "How can I determine the Window size?")

Not required, but widely used, are two additional child elements for each `<item>` element: `<author>` and `<pubdate>`. Finally, the last two lines close the `<channel>` and `<rss>`elements. The syntax for writing comments in RSS is similar to that of HTML: `<!-- This is my RSS comment -->` There are other RSS elements that we won't use, including images; you can find out more here about RSS syntax.

Creating Your Feed File

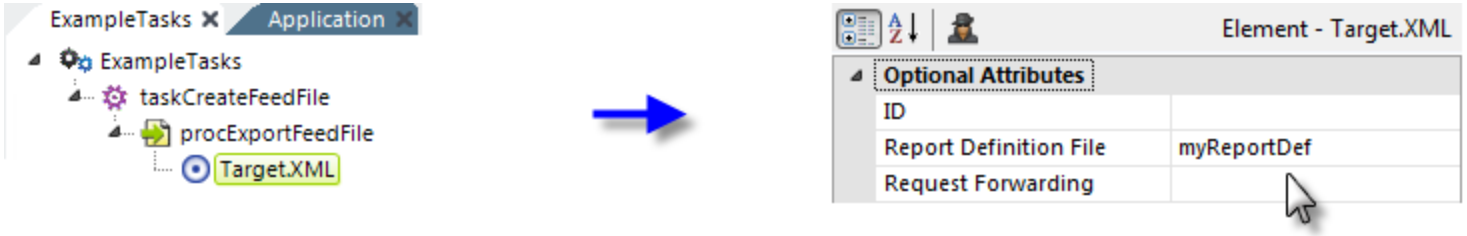
Now that you know what goes into your feed file, you need to provide a link to it from your application or web site. We recommend that you use the feed icon. The icon's hyperlink is going to be similar to:

`http://myWebServer/myRss/myFeedFile.xml` where "myRss" is a folder containing your feed file.

Free, standardized feed icons are available in a variety of sizes and graphic types at this web site.

To generate your feed file:

1. Create a Process definition (or use an existing Process definition).
2. Add a new **Task** element to it ("taskCreateFeedFile" in the example above).
3. To the task, add a **Procedure.Export XML** element. Set its **Filename** attribute to the full file system path and filename for your feed file. This is an excellent place to make use of the `@Function.AppPhysicalPath~` token. Be sure that your filename ends with ".xml".



4. Add a **Target.XML** element, as shown above, beneath the Procedure.Export XML element and set its **Report Definition File** attribute to the report definition that will provide your feed data.

One way to put the data from your report into the correct format and syntax for the feed file is to apply an **XSL Transform** to the raw XML that comes from the Export XML procedure. To do this you'll need to create a transform (.xsl) file and place it in the _SupportFiles folder in your application project folder.

What the Transform Does

A full explanation of **XPath** and **XSL Transforms** is beyond the scope of this topic. However, the following should give you a preliminary understanding and help you create your transform file. XPath allows you to navigate through the raw XML data and the transform outputs it in the proper format. As an example of how a transform works, follow what happens when a hypotheticalLogi Analytics DevNet Discussion Forum feed is created. Here's one record of the data returned by the SQL query in our feed report definition:

TID	Subject	Message	CreatedAt	UserName	ForumTitle
711020	Setting Window Size	How can I get the size of a browser window at runtime?	Thur 12 Nov 2015 15:12:01 EST	John Doe	Logi Info

Six columns of data are returned and, in the report definition (but not shown here), other elements are used to add three extra columns to the datalayer: "msgDate" containing a formatted version of the CreateAt date, "rssDate" containing a formatted version of today's date, and "URL" combining the TID column value with this URL:

```
https://devnet.logianalytics.com/rdPage.aspx?rdReport=ForumThreadPosts&
```

When the **taskCreateFeedFile** task created earlier executes, it runs the report, gets the data discussed earlier, and outputs it using the Export XML procedure to a file.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rdData>
 <MyDataTable Subject="Setting Window Size" Message="How can I get the size of a browser windows at runtime?"
 CreatedAt="2015-11-12T15:12:01.0370000-05:00" UserName="John Doe" ForumTitle="Logi Info" msgDate="Thursday,
November12, 2015 3:12:01 PM" rssDate="Mon, 16Nov 2015 14:05:00"
 URL="https://devnet.logianalytics.com/rdPage.aspx?rdReport=ForumThreadPosts&ThreadID=71102"/>
</rdData>
```

In its raw form, before we apply the transform, the XML data for our one record would look like the example shown above.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <!-- Results will be XML -->
 <xsl:output method="xml" />
 <xsl:template match="rdData">
<xsl:text></xsl:text>
 <rss version="2.0">
<xsl:text></xsl:text>
<channel>
 <xsl:text></xsl:text>
 <title>Logi Analytics DevNet Discussion Forum</title>
 <xsl:text></xsl:text>
```

```
 <link>https://devnet.logianalytics.com/rdPage.aspx?rdReport=DiscussionForum</link> <xsl:text></xsl:text>
 <description>Logi Analytics DevNet Discussion Forum</description>
 <xsl:text></xsl:text>
 <language>en-us</language>
 <xsl:text></xsl:text>
 <webMaster>DevNet@LogiAnalytics.com</webMaster>
 <xsl:text></xsl:text>
 <pubDate>
 <xsl:value-of select="@rssDate" />
 </pubDate>
 <xsl:for-each select="MyDataTable">
 <xsl:text></xsl:text>
 <item>
<xsl:text></xsl:text>
 <title>
 <xsl:value-of select="@Subject" />
 </title>
 <xsl:text></xsl:text>
 <link>
 <xsl:value-of select="@URL" />
 </link>
 <xsl:text></xsl:text>
<description>
 <xsl:value-of select="@Message" />
</description>
```

```
 <xsl:text></xsl:text>
<author>
 <xsl:value-of select="@UserName" />
</author>
 <xsl:text></xsl:text>
<pubDate>
 <xsl:value-of select="@msgDate" />
 </pubDate>
 <xsl:text></xsl:text>
 </item>
 </xsl:for-each><xsl:text></xsl:text>
 </channel>
 <xsl:text></xsl:text>
 </rss>
 </xsl:template>
</xsl:stylesheet>
```

The **XLS Transform** used to iterate through and format the raw XML data into a feed file is shown above. Notice that it has a section at the top (starting with `<channel>` that describes the feed, and another (starting with `<xsl: for-each>`) the describes the repeating items (data rows), just as the RSS 2.0 standard requires.

Notice that, in the transform file, the "select" attribute for the`<xsl:for-each>`element is "MyDataTable". This is the name of the Data Table in the feed report definition.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version="2.0">
```

```
 <channel>
<title>Logi Analytics Support Discussion Forum</title>
<link>https://devnet.logianalytics.com/rdPage.aspx?rdReport=DiscussionForum</link>
<description>Logi Analytics Support Discussion Forum</description>
<language>en-us</language>
<webMaster>DevNet@LogiAnalytics.com</webMaster>
<pubDate>Mon, 16Nov 2015 14:05:00</pubDate>
<item>
<title>Window Size</title>
<link>https://devnet.logianalytics.com/rdPage.aspx?rdReport=ForumThreadPosts&ThreadID=71102</link>
 <description>How can I get the size of a browser windows at runtime?</description>
<author>John Doe</author>
 <pubDate>Thursday, November12, 2015 3:12:01 PM</pubDate>
</item>
</channel>
</rss>
```

And, finally, the example above shows the data in the feed file, after the transform has been applied. These links will lead you to more information about XPath and XSL Transforms.

To apply your transform:

1. Create your transform file and, in Studio, add it to the _SupportFiles folder in your project folder.
2. In your process definition, add an **XSL Transform** element beneath your Target.XML element, as shown above, and set its **XSL Transform File** attribute to the name of your transform file.

Test your task to ensure that it's writing your feed file in the correct *format* and to the correct *location*.

Finally, put your feed icon, with its link to your feed file, in place and you're all set to allow users to get your feed. Work with your browser to see what happens when you access a feed and how your browser deals with subscriptions to it.

Scheduling Generation of Your Feed File

To update your feed file on a regular basis, you can schedule your taskCreateFeedFile task to run periodically using **Logi Scheduler**. It will overwrite the feed file each time it runs. For complete information about scheduling tasks, see *Logi Scheduler*.

jQuery

jQuery is a popular, cross-browser JavaScript library designed to simplify client-side scripting. It's widely used and can be integrated into Logi applications, allowing you to take advantage of a variety of components, effects, and features.

The following topics discuss how to work with jQuery and the JSON data format within Logi definitions:

- Including and Using jQuery Components
- Converting XML into JSON Data
- Converting JSON Data into XML

About jQuery and JSON

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animation, and Ajax interactions for rapid web development and is the most popular JavaScript library in use today. jQuery is free, open source software and provides capabilities for developers to create plug-ins on top of the JavaScript library. You can download the library and documentation, and learn more about it at the official jQuery website.

Logi Info features elements that make it easy to include the jQuery libraries, either from local copies or remotely via a URL, in your report definitions.

Google provides free online access to a number of development API libraries, including jQuery and jQuery UI. This allows you to work with jQuery without having to maintain a local copy of the libraries and the examples in this topic use this approach. More information is available at the Google Libraries API web page.

JavaScript Object Notation, or **JSON**, is a lightweight, text-based, open standard for data interchange and works with jQuery. It's derived from the JavaScript language for use representing simple data structures and associative arrays. The JSON format is used

primarily to transmit data between a server and web application, serving as an alternative to XML. More information is available at the officialJSON website.

Logi Info features elements that make it easy to retrieve data from JSON data files into standard Logi datalayers and to convert datalayer XML data to the JSON format.

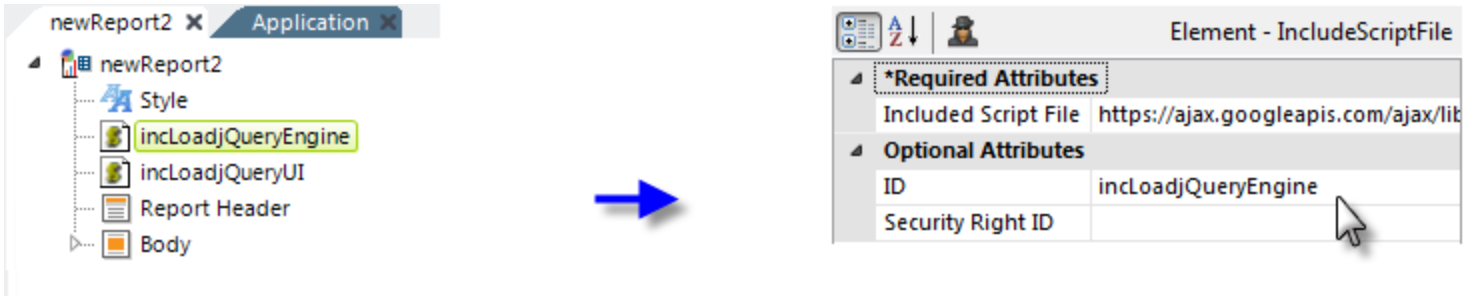
Including and Using jQuery Components

Here are two examples that illustrate how to include the jQuery libraries and jQuery code. The first one displays the "date picker" UI component.

First, we configure our report definition's **Style** element to use a jQuery style sheet hosted online by Google:

```
http://ajax.googleapis.com/ajax/libs/jqueryui/1.11.4/themes/smoothness/jquery-ui.css
```

The jQuery versions used here are representative and other versions will work as well.

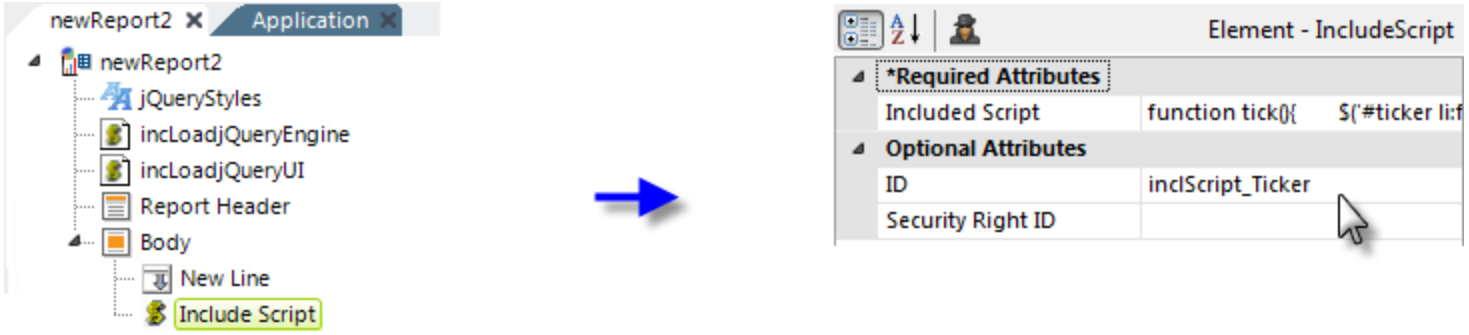


Then we add two **Include Script File** elements to our definition, as shown above, and configure their attributes to include the jQuery libraries hosted online:

```
http://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js
http://ajax.googleapis.com/ajax/libs/jqueryui/1.11.4/jquery-ui.min.js
```

The Google API hosting page suggests using these URLs with the "https:" protocol. However, if your Logi application is not specifically configured for SSL, then using "https:" URLs to include these libraries will *not* work. Use "http:" instead, as shown.

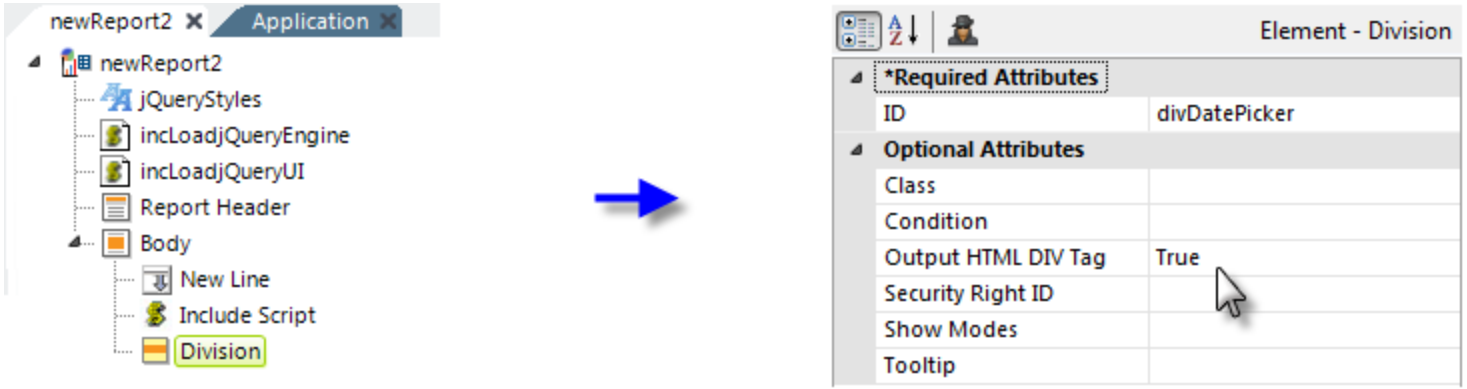
You could, of course, use local copies of the jQuery style sheet libraries, downloaded into _SupportFiles, by just selecting their file-names instead.



Next we add an **Include Script** element and enter our jQuery code in its **Included Script** attribute, as shown above. The full code is:

```
$(document).ready(function() {
$("#divDatePicker").datepicker();
});
```

And finally, we need to add a container (a **Division** element) for the date picker,



as shown above. The **Output HTML Div Tag** attribute must be set to *True* and notice that the **ID** of the division is used in the jQuery code, which is case-sensitive.

When you run the report, you should see a fully-operational date picker calendar similar to the one shown above.

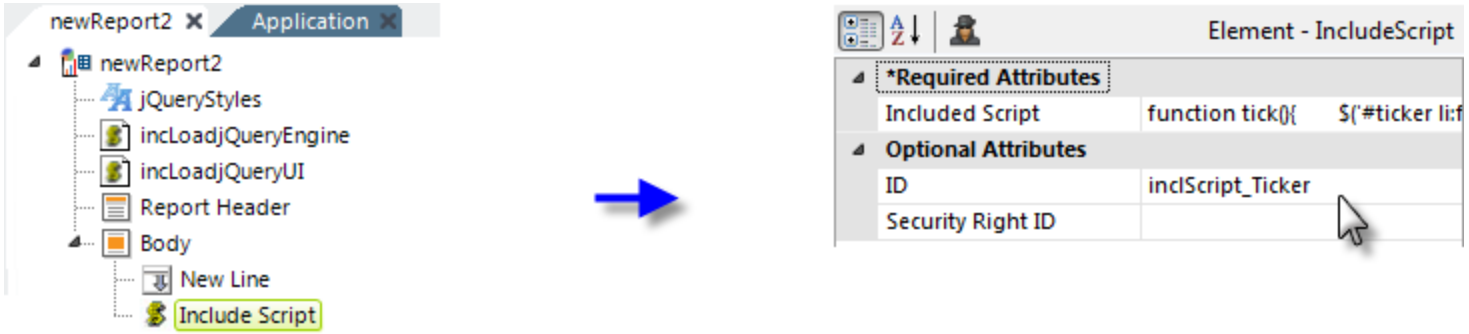
The date picker highlights the current date (default), as well as the selected date.

The jQuery News Ticker

The next example demonstrates a little more integration with standard Logi elements. It uses a datalayer as the source for "news headlines", one of which is displayed for a fixed interval before being replaced by the next headline, a classic "news ticker". Begin by adding the following two style classes to a local style sheet:

```
#ticker {
 height: 20px;
 width: 500px;
 overflow: hidden;
 border: 2px Green Solid;
 padding: 5px;
}

#ticker li {
 height: 30px;
 font-family: Verdana, Arial, sans-serif;
 font-size: 12pt;
 color: orange;
}
```

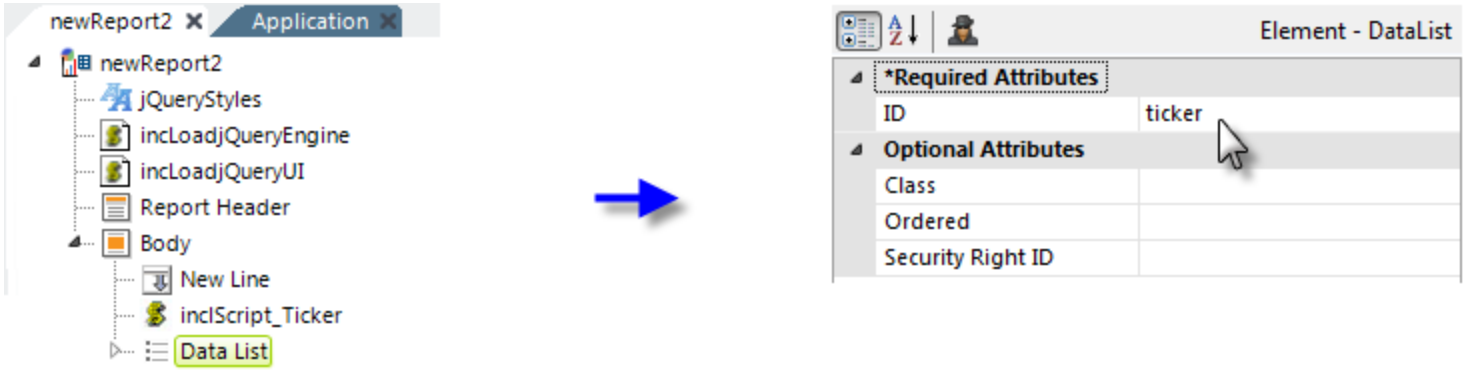
Then assign that style sheet to your report definition, using a **Style** element.



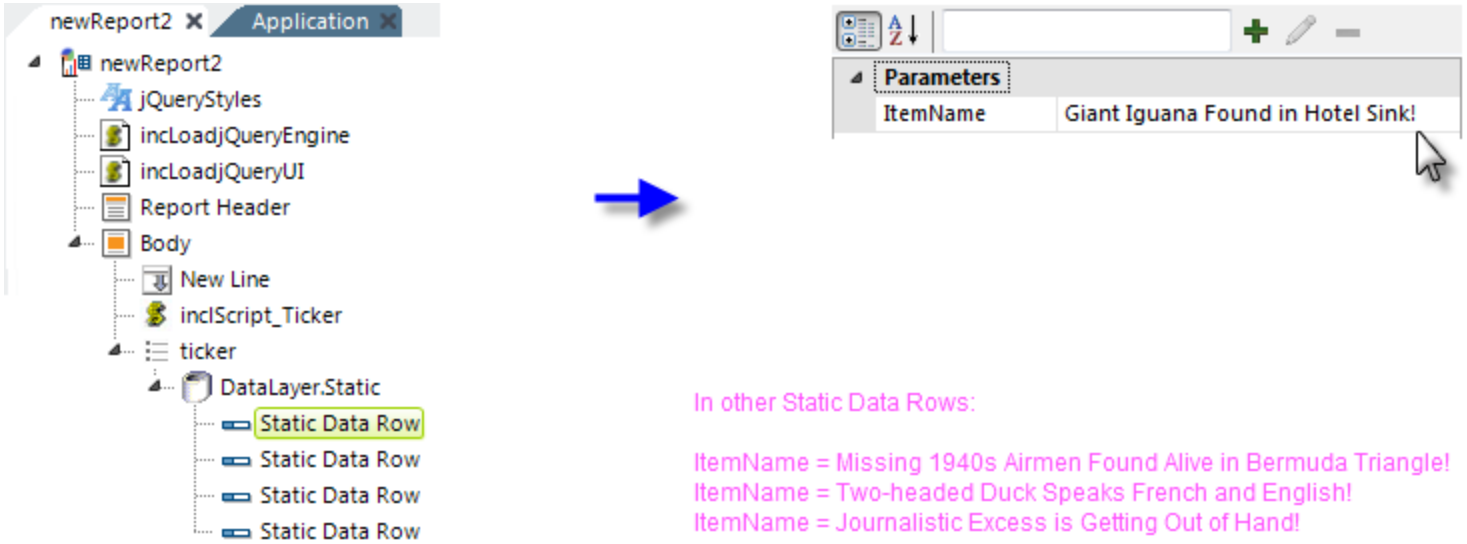
Use the same two jQuery libraries from the previous example. Add an **Include Script** element, as shown above and use the following jQuery code for its **Included Script** attribute value:

```
function tick(){
 $('#ticker li:first').animate({'opacity':0}, 200, function () { $(this).appendTo($('#ticker')).css('opacity',
1); });
}
setInterval(function(){ tick () }, 4000)
```

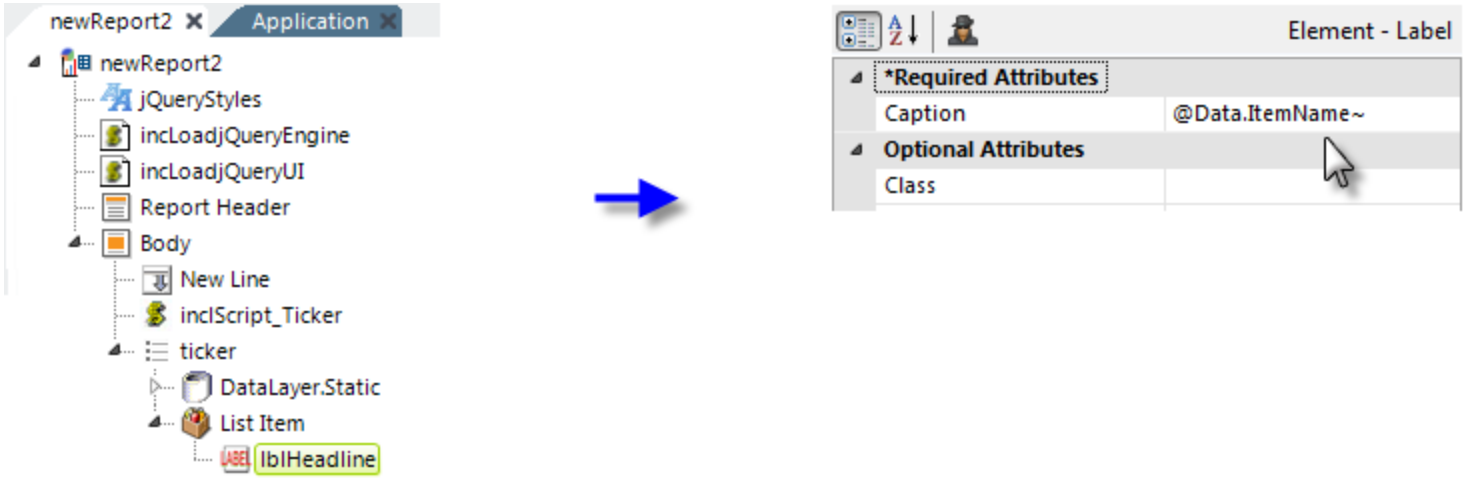
There is only one long line of code in the function tick( ) but it may have wrapped to look like two lines in your browser.



Now, add a **Data List** element in the report Body, as shown above. Its ID, "ticker", matches identifiers in the jQuery code and CSS, so you need to enter it exactly as shown.



Next, add a **DataLayer.Static** element and four **Static Data Row** elements. This, of course, could be any type of datalayer. Set the four data row elements to each have a single column called "ItemName" and enter one of the four headlines for each element, as shown above.



Finally, add a **List Item** element and, beneath it, a **Label** element. Set the Label element's **Caption** attribute as shown above.

So - what you've created is a datalayer-driven, HTML un-ordered list and the jQuery code will display one line of the list at a time.

Missing 1940s Airmen Found Alive in Bermuda Triangle!

Run the application and you should see something like the example above, with each headline rotating through it.

Now you can begin to see how the jQuery code can interact with other HTML components created by Logi elements, with CSS, and with data from a datalayer. The next sections look at data manipulation in more detail.

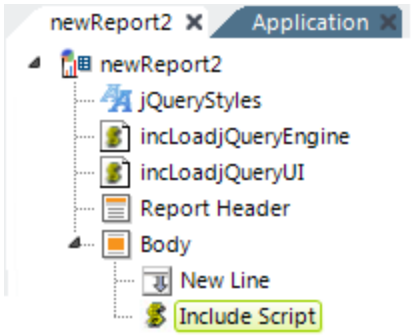
Converting XML into JSON Data

Many jQuery objects and plug-ins work with data in the JSON format. An easy way to get data into this format in a Logi application is to use the **JSON Data** element. This element runs a child datalayer and inserts the data it retrieves into a JavaScript array, which enables browser-side components, such as JQuery components, to use and display the data.

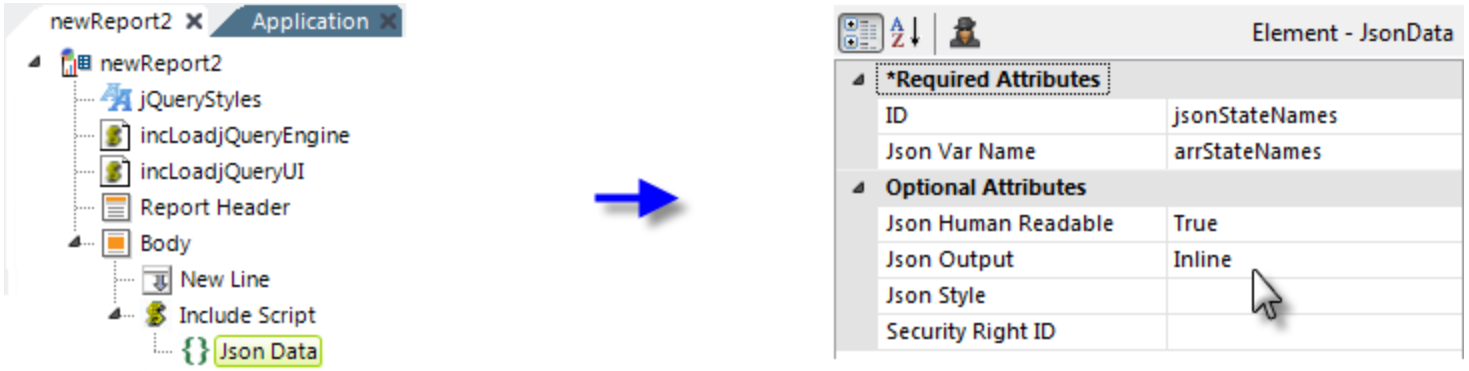
There are two options for including the JavaScript array data: *Inline* and *File*. If the Inline option is selected, the data is embedded directly into the generated report page HTML, between <SCRIPT> tags. If the File option is chosen, the data is written out to a temporary file in your app's rdDownload folder and included using a <SCRIPT SRC= filename.js> tag in the HTML. In both cases, your jQuery code uses the array variable name to access the data.

In many cases, the array object will match the data format expected by your jQuery code but, in some it may not, in which case you may need to write a little JavaScript code to extract the data you need.

Here's an example, using the jQuery AutoComplete widget.

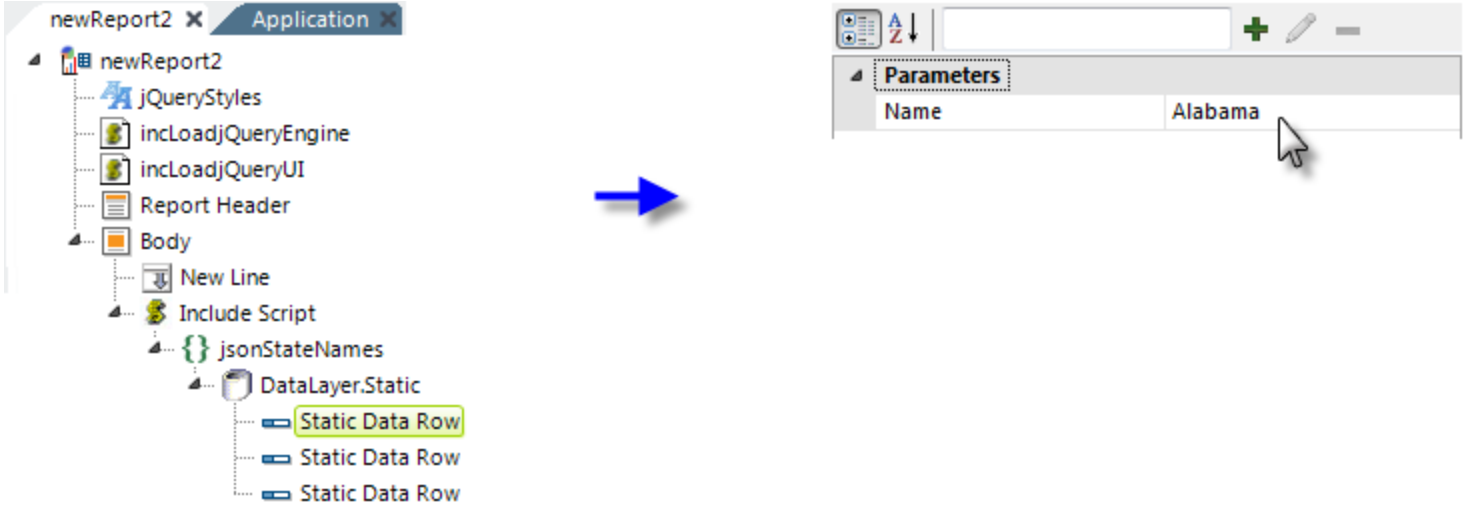


Start by using the same **Style** and **Include Script File** elements, and their settings, from the first example in this topic. Then add an **Include Script** element beneath your report definition's Body element, as shown above. We'll come back to it and write its script later.

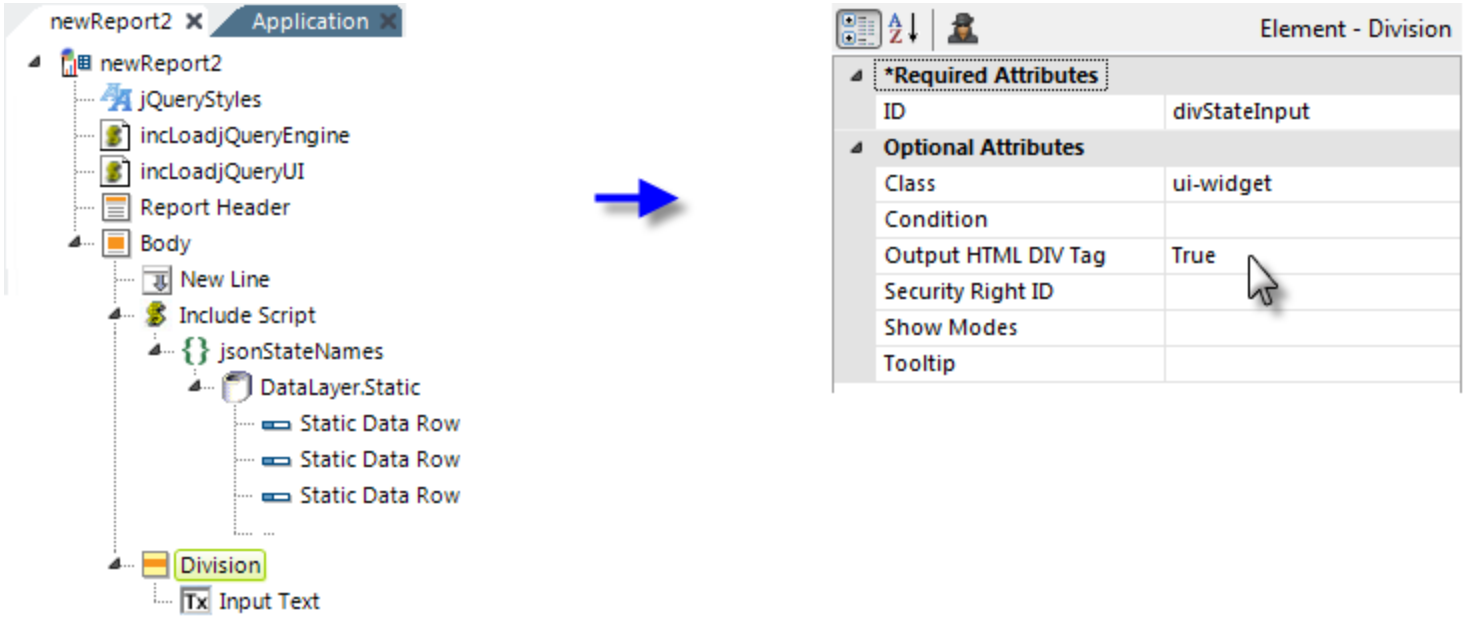


Beneath the Include Script element, next add a **JSON Data** element and set its attributes as shown above. Its unique attributes are:

- **Json Var Name** - Specifies the variable name to be used for the data array, which is used in the jQuery code.
- **Json Human Readable** - Specifies whether spaces and LFs are used to format the output JSON data, for easier debugging.
- **Json Output** - Specifies whether the data array will be embedded directly in the HTML (Inline) or included as a external file (File). Using an external file might help with performance for larger data volumes or might help by shielding the data from examination using View Source.
- **Json Style** - Specifies the format of the data output; either as *RowsToObjects* or as *ColumnsToPropertyArrays*. See the discussion below with output examples. Default: *RowsToObjects*



Next, add a datalayer beneath the JSON Data element. This could be any type of datalayer, but for the purposes of this example, we'll use **DataLayer.Static** which makes it easy to see the data. Add seven **Static Data Row** elements beneath it, adding values to a column named "Name". The values should be "Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", and "Connecticut".



Finally, add a **Division** element and a child **Input Text** element beneath the Body, as shown above. Set the Div's attributes as shown (the *ui-widget* class assigned to the Div is from the Google-hosted jQuery style sheet). Set the Input Text element's **ID** to *inpState*.

```
var arrStateNames = [
{"Name" : "Alabama"},
{"Name" : "Alaska"},
{"Name" : "Arizona"},
{"Name" : "Arkansas"},
{"Name" : "California"},
```

```
var arrStateNames = {
 "Name" : ["Alabama","Alaska","Arizona",
 "Arkansas","California","Colorado",
 "Connecticut"]
};
```

```
{"Name" : "Colorado"},
{"Name" : "Connecticut"}
];
```

Json Style =*RowsToObjects*

Json Style =*ColumnsToPropertyArrays*

If you **Preview** the application right now, then right-click and select **View Source** to see the underlying HTML, you should see the data embedded in the HTML, as shown above. This is the JavaScript array of data in JSON format, shown using the two different formatting styles available.

```
var data= [
{"Year": "2010", "TotalSales": 12540},
{"Year": "2011", "TotalSales": 21450}
];
```

Json Style =*RowsToObjects*

```
var data = {
 "Year": ["2010","2011"],
"TotalSales": [12540,21450]
};
```

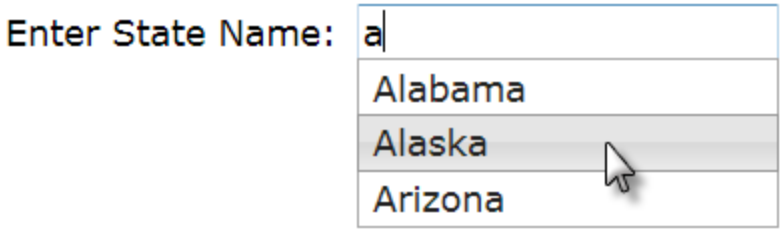
Json Style =*ColumnsToPropertyArrays*

Here's another example, shown above, which is unrelated to our exercise but which includes multiple columns, to further illustrate the effects of the **Json Style** attribute. *RowsToObjects* serializes datalayer rows into an array of objects, where each object represents a data row and the objects have a property for each column. *ColumnsToPropertyArrays* serializes the datalayer into a single object, where each column is a property with all the row values contained in an array. Now go back to the exercise defin-

JavaScript code: select the **Include Script** element and enter the following as its Included Script attribute value:

```
var len = arrStateNames.length;
var arrTemp = new Array(len);
for(var i = 0; i < len; i++) {
 arrTemp[i] = arrStateNames[i].Name;
}

$(document).ready(function() {
 $( '#inpState').autocomplete({
source: arrTemp
 });
 });
```

You'll recognize the bottom section as jQuery code; note the use of the Input Text element's ID ("inpState") and a temporary array variable name ("arrTemp") in it. The upper section creates the temporary array from the JSON data, because the widget is expecting a one-dimensional array of strings, while the JSON data is, as we've seen, two-dimensional.



If you Preview the application now, you should see the Input Text control. If you type the letter "a" into it, a list of the choices that begin with that letter will appear. Typing more characters will narrow the list further. You may care to go back and change the JSON Data element's Json Output attribute to **File**, Preview again, and then examine the temporary file created in rdDownload. It will have a GUID filename with a .js extension.

Managing JSON Data Columns

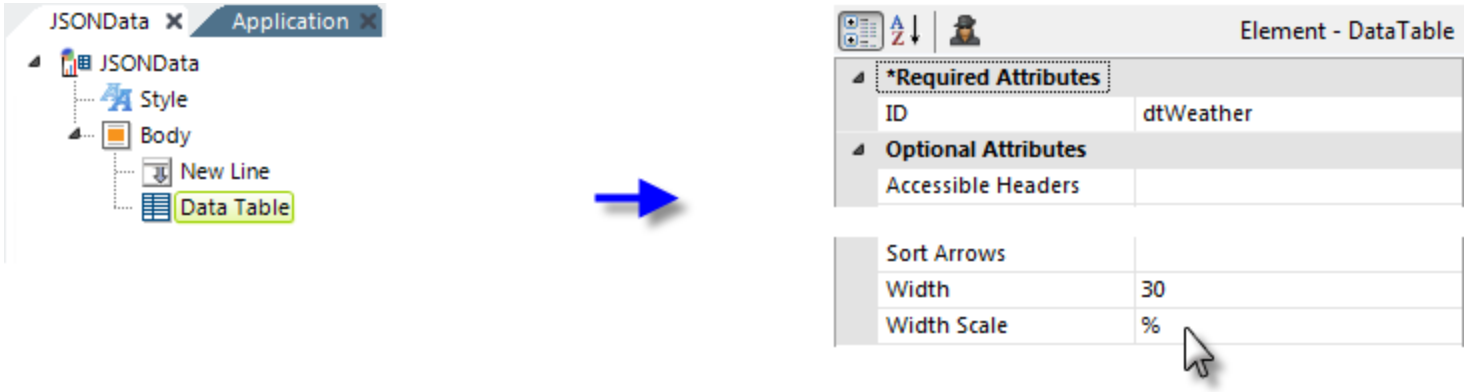
Normally, the Json Data element will output *all* of the columns in the datalayer, using the same column names. However, the **Json Column** element lets you manage the output columns. You include a Json Column element for each datalayer column that you want to output; which means you can output *fewer* than all of the datalayer columns if you desire. The element's attributes let you specify a different name for the column and its data type in the output as well.

Converting JSON Data into XML

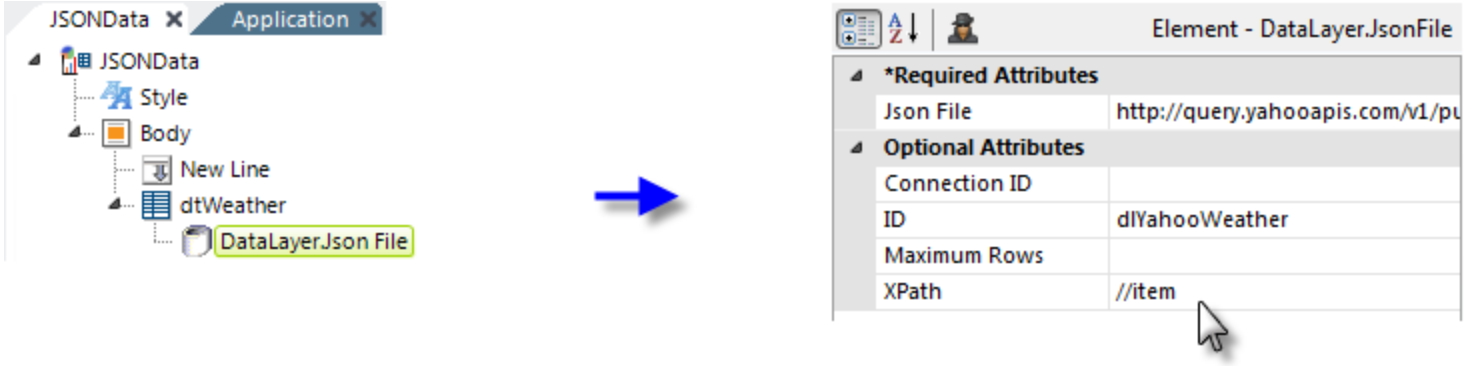
"Converting XML into JSON Data" on page 576 demonstrated how XML data retrieved into a datalayer could be converted into JSON data for use with jQuery components that work with that data. This topic describes the reverse: how to retrieve JSON data and turn it into XML data for use with Logi elements.

This example gets a weather forecast from Yahoo in JSON format, using a special datalayer, **DataLayer.JSON File**, and converts it to regular Logi XML data.

Start this example by setting the **Style** element to one of the standard Logi Analytics Themes, but don't use a style sheet. You won't need to include any script files, either.



Add a **Data Table** element and set its attributes as shown above.



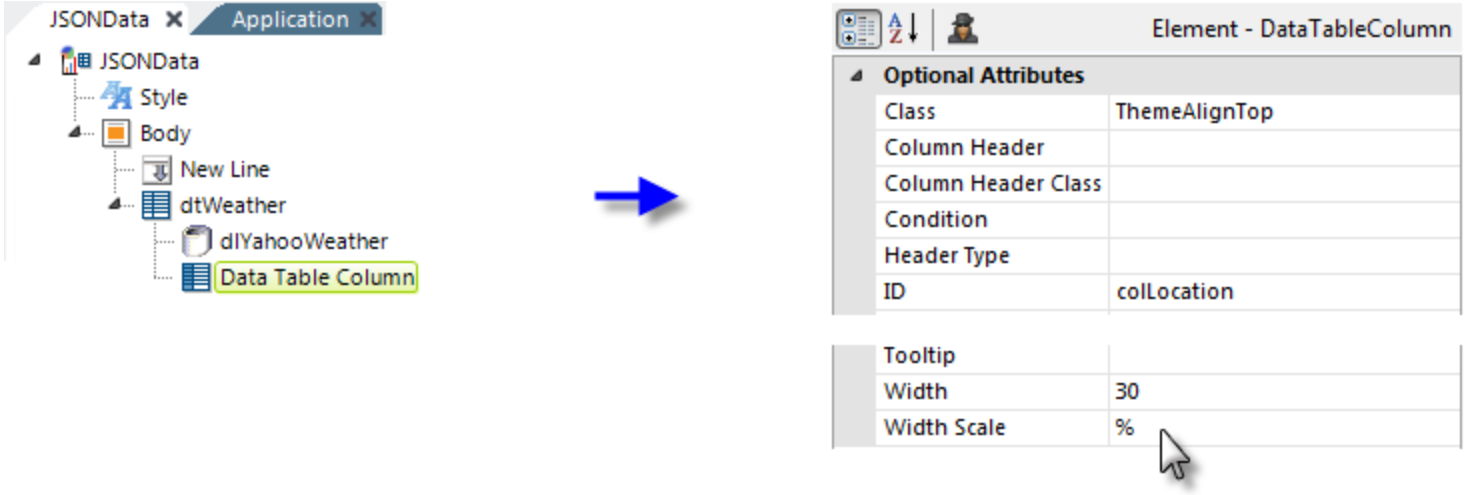
Next, add a **DataLayer.JSON File** element beneath the Data Table, as shown above. Here's the complete value to use for its **Json File** attribute value:

```
http://query.yahooapis.com/v1/public/yql?q=select%20item%20from%20weather.forecast%20where%20location%3D%2222102%22&format=json
```

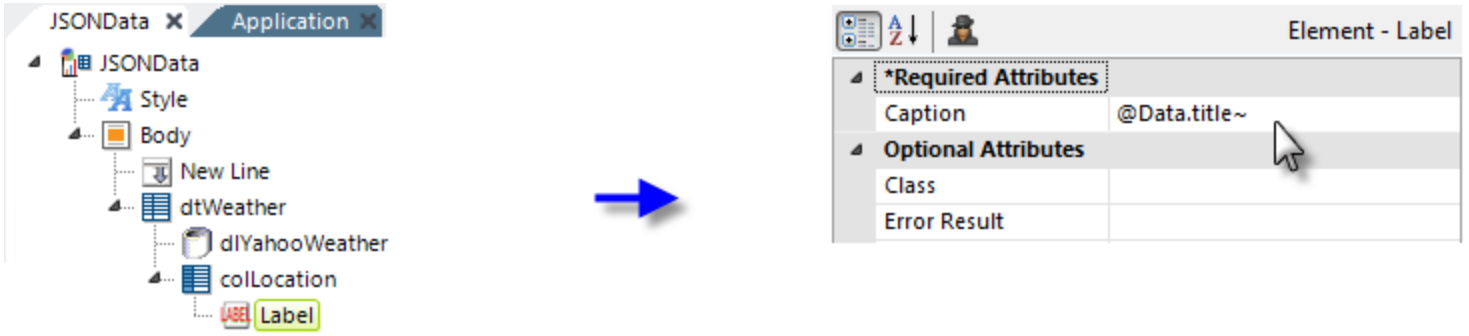
This may wrap in your browser, but it's all one line. The highlighted bit is a U. S. postal Zip Code, which determines the forecast location.

The Json File attribute value can be either a fully-qualified file path and name or a URL. By default, the server will look in the app's _SupportFiles folder, in which case you can simply specify the filename (such as mydata.js).

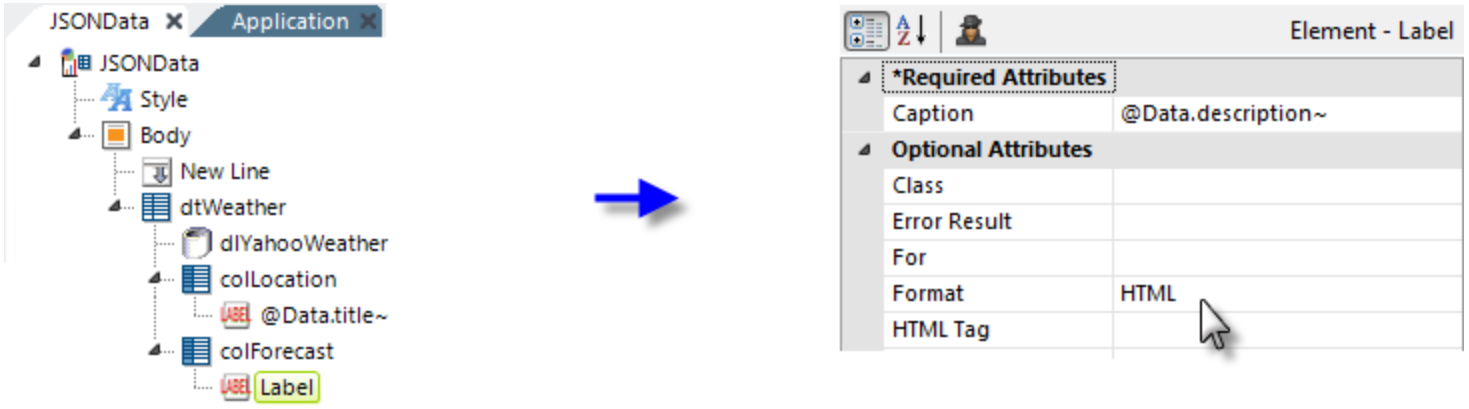
We're also using XPATH to extract the data we want from the returned data. You may want to turn on debug and look at the data in the various stages of processing.



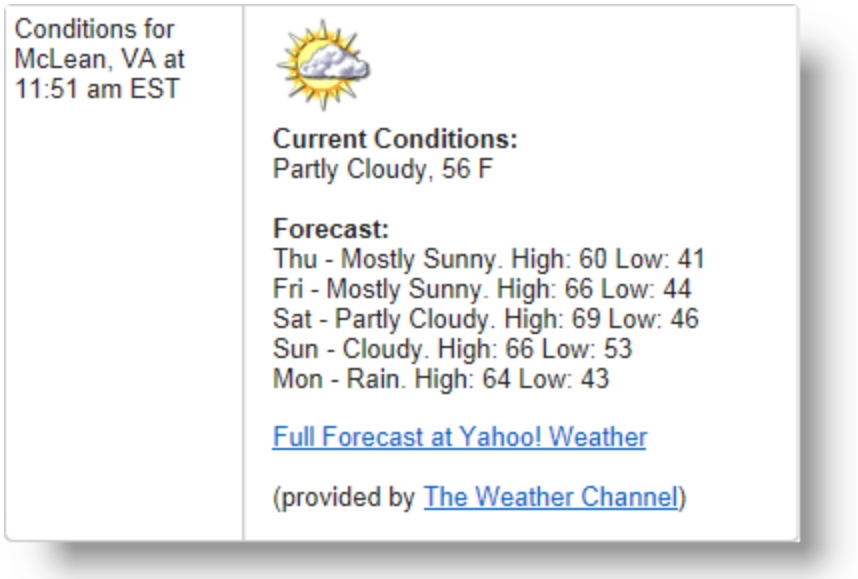
Next, add a **Data Table Column** element and set its attributes as shown above.



Followed by a **Label** element to display the data, as shown above.



Finish by adding another Data Table Column and Label combination. Set the Data Table Column element's **Class** attribute to *ThemeAlignTop* as before but leave its **Width** and **Width Scale** attributes blank. Set the Label attributes as shown above.



If you Preview the report now, you'll see the local forecast for the specified zip code, which might look like the example above.

In summary, we've seen how jQuery can be used in Logi report definitions, how XML data can be included as JSON data, and how JSON data can be converted to XML for use with Logi elements. A final, important reminder: jQuery, like all JavaScript, is *case-sensitive* and you can head off many errors by paying strict attention to case.

Token Reference

A Logi "token" is a **place-holder variable** that represents a value and is resolved at runtime. They're essential for creating dynamic visualizations and reports.

The following topics discuss tokens:

For information about functions and operators, see "Built in Functions and Operators" on page 632.
For information about special query string parameters, see "Query String Parameter Reference" on page 660.

About Tokens

Tokens act as variables in Logi applications and are the primary means of accessing the values of data, request and session variables, cookies, constants, and more. They're resolved into their values at runtime and inserted into the HTML generated by the Logi engine.

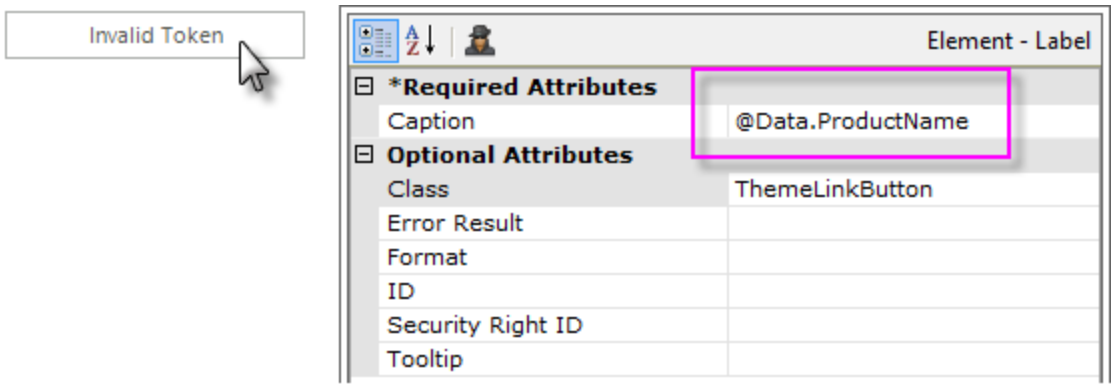
Tokens take this general format: `@<TokenType>.<Identifier>~` and here are some examples:

- `@Data.CustomerID~`
- `@Request.StartDate~`
- `@Function.UserName~`

They must start with the **@** symbol and end with the **~** (tilde) symbol, and they're **case-sensitive**.

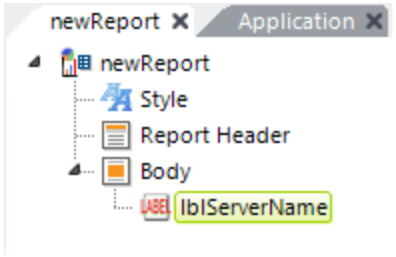
`@Data.CustomerID~` does *not* equal `@Data.customerid~`

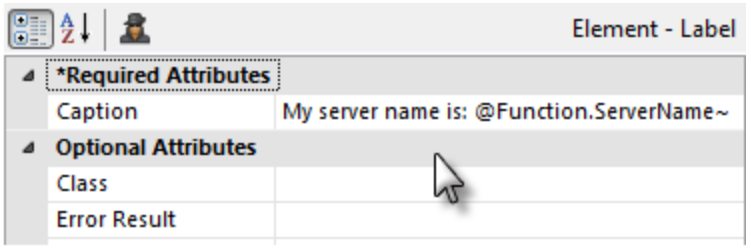
That bears repeating as developers typically run afoul of these two things: tokens *must* end with a ~ and they're *case-sensitive*.



Logi Studio includes a "token validator" feature. If you enter an invalid token inan attribute value, Studio will display the "Invalid Token" button shown above. In the example, the ~ (tilde) has been left off of the token. Clicking the button will take you to the invalid token(s) in the definition.

Tokens can be used in many, but not all, element attributes to provide a dynamic value. Here are three examples of tokens in action:

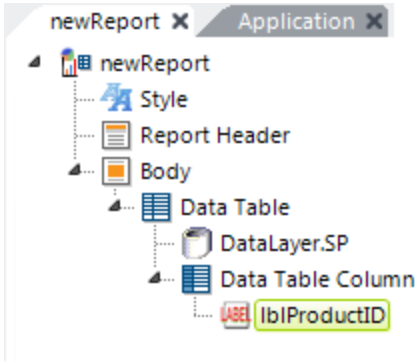


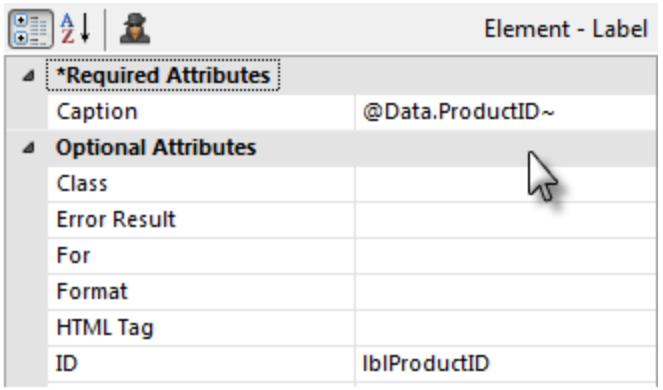


1. In the example above, a token is used to provide part of a Label element **Caption**. At run time, the real server name will be substituted for the token. The output will look like this:

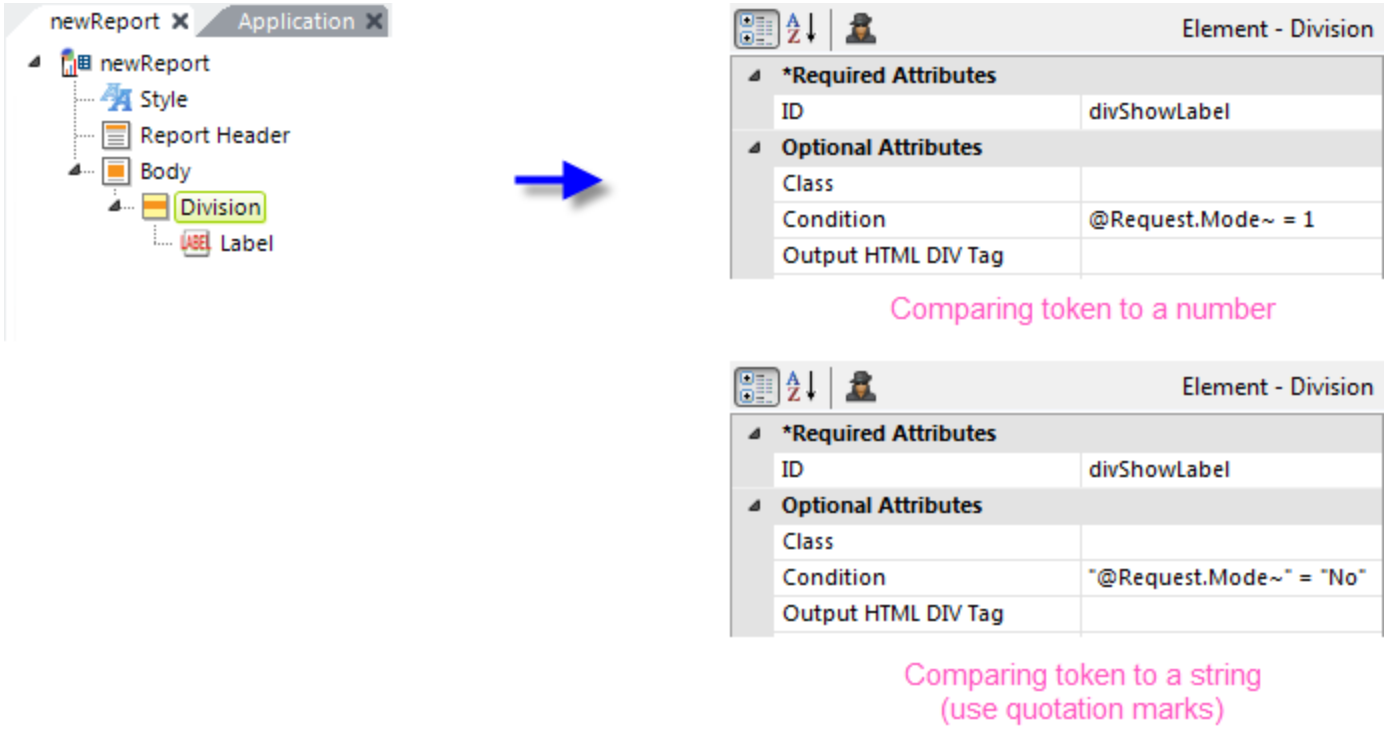
```
My server name is: MyTestServer
```

Tokens are extremely literal and are "data type neutral". For instance, in the above example, even though the server name is a string, there is no need to *concatenate* the "My server name is:" part of the caption (using + or & symbols) and the token.





2. As mentioned earlier, tokens are case-sensitive. In the example above, a Data token is used to represent the value in the **ProductID** column of each row of data in the datalayer. The spelling and capitalization of the column name in the token must *exactly* match that of the column name in the datalayer.



3. In our final example, shown above, a token is used in a Division element's **Condition** attribute. This attribute controls whether or not the Division, and all its child elements, are displayed: if the attribute evaluates to *True*, it's shown; if it evaluates to *False*, the element is hidden. A token can be used in this attribute in a **formula** that evaluates to *True* or *False*. More information is available about this in "Conditions" on page 670.

The example uses a Request-type token, which represents data either POSTed by an HTML form or values from parameters in the query string used to call reports. Therefore, if the URL for this report was:

```
http://localhost/helloworld/rdPage.aspx?&rdReport=Default&Mode=1
```

then the Division element *would* be shown. These URLs:

```
http://localhost/helloworld/rdPage.aspx?&rdReport=Default&Mode=2
http://localhost/helloworld/rdPage.aspx?&rdReport=Default
```

would cause the Division element to *not* be shown. If the Mode parameter in the query string is non-existent, as in the third URL above, then the token is empty and the formula evaluates to *False*.

When you're comparing a token to a string value, be sure to surround the token *and* the string in quotation marks, as shown in the second set of attributes above.

Logi Studio also includes an "Intelligent Token Completion" feature that makes it easy to enter tokens in attribute values, thus avoiding spelling and case-related errors.

"Nesting" Tokens

Because the token is a placeholder that gets resolved at runtime, it's possible to make token identifiers dynamic. This is done by making part of the token identifier itself a token. Consider this token construction:

```
@Request.@Local.VarName~~
```

Notice that it has *two* @ signs and *two* tildes. For clarity in understanding how it's processed, imagine there are parenthesis in there, like this:

```
@Request.(@Local.VarName~)~
```

As you can probably now guess, when the report runs, the "inner token" (inside the parenthesis) is resolved first, then the "outer token", which now includes the literal value of the inner token as its identifier. So, for example, if:

- A Local Data element returns data in a column named VarName, value = "Mode", and
- A Request variable exists named Mode, with a value of "Standard"

Then the token `@Request.@Local.VarName~~` would resolve to "Standard".

Limitations

Nesting only works with Constant, Data, Local, Request, and Shared tokens, and only *one* level of nested tokens will be resolved, so this *won't work*:

```
NO: @Request.@Local.@Local.VarName~~~
```


You may find that it's possible to nest the token *type* as well and even insert literal text into the token and make it work, but

those combinations are not supported officially and may break from one Info version to the next - proceed down that path at your own risk.

Token Types

The following types of tokens are available in Logi reporting products:

Token Type	Description
@Application	Represents values set in the **Application** object from Application State, set programmatically by another application. Does *not* work for values set directly in `Web.config` or `web.xml` files. These values are for global variables that span *all* instances of the application. Format: @Application.<*session param ID*> Example: @Application.LogFileFolder~
@Chart	Represents a data value from the child datalayer of a **Chart** or **Chart Canvas** element. Format:@Chart.<*column name*>~ Example: @Chart.OrderDate~
@Compare	Represents the *True* or *False* result of a **Compare Filter** or **SQL Compare Filter** evaluation. Format:@Compare.<*Compare Filter Element ID*>~ Example: @Compare.fltContainsDate~
@Constant	Represents the value of a user-defined constant, as defined in the _Settings definition in the **Constant** element. Format:@Constant.<*Name Attribute*>~ Example: @Constant.CompanyName~
@Cookie	Represents the value of a cookie. Cookies can be set in Logi applications using the **Save In Cookie** attribute of certain Input elements or, in Logi Info, by using the **Set Cookie Vars** procedure element.

Token Type	Description
	Format:@Cookie.<*Input Element ID*>~ Example: @Cookie.Order~
@Data	Represents the data value from the child datalayer of an element, such as a **Data Table**, that is not a chart. Format:@Data.<*Column Name*>~ Example: @Data.CustomerID~
@Date	Represents relative dates for reports and report scheduling (see "@Date Tokens" on page 603). Format: @Date.<*date identifier*>~ Example: @Date.Yesterday~
@FileUpload	Returns information in a procedure task about a file upload process (see "@File Upload Tokens" on page 611).
@Function	Represents the value of predefined functions (see "@Function Tokens" on page 599) Format: @Function.<*function name*>~ Example: @Function.DateTime~
@Input	Input parameter variable, used inside external JavaScript files. Format: @Input.<*value ID*>~ Example: @Input.FileName~
@Local	Represents a data value from the *first* row of the child datalayer of a Local Data element, see *Datalayer Introduction*. Local data is accessible anywhere within the report definition.  Child elements of this datalayer, such as a Calculated Column or Condition Filter, reference the data

Token Type	Description
	with @Data tokens. Format: @Local.<*value ID*>~ Example: @Local.ReportTitle~
@Measure	Represents the value of XOLAP measures; used only in the **XOLAP Formula** attribute of the **XOLAP Calculated Measure** element. Format: @Measure.[<*measure name*>]~ Example: @Measure.[Sales Amount]~
@Procedure	Represents results of operations by Process task procedures, including datasource interactions, file system and email operations, and associated error messages (see "@Procedure Tokens" on page 609). Format: @Procedure.<*procedure ID*>.<*value ID*>~ Example: @Procedure.GetTaxes.SalesTax~
@Repeat	Represents a data value from the child datalayer of a **Repeat Elements** element. Format: @Repeat.<*Column Name*>~ Example: @Repeat.data_type~
@Request	Represents values supplied in a URL's query string and in **Link Parameters**, and the values of User Input elements POSTed by an HTML form. Format: @Request.<*request variable name*>~ Example: @Request.EmployeeID~
@Session	Represents the values of the application's Session variables, which are unique for each user session. Format: @Session.<*session param ID*> Example: @Session.SessionID~

Token Type	Description
@Shared	Represents the parameter value passed from a Shared Element Params element. Format: @Shared.<*shared param ID*> Example: @Shared.myValue1~
@SingleQuote	A token prefix that provides special processing for other tokens. Prefix a token with @SingleQuote to wrap its values in single quotes. Used primarily to put single quotes around a comma-separated string of values, so they can be used in SQL queries. For example, values before: `Red,White,Blue` and after: `'Red','White','Blue'`. Cannot be used with @Data, @Chart and @Heatmap tokens. Format: @SingleQuote.<*token type*>.<*identifier*>~ Example: @SingleQuote.Request.MyElementID~

@Function Tokens

The identifiers listed in the table below are used with the **@Function** token. For example, to return the current date and time, use `@Function.DateTime~`.

Token	Resolves To
@Function.AppPhysicalPath~	The complete physical path of the application's virtual directory on the web server. Example: `C:\inetpub\wwwroot\MyLogiApp`
@Function.AppCachePath~	The physical path of the application's rdDataCache folder on the web server. Example: `C:\inetpub\wwwroot\MyLogiApp\rdDataCache`
@Function.AppDownloadPath~	The physical path of the application's rdDownload folder on the web server. Example: `C:\inetpub\wwwroot\MyLogiApp\rdDownload`
@Function.AppVirtualPath~	The HTTP Request.Path string.
@Function.Browser~	The HTTP Request.Browser string.
@Function.BrowserDecimalChar~	The client browser's decimal character.
@Function.BrowserMajorVersion~	The HTTP Request.Browser.MajorVersion string.
@Function.BrowserMinorVersion~	The HTTP Request.Browser.MinorVersion string.
@Function.BrowserThousandsSeparatorChar~	The client browser's thousand separator character.

Token	Resolves To
@Function.BrowserUserAgent~	The HTTP Request.UserAgent string.
@Function.BrowserVersion~	The HTTP Request.Browser.Version string.
@Function.Date~ *Deprecated, use @Date.Today~*	The current web server date as a *short date* string. E.g.: *05/21/2014*
@Function.DateTime~ *Deprecated, use @Date.Today~*	The current web server date and time as a *general date* string. E.g.: *05/21/2014 10:21:37*
@Function.ErrorDataLayerID~	When the If Data Error element is processed, provides the element ID of the datalayer that encountered the error.
@Function.FileUpload~	The information in a procedure task about a file upload process.
@Function.FUID~	A Functional Unique Identifier string, a unique 16-byte identifier used in Windows Portable Devices.
@Function.GlobalTheme~	Provides the name of the theme, if any, assigned to the application using a **Global Style** element in the _Settings definition.
@Function.GUID~	A random Globally Unique Identifier string. This function is useful for creating unique filenames. E.g.: *a949f8c9-a83d-46fd-970b-2bfb625bd2ab*

Token	Resolves To
@Function.HostAddress~	The IP address of the client (browser) computer.
@Function.InstanceID~	The Instance ID value of a Dashboard Panel, in Dashboards that have Multiple Instances set to *True*.
@Function.LastErrorMessage~	The last error message string for any process task executed in the current session.
@Function.PageCount~	The number of Data Table pages. Only valid when Interactive Paging or Printable Paging elements are in use.
@Function.PageNumber~	The current Data Table page number. Only valid when Interactive Paging or Printable Paging elements are in use.
@Function.QueryString~	The entire query string (everything after the "?").
@Function.Referer~	The complete URL of the calling web page.
@Function.RowNumber~	The current Data Table row number.
@Function.ServerEngineVersion~	The Logi Info Server Engine version number. E.g: *12.2.047*
@Function.ServerName~	The name of the web server. E.g.: *localhost*
@Function.ServerOperatingSystem~	The web server OS name, either *Windows*, *Unix*, or *Mac*.

Token	Resolves To
@Function.SessionID~	The current web server session ID.
@Function.TimeUtc~	The current web server time, translated to UTC (Greenwich Mean Time) time, in 24-hour format, as `hh:mm:ss`
@Function.UserCulture~	The browser's primary language string. E.g.: *en-us*
@Function.UserID~ @Function.UserName~	The current, logged-in user ID or name, when Logi Security (see *Intro to Logi Security*) is in use.
@Function.UserRights~	A comma-delimited list of the current, logged-in user's security rights, when Logi Security (see *Intro to Logi Security*) is in use. The token value is not available until the Security element has been processed completely.
@Function.UserRoles~	A comma-delimited list of the current, logged-in user's security roles, when Logi Security (see *Intro to Logi Security*) is in use. The token value is not available until the Security element has been processed completely.

@Date Tokens

The token identifiers listed in the table below all deal with dates. For example, `@Date.Yesterday~` will provide yesterday's date.

Dates are expressed by default in the *yyyy-M-d* format, which does not include leading zeros for single-digit day or month values. Examples: April 1st = "2009-4-1", April 25th = "2009-4-25", October 31st= "2009-10-31".

If needed, you can override this default format by specifying a new format in the Globalization element's **Default Input Date Reformat** attribute. For more information on globalization, see *Internationalization and Localization*. The Default Input Date Reformat attribute ensures that dates entered by the user are formatted identically to dates returned from @Date tokens. This is especially useful if you are using the @Date tokens with the Validation.Date element.

The Globalization element can be added in the _Settings definition.

To work with the `@Date.FiscalQuarter~` and `@Date.FiscalYear~` tokens, use the Globalization element to set the first day of the fiscal year.

@Date.TodayUtc~ returns the UTC (Greenwich Mean Time) date.

The TimePeriodColumn element can also be used to parse specific date parts out of @Date token values. For more information on the TimePeriodColumn, see *Send Cell Phone SMS Messages*.

Below is a list of the available @Date tokens:

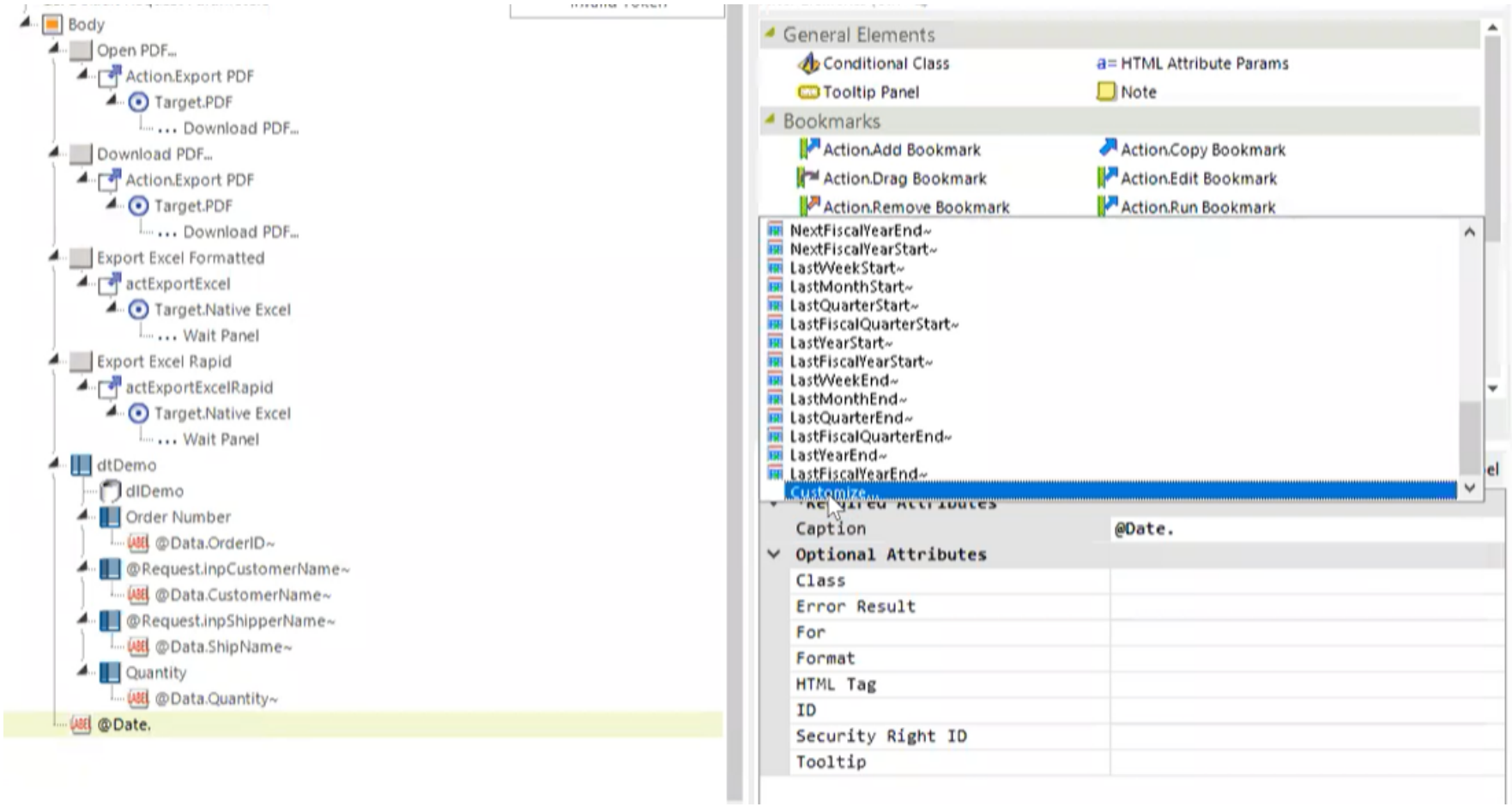
@Date.Today~	@Date.ThisWeekStart~	@Date.ThisQuarterStart~	@Date.ThisFiscalQuarterStart~
@Date.TodayUtc~	@Date.ThisWeekEnd~	@Date.ThisQuarterEnd~	@Date.ThisFiscalQuarterEnd~
@Date.Yesterday~	@Date.NextWeekStart~	@Date.NextQuarterStart~	@Date.NextFiscalQuarterStart~

@Date.Tomorrow~	@Date.NextWeekEnd~	@Date.NextQuarterEnd~	@Date.NextFiscalQuarterEnd~
@Date.10DaysAgo~	@Date.LastWeekStart~	@Date.LastQuarterStart~	@Date.LastFiscalQuarterStart~
@Date.30DaysAgo~	@Date.LastWeekEnd~	@Date.LastQuarterEnd~	@Date.LastFiscalQuarterEnd~
@Date.60DaysAgo~			
@Date.90DaysAgo~	@Date.ThisMonthStart~	@Date.ThisYearStart~	@Date.ThisFiscalYearStart~
@Date.180DaysAgo~	@Date.ThisMonthEnd~	@Date.ThisYearEnd~	@Date.ThisFiscalYearEnd~
@Date.365DaysAgo~	@Date.NextMonthStart~	@Date.NextYearStart~	@Date.NextFiscalYearStart~
	@Date.NextMonthEnd~	@DateNextYearEnd~	@Date.NextFiscalYearEnd~
	@Date.LastMonthStart~	@Date.LastYearStart~	@Date.LastFiscalYearStart~
	@Date.LastMonthEnd~	@Date.LastYearEnd~	@Date.LastFiscalYearStart~

Customizing @Date Tokens

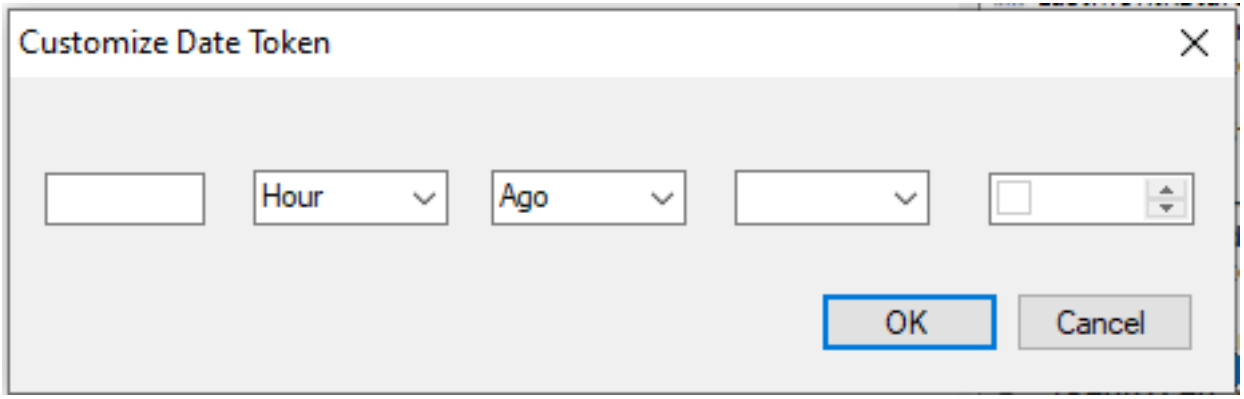
You can also customize the @Date token to retrieve relative dates for reports and report scheduling. This feature is available anywhere that you use existing @Date tokens.

1. Begin by double- clicking the **Customize...** option in the token value list:

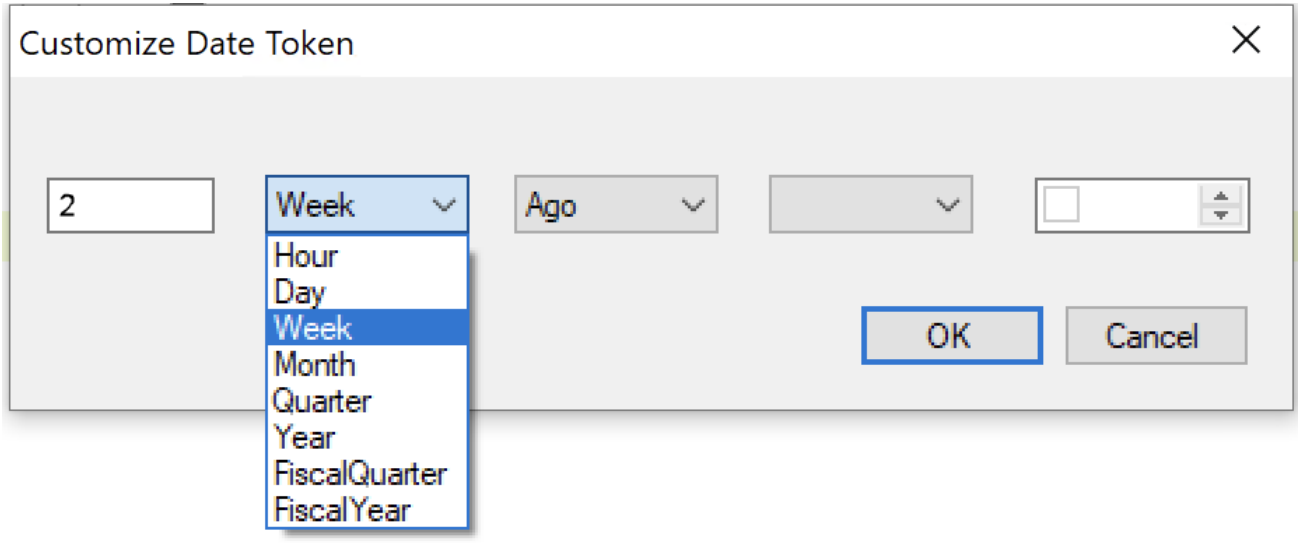


The Customize Date Token dialog displays.

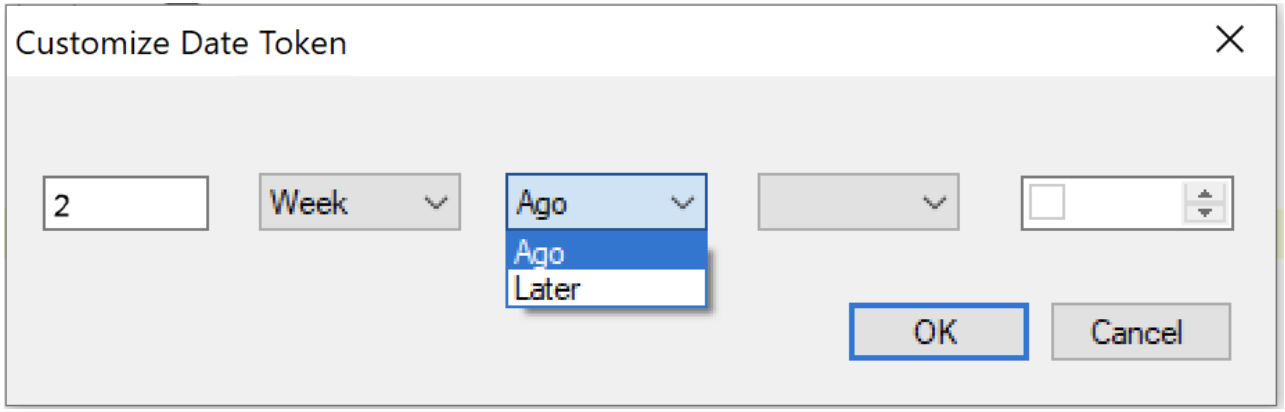
2. Enter a number in the first field:



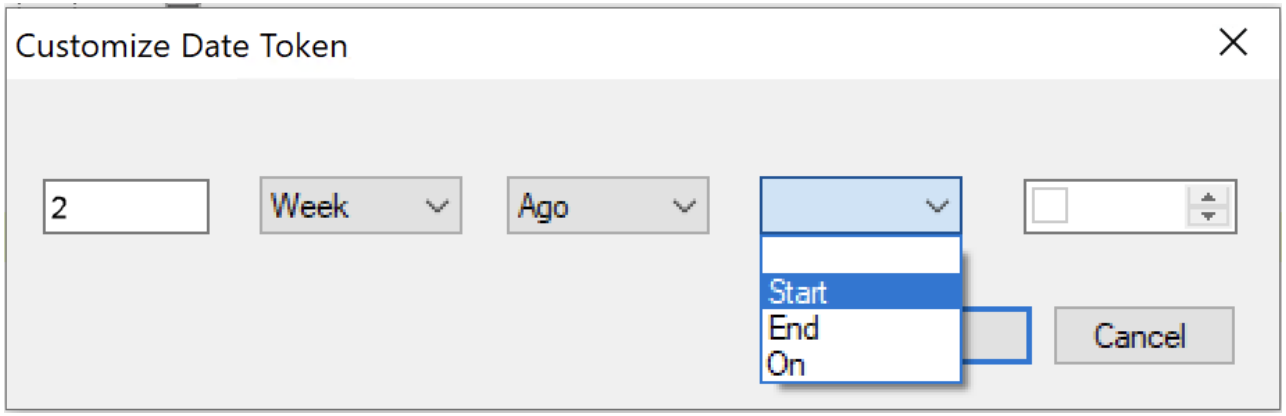
3. Then, select the **Date Section** drop-down and choose a **value**:



4. Next, select the **Date Direction** drop-down and choose a **direction**:



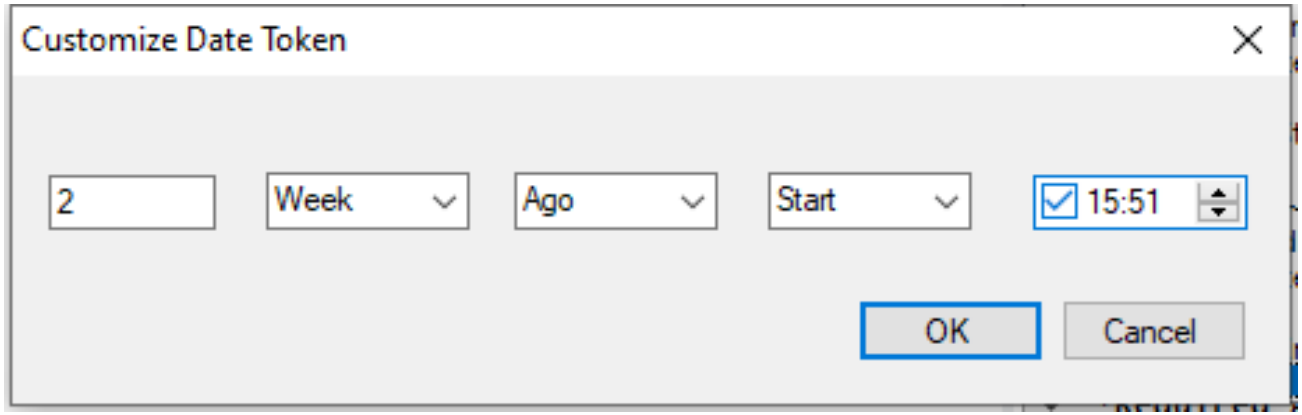
5. Select the **Date Edge** drop-down and select an **edge**:



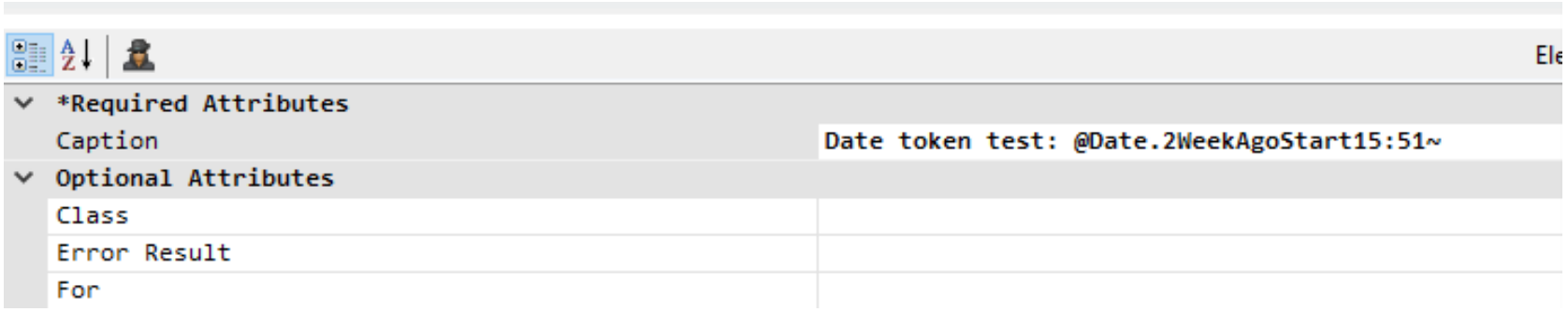
Depending on your selections, some options may not be available in other drop-downs. For example, if Date Section is set to "Hour", "On" is not available in this drop-down.

6. Select the **check box** to enable time input. Once enabled, enter the time (HH:mm). Depending on your selections, the time input may be disabled.

7. When finished, select **OK**:



The @Date token value automatically populates, like below:



You can also customize the @Date token by entering the value directly. This value is case sensitive.

@Procedure Tokens

These tokens are used within Process definition tasks:

Token	Resolves To
@Procedure.*myProcedureID*.*ColumnName*~	The data returned in the first row for the named column after **Procedure.SQL** executes. The element's **SQL Return Type** attribute must be set to *FirstRow*.
@Procedure.*myProcedureID*.ErrorMessage~	The last error message string for the procedure with the specified ID. Usage: `@Procedure.MyProcedureID.ErrorMessage~`
@Procedure.*myProcedureID*.ErrorOutput~	When using the **Procedure.Run Shell Command** element, if the element's Error Output Filename attribute has been left blank, this token contains the shell command's or application's error output. If no error has occurred, this token will have no value.
@Procedure.*myProcedureID*.ExitCode~	When using the **Procedure.Run Shell Command** element, contains the "exit" or "return" code from the shell command or application, often *0* for success.
@Procedure.*myProcedureID*.*MethodName*~	The data returned from an external web service after **Procedure.Web Service** executes. The Web Service Method element used in the procedure must have its **Return Type** attribute set to *String*.
@Procedure.*myProcedureID*.rdReturnValue~	The value of the stored procedure's Return Value, if any, after **Procedure.SP** executes. For Procedure.File Exists and Procedure.Folder Exists, this token will return "*True*" or "*False*".
@Procedure.RowsAffected~	The number of rows affected by an INSERT, UPDATE, or DELETE statement, after **Pro-**

Token	Resolves To
	cedure.SQL executes. Procedure.SQL element's **SQL Return Type** attribute must be set to *RowsAffected*.
@Procedure.*myProcedureID*.StandardOutput~	When using the **Procedure.Run Shell Command** element, if the element's Standard Output Filename attribute has been left blank, this token contains the shell command's or application's standard console output. If there is no standard output, this will have no value.
@Procedure.*myProcedureID*.*StoredProcedureOutputParamID~*	The data in an SP output parameter with the specified ID, after **Procedure.SP** executes. Usage: `@Procedure.MyProcedureID.MyOutputParamID~`
@Procedure.*myProcedureID*.TimedOut~	When using the **Procedure.Run Shell Command** element, contains *True* if a timeout occurred; otherwise contains *False*.

Four additional Procedure tokens specific to the File Upload process are shown in "@File Upload Tokens" on the next page.

@File Upload Tokens

A file upload is a two-step operation, as follows:

1. An Input File Upload element is used in a report definition to receive the file and path information for the file to be uploaded and it passes this information to a process task when the report is submitted.
2. The task uses a Save File Upload procedure to save the uploaded file.

Two different types of tokens are used to return the data from the steps in the upload operation. Due to the nature of the upload protocol, the values in the tokens are provided *after* the upload has already occurred. For more information, see "Upload Files to the Web Server" on page 453.

Token	Resolves To
@FileUpload.UploadFileName~	The file name and extension entered in the Input File Upload element by the user, without any path information.
@FileUpload.UploadFileExtension~	The file extension of the file name entered in the Input File Upload element.
@Procedure.*myProcedureID*.UploadFileName~	The file name and extension of the uploaded file.
@Procedure.*myProcedureID*.UploadFileExtension~	The file extension of the uploaded file.
@Procedure.*myProcedureID*.UploadFileContentType~	The MIME type or content type string of the uploaded file.

Token	Resolves To
@Procedure.*myProcedureID*.UploadFileLength~	The size of the uploaded file, in bytes.

@Crosstab Tokens

These tokens provide access to values within a Crosstab Table or Crosstab-filtered chart:

Token	Resolves To
@Chart.rdCrosstabColumn-*n*~	A datalayer column value, in a crosstab chart, where *n* equals the index, beginning with 1, of the column.
@Chart.rdStackedChartPercentage~	A stacked chart percentage value in a crosstab chart.
@Data.rdCrosstabColumn~	The crosstab column header value.
@Data.rdCrosstabValue-*n*-ExtraValueCol~	The Extra Crosstab Value Column value, where *n* equals the index, beginning with 1, of the column.
@Data.rdCrosstabValue~	The crosstab row value.
@Data.rdCrosstabValCount~	The number of rows that were used to calculate a crosstab row value.

Other Special Tokens

These tokens are used for special purposes:

Token	Resolves To
@Chart.rdExtraColumnID~	References different parts of the data (such as bar segments) when using an **Extra Data Column** element. Useful in determining which bar segment was clicked, for example. *This token is only available for Classic static XY-type charts, not animated or Chart Canvas charts.*
@Data.rdSalesforceTable~	A list of table names returned from a LIST TABLES query to Salesforce.com.
@Data.rdSalesforceField~	A list of field names returned from a LIST <*tablename*> query to Salesforce.com.
@Request.rdReport~	The name of the currently loaded report definition. Example: *Default*
@Request.rdReportFormat~	Sets/gets the exported report format as one of: *PDF*, *NativeExcel*, *NativeWord*, *CSV*, *HtmlExport*, *HtmlEmail*, *Excel*, *Word*, *XML*, or *GoogleSpreadsheet*. Doesn't exist (is null) for reports in browser.
@Session.SessionID~	The current session ID.
@Session.rdLastErrorLogFilename~	The error log file name, in the application's rdErrorLog folder, when error logging is enabled.

Token Encoders

Under certain circumstances, you may want a token value to be *encoded* in order to make it "safe" for use (meaning that it won't cause errors due to invalid characters). For example, a URL stored as data may contain characters that are actually invalid in a Logi page, or a session variable value may contain characters that are invalid when processed using JavaScript, or a user may enter non-ASCII characters into a text box. Encoding these characters avoids errors that might otherwise result.

Encoders can be used with @Data, @Local, @Request, and @Session tokens. The notation for a token with an encoder included takes this format:

```
@<TokenType>!<EncoderType>.<Identifier>~
```

Two "token encoders" are available: one that makes values safe for URLs (`!Url`), by applying HTML URL encoding, and one that makes values safe for use with JavaScript (`!Js`), by applying UTF-8 encoding. Like other aspects of tokens, the encoder names are *case-sensitive*.

A new token encoder (`!Json`) has been added for use with JSON data. It encodes the characters, such as double-quotes and line feeds, that are invalid in JSON data strings. An example use-case is encoding data entered by a user into an Input Text Area control prior to sending it in an array of JSON data to a web service using REST.

Here are some examples of them in use:

Original Value	Token with Encoder	Resulting Value
http://www.logixm.com\test	@Data!Url.SiteAddress~	http%3a%2f%2fwww.logixml.com%5ctest

Original Value	Token with Encoder	Resulting Value
www.xyz.co.uk?ra=1&id=56	@Session!Url.Redirect~	www.xyz.co.uk%3fra%3d1%26id%3d56
Rain and/or snow	@Request!Js.Weather~	Rain and\x2for snow
Topics	@Local!Js.Headings~	\x3cb\x3eTopics\x3c\x2fb\x3e
My "dog" has fleas	@Request!Json.inpText~	My \"dog\" has fleas

Encoders allow you to very selectively encode data right at the point in the definition where it will be used.

Dsexpression Reference

Many of the elements discussed here are available in Logi Info v12.5. They have been deprecated in later Info versions; consult the Release Notes for specific details.

The following topics provide details of the Logi Expression grammar, which is used with Discovery Module 3.x and provides a syntax for Data Service filter and calculation operations:

- General Syntax
- Math Functions
- String Functions
- DateTime Functions
- Logical Functions
- Conversion Functions

These are *not* the Intrinsic Functions and Operators generally used by Logi Info elements and only applies to Data Services operations using elements such as **DsCondition Filter** and **DsCalculated Column**. For more information about Logi Platform Services technology and Dataviews, see *Dataviews*.

General Syntax

The functions listed in the table below are for SQL queries and should not be used with inline scripts.

The following table lists the general syntax used in Logi Info:

Item	Description/Example
Comparison operators	`==`, `!=`, `<`, `>`, `<=`, `>=` Note that = and <>are *not* supported.
Logical operators	`AND`, `OR`, with precedence controlled using parentheses
Column data value placeholder	The column name, wrapped in square brackets: `[UnitPrice]` The column name is case-sensitive and there is no need to explicitly designate values as strings by surrounding their bracketed column names with quotation marks.
Token usage	Tokens, with the exception of @Data, may be used.

Math Functions

The functions listed in the table below are for SQL queries and should not be used with inline scripts.

The following table lists the math functions used in Logi Info:

Function	Description/Example
ABS(number)	Returns the absolute value of a number. e.g. ABS(-10) returns 10
EXP(number)	Returns e raised to the power of a given number. e.g. EXP(2) returns 7.38905609893065
INT(number)	Rounds the number down to the nearest integer. e.g. INT(3.995) return 3
LOG(number, base)	Returns the logarithm of a number to the base you specify. e.g. LOG(10,2.71828182845905) returns 2.30258509299405
MAX(number1, number2,...)	Returns the largest number in a set of values. Ignores logical values and text. e.g. MAX(10.01, 34, 22, 13.5, 21.563, -34) returns 34
MIN(number,...)	Returns the smallest number in a set of values. Ignores logical values and text. e.g. MIN(10.01, 34, 22, 13.5, 21.563, -31) returns -31
MOD(number, divisor)	Returns the remainder after a number is divided by a divisor. e.g. MOD(10, 4) returns 2
POWER(number, power)	Returns the result of a number raised to the power. e.g. POWER(2, 3) returns 8

Function	Description/Example
RANDOM()	Returns a random float between 0 and 1. Does not accept arguments. e.g. RANDOM() returns 0.937688089853821
ROUND(number, decimals)	Rounds a number to a specified number of digits. e.g. ROUND(10.43432,2) returns 10.43
SIGN(number)	Returns the sign of a number. 1 if the number is positive, 0 if the number is zero, or -1 if the number is negative. e.g. SIGN(345.342) returns 1
SQRT(number)	Returns the square root of a number. e.g. SQRT(9) returns 3

String Functions

The functions listed in the table below are for SQL queries and should not be used with inline scripts.

The following table lists the string functions used in Logi Info:

Function	Description/Example
CONCATENATE (string1, string2, ...)	Returns a string that is the result from joining two or more string values. e.g. CONCATENATE('Hello ', 'World') returns 'Hello World'
FIND(string,sub-string, start)	Returns the starting position of substring in string. The first character is in position 1 and the function returns 0 if the substring is not found. The optional start parameter can be used to control the search start position. e.g. FIND('Hello World', 'ello',1) returns 2
INSERT(string, start, length, sub-string)	Inserts a substring into string, at the specified position. The length determines how many of the original characters are overwritten by the new substring. E.g. INSERT('This is a test', 5, 4, ' really') returns 'This reallya test' and INSERT('This is a test', 5, 0, ' really') returns 'This really is a test'
LEFT(string, num-ber)	Returns the specified number of characters from the left side of the string. e.g. LEFT('Hello World', 5) returns 'Hello'
LEN(string)	Returns number of characters in a text string. e.g. LEN('Hello World') returns 11
LOWER(string)	Returns a string with all the letters in lowercase. e.g. LOWER('Hello World') returns 'hello world'

Function	Description/Example
MID(string, start, length)	Returns the characters from the start position for the specified length. e.g. MID('Hello World', 4, 4) returns 'lo W'
RIGHT(string, number)	Returns the specified number of characters from the right side of the string. e.g. RIGHT('Hello World',5) returns 'World'
SQL_FUNCTION (string)	(For Microsoft SQL Server only) Adds the provided string directly to the final T-SQL command sent to the database server, without any changes. In order to prevent SQL injection attacks, by default this function is disabled. Update the configuration in order to enable support. e.g. SQL_FUNCTION(' [Freight] + 5') would send [Freight] + 5 to the database.
TRIM(string)	Returns a string with any leading and trailing spaces removed. e.g. TRIM(' Hello World ') returns 'Hello World'
UPPER(string)	Returns a string with all the characters in uppercase. e.g. UPPER('Hello World') returns 'HELLO WORLD'

DateTime Functions

Supported date_part values are *Second*, *Minute*, *Hour*, *Day*, *Week*, *Month*, *Quarter*, and *Year*.

The functions listed in the table below are for SQL queries and should not be used with inline scripts.

The following table lists the datetime functions used in Logi Info:

Function	Description
DATEADD('date_part', interval, datetime)	Returns a datetime value by adding the interval to datetime. The type of interval is determined by date_part. e.g. DATEADD(Month, 4, '1997-04-20T11:30:15') returns '1997-08-20T11:30:15'
DATEDIFF('date_part', date1, date2)	Returns the difference between date1 and date2 in units of date_part. e.g. DATEDIFF('Month', '1997-04-20T11:30:15', '1997-07-20T11:30:15') returns 3
DATENAME('date_part', date)	Returns date_part of the datetime value as a string. e.g. DATENAME('Month', '1997-04-20T11:30:15') returns 'April'
DAYOFMONTH(date)	Returns the day of the month (number from 1 to 31). e.g. DAYOFMONTH('1997-04-20T11:30:15') returns 20
DAYOFWEEK(date)	Returns the weekday index of the date (1 = Sunday, 2 = Monday, ..., 7 = Saturday). e.g. DAYOFWEEK ('1997-04-20T11:30:15') returns 1
DAYOFYEAR(date)	Returns the day of the year (number from 1 to 366). e.g. DAYOFYEAR('1997-04-20T11:30:15') returns 110

Function	Description
DATEPART('date_part', date)	Returns date_part of the datetime value as an integer. e.g. DATEPART('Day', '1997-04-20T11:30:15') returns 20
NOW()	Returns the current date and time. Does not accept arguments.
TO_DATE(datetime)	Returns the date part of datetime value. e.g. TO_DATE([OrderDate]) returns the date without the time component.
TODAY()	Supported temporal values are Second, Minute, Hour, Day, Week, Month, Quarter, and Year.

Logical Functions

The functions listed in the table below are for SQL queries and should not be used with inline scripts.

The following table lists the logical functions used in Logi Info:

Attribute	Description
CONTAINS (string, sub-string)	Returns *true* if the string contains the specified substring. e.g. CONTAINS('Hello World','World') returns true if filter or 1 for calculated column.
ENDSWITH (string, sub-string)	Returns *true* if the string ends with the specified substring. E.g. ENDSWITH('Hello World', 'ld') returns true for filters or 1 for calculated columns.
IFNULL(value, default_value)	Returns the value if it is not null, otherwise the default_value is returned. Argument types for value and default_value must match. e.g. IFNULL([ShipVia],2) -if ShipVia is null then 2 is returned otherwise the value of ShipVia is returned.
IIF(expression, true_part, false_part)	Returns the true_part if the boolean expression is true, otherwise the false_part is returned. E.g. IIF(45 > 40, 'big', 'small') returns 'big'.
INLIST(test_ value, value1, ... valueN)	Returns *true* if the specified value matches any in the list. e.g. INLIST(4, 3, 5, 6, 4) returns *true* for a filter or 1 for a calculated column. Sample: INLIST([ShipVIA],1,2,3).

Attribute	Description
ISNULL(value)	Returns *true* if the value is null, false if it contains any value. e.g. ISNULL([ShipRegion]) returns *true* or *false* for filters and *0* or *1* for calculated columns.
STARTSWITH (string, sub-string)	Returns *true* if the string begins with the specified substring. E.g. STARTSWITH('Hello World', 'He') returns *true* for filters or 1 for calculated columns.

Conversion Functions

The functions listed in the table below are for SQL queries and should not be used with inline scripts.

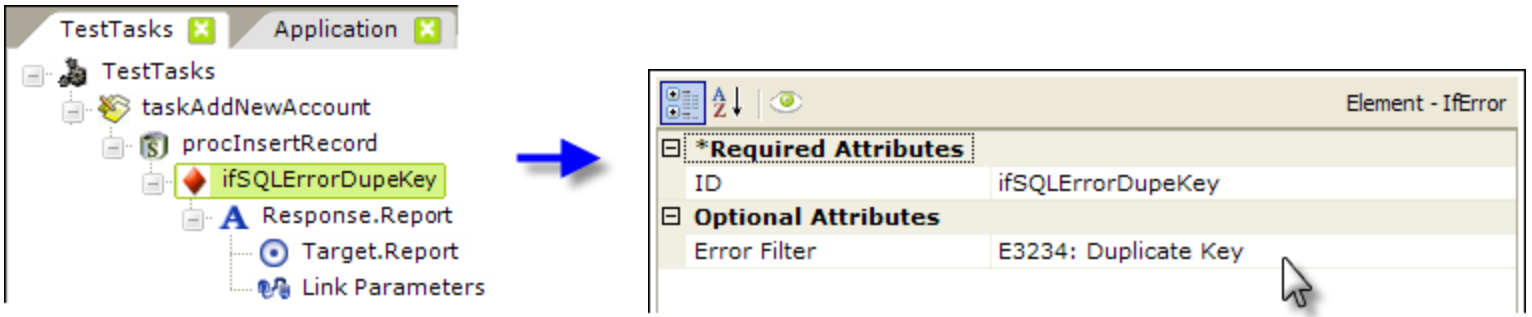
The following table lists the conversion functions used in Logi Info:

Function	Description
BOOLEAN_TO_STR (boolean)	Converts a boolean value to a string. Each data sources stores boolean values using different types. Some use integers, others use bit, and a few actually have a boolean type. Because of these type issues, the function code for each system expects sightly different values. In order to test this function or any of the boolean functions, please use a boolean field. e.g. BOOLEAN_TO_STR([Invoiced]) returns 'true' if Invoiced is *true*.
DATE_TO_UNIXTIME (datetime_value)	Converts a date time value into a unix timestamp. e.g. DATE_TO_UNIXTIME([OrderDate]) returns an integer like 1231451515
INT_TO_DATETIME (integer)	Converts a numeric integer value to date time value. e.g. INT_TO_DATETIME(1231451515) returns 2009-01-08 21:51:55
STR_TO_DATETIME (string)	Converts a string representation of a date value to date time value. String must be ISO8601 format. e.g. STR_TO_DATETIME('2015-04-30T22:45:33')
STR_TO_	Converts string value to long value (64bit integer). e.g. STR_TO_LONG('723874928') returns 723874928

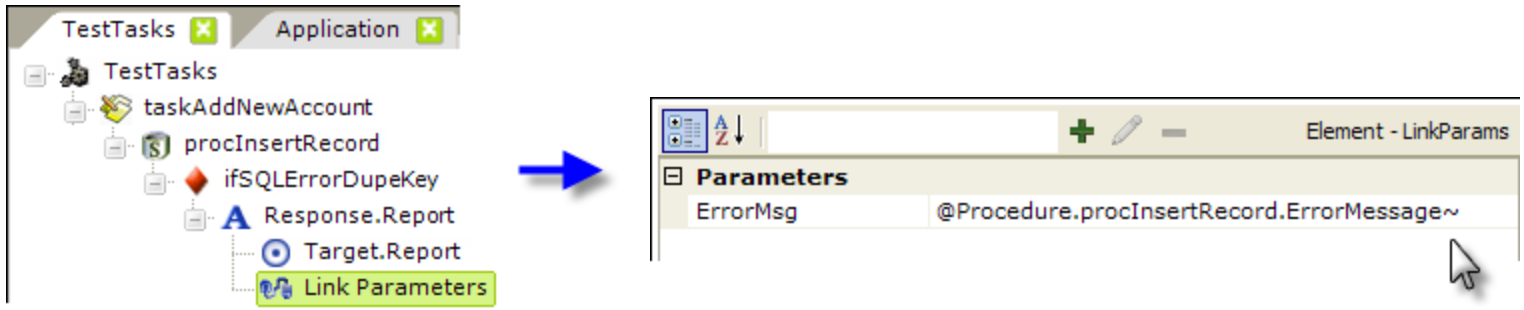
Function	Description
LONG(string)	
STR_TO_DECIMAL (string)	Converts string value to decimal value. e.g. STR_TO_DECIMAL('10.53') returns 10.53
STR_TO_BOOLEAN (string)	Converts string value to boolean value. e.g. STR_TO_BOOLEAN('true') depending on internal boolean type of system it could be a 1 or true
TO_STRING (any)	Converts value to string value. e.g. TO_STRING(20) will be converted to '20'

Handle SQL Errors in Tasks

Logi Info developers are able to use Process definition Tasks to manipulate their database tables. This could generate SQL errors, so it's good to be able to handle generic and specific errors. The following example assume that account names must be unique in the Accounts table.



In the Process definition shown above, a task has been added to add new accounts. It includes a **Procedure.SQL** element (procInsertRecord) which will insert the new account records. An **If Error** element has been added beneath it to handle any errors that occur, and to route processing back to a report definition. To handle *generic* SQL errors, you can leave the If Error element's **Error Filter** attribute value blank and it will "catch" any error that occurs as a result of the SQL operation. If you want to handle *specific* SQL errors, you can fill-in the **Error Filter** attribute with some of the text that the SQL server will return in an error message caused by a specific type of error. The example above suggests an error number and partial text from a duplicate primary key error message. If the text in the attribute value is contained in the server's error message, processing will be routed to the If Error element's child elements. You can stack up more than one If Error element, for example, to first handle several different specific errors and then finally handle a generic error.



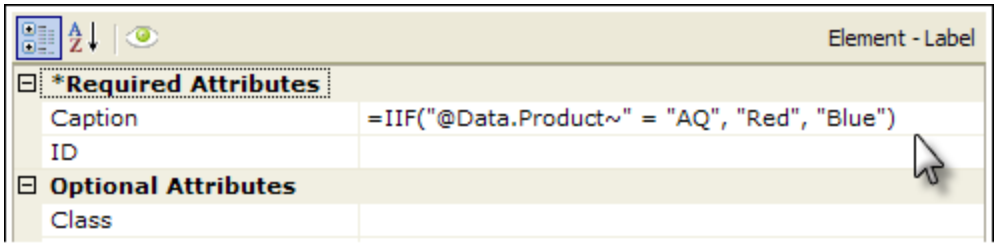
You can send the actual error message returned from the SQL server to the target report, as shown above. A special token is used for the error message. It starts with "@Procedure.", then includes the ID of the Procedure.SQL element used, another dot, and finally the words "ErrorMessage". Don't forget the tilde character ("~") that terminates every token!

```
@Procedure.procInsertRecord.ErrorMessage~
```

This technique allows you to show your own "user friendly" error messages for specific SQL errors, while sending along the exact SQL error message for any other SQL errors.

Label Elements Support Immediate IF

In Logi Studio, **Label** elements can make use of an intrinsic Immediate (or sometimes called "Inline") IF function. This allows you to do conditional processing in the label's Caption attribute value.



An example of this in action is shown above, where the general syntax for the IIF function is:

```
=IIF(statement to evaluate, true value, false value)
```

When using this function in a Caption attribute remember that you must preface it with an equals ("=") sign and, in the statement to be evaluated, when comparing strings the values on both sides of the equals sign must be within double-quotes, as shown.
More information about intrinsic functions can be found in "Built in Functions and Operators" on the next page.

Built in Functions and Operators

Logi Info includes a set of built-in scripting functions and operators that provide commonly-required functionality. They're available in all "formula"-capable element attribute values.

The following topics identify these functions and operators, as well as some reserved words you should be aware of:

- Functions
- Operators

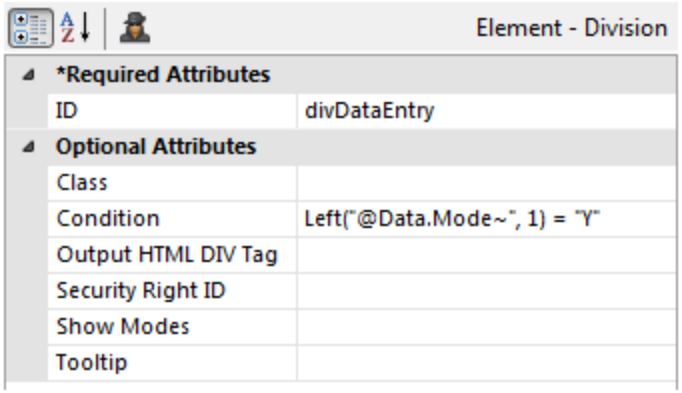
About Functions and Operators

Formulas are expressions made up of literals, tokens, functions, and operators. **Function names** are case-insensitive *reserved words*. Functions return values, which are usually the result of computations based on data and constants.

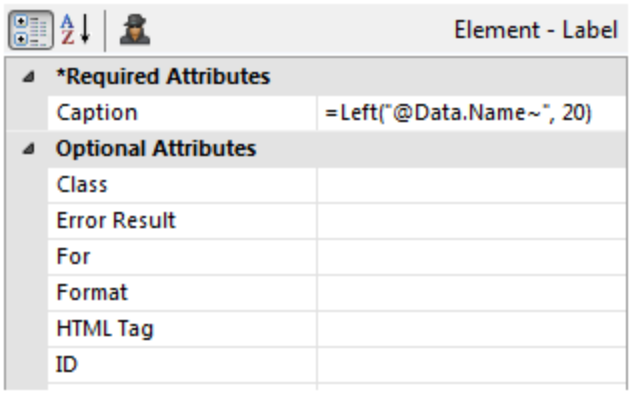
Using "Formula Attributes"

During development using Logi Studio, formulae with functions can be entered into many, but not all, attribute values.


So-called "formula attributes" are those that can evaluate a formula and/or resolve tokens in their values; *not all attributes are formula attributes*. With the exception of the ID attribute, token resolution is supported in *most* attributes. However, for reasons of backward compatibility, some tokens are not resolved in super-element formulae. There is no comprehensive list of the formula attributes - mostly because not all tokens are resolved equally, so for each attribute it's often more about what *type* of tokens can be evaluated, as opposed to whether tokens are evaluated at all. The method used to signal that an formula evaluation is needed depends on the attribute characteristics:



Division Element: Condition attribute uses boolean value, will evaluate a formula



Label Element: Caption attribute expects Text, will evaluate - *if* there's a leading equals sign

In attributes that expect a boolean, *True* or *False,* value, such as the **Division** element's **Condition** attribute, shown above left, any formula entered will be automatically evaluated. Similarly, any attributes that are actually named "Formula" or "Expression", as in the Calculated Column element, will expect to evaluate a formula.

However, in some attributes that typically expect text, such as the **Label** element's **Caption** attribute, shown above right, a leading equals ("=") sign must be used to trigger a formula evaluation. There is no published list of the elements that fall into this category, so learning them is a matter of experience. When in doubt, try it with and without the leading equals sign.

Evaluation errors will be displayed as either an empty value or "???" - when this occurs, syntax errors are a common cause. The "Debug Reports" on page 56 often provides detailed information about these errors.

Using Tokens

Tokens are placeholders for data or values, and are resolved at runtime by the Logi Server Engine. Datalayer data is represented in formulae using @Data tokens, in this format: `@Data.UnitPrice`, where UnitPrice is a column name. Tokens are *case-sensitive*. Some of the other tokens include @Request, @Local, and @Session. More information about tokens can be found in "Token Reference" on page 588.

Here are some examples of formulae using @Data tokens:

1. Multiply a data column by an constant value to calculate the tax applied to the price:

```
@Data.UnitPrice~ * .04
```

2. Get the number of days from the order date to the shipment date:

```
DateDiff("d", CXMLDate("@Data.OrderDate~"), CXMLDate("@Data.ShippedDate~") )
```

3. Concatenate columns and strings together, in a Label caption (example result: "Smith, John"):

```
=@Data.LastName~ +", " +@Data.FirstName~
```

Using Super-Elements

In super-elements, such as the Analysis Grid, users can build formulae at runtime in the element's user interface; functions can be part of these formulae and they're typed directly into the appropriate UI panels, or assembled using UI tools. Data is represented in these formulae by enclosing the column name within *square brackets*, for example:

```
[UnitPrice]
```

Here are some examples of formulae in a super-element user interface: 1. Multiply two data columns, UnitPrice and Quantity, to make an ExtendedPrice column:

```
[UnitPrice] * [Quantity]
```

2. Get the number of weekdays since the shipment date:

```
DateDiff("w", [ShippedDate], Now )
```

3. Concatenate columns and strings together (example result: "Smith, John"):

```
[LastName] + ', ' + [FirstName]
```

Functions

The following table describes the built-in functions, which accept one or more parameters and return a single value. Double-quotes are used around string parameters and tokens that contain date/time values. In the Syntax column below, parameters in square brackets are *optional*.

Function	Description	Syntax	Notes
Abs	Returns the absolute value of a number.	Abs(*number*)	Abs(-5) = 5
CXMLDate	Converts date in ISO format (such as those returned from SQL Server) into compatible format for manipulation by built-in functions.	CXMLDate("*date value*") Usage example: `If myDate = 2014-10-02T13:30:00 then CXMLDate("@Data.myDate~") returns "10/2/2014 13:30:00"`	ISO 8601 format is `2014-05-31T13:30:00` representing `yyyy-mm-ddThh:mm:ss` See "Special Functions and Attributes" on page 655.
Date	Returns the current date.	Date()	

DateAdd	Adds or subtracts an interval of time from a date or time; returns a date value	DateAdd("*interval*", *number*, *"date value"*) Usage examples: `where Posted = "10/21/2014", DateAdd("d", 7, "@Data.Posted~") returns "10/28/2014" DateAdd ("d", -7, "@Data.Posted~")`	*Interval* may be:
			`yyyy = year`
			`q = quarter`
			`m = month`
			`y = day of year`
			`d = day`
			`w = weekday`

Function	Description	Syntax	Notes
		returns "10/14/2014"	ww = week of year h = hour n = minute s = second
DateDiff	Returns the difference between two dates.	DateDiff("*interval*", *"date value 1"*, *"date value 2"*)Usage example: where StartDate= "10/02/2014" and EndDate = "11/02/2014", DateDiff ("d","@Data.StartDate~", "@Data.EndDate~") returns 30	For valid *interval* values, see DateAdd function
DatePart	Returns a part of a date.	DatePart("*interval*", *"date value"* [, *FirstDayofWeek* [, *FirstWeekofYear*]]) Usage example: where myDate = "10/02/2104", DatePart("m", "@Data.myDate~") returns 10	For valid *interval* values, see DateAdd function. *FirstDayofWeek* may be: 0 = Use SystemDayOfWeek 1 = Sunday (default) 2 = Monday 3 = Tuesday 4 = Wednesday 5 = Thursday

Function	Description	Syntax	Notes

			`6 = Friday`
			`7 = Saturday` *FirstWeekofYear* may be:
			`0 = Use System`
			`1 = Week in which Jan 1 occurs (default)`
			`2 = The first week with at least four days in new year`
			`3 = The first full week of the new year`

DateSerial	Combines date parts together to make a complete date.	DateSerial(*year*, *month*, *day*)
		Usage example:
		`where myYear = 2014, DateSerial(@Data.myYear~, 10, 2)`
		`returns "10/2/2014"`	

DateValue	Returns a valid date-type value created from a text-type argument. Can convert text dates in many different formats.	DateValue("*date string*") Usage examples: `where myDate = "2014-10-2"), DateValue("@Data.myDate") - or - DateValue("2-Oct-2014") - or -`	

Function	Description	Syntax	Notes
		`DateValue("October 2, 2014")` returns `"10/2/2014"`	
Day	Returns the day of the month.	Day(*date*)	Possible return values are from 1-31.
Exp	Returns "e" raised to a power. "e" is the base of natural logarithms, called the antilogarithm	Exp(*number*)	
FormatCurrency	Formats a number value into currency	FormatCurrency(*number* [, *NumDigitsAfterDecimal* [, *IncludeLeadingDigit* [, *UseParensForNegativeNumbers* [, *GroupDigits*]]]])	
FormatDateTime	Formats a date-time value.	FormatDateTime(*date* [, *NamedFormat*])	*NamedFormat* may be `vbGeneralDate`, `vbLongDate`, `vbShortDate`, `vbLongTime`, or `vbShortTime`
FormatNumber	Formats a number.	FormatNumber(*number* [, *NumDigitsAfterDecimal* [, *IncludeLeadingDigit* [,	

Function	Description	Syntax	Notes
		UseParensForNegativeNumbers [, *GroupDigits*]]]])	
FormatPercent	Formats a number as a percentage.	FormatPercent(*number* [, *NumDigitsAfterDecimal* [, *IncludeLeadingDigit* [, *UseParensForNegativeNumbers* [, *GroupDigits*]]]])	
Hour	Returns the hour of the day	Hour(*time*)	Possible return values are 0-23.
IIF	Returns one value if the expression evaluates *True*, another value if *False*. Can be nested./	IIF(*expression*, *true-value*, *false-value*	*expression* is a formula that returns *True* or *False*. Example: this returns "Blue": IIF(1=2,"Red","Blue") See "Special Functions and Attributes" on page 655.
InStr	Returns the starting character position where one string is found within another string.	InStr([*start*,] *string1*, *string2* [, *compare*])	*string1* is string to search, *string2* is string to search for. Returns *0* when the string is not found. The first characters is at position 1. Set *compare* to 1 for case-insensitive searches.

Function	Description	Syntax	Notes
InStrRev	Same as InStr( ), but search begins from end.	InStrRev(*string1*, *string2* [, *start*,] [, *compare*])	See Instr( )
Int	Returns the integer portion of a number, removing any decimal places.	Int(*number*)	
IsDate	Returns *True* if the text is a date.	IsDate(*text*)	
IsNumeric	Returns *True* if the text is a number.	IsNumeric(*text*)	
LCase	Converts all characters to lower case.	LCase(*text*)	
Left	Returns x characters from text, starting from left.	Left(*text*, *x*)	
Len	Returns the number of characters in the text.	Len(*text*)	
LTrim	Removes any spaces from left end of text.	LTrim(*text*)	

Function	Description	Syntax	Notes
Mid	Returns sub-string of characters from the middle of the text.	Mid(*text*, *start* [, *length*])	*start* is the position of first character to be returned. The first character of the entire text is at position 1. *length* is the number of characters to be returned.
Minute	Returns the minute of the hour.	Minute(*time*)	Possible return values are 0-59.
Month	Returns the month of the year.	Month(*date*)	Possible return values are 1-12.
MonthName	Returns the name of the month.	MonthName(*month#* [, *abbreviate*])	Set *abbreviate* to *True* for an abbreviated month name.
Now	Returns the current date and time.	Now()	
Replace	Replaces instances of text with other text.	Replace(*text*, *textFind*, *textReplaceWith* [, *start* [, *count* [, *compare*]]])	*text* is the original text, *textFind* is the text to be replaced, *textReplaceWith* is the replacement text *start* is the starting character position to be searched

Function	Description	Syntax	Notes
			count is the maximum number of replacements before stopping. Set *compare* to 1 to replace characters regardless of case. Example: Replace("ABC", "abc", "123", 1, 1) produces "123".
Right	Returns x characters from text, starting from right.	Right(*text*, *x*)	
Rnd	Returns a random number between 0 and 1.	Rnd([*number*])	If *number* is: not supplied - returns the next random number in the sequence < 0 - returns the same number every time > 0 - returns the same number every time = 0 - returns the most recently generated number
Round	Returns a number rounded	Round(*expression* [, *num-*	v23.1 If midpointRounding *num-*

Function	Description	Syntax	Notes
	to a specified number of decimal places.	*decimalplaces*, *midpointRounding*])	*ber* is: 0: The strategy of rounding to the nearest number. When a number is halfway between two others, it's rounded toward the nearest even number. 1: The strategy of rounding to the nearest number. When a number is halfway between two others, it's rounded toward the nearest number that's away from zero. The default Value is 0. Example: Round([Freight], 1) - 30+0.05 You cannot use midpointRounding when the Round formula uses an Active SQL AG/Table/Crosstab.
RTrim	Removes any trailing	RTrim(*text*)	

Function	Description	Syntax	Notes
	spaces from right end of text.		
Second	Returns the second of the minute.	Second(*time*)	Possible return values are 0-59.
Sgn	Returns indication of number's sign.	Sgn(*number*)	Returns: -1 if the number is negative, 1 if the number is positive, 0 if the number is 0.
Space	Returns a string consisting of the designated number of spaces.	Space(*number*)	
Sqr	Returns the square root of a number.	Sqr(*number*)	
String	Returns text consisting of the character duplicated x number of times.	String(*x*, "*character*")	
StrReverse	Returns the text with the characters in reverse order.	StrReverse(*text*)	

Function	Description	Syntax	Notes
TimeValue	Returns a valid time-type value created from a text-type argument. Can convert text dates in many different formats.	TimeValue(*time*)	
Trim	Removes both leading and trailing spaces from the text.	Trim(*text*)	
UCase	Converts all characters to upper case.	UCase(*text*)	
Weekday	Returns the number of the day of the week.	Weekday(*date* [,*firstdayofweek*])	Possible return values are 1-7. The first day of the week defaults to 1 = Sunday, but the optional argument can be used to set it differently: 2 = Monday, 3 = Tuesday, etc.
WeekdayName	Returns the name of the day corresponding to the weekday number.	WeekdayName(*numberWeekday*, *abbreviate*, *firstdayofweek*)	
Year	Returns the number of the year of the specified date.	Year(*date*)	

Operators

The following table describes the built-in operators, which are used for arithmetic operations and logical comparisons. Some built-in operators may be overridden depending on scripting language choice, as noted.

Operator	Description
-	Negation
*	Multiplication
/	Division
\	Integer Division
%	Modulus
+	Addition
-	Subtraction
+	String Concatenation
==	Equal Comparison
=	Assignment
!=	Not Equal Comparison

Operator	Description
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To
!	Logical NOT
&&	Logical AND
||	Logical OR
(and)	Parenthesis, to manage precedence

You may represent true and false values as *True* and *False*.

Special Constants

Logi Info makes use of a set of special constants and "reserved" words that developers should be aware of. The constants can often be used to enable or disable specific behavior or functionality.

The following special **constants**, which can be defined in the _Settings definition, have specific meaning for the Logi Engine. The version numbers indicate when they were introduced but they may have been phased out in later product versions. Some constants may be automatically created by the New Application wizard. Remember that constants are *case-sensitive.*

Constant	Description
FirstDayOfQ1	Sets the month to be used for Fiscal Quarter calculations by the Globalization and Time Period Column elements, using the format *MM/YY*.
rdAnimatedChartRenderStyle	Specifies the rendering style, *JavaScript* or *Flash*, for Animated Charts and Gauges. If this constant is not present, the default is *Flash*.
rdCalculationsIncludeNulls	By default, aggregating functions don't include null values in their calculations. This constant can be created and set to *True* to cause null values to be included. This is global in scope.
rdComboLoader	By default, "combo loading" is used to pre-load interface script files. Set this constant to *False* to disable the pre-loading.
rdDefaultCsvEncoding	Exports to CSV default to the "utf-8" format, which opens in Excel. Developers who wish to change the encoding to "ASCII" can create and set this constant to *ISO-8859-1*.

Constant	Description
rdFilterRetainJoinType	Used to control filtering behavior of tables that have Left joins. When set to *True,* retains the Filter Type for exclusive filters on Left-joined tables. If missing or not *True*, the default behavior will be to always filter the full dataset.
rdFlexSort	Sorting performance improvements were introduced in this version, using a scheme named **FlexSort**, and became the default sorting mechanism. Developers who wish to continue to use the older sorting scheme can do so by creating and setting this constant to *False*.
rdInfinityErrorResult	Specifies a global result for formulae that may produce a divide-by-zero situation (result: infinity), sparing developers the task of specifying a result on a per-element basis in their Error Result attributes.
rdJavaPdfClearCache	(Java apps only) - Create and set this constant to *True* to force clearing of the PDF cache for each export.
rdJavaPdfStitchSize	(Java apps only) - Controls the size of blocks of HTML exported to the PDF rendering engine, in megabytes, which are then "stitched" together. The default value is 5MB. When exporting very large amounts of data (thousands of pages) this constant can be used to increase the block size, decreasing the chance that memory will be exhausted by the process. This also increases the rendering time, however.
rdJavascriptInfinityIsError	When set to *True*, specifies that formulae that produce a divide-by-zero situation (result: infinity) will trigger an error, which is handled by elements' Error Result attribute or, if set, the rdInfinityErrorResult constant. If rdJavascriptInfinityIsError is not specified, the Script-

Constant	Description
	ing Engine will return the value *NaN* (for "Not a Number") by default when a division by zero occurs.
rdMemoryStreamLimit	Used with the hybrid data engine. Sets the threshold, in MB, for shifting from memory caching to file system caching. A value of *0* will force all caching to the file system; a value of *2048*, the maximum, will force all caching to memory. The default value is *10*.
rdMinifiedJavaScript	When set to *False*, disables the "minification" of Logi Engine JavaScript code. This can be useful when debugging script-related errors. The default value is *True*.
rdMultiThreadXslDataLayers	When set to *False*, disables multi-threaded processing of datalayers (which was introduced in v12). The default value is *True*.
rdPdfRenderingType	In order to enable Printer Page Breaks, in reports that use them, when the report is exported to PDF, this constant should be created, with the value *MSHTML*.
rdScriptingSingleQuotedString	By default, single- and double-quotes are supported as quoted string delimiters in scripting. However, single-quote support can be turned off by creating and setting this constant to *False*.
rdSQLIncludeGMTOffset	Used to speed up the internal processing of DateTime-type data into ISO 8601 (yyyy-mm-dd) format, by skipping Time Zone information. Does not produce significant performance improvements with a small number of DateTime columns; intended for result sets with large number of DateTime columns.

Constant	Description
rdTempFileCleanupInterval	Used to manage cached temporary files. Set this constant to specify the time, in minutes, between clean ups. If this value is *-1*, clean up will not run at all; if the value is *0*, clean up will run with *every* page request. Default value: *5 minutes*
rdTempFileLifespan	Used to manage cached temporary files. Set this constant to specify the minimum file age, in minutes, before a file qualifies to be deleted when the clean up runs. Default value: *60 minutes*. Your web server's **Session Timeout** setting should *not* be configured to be *less* than the value assigned to `rdTempFileCleanupInterval` or its default value of 5 minutes.

Reserved Words

Logi Info makes use of a set of special constants and "reserved" words that developers should be aware of. Developers should avoid using the reserved words in their definitions as identifiers as they may cause errors when application runs.

The Logi Server Engine deals with elements and data as XML and this can produce some parsing challenges. Developers should be careful not to assign names to elements, session variables, request variables, or link parameters that match the following words:

- rd* (any word that begins with "rd")
- Action

A general rule of thumb is to avoid using any attribute name as an ID or as a name for elements, session variables, request variables, or link parameters. Common attribute names include:

Axis	Font	Location	Size
Border	Format	Longitude	Source
Caption	Formula	Method	Style
Class	Function	Modal	Text
Color	Height	Namespace	Theme
Columns	HTML	Note	Thickness
Command	ID	Ordered	Title
Condition	Image	Orientation	Tooltip
Cube	Increment	Password	Top
Database	Iteration	Period	Type
Display	Javascript	Port	URL
Domain	Keyword	Radius	User
Event	Language	Refresh	Value

Expression	Latitude	Rows	Width
Filename	Layout	Scrolling	Worksheet
Folder	Length	Server	XPath

When in doubt, use the Element Referencepage to check attribute names.

Special Functions and Attributes

Logi Info includes special functions and attributes for special purposes.

The following topics discuss the special functions and attributes:

- IIF Function
- CXMLDate Function
- DontResolveTokensInData Attribute

IIF Function

The built-in IIF() function is similar to the "inline if" found in other languages and the ternary operator (?) found in standard JavaScript. This function, which allows *conditional processing* in a single line of code, is very useful in element attributes that accept expressions. The function's syntax is

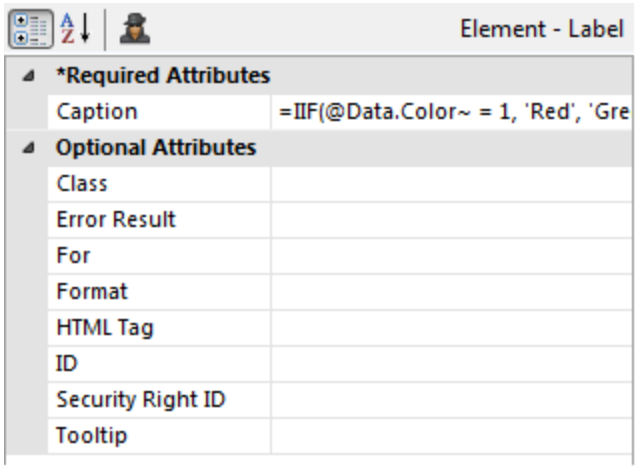
```
IIF(<expression>, <value if true>, <value if false>)
```

The following examples illustrate some of the ways to use the IIF function.

The function name may need to be prefixed with an equals sign depending on the element and attribute.

In a Label element's Caption attribute

The IIF function can be used in any "formula attribute", an attribute that is capable of evaluating a formula, such as the **Label** element's **Caption** attribute.



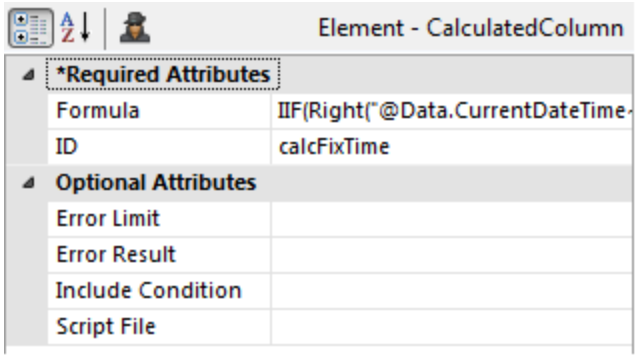
For example, as shown above,

```
=IIF(@Data.Color~ = 1, 'Red', 'Green')
```

will display the appropriate text, Red or Green, in the Label based on the data value.

In a Calculated Column Element's Formula Attribute

The IIF function can be used in combination with other intrinsic string functions in an expression to generate the correct value.



If used in the **Calculated Column** element's **Formula** attribute, as shown above, the IIF function can be used to generate a new column that replaces part of a SQL time string with the Eastern Daylight Time or Eastern Standard Time acronym (EDT or EST):

```
IIF(Right("@Data.CurrentDateTime~",6) = "-04:00", "EDT", "EST")
```

The equals sign is not required before the "IIF" in this attribute. Relatively complex combinations of string functions can be created, including those that incorporate nested IIF functions.

CXMLDate Function

DateTime-type data returned into a datalayer from a databases is often in ISO 8601 format, which looks like this:

```
2007-05-31T13:30:00 (representing yyyy-mm-ddThh:mm:ss)
```

If you wish to use script functions to **compare, manipulate,** or **format** the date data, you need to convert it into a compatible format. Logi's intrinsic **CXMLDate** function has been provided for this purpose. The following example uses an intrinsic function to add one day to a date value,

```
=DateAdd("d", 1, CXMLDate("@Data.EnrollmentDate~"))
```

and the CXMLDate function has been used to convert the data retrieved from the database. Here's another example that finds the difference, in days, between dates in two separate date-type columns,

```
=DateDiff("d", CXMLDate("@Data.OrderDate~"), CXMLDate("@Data.ShippedDate~"))
```

and, once again, the CXMLDate function is used to convert the date values for use with the intrinsicDateDiff() function.

DontResolveTokensInData Attributes

There may be times when you need to store text in a table column that includes Logi tokens. This might happen in documentation, for example. Under normal circumstances, when this text is retrieved into a datalayer, the Logi Server Engine will attempt to *resolve* any tokens in the text and if it does, the representation of the token will disappear from the text and be replaced with (usually) an empty space.

Token processing can be *suppressed* by manually adding a special attribute to the General element in the _Settings definition. This is done in Logi Studio by opening the _Settings definition, clicking the Source tab at the bottom of the Workspace panel, and manually edit the General element's source code. Add the special attribute, **DontResolveTokensInData**, set to *True*, to the General element: `<General DontResolveTokensInData="True" rdDebuggerStyle="DebuggerLinks" ... etc. />`

The Debugger Style or any other attributes in the source code should remain as you found them. Save your definition and tokens will now be left alone when they appear in your data.

Query String Parameter Reference

Logi products use a number of **reserved words** to identify **tokens** and special **query string parameters**. This topic presents the query string parameters available for use in Logi Info.

If you're looking for information about tokens, see "Token Reference" on page 588.

Query string parameters are useful for passing data between web applications and use the familiar syntax:

```
http://<URL>?<Parameter>=<Value> http://<URL>?<Parameter1>=<Value1>&<Parameter2>=<Value2>
```

In Logi reports, **@Request** tokens are used to retrieve the data POSTed by an HTML form and the values from parts of the **query string** used to call reports. The following special @Request token identifiers are also reserved words and have special meanings in Logi reports:

Parameter	Description
rdAcRefreshData	For use with the **Analysis Chart** element. Refreshes the chart's data from the data source while maintaining any chart customizations the user has made during this session. To refresh data, call report with this parameter set to *True*.
rdAcReset	For use with the **Analysis Chart** element. Users' settings are automatically maintained during the session; clear these settings by calling the report with the rdAcReset parameter set to *True*.
rdAgLoadSaved	For use with the **Analysis Grid** element. Users' runtime customizations of the grid can be saved *between* sessions and Logi Info developers can implement this using the process task Procedure.Save Analysis Grid. A saved Analysis Grid can be reloaded using the rdAgLoadSaved parameter: *rdAgLoadSaved=myFileName.xml*

Parameter	Description
rdAgRefreshData	For use with the **Analysis Grid** element. Refreshes the grid's data from the data source while maintaining any grid customizations the user has made during this session. To refresh data, call report with this parameter set to *True*.
rdAgReset	For use with the **Analysis Grid** element. Users' settings are automatically maintained during the session; clear these settings by calling the report with the rdAgReset parameter set to *True*.
rdAfReset	For use with the **Analysis Filter** element. Users' settings are automatically maintained during the session; clear these settings by calling the report with the rdAfReset parameter set to *True*.
rdDebugPdf	When set to *True*, runs the PDF report and returns the HTML rather than generating a PDF from the HTML.
rdDgLoadSaved	For use with the **Dimension Grid** element. Users' runtime customizations of the grid can be saved *between* sessions and Logi Info developers can implement this using the process task Procedure.Save Dimension Grid. A saved Dimension Grid can be reloaded using the rdDgLoadSaved parameter: *rdDgLoadSaved=myFileName.xml*
rdDgRefreshData	For use with the **Dimension Grid** element. Refreshes the grid's data from the data source while maintaining any grid customizations the user has made during this session. To refresh data, call report with this parameter set to *True*.
rdDgReset	For use with the **Dimension Grid** element. Users' settings are automatically maintained during the session; clear these settings by calling the report with the rdDgReset parameter set to *True*.

Parameter	Description
rdFormLogon	Set to *True* when using custom logon forms.
rdMobile	Set to *True* to indicate that the report definition specified in the rdReport parameter is a Mobile Reports definition.
rdOgLoadSaved	For use with the **OLAP Grid** element. Users' runtime customizations of the grid can be saved *between* sessions and Logi Info developers can implement this using the process task Procedure.SaveOlapGrid. A saved OLAP Grid can be reloaded using the rdAOLoadSaved parameter: *rdOgLoadSaved=myFileName.xml*
rdOgRefreshData	For use with the **OLAP Grid** element. Refreshes the grid's data from the data source while maintaining any grid customizations the user has made during this session. To refresh data, call report with this parameter set to *True*.
rdOgReset	For use with the **OLAP Grid** element. Users' settings are automatically maintained during the session; clear these settings by calling the report with the rdOgReset parameter set to *True*.
rdPaging	Specify either *Interactive*, *Printable* or *NoPaging*.
rdPassword	The password value used during login when using security.
rdProcess	Specifies the file name (without a file extension) of a process definition that contains a task to execute.
rdPrompt	When set to *True*, a page listing all @Request tokens referenced in the definition file is displayed. The

Parameter	Description
	user can display default values for each parameter by specifying rdPromptxxxxx at the prompt, where *xxxxx* is the name of the request parameter.
rdReport	Specifies a report definition to run.
rdReportFormat	Specify any of the following formats: *PDF*, *NativeExcel*, *NativeWord*, *CSV*, *HtmlExport*, *HtmlEmail*, *Excel*, *Word*, *XML*, or *GoogleSpreadsheet*
rdShowModes	Specifies ShowModes defined for display in the current report.
rdSort	Indicates the column and direction of a DataTable sort. The format of the query string variable is: `rdSort=<dataTableID>~<columnName>~<columnDataType>~<Ascending|Descending>~`
rdTaskID	Specifies a Task ID to execute.
rdTaskLogFilename	Returns the name of the logfile created by Task Manager for a specific job. This token is used within the Process definition that creates the job.
<TabsElementID>	When a page containing a **Tabs** element is submitted, a request variable is automatically generated named for the ID of the Tabs element and assigned the value of the ID of the currently selected Tab Panel. Example: *@Request.MyTab~ = "Tab1"*
rdTemplate	Specify the template definition to run.
rdUserName	The User Name value used during login when using security.

SQL Queries and Comma-Delimited Lists

Developers may encounter a need to pass in, as a single parameter, a **comma-delimited list** of values to a SQL query or stored procedure. Unfortunately, SQL parameters don't have a way to represent comma-delimited lists. Here are two solutions, one for MS SQL Server and the other for Oracle, that solve this problem, using a sub-query and the IN search qualifier respectively. The following query is for MS SQL Server, where @sel_customers is the parameter with the delimited list of values:

```
-- my stored procedure

@sel_customer varchar(200) -- this is the parameter

SELECT SalesOrderID, OrderDate, CustomerID
FROM AdvWorks.Sales.SalesOrderHeader
WHERE CustomerID IN
(SELECT CustomerID FROM AdvWorks.Sales.Customer WHERE CHARINDEX(STR(CustomerID), @sel_customers) > 0)
```

With this technique, the SQL statement does not vary between executions and will therefore be cached. By putting the expression into a sub-query, the DBMS optimizer will only execute it once (because it's not correlated). Since the sub-query just accesses a lookup table, the speed will be very fast as opposed to evaluating the expression for each row of the primary query table. The example above shows a query in a stored procedure, but the same query could be used by itself, replacing the parameter with an @Request token. Here's a similar query for use with an Oracle DB:

```
SELECT ACTIVITY_ID, CODE_VALUE, CATEGORY_ID,
FROM SYSADM.ACTIVITY_TYPE
WHERE CATEGORY_ID IN
(SELECT CATEGORY_ID
FROM SYSADM.ACTIVITY_CATEGORY
WHERE instr(:@sel_ActivityCategories, '''' || to_char(CATEGORY_ID) || '''') > 0 )
```

This sub-query is slightly more complex because quotes are needed around the search argument (CATEGORY_ID) in order to prevent partial key matches (e.g. the key value '1' is part of the key '123'). This isn't an issue in MS SQL Server because Microsoft conveniently adds a space at the beginning of the number for sign, therefore eliminating partial key matches.

PostgreSQL v8 Object Case Sensivity

Here's an interesting fact about **PostgreSQL** v.8 schemas: for some Catalog normalization requirements within the PostgreSQL DB engine, all object names should be referenced in *lower-case* from Logi applications, including database, table, and column names. For instance:

```
SELECT productname, productid FROM products
```

Under normal circumstances, the query shown above is valid,

```
SELECT ProductName, ProductId FROM Products
```

but the second one is not.
However, it is possible to define objects that have mixed- or upper-case names. If you encounter a DB where this is the case, then you may use mixed- or upper-case names in your queries, *if* you wrap object names with double-quotes:

```
SELECT "ProductName", "ProductId" FROM "Products"
```

as shown above. Perhaps the most noticeable impact will be with the functioning of Logi Studio's **Query Builder** and **Database Browser** tools. Neither tool will wrap object names in double-quotes and will therefore fail to work if you have any upper-case characters within the table, view, or database list.

Call Logi Engine Functions with JavaScript

Ever need to call a process task or make an AJAX request after running a custom JavaScript validation method? Since you can't have sibling or nested Action elements (except for bookmarks), developers must call Logi Server Engine functions directly.

The syntax of these functions may change depending on Logi Info version. You can always check their syntax by looking for them in JavaScript files in the `<yourLogiApp>\rdTemplate` and `\rdTemplate\rdAjax`folders. To use these scripts, add an **Action.Javascript** element (or **Action.Link** and **Target.Link** elements) beneath a **Label**, **Image**, or **Button** element. Enter your Javascript code in the Javascript attribute (or Target.Link element's **URL** attribute, beginning with "javascript:",without the quotes).

To submit a Logi report page and/or navigate to another report

Use: SubmitForm(sPage, sTarget, bValidate, sConfirm, bFromOnClick, waitCfg) Where:
sPage = URL of the Logi definition to load next, with any request params
sTarget = the target window: *_blank*, *_self*, *_parent*, *_top*, *<frameName>*
bValidate = validate input elements before submit: *true*, *false*
sConfirm = show this string as a confirmation prompt, if blank do not show anything
bFromOnClick = decode event: *true*, *false*
waitCfg = show Wait Panel: *true, false*
Code the following script:

```
SubmitForm('rdPage.aspx?rdReport=yourReportDef&FirstTime=True', '_parent', 'false', '', 'false',
'true');
```

To submit a report page and/or execute an AJAX call (no Request Forwarding)

Use: rdAjaxRequest(commandParams, bValidate, sConfirm, bProcess, fnCallback, waitCfg, isUpload) Where:

commandParams = URL of the Logi definition to load next, with any request params

bValidate = validate input elements before submit: *true*, *false*

sConfirm = show this string as a confirmation prompt, if blank do not show anything

bProcess = definition to load next is a Process definition: *true*, *false*

fnCallback = JavaScript function to run after this function completes

waitCfg = show Wait Panel: *true, false*

isUpload = this is a file upload operation (uses POST and set other upload-related features): *true*, *false*

Code the following script:

```
rdAjaxRequest('rdPage.aspx?rdReport=yourReportDef&FirstTime=True', 'true', 'Submit this page?', 'false',
null, 'false', 'false')
Using Callback function example:
```

```
rdAjaxRequest('rdPage.aspx?rdReport=yourReportDef&FirstTime=True', 'true', 'Submit this page?', 'false',
myCallbackFunction(), 'false', 'false')
Example callback function in Include Script element:
```

```
function myCallbackFunction() { alert('this is an example'); }
```

To submit a report page and/or execute an AJAX call (with Request Forwarding):

Use: rdAjaxRequestWithFormVars(commandURL, bValidate, sConfirm, bFromOnClick, bProcess, fnCallback, waitCfg, copyQueryString) Where:

commandURL = URL of the Logi definition to load next, with any request params

bValidate = validate input elements before submit: *true*, *false*

sConfirm = show this string as a confirmation prompt, if blank do not show anything

bFromOnClick = decode event: *true*, *false*

bProcess = definition to load next is a Process definition: *true*, *false*

fnCallback = JavaScript function to run after this function completes

waitCfg = show Wait Panel: *true, false*

copyQueryString = append query string to end of commandURL: *true*, *false*

Code the following script:

```
rdAjaxRequestWithFormVars('rdProcess.aspx?rdReport=yourProcessDef', 'true', 'Submit this page?', 'true', 'true', null, 'false', 'true') See previous example for callback function usage.
```

Conditions

The **Condition** attribute allows a variety of elements to be rendered or hidden, and can control other behavior dynamically, based on evaluation of an expression.

The following topics show developers how to implement the conditions attribute:

- Elements Using the Condition Attribute
- Conditions vs. Show Modes
- Evaluating the Condition Expression
- Example: Setting Session Variables
- Example: Using the Division Element
- Example: Using the Data Table Column Element
- Example: Using the Conditional Class Element
- Example: Using the Procedure.If Element

Elements Using the Condition Attribute

The following elements have a **Condition** attribute. In the first group, the Condition attribute simply determines whether or not the element and its child elements are *rendered* on the page:

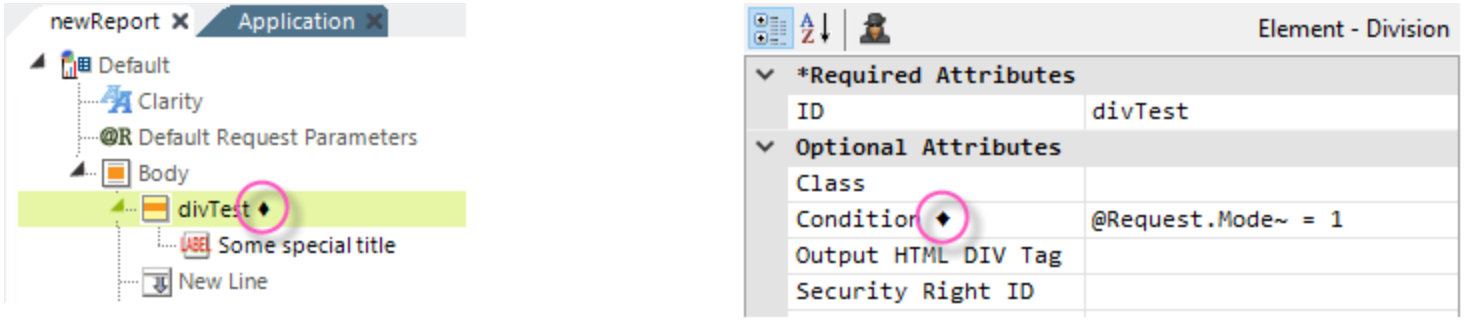
- Column Cell
- Crosstab Table Label Column
- Data Table Column
- Division
- Local Data
- More Info Row
- More Info Row Column
- Include Frame (also called Sub-Report)
- Row
- Security Filter
- Set Session Variables
- Tooltip Panel

These two elements use the **Condition** attribute slightly differently to control other behaviors:

- Conditional Class - the Condition attribute determines whether the element's style sheet class is applied to all of its child elements that do not have a class specified.
- Condition Filter - the Condition attribute determines which data layer result rows are kept and which are filtered out.

Many elements also have an **Include Condition** attribute, which determines if the element is included in processing. Typically, elements with this attribute are filter and column elements used to manipulate data.

Elements that have a value in their **Condition** or **Include Condition** attributes are now indicated by a diamond icon:



The diamond icon, shown circled above, appears in both the Element Tree (left) and in the Attributes Panel.

Conditions vs. Show Modes

It may appear that the **Show Modes** and the **Condition** attributes do the same thing: show and hide elements. However, Show Modes processing occurs in the *client* (the browser), whereas Conditions are processed in the *server.* This means Show Modes are more browser-dependent and may not always behave exactly as expected with different browsers, making Conditions a better choice in many situations.

Whereas Show Modes simply hide page parts in the browser, Condition actually keeps them from being generated at all. For example, a Data Table column hidden using a Condition never has its data rendered; it's never sent by the server to the browser. However, if Show Modes are used for the same purpose, then the data column *is* rendered and *is* sent to the browser, where it's then hidden (but might be seen by viewing the page's HTML source code). So, using the Condition in this case is more secure, in that it ensures that data is not sent to the browser at all.

However, Show Modes do not require a roundtrip to the server to make dynamic changes, as Conditions do, so there may be performance implications involved in deciding which mechanism to use.

Condition attributes may also have a performance impact: If an element isn't rendered then, obviously, its child elements aren't rendered either. Child datalayers that aren't rendered, won't run to retrieve data. So Conditions can be used in some cases to determine when, or if, datalayers run.

When exporting, Show Modes include built-in values that make hiding or showing elements very easy, so they may be a better choice for that purpose.

For more information about Show Modes, see *Show Modes*.

Evaluating the Condition Expression

In all instances, the Condition attribute's value is an *expression* that evaluates to either *True* or *False*. Leaving the attribute value blank equates to setting it to *True*. Expressions should be in JavaScript or intrinsic function syntax.

Tokens of all kinds may be included in expressions.

Expressions used in Conditions *do not* require the leading "=" sign which is typically used before formulae in other types of attributes. Values are generally compared using a comparison operator and, when expressions use string or ambiguous data types, both sides of the equation should be enclosed in double-quotes. More complex expressions can be created using logical operators and parentheses.

Here are some example expressions:

Expression	Comment
"@Data.LastName~" = "Smith"	Both sides of the equation are known to be strings.
@Data.CustomerID~ <> 1234	Both sides of the equation are known to be numbers.
1 = 1	This expression can be used to force a *True* value.
("@Request.Page~" = "") Or ("@Request.Page~" = "1")	Logical operators can be used.
InStr("@Data.Categories~", "1") > 0	Script functions are supported.

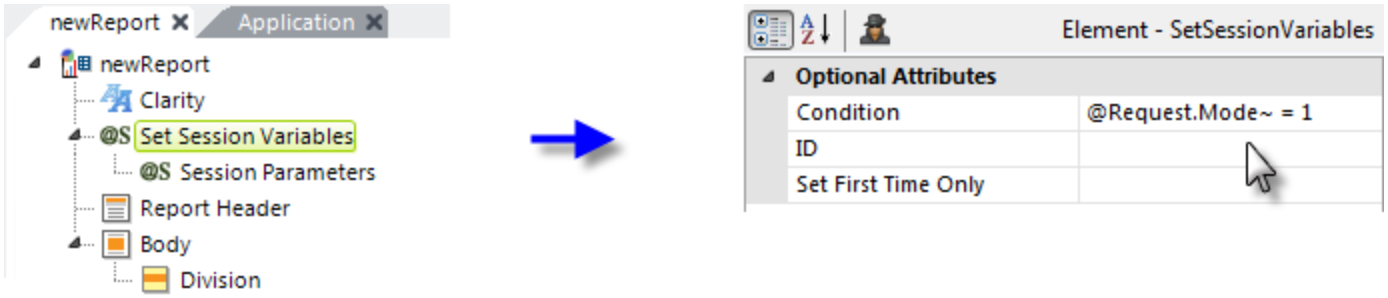
Expression	Comment
"@Cookie.UserCategory~" = "1"	Cookies are evaluated as **text**.
"@Data.City~" = "Toronto"	When used with the Condition Filter element, causes all datalayer rows where the City column value is not "Toronto" to be removed.

Complete information about "Built in Functions and Operators" on page 632is available for your reference.

Generally, if the expression contains an error of some kind, such as a data type mismatch or a missing function argument, no error message is displayed in the browser; instead, the Condition just doesn't work as desired. However, an error message *will* usually be included in the Debugger Trace Page (see "Debug Reports" on page 56) at the point at which the expression is evaluated. So, if your Condition isn't working as you expected, check the Debugger page.

Conditions Example: Setting Session Variables

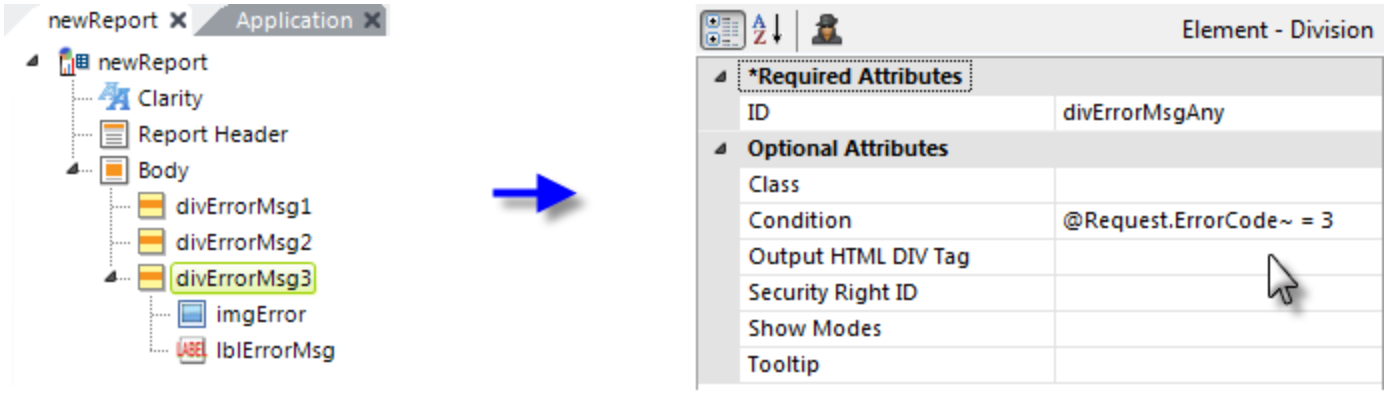
The ability to conditionally set Session variables is very useful. These variables with global scope provide a great way to share small amounts of data across report definitions. For more information, see Session Variables.



As shown in the example above, the **Set Session Variables** element will run and create Session variables *only* if the formula in its **Condition** attribute is true.

Conditions Example: Using the Division Element

A **Division** element is very useful as a *container* for other elements and, through use of its Condition attribute, allows you to selectively control whether it and its child elements, possibly entire areas of application pages, are rendered.

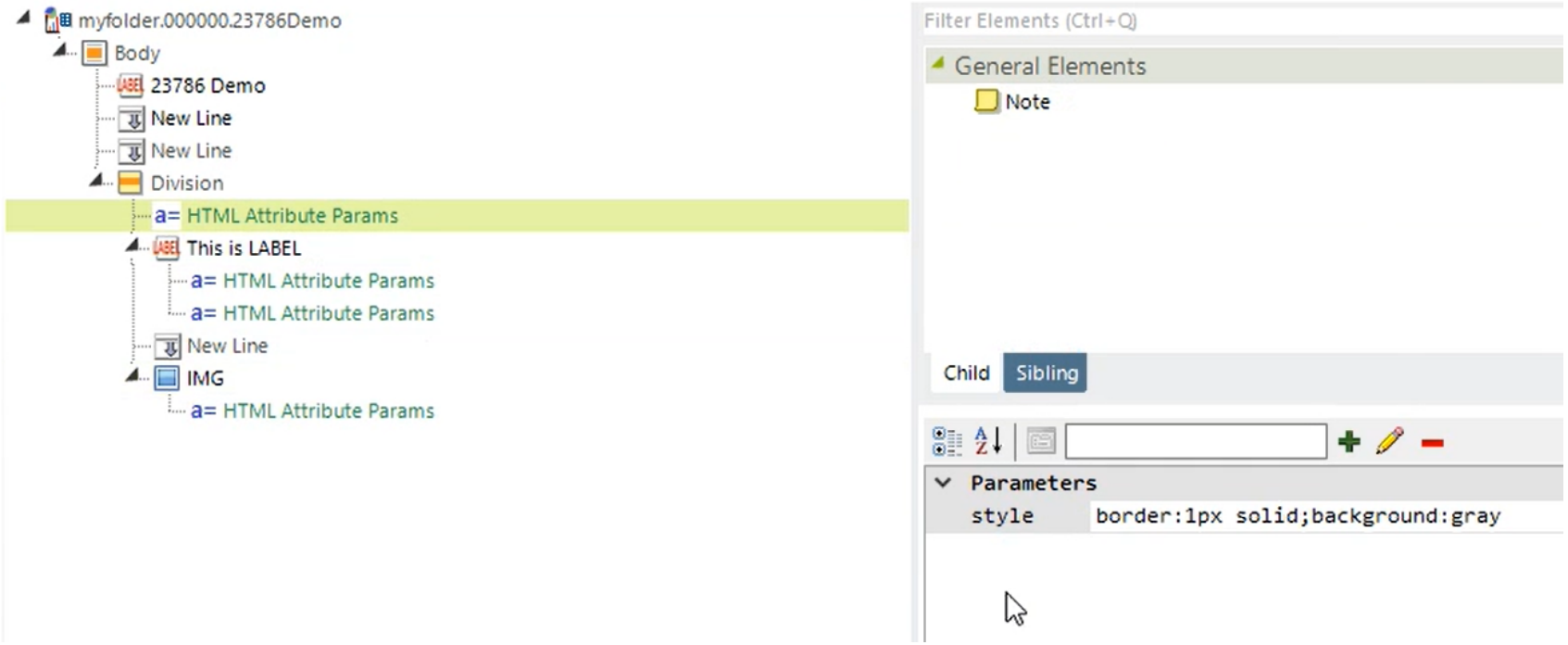


In the example shown above, three **Division** elements have been used to contain responses to specific potential errors.

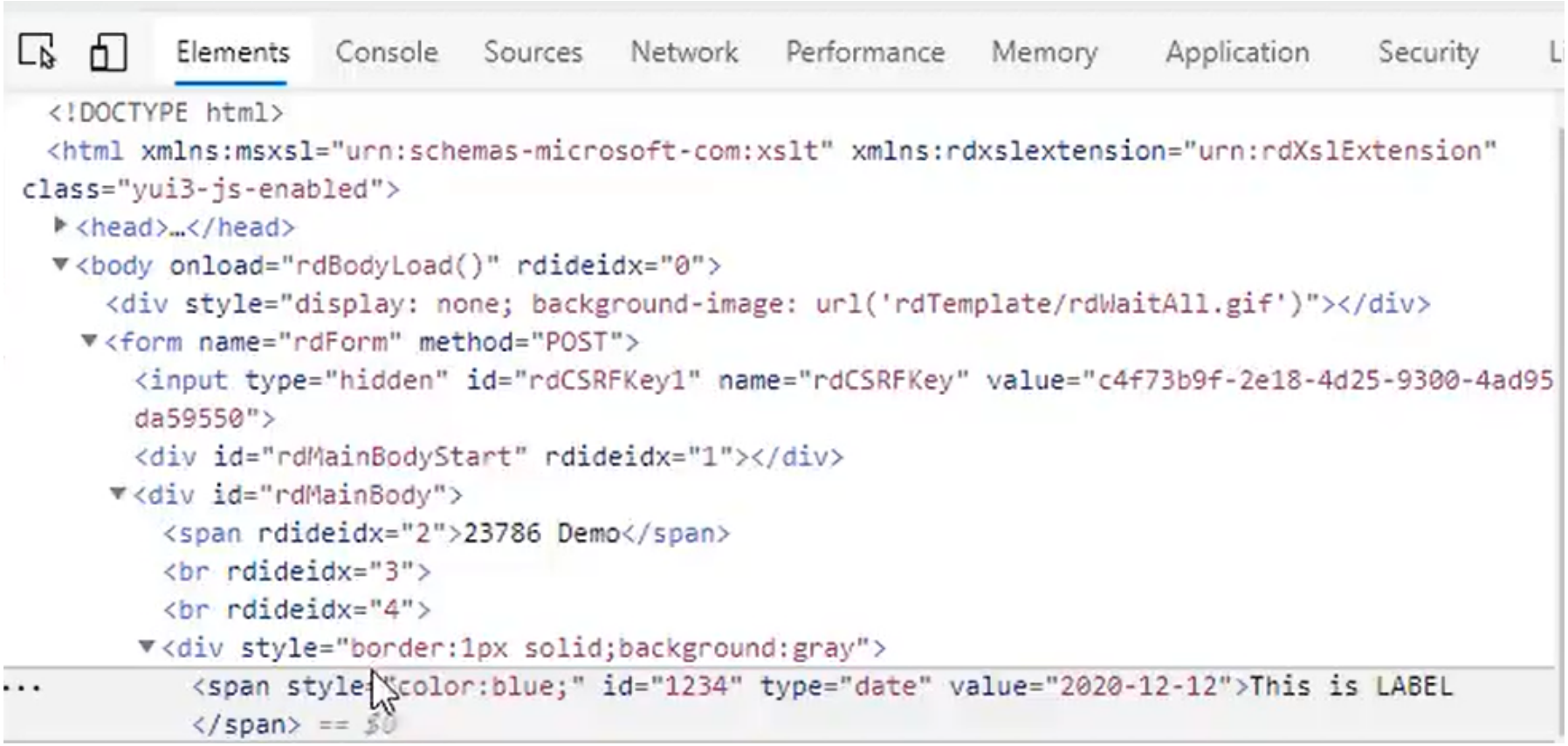
If the Request variable and value ;"ErrorCode=3" is passed to this page when it's called, then the "divErrorMsg3" element will be displayed, along with its child elements, causing an error message to appear on the page. The other two Divisions, which have a similar expression in their Condition expressions but with a comparison to values *1* and *2*, will not be displayed.

As mentioned earlier, the effect of displaying or hiding a Division element extends beyond just visibility; for example, datalayers within divisions *will not run* if the division is not displayed.

Division elements include "HTML Attribute Params", enabling you to apply your application to work with other frameworks or libraries easier. With the HTML Attribute Params, you have the option to include "style" parameters:



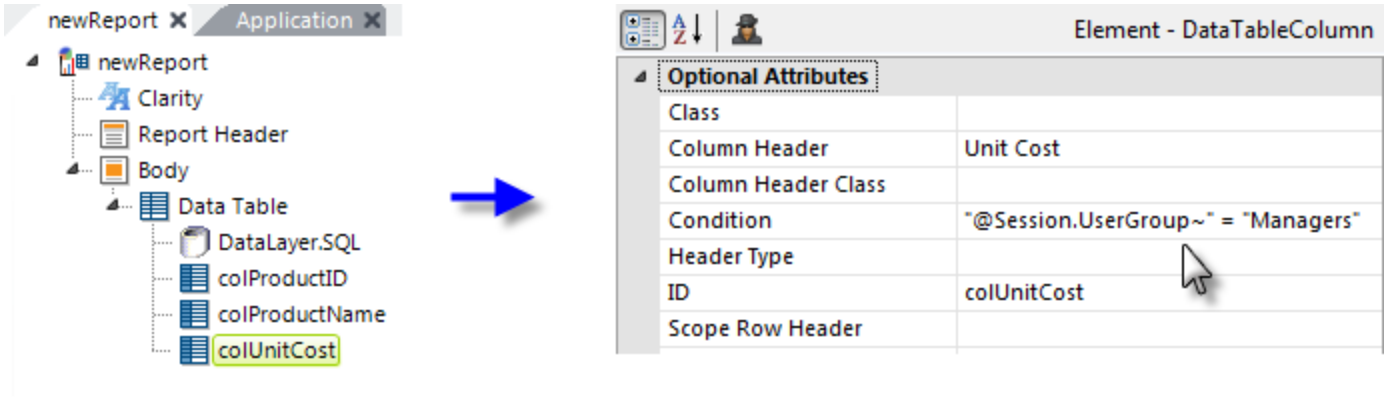
After the HTML Attribute Params are applied:



If an attribute was set in both Element and HTML attribute params, the one set in the HTML attribute params will be ignored.

Conditions Example: Using the Data Table Column Element

A **Data Table Column** element's Condition attribute allows the column to be displayed or hidden dynamically, based on criteria you set.



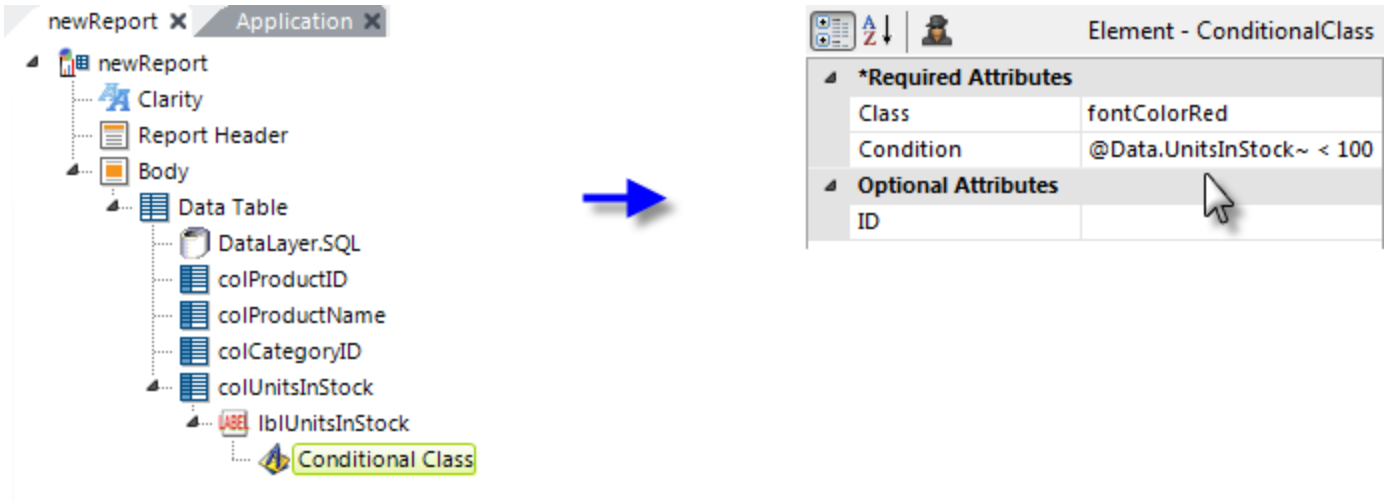
The example report definition shown above displays Product information in a Data Table. However, the display of *sensitive* data, such as the unit cost, needs to be restricted so that it can only be viewed by specific users. If the application starts by authenticating the user and classifying them using a Session variable before calling this report, sensitive data can be hidden, using a Condition attribute, from everyone except authorized users. The Unit Cost column will not be displayed at all unless the Condition evaluates to *True*.

Conditions Example: Using the Conditional Class Element

The **Conditional Class** element allows you to dynamically apply different style sheet **classes**, based on the Condition attribute.

Product ID	ProductName	CategoryID	Units In Stock
1	Chai	1	240
2	Chang	1	185
3	Aniseed Syrup	2	67
4	Chef Antons Cajun Seasoning	2	110
5	Chef Antons Gumbo Mix	2	143
6	Grandmas Boysenberry Spread	2	96
7	Uncle Bobs Organic Dried Pears	7	115

Suppose product managers want to be *visually alerted* if product stock levels fall below 100 units. In fact, when an item falls below that threshold, they'd like to see its "in stock" count displayed in the table in red, as in the example above.



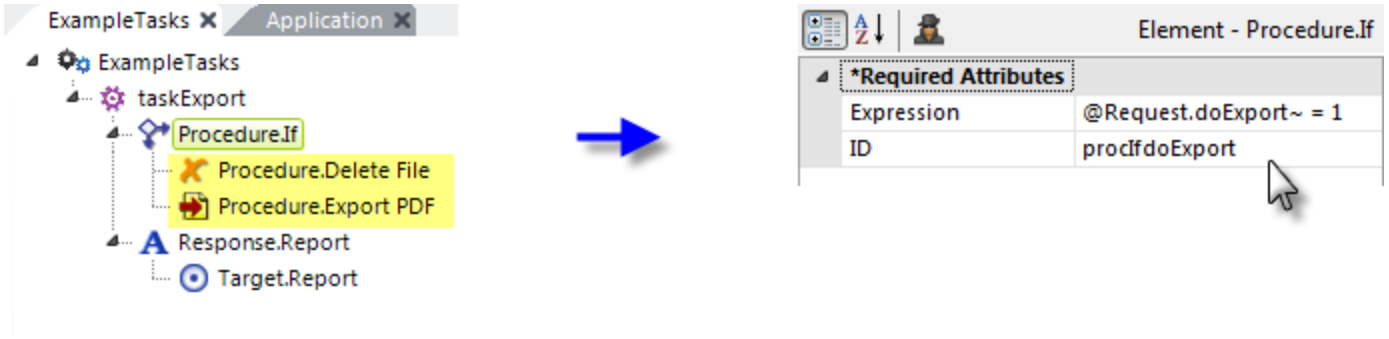
In the example shown above, the **Label** in the last Data Table column displays the number of units in stock. Its font color may be explicitly set in its own Class attribute, or it may be inherited from a parent element or a theme.

To give the product managers what they want, a **Conditional Class** element is added beneath the Label, and its attributes set as shown above. The expression in the Conditional Class element's **Condition** attribute sets the threshold and its **Class** attribute specifies which class to apply if the Condition expression evaluates to *True*.

Multiple Conditional Class elements can be used beneath a parent element (the maximum is nine Conditional Class elements). In this case, the class from the *first* one of these elements that has a Condition attribute that evaluates to *True* will be applied; any remaining Conditional Class elements below it will *not* be evaluated.

Conditions Example: Using the Procedure.If Element

Tasks within Process Definitions can make use of the **Procedure.If** element to provide conditional processing. This element functions like a typical IF-THEN statement in procedural programming:



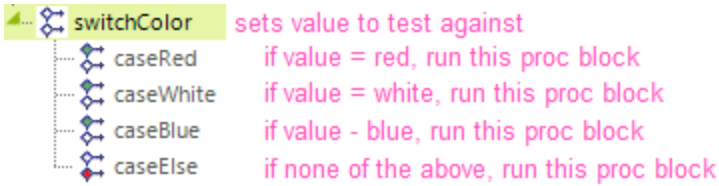
In the process definition example above, the "taskExport" task includes a **Procedure.If** element which has a conditional **Expression** attribute. If this expression evaluates to *True*, then all of the Procedure.If element's child elements will execute. If the expression is *False*, then processing will continue with the **next element** following the Procedure.If element's child elements. So, as shown above, if the "doExport" Request parameter equals *1*, then the yellow-highlighted elements will execute; otherwise they'll be skipped.

Several other elements are available for use in controlling conditional branching:

Procedure.Else - This element *must* be a sibling of, and immediately follow, a Procedure.If element. It identifies a conditional block of procedures that will be run if the Procedure.If element immediately above it evaluates to *False*. If the Else block ran, the token `@Procedure.myProcedureID.rdReturnValue~`returns *True,* otherwise it returns *False*.

Procedure.Switch - This element works with one or more child **Procedure.Switch Case** elements to define conditional blocks of procedures to be run when a specified value matches. Use it when there are multiple conditional blocks defined and just one of

them is to be run, based on the value of a variable. You can also use a child **Procedure.Switch Else** element, which runs its block of procedures if *none* of the previous Switch Case elements match the specified value.



The value to test against is specified in the Expression attribute and can be literal value, a token, or even JavaScript that evaluates to a value. The Data Type attribute ensures comparisons, especially of dates, are made correctly.

Conditions give developers a useful tool, providing the ability to dynamically change their reports and allowing great flexibility when developing Logi applications.

Scripting

Logi Info lets you take advantage of scripting objects and functions in your Logi applications. These include inline intrinsic functions, JavaScript code embedded in your definitions, and code in external script files.

The following topics describe the Logi elements available for working with scripts in your applications:

About Scripting

Developers using Logi products have several different scripting language options available to them:

Script Type	Description
Intrinsic	These built-in script functions are available for use in expressions and embedded script. For more information, see "Built in Functions and Operators" on page 632.
JavaScript	This is the scripting language default, using standard JavaScript syntax.
VBScript	*VB Script has been deprecated - use JavaScript instead for all new applications*.

Where Does Scripting Execute?

It's important to understand *where* scripts execute in order to get them to work correctly.

In the Browser or "Client-Side"

Scripts that execute in the browser, or "client-side", recognize the browser's Document Object Model (DOM) with its Window and Document objects and DHTML events, such as *OnClick* and *OnChange*, which occur in the browser. Client-side scripts are useful for calculations, input validation, dynamic page changes, and other activities that do not require an exchange with the web server. This site documents the DOM objects and syntax.

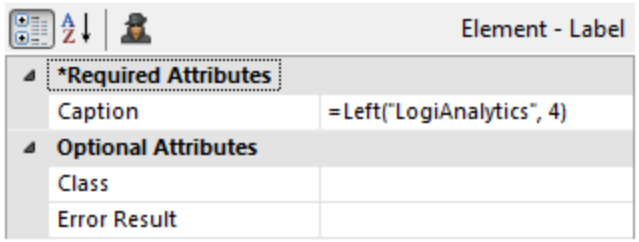
 All browsers may not recognize all objects, events, and functions in exactly the same way, so cross-browser testing is *highly recommended* if your users will use a variety browsers and/or browser versions.

On the Server or "Server-Side"

Scripts that execute on the web server, or "server-side", are usually very efficient and secure, but they can't access browser DOM objects or react to user interface events. They *may* be able to access server system resources, such as files, using special objects and third-party APIs, though this is often considered a security risk.

Inline Scripting with "Formula" Attributes

Some element attribute values, such as the **Label** element's **Caption** attribute and the **Calculated Column** element's **Formula** attribute, are categorized as "formula" attributes. This means they can evaluate inline script expressions and use the results as their values. In attributes where what to do might be ambiguous - display the text or evaluate it - the use of a leading equals sign (=) is used to indicate that the text should be evaluated. Typically, the functions called in these expressions are the standard intrinsic functions and run *at the server*.



For example, consider the Label element attributes shown above. The **Caption** attribute contains an expression that starts with an equals sign and calls the intrinsic Left() function in an expression, and will result in the word "Logi" appearing on the report page. This is an easy way to work with individual functions.

Later, we'll see how to include external files with custom functions that can be called in formula attributes.

Error Handling

Elements that support inline scripting also have an **Error Result** attribute. If something is specified in this attribute, it will be displayed or used instead of the expression result if an error occurs.

The default Error Result varies depending on the element. For example, for the Condition Filter, the default is *False*. For the Calculated Column and Extra Crosstab Calculated Column elements, the default is *blank*. For other elements, the default is *???*.

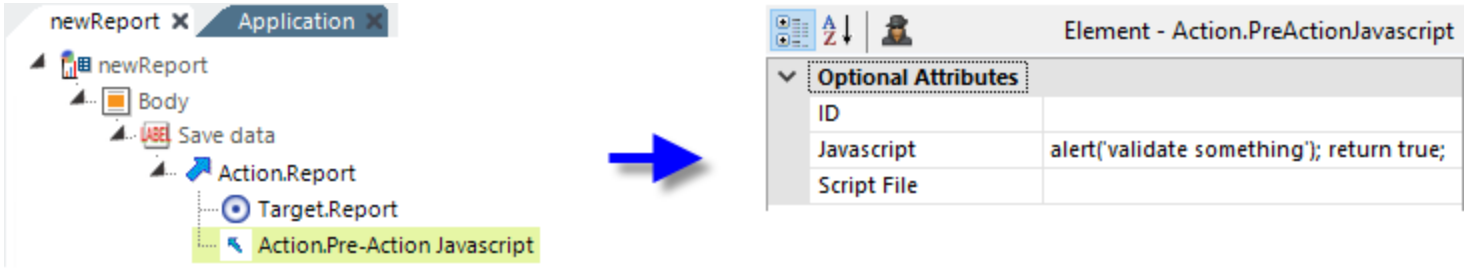
Two special constants are available to control error handling related to expressions that have the possibility of causing a divide-by-zero situation. If needed, specify these constants in the _Settings definition:

`rdInfinityErrorResult` - Specifies a global result for expressions that may produce a divide-by-zero situation (result: infinity), sparing developers the task of specifying a result on a per-element basis in their Error Result attributes.

`rdJavascriptInfinityIsError` - When set to *True*, specifies that expressions that produce a divide-by-zero situation (result: infinity) will trigger an error, which is then handled by elements' Error Result attribute or, if set, the rdInfinityErrorResult constant. If rdJavascriptInfinityIsError is not specified, the Logi Server Engine will return the value *NaN* (for "Not a Number") by default when a division-by-zero attempt occurs.

Scripting with Action.Pre-Action JavaScript

The **Action.Pre-Action Javascript** element allows you to run code after a link, button, etc. has been clicked but before its Action element executes. This makes it much easier to do specialized input validation or to perform custom UI changes before the page is submitted.



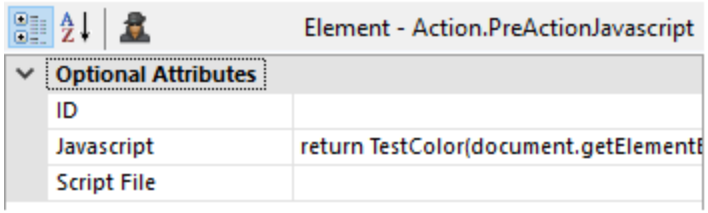
As shown above, Action.Pre-Action Javascript is available as a child element of most other Action elements. If the JavaScript it runs returns any value other than *false*, then the parent Action element will execute. If it returns *false*, the parent Action element will *not* execute. You may use multiple Action.Pre-Action Javascript elements beneath a parent Action element.

The JavaScript code can be inline, as shown above, or it can be a function in a separate script file, or a function included using an **Include Script** element. Here's a simple example of the latter:

```
function TestColor(inpColor) {
if ( inpColor == 'red') {
return true;
}
else {
alert('You must enter red');
```

```
return false;
 }
}
```

The code shown above is placed in an Include Script element in the report definition.



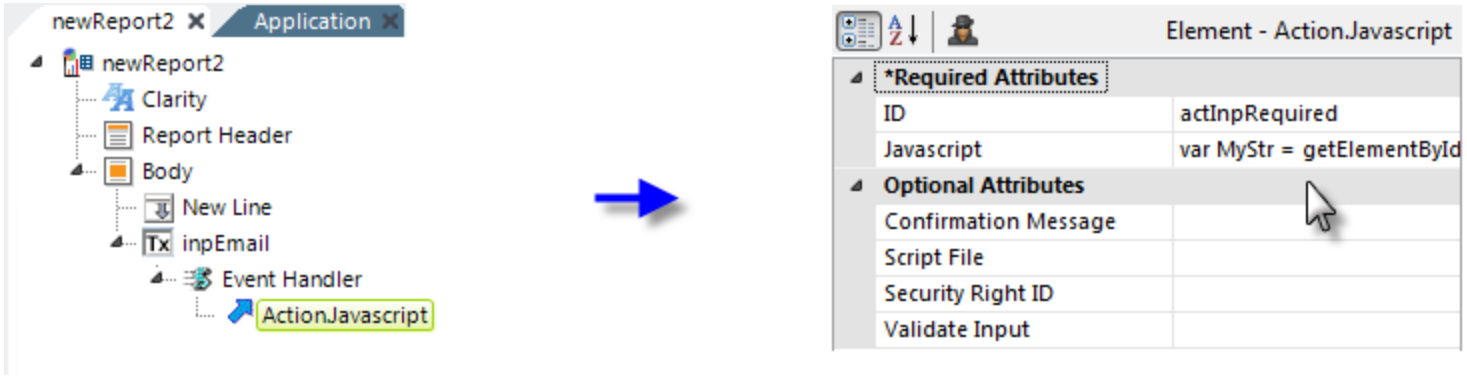
The Action.Pre-Action Javascript element code, shown above, is:

```
return TestColor(document.getElementById('inpColor').value);
```

If the text "red" is entered in an Input Text element on the page with an ID of "inpColor" (not shown) and the "Save data" link is clicked then the next report *will* be shown. If any other text is entered and the link is clicked, a warning message will appear and navigation to the next report *will not* occur.

Scripting with Action.Javascript

Scripting can be run using the **Action.Javascript** element and it will run *in the browser*.



The example above shows how you might validate data entry, using the DHMTL *onBlur* event, which fires when the cursor exits the input control. You can enter multi-line JavaScript code directly into the Action element's **Javascript** attribute value. For example, we might use this code to ensure that an email address has been entered:

```
var mStr = document.getElementById('inpEmail').value;
if ( mStr.indexOf('@') == -1 ) {
 alert('Invalid email format');
 }
```

The **AttributeZoom** window comes in handy when entering code (double-click the attribute name).

Support for JavaScript "this" Keyword

The JavaScript "this" keyword is supported. When working with Event Handlers in Logi apps, this generally means that instead of code like `document.getElementById('inpEmail').value` you can use `this.value` instead. This is nice bit of shorthand for referring to the "owner" of the event and saves a lot of keystrokes. Here are some other JavaScript validation examples:

Example 1: Ensure some data is entered

```
var myStr = document.getElementById('inpLoginID').value;
if (myStr.length == 0 ) {
 alert('You must enter some message text.');
}
```

It's beyond the scope of this topic to delve too deeply into JavaScript or the object model, but essentially the code above assigns the data in inpLoginID to a variable, then tests the length of that variable's value, and displays a message box if the length is zero. This is very similar to the **Validation.Required** element's functionality. The Document object's getElementById method is very handy in this situation (remember: JavaScript is case-sensitive).

Example 2: Ensure some data is entered but not more than 4,000 characters

```
var myStr = document.getElementById('inpTextArea').value;
if (myStr.length > 4000) {
 alert('Your message is too long (' + myStr.length + ' chars) - maximum length is 4,000 characters.');
}
if (myStr.length == 0 ) {
 alert('You must enter some message text.');
}
```

The example script above does something special: it *limits the length*of data that can be entered into an **Input Text Area** element, which doesn't have a Maximum Length attribute, and, if it's too long, reports the actual number of characters currently entered. It also requires that *some*data be entered.

Example 3: Replace all occurrences of '@' with '#'

```
var myStr = document.getElementById('inpTextArea').value;
myStr = myStr.replace(/[@]/g, '#');
document.forms[0].inpTextArea.value = myStr;
```

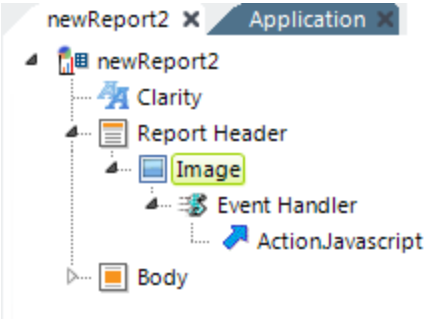
The example code above demonstrates a method for replacing certain characters in the data entered, using a JavaScript *regular expression*. In the example, any @ characters that were entered by the user are replaced with the # character, and then the altered data is substituted for the data that was entered. Regular expressions are very powerful, can be quite complex, and can

perform actions such as adding, replacing, or removing characters, changing character case, and expanding or contracting phrases.

As you can see, the ability to run JavaScript whenever data changes in user input elements provides developers with a very powerful validation tool.

"On Load" Scripting

Developers often want scripting functions to run as soon as the report page is loaded into the browser. Use-case examples include redirecting users who are not logged-in and setting the focus to a specific user input control. Let's see how this can be done in a Logi application.



In the example shown above, an **Image** element has been added to the report header. The image is a 1-pixel by 1-pixel white dot that blends into the header background and is effectively invisible. When the image is loaded, the **Event Handler** element, set for the *OnLoad* event, will run the **Action.Javascript** element. This "invisible image" technique is very common and reliable.

```
if ( '@Function.Username~' == '' ) {
 window.location = '@Constant.myLogiAppURL~/rdPage.aspx?rdReport=Login';
}
```

The code shown above is used in the Action.Javascript element to test the token that, when using Logi Security, contains the user name if a successful login occurred. If the token is empty, then the browser is redirected to the login page.

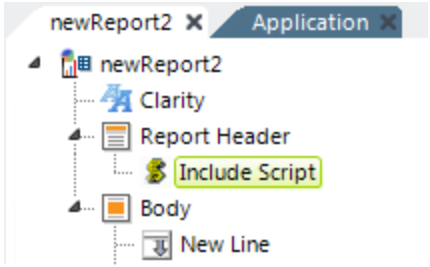
Using JavaScript and the DOM, you can access any of the components on the page. For example, to set the focus to a specific Input control when the page loads, you would add

```
document.getElementById('myTextInputID').focus();
```

to the code so that it runs if the @Function token has a value.

Inserting Code Directly

The **Include Script** element can be used to insert script, entered into one of its attributes, directly into the HTML output when a page is generated. Script inserted in this way runs in the browser.



The example above shows the Include Script element as a child of the **Report Header** but it can be the child of the Body, Page Footer, and about 20 other elements. The inserted script will appear in the HTML code between < SCRIPT> tags (do *not* include these tags in the Include Script element's **Included Script** attribute value).

You can also use Include Script to insert tags, such as HTML < META> tags, into the page's < HEAD> section. For more information, see *Insert HTML into Reports*.

Scripting within External Files or Third-Party Libraries

In "Inserting Code Directly " on the previous page we saw how script can be inserted directly into the HTML page. However, you may want to create and use your own custom functions that are stored in a external file, or use third-party libraries, such as JQuery. This can be done by including the external files using these elements:

Element	Runs At	Description
Formula Script File	Server	Used in Report definition, specifies custom or third-party script library.
Include Script File	Browser	Used in Report definition, specifies custom or third-party script library.
Procedure.Script	Server	Used in Process definition, specifies custom or third-party script library.
Additional Script File	Server	Optional child of the other three elements, specifies additional external files that contain supporting functions.

One advantage of using custom external libraries is that they can be *shared* and *re-used* in different applications. Functions in these files can be directly called from a formula attribute.

Script files can be created and edited in Studio. JavaScript and the deprecated VBScript files are listed as options when you right-click the _SupportFiles folder in Studio's Application panel and select Add → New File...

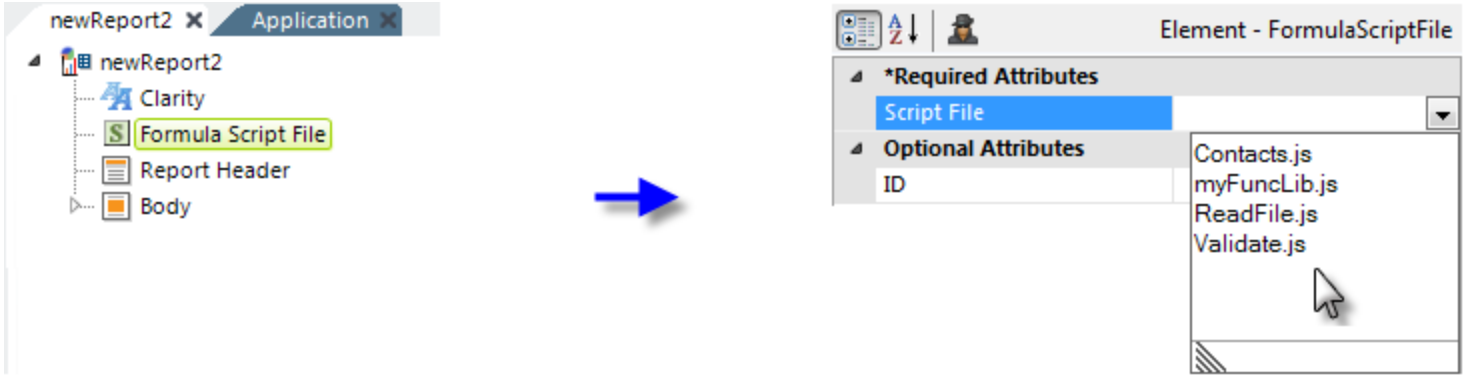
Scripting in files included using the elements listed above does not require < SCRIPT> tags; they're added automatically.

```
/* this is MyFunctionLib.js */
function sayHello(UserName) { // user name comes from his login
return 'Hello ' + UserName + ' from Logi Analytics';
}
```

Script files used with Logi applications are "regular" in their syntax and form. Everything that's normally available in the scripting language, such as comments, can be used, as shown above.

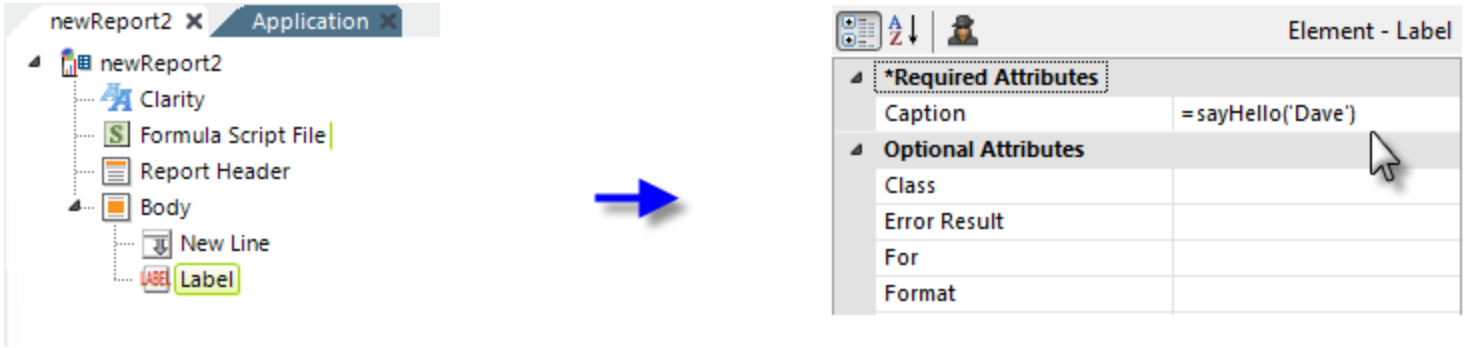
Including Script Files

In order to include a script file in your Logi application, you need to configure one of the elements mentioned earlier. Here's an example:



1. Add a **Formula Script File** element beneath the root element, as shown above.
2. Its **Script File** attribute value is set to the name of the script file. If the file is in the _SupportFiles folder in Logi Studio's Application Panel, it will appear in a list of available files, as shown above. Otherwise, a fully-qualified file path can be entered.

Once this has been configured, the functions in the script file are available for use within the Report definition.



The example above shows how to call a custom function in your formula script file. It's done in exactly the same way as an intrinsic function: in a expression preceded by an equals sign (=). Once a Formula Script File element has been added, the functions in the script file it identifies are *available directly* to all "formula" attributes within the Report definition.

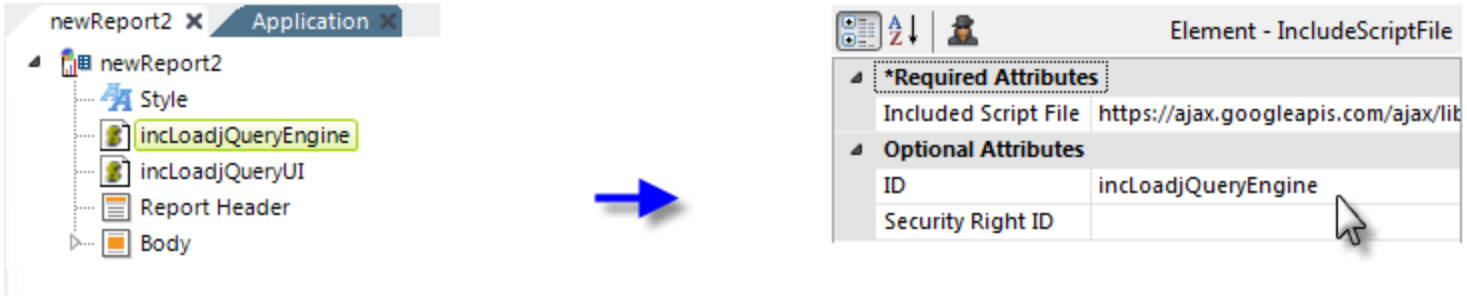
Working with jQuery

jQuery is a cross-browser JavaScript library designed to simplify client-side scripting and special elements are available to make it easy for developers to work with this library. Here's a quick example that uses the jQuery "date picker" UI component:

First, we configure our report definition's **Style** element to use a jQuery style sheet hosted online by Google:

```
http://ajax.googleapis.com/ajax/libs/jqueryui/1.11.4/jquery-ui.min.js
```

The jQuery versions used here are representative and other versions will work as well.

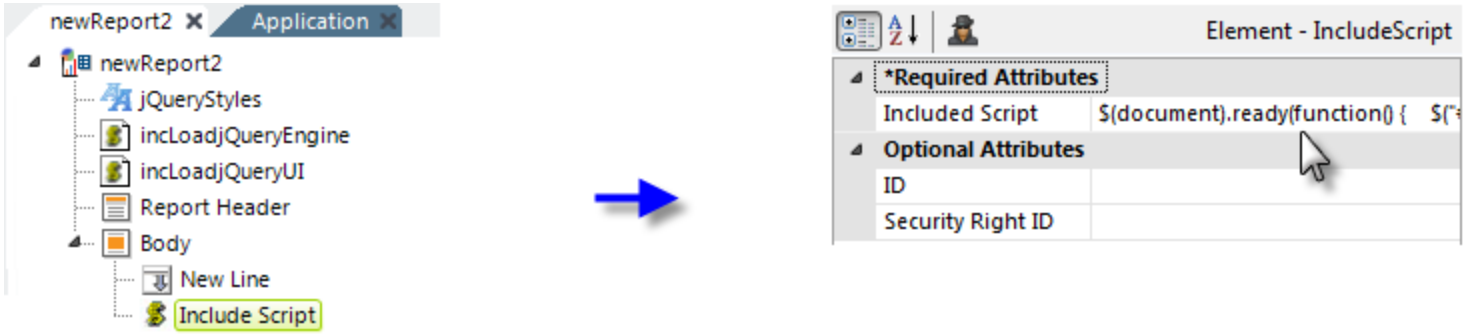


Then we add two **Include Script File** elements to our definition, as shown above, and configure their attributes to include the jQuery libraries hosted online:

```
http://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js
http://ajax.googleapis.com/ajax/libs/jqueryui/1.11.4/jquery-ui.min.js
```

The Google API hosting page suggests using these URLs with the "https:" protocol. However, if your Logi application is not specifically configured for SSL, then using "https:" URLs to include these libraries will *not* work. Use "http:" instead, as shown.

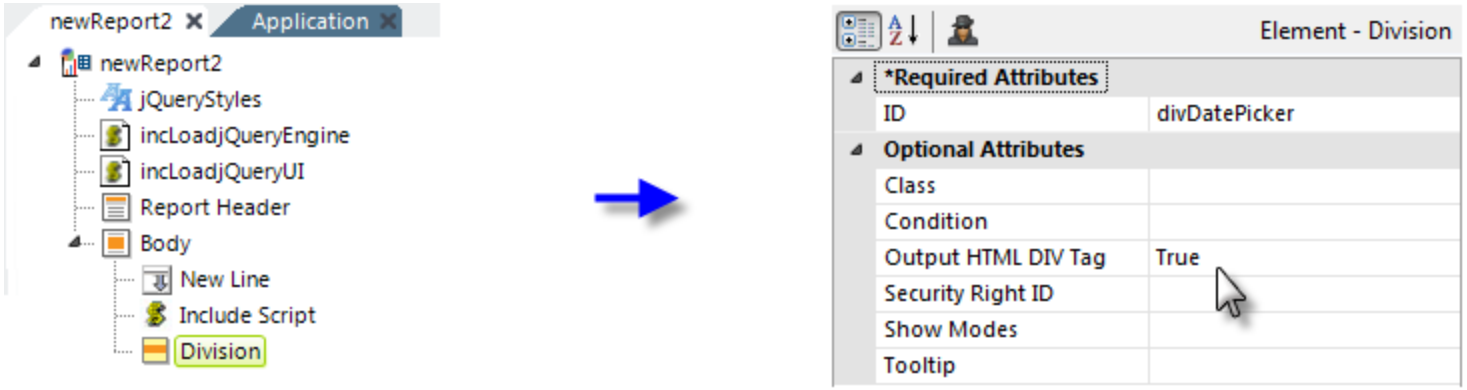
You could, of course, use local copies of the jQuery style sheet libraries, downloaded into _SupportFiles, by just selecting their file names instead.



Next we add an **Include Script** element and enter our jQuery code in its **Included Script** attribute, as shown above. The full code is:

```
$(document).ready(function() {
$("#divDatePicker").datepicker();
});
```

And finally, we need to add a container (a **Division** element) for the date picker,



as shown above. The **Output HTML Div Tag** attribute must be set to *True* and notice that the **ID** of the division is used in the jQuery code, which is case-sensitive.

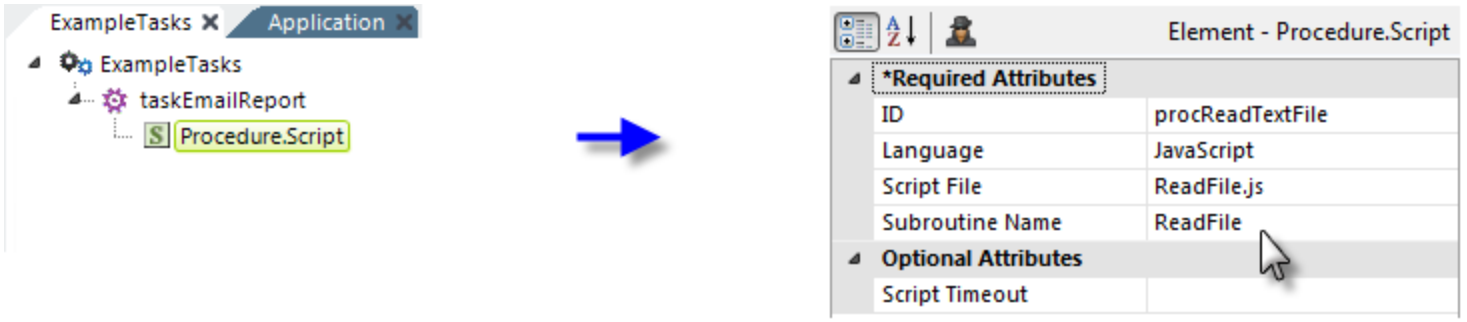
When you run the report, you should see a fully-operational date picker calendar similar to the one shown above. The date picker highlights the current date (default), as well as the selected date.

This, of course, is an extremely basic example but it gives you the idea. Third-party libraries like jQuery offer a lot of functionality and, with these Logi elements, it's easy to make them work in your Logi app.

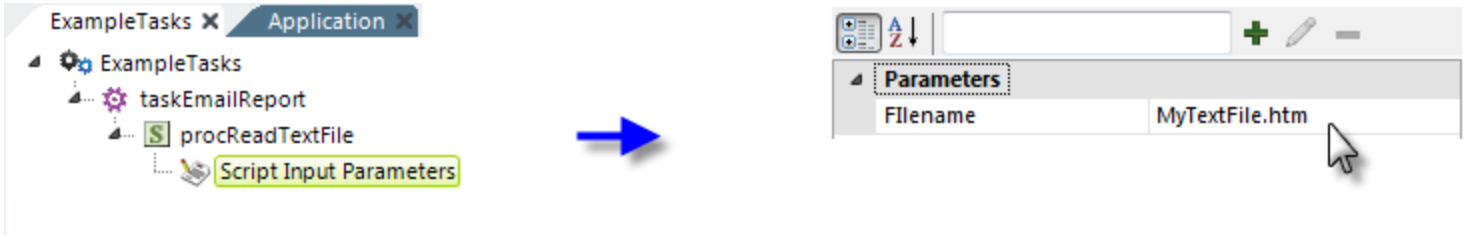
We have a more comprehensive discussion of jQuery in "jQuery" on page 567.

Process Tasks and Script Files

Scripting can also be used in Process definition tasks. Let's look at an example that uses scripting to read a text file on the web server:

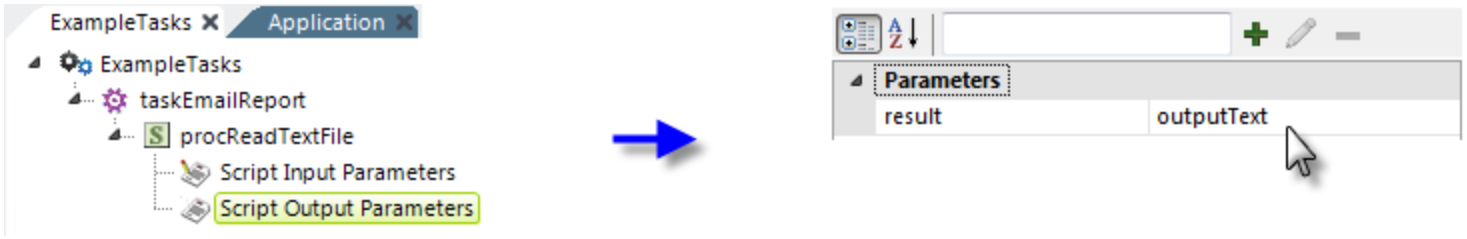


1. In the example above, a **Procedure.Script** element has been added to a task, and its attribute values have been set to point to the script file and function to be called. The Procedure.Script element throws an 'Unsafe class reference' error by default when accessing the file system object. To overcome this error when running a procedure, set the Unsafe Script Allowed attribute in the Process Element to '*True*'. For more information about Process Tasks and its associated elements, see *Process Tasks*.

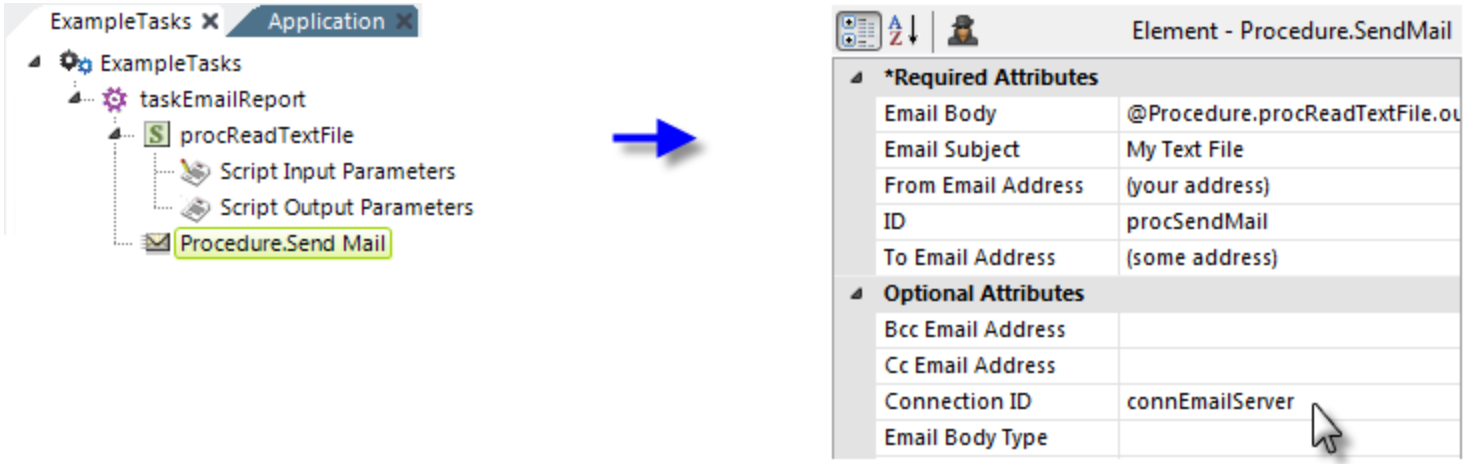


2. Next, a **Script Input Parameters** element is added beneath Procedure.Script. This is the element that passes arguments to the function and its attribute values have been set as shown.

   The arbitrary parameter name "Filename" will become part of a special token we'll use in the actual function to access the passed value. Its value is set to the name of the text file to be read. This assumes that the file is in the same folder as the script file; the value could instead include a more complete path or a combination of the @Function.AppPhysicalPath~ token and other folder/filename information.



3. Next, a **Script Output Parameters** element is added and its attributes set to arbitrary values. This is the element that receives the value (the text from the text file) the function returns. The parameter name "result" will be the return variable in the function and we'll reference the value "outputText" later in the task to view the returned text.



4. Finally, an element is added to do something with the returned text, in this case, a **Procedure.Send Mail** element. Its attributes are set as shown, and a token that incorporates the value from the Script Output Parameters, `@Procedure.procReadTextFile.outputText~`, is used to access the text returned from the function.

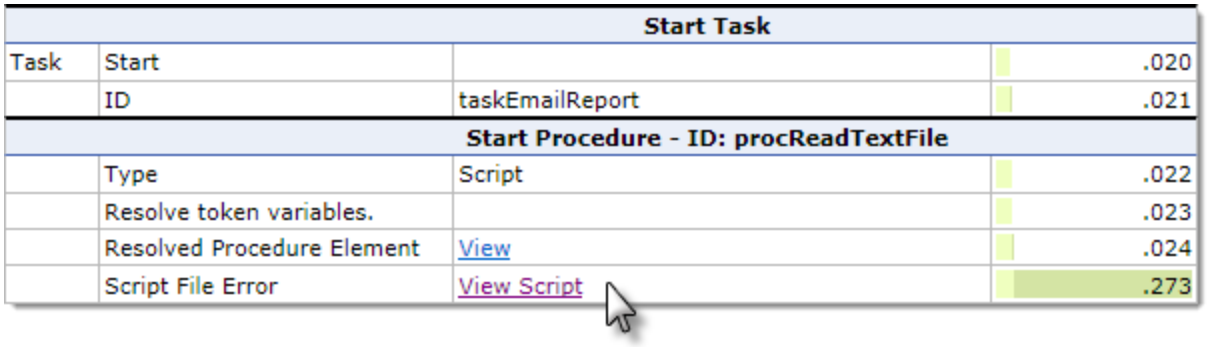
```
function ReadFile() {
var forReading = 1;
var fso = new ActiveXObject('Scripting.FileSystemObject');
var tf = fso.OpenTextFile('@Input.Filename~', forReading);
result = tf.ReadAll();
tf.Close();
}
```

An example of the JavaScript that might be used for this purpose is shown above. Note the way in which the @Input token and result variables are used.


The example above is for illustration purposes only. It uses a Logi Server Engine object now understood to expose substantial security vulnerabilities and is *not* available on Windows 64-bit or Java platforms.

Debugging Script Files

You can also cause debugging to extend to scripts. When enabled, a special link willbe included in the standard Debugger Trace page:



Start Task			
Task	Start		.020
	ID	taskEmailReport	.021
Start Procedure - ID: procReadTextFile			
	Type	Script	.022
	Resolve token variables.		.023
	Resolved Procedure Element	View	.024
	Script File Error	View Script	.273

To enable this, configure the **General** element's **Script Source Debugger Style** attribute.

Its *None* value is the default, or you can set it to *OnError*, in which case the View Script File link shown above appears in the Debugger Trace Page when an error occurs. This link displays the script as generated when the report was run. If the script is a simple, single line of script, the actual error may appear here instead of a link.

A third value option, *Always*, causes the link to appear in the trace page at all times. *The routine use of Always is discouraged as it will incur a significant performance hit.*

Divisions

Developers can use the **Division** element in their Logi reporting applications to contain and organize groups of elements within a report. The Division element makes working with groups of element in Studio easier and provides the ability to dynamically apply style classes, or to show and hide report sections.

The following topics discuss the Division element and techniques for its use:

- Hiding/Showing Report Sections Using Conditions
- Hiding/Showing Report Sections Using Show Modes
- Hiding/Showing Report Sections Using Security Rights
- Hiding/Showing Report Sections Using Action.Refresh Element

About the Division Element

In Studio, the **Division** element functions as a "container" for other elements. It's a parent element that can have a wide range of child elements.

The presence of a Division element normally causes a set of HTML <SPAN> tags to be generated when the report is run. However, the element has an attribute that allows <DIV> tags to be generated instead. Generally, the DIV tag is considered a "block" container and SPAN an "in-line" container. A DIV tag also automatically produces a **Line Break** after it.

You also have the option to specify the Output HTML DIV Tag attribute for all elements that are not set manually at the application level. The **Output HTML DIV Tag** attribute in the Division element has two possible values: True or False. If you select **True**, `<div>` is the wrapper tag for your element. If you select **False**, or if the attribute is blank/undefined, `<span>` is the wrapper tag for your element. The global constant, `rdOutputHtmlDivTagDefault`, specifies the default rendering results when the Output HTML

DIV Tag of the Division element is not set explicitly. The default value for this constant is blank, or False. When set to True, the default rendering result is `<div>`.

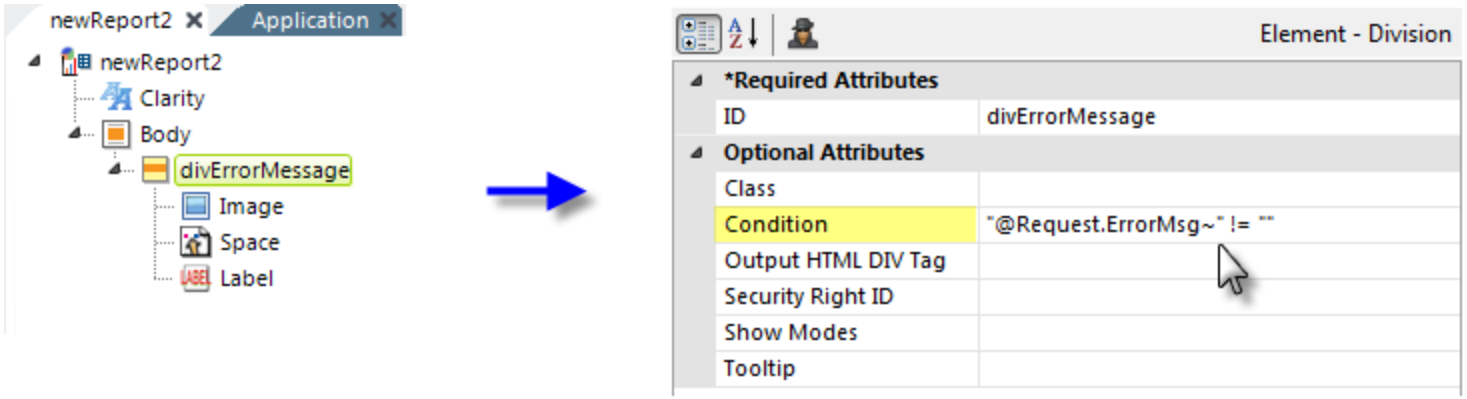
You can assign a style class to a Division element and, as a container, that class will affect all of the Division's child elements. For example, a style class that sets a *font size* can be applied once, to a Division element, and all of the elements the Division contains will also use that font size (unless individually overridden). Division elements also include "HTML Attribute Params", enabling you to apply your application to work with other frameworks or libraries easier. With the HTML Attribute Params, you have the option to include "style" parameters. If an attribute was set in both Element and HTML attribute params, the one set in the HTML attribute params will be ignored.

Divisions can be also be used as an *organizing* entity. Report sections, containing any number of elements, can be placed within a Division. Then the Division can be shown or hidden, using dynamic criteria, causing the section to appear or disappear in the report. This provides tremendous flexibility and can be used to reduce the number of report definitions that need to be developed and maintained and can be used to make a report appear differently in different circumstances or for different users.

The Division element's other attributes, **Condition**, **Show Modes**, and **Security Right ID**, can be used to show and hide the container.

Hiding/Showing Report Sections using Conditions

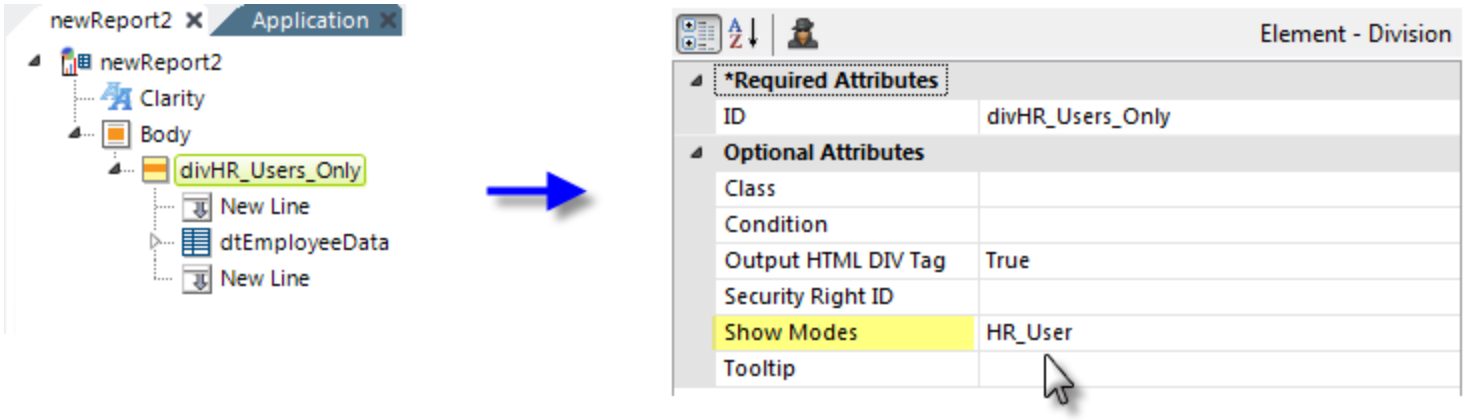
The Division element also has a **Condition** attribute, which can also be used to set its visibility (and the visibility of its child elements). As discussed earlier, circumstances may dictate that only certain parts of a report be visible to a user.



This can be achieved by placing elements beneath a Division element, then setting the Division element's **Condition** attribute accordingly, as shown above. If the Condition attribute value evaluates to *True*, then the Division and its child elements will be made visible; otherwise they'll be hidden. For more information, see "Conditions" on page 670.

Hiding/Showing Report Sections Using Show Modes

You can also use **Show Modes** attribute to set Division visibility (and the visibility of its child elements). Under some circumstances, this may be more desirable than using a Condition expression.

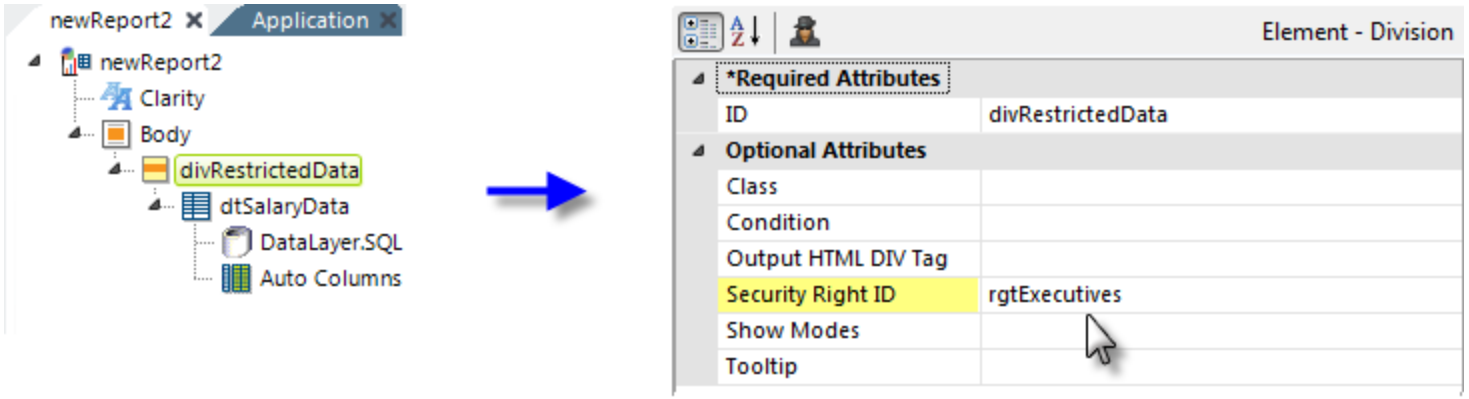


This can be achieved by placing elements beneath a Division element, then setting the Division element's **Show Modes** attribute accordingly, as shown above.

There are built-in Show Modes values available which can be very useful. For example, using the *rdBrowser* value will cause the division to be seen only when the report is displayed as HTML in a browser; using *rdExportPdf* will cause the division to be visible only when the report is exported as a PDF. For more information, see *Show Modes*.

Hiding/Showing Report Sections Using Security Rights

The Division element also has a **Security Right ID** attribute, which can also be used to set its visibility (and the visibility of its child elements). As mentioned earlier, circumstances may dictate that only certain parts of a report be visible to certain users.

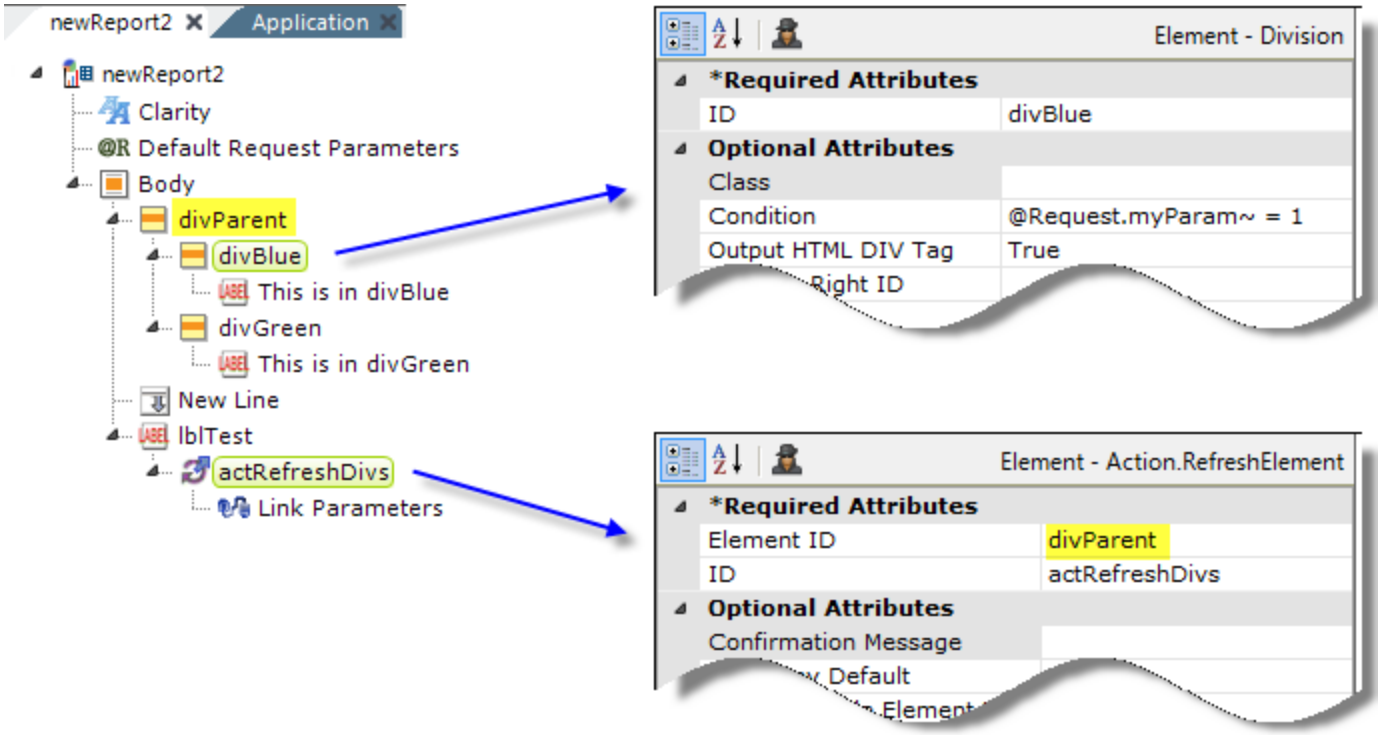


This can be achieved by placing elements beneath a Division element, then setting the Division element's **Security Right ID** attribute accordingly, as shown above. If Logi Security is enabled and a user's **security role** has the security right identified in this attribute, then the Division and its child elements will be made visible; otherwise they'll be hidden. For more information, see *Intro to Logi Security* .

Hiding/Showing Report Sections Using Action.Refresh Element

The **Action.Refresh Element** element, using AJAX technology, allows you to *refresh* a portion of a report page instead of the regenerating the entire report page. It can be used, for example, to refresh Division elements and, when used with a **Link Parameters** element, can pass parameters that can cause divisions to be shown or hidden.

To get the refresh to work properly, you must place the Division element(s) to be refreshed *beneath* a parent Division element and then make that parent element the target of your Action.Refresh Element element. Do not target the child Division element (s).



In the example shown above, two Division elements have their Condition attributes set so they're displayed based on a Request parameter value. They're both children of the "divParent" element. Clicking the link "lblTest" will refresh "divParent" and pass a parameter value. The two child Division elements will be shown or hidden based on that parameter value.

Tokens may be used in theAction.Refresh Element element's **Element ID**attribute value.

Glossary

A

API

API, short for Application Program Interface, is a set of routines, protocols, and tools for building software applications. In business intelligence, APIs may be used to enable end-users to directly update source systems.

Authentication

Authentication is the verification of a user's identity.

Authorization

After a user's identity has been authenticated, authorization grants or denies access to reports, columns, and records to selected users or user-groups.

B

Big Data

Refers to both the ever-growing volumes of data in use today and also to services that are specifically engineered to provide and manipulate very large data volumes.

Business Analytics

Business analytics, or business intelligence (BI), gives customers the ability to rapidly create scalable, interactive data analysis applications, and self-service capabilities users can access from anywhere and on any device.

C

Columnar Data Store

Columnar data store is a type of big data repository containing structured data in columns and rows. The main benefits are that the data can be highly compressed and is easily searchable.

CRM

A Customer Relationship Management (CRM) system is a database-based system that records a company’s daily customer-related transactions. CRMs can help customer representatives to provide better service, close more deals, and increase revenue.

CSS

Cascading Style Sheets (CSS) is a technology that allows the presentation aspects of web pages to be separated from the page content. It can be used to add "styling" (e.g. apply fonts, colors, alignment, spacing, and more) to web pages.

D

Data Discovery

Data discovery is the capability to analyze data on-the-fly and uncover insights from it.

Data Enrichment

Data enrichment is a method of preparing data to make it ready for analysis and exploitation, and can include formatting, adding calculations, joining with other data, and more.

DevNet

The Logi Developer Network website.

Drill Down

Drill Down is a capability that allows the user to get a view of the underlying or supporting data used in an analysis.

Drill Through

Drill Through is similar to Drill Down but takes it one step further by applying analysis to the underlying or supporting data.

E

Elemental Development

A development approach used in Logi Info that lets developers build feature-rich applications by using reusable, pre-built elements, rather than by writing low-level code.

F

Forecasting

A technique involving data mining and analysis leading to predictions about what will happen in the future.

G

Geo Mapping

The combination of geographic and other data to produce map visualizations, such as Google or Leaflet maps.

H

Heatmap

A Heatmap chart, sometimes called a "tree map", which uses a unique arrangement of rectangles to represent data and relationships, using color and size.

I

Interpolation

The process of evaluating a literal value match containing one or more placeholders, yielding a result in which the placeholders are replaced with their corresponding values.

J

JavaScript

JavaScript is a programming language supported by the majority of modern web browsers and used by many websites.

JDBC

Java Database Connectivity (JDBC) is an API used to access relational databases. Open Database Connectivity (ODBC) is a similar API designed for use with Java.

JSON

JavaScript Object Notation (JSON) is a lightweight data-interchange format that's easy for humans to read and write, and easy for computers to parse and generate.

K

KPI

Key Performance Indicators (KPIs) are visual indicators, in the form of color-coded shapes, which are tied to a pre-defined, critical threshold.

L

LDAP

The Lightweight Directory Access Protocol (LDAP) is an Internet protocol applications use to look up information from a server and is frequently used for containing user login information.

M

My Term

My definition

N

NoSQL

"Not only SQL" (NoSQL) is an alternative to traditional relational databases, and doesn't rely on tables and a pre-determined schema. NoSQL databases are especially useful for working with large sets of distributed data.

O

ODBC

Open Database Connectivity (ODBC) is an API used to access relational databases. Java Database Connectivity (JDBC) is a similar API designed for use with Java.

OLAP

Online Analytical Processing (OLAP) is the process of analyzing data stored in multi-dimensional "cubes".

R

REST

Representational State Transfer (REST) is a type of API used to provide interoperability between computer systems on the Internet.

S

SSM

The Self-Service Module (SSM) is a package that includes Logi Info + SSRM + Discovery or Logi Platform Services.

SSRM

The Self-Service Reporting Module (SSRM) is a Logi Info add-on module that adds special elements to Info and includes the InfoGo application.

W

Write-Back

The ability to update data sources, typically by adding, editing, or deleting data.