# 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material may assist with understanding historical functionality, please be aware that the software described is no longer maintained at this version level and may contain outdated or inaccurate information.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.

# Design a Logi Report

The following topics introduce developers to the basic concepts involved in designing a typical Logi report:

- About HTML Containers
- Report Layout
- Adding Live Data
- Finishing Up

## Purpose and Prerequisites

As mentioned above, the purpose of this topic is to give developers who are new to Logi Analytics technology and "Elemental Development" an introduction to designing a Logi report.

Ideally, developers who are going to work with Logi products should have a good understanding of:

- HTML and XML
- CSS
- SQL (or whatever it takes to access the datasource they're working with)
- JavaScript
- "Stateless" web application concepts and web server technology

The Logi Server Engine, running at the web server, combines page definitions with data and returns the report as a complete web page to the requesting browser. The returned page consists of HTML, XHMTL, and JavaScript, which the browser processes and displays.

For a developer, understanding the potential, and limitations, of these technologies is the key to a satisfactory development experience with Logi products.

## Standards? What Standards?

The HTML and browser paradigm was conceived with the idea of universal and generic presentation of information. Despite the efforts of international standards organizations, this lovely idea has morphed over time into something a bit different and sometimes it seems as if there are *no standards*. In reality, browsers vary by manufacturer and version in their presentation of web pages, providing a challenge for those developers whose target audience uses a variety of browsers.

Logi Info makes a serious effort to ensure that their page output is generic XHTML, CSS, and JavaScript, in order to promote cross-browser compatibility. However, there's no way to predict what changes the browser manufacturers will uncork with their next release, so developers *shouldfrequentlytest* their Logi apps against the range of browsers and browser versions that could be used to view their applications to ensure compatibility.

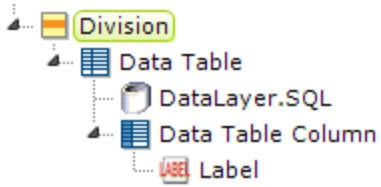
There are a number of online testing sites that make it easy to see what your Logi app will look like in different browsers. Some, like BrowserStack, display images of your app in different browser, while others, like Browserling, show your app in a specific browser in an iFrame.

## HTML5

The ever-evolving HMTL5 standard is another moving target for developers. Different browsers support different sets of HTML5 features, and sites like HTML5 Test that can help you discover the related capabilities of your specific browser make and version. Another useful site, Can I Use, indicates which HTML5 features are available in various browsers on an feature-by-feature basis. Logi Analytics is working to include more HTML5 features in our products with each release.

# About HTML Containers

A key concept in the design process is the **container**. A container is an HTML entity, defined in the code with starting and ending tags, that contains or surrounds other entities on the page. Sometimes they actually appear on the page and sometimes they don't. Containers can be used to limit the size of parts of the report page, to apply style to their contents, to isolate part of the page from other parts, etc. A variety of Logi elements are containers, and the parent-child element relationships found in Logi Studio often identify them.




In the example above, the **Division** element is the outer container for all of the other elements; the **Data Table** contains the datalayer and the **Data Table Column**, and the Data Table Column contains the **Label** element. As mentioned earlier, through inheritance CSS style applied to a container is generally applied to all the items it contains.

In a Logi app page, containers are the **geographic building blocks** of the page. Successful Logi developers begin report development by visualizing the desired end result and breaking it down into these building blocks, which then dictate the report layout.

## Responsive Design Elements

Responsive Design elements, including **Responsive Row** and **Responsive Column**, may provide an improved viewing experience. They can automatically change their size and arrangement depending on the device screen size. This is very helpful when designing applications to be seen on a variety of devices, from desktops to smart phones. See *Responsive Design Elements*for more information about these elements.

# Using a Master Report

The Master Report concept, introduced in v12.1, provides a special report that contains elements that are to be included in other reports. One common use is to define a Master Report with a header, a horizontal or left Side Bar menu, and a footer. Regular report definitions then reference the Master Report and, at runtime, the two are merged together. Logi Studio includes a Master Report Layout wizard that can be used to quickly generate a Master Report and this is a great way to design your application.
More information is available in "Master Reports" on page 71.

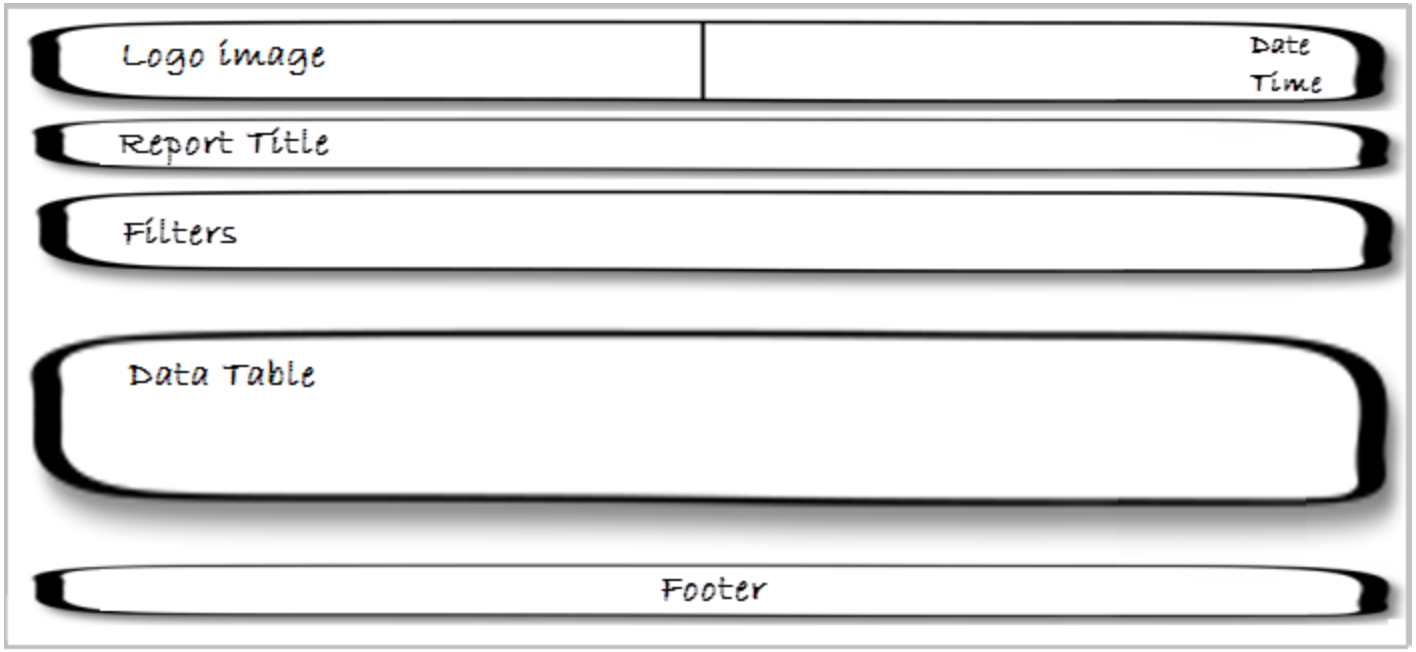
# Report Layout

Desktop application developers are used to working with "absolute positioning", where each item on the screen can be accurately positioned with X-Y coordinates. This works because of the tight relationship between the OS, the video hardware, and the application.

However, on a web page, things are a little looser: the browser doesn't know about the web server OS and adjusts to whatever screen it's being viewed on; absolute positioning generally lacks the flexibility needed for this situation.

Therefore, when developing a Logi report, "flow positioning" is typically used*. With it, page components are positioned in relation to other components, rather than at fixed points on the screen. As a developer, youhave to keep in mind how information on the page will flow to, in, and even around, containers. In particular, spacing and alignment work differently in flow positioning, as we'll see.

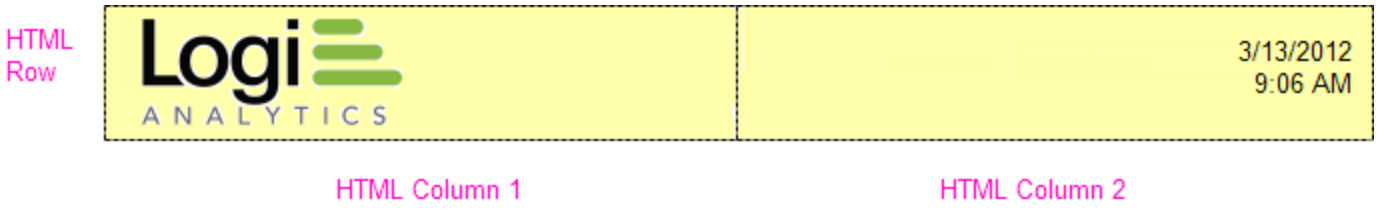
It's a good design practice to **start** by envisioning the **end result**: for example, we want a professional-looking report, with a Data Table and some data selection filters in it. Let's give that vision some substance by drawing on paper the large regions that will be the report's top-level containers.



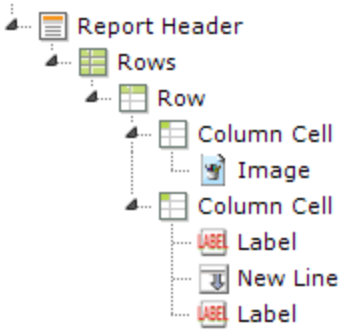

Our rough drawing of the containers is shown above. It gives us an idea of the number and the order of the containers needed. We can also start thinking about *alignment* at this point; notice that the top container is divided into left and right halves- this is because its contents will be left- and right-justified, respectively.


3/13/2012
9:06 AM
Title goes here
Input Select List and Input Date go here
Order #
Cust ID
Emp ID
Ordered
Shipped
Ship To
City
Country
Data table data goes here
Proprietary Information
Do not distribute externally

Now let's imagine that the five containers we drew on paper are overlaid on our report page in different colors, so we can see how the rough drawing correlates to the real page. We'll use this page in the following discussion to understand how to achieve the desired layout using elements and CSS.



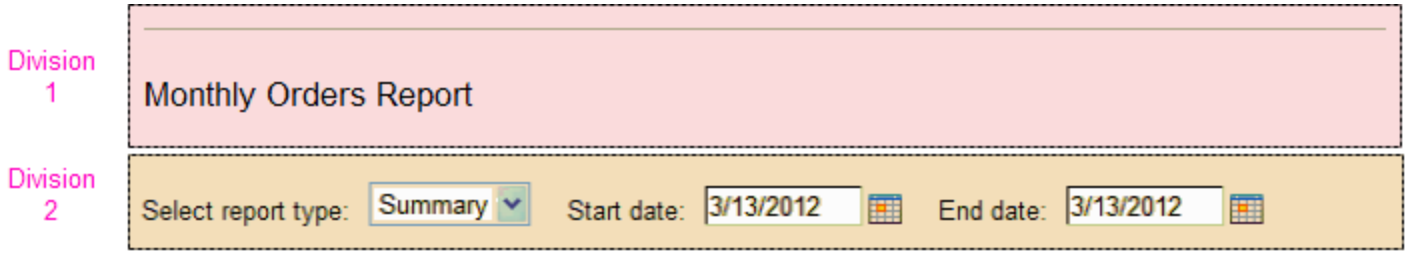

Looking at the top container in the page, it's easy to see that it can be displayed as one"row" in a table, with two "columns".



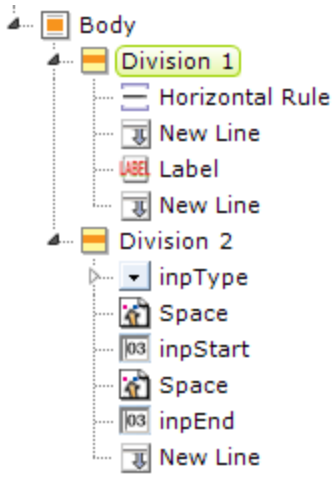

The elements shown above (**Rows**, **Row**, **Column Cell**) provide the HTML table components needed to create this effect, see "HTML Tables" on page 416. The **Width** attribute of the Rows element is set to 100% and the width of each Column Cell is set to 50%. The **Class** attributes of the Column Cells are set to *AlignLeft* or *AlignRight* to get their contents to justify properly (an example of container contents being affected by CSS). Use of an HTML table like this is a very common technique for aligning content.

You may also notice that the image and input text items are not "flush" with the outer edges of the row columns; there's a little bit of space there. This is achieved by adding CSS **padding** to the styling of the Column Cell elements.

At this point we should recognize that this container, the "report header", could be used for other reports, too. That's a hint that this set of elements is a good candidate to be made into *Shared Elements*so it can be *included*, rather than redefined, in every report. Shared elements are a powerful Logi Info feature.




The next two containers in our plan are similar but no justification is involved, so let's use **Division** elements as containers, rather than an HTML table, and we'll place them in the BODY of the report.




The elements needed for this are shown above. Division elements produce either SPAN or DIV tags in the final HTML and are very handy. They don't have a physical appearance in the report page but can be used to apply style to their contents *and* they have a **Condition** attribute which lets you show or hide them dynamically. They're also a great way to group elements in the definition, making it easier to copy them, and for plain old visual organization purposes.

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>`.

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.

## Tables or Divs?

The use of HTML tables is considered a bit "old school" these days, with many developers preferring to use Divisions and CSS to arrange and align content. For example, Divs can be assigned classes with the *float* property, in order to left-, center-, or right-align their content. There's some evidence that this approach also works better for mobile devices but, ultimately, the approach you choose for your Logi application depends on what works best for you.

If you've ever heard the acronym **KISS** ("Keep It Simple, Stupid") then this is a great place to keep it in mind. A layout that involves a lot of nesting, spanning, and other convolutions is very likely to stand up poorly when delivered to a variety of browsers. Overly-complicated layouts often suffer from incomplete or unexpected CSS inheritance and spell display disaster! You should always look for the *least-complicated* way, using the fewestnumber of elements, of implementing your report.

So far, our definition is simple and straightforward. Now we're ready to look at the next container, the one where there's some actual data displayed.

## What About Menus?

Our example doesn't require a menu, but many Logi applications do. We're not going to discuss them here but Logi Info provides several ways to implement different menus, depending on your needs. Menu Tree elements are used in Master Reports and, with or without them,offer a quick and easy way to build a menu, while Pop-up menus display a panel with menu options, and the Report Center Menu is useful for displaying a menu based on the available reports. See *Work with Menu Tree Menus*, *Work with Popup Menus*, and *ReportCenter Menu* respectively, for more information.

Logi Studio supports absolute positioning as an option but we generally do not recommend that Logi apps be developed using it.

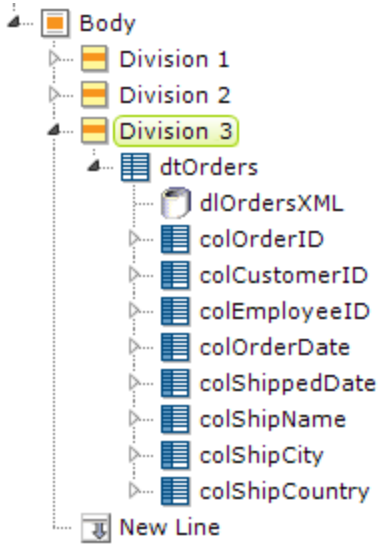
# Adding Live Data

There are multiple techniques available in Logi Info for displaying data; in our example, we're displaying it as text in a tabular format. We'll also see how we can display it in an alternate"form-style" format.

Division 3

| Order | Cust ID | Emp ID | Ordered | Shipped | Ship To | City | Country |
|---|---|---|---|---|---|---|---|
| 10248 | VINET | 5 | 7/4/1996 | 7/16/1996 | Vins et alcools Chevalier | Reims | France |
| 10249 | TOMSP | 6 | 7/5/1996 | 7/10/1996 | Toms Spezialitäten | Münster | Germany |
| 10250 | HANAR | 4 | 7/8/1996 | 7/12/1996 | Hanari Carnes | Rio de Janeiro | Brazil |

Like the previous containers, this one uses a Division element. However, it includes a **Data Table** within the container and, as you'll see later, some of its Data Table Columns can be containers themselves.



The elements for the third division and the Data Table are shown above. The Division element has another useful characteristic that makes it a good choice as a container for Data Tables and charts: it can be *refreshed* using an AJAX call. This is usually done with an Action.Refresh Element, and when the action occurs, *only* the Division's contents will be refreshed, not the entire page. This is usually faster than a full page refresh, and cleaner - the user doesn't see the entire page "blink". For more information about Action.Refresh Elements, see *Action Elements*.

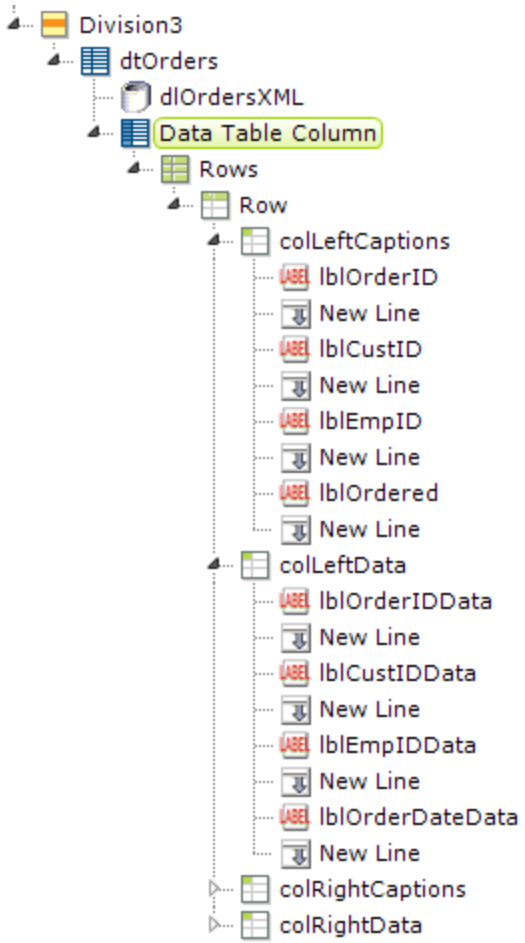
Division elements also include "HTML Attribute Params", enabling you to apply your application to work with other frameworks or libraries easier. With the HTML Attribute Params, you have the option to include "style" parameters. If an attribute was set in both Element and HTML attribute params, the one set in the HTML attribute params will be ignored.

## Presenting Data in a Form

Suppose you want to display the data from just *one* record at a time, in which case embedding a Data Table in your container may be overkill. You'd like to display it in a traditional form-style arrangement, like this:

| | | | |
|---|---|---|---|
| Order ID: | 10248 | Shipped: | 7/16/1996 |
| Customer ID: | VINET | Shipped To: | Vins et alcools Chevalier |
| Employee ID: | 5 | City: | Reims |
| Ordered: | 7/4/1996 | Country: | France |

This can be done by using just *one* Data Table column, as a container, and putting an HTML table inside it.

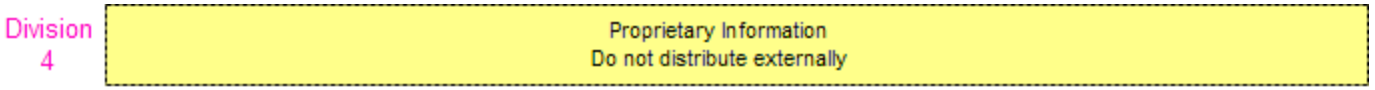



The necessary elements are shown above. It takes a bit of work to get all the elements configured but Studio's element "copy andpaste" feature helps a lot. An HTML table is used so that captions and data can be formatted into justified columns, and it's located beneath a Data Table column so that it will be "in scope" for the datalayer and have access to the data as @Data tokens.

There is another way to use an HTML table to display data in a form, *without* it being contained by a Data Table column. This is accomplished by using a Local Data element at the top of the definition; it's particularly well-suited for this purpose because it will only retrieve one record of data and its datalayer's scope is the entire report. So, if your data query is going to retrieve just one record, you can use @Local tokens to access the retrieved data *anywhere* in the report, freeing you to put your HTML table wherever you'd like. For more information, see *Using Local Data*.

# Finishing Up

The final container in our report is the "proprietary information" warning at the bottom.




This container can be another Division element and is another opportunity to create a shared element. Traditionally, you'd place it beneath the **Report Footer** element.

The Report Header and Report Footer elements provide convenient places to put that kind of information, but they're not special in any way. If you prefer, you can place your header and footer container elements right in the BODY of the report instead, and they'll look and behave in the same way.

3/13/2012
1:06 PM

## Monthly Orders Report

Select report type: Summary  Start date: 3/13/2012  End date: 3/13/2012

| Order # | Cust ID | Emp ID | Ordered | Shipped | Ship To | City | Country |
|---|---|---|---|---|---|---|---|
| 10248 | VINET | 5 | 7/4/1996 | 7/16/1996 | Vins et alcools Chevalier | Reims | France |
| 10249 | TOMSP | 6 | 7/5/1996 | 7/10/1996 | Toms Spezialitäten | Münster | Germany |
| 10250 | HANAR | 4 | 7/8/1996 | 7/12/1996 | Hanari Carnes | Rio de Janeiro | Brazil |
| 10251 | VICTE | 3 | 7/8/1996 | 7/15/1996 | Victuailles en stock | Lyon | France |
| 10252 | SUPRD | 4 | 7/9/1996 | 7/11/1996 | Suprêmes délices | Charleroi | Belgium |
| 10253 | HANAR | 3 | 7/10/1996 | 7/16/1996 | Hanari Carnes | Rio de Janeiro | Brazil |
| 10254 | CHOPS | 5 | 7/11/1996 | 7/23/1996 | Chop-suey Chinese | Bern | Switzerland |
| 10255 | RICSU | 9 | 7/12/1996 | 7/15/1996 | Richter Supermarkt | Genève | Switzerland |
| 10256 | WELLI | 3 | 7/15/1996 | 7/17/1996 | Wellington Importadora | Resende | Brazil |
| 10257 | HILAA | 4 | 7/16/1996 | 7/22/1996 | HILARION-Abastos | San Cristóbal | Venezuela |

Proprietary Information
Do not distribute externally

And, at last, we can see what our finished report looks like in the example shown above.

In summary, we've seen the correlation between laying out a report design in terms of geographic containers and how those containers, in the form of either an HTML table, a Division, or a Data Table column, are realized using Logi elements. You should now have a good foundation for designing your own reports.

# Report Author

The **Report Author** super-element allows end-users to work with a Dashboard-like interface to create their own basic reports at runtime.

The following topics discuss the use of the Report Author super-element:



## About Report Author

The **Report Author** super-element is available in Logi Studio once the Self-Service Reporting Module (SSRM) add-on is installed. More information is available in *About Add-on Modules*.

Reports created with Report Author can include charts, tables, images, labels, links, PDF, and Excel export links. Your report can also be configured to exclude each of these items individually using the "Include in Schedule" check box in the component settings. For more information, see "Report Components" on page 41.

Charts and tables, which are created using an Analysis Grid, are retrieved when using the Report Author from a "Gallery" file.

The Gallery file is created using the Analysis Grid's "Add to Dashboard" feature but without actually involving a Dashboard - the data is simply saved to a file that isn't related to the usual Dashboard "Save" file. This is discussed in more detail in *The Analysis Grid for End Users*.

Images can be included in Report Author reports by users at runtime by uploading files to a designated folder and then selecting them during the report design process.

Gallery, Save, and uploaded files can also be stored in a database, see *Storing Bookmark, Gallery, and Save Files in a Database*.

 The Report Author is often used with the Auto Bookmark element, allowing all aspects of a report to be automatically saved as the user works with the report. For more information, see *Adding Automatic Bookmarks*.

The appearance of the Report Author user interface is controlled by any theme or other styling you may care to apply.

The data used in the Report Author is described in a Gallery file, which in turn is based on metadata. If that metadata changes, such as the removal of a column, the Report Author will display a warning about the missing column, instead of the expected visualization.

# Report Components

A Report Author report consists of the following components:

Design View

1 Sales Staff Performance

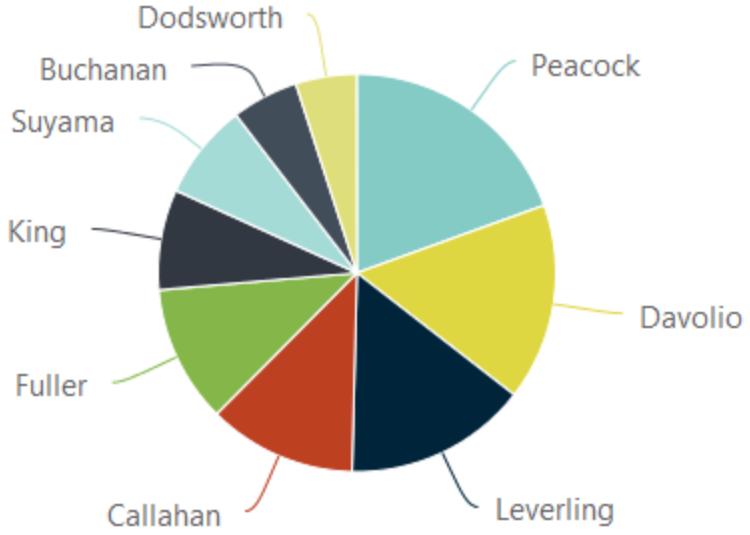
2 

3

4 Click for more information

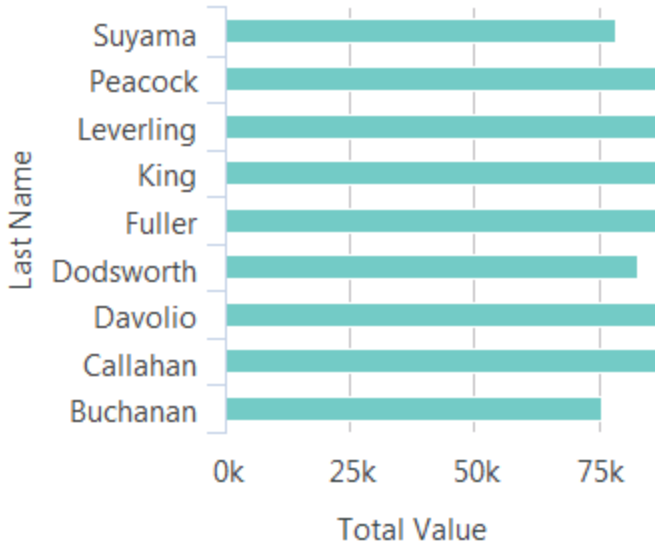
5 Download as PDF

6 Employee Orders




7

Total Value by Sales Person



In View mode, the report components are displayed. They include:

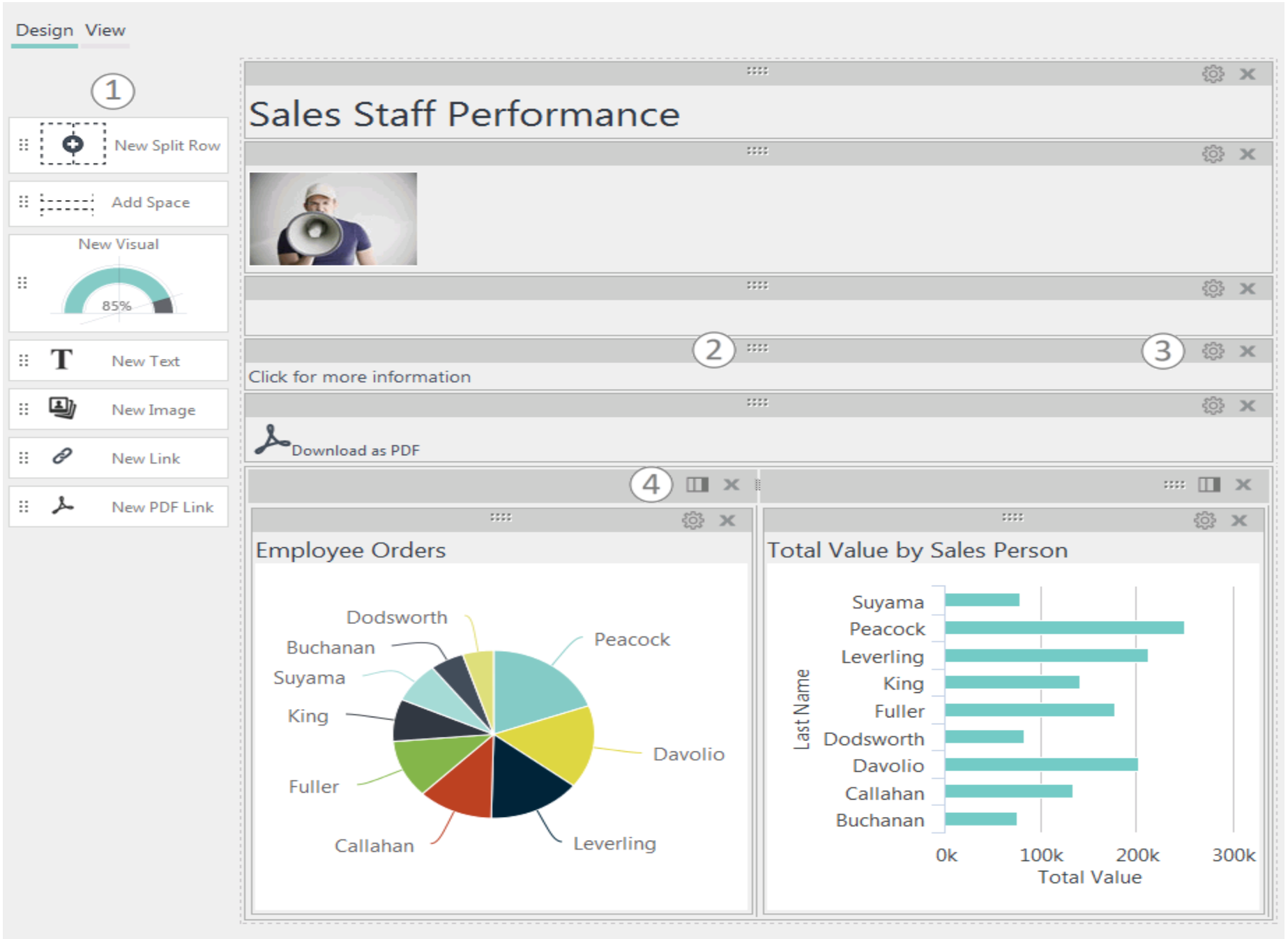
1. **Text** - You can enter text to be displayed in the report. Various font attributes like *size* can be set as desired. Dates and times entered here will be internationalized, based on the configuration of the Globalization element's User Culture attribute in the _Settings definition.
2. **Image** - You can upload an image for display in the report.
3. **Line Space** -You can add blank lines to provide spacing and reading clarity.
4. **Link** - You can add links to other reports or web pages.
5. **PDF Link** - You can add a link that will download the report to PDF format.
6. **Excel Link** - You can add a link that will download the report to Excel.
7. **Split Row** - You can add an empty row that's divided into multiple columns, then drag content into them.
8. **Visual** - You select a visual from your Visual Gallery for display in the report.

You may have as many of these components in your report as you'd like.

# Design Mode

At the top of the Report Author report, you can see the **Design** and **View** links, which let users switch modes.

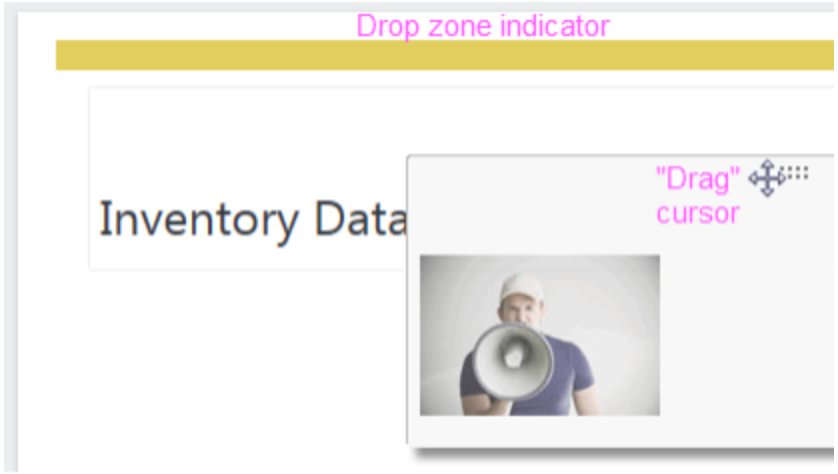
In Design mode, users can work with components to create or edit a report. Components in the report will appear in a series of rows or panels:


Design View
1
New Split Row
Add Space
New Visual
85%
New Text
New Image
New Link
New PDF Link
Sales Staff Performance
2
3
Click for more information
Download as PDF
4
Employee Orders
Dodsworth
Buchanan
Suyama
King
Fuller
Callahan
Peacock
Davolio
Leverling
Total Value by Sales Person
Last Name
Suyama
Peacock
Leverling
King
Fuller
Dodsworth
Davolio
Callahan
Buchanan
0k
100k
200k
300k
Total Value

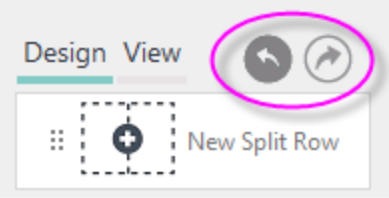
Design mode, shown above, let's users build or modify a report at runtime.

1. Drag components from the **Component Toolbox** on the left onto the report canvas to add them.
2. **Re-arrange** components on the canvas by clicking on their **drag** icon - ⠿ - and dragging them into a new location.When you do this, the cursor changes to a *drag* cursor and a "Drop Zone" indicator (a yellow bar) will appear as you drag the component toward a new location. If the drop zone is above or between other components, they'll move when the component is dropped.
3. **Configure** component settings or delete them by clicking the **gear** or **X** icons.
4. **Add/Remove columns** for the Split Row component by clicking the **columns** icon.

When you add a Visual component to the canvas, your visual gallery will be displayed and you can select one or more visuals to be inserted into your report, as shown in "Adding Visuals" on page 48. Chart animation, resizing, hover highlighting, and quicktips will all be active in the report. Each selected visual will be inserted into its own separate panel.



Components can be re-arranged by dragging their drag icon. As shown above, the cursor changes to a *drag* cursor and a "drop zone indicator" bar will appear as you drag the component toward a new location. If the drop zone is above or between other components, they'll move when the component is dropped.




If an **Auto Bookmark** element is used as a child of the Report Author to automatically record changes, the Undo/Redo icons, shown above, will appear.

An optional Report Author attribute setting, discussed later, allows you to control the default mode, View or Design, and whether editing is allowed.

Tokens used to configure the Report Author are *not* resolved in Design mode.

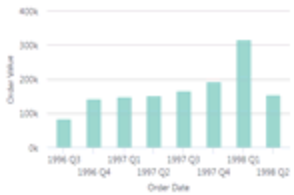
# Adding Visuals

When the **New Visual** component is dropped on the report canvas, your gallery of visualsis displayed:

## Visual Gallery

X

Find Type to search... Gallery Sort By Newest



**Sum of Order Value by Order Date Quarter**
Created: 2/12/2016 10:18 AM
This Bar Chart shows the quarterly order values over time from 1996 - 1998

Add Delete from List

---

**Average of Total Value**
Created: 9/16/2015 11:53 AM

Add

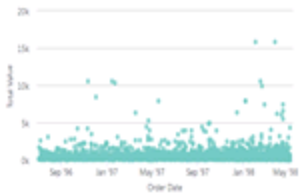
Visuals may be configured so they cannot be deleted.

---

**Total Value by Customer ID**
Created: 9/16/2015 11:47 AM

Add Delete from List

---



**Total Value by Order Date**
Created: 9/16/2015 11:46 AM

Add Delete from List

---

Done

As you can see above, it's a collection of all of the visuals saved in the gallery. You can recognize them from their thumbnail images, and from the titles and descriptions entered when they were saved.

The **Find** control let's you filter the displayed visuals by typing in the full or partial title of a visual, and the **Sort** control let's you set the display order of the visuals by *Newest* or *Oldest*, or by *Title* alphabetically.

If your application has been configured for *multiple* galleries, discussed later, at first you'll see *all* of their visuals combined into one big collection. You'll also see the **Gallery** selection list, which allows you to filter the displayed visuals by gallery.

Each visual has an **Add** button that adds it to the report. Once a visual has been added, its Add button disappears and "Added" is displayed.

Visuals can be removed from the reportusing its X icon. If a visual is removed, it will appear in the gallery with its Add button displayed again.

Each visual from *your* gallery also has a **Delete from List**button, which deletes it from the gallery. If multiple galleries are in use, visuals from them may or may not have this button, depending on whether their creator configured the gallery to be "read-only".

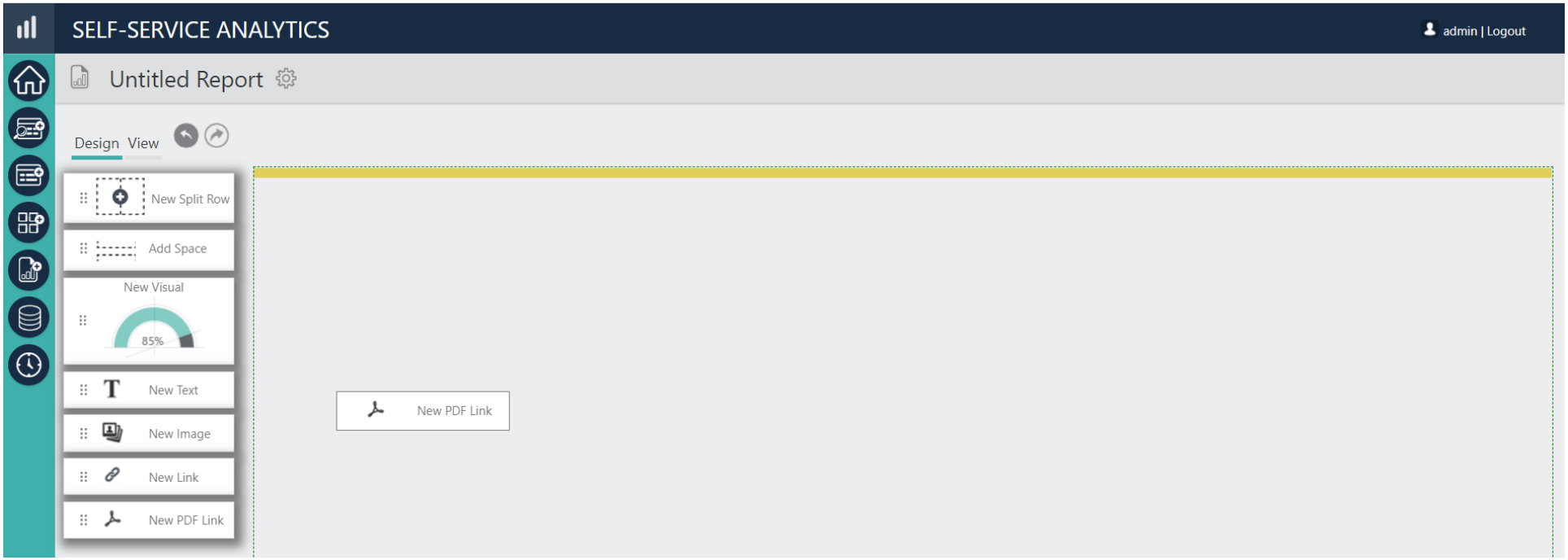
Use the **Done** button to close the Visual Gallery.

# Exporting Visuals

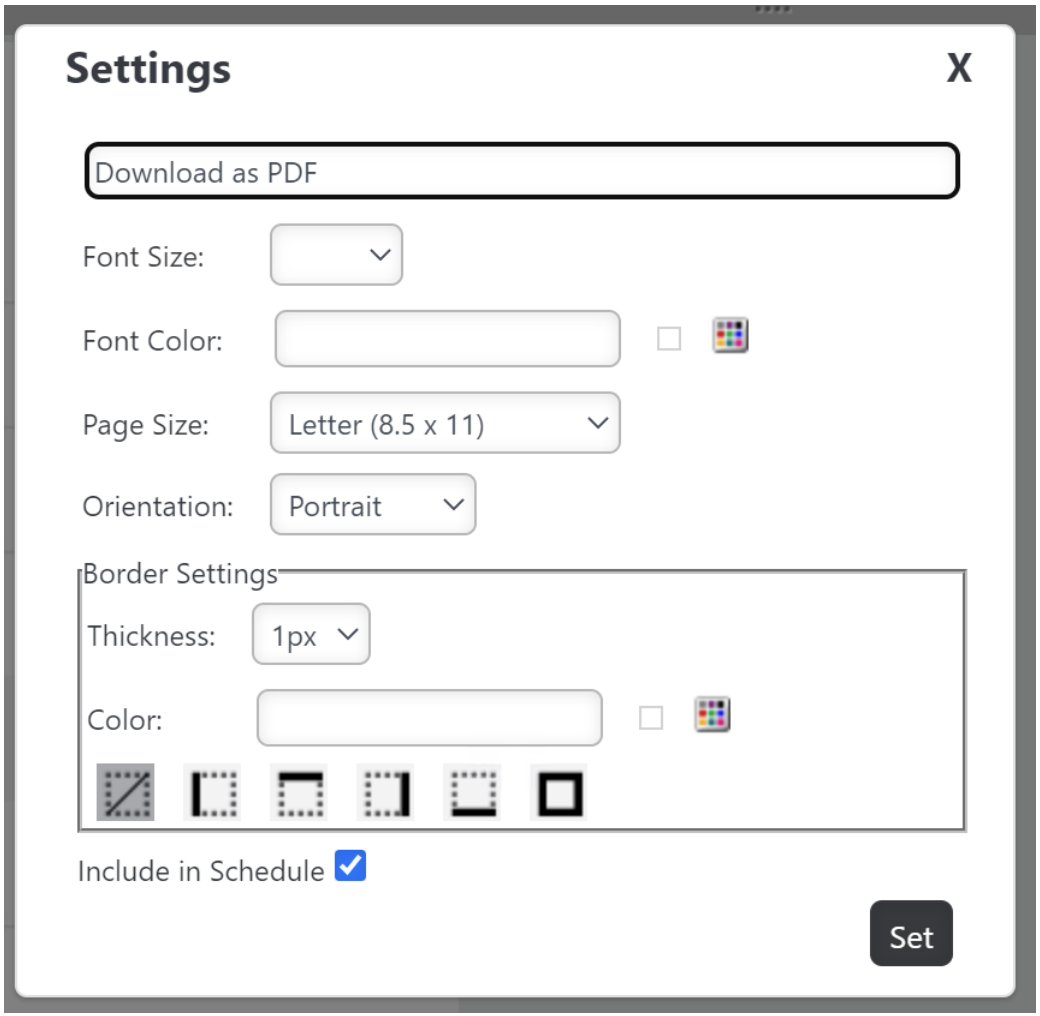
As mentioned in "Report Components" on page 41, you can drag the PDF and Excel links into your Author Report to allow your users to export the report for further analysis.

## Export to PDF

1. In Design mode, begin by dragging the New PDF Link to the report:




Info displays a Settings dialog, shown below:



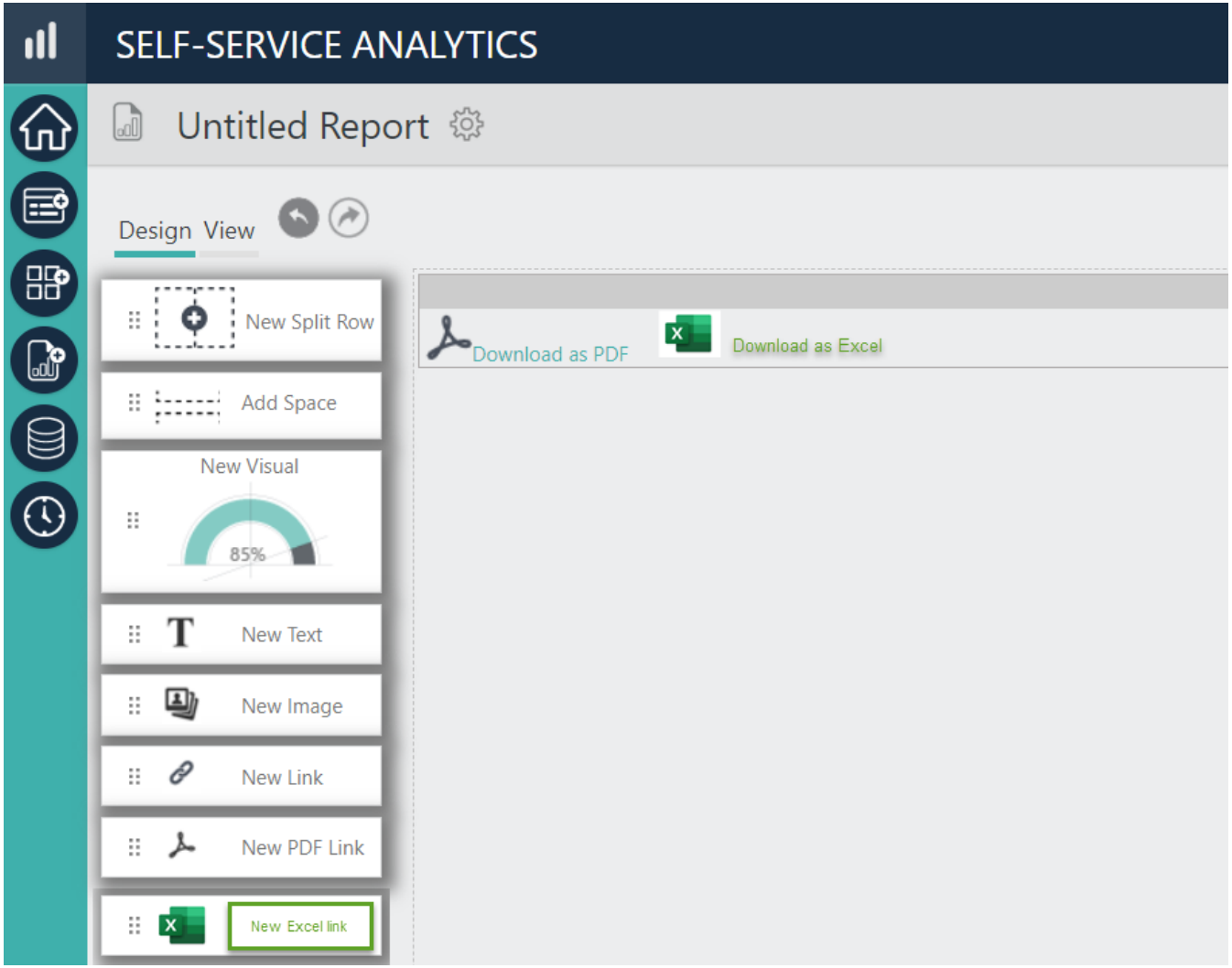

Here, you have the ability to customize the font size, font color, page size, and orientation of the report, as well as additional border settings.  Un-check the "Include in Schedule" check box to remove an object from the final scheduled output.

2. Make any necessary changes, then, select **Set**.
3. Next, select **Download as PDF**:

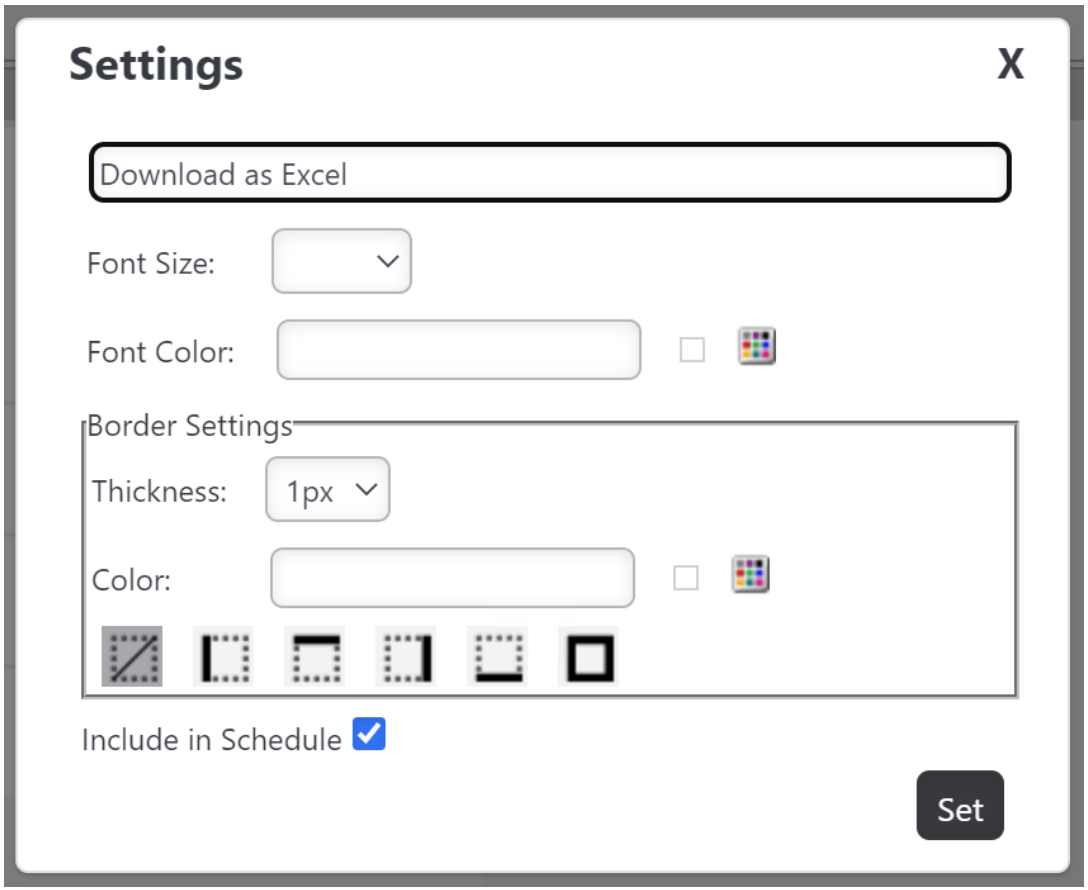

The report downloads automatically in the browser. Your Author Report is now available for further analysis in PDF format.

## Export to Excel

1. In Design mode, begin by dragging the New Excel Link to the report:




Info displays a Settings dialog, shown below:




Here, you have the ability to customize the font size, font color, page size, and orientation of the report, as well as additional border settings.  Un-check the "Include in Schedule" check box to remove an object from the final scheduled output.

2. Make any necessary changes, then, select **Set**.
3. Next, select **Download as Excel**:

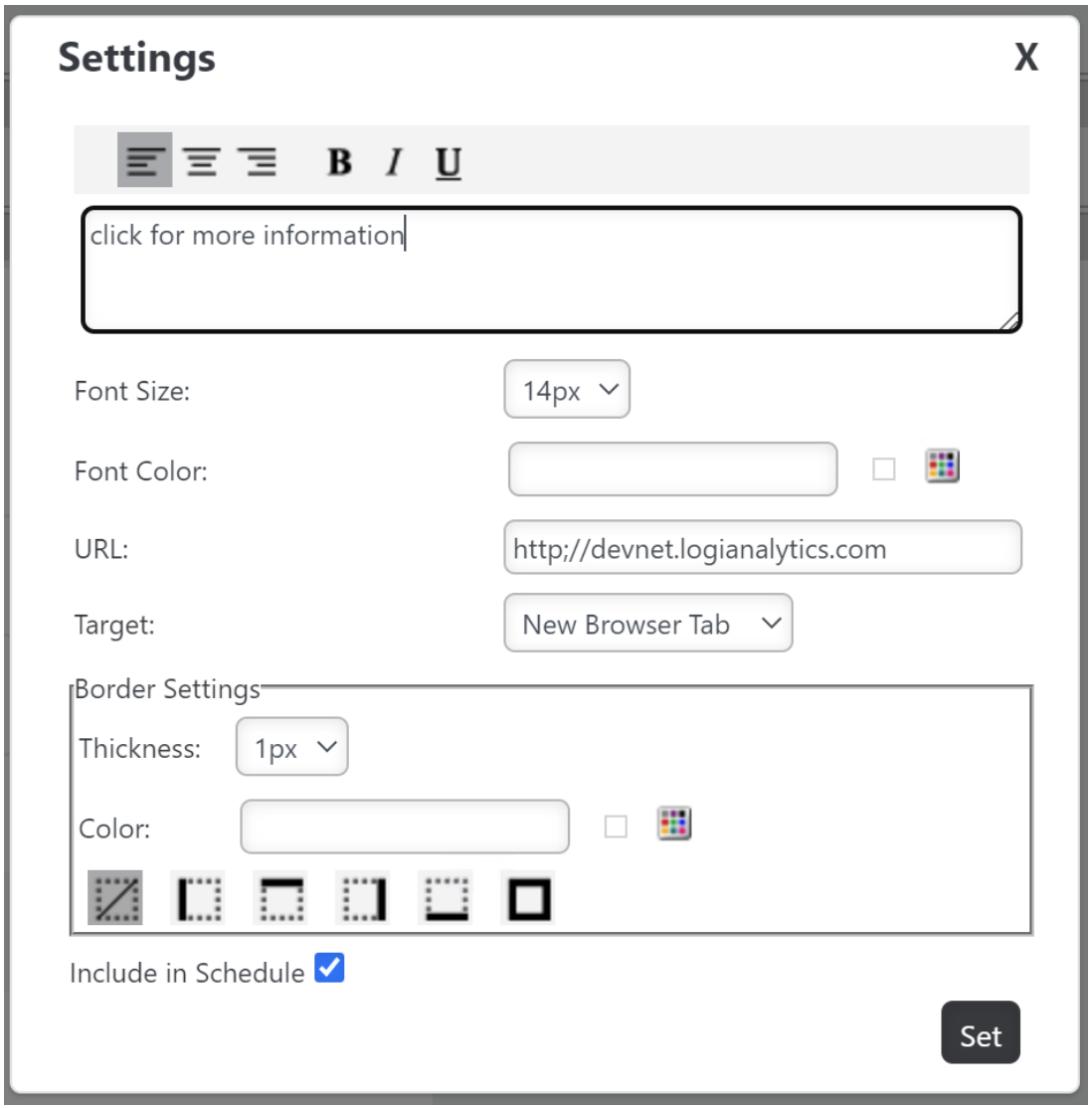
The report downloads automatically, or will open within the browser, depending on your Excel application settings. Your Author Report is now available for analysis in Excel.

Excel results will be HTML-based output.

# Component Settings

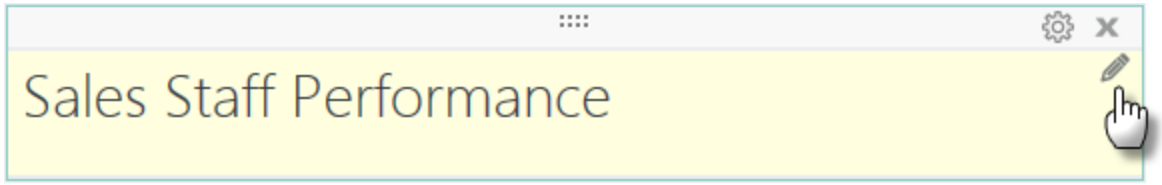
When other components are dropped on the canvas, their Settings panel is displayed. You can also edit the settings by clicking the component's "gear" icon.

As you'd expect, the settings will be different for different components. The settings for the **Link** component are shown below:

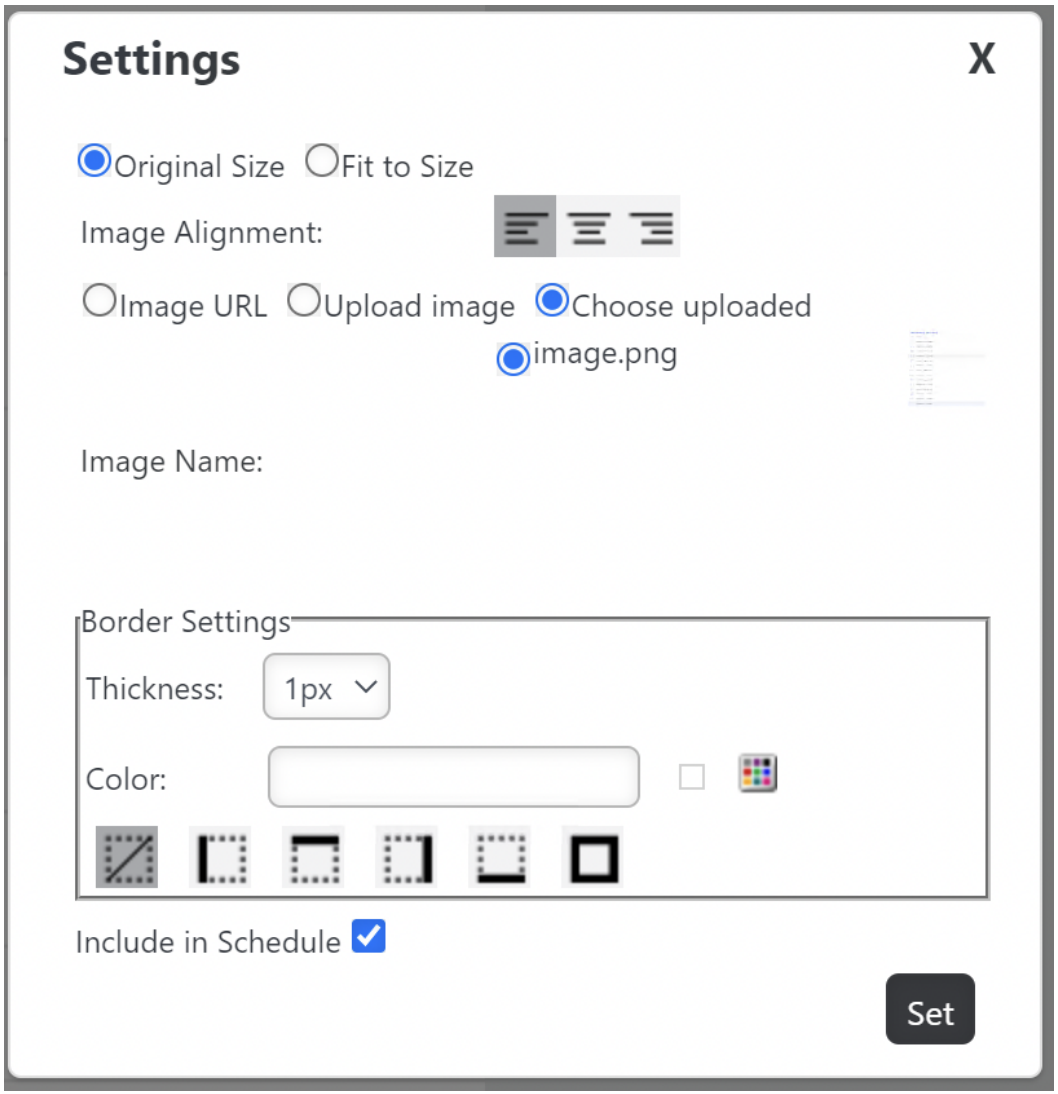



Un-check the "Include in Schedule" check box to remove an object from the final scheduled output.

The **Text** component has a similar Settings panel. However, for quick, text-only changes, users can edit the text "in place". They just hover their mouse over the right end of the component, as shown above, to see its special "pencil" icon, then click it to edit the text.




The **Image** component Settings panel, shown below, lets users include images using a URL or by uploading them. They can also select them from a list of uploaded images stored in a designated folder.




The **PDF Link** component's Settings panel (not shown) includes a **Java AutoFit** control. Set it to *False* when exporting to PDF in Java applications in order to turn off AutoFit and center align text.

Un-check the "Include in Schedule" check box to remove an object from the final scheduled output.

# Report Author Attributes

The Report Author element has the following attributes:

| Attribute | Description |
|---|---|
| Gallery File | (Required) Specifies the fully-qualified path and file name for the XML data Gallery file. This file contains the charts and tables that can be added when the **New Visual** component is dropped on the report canvas. The file is created using an Analysis Grid's "Add to Dashboard" feature, but a Dashboard is not required. The path must be within the application root folder.<br><br>If Logi Security has been used to control access, 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:<br><br>`@Function.AppPhysicalPath~\GalleryFiles\@Function.UserName~.xml`<br><br>which might translate to:<br><br>`C:\inetpub\wwwroot\MyReportApp\GalleryFiles\BGates.xml`<br><br>Gallery files can also be stored in a database, see *Bookmarks*. |
| ID | (Required) Specifies a unique element ID. |
| Save File | (Required) Specifies the fully-qualified file path and name of an XML "Save" file where the report design changes users make at runtime are saved. Tokens may be used in this attribute, however, @Request tokens are *not* recommended as they're not persistent. If the file doesn't exist it will be created at first use; however |

| Attribute | Description |
|---|---|
| | any folder containing it *will not* be automatically created - you must create it manually. The path must be within the application root folder.<br><br>Do *not* set this attribute when using the **Auto Bookmark** element in conjunction with Report Author.<br><br>If Logi Security has been used to control access, the current user's name will be available in the @Function.UserName~ token. This can be embedded into this attribute to provide separate Save files for individual users. For example, the attribute value might be:<br><br>`@Function.AppPhysicalPath~\SaveFiles\@Function.UserName~.xml`<br><br>which might translate to:<br><br>`C:\inetpub\wwwroot\MyReportApp\SaveFiles\MZuckerberg.xml`<br><br>Save files can also be stored in a database, see *Bookmarks*. |
| Disable Gallery Updates | Specifies whether users can delete visuals at runtime from the gallery specified above in the Gallery File attribute. The default value is *True*. |
| Gallery Caption | Specifies a caption for the gallery specified above in the Gallery File attribute. This caption appears in the list of galleries in the Visual Gallery panel. |
| Report | Specifies the display mode and editing options for the report. Options include: |

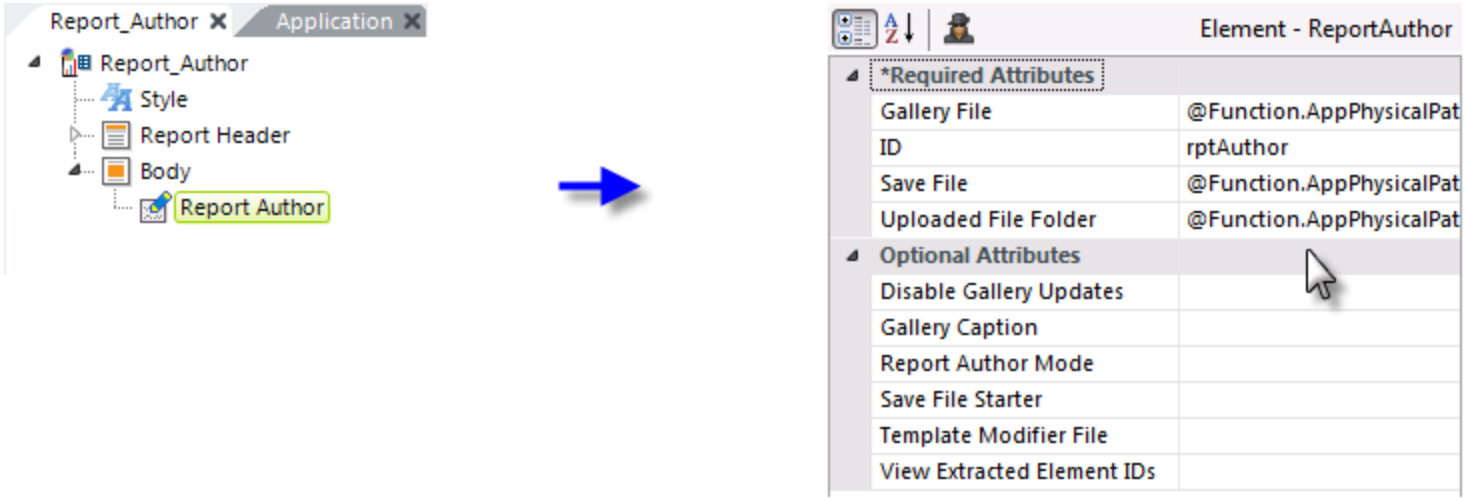
| Attribute | Description |
|---|---|
| Author Mode | *ViewEdit* - Initially displays the report in View mode. Users can click **Design** to switch to Design mode. This is the default mode for existing/bookmarked reports.<br><br>*DesignEdit* - Initially displays the report in Design mode. Users can click **View** to switch to View mode. This is the default mode for new reports.<br><br>*ViewContained* - Shows the report in View mode, without the ability to switch to Design mode. An existing Save file or bookmark is required. Other elements above and below the Report Author element, such as Report Headers and Report Footers, are also included in the report display.<br><br>*ViewExtracted* - Shows the report in View mode, without the ability to switch to Design mode. An existing Save file or bookmark is required (see *Bookmarks*). Only the Report Author report is displayed, all other elements in the report are suppressed. This mode is automatically applied to reports downloaded using the **PDF Link** component. |
| Save File Starter | Specifies the fully-qualified file path and name of an XML data file that will provide a default report design. It will be used only when the Save File does not exist. For example, when the Save File is different for each user, the Save File Starter file can provide a set of default components.<br><br>Create a Save File Starter file by first setting a Save File value, then running the application and creating a report that includes the desired default components. Then move and/or rename the Save File into the Save File Starter location and set this attribute. For example: |

| Attribute | Description |
|---|---|
| | `@Function.AppPhysicalPath~\SaveFiles\DefaultDesign.xml` |
| Template Modifier File | Specifies the name of an optional Template Modifier File, used to programmatically modify the Report Author super-element UI or behavior at runtime. See "Customizing Report Author Appearance" on page 69 for more information about template modifiers. You can provide a fully-qualified file path and name for the file, within the application root folder or, if a fully-qualified file path is not provided, the application expects the file to be in its `_SupportFiles` folder. |
| Uploaded File Folder | (Required prior to v12.2.116 SP3) Specifies the fully-qualified file path and name of a *folder*, within the application root folder, where uploaded images will be stored. These are optionally uploaded from the Images component Settings panel, and uploaded files appear in the list of available images in that panel.<br><br>If Logi Security has been used to control access, the current user's name will be available in the @Function.UserName~ token. This can be embedded into this attribute to provide separate image folders for individual users. For example, the attribute value might be:<br><br>`@Function.AppPhysicalPath~\ReportImages\@Function.UserName~`<br><br>which might translate to:<br><br>`C:\inetpub\wwwroot\MyReportApp\ReportImages\MMayer`<br><br>v12.2.116 SP3 - Uploaded files can be stored *outside* the application root folder. This requires setting a path in the _Settings definition's General element's Report Author Upload Folder Override attribute, which overrides |

| Attribute | Description |
| --- | --- |
| | this attribute.<br><br>Uploaded files can also be stored in a database, see *Storing Bookmark, Gallery, and Save Files in a Database*. |
| View Extracted Element ID | Specifies one or more element IDs, in a comma-separated list, of elements that may be necessary for the proper functioning of the report, such as Definition Modifier File or Plugin Call. These are stored in the report's bookmark and included when the report is generated. |

# Using the Report Author

Using the Report Author element is very easy:




As shown above, add the Report Author element to your report definition. Examples of the attribute values are:

- *Gallery File* = `@Function.AppPhysicalPath~\GalleryFiles\RAGallery.xml`
- *Save File* = `@Function.AppPhysicalPath~\GalleryFiles\RASaveFile.xml`
- *Uploaded File Folder* = `@Function.AppPhysicalPath~\GalleryFiles\images`

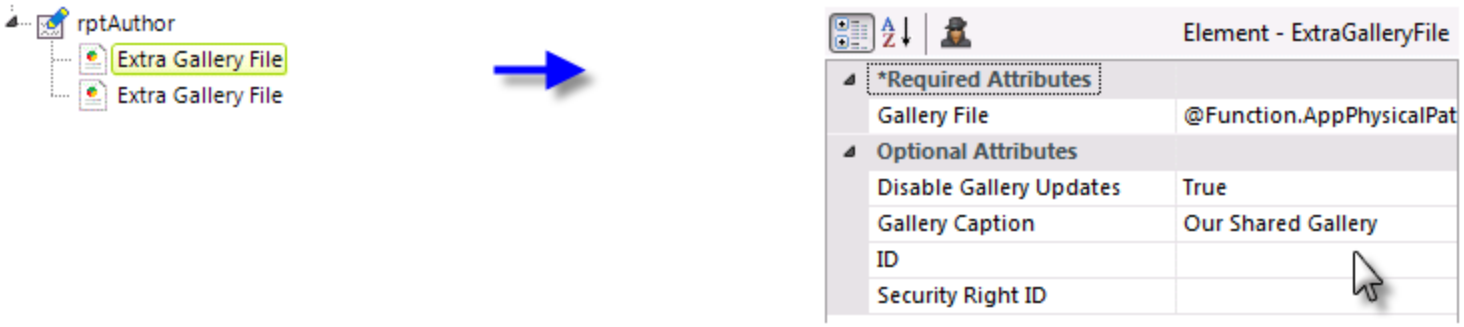
As described in "Report Author Attributes" on page 61, if Logi Security is in use, you could further differentiate the files and folders using the `@Function.UserName~` token.

That's all you need to do. The Report Author element provides a huge amount of functionality with very little developer effort.

# Extra Gallery Files

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

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 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. Multiple Extra Gallery File elements may be used.

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.

Gallery files can also be stored in a database, see *Storing Bookmark, Gallery, and Save Files in a Database*.

# Customizing Report Author Appearance

Report Author 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 *Using the Theme Editor* tool.

## Changing Appearance Using Style Classes

Report Author appearance can be also customized using **style classes** and Info Studio provides the following standard style sheet:

```
<YourAppFolder>\rdTemplate\rdReportAuthor\reportAuthor.css
```

Developers can override classes in this style sheet by *copying* them to their application style sheet and modifying them there.

 Do not make changes in the standard style sheet in the rdTemplate folder.

## Changing Appearance Using Template Modifiers

The Report Author 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 Report Author 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.

The appearance of the Visual Gallery pop-up panel can also be modified using a Template Modifier File.

For example, the Report Author template file:

```
<yourAppFolder>\rdTemplate\rdReportAuthor\ReportAuthorTemplate.lgx
```

contains several Label elements. One of them has an ID = "lblSwitchToDesignModeTrue"; this controls one of the Design button captions. It can be modified by changing the Caption associated with that Label element. To change the caption from its default "Design" to "Create", create your own XML file, identify it in the Template Modifier File attribute, and add this code to it:

```
<TemplateModifier>
 <SetAttribute ID="lblSwitchToDesignModeTrue" Caption="Create" />
</TemplateModifier>
```

You can set the attributes for any number of elements in this file; examine the `ReportAuthorTemplate.lgx` file to learn the ID and Caption attributes available. Other template files exist in the same folder for the Settings panels.

The template modifier file can be in any folder accessible to the 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 its `_SupportFiles` folder.

More detailed information about template modifier files can be found in *Template Modifier Files*.

# Master Reports

A **Master Report** is a special report definition that includes elements that are to be included in other reports. It's frequently used to provide a template-like frame, often defined with a header, a menu, and a footer, for integration with other reports.

The following topics discuss Master Reports:

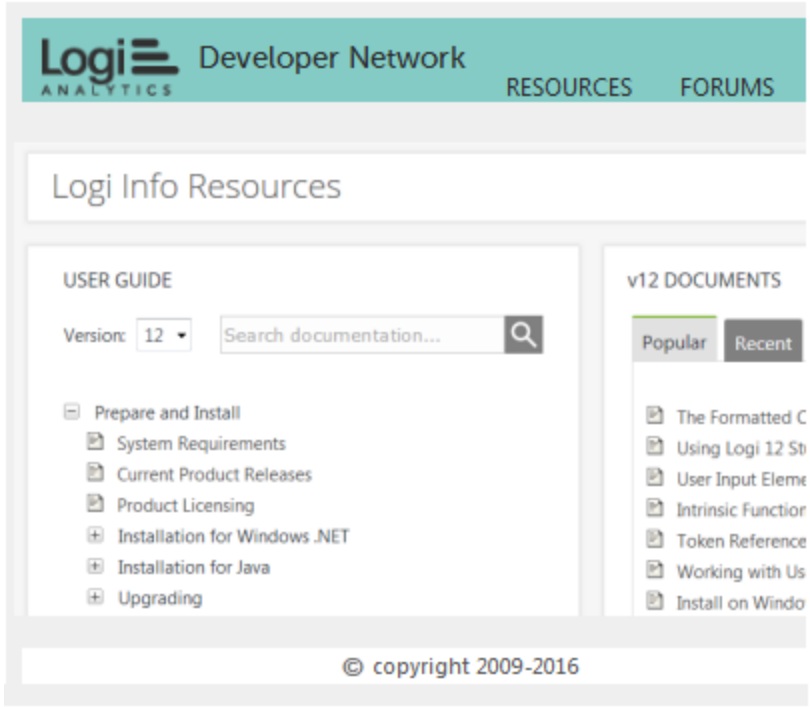
- Master Report Elements
- Creating a Master Report
- Customizing the Master Report

## About Master Reports

The **Master Report** concept, introduced in Logi Info v12.1, extends the idea of *Shared Elements* by providing a special report that contains elements that are to be included in other reports. One common use is to define a Master Report with a header, a horizontal or left sidebar menu, and a footer. Regular report definitions (the "other" reports) then reference the Master Report and, at runtime, the two are merged together.


The images above show two typical layouts generated by the Master Report Layout wizard in Logi Studio. On the left, the Master Report includes a header with a logo image and an optional horizontal menu, and a footer. On the right, it includes a header with a "navicon" or "hamburger" icon and a logo image, and a footer. Clicking the icon toggles the visibility of a sidebar with an optional vertical menu.

A Master Report is a regular Logi Info report definition, so you can customize it as much as you'd like. Styling for the common elements (header, menu, footer) are included in standard Logi Info themes and can be edited using the *Using the Theme Editor*.



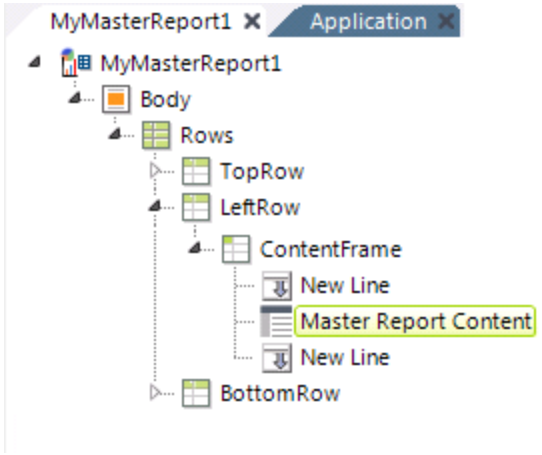


For example, if DevNet was re-created using the two Master Report layouts we saw previously, it might look like one of the images shown above.

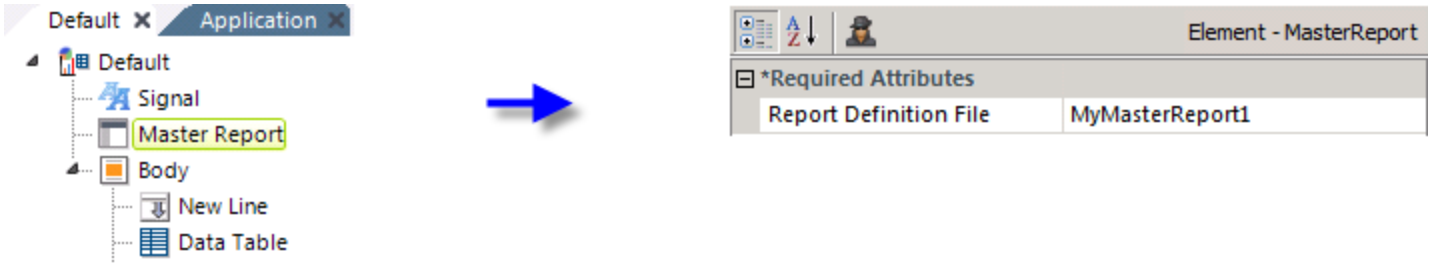
However, a Master Report does not always have to be used to provide a header and footer; it can be used to add any elements or structures to other reports.

# Master Report Elements

A **Master Report** is a regular Report definition that can includeany elements you feel are required.



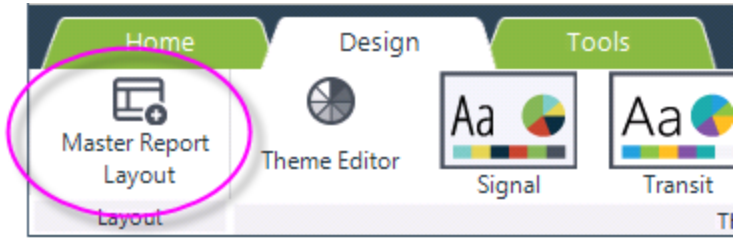

The only element it *must* have is the **Master Report Content** element, as shown in the example above. This is the "placeholder" for the content from other report definitions. It tells the Logi Server Engine where to insert the other definitions at runtime. This element has no attributes and only one of them is allowed in a Master Report definition.



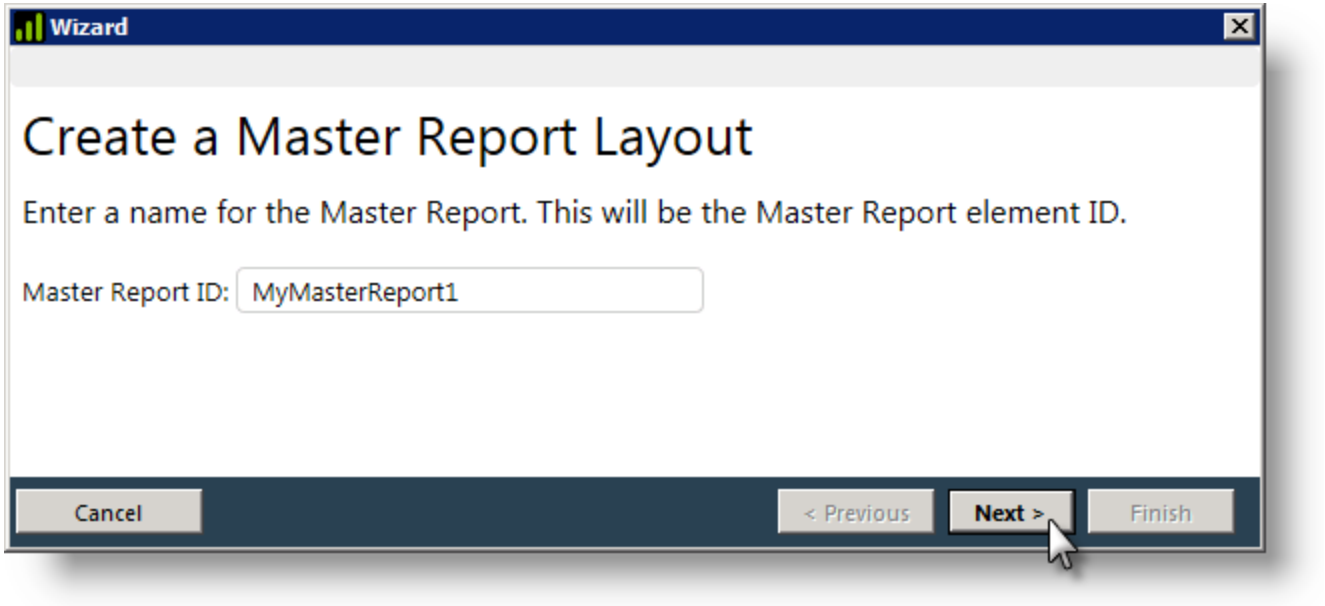
In the other report definitions in your application, add a **Master Report** element, as shown above. Set its **Report Definition File** attribute value to the name of your Master Report definition (tokens may *not* be used here). This will cause the integration of the definitions at runtime.

# Creating a Master Report

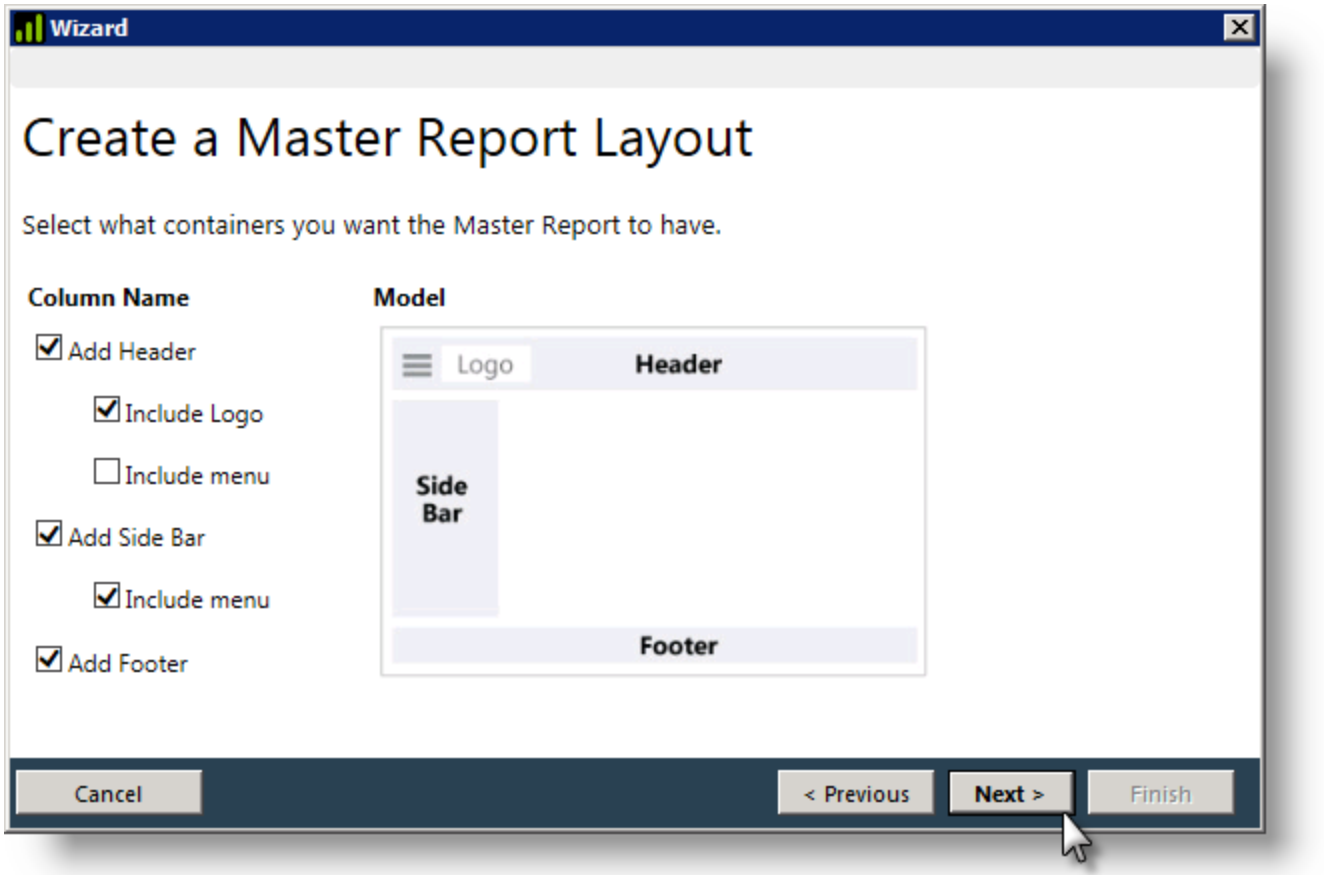
The easiest way to create a Master Report is to use a special wizard in Logi Studio:




You can launch the wizard from Logi Studio's main menu Design tab, as shown above.



The wizard will first prompt you for a name for the Master Report definition, as shown above. When created, the definition will appear in Studio in the Application Panel along with all your other Report definitions. Click **Next** to continue.




Next, you'll be asked to select the structure of the Master Report, as shown above. If you select the sidebar, an icon for toggling its visibility will appear in the header. Click **Next** to continue, then **Finish** in the next dialog box.

The wizard will generate and open a new definition, with all of the elements required for the layout you selected, and will include a Master Report Content element.

If you have a Report definition open in the editor, the wizard will automatically insert a Master Report element, configured for your new Master Report, into it.

If you prefer, you can create a Master Report definition from scratch, without using the wizard.

# Customizing the Master Report

The definition generated by Studio's Master Report Layout wizard includes a number of elements that you can customize.

For example, in the header you can specify your own logo image and title. You can copy and paste Menu Branch and Menu Leaf elements as often as needed to complete your menu. And you can place your own message in the footer.

Of course, you're not limited to working with just the elements and styling that the wizard generates - you can tailor the definition to meet your needs.

## Using Tokens in a Master Report

Due to the way that the Logi Server Engine combines a Master Report definition and the other report definition, you can include certain tokens in the Master Report representing data from the other report.

For example, if the other report is called with a URL that includes Request variables, then you can access those values using @Request tokens in the Master Report definition. As such, there's no way to, nor need to, use elements to "pass" parameters to the Master Report.

Similarly, if the other report uses **Local Data** to retrieve data, then you can use @Local tokens to access those values in the Master Report definition.

Tokens with universal scope, like @Session, @Function, @Constant, and others, can also be used in the Master Report definition and will be evaluated properly.

## Styling a Master Report

If you want to use a theme with your application we generally recommend that you apply it in your application's _Settings definition or in the other reports, not in a Master Report. If a Master Report and the other report definition are assigned *different* standard themes, the other report's theme will be used.

You *can* include a style sheet in a Master Report to style it, but be aware that the styling is not restricted to the Master Report. Due to the way the definitions are combined at runtime, unless overridden by a class in a theme assigned to the other report, classes in the Master Report could affect content in the other report.

## Other Considerations

You may use **User Input** elements in a Master Report and their values will be passed as Request variables, as usual, when the combined reports are submitted.

Master Reports are executed every time another report that references them is loaded or re-loaded so, for best performance, you probably *do not* want to place long-running queries in them.

# Logi Mobile Reports

Smart phones and tablets are creating a big demand for access to business intelligence on hand-held devices. Logi Info developers can leverage their experience with Logi reporting, using special Logi **Mobile Report** definitions, to satisfy this demand.

The following topics introduce the concepts, Studio features, and special elements available for developing mobile reports:

- Mobile Reporting Features in Logi Studio
- Elements: Additions and Restrictions
- Performance Considerations

## About Logi Mobile Reports

Logi Mobile Reports bring business intelligence reporting to mobile devices. This is achieved by generating web pages that have been optimized for the mobile device environment.

Logi Mobile Reports are *not* compiled native device apps; they're pages displayed within the device's browser. Developers design and code Mobile Report apps in Logi Studio, using many of the same techniques used to develop standard Logi reports. An experienced Logi developer already has most of the skills needed to develop a Mobile Report application.

Mobile Reports can include most standard Logi application features, such as charts, tables, tabs, dynamic pages, drill-downs, scrolling, etc. A subset of the standard collection of elements can be used, along with some specialized elements created just for this type of report. Studio automatically provides the mobile-ready elements when you're working in a Mobile Report definition.

Animated Charts, based on JavaScript, are available in Mobile Reports. Mobile Reports can be displayed on devices that include full-featured browsers, such as:

- iPhone and iPod Touch
- Android devices
- Blackberry devices
- Windows Phone 7+ devices
- iPad and Android tablets (these however may be better candidates for regular Logi reports)

Obviously, the mobile device environment and technology differs from that of a regular browser on a PC. Therefore, developers should be aware of the following considerations.

### Regular or Mobile?

Developers may want to create mobile versions of existing regular reports, and give them both the same name. How will the Logi Server Engine know which report to provide when the request is received? It's likely in the future that the product will include some form of "browser detection" that would allow this to happen automatically, but in the near term there are several things you can do:

- **Use a Query string flag** - If the query string of the Logi report request includes the argument "`rdMobile=True`" then the engine will respond with the mobile report rather than the regular report.
- **Use separate reports, apps, or domains** - Use different URLs for regular and mobile reporting. For example, give mobile reports slightly different names, such as prefixing them with "m", or create different Logi applications for each type of reporting, or use sub-domains, such as a `mobile.mySite.com` instead of `www.mySite.com`.
- **Give your users a choice** - It's a common practice to place a "Show Full Version" link in the mobile version of a report. This provides an option to break-out of the mobile view into a more standard view of the same content. This can also be done in reverse.

- **Use script-based detection or other methods** - You could insert some JavaScript in your reports to manage browser detection and redirection to mobile versions of reports. This can be tricky and may be somewhat unreliable due to a lack of consistency in the information reported by different devices, and some mobile browsers even disable JavaScript, but it might work for your situation. In addition, there are third-party tools available on the Internet: search for "mobile browser detection".

### Screen Real Estate

The size of a mobile device screen is obviously smaller and has a different aspect ratio than that of a traditional computer monitor. Therefore, there are a number of report design considerations related to the amount of screen real estate available, such as the readability of the material being displayed, font and image sizes, etc.

In addition, different mobile devices have different screen sizes and developers may need to design for all device possibilities. Logi Studio provides a very useful feature for previewing Mobile Reports in a variety of device screen sizes (discussed in the next section).

Most devices also change their display size and orientation when the device is rotated from "portrait" to "landscape", and developers may want to provide application behavior that can take advantage of this.

Logi Info provides *Responsive Design Elements* that can be used to dynamically alter report layout and component visibility, depending on the device screen size.

### User Interface

Mobile device browsers support HTML5, the latest extension of the HTML standard most web developers are familiar with, and CSS3. In addition, many mobile devices bind user input controls displayed in their browser into controls in their operating system.

This causes the "standard" HTML controls, such as a select list, to be displayed in a different manner and perhaps differently in different devices.

In many cases, Logi developers don't need to do anything: standard Logi user input elements will be automatically converted into a device's specialized input representation.


PC Internet Explorer


iPhone Safari

The examples above show the Logi Input Select List element in use. When *clicked* in a PC browser, it appears as a drop-down list of choices, but when *tapped* in an iPhone, it's automatically presented as a "Picker" control.

Logi Studio also provides some special user input elements for Mobile Reports, including **Input Email** and **Input Telephone**, which are described in "Elements: Additions and Restrictions" on page 93.

Developers can refer to a number of interface design guides available from the manufacturers of mobile device operating systems for guidance and best practices:

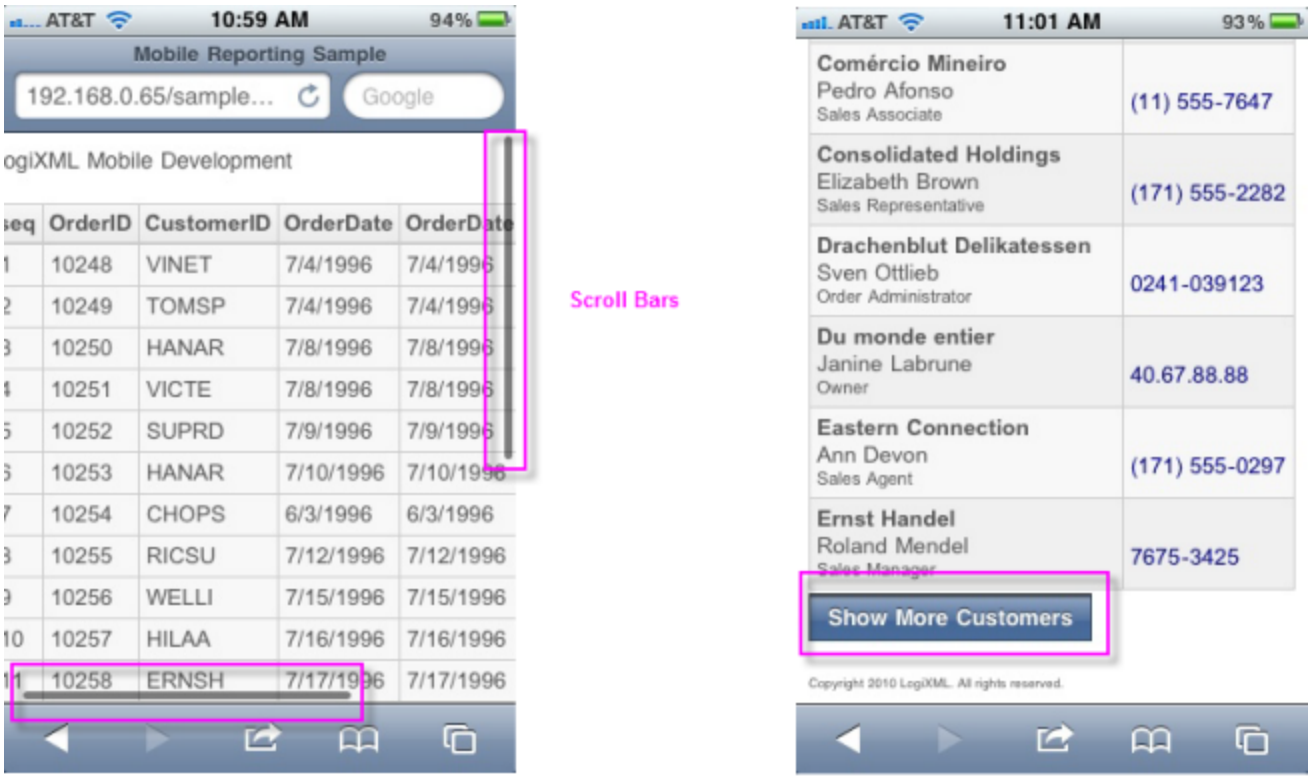
Apple: *iOS Human Interface Guidelines*
Google: *Android User Interface Guidelines*
Microsoft: *Guidelines for UWP Apps*

## General Navigation

Mobile device browsers enable user interaction with touch gestures, such as tap, double-tap, flick, wipe, touch 'n' drag, and pinch. In Logi Mobile Reports, objects such as links are *tapped* instead of *clicked*. Action elements and drill-down links that redirect to other Mobile Report definitions will behave as usual. Developers should take care that objects such as links are in a large enough font, and positioned in such a way, that recognizing them and tapping them is easy. Crowding links together, for example, might make it difficult to tap them without tapping their neighbors, too.

## Scrolling and Paging

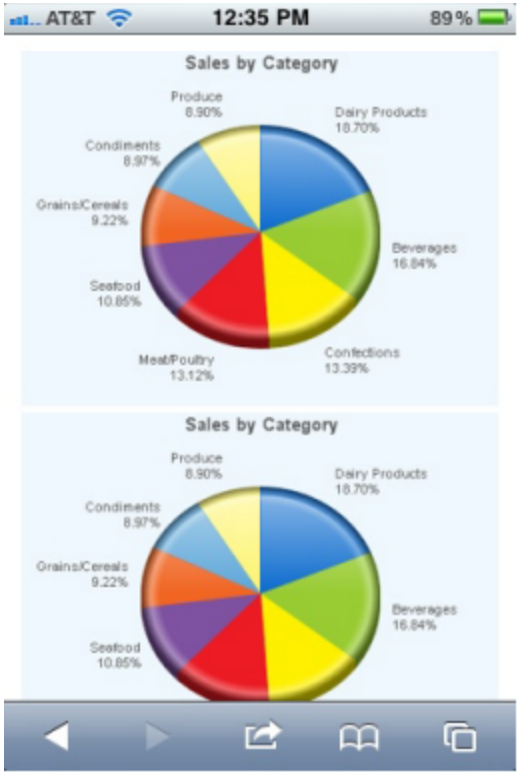
Due to the relatively limited amount of screen space on a mobile device, developers need to design Data Tables and charts so that they're easily viewed. Displaying thousands of rows in a Data Table, for example, is potentially going to require *a lot* of scrolling by the user, which is not desirable.




Data Tables, for example, can be scrolled vertically and horizontally on mobile devices by flicking or wiping the screen and by dragging the scroll bars, as shown above left, that appear when the table is tapped. A special element, **Append Paging**, is available for use with Data Tables; it provides vertical paging and the "show more" button, as shown above right. This replaces the functionality of the Interactive Paging element, whose links are deemed to be too small for mobile device use.

## Chart Sizing

Sizing charts for mobile devices presents a challenge, especially if a Mobile Report will be viewed on different devices. A special element, the **Auto Sizer**, is available to assist with this: it automatically expands or shrinks a chart to fit the available space.





As shown in the examples above, the Auto Sizer also resizes the chart automatically when the device orientation changes.

## Themes

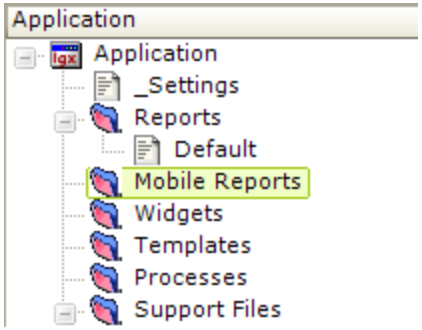
The standard Themes provided in Studio for use with Logi applications have been extended for use in Mobile Reports. As usual, developers can also create their own custom themes. Themes include a special style sheet file called `MobileTheme.css`, which is active only for Mobile Reports and which makes fonts a little larger, increases padding so that to it's easier to tap on links, etc., to generally improve the user experience.

## Security

All of the regular Logi security models are supported in Mobile Reports.

# Mobile Reporting Features in Logi Studio

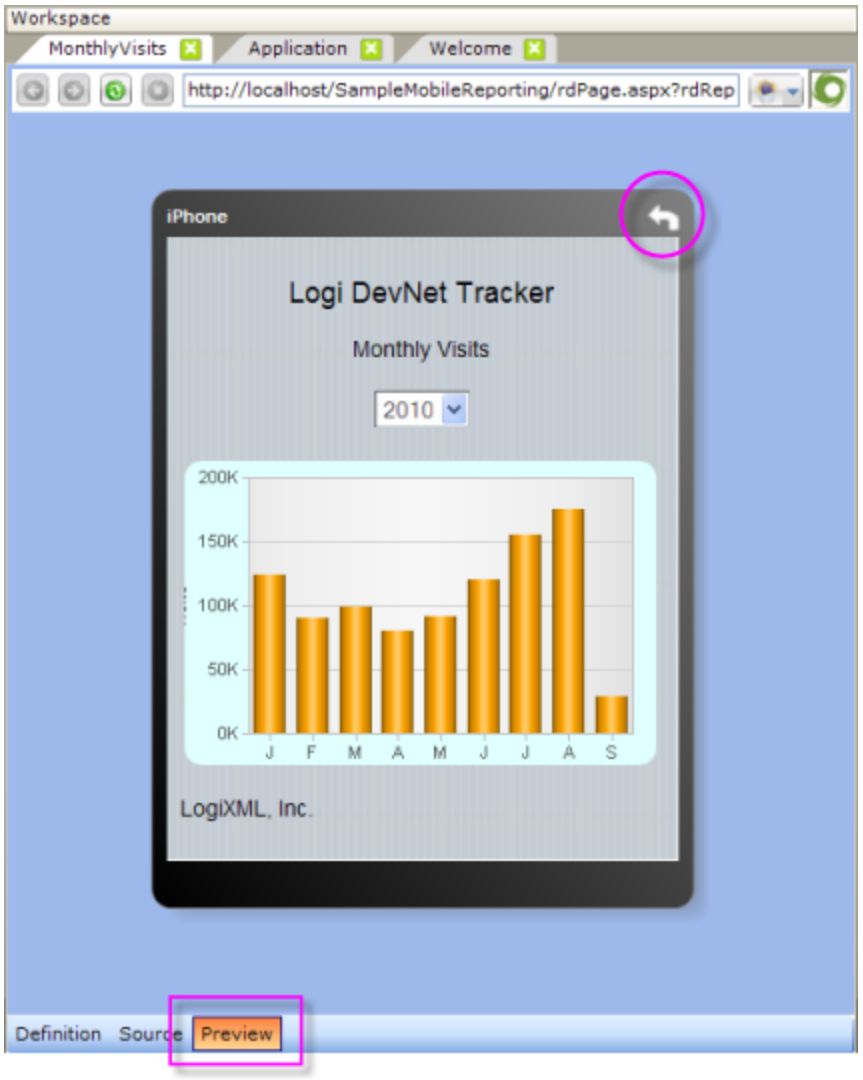
Logi Studio includes a number of unique features for Mobile Report definitions, beginning with a special type of report definition:



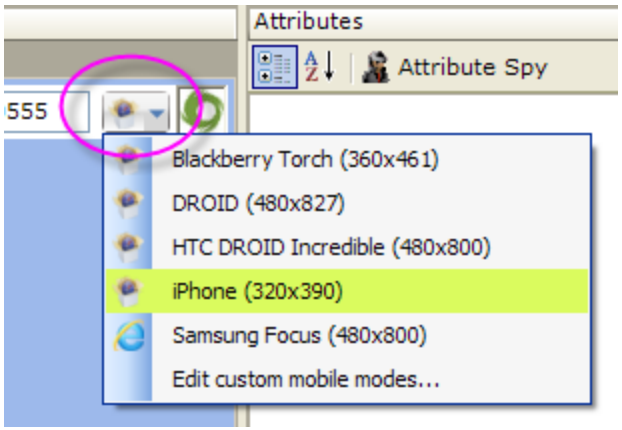

The Application panel includes a folder, **Mobile Reports**, as shown above. This folder can be right-clicked to create a new Mobile Report definition. When working with a Mobile Report definition, special elements will become available for use and certain other elements, which are not suitable for mobile reports, will be unavailable.

## Preview Masks

In order to allow developers to see their work in the proper context, Studio includes a special preview mode for mobile reports:




As shown above, when the Preview button is pressed, a Mobile Report definition will be displayed in the Workspace panel within a mask that approximates the size of the target mobile device, using the appropriate browser for that device. The mask can be rotated to landscape orientation by clicking the white arrow on the mask frame.




Different device preview masks can be selected from a list (click the button next to the Preview URL); selecting a different mask immediately changes the displayed report preview. Standard masks are provided and developers can add custom masks as well.


These preview masks are only geographic simulators; they show you how your report will look in the available space. As shown above, they do not display any onscreen buttons or status bars that may normally appear above and below the space on the device screen. The real device OS is not being loaded, so Logi user input elements are not bound to OS controls; recalling the example in the previous section, this means no "Picker" control will appear if you click the Input Select List in the preview.

Full-fledged mobile device emulators are not included because they would present some operational problems, not the least of which would be the relatively long amount of time it would take to load and start them when the Preview button was clicked.

# Elements: Additions and Restrictions

The following special elements have been added for exclusive use with Mobile Report definitions:

| Element | Description |
|---|---|
| Action.Dial Phone | Launches the device's phone app and feeds it the number in its Phone Number attribute, if any. The call is not placed until the user clicks the app's "Call" button. |
| Action.Draft Email | Launches the device's email app and feeds it the addresses, subject, and body text, if any, from this element's attributes. The message is not sent until the user clicks the app's "Send" button. |
| Action.Draft Text Message | Launches the device's SMS text messaging app and feeds it the number in its Phone Number attribute, if any. The message is not sent until the user clicks the app's "Send" button. |
| Action.Map Location | Runs a Google Maps query to show a map with one or more locations highlighted. Locations can be specified in its attributes with either Place Query or specific Latitude and Longitude coordinates. Assumes the mobile device has mapping capabilities. |
| Append Paging | Used to paginate a Data Table. Starts by displaying a specified number of rows and a button below the table. Clicking the button displays the next set of rows, beneath the previous set of rows. The button may be clicked repeatedly until all rows are visible, at which time the button will no longer appear. |
| Auto Sizer | Auto Sizer automatically sets the width of a chart so that it fits the full width of its container. This is very useful for making charts as large as possible, within the browser dimensions, and it also resizes the chart when the device orientation changes. |

| Element | Description |
|---|---|
| Data Menu | Creates a menu, and optional sub-menus, for use on a mobile device. Information used to construct the menu (caption, image, target, etc.) is taken from a datalayer at runtime. |
| Input.Email | Allows entry of an email address. On some mobile devices, the screen keyboard is configured specifically for entering email addresses. A comma- or semi-colon-separated list of multiple email addresses can be entered. |
| Input.Number | Allows entry of a number. On some mobile devices, the screen keyboard is configured specifically for entering numbers. |
| Input.Telephone | Allows entry of a telephone number. On some mobile devices, the screen keyboard is configured specifically for entering phone numbers. |
| Mobile Dashboard | Allows mobile report developers to "share" a Dashboard created in a regular Logi report definition, using the Dashboard element. The Mobile Dashboard applies special formatting and functionality to the visualizations in the existing regular Dashboard. |
| Validation.Email | Ensures that a valid Email Address has been entered, using Input.Email. The address is validated for the "@" symbol and domain name. Certain special characters are not allowed. If the validation does not pass, an error message will be displayed in a message box. Acomma- or semi-colon-delimited list of multiple email addresses can be entered. |

A subset of the standard elements is currently available for Mobile Reports. As development of Mobile Reports progresses in the product, more elements will become available.

Certain attributes for some elements, such as Tooltips and target Frame IDs, make no sense in the mobile device context and are unavailable for Mobile Reports.

# Performance Considerations

For a variety of reasons, connectivity to mobile devices may be more sensitive to bandwidth limitations and performance issues than a PC. Developers who wish to improve performance have several measures they may care to use:

Actual transmission performance can be enhanced using HTTP compression to reduce network bandwidth usage. Administrators can configure their web servers so that application and/or file output is compressed. Microsoft offers this information about enabling compression on IIS 7, and numerous sources are available online describing the same for Apache Tomcat.

In addition, there are a number of things developers can do to ensure good Mobile Report performance. For example, limiting the amount of data being delivered to the device in any one request, and breaking complex reports into a number of simpler reports.

There are examples of complete Mobile Report definitions in "Working with Logi Mobile Reports" on the next page.

# Working with Logi Mobile Reports

The following topics provide examples for Logi Info developers who wish to create Mobile Reports:

- Create a New Mobile Report Definition
- Create a Supplemental Stylesheet
- Build the Default Definition
- Build a Data Table Definition
- Build a Chart Definition

## About Logi Mobile Reports

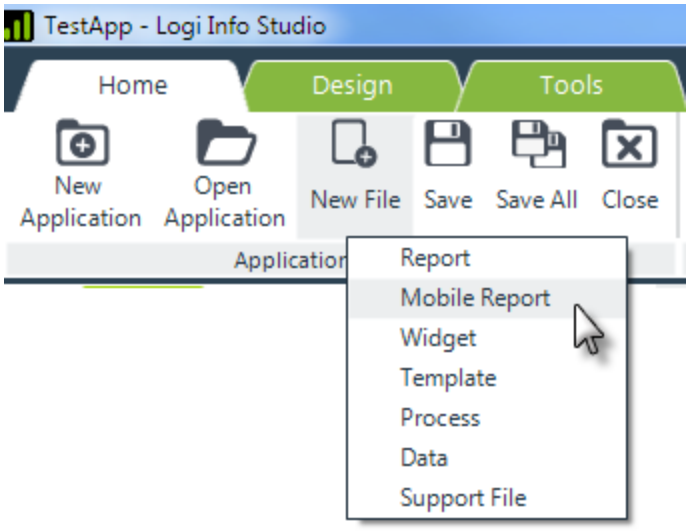
If you haven't done so, we encourage you to read "Logi Mobile Reports" on page 81before proceeding with this topic.

Logi Mobile Reports allow developers to create reports that are to be viewed on a mobile device. This is achieved by generating web pages that have been optimized for the mobile device environment and that run on the "mobile web". Developers design and code Mobile Report definitions in Logi Studio, using many of the same techniques used to develop standard Logi reports.

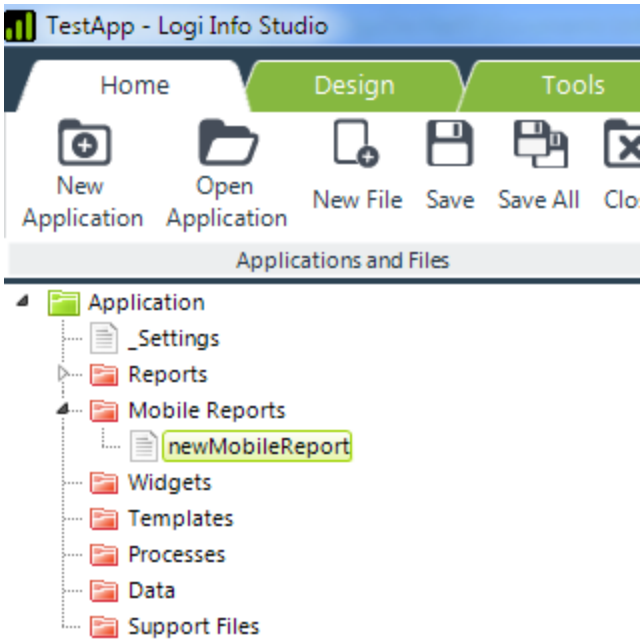
The menu in the following examples is built based on a Data Table. In a separate topic, *Data Menu*, you can learn how to use the Data Menu element to create more complicated mobile device-specific menus.

# Create a New Mobile Report Definition

Creating a new Mobile Report definition is similar to creating a regular report definition:




1. In Logi Studio's main menu, in the Home tab, click the **New File** menu item.
2. Select the **Mobile Report** item in the pop-up menu, as shown above.




3. The new definition will be created, with a temporary name, and opened in the Workspace editor.
4. Rename the new definition, as desired. We'll use "mDefault".

You're almost ready to build your first Mobile Report definition. But first, let's create a special style sheet.

# Create a Supplemental Stylesheet

This topic demonstrates how to create a supplemental stylesheet that will be used along with a theme and will help control the main menu and Data Table appearance. This is not an absolutely necessary step and, for your mobile device, some adjustments may be necessary to achieve the desired look. These classes will work fine for an iPhone.

1. Select and right-click the **Support Files** folder in Studio's Application Panel.
2. Select **Add** and **New File...** from the pop-up menus.
3. In the dialog box that appears, select **Stylesheet** and click **OK**.
4. The new stylesheet file will be created, with a temporary name, and opened in the Workspace editor.
5. Rename the new file, as desired. We'll use "mobileStyles.css".
6. Copy the text below into your stylesheet file:

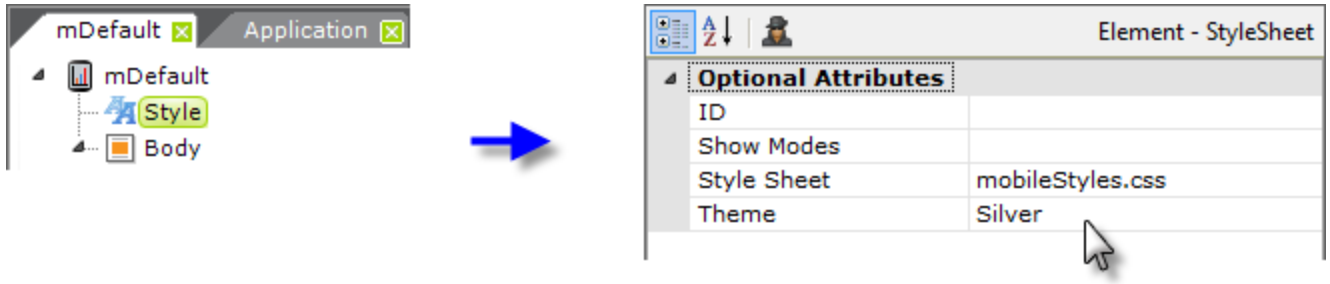
```
#myAppendPaging {padding-top:5px;}
            .infoRow {padding-left:5%; padding-right:5%;
 border-right-width:0px;}
            .menuRow {height:30px;
 padding-left:5%;border-left-width:0px;border-right-width:0px;}
            .mobileTable {margin-left:5%;
 border-width:1px;border-style:solid;border-color:
 Silver;border-collapse:collapse;}
```

7. Save the new stylesheet file and close it.

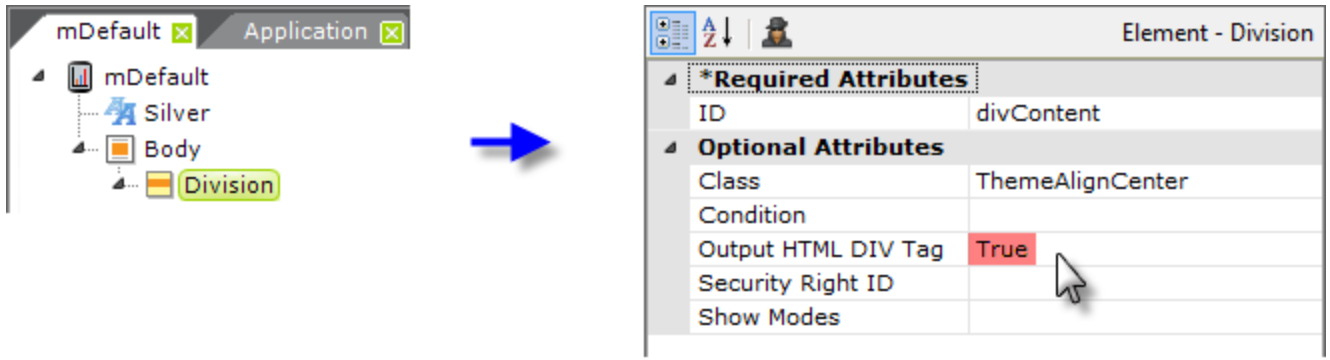
Now we're ready to proceed with building our first Mobile Report definition.

# Build the Default Definition

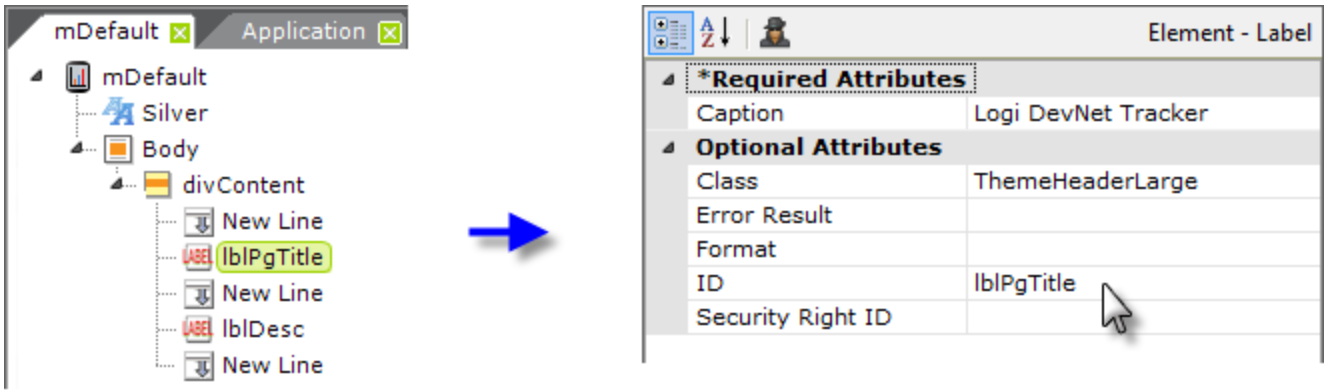
In the Workspace panel editor, select the tab for your new Mobile Report definition.



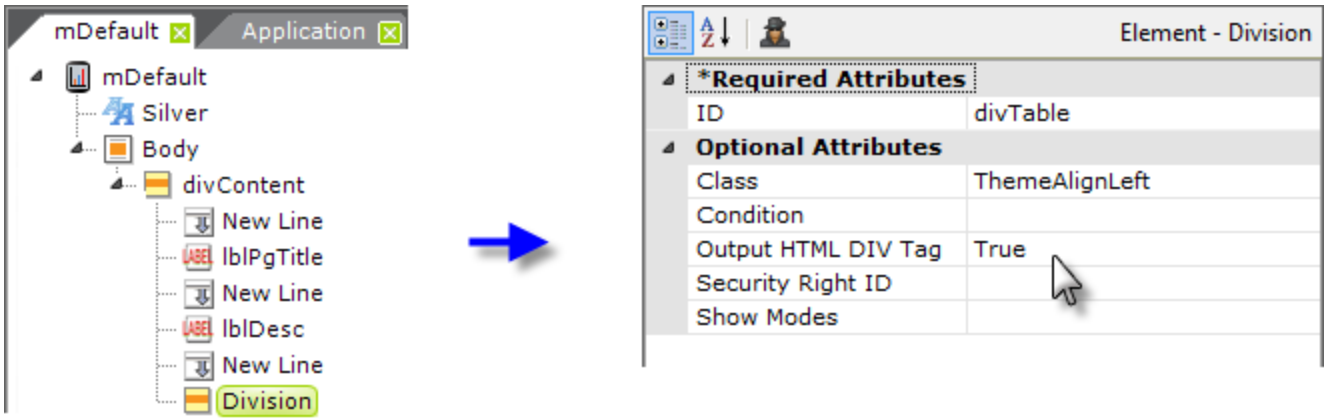

1. Select the **Style** element that was added by default to your definition, and set its **Style Sheet** and **Theme** attributes as shown above.




2. Beneath the Body element add a **Division** element, and set its attributes as shown above. This will ensure that everything displayed on the mobile device is neatly centered. Note that the **Output HTML DIV Tag** attribute has been set to *True*. Without this, the centering will not work.

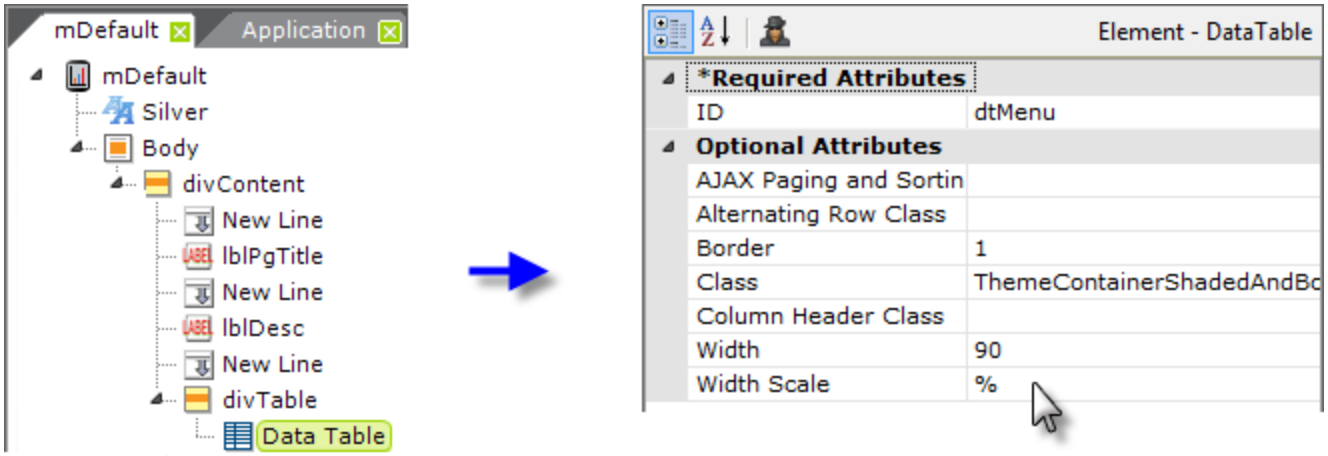



3. Add as many **Label** and **New Line** elements as necessary to put a title and description in the report. As shown above, use the classes from your Theme, available in the pull-down list, to provide consistent font styles for your labels.

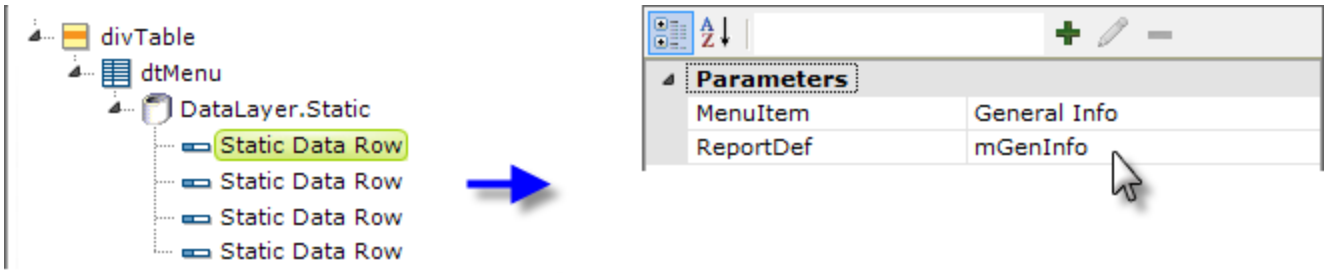



4. We're about to build our main menu, but first we need to add another **Division** element. Without it, our menu items will also be centered because the style of the first Div will be inherited by the Data Table we'll use for our menu. So add another Div

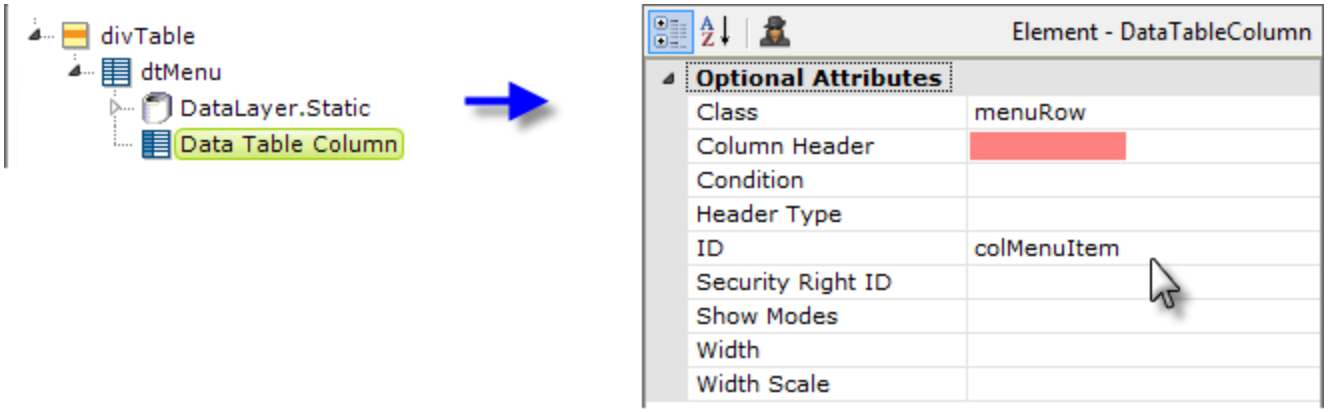
and set its attributes as shown above.



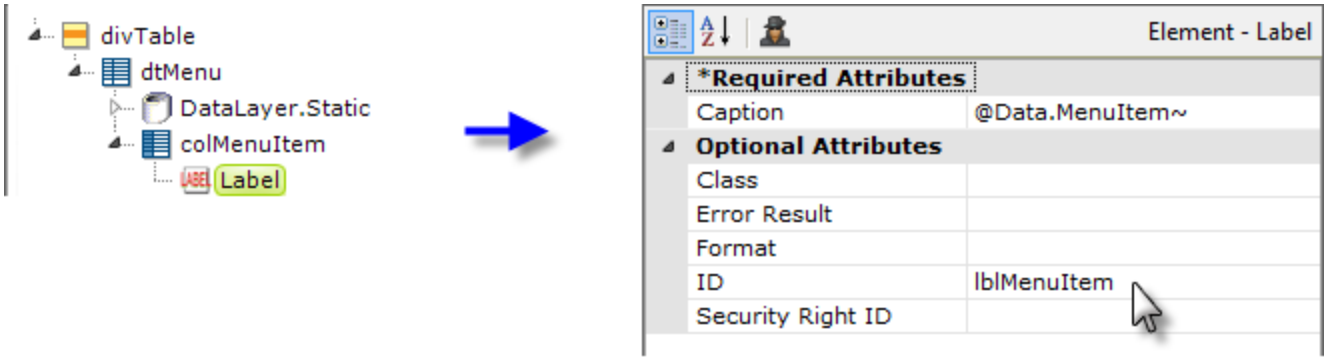

5. Add a **Data Table** element beneath the last Div, and set its attributes as shown above (some blank attributes have been removed from the image to save space). Its complete **Class** attribute value, which combines a theme class and a supplemental style sheet class, is: *ThemeContainerShadedAndBordered,mobileTable* .



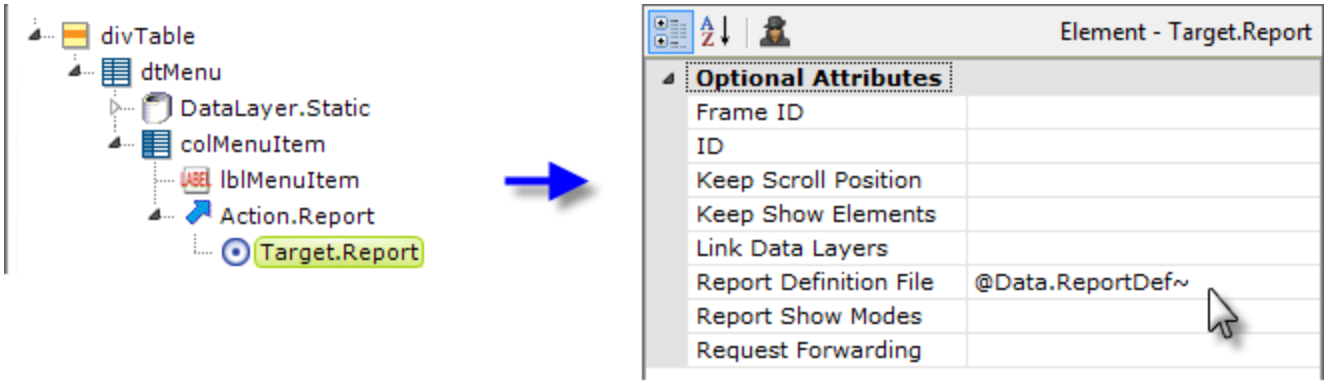
7. Add a **DataLayer.Static** element beneath the table and several **Static Data Row** elements beneath it, as shown above. Of course, any type of datalayer could be used here. Our static data rows will include a column for the menu item text to be displayed and the name of the Mobile Report definition to be called when the menu item is tapped.



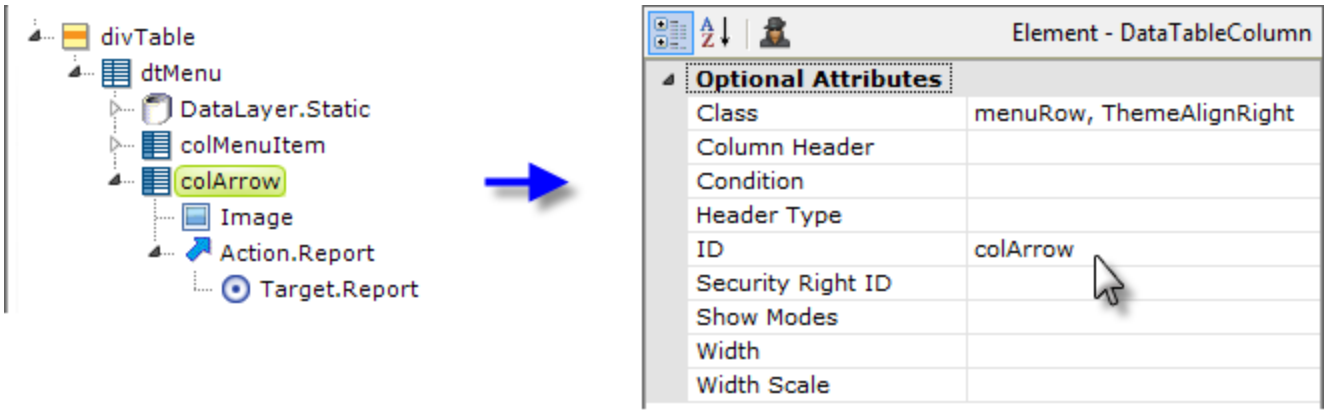

8. Add a **Data Table Column** element beneath the Data Table and set its attributes as shown above. Notice that you're leaving the **Column Header** attribute blank. This will result in no column header being shown at all, which helps our Data Table look more like a list of menu options than a Data Table.



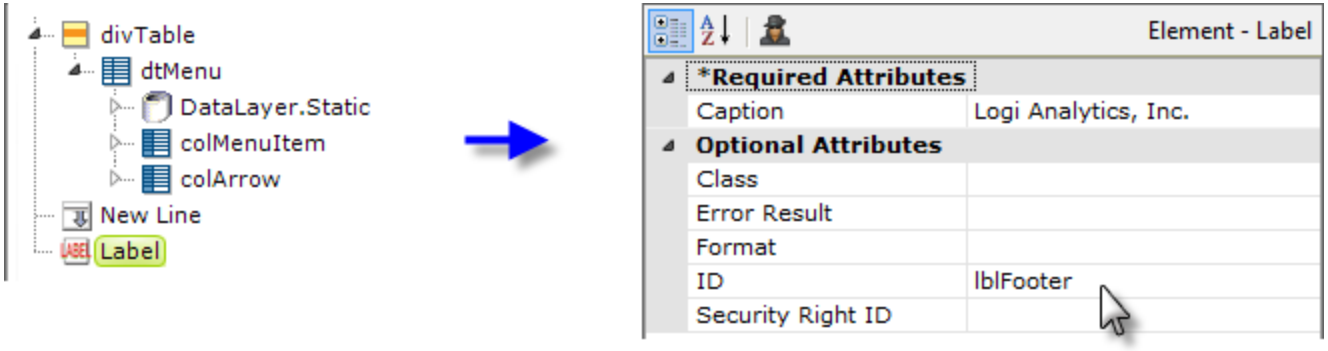
9. Add a **Label** element beneath the Data Table column and set its attributes as shown above. The text for the menu item is drawn from the data in the datalayer.



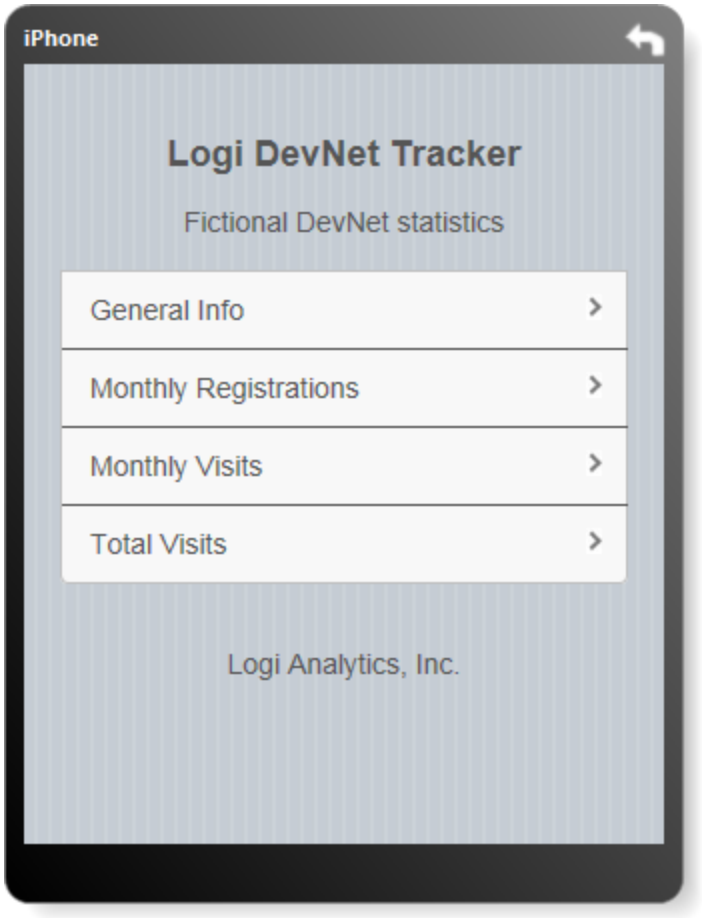

10. Add an **Action.Report** and a **Target.Report** element beneath the **Data Table Column** element (*not* the Label element) and set its attributes as shown above. This is one of those mobile device usability issues: it's easier for the user to tap a large space (anywhere in the Data Table row) rather than a comparatively small text link in the row.



11. Now *copy* the existing Data Table Column element and *paste* it beneath the Data Table, as shown above. Change its element **ID** and modify its **Class** attribute. Replace the Label element beneath it with an **Image** element. Set the Image element's **Caption** to the file name of an image of an arrow that points horizontally to the right.




12. Finish off the definition by adding a New Line and Label element, under the *first* Division element, to center it, as shown above. Put some footer text in the Label element's Caption attribute.

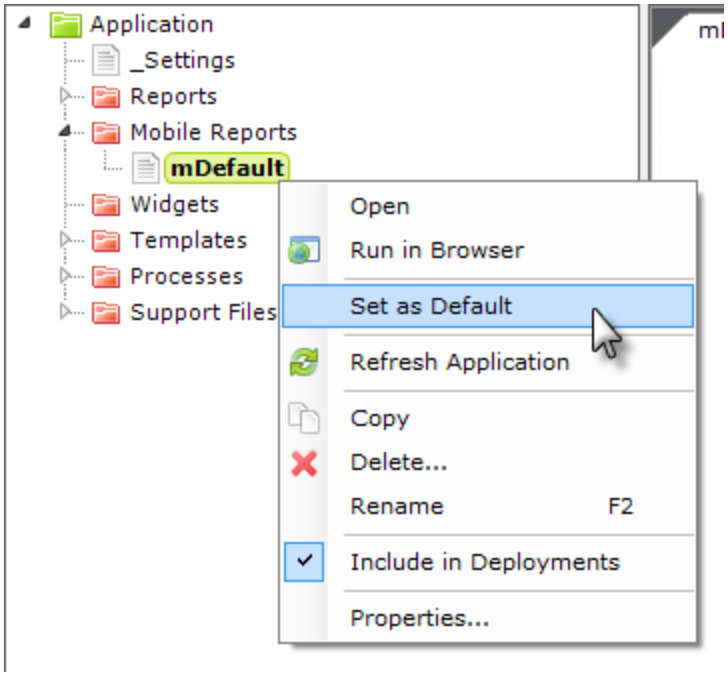



When you preview your definition, it should look something like the example shown above (although the text itself will vary, depending on what you chose to enter).

You can easily test your definition right on your own mobile device, assuming that you can connect it to your network. Many companies, for example, have Wi-Fi available internally and you may be able to connect your device to it. If so, then you should be able to browse your development web server.

The "localhost" designation used inside Studio will not work for this; you'll need to get the **IP address** of your development machine and use it instead. The URL you use in your mobile device browser will look something like: `http://192.168.0.12/yourLogiAppName`

You will need to set your Mobile Report definition to be the Default Report for your application:



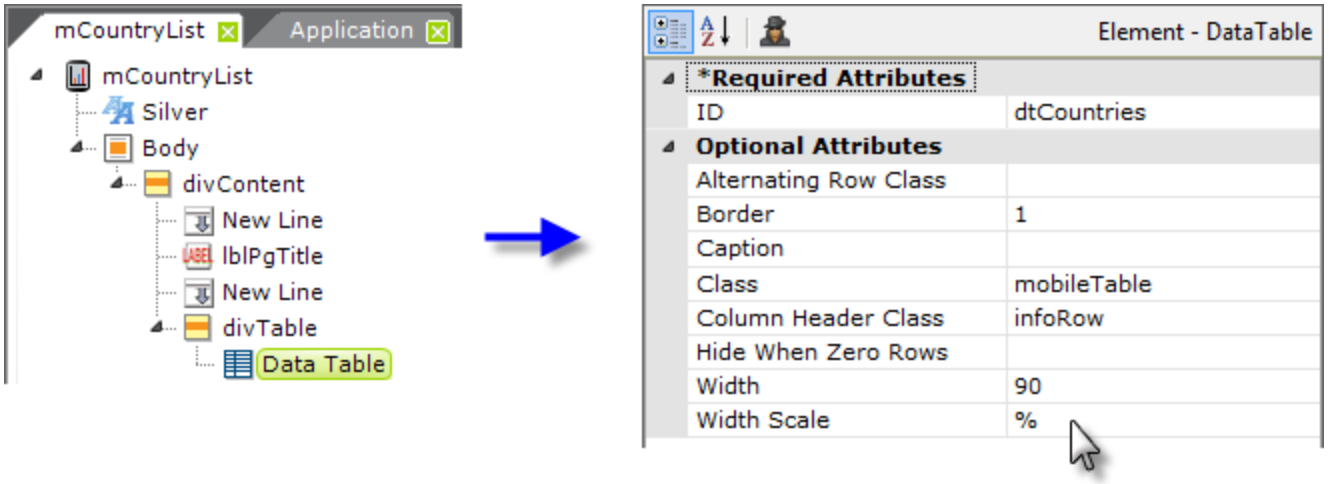

This can be done in Studio in several ways and one of the easiest is to select and right-click your definition in the Application Panel, and then select *Set As Default* in the popup menu, as shown above. This topic continues on the next page.

# Build a Data Table Definition

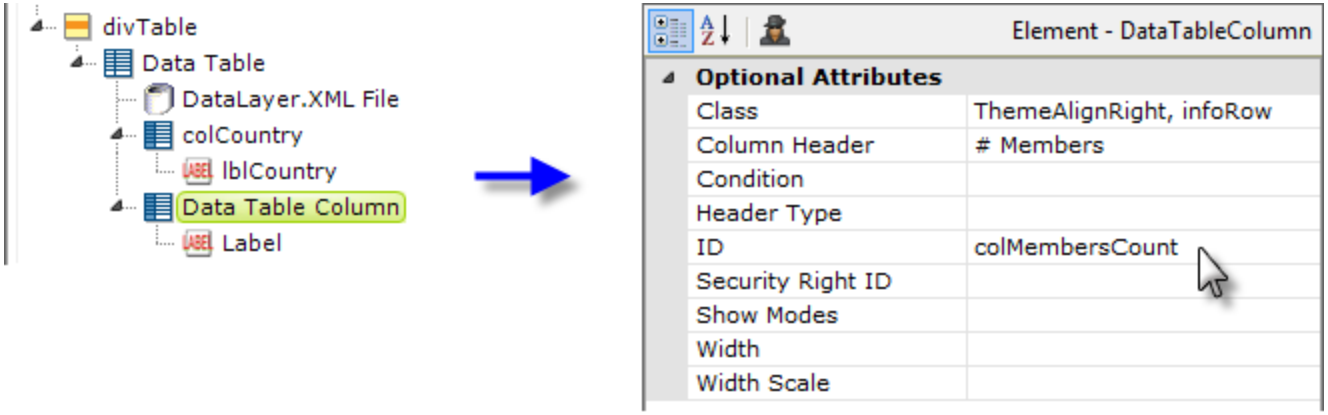
Now that we have a menu, let's build a few Mobile Report definitions for our menu to select. The first of these will display a Data Table.

1. Start by creating a new definition and duplicating the initial steps used for the mDefault definition: create a new Mobile Report definition, add a Style element, a centering Division, title text, and a second Division to left-align the table contents.

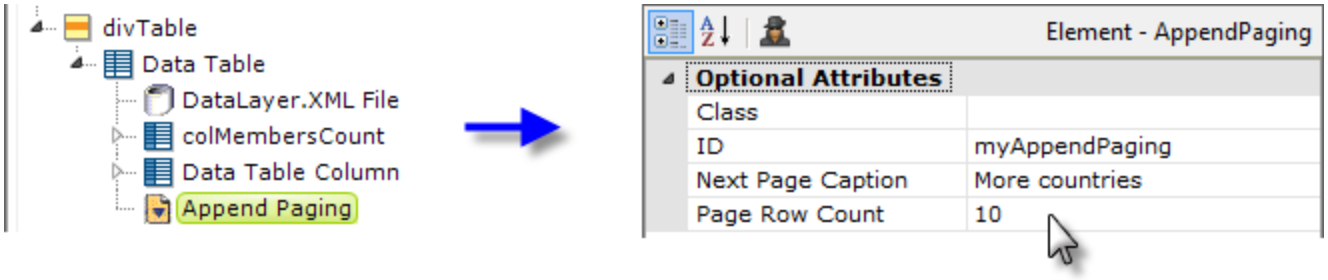
   A quick way to do this would be to right-click and *copy* mDefault in the Application panel, then select the Mobile Reports folder, right-click and *paste* the copied definition, and rename it. Then open it and delete the dtMenu element and its children.
   Make sure the name of your new definition agrees with its entry in the menu data (static or otherwise) in the mDefault definition, so tapping the right menu item will redirect to this report.



2. Your definition should look something like the example shown above. Next add a **Data Table** element beneath the second Division, and set its attributes a shown (some blank attributes have been removed to save space).
   This Data Table is only going to have two columns and will fit horizontally on the mobile device screen. However, if a table has many columns, users will be able to use flick or drag gestures to scroll the table sideways to see all the columns.




3. Add an appropriate **datalayer** element and two sets of **Data Table Column** and **Label** elements, as usual, to display the data. This Data Table *does* use Column Headers and that the columns use both a theme class *and* a supplemental stylesheet class. In the Label elements, use @Data tokens to reference the datalayer data.



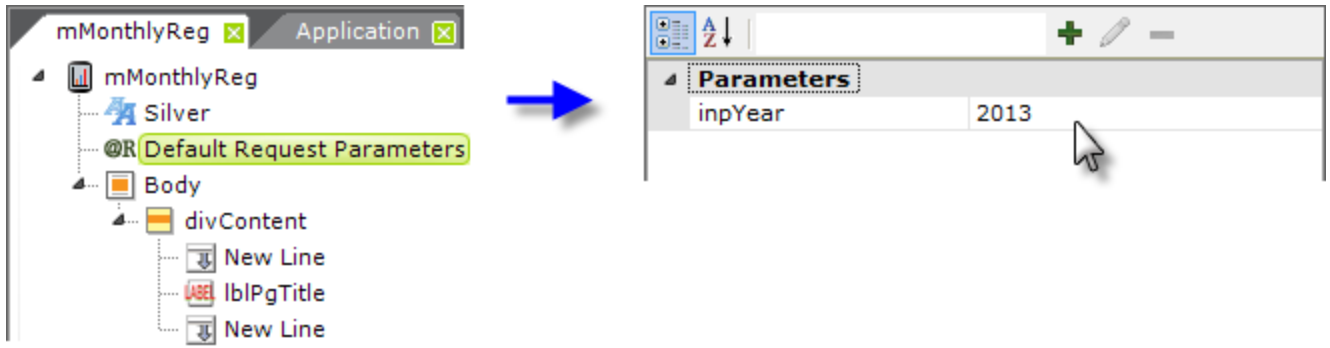
4. Next, add one of the specialized Mobile Reporting elements, **Append Paging**, beneath the Data Table and set its attributes as shown above. The table will display ten rows and a button, with the caption "More countries" just below the table. When the user taps the button, the next ten rows will be appended to the table. Each tap will append ten more rows. The supplemental stylesheet includes a class assigned by IDto the "myAppendPaging" element ID, so if you choose to give this element a different ID, make sure you change it in the stylesheet, too.
5. Finally, finish up by adding a **Label** element as a footer, if desired, as you did in the mDefault definition.


Preview your definition and it should look similar to the example above. You can clickthe "More countries" button to see how it works. Use your mobile device to browse to the mDefault definition, then tap the menu to navigate to your mobile Data Table definition.

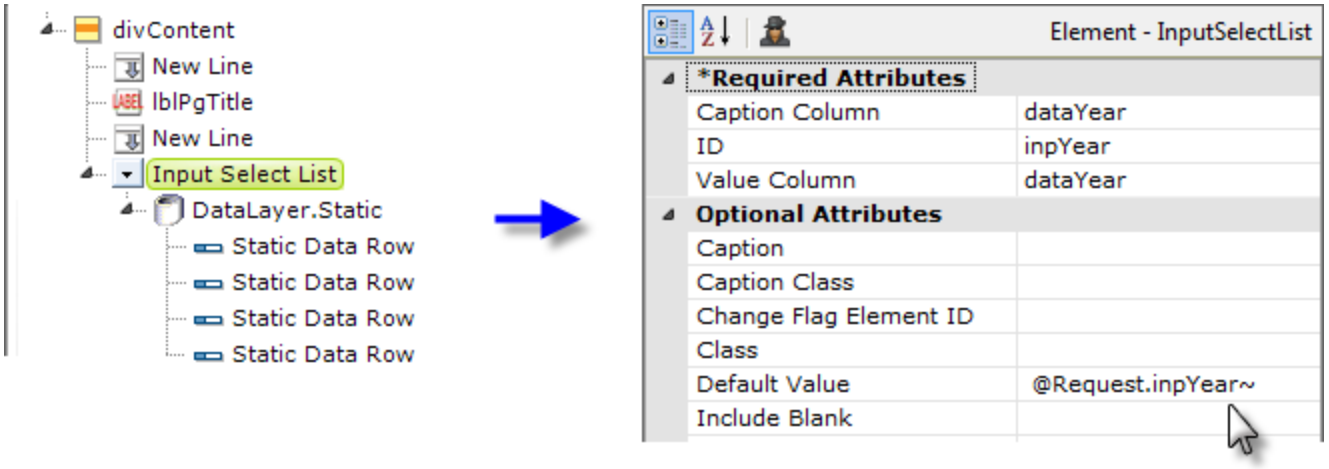
# Build a Chart Definition

And now, let's build a Mobile Report that displays a chart. In the example that follows, we'll be displaying data that can be filtered by year, and include an **Input Select List** to allow the user to select different years to display.

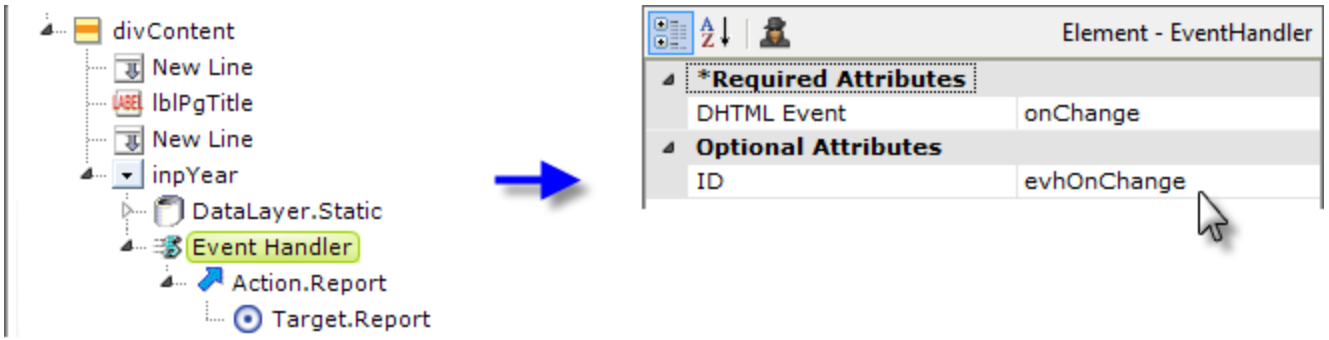
1. Once again, create a new definition duplicating the initial steps used for the mDefault definition or copy and alter the mDefault definition as described in "Build a Data Table Definition" on page 110. However, *no second Division* for left-alignment is necessary this time.
   Make sure the name of your definition agrees with its entry in the menu data (static or otherwise) in the mDefault definition, so tapping the right menu item will redirect to this report.



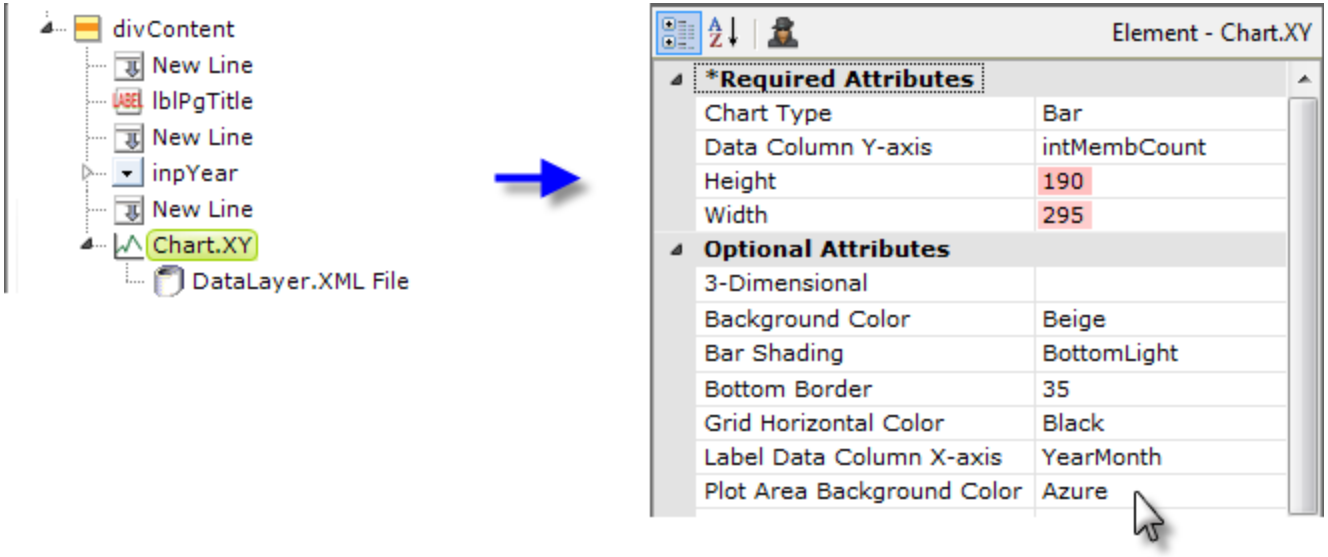

2. To ensure that our data filtering always has a default value, add a **Default Request Parameters** element to the definition, and set its attributes as shown above.



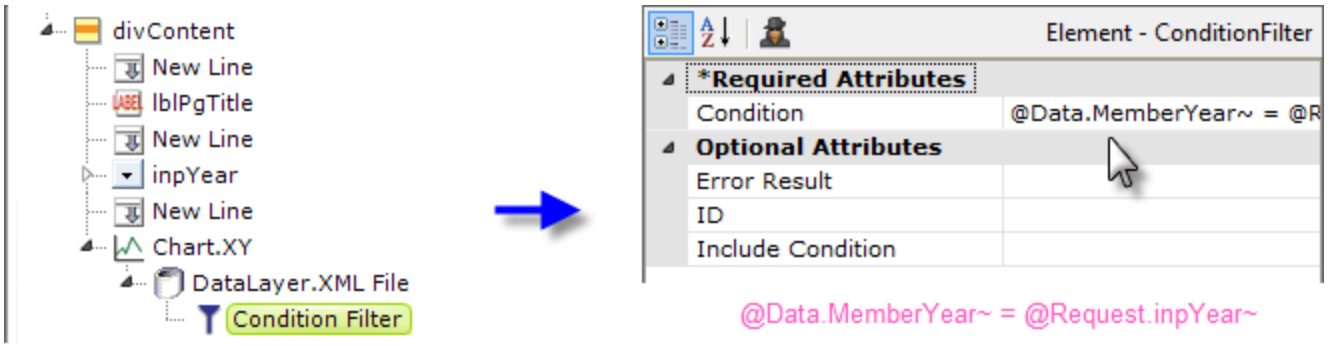

3. Next, add an **Input Select List** element, with datalayer and supporting elements, shown above, to your definition. In the example, we've used **Static Data Rows** to provide the years 2010-2013 in the select list. The list's default value will be its last selected value or, the first time the report is run, the value of the Default Request Parameter you created in the previous step.



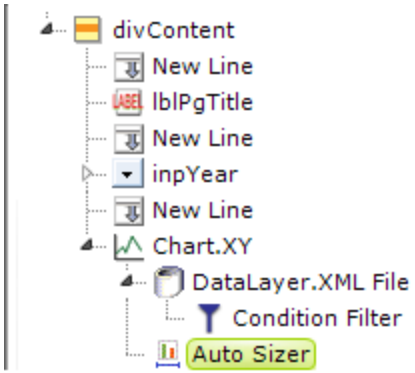
4. Next, add an **Event Handler** element beneath the Input Select List and set its attributes as shown above. Beneath it, add **Action.Report** and **Target.Report** elements. The Target.Report element's **Report Definition File** attribute should be set to the name of the current report definition, i.e. *mMonthlyReg* or whatever your definition is named.



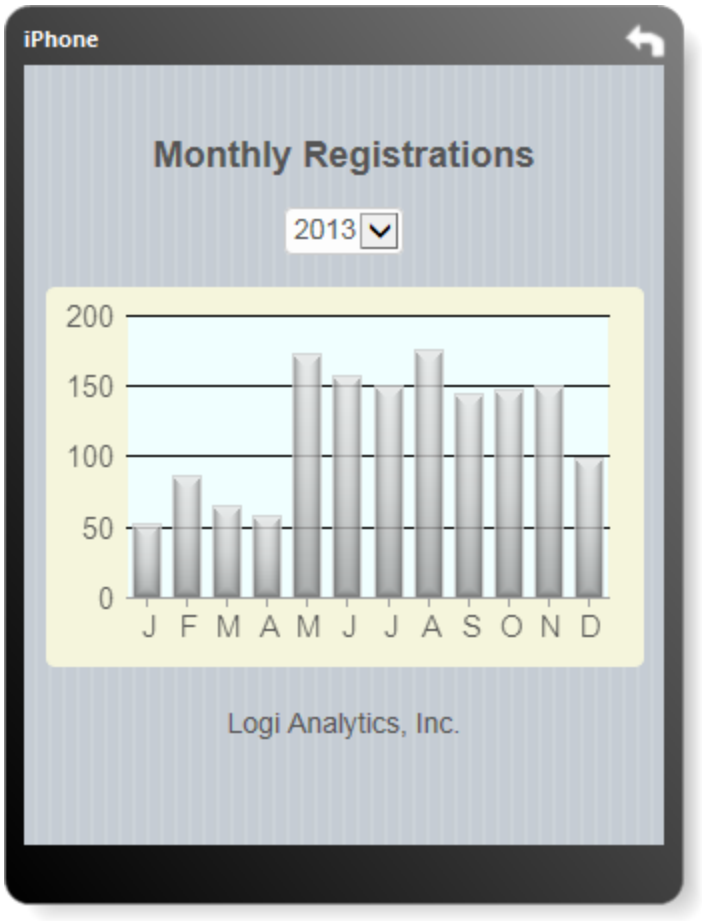

5. Add a **Chart** element, as shown above, and a datalayer to retrieve the data for it. Configure their attributes appropriately. The chart's **Height** and **Width** attributes do not have to be completely accurate, as we'll be automatically sizing it, but the *ratio* of the values you enter is relevant, as it will be preserved during the automatic sizing. You may care to experiment with different values later to get the right aspect ratio.




6. Add a **Condition Filter** element beneath your datalayer and set its attributes as shown above. This will ensure that the data displayed matches the selection the user makes in the Input Select List.




7. Now add an **Auto Sizer** element beneath the chart element, as shown above. There are no attributes to set for this element. It will automatically resize the chart to fit the available width in the mobile device browser window, and resize it if the device is rotated to landscape orientation.
8. Finally, finish up by adding a Label element as a footer, if desired, as you did in the mDefault definition.




Preview your definition and it should look similar to the example above. You can click the Input Select List and select another year to see how it works. Use your mobile device to browse to the mDefault definition, then tap the menu to navigate to your mobile chart definition.

# Logi Scheduler

Logi Info includes the Logi Scheduler, which allows Logi applications to be run by administrators on a scheduled basis. Scheduling features can also be embedded in reports, allowing runtime scheduling by end users.

The following topics discuss the Logi Scheduler:



## About The Logi Scheduler

In simple terms, the **Logi Scheduler** is a proprietary service that runs "in the background" as either a Windows Service or a Linux/UNIX daemon. It maintains its own database of scheduled events and runs them at the desired times and intervals. The Scheduler service has no user interface of its own, but is instead directed to create, maintain, and delete scheduled events by other applications through an application interface.

This service is configured to start up automatically when its host server is started and therefore runs without user intervention. From a performance perspective, it consumes very few resources on the server and spends most of its time idling.

When the Scheduler "runs" an event from its database, it does so by opening a new web server session and calling a specific task in a Logi Info **Process definition**. That process task can then run reports, send emails, etc. - any of the usual task-driven activities.

The Scheduler service release and its corresponding Logi Info release are tightly-coupled.

## Additional Resources

This topic concerns itself primarily with the preliminaries: understanding the Scheduler service and installing it. A sister topic, "Using Logi Scheduler" on page 132, is also available with more detailed information about how to configure the Scheduler and work with it from within a Logi application.

In addition, there are two sample scheduling applications available from DevNet that can be downloaded and examined for a better understanding of the elements that support the Scheduler. One of these, Scheduler Console,in the General category, is designed for administering the Scheduler on your server; this is the primary tool we provide for this purpose. The other sample application is an example of a report that allows end users to schedule reports at runtime.

# Logi Scheduler Terminology

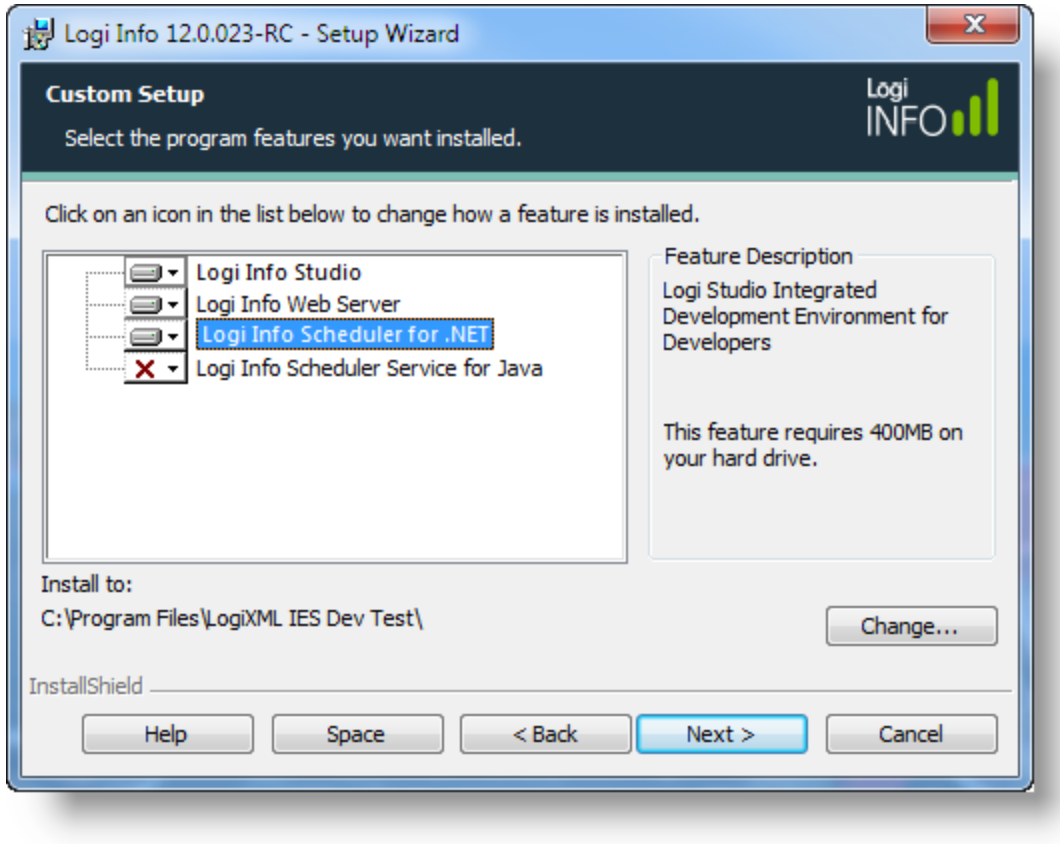
For the sake of clarity, the following terms are used in these Scheduler documents:

| Term | Description |
|---|---|
| Scheduler | The Windows service or Linux daemon that runs in the background and runs events at the times and intervals detailed in its database. |
| Scheduler Task | An event (one record) in the Scheduler's database. Contains details of time, frequency, interval, and the ID of the target process to run. |
| Process Task | A task within a Logi Info process definition in a Logi application. A Scheduler Task "runs" a Process Task at the desired time/interval. |
| Schedule XML | The XML representation of the scheduling details for a Scheduler Task. May include date, time, frequency, etc. |
| Schedule | An element used in report definitions which includes a user interface with controls, for the purpose of creating or editing a Scheduler Task. |
| Schedule Intervals | The time intervals that can be used in a Scheduler Task: *Once*, *Daily*, *Weekly*, *Monthly, Minutes,* and *Hourly.* |

The Scheduler is supported by special elements for both report and process definitions and these are introduced later on.

# Deployment for Windows

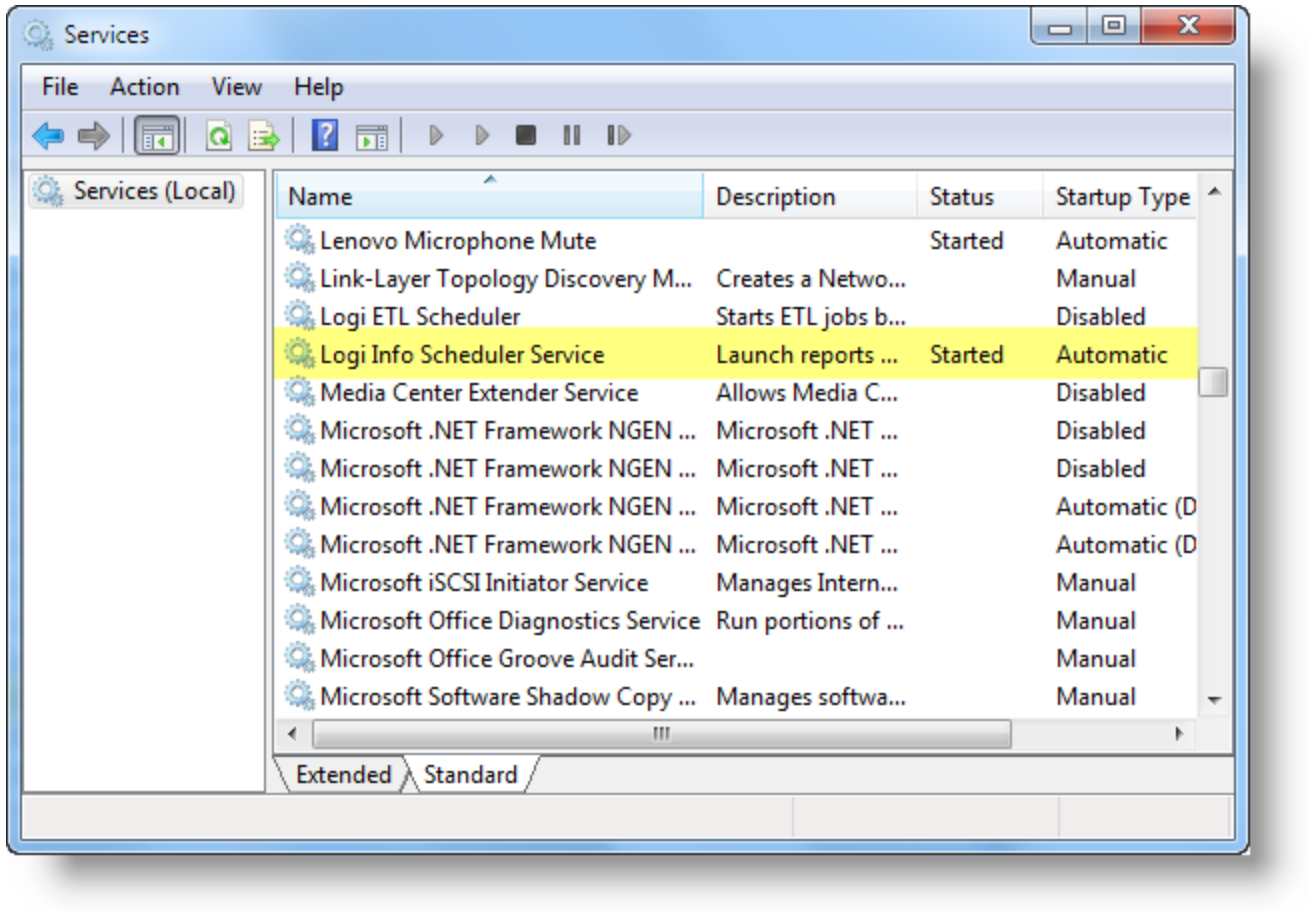
In a Windows environment, installation of the **Logi Info Scheduler Service for .NET** is accomplished as part of the Logi Info product installation. However, it's not selected for installation by default - you need to select the **Custom** installation option and indicate that it be installed. You can run the installation program again at any time and modify the installation to include the Scheduler if you need to.



As shown above, you must click the icon next to the Scheduler item and select "This feature will be installed on the local hard drive." from the drop-down list of options.

There are separate schedulers for .NET and Java applications.

By default, the .NET scheduler files will be installed to: `C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service`. In the rest of this topic, the installation folder you specified, the default or a custom location, will be abbreviated as `<installFolder>`.



After installation is complete, you can use your Windows administrative tools to examine your services. The **Logi Info Scheduler Service** should appear as shown above, its status should be "Started", and it should be configured to start automatically.

Due to the flexible nature of the Scheduler and its interactions with Logi Info applications, you can install the Scheduler on a *different server* than the application web server, separate from Logi Info, if desired.

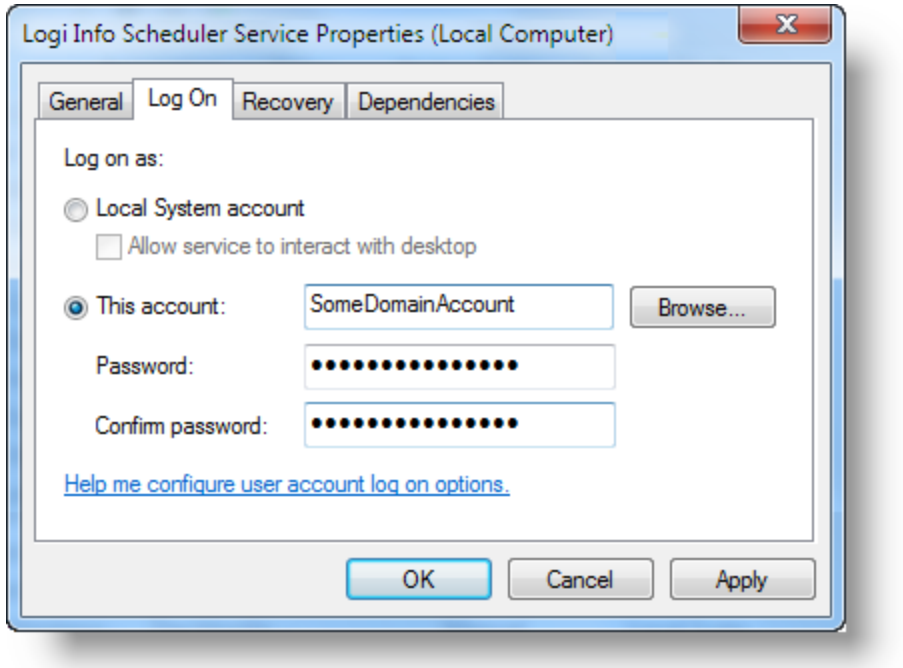
## With Logi Apps Using Logi Security and AuthNT

If you're planning to use the Scheduler to run secure Logi applications that use the "AuthNT" authentication source, then you need to configure the Scheduler service to log on using an account that can be authenticated *in the domain*. The default installation configuration uses the Local System account, which is only valid on the local machine. This configuration is independent of setting the **Run As** name for Scheduler tasks or the **Scheduler User Name** in the_Settings definition.

If the Logi application is using any Active Directory groups to filter domain users, then the domain account used by the Scheduler must also belong to those groups. The account doesn't have to be an Administrator on the server running Scheduler, but it does need these minimum permissions to accessthe Scheduler database, log errors, etc.:

| Folder or File | Permission |
|---|---|
| C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service | Read |
| C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\Schedules.vdb3 | Read, Write |
| C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\Log | Read, Write, Create |
| C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\RunNowTasks | Read, Write, Create, Delete |

To change the account being used by the Scheduler:




In the Services administrative tool, double-click the Logi Info Scheduler Service to view its properties, as shown above, then select the **Log On** tab. Check **This account:** and provide valid domain account information and click OK. Then stop and restart the Scheduler service.

# Deployment for Java

The **Logi Info Scheduler Service for Java** runs on the Linux/UNIX or Windows operating systems. It's distributed as part of the Logi Info product and, regardless of the server type that you plan to run it on in the end, in order to access its files it must first be installed on a Windows machine, using the Windows product installer, as shown in "Deployment for Windows" on page 122. Naturally, during the process you select the Scheduler Service for Java feature option, *not* the service for .NET.

The default installation location is: `C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service Java`. In the rest of this topic, the installation folder you specified, the default or a custom location, will be abbreviated as `<installFolder>`

## On a Windows Server

By default, the Scheduler Service for Java, installed on a Windows server, isn't configured to start automatically. Instead, it's started by running:

```
<installFolder>\bin\Scheduler.bat.
```

When running the Scheduler Service for Java on Windows 7 or Windows Vista, the User Account Control (UAC) system can interfere with normal operations of the Scheduler, as the `<installFolder>\Log`, `\RunNowTasks`and `\Schedules`folders require Write file access permissions. There are two options to resolve this: either disable UAC on the server, or use UAC to set the privileges of these directories to allow Write access. This MSDN article provides more information about UACshould you need it.

If you wish to start the Scheduler service automatically, and you feel comfortable directly modifying system-level configurations, use the Registry editor to add a String value in

```
HK_Local_Machine\Software\Microsoft\Windows\CurrentVersion\Run
```

Set its Value Name to `LogiSchedulerJava`and its Value Data to `<installFolder>\bin\Scheduler.bat.`

## On a Linux/UNIX Server

In order to subsequently install the Scheduler Service for Java on a Linux/UNIX server, copy *all* of the subfolders and files from the `<installFolder>`on the Windows machine to the desired installation folder on your Linux/UNIX server. That folder on your Linux/UNIX server now becomes your `<installFolder>`for the rest of this discussion.

The Scheduler is started by running `<installFolder>/bin/Scheduler.sh`. Don't use a symbolic link for this, as it may lead to unstable behavior.

To start the Java scheduler automatically, add a line to `/etc/rc.d/rc.local`. The invocation should be similar to

```
/opt/LogiSchedulerJava/Scheduler.sh &
```

 The ampersand (&) at the end *is* significant: it causes the application to load as a background process.

Due to the flexible nature of the Scheduler and its interactions with Logi Info applications, you can install the Scheduler on a *different server* than the application web server, separate from Logi Info, if desired.

# Configuring Scheduler Communications

The Logi Scheduler "listens" for requests on a specific port. For security purposes, a key value is set for each instance of the Scheduler. Logi Info applications that wish to communicate with the Scheduler have to be internally configured with the same key value in order to be identified as legitimate clients of the service. By default, the appropriate Logi Info elements and the Scheduler are installed already configured to communicate on the same port and with the same default key value. However, both of these attributes can be changed by (1) setting the appropriate attribute values in your Logi Info application's _Settings definition, and (2) editing the text file named `_Settings.lgx`in the Scheduler's `<installFolder>`. For more information about this procedure and the Scheduler in general, see " Using Logi Scheduler" on page 132.

# Schedule Task Databases

The Scheduler Service for .NET normally stores its scheduled task data using an embedded instance of **VistaDB,** a text-based database; for the Scheduler Service for Java, an embedded instance of **Apache Derby** is used.

You also have the option of storing the data instead in a networked **Microsoft SQL Server**, **Oracle**, **MySQL** or, in Logi Info v12.5+, **PostgreSQL** database. In order to use one of these SQL database servers, you must first create the Tasks table in a database of your choice, and then configure the Scheduler's `_Settings.lgx`file appropriately, as described in" Using Logi Scheduler" on page 132. Example SQL scripts for creating the Tasks table on each of the supported database servers are installed with the Scheduler in its `<installFolder>`for your use.

# Multiple Scheduler Instances

Support for storing task data in networked databases allows **multiple instances** of the Scheduler service to run at the same time, on multiple servers, creating a fault-tolerant configuration. If one of the Scheduler instances goes down, the remaining instances will continue to function. Having multiple Scheduler service instances will *not* result in a task being run more often than its designated frequency. The Scheduler service can be installed independently of the full Logi Info product.

A multi-instance configuration is *not* supported when using the standard embedded VistaDB (.NET) or Derby (Java) databases.

For the purposes of interacting with multiple Scheduler service instances from a Logi application, the application's _Settings definition must be properly configured to address all of the instances. This is discussed in " Using Logi Scheduler" on page 132.

# Quick and Easy Report Scheduling

Developers or administrators who just want to run Logi applications on a scheduled basis can do so by:

1. Installing the Logi Scheduler.
2. Downloading the Scheduler Console Sample Application.
3. Registering the sample application with your web server via Logi Studio.
4. And running it to create and manage Scheduler tasks.

# Using Logi Scheduler

The Logi Scheduler provides the ability to execute reports and other activities on a scheduled basis. Developers can build Logi applications that take advantage of Scheduler features.

The following topics provide guidance in the use of these features:



## Developer Overview

Developers who have not already done so are encouraged to read our introductory topic "Logi Scheduler" on page 119 in order to get a basic understanding of the Scheduler product and its installation before proceeding. From a development perspective, the Logi Scheduler consists of three parts:

1. An autonomous service that automatically runs scheduled reports and other Logi application activities without direct user interaction (much like the Windows Task Scheduler).
2. A database of scheduled tasks which is read and updated by the Scheduler service.

3. Special elements that let developers build definitions for creating and managing the scheduled task database. These may be used to create administrative tools for system admins, or they may be incorporated into reports so that end-users can do their own runtime scheduling. The **Schedule** element, for example, is a "super-element" with built-in user input controls for presenting scheduled task data.

# Communicating with the Scheduler

The first step in using the Scheduler with a Logi application is to create a *connection* to the Scheduler service or daemon.

The **Connection.Scheduler** element is used, in a Logi application's _Settings definition, to create this connection. The ID of this element is then used in other elements to connect them to the service.

The Scheduler has its own communication settings and Connection.Scheduler must be configured to match them in order for a connection to be made.

 For enhanced security, any Logi application that contains Process Tasks that will be called by a Logi Scheduler instance in a *different* Logi application needs to have a properly configured **Connection.Scheduler** element connecting it to the scheduler.

Default values are provided that allow communication "out of the box", but in case you wish to change them, the Connection.Scheduler attributes are:

| Attribute/Setting | Description |
|---|---|
| ID | (Required) Specifies a unique value identifying this connection; used by other elements to access the connection. |
| Pass Key | (Required) Specifies the pass key, a password-like string used to secure access to the Scheduler service. Must match the PassKey value set in the Scheduler service's `_Settings.lgx` file. Tokens are supported in this attribute.<br>*Default: myKey* |
| Port | Specifies the TCP/IP port used to gain access to the Scheduler service. Must match the Port value set in |

| Attribute/Setting | Description |
|---|---|
| | file. Tokens are supported in this attribute.<br>*Default: 56982*The Logi Scheduler supports multiple instances and a comma-delimited list of port values can be entered if multiple instances are being used. The order of the port numbers in the list must correspond with the order of names in the Server Name attribute. |
| Server Name | Specifies the name or IP address (IPv4) of the server computer where the Scheduler service is installed. Tokens are supported in this attribute.<br>*Default: localhost*The Logi Scheduler supports multiple instances and a comma-delimited list of server names or IP addresses can be entered if multiple instances are being used. The order of the names in the list must correspond with the order of port numbers in the Port attribute. |

## The Scheduler's Settings File and Logs

In most cases, communication with the Logi Scheduler will work fine without any further configuration, using default values. However, details of the settings file are presented here in case custom configurations are desired.

The Scheduler's settings are stored in its own settings file. In a Windows environment, using default installation options, this is:

```
(.NET) C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\_Settings.lgx
(Java) C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service Java\_Settings.lgx
```

In a Linux/UNIX environment, the Scheduler folders and files are installed in a folder selected by the developer, and the `_Settings.lgx` file can be found there.

```
<Setting>
<RemoteApi Port="56982" PassKey="myKey"/>
<WebRequest Timeout="20"ConcurrencyLimit="5" Stagger="1" LoggingLevel="WARNING" />
 </Setting>
```

The `_Settings.lgx` file's default contents, shown above, contain a **RemoteApi** tag with **Port** and **PassKey** values, which need to match the Connection.Scheduler element attribute values discussed earlier.

In addition, the file can contain these **WebRequest** values, which can be altered manually if necessary:

- **Timeout** - Specifies the number of minutes allowed for a task to complete before the Scheduler abandons it. Default value: *20 minutes*.
- **ConcurrencyLimit** - Specifies the maximum number of concurrently running requests/tasks allowed. When the limit is reached, additional scheduled tasks must wait for one of the other running tasks to complete before it can start. Default value: *5 tasks*.
- **Stagger** - Specifies the number of seconds to wait before starting the next request/task. Default value: *1 second*.
- v23.1 **LogsInDatabase** - Controls the ability to store log files in a database. This attribute must be manually configured and set to "True" to enable database storage. See section below for more information.
- **LoggingLevel** - Specifies the logging level for events, using these values: *ERROR*, *WARNING*, *INFO*, *DEBUG,* or *NONE*.

  The Scheduler Service for .NET logs operational events to files in (if default installation location was used):

  `C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\Log`

  The Scheduler Service for Java logs events to files in `<schedulerInstallFolder>/Log/LogiScheduler.log`

  The *INFO* logging level value provides a moderate amount of detail about Scheduler events in the logs, while the *DEBUG*

value records every web request made, including creating a task, querying the task list, getting a task, updating a task, deleting a task, and many details about running tasks. This value should *not* be used casually, as it will generate many, many log entries. Default value: *WARNING*.

- **PollingFrequency** - (not shown) Specifies the number of seconds that elapse before the database is polled for a task to run. This should normally not be modified. Default value: *60 seconds*.
- **MaxFieldLength** - (not shown) Specifies the maximum length of the ProcessXML value, which is 4,000 characters by default. You may need to increase this limit if you're scheduling tasks with a very large number of, or very long, request parameters. *Note - If you provide a longer value here, you must also manually alter the associated column in your Scheduler database table to match*.


You may need to stop and restart the Logi Info Scheduler service or daemon after saving changes to this settings file.

The settings file also includes an explanatory comments section. This section includes example tags for configuring connections to database servers. These are used if you want to store your schedule task data in a networked SQL database rather than in the standard, file-based database. Information about configuring these connection can be found in "Scheduled Task Data Storage Options" on page 163.

Once you've added and configured a Connection.Scheduler element to your Logi application, you're ready to begin working with the other Scheduler elements.

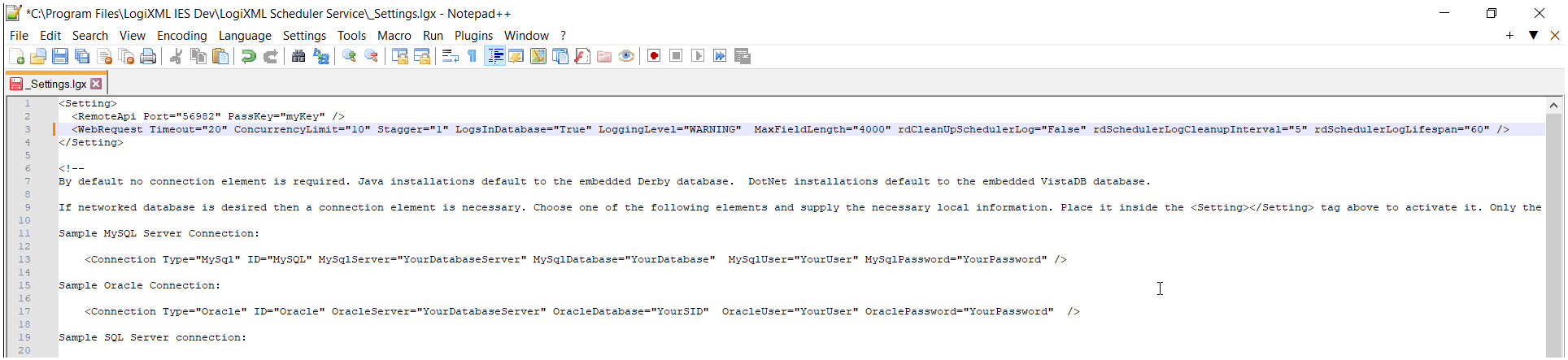
### v23.1 Storing Log Files in Database

You can now store log information in a database, enabling you to send a database query to observe specific log information and run statistics on the events or log behaviors. The attribute "LogsInDatabase" controls whether the log is written to the database

or file system. When set to *True* (case sensitive), the log records to the task database; otherwise, it will be written to the file system (current scheduler behavior).

To enable this capability, access the LogiXML Scheduler Service folder on your machine and open the **_settings.lgx** file.

In _settings.gx, change the existing attributes in WebRequest to LogsinDatabase="True", as shown below:




''

Upon adding this attribute, a table will be created in the existing available database. The default table name of the task log is “TaskHistory”. You are unable to change this name when using default databases Derby and VistaDB. For other databases, you can customize the table name using the attribute “Setting/Connection/@xxxHistory“, similar to setting the task table name.

To change the table name, go to the Connection element (shown in screenshot above) in the _settings log and add your own history table name. For example, "MySQLHistory" or "OracleHistory". Then, restart the scheduler.

Each database has a minimum number of columns; however, you can expand and add more tables/columns. If you already created one table (like "MySQLHistory"), the system will use that. If there is no table in the DB, the task log table generates

automatically when the scheduler service starts. For additional information about custom tables in a database, see "Scheduled Task Data Storage Options" on page 163.

# Getting Data with DataLayer.Scheduler

A special datalayer element, **DataLayer.Scheduler**, is used to retrieve data from the Scheduler database. The query it uses can be restricted by providing default values for some of the element's attributes. When examining or editing a Scheduler task with User Input elements, this datalayer is typically used with **Local Data** so that the data retrieved is widely available to other elements in the definition, using @Local tokens. It can also be used beneath a **Data Table** to display all Scheduler tasks.

DataLayer.Scheduler has the following attributes:

| Attribute | Description |
|---|---|
| Connection ID | (Required) Specifies the element ID of a Connection.Scheduler element in the _Settings definition. Connects the datalayer to the Scheduler database. |
| ID | Specifies a unique element ID. |
| Include Task Result Error Message | Specifies whether task error messages will be included in the data. If set to *True*, the datalayer will include an ErrorMsg column containing error message text from any failed Scheduler tasks. The default value is *False*. |
| Scheduler Application ID | Specifies the first 20 characters of the name of a Logi application or any other categorizing text developers want to use. If not blank, acts as a Condition Filter and limits the datalayer query results to Scheduler tasks with matching values for this column. Max: 20 characters. |
| Scheduler Custom Column 1 | Specifies a value of any kind with a maximum length of 1,000 characters. If not blank, used to limit the datalayer query results to Scheduler tasks that match this column's value. |

| Attribute | Description |
|---|---|
| Scheduler Custom Column 2 | |
| Scheduler RunAs | Specifies a User Name value to be used when a Scheduler task is executed and Logi Security is enabled. The name is provided to Logi Security in order to ensure that reports are run with appropriate security control. Leave blank if Logi Security is not enabled. Max: 50 characters. |
| Scheduler Task ID | Specifies the unique ID of a Scheduler task (these IDs are created and returned when new Scheduler tasks are created). If not blank, used to limit the datalayer query results to Scheduler tasks for the specified task ID. |
| Scheduler Task Name | Specifies a unique name given to a Scheduler task, generally for easier task identification. If not blank, used to limit the datalayer query results to Scheduler tasks for the specified task name. Max: 50 characters. |

When DataLayer.Scheduler runs, it retrieves the following data columns for each stored Scheduler task:

| Column | Description |
|---|---|
| TaskID | Unique Scheduler task ID. |
| ApplicationID | Logi application name. |
| TaskName | Descriptive name for Scheduler task. |

| Column | Description |
|---|---|
| CustomColumn1 | Value selected by developer. |
| CustomColumn2 | Value selected by developer. |
| IsDisabled | "True" or "False" - Indicates state of this Scheduler task; disabled tasks are not executed. |
| ScheduleXML | XML data containing date-time-interval specific information. Example:<br>`<Schedule Type="Once" StartDate="2013-02-05" EndDate="2013-02-27" FirstRunTime="13:48" />` |
| ProcessXML | XML data containing Process-specific information. Example:<br>`<ProcessTask ProcessFile="myProcess.Run" TaskID="RunSalesReport" URL-L="http://localhost/myApplication" PhysicalPath="C:\Projects\myApplication"><LinkParams inpCategory="3" /></ProcessTask>` |
| TimeCreated | ISO format timestamp for creation of this Scheduler task.<br>Example: `2013-02-03T16:37:18-05:00` |
| TimeModified | ISO format timestamp for modification of this Scheduler task.<br>Example: `2013-02-03T16:37:18-05:00` |
| TimeLastRun | ISO format timestamp for this Scheduler task's last run.<br>Example: `2013-02-03T16:37:18-05:00` |
| TimeNextRun | ISO format timestamp for this Scheduler task's next scheduled run.<br>Example: `2013-02-03T16:37:18-05:00` |

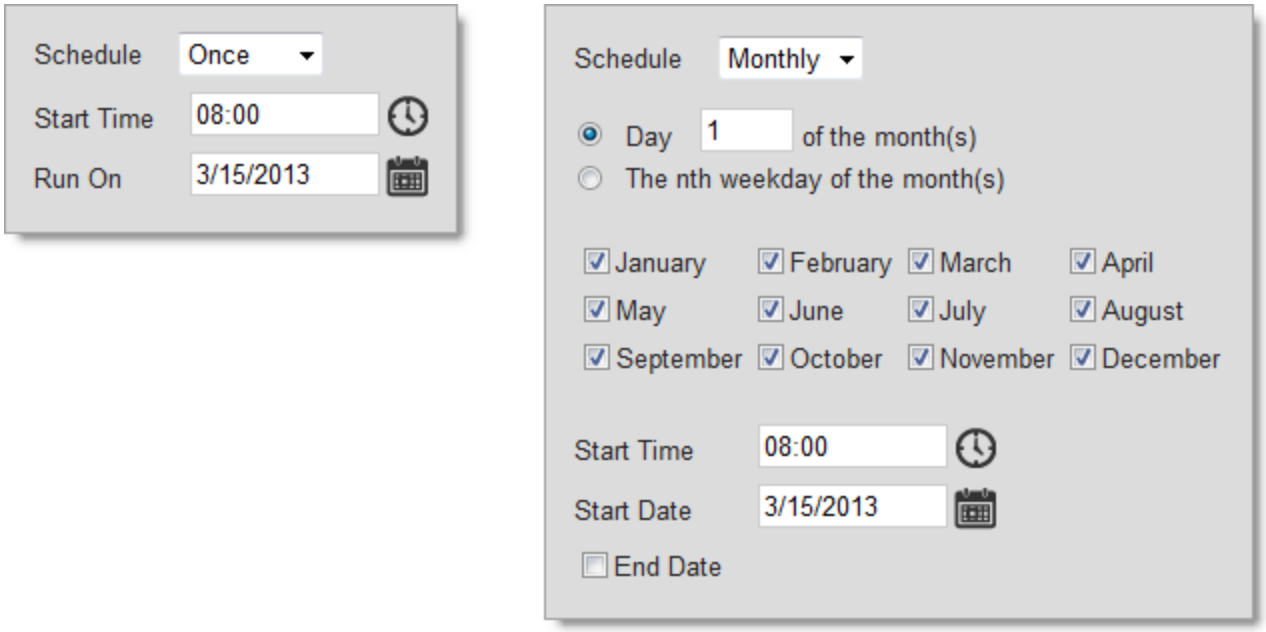
| Column | Description |
|---|---|
| TimezoneLastRun | Last runtime of the specified time zone. Data type and format is the same as the column "TimeLastRun". If there is no time zone, this column defaults to TimeLastRun. |
| TimezoneNextRun | Next runtime of the specified time zone. Data type and format is the same as the column "TimeNextRun". If there is no time zone, this column defaults to TimeNextRun. |
| TimezoneName | Name of the time zone. Data type is string(80). |
| RunAs | The User Name to pass to Logi Security when the Scheduler task is executed. Ignored when the Schedule task runs if Logi Security is not enabled. For more information, see "Running and Deleting Scheduler Tasks" on page 154. |
| WasSuccessfulLastRun | "True" or "False". Indicates if last run of this Scheduler task was successful (occurred without errors). |
| TaskResults | Fully-qualified path and name of task results file on the server. Will not exist if tasks have never been run. Examples:<br>`(.NET) C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\Log\ rdSchedulerTask-8-d55cadb8-e243-437d-8587-d190417bdcb2.xml`<br>`(Java) C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service Java\Log\ rdSchedulerTask-8-d55cadb8-e243-437d-8587-d190417bdcb2.xml` |
| IsRunning | "True" or "False" -Indicates if this Scheduler task is currently running. |
| ErrorMsg | If the datalayer's **Include Task Result Error Message** attribute is set to *True*, this column will |

| Column | Description |
|---|---|
| | include the error message text for any failed tasks. |
| ScheduleDescription | Plain language description of date-time-interval information for this task:<br>Example: `At 1:48 PM on 02/05/2013` |
| ProcessUrl | URL for the web application in which the Process definition file named in the next field resides.<br>Example: `http://localhost/myWebApp` |
| ProcessFile | Name of the Process definition file, without a file extension, that contains process task to be executed when the Scheduler task runs. *Do not include the .lgx file extension*. |
| ProcessTaskID | Name of the process task, within the process definition named in the previous attribute, that will be executed when the Scheduler task runs. |
| ProcessParams | List of parameters to be passed to the process task that will be executed when the Scheduler task runs. Example: `<LinkParams CategoryID="4" />` |

In order to edit an existing Scheduler task, it's the developer's responsibility for retrieve the data for the task and then provide it to the Schedule element, described in "The Schedule Element" on the next page.

# The Schedule Element

The **Schedule** element makes it easy to create and edit Scheduler Tasks. Because it's a super-element, adding it to a report definition automatically creates a user interface for a Scheduler task.




As shown above, the user interface presented by the Schedule element allows the user to select criteria for the Scheduler task. It's dynamic and will present more or fewer controls as required. For example, in the image above right, when a Monthly schedule is selected, the element expands vertically to display the months (a horizontal expansion, to the right, instead is also an option).

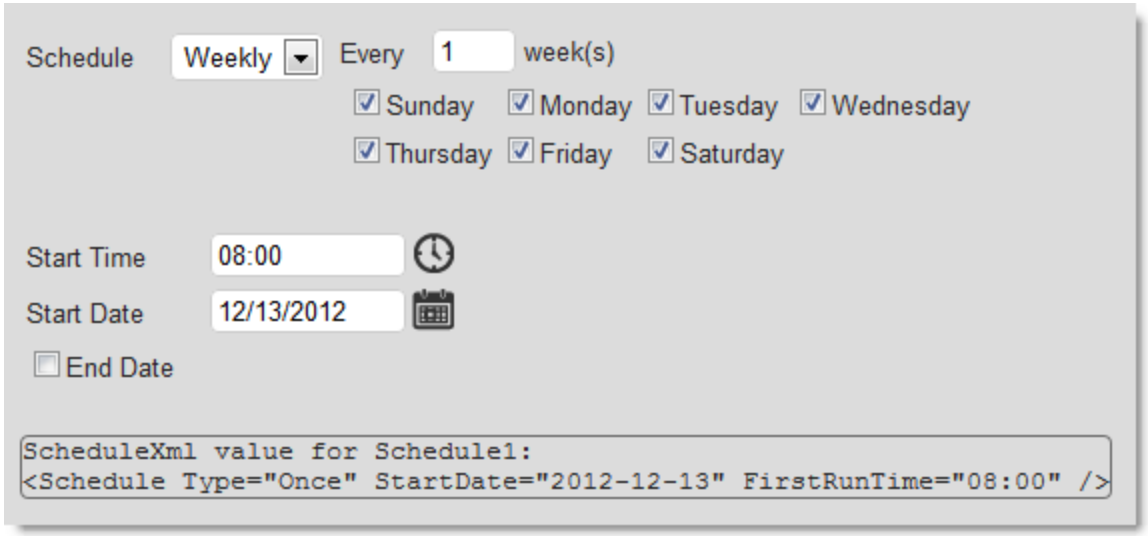
The Schedule element is built as an HTML table and has typical table attributes, such as Layout, Width, and Width Scale, used to control its size. The Schedule element also uses the following other attributes:

| Attribute | Description |
|---|---|
| Format | Specifies the format of the Start and End dates in the UI. Other format choices are shown but have no effect. Default: *System Short Date format* |
| Schedule Intervals | A comma-separated list of the interval modes that will be displayed in the UI. This can be used to limit the type of Schedules tasks that can be created and saved. Valid choices are *Once*, *Daily*, *Weekly*, *Monthly, Minutes,* and *Hourly*. Default: *All modes* |
| Schedule Orientation | Specifies how the UI expands when certain interval modes are selected. *Horizontal* places the configuration options for the selected interval mode to the right of the interval, start date, end date and start time inputs. *Vertical* (the default) places the configuration options below them. |
| ScheduleXML | A string of XML data that represents the date-time-interval data for a Scheduler task. This can be typed into this attribute's value as a "hard-coded" string or placed there using tokens such as @Local. This provides the mechanism for reading a schedule task, using Local Data and DataLayer.Scheduler, from the Scheduler database and placing the retrieved data into the Schedule element for manipulation by the user. |
| Show Days Of The Week | Specifies that a checklist of days-of-the-week will be displayed when the *Weekly* interval mode is selected. Default: *True* |
| Show Months | Specifies that a checklist of months will be displayed when the *Monthly* interval mode is selected. Default: *True*. |
| Show Process Parameters | Specifies that additional controls, used to add any number of name-value pairs of parameters used to run a Scheduler task, will be displayed. This is primarily used for administrative applications. Default: *False*. |

| Attribute | Description |
|---|---|
| Show Schedule XML | Specifies that, as options are selected in the UI and the XML data is generated, it will be displayed to the developer below the element. This is intended to be used for debug purposes only. Default: *False* |
| Template Modifier File | Specifies the name of a template modifier file that can be used to alter the user interface. Can be used, for example, to change language- and culture-specific control Captions. You can also use this attribute to set a default Time Zone for scheduled reports. |

The element's UI *does not* display the ProcessXML value, a string of XML data that includes the task name, application URL, process definition file name, and process task ID, nor does it display the Task Name and Run As values. These values are available, however, from the Scheduler database as columns for retrieval using DataLayer.Scheduler and, if desired, the developer can create a UI to display and update them outside of the Schedule element interface.

As mentioned above, the **Schedule** element has a **Show Schedule XML** attribute which can be turned on during development to assist in understanding scheduling intervals:




The image above shows what the display looks like when the Show Schedule XML attribute is set to *True*. If you change the interval selections, the ScheduleXML display will be immediately updated with new values.

Parameters that are to be passed to the process task at runtime can also be stored in the Scheduler task. For more information, see "Working with Process Parameters" on page 152.

## Using CSS with the Schedule Element

Many presentation aspects of the Schedule element can be customized through the use of CSS classes. The default style sheet used by the element is `<application folder>\rdTemplate\rdSchedule\rdScheduleStyle.css`

Classes in this default style sheet can be overridden by copying them into, and altering them in, your own application style sheet. *Do not modify the default style sheet.*

For example, this class adds additional **space** between the UI's labels and input controls:

```
.rdScheduleColumnGap {
width: 15px; /* add more space between labels and controls */
 }
```

and this one adds a **background color** to the element:

```
.rdSchedule TABLE {
border-collapse: collapse;
border-width: 0px;
background-color: AliceBlue; /* add BG color so element is obvious */
 }
```

 HTML tables are used within the Scheduler super-element, and so you can count on working with table-related HMTL tags to make some CSS changes.

# Creating and Editing Scheduler Tasks

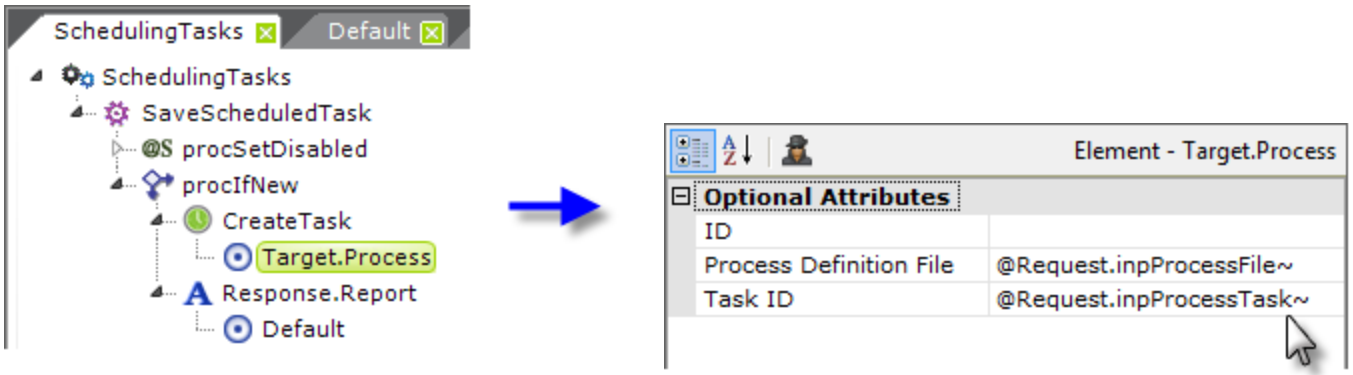
Developers need to create Process definition tasks to create and edit the Scheduler task data that, typically, has been collected using the Schedule element.

The **Procedure.Scheduler Create Task** and **Procedure.Scheduler Update Task** elements are used for this purpose and have identical attributes, with the exception that the Scheduler Update Task element also includes a Scheduler Task ID attribute. They have the same attributes as the Schedule element, which was discussed on the previous page.

These two elements have an additional attribute: Scheduler Session Variables, which let your enumerate, in a comma-separated list, the Session variables that should be passed to the scheduled task when it's executed.

Both of these elements are used in Process tasks that might, for example, be called from a report definition using a "Create a New Scheduled Task" or an "Edit a Scheduled Task" link on a report page.

 They require **Target.Process** child elements and use them in an unusual fashion. Instead of identifying a **destination** for program flow to be redirected to, as most Target-type elements do, these child elements' attributes are used instead to supply the Scheduler database with the Process definition **file name** and Process **task name** for the Scheduler task being created or updated. Typically these are in the form of Request tokens passed from the calling report definition.

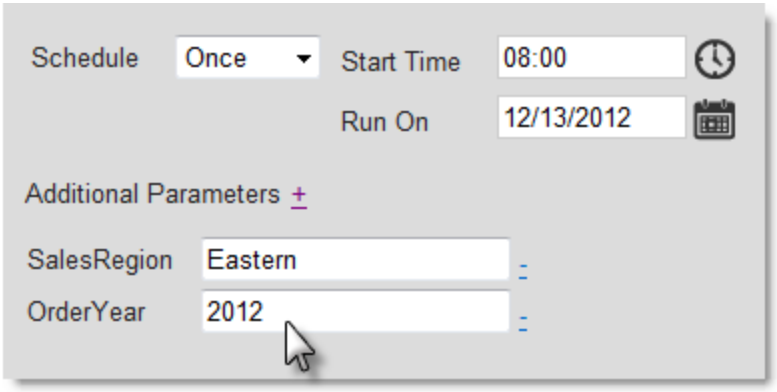



As shown in the example Process definition above, a Procedure.Create Scheduler Task element ("CreateTask") has been added to a Process task. Beneath it, a **Target.Process** element is used to identify the Process definition file and Process task that will be "called" by the Scheduler. A definition using Procedure.Update Scheduler Task looks very similar.

When a new Scheduler task is created, its **Task ID** is returned and can be accessed using a procedure token. In the example shown above, it would be available as: `@Procedure.CreateTask.TaskID~` (remember that tokens are case-sensitive).

# Working with Process Parameters

Parameters, in the format of name-value pairs, can be stored with a Scheduler task and passed to the target Process task when the Scheduler task runs. The **Schedule** element includes an user input control set that it displays for entering and editing these parameters. Its appearance in the UI is controlled by the element's **Show Process Parameters** attribute.



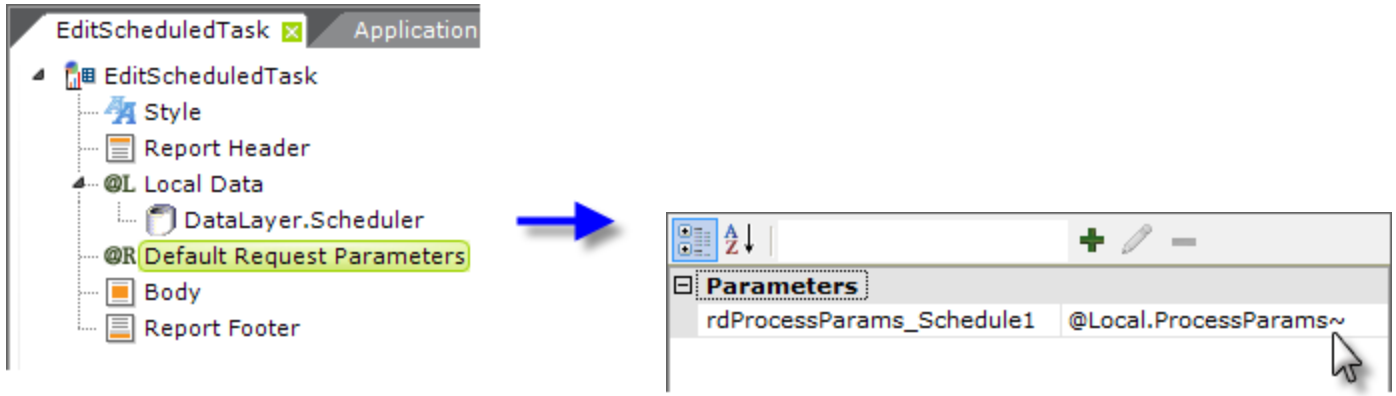

As shown above, the Schedule element's interface can be used to add a number of parameters at runtime. As the parameters are entered, they're assembled into an XML string and placed into a hidden input control with the ID *rdProcessParams_<Schedule Element ID>*.

## Saving the Parameters

When a Process task is called to save the task data, the XML string of parameters is passed to it as a request variable. The elements used to store Scheduler data, Procedure.Scheduler Create Task and Procedure.Scheduler Update Task, will automatically look for and use this request variable and store the parameters with the Scheduler task.

## Retrieving the Parameters

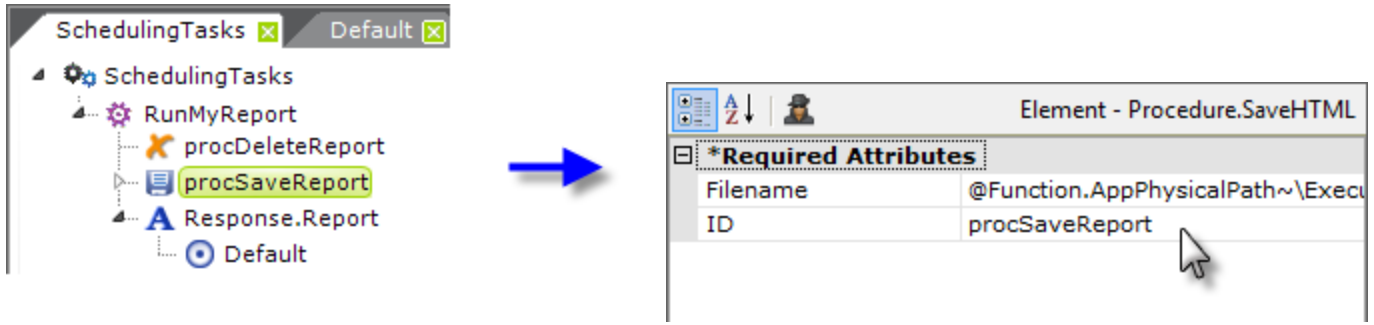
We need to reverse the previous process to retrieve the stored parameters and load them into the Schedule element:




The example shown above is of a definition that will allow users to edit stored Scheduler tasks. A DataLayer.Scheduler, a child of Local Data, is used to retrieve data for the desired Scheduler task. Then a **Default Request Parameters** element is used with a Local token to create a request variable named *rdProcessParams_<Schedule Element ID>* with the value of the ProcessParams column in the datalayer. This will automatically load any stored parameters into the Schedule element interface, so that they can be edited.

# Running and Deleting Scheduler Tasks

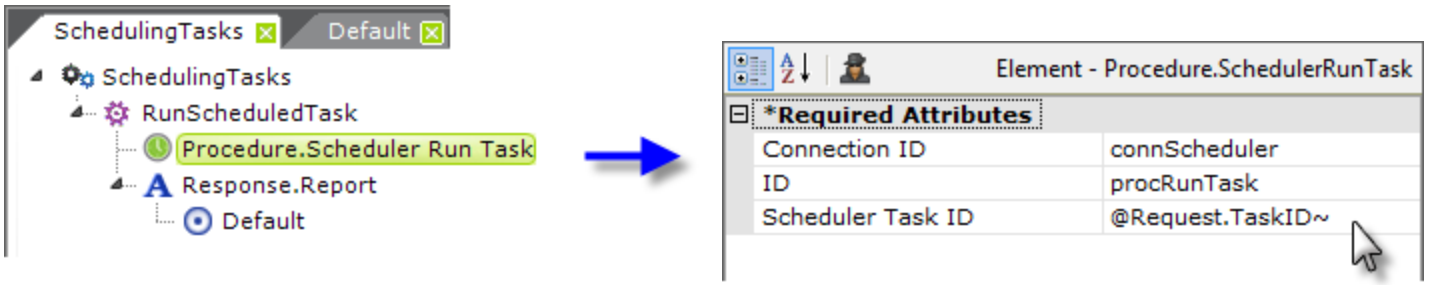
When the Scheduler runs one of its scheduled tasks, it calls a task in a Process definition:




The example shown above is of a task that runs a report and saves it as an HTML file. First, any older version of the file needs to be deleted, then a Procedure.Save HTML element is used to save the desired report as a file on the web server. After that users could presumably browse it or further procedure tasks could distribute it as email, etc. Finally, the task needs to end with some kind of Response element (although its target report will never be visible to anyone because the Scheduler runs tasks in their own session). So, at the correct time, the Scheduler would simply call the RunMyReport task to get the job done.

Two other elements, **Procedure.Scheduler Run Task** and **Procedure.Scheduler Delete Task**, can be used to immediately run Scheduler tasks, and to delete them. These elements only require the connection ID of the Connection.Scheduler element and a TaskID.

For instance, you may wish to give your users the option to run a report immediately, instead of waiting for the scheduled time. This is done by creating a Process task that tells the Scheduler to run the desired Scheduler task right now.




As shown above, this is done in a Process task using the Procedure.Scheduler Run Task element. Its attributes identify an existing Scheduler task by ID.

## Running Scheduler Tasks with Logi Security

If the Scheduler runs Process tasks within an application that implements Logi Security, there will be no physical user at the time (and no Login page), so what security credentials and rights will the Scheduler use?

The "user" in this case will be the one named in the value of the **RunAs** column in the Scheduler Task record in the Scheduler's database. Enter a valid user name in this column. If you want dynamic values and, for example, you're using Procedure.Create Scheduler Task to create the task, you can enter the @Function.UserName~ token in the RunAs field. This will automatically set the value to the user name of the currently logged-in user. In that way, the Scheduler will run the Process task using the name of the user who created the scheduled task.

In addition, if you're using a query of some kind to get Security Rights in the _Settings definition, be sure to use that same token, @Function.UserName~, in the query *insteadof* @Request.rdUserName~. That will ensure that the query will return the correct rights when the Scheduler executes the application.

## Deleting a Task

The coding for a Process task for deleting a Scheduler task is identical to that shown above for creating a task, except that it uses a Procedure.Delete Task element.

# Scheduler Results and Logging

The Scheduler service can write out two types of information about its attempts to run tasks: **results files** and an **operational log**.

The results file contains details about the parameters passed to the targeted Process task, time stamps, a final status, and any error message passed through the Logi Server Engine.

Operational logs contain lower-level details about the internal operations of the web service itself and any errors it encounters. Whether there will be *any* logging and the amount of detail in the log depends on the "logging level" specified in the Scheduler's `_Settings.lgx` file.

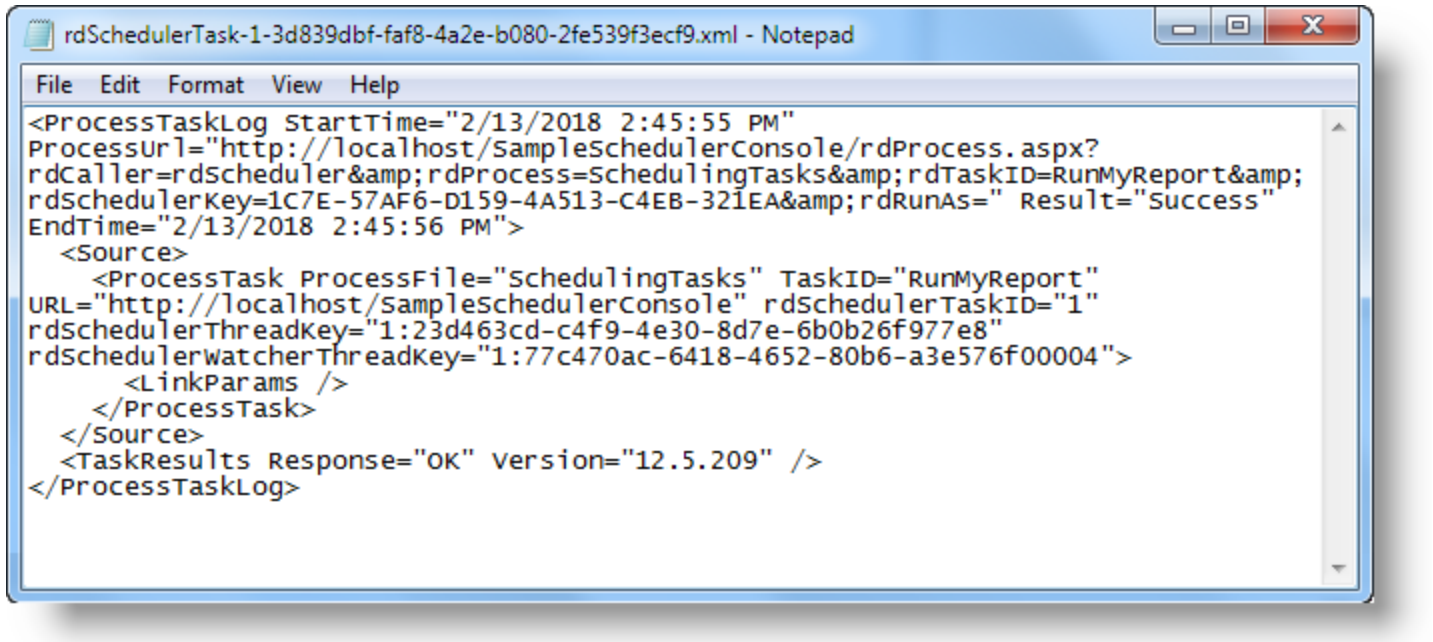
In addition, the Scheduler updates a scheduled task's database record after each run attempt. For example, the success or failure of each Scheduler task's last attempted run is written in its database record in the "WasSuccessfulLastRun" column, and the fully-qualified path and filename of the results file generated for each run attempt is written to the "TaskResults" column. This allows easy programmatic access to run attempt results from within reports, using DataLayer.Scheduler, and can also be very helpful in diagnosing problems.

## Results Files

As mentioned above, a results file will be created for each run attempt. These are .xml text files and are stored in:

```
(Windows .NET) C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\Log
(Windows Java)C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service Java\Log
(Linux/UNIX) <installFolder>/Log
```

A typical results file name is: `rdSchedulerTask-8-d55cadb8-e243-437d-8587-d190417bdcb2.xml`




The example above shows the XML data in a results file. The contents will vary and may include an error message. These files can be read using a Logi report definition for the purpose of displaying task run status information (this is how the Scheduler Console sample app gets its task status).

Results files are small, typically about 800 bytes, but they will accumulate over time and may consume excessive server storage if ignored. It's the developer's or server administrator's responsibility to manage these files.

## Error Notifications

You now have the option to receive automated email notifications when a Scheduler report errors. Developers have the flexibility to define their own custom logic for error handling. If the Scheduler service has access to @Session error messages, the developer can use that to write a custom logic accordingly.

This feature can be enabled by adding the constant "SchedulerErrorEmail" in application _Settings. The value of this constant is the admin's email address; however, multiple addresses are supported, separated by ";". If there is no constant or the value is empty, this function is disabled.

 If the receiving email address is invalid, the error will be also logged in Scheduler task log file and the error message email will not be sent.

The first smtp connection in application _Settings is used to send an error message email. If there is no smtp connection, an error will be logged in the Scheduler task log file.

You can customize error message email content (send and receive email address, subject, body) by creating `DMF rdWeb1/_SupportFiles/rdSchedulerEmail_DMF.xml` Initial content can be customized like this:

```
<ScheduleMods>

<SetAttribute XPath="." AttributeName="EmailSubject" Value="Failed to schedule task" />

<SetAttribute XPath="." AttributeName="EmailBody" Value="Error message: " />

<SetAttribute XPath="." AttributeName="FromEmailAddress" Value="" />

<SetAttribute XPath="." AttributeName="ToEmailAddress" Value="" />

</ScheduleMods>
```

The following template points to the above-mentioned DMF file: `rdTemplate/rdEmail/rdSchedulerErrorEmailTemplate.lgx`

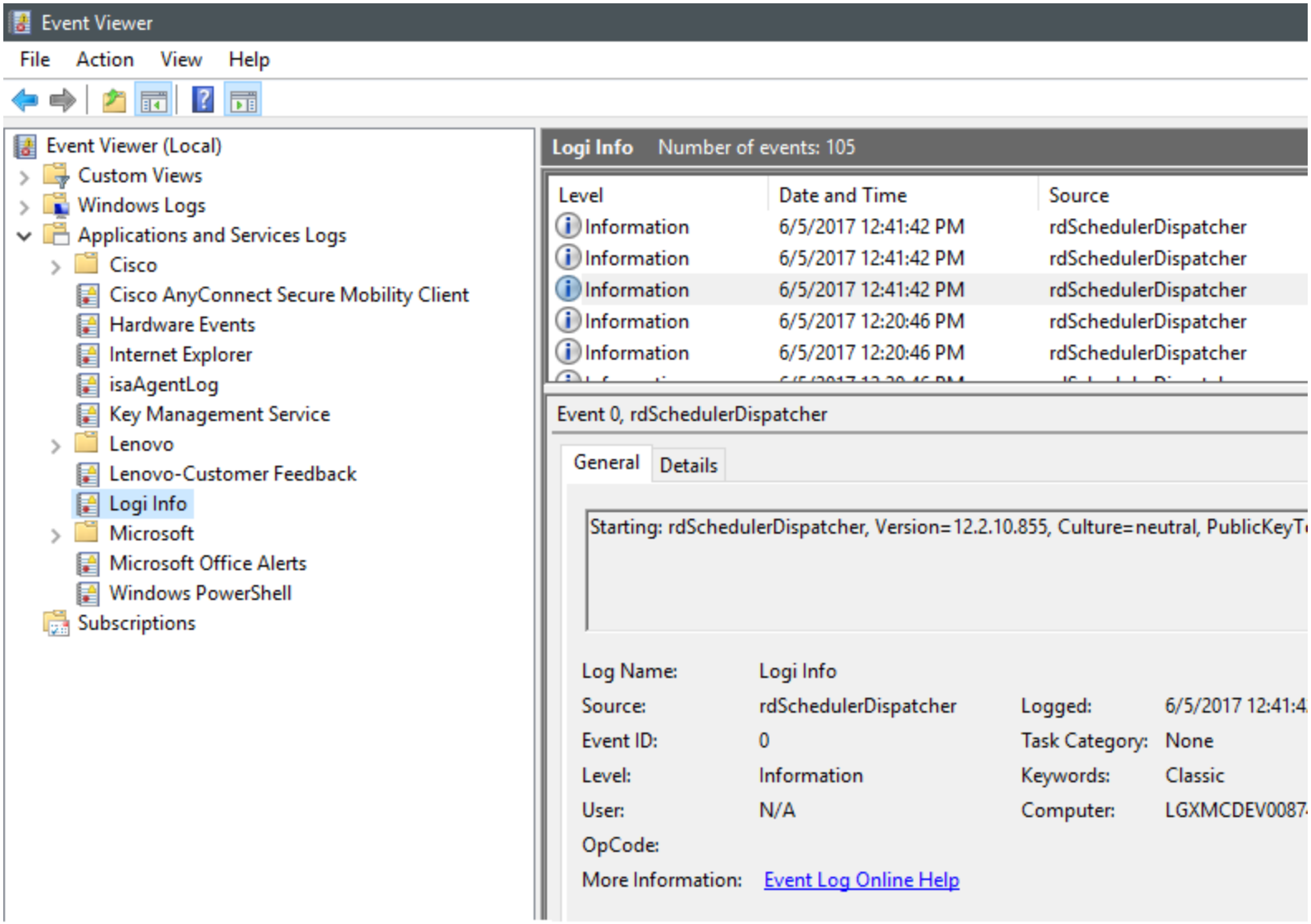
## Operational Log

As mentioned above, if the Scheduler's settings file is so configured, operational details will be written to a log.

The log is:

```
(Windows .NET) The Windows Event Log
(Windows Java)C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service Java\Log\LogiScheduler.log
(Linux/UNIX) <installFolder>/Log/LogiScheduler.log
```

The Java log file, "LogiScheduler.log", is a text file written using the Apache Log4j 2 Logging Library and it can be viewed with any generic text reader or log viewer tool. You must update to Log4j v2.15 to mitigate this 0 day vulnerability. For more information about this vulnerability, refer to this statement.


Event Viewer
File Action View Help
Event Viewer (Local)
Custom Views
Windows Logs
Applications and Services Logs
Cisco
Cisco AnyConnect Secure Mobility Client
Hardware Events
Internet Explorer
isaAgentLog
Key Management Service
Lenovo
Lenovo-Customer Feedback
Logi Info
Microsoft
Microsoft Office Alerts
Windows PowerShell
Subscriptions
Logi Info Number of events: 105
Level Date and Time Source
Information 6/5/2017 12:41:42 PM rdSchedulerDispatcher
Information 6/5/2017 12:41:42 PM rdSchedulerDispatcher
Information 6/5/2017 12:41:42 PM rdSchedulerDispatcher
Information 6/5/2017 12:20:46 PM rdSchedulerDispatcher
Information 6/5/2017 12:20:46 PM rdSchedulerDispatcher
Event 0, rdSchedulerDispatcher
General Details
Starting: rdSchedulerDispatcher, Version=12.2.10.855, Culture=neutral, PublicKeyT
Log Name: Logi Info
Source: rdSchedulerDispatcher Logged: 6/5/2017 12:41:4
Event ID: 0 Task Category: None
Level: Information Keywords: Classic
User: N/A Computer: LGXMCDEV0087
OpCode:
More Information: Event Log Online Help

The Windows Event Log, shown above, can be viewed using the **Event Viewer** administrative tool, and entries are listed in Applications and Service Logs, under "Logi Info".

You can automate the cleanup of scheduler logs by adding three attributes to the scheduler settings file:

1. `rdCleanUpSchedulerLog`: This attribute enables or disables Scheduler Log Clean up. The default value of this constant is "False", no clean up of Scheduler Logs. To automate clean up, set this value to "True".
2. `rdSchedulerLogCleanupInterval`: Set this attribute to specify the time, in minutes, between clean ups. Default value is 5 minutes. It's triggered by scheduler pings, not by the Info Engine.
3. `rdSchedulerLogLifespan`: The minimum file age, in minutes, before a file qualifies to be deleted when the clean up runs. The default value of this attribute is *60* and the unit of measurement is minute.

# Scheduled Task Data Storage Options

The Scheduler Service for .NET normally stores its scheduled task data using an embedded instance of **VistaDB,** a text-based database; for the Scheduler Service for Java, an embedded instance of **Apache Derby** is used.

The *scope* of the Scheduler database is significant, as it can contain records for scheduled events for *multiple* Logi applications on the same web server. It's important to keep this in mind when developing applications using the Scheduler, so that the correct application is identified when creating or modifying stored data.

For developers upgrading from early versions of Logi Info, which used the **Windows Task Scheduler** to schedule reports, there is no way to import or migrate schedules established in the Windows Task Scheduler into the Logi Scheduler. However, any legacy scheduled events created with the Windows Task Scheduler will still run in later Logi Info versions.

## Backing Up the Scheduler Database Files

As mentioned earlier, the default Logi Scheduler configuration stores scheduled event information in embedded database files that can be backed up using any standard file copy or backup methods.

The relevant files are:

```
(Windows .NET) C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service Java\Schedules.vdb3
(Windows Java)C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service Java\Schedules\*.*
(Linux/UNIX) <installFolder>/Schedules/*.*
```

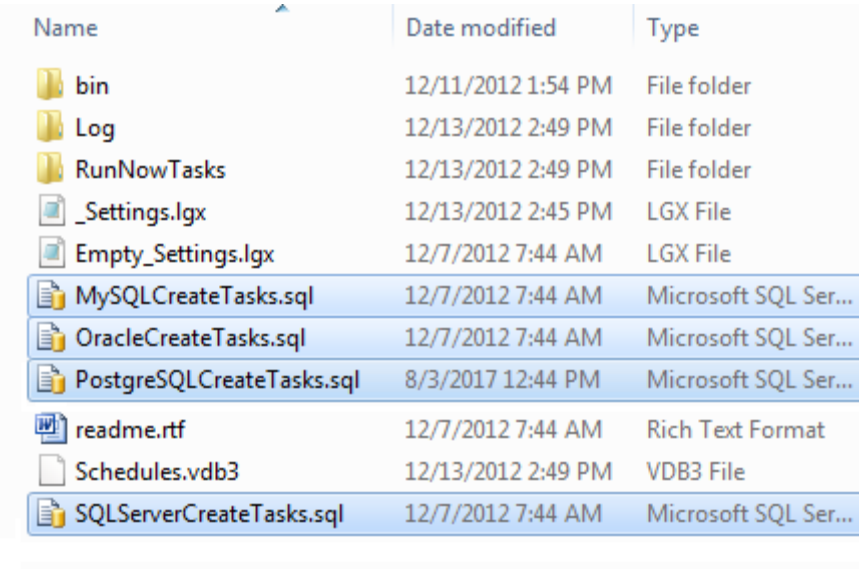
These files can even be backed up from within a Logi Process definition, so the Scheduler can schedule and run its own backup. See the Scheduler Console sample application for an example of how this is done.

# Using a Database Server

You have the option of storing Scheduler task data in a networked **Microsoft SQL Server**, **Oracle**, **MySQL** or, in Logi Info v12.5+, **PostgreSQL** database.

If you choose to use Microsoft SQL Server, for the best performance, we recommend that it be **SQL Server 2008** or later, in order to avoid the "8KB page size" issue found in older SQL Server versions. This script *will* also work with earlier SQL Server versions but you may experience performance degradation *if* the data in your fields exceeds 8KB *and* the number of stored Scheduler tasks is large (exceeding 8KB triggers behind-the-scenes row splitting in older SQL Server versions).

In order to use one of these SQL database servers, you must first create the Tasks table in a database of your choice, and then configure the Scheduler's `_Settings.lgx` file to use it.



In your Scheduler installation folder, four example SQL scripts shown above have been provided to help you create the Tasks table, and they include comments about their use. Modify a copy of the appropriate script and run it on your database server to create the table.

Using a *case-sensitive* database? The Scheduler service issues a SQL query that uses *mixed case*: "`SELECT TaskID, ProcessXml, RunAs FROM Tasks WHERE...`" so be sure to edit the example script you use to create the table to match this case.

```
<Setting>
<RemoteApi Port="56982" PassKey="myKey"/>
<WebRequest Timeout="20" ConcurrencyLimit="5" Stagger="1" LoggingLevel="WARNING"
/>
 <Connection Type="MySql" ID="MySQL" MySqlServer="YourDatabaseServerName"
 MySqlDatabase="YourDatabase" MySqlUser="YourUser" MySqlPassword="YourPassword" />
</Setting>
```

To complete the configuration, you must edit the Scheduler service's `_Settings.lgx` file to include a connection element for the database server you're going to use. The settings file is distributed with example connection elements in its comments section; copy the code for your database server type, paste it into the file as shown above, and configure it.

Stop and restart the Scheduler service to begin sending Scheduled task data to the database.

If you have concerns about the security of the database credentials stored in the Scheduler's `_Settings.lgx` file, you can use the Batch Obfuscation Tool (see *Using Logi 12 Studio*) installed with Logi Studio to obfuscate the file. This is done from a command prompt (Windows only) with a command like this (assumes default Logi Info installation location):

```
C:\Program Files\LogiXML IES Dev>LogiObfuscation "C:\Program Files\LogiXML IES Dev\LogiXML Scheduler Service\_Settings.lgx" yourPassword/obfuscate
```

A similar command, using the `/deobfuscate` argument, can be used to de-obfuscate the file for maintenance.

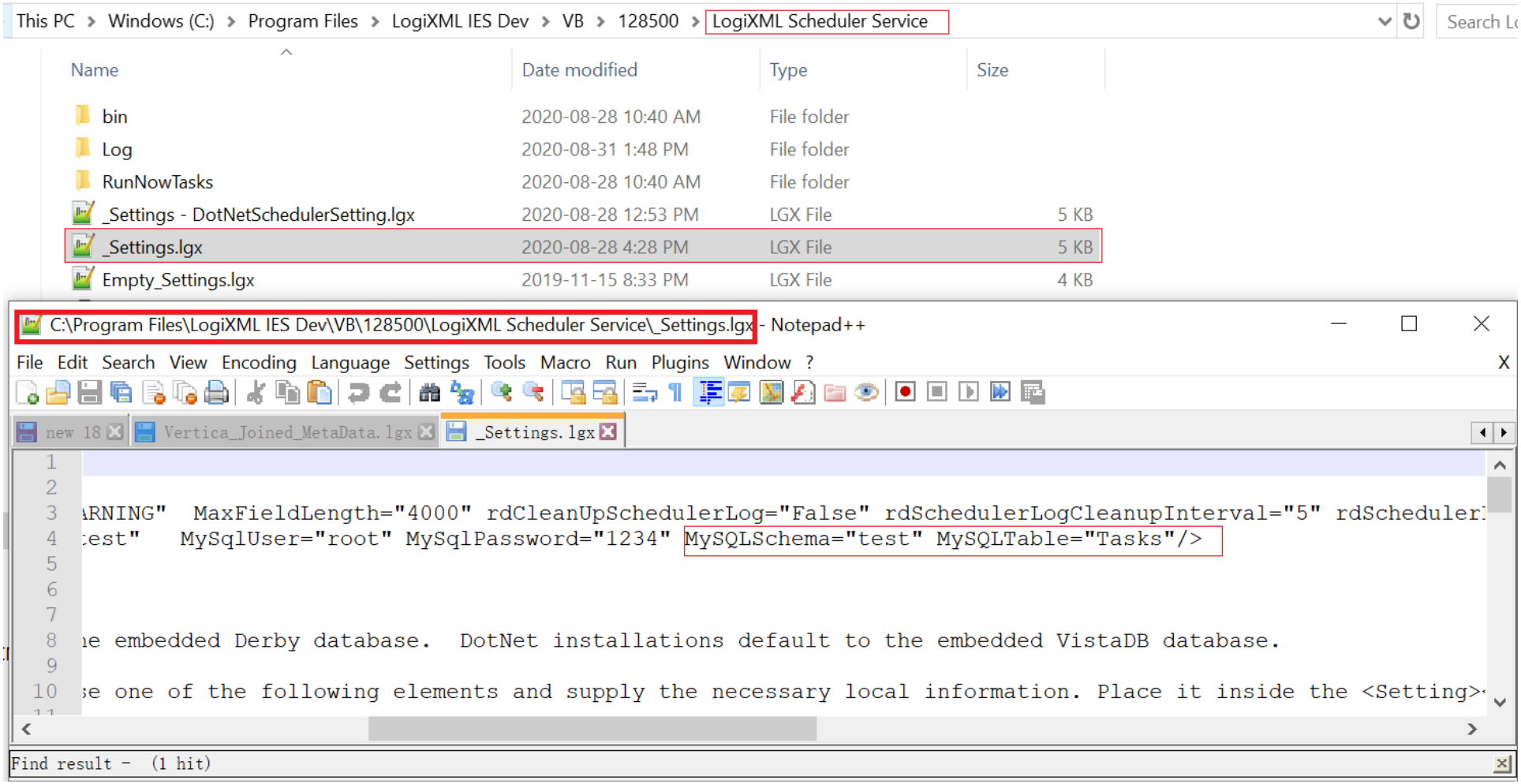
Logi Analytics has no way to de-obfuscate the file for you if you forget your password.

If you wish to transfer existing data from the default database files into a SQL database, you will find that there are export tools for both VistaDB and Derby, either built-in or available online, that allow you to export data in a variety of formats (CSV, XML, etc.) for subsequent insertion into SQL databases.

## Use Custom Table and Schema in Database

If you do not want to use the default table 'Tasks', you have the option to use a custom table and schema in your database. First you must add two new attributes, Schema and Table, to the Connection element via the settings file: `%LogiXML Scheduler ServiceHome%/_Settings.lgx`. By default, no Connection element is required. Java installations default to the embedded Derby database. DotNet installations default to the embedded VistaDB database.

If you want to use a networked database, then a Connection element is necessary. Choose one of the elements (Schema or Table) and supply the necessary local information. Place it inside the <Setting></Setting> tag to activate it, like below:




Only the MySql, Oracle, PostgreSQL and SQL Server databases are currently supported for networked Scheduler access.

### Sample MySQL Server Connection:

```
<Connection Type="MySql" ID="MySQL" MySqlServer="localhost" MySqlDatabase="test" MySqlUser="root" MySqlPassword="1234"
MySQLSchema="test" MySQLTable="Tasks 1"/>

<Connection Type="MySql" ID="MySQL" MySqlServer="localhost" MySqlDatabase="test" MySqlUser="root" MySqlPassword="1234"
/>
```

### Sample Oracle Connection:

```
<Connection Type="Oracle" ID="Oracle" OracleServer="qad10.jinfonet.com.cn" OracleDatabase="PDBORCL" OracleUser-
r="LOGIINFO" OraclePassword="TEST1234" />

<Connection Type="Oracle" ID="Oracle" OracleServer="qad10.jinfonet.com.cn" OracleDatabase="PDBORCL" OracleS-
chema="LOGIINFO" OracleTable="TASKS 3" OracleUser="LOGIINFO" OraclePassword="TEST1234" OracleSequence="TASKS 3_SEQ" />
```

### Sample SQL Server Connection:

```
<Connection Type="SqlServer" ID="SqlServer" SqlServer="qad10.jinfonet.com.cn" SqlServerDatabase="REPDEV22303"
SqlServerUser="REPDEV22303" SqlServerPassword="Test1234" />

<Connection Type="SqlServer" ID="SqlServer" SqlServer="qad10.jinfonet.com.cn" SqlServerDatabase="REPDEV22303"
SqlServerUser="REPDEV22303" SqlServerPassword="Test1234" Schema="REPDEV22303" Table="Tasks1"/>

<Connection Type="SqlServer" ID="SqlServer" SqlServer="qad10.jinfonet.com.cn" SqlServerDatabase="REPDEV22303"
SqlServerUser="REPDEV22303" SqlServerPassword="Test1234" Schema="REPDEV22303" Table="Task s2"/>
```

Sample PostgreSQL Connection:

```
<Connection Type="PostgreSQL" ID="PostgreSQL" PostgreSQLServer="qad13.jinfonet.com.cn" PostgreSQLDatabase="logiinfo"
PostgreSQLUser="repdev22303" PostgreSQLPassword="Test1234"/>

<Connection Type="PostgreSQL" ID="PostgreSQL" PostgreSQLServer="qad13.jinfonet.com.cn" PostgreSQLDatabase="logiinfo"
Schema="repdev22303" Table="tasks1" PostgreSQLUser="repdev22303" PostgreSQLPassword="Test1234"/>

<Connection Type="PostgreSQL" ID="PostgreSQL" PostgreSQLServer="qad13.jinfonet.com.cn" PostgreSQLDatabase="logiinfo"
Schema="repdev22303" Table="task s2" PostgreSQLUser="repdev22303" PostgreSQLPassword="Test1234"/>
```

# Implementing Multiple Scheduler Instances

Support for storing task data in networked databases, discussed in "Scheduled Task Data Storage Options" on page 163, allows **multiple instances** of the Scheduler service to run at the same time, on multiple servers, creating a fault-tolerant configuration. If one of the Scheduler instances goes down, the remaining instances will continue to function.

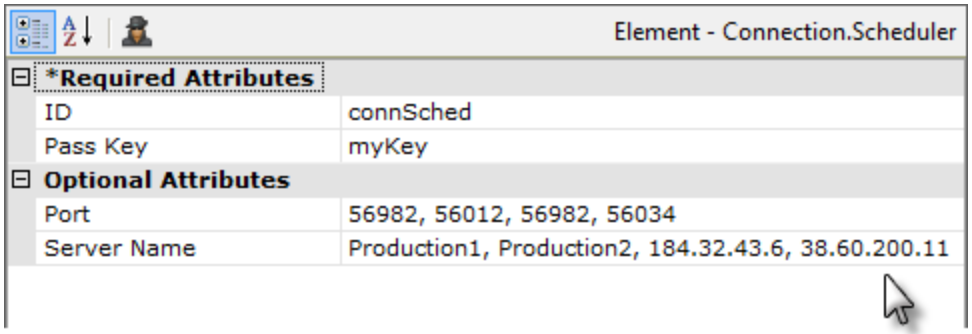
A multi-instance configuration is *not* supported when using the standard embedded VistaDB (.NET) or Derby (Java) file databases.

Assuming you've already set up data storage on networked database server, the first step in using multiple instances is to install the Scheduler service on multiple servers and configure the Scheduler's `_Settings.lgx` file as desired. To facilitate this, the Scheduler service can be installed independently of the full Logi Info product.

For the purpose of this discussion, let's assume you've installed the Scheduler on four separate servers, with these settings file values:

| Server Name or IP Address | Port | Pass Key | MySQLServer |
|---|---|---|---|
| Production1 | 56982 | myKey | adminData2 |
| Production2 | 56012 | myKey | adminData2 |
| 184.32.43.6 | 56982 | myKey | adminData2 |
| 38.60.200.11 | 56034 | myKey | adminData2 |

Note that each of the Scheduler services has been configured to access the same networked database server where the Scheduler task data is stored.




Now, to engage the instances, in the application's _Settings definition, the Connection.Scheduler element's **Port** and **Server Name** attributes must be given a comma-delimited string of the ports and names/addresses, as shown above. Notice that the first port number in the list correlates to the first server name in the list, the second port to the second server, and so on.

Having multiple Scheduler service instances will *not* result in a task being run more often than its designated frequency.

# Customizing Scheduler Appearance

The Schedule 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 special tags that will override elements in the template file, changing the appearance and some functionality.

More detailed information about template modifier files can be found in *Template Modifier Files*.

# Combine Text and Data

There are several flexible ways to combine and display text and data in a Logi report.

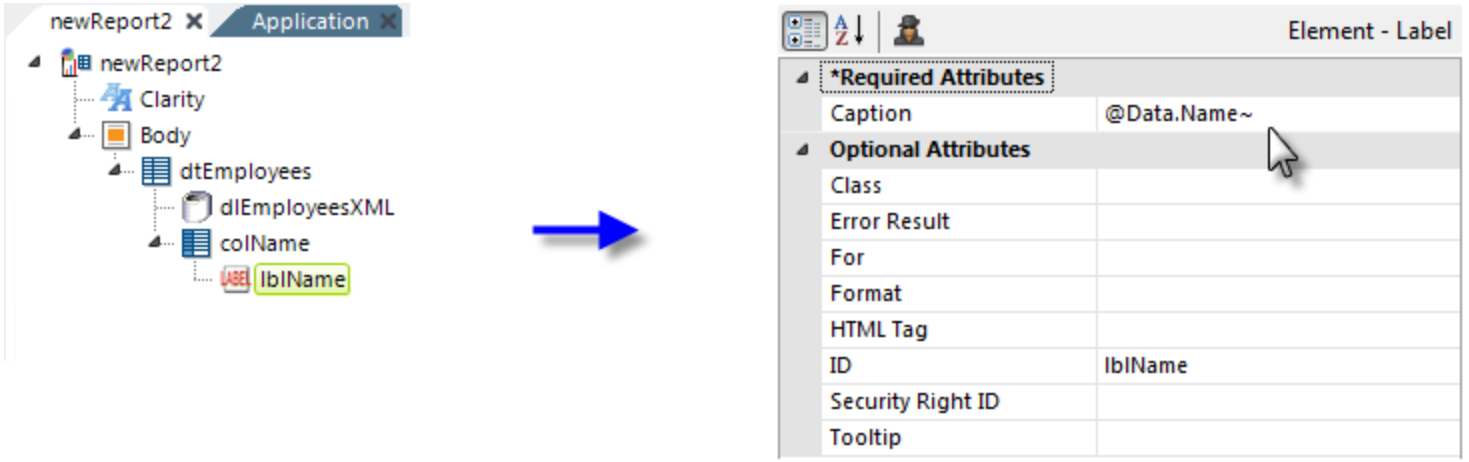
The following topics discuss these techniques:

# Combining Data in a SQL Query

In all of the following examples, the goal is to retrieve FirstName and LastName text data and combine it into a full name. One of the easiest ways to this is to let the database server do it for you. This SQL query:

```
SELECT FirstName + ' ' + LastName AS Name FROM Employees
```

returns a single column, **Name**, which will contain the FirstName and LastName data, separated by a space. Like any other column in the datalayer, the Name column value is accessible using an @Data token:



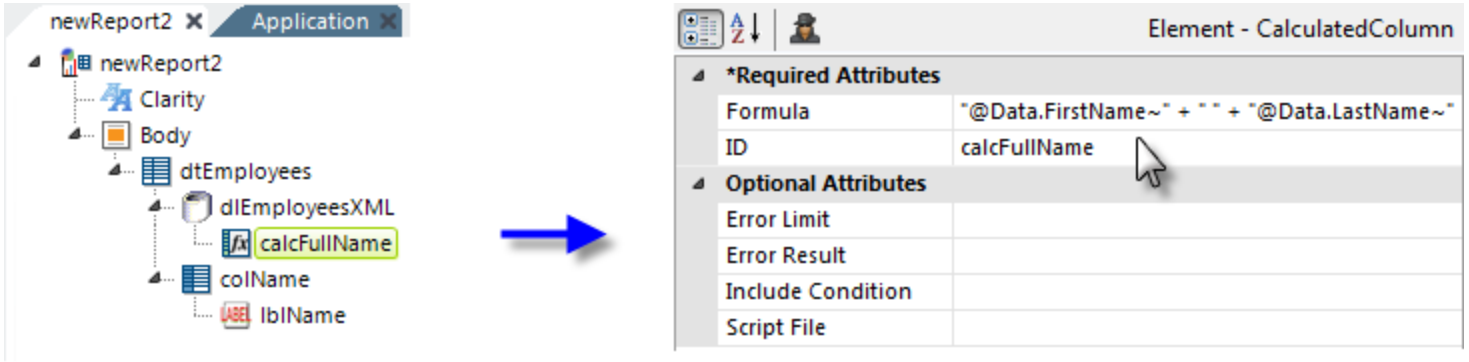

After using the SQL query shown earlier, we can display the Name data in a Data Table column using **Label** element whose **Caption** attribute is set to @Data.Name~, as shown above.

# Using a Calculated Column

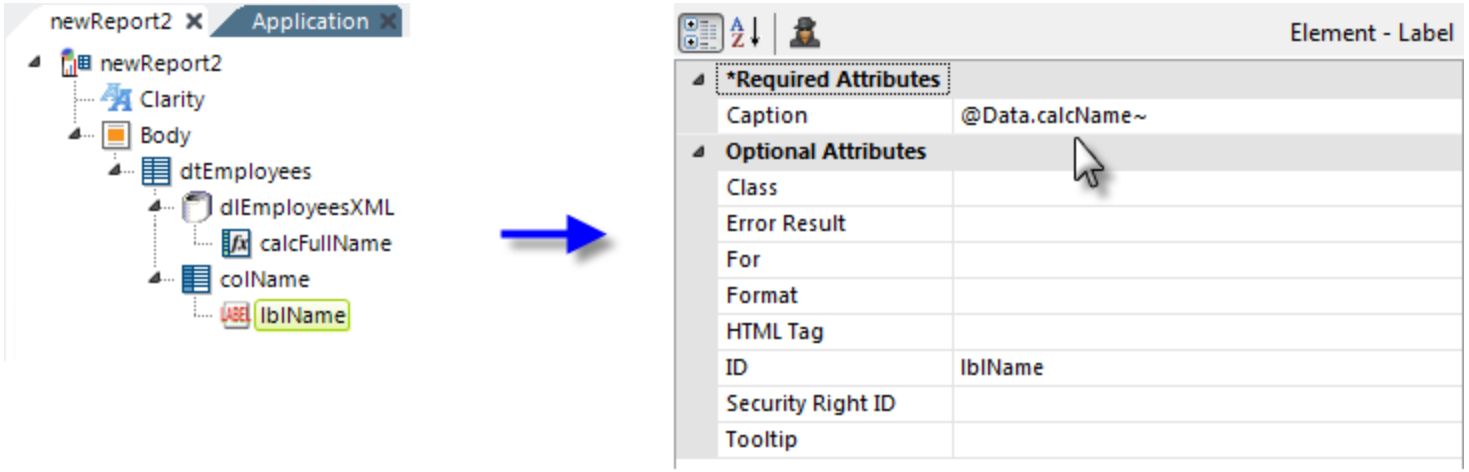
The **Calculated Column** element adds a column to a datalayer and is another fine way to synthesize data, i.e. the full name, not found in the database. In this case, the SQL query is straightforward:

```
SELECT FirstName, LastName FROM Employees
```

so both columns are returned to the datalayer.



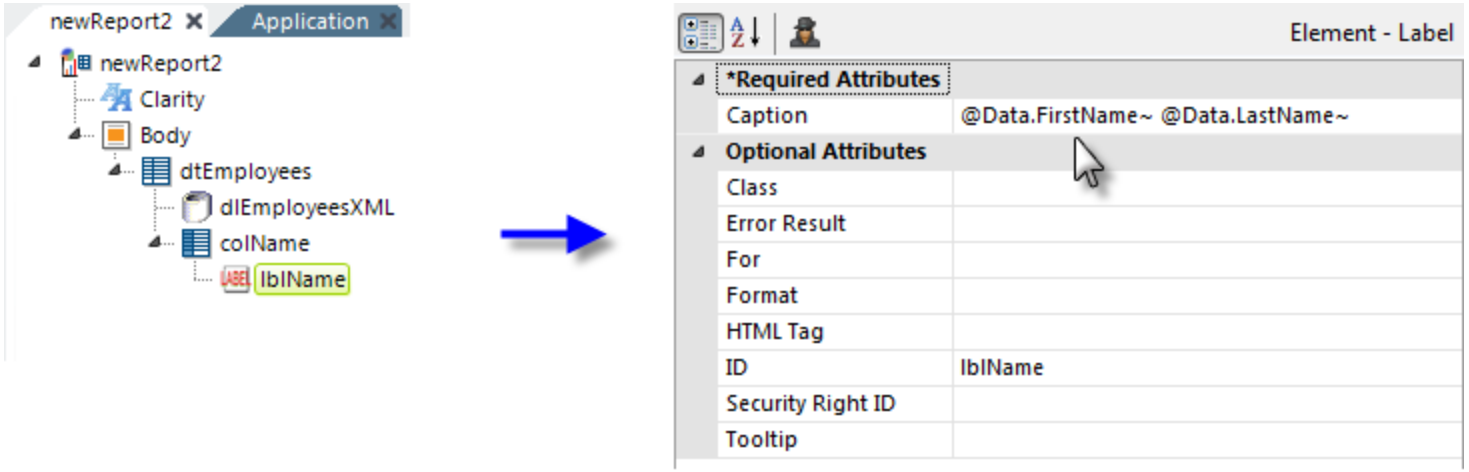

As shown above, a **Calculated Column** element has been added as a child of the datalayer. Its **Formula** attribute value is set to concatenate the two name columns, with a space separating them. Its **ID** attribute becomes the name of the new column added to the datalayer.




The **Label** element that will display the full name, shown above, references the Calculated Column's **ID** in an @Data token to retrieve that data.

# Displaying Tokens Adjacent to One Another

One of the simplest way to concatenate text is simply to include their @Data tokens **adjacent** to one another. If we use the SQL query from the previous example, so that two columns are returned, but choose *not* to use a Calculated Column element, we can achieve the same results with this method.

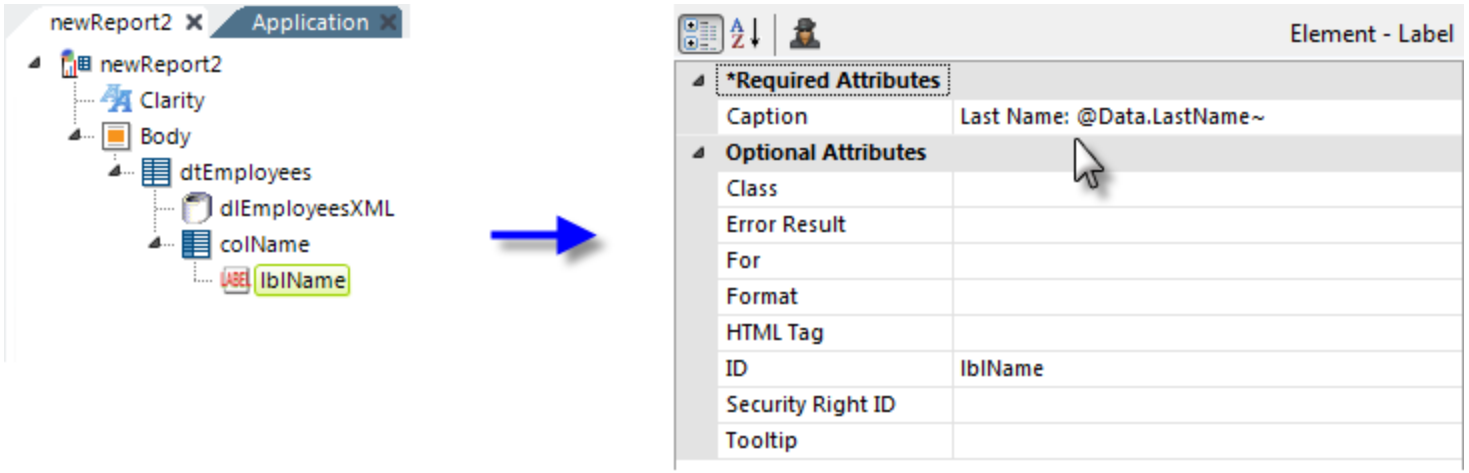



As shown above, this time the **Caption** attribute of the **Label** element is set to both @Data tokens with a space between them. When tokens are resolved, the two names will be displayed along with the separating space.

As you may know, browsers will usually render *one* space within text, but will ignore any other consecutive spaces. So don't bother inserting multiple spaces in a Caption within text or around tokens; all of them after the first one will be ignored by the browser.

# Displaying Literal Text and Data

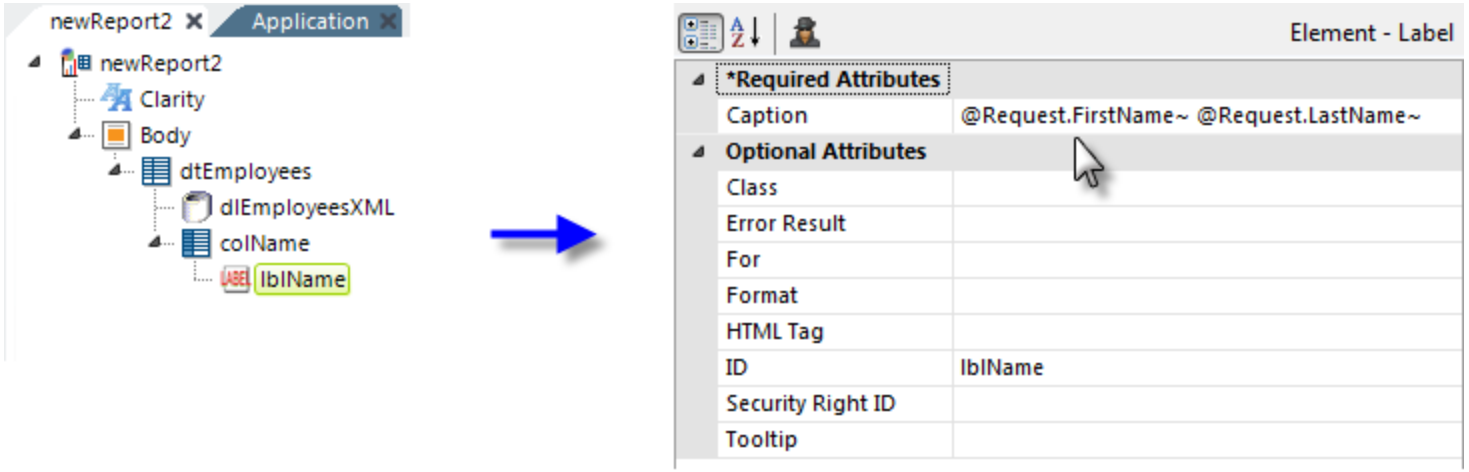
In the same way that tokens can be combined, as discussed in "Displaying Tokens Adjacent to One Another" on the previous page, literal text and tokens can be shown together:




As shown above, the literal text "Lat Name:", and a separating space, is entered, followed by a Data token. When the page is rendered, the token will be replaced by the data value and will be preceded by the literal text in each table row.

# Displaying Request Tokens

Text data, of course, can come from sources other than a database. One frequent source is the **Query String** used to call the current page. Data is often passed in "request variables" in the Query String and these are available using @Request tokens in your report.




In the example above, the **Label** element is used to display two @Request tokens, once again, entered adjacent to one another and separated by a single space.

# Sample Definition

The following is a **sample definition** you can experiment with. Just select all the text, copy and paste it into Notepad, then save it with an .LGX extension in the _Definitions_Reports folder of a test application. If the application is already open in Studio, right-click the Reports folder in the Application panel and select "Refresh Application" to make the sample definition appear in the list of report definitions.

The sample definition you have a connection to the Northwind database and that it's thefirst Connection in the application's _Settings definition. The blue.css style sheet, included with your Logi product installation, is also used.

```
>/Report<

> /"" myLastName=""ideTestParams myFirstName=<

>ReportFooter /<

>/Body<

> /"lblName" ID="@Request.myFirstName~ @Request.myLastName~"Label Caption=<

>LineBreak /<

> /"lblInstruction4" ID="4. This label concatenates two text values from request tokens."Label Caption=<

>LineBreak /<
```

```
>/DataTable<

>/DataTableColumn<

> /"lblName" ID="@Data.FirstName~ @Data.LastName~"Label Caption=<

>"cell" Class="Employee Name" Header="colName3"DataTableColumn ID=<

>/DataTableColumn<

> /"lblEmpID" ID="@Data.EmployeeID~"Label Caption=<

>"cell" Class="Employee ID" Header="colEmpID3"DataTableColumn ID=<

> /"Collapsed" IdeDisplayStatus="Select * From Employees" Source="dlEmployees3" ID="SQL"DataLayer Type=<

>"Collapsed" IdeDisplayStatus="tableheader" ColumnHeaderClass="table" Class="dtEmployees3"DataTable ID=<

> /"lblInstructionTable3" ID="3. This table uses @Data tokens to display the First Name and Last Name togeth-
er."Label Caption=<

>LineBreak /<

>/DataTable<
```

```
>/DataTableColumn<

> /"lblName" ID="@Data.calcName~"Label Caption=<

>"cell" Class="Employee Name" Header="colName2"DataTableColumn ID=<

>/DataTableColumn<

> /"lblEmpID" ID="@Data.EmployeeID~"Label Caption=<

>"cell" Class="Employee ID" Header="colEmpID2"DataTableColumn ID=<

>/DataLayer<

> /"calcName" ID="quot;&quot;@Data.LastName~&quot; + &quot; &quot;+ &quot;@Data.FirstName~&"CalculatedColumn For-
mula=<

>"Select FirstName, LastName, EmployeeID From Employees" Source="dlEmployees2" ID="SQL"DataLayer Type=<

>"Collapsed" IdeDisplayStatus="tableheader" ColumnHeaderClass="table" Class="dtEmployees2"DataTable ID=<

> /"lblInstruction2" ID="2. This table uses a calculated column to concatenate the First Name and Last Name
together."Label Caption=<

>LineBreak /<
```

```
>/DataTable<

>/DataTableColumn<

> /"lblName" ID="@Data.Name~"Label Caption=<

>"cell" Class="Employee Name" Header="colName"DataTableColumn ID=<

>/DataTableColumn<

> /"lblEmpID" ID="@Data.EmployeeID~"Label Caption=<

>"cell" Class="Employee ID" Header="colEmpID"DataTableColumn ID=<

> /"Collapsed" IdeDisplayStatus="Select FirstName + ' ' + LastName As Name, EmployeeID From Employees" Source-
e="dlEmployees1" ID="SQL"DataLayer Type=<

>"Collapsed" IdeDisplayStatus="tableheader" ColumnHeaderClass="table" Class="dtEmployees1"DataTable ID=<

> /"lblInstruction1" ID="1. This table uses SQL to concatenate the First Name and Last Name together."Label Cap-
tion=<

>Body<
```

```
>ReportHeader /<

> /"blue.css"StyleSheet StyleSheet=<

> /"Walton" myLastName="Sam"DefaultRequestParams myFirstName=<

>"11.4.046" EngineVersion="4/4/2008 1:30 PM" SavedAt="Sam" SavedBy="CombinedText"Report ID=<
```

# SubReports

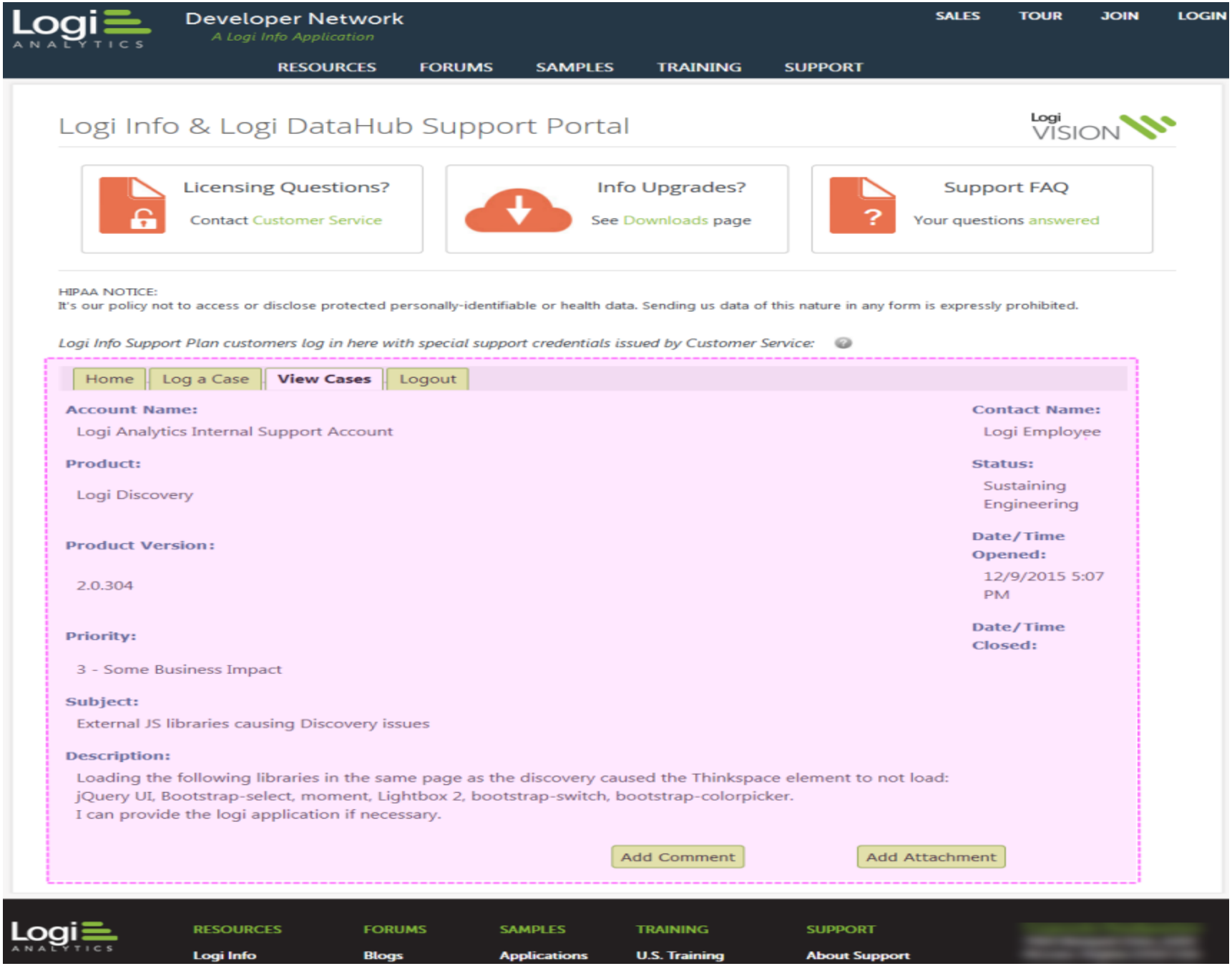
Logi developers often find a need to embed other reports inside their Logi reports, either as readily visible parts of the page or as hidden areas that are displayed on demand.

The following topics discuss use of the SubReport element:

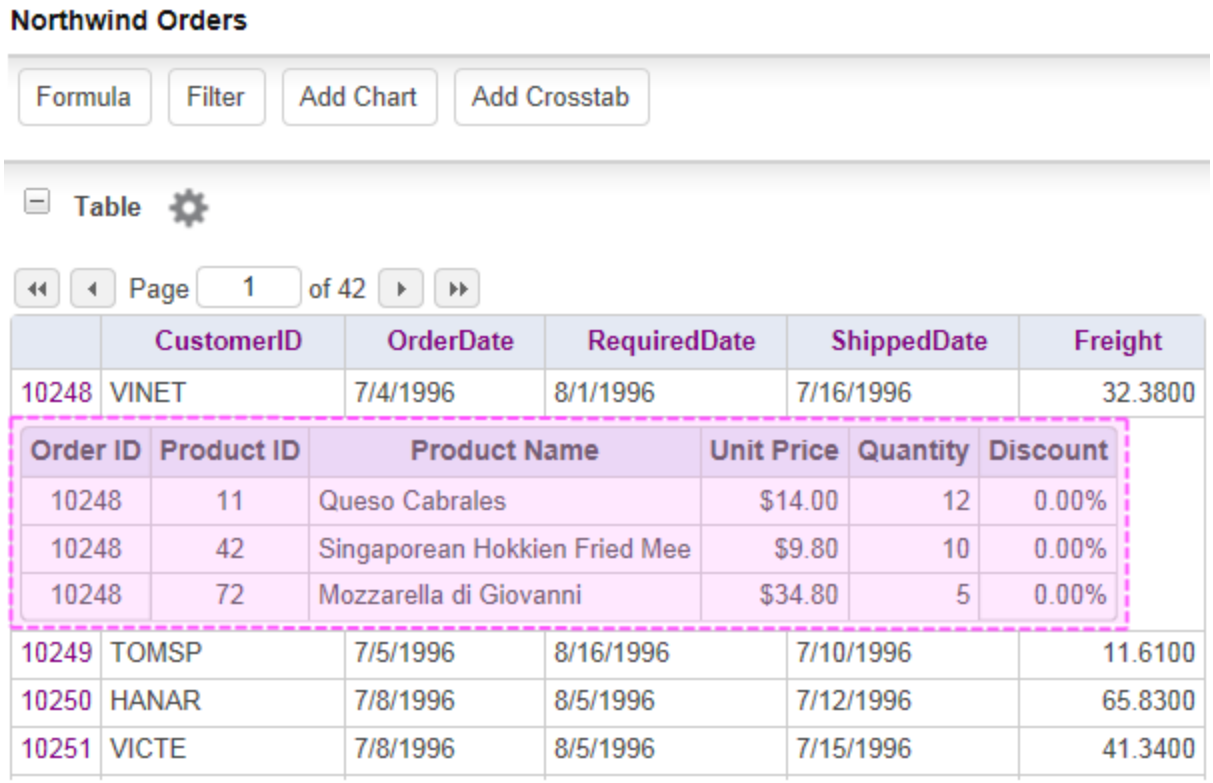


## About SubReports

A "subreport" in Logi parlance is an HTML page that's displayed *inside* a Logi report page. This could be a different Logi report page, a page from another Logi application, or even a page from an external website. The method discussed here for doing this uses the **SubReport** element. Other available methods, which are not discussed here, include use of the Include Html File (see "Insert HTML into Reports" on page 427) element and our JavaScript-based *Embedded Reports API*. For non-hierarchical tabular data, using a subreport with a Data Table is recommended instead of using a **SubData Table**. Let's look at two examples of the SubReport element in action:


Developer Network
A Logi Info Application
SALES
TOUR
JOIN
LOGIN
RESOURCES
FORUMS
SAMPLES
TRAINING
SUPPORT
Logi Info & Logi DataHub Support Portal
Logi VISION
Licensing Questions?
Contact Customer Service
Info Upgrades?
See Downloads page
Support FAQ
Your questions answered
HIPAA NOTICE:
It's our policy not to access or disclose protected personally-identifiable or health data. Sending us data of this nature in any form is expressly prohibited.
Logi Info Support Plan customers log in here with special support credentials issued by Customer Service:
Home
Log a Case
View Cases
Logout
Account Name:
Logi Analytics Internal Support Account
Contact Name:
Logi Employee
Product:
Logi Discovery
Status:
Sustaining Engineering
Product Version:
2.0.304
Date/Time Opened:
12/9/2015 5:07 PM
Priority:
3 - Some Business Impact
Date/Time Closed:
Subject:
External JS libraries causing Discovery issues
Description:
Loading the following libraries in the same page as the discovery caused the Thinkspace element to not load:
jQuery UI, Bootstrap-select, moment, Lightbox 2, bootstrap-switch, bootstrap-colorpicker.
I can provide the logi application if necessary.
Add Comment
Add Attachment
RESOURCES
Logi Info
FORUMS
Blogs
SAMPLES
Applications
TRAINING
U.S. Training
SUPPORT
About Support

In the first example, shown above, the subreport is the shaded area inside the dotted lines, embedded into the DevNet report page. DevNet is a Logi application and, in the example, the Support Plan Portal content (once upon a time a service provided by a third-party vendor) was embedded using a SubReport element. The embedded web pages, forms, and reports go right into the DevNet page. This technique allows us to maintain a consistent "look and feel" for the DevNet page while taking advantage of external web-based applications and services.




The secondexample, above, shows an **Analysis Grid** super-element that uses a child More Info Row (see *Working with "More Info Rows"*) element to display detail data. Clicking the *Order ID* data causes a space to open in the table beneath the clicked row

and a subreport, shown in the shaded area within the dotted lines, is displayed in it. The subreport consists of a separate Logi report definition that contains only a Data Table. One useful aspect of this technique is that it allows the subreport definition to be independently developed and tested, and then integrated into the Analysis Grid report definition. As we've seen, the SubReport element is capable of including a subreport that's either a separate HTML page from the same or another web site, or another Logi report definition. The functionality of any pages included is preserved, so data entry, links, video, etc. are all active. Several sub-report *modes* are used for including files: *Embedded, IncludeFrame,* and *IncludeFrameAsynch*. These and the elements needed to create the two examples we've just seen are described in "Using IncludeFrame Mode" on page 194 and "Using IncludeFrameA-synch Mode" on page 198.

# SubReport Attributes

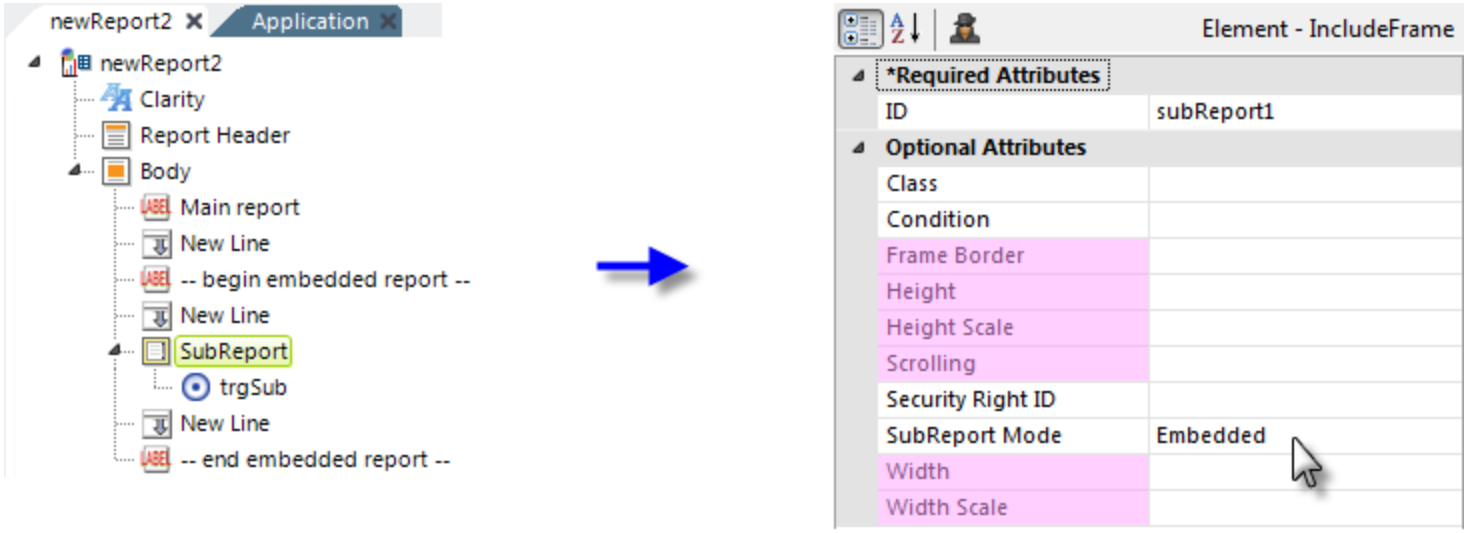
The SubReport element has the following attributes:

| Attribute | Description |
|---|---|
| ID | (Required) A unique element identifier. |
| Class | Specifies a style class to be applied to the frame. May be overridden if a Theme is used. |
| Condition | Specifies an expression that evaluates to a value of *True* or *False*. If *True*, then the column is displayed, otherwise it's removed. Expressions should be in JavaScript or intrinsic function syntax. Typically, expressions compare values using a token, such as @Data.value~ < 0. Enclose both sides of the expression in quotes when working with strings: "@Data.myColumn~" == "SomeValue". |
| Frame Border | Specifies if a border will be drawn around the frame. The default value is *True*. |
| Height Height Scale | Specifies a *fixed* height for the frame. Leaving this value blank will cause the frame to dynamically resize itself based on its content. The adjacent **Height Scale** attribute specifies whether this value is being specified in pixels or as a percentage. |
| Scrolling | Specifies whether scroll bars will appear within the frame. The default value is *Auto*. |
| Security Right ID | Controls access to the frame using Logi Security. Provide the ID of a Right defined or determined in the application's _Settings definition and only users that have that Right will be able to see the frame and its contents. Multiple Right IDs, separated by commas, may be entered. |

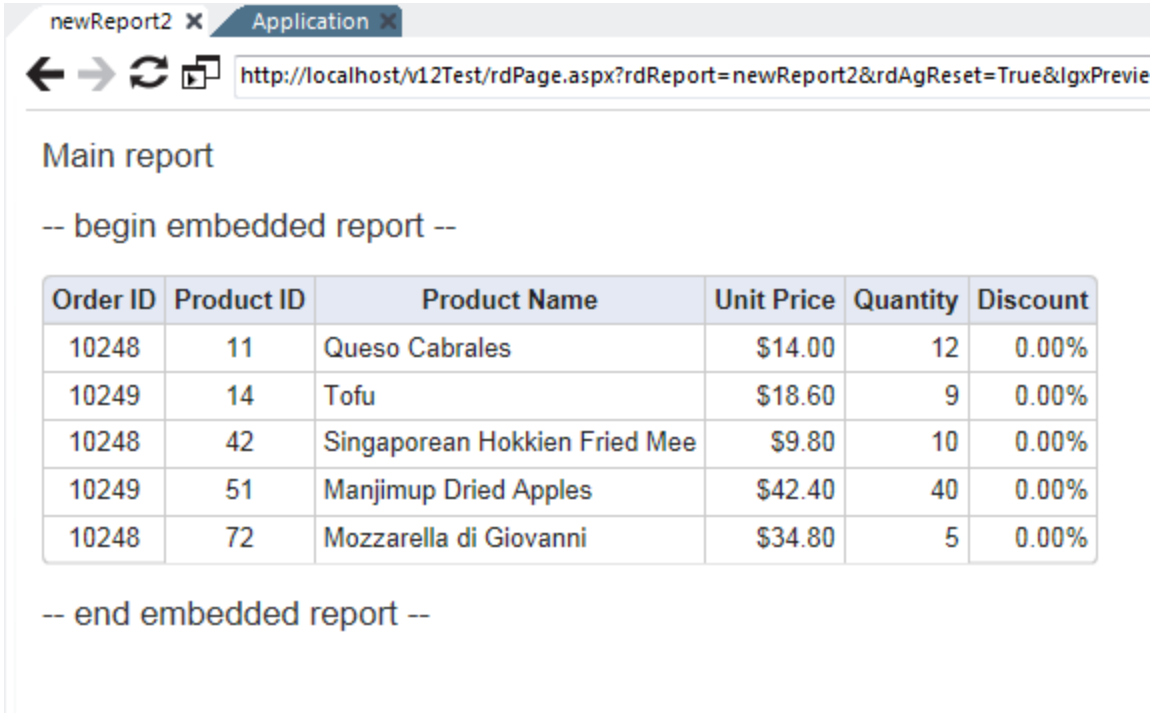
| Attribute | Description |
|---|---|
| SubReport Mode | Specifies the mode to be used to insert the subreport into the parent report. See "Using Embedded Mode" on the next page, "Using IncludeFrame Mode" on page 194, and "Using IncludeFrameAsynch Mode" on page 198 for detailed explanations of the three modes. |
| Width<br>Width Scale | Specifies a *fixed* width for the frame. Leaving this value blank will cause the frame to dynamically resize itself based on its content. The adjacent **Width Scale** attribute specifies whether this value is being specified in pixels or as a percentage. |

# Using Embedded Mode

When the SubReport element's **SubReport Mode** attribute is set to *Embedded*, the subreport *must* be a Logi report definition.




In the example above, the SubReport element has been added and its **SubReport Mode** attribute value has been set to *Embedded*. With this mode setting, the attributes that have been covered with a shaded block are *ignored* and a **Target.Report** child element *must* be used to specify the Logi report definition to be embedded. Other types of Target elements may not be used.




When the Logi Server Engine renders the page, it will render the embedded report *first* and then *merge* its HTML code right into the HTML code for the main report as it generates it, surrounding it with <DIV> tags. The page resulting from the sample code looks like the above.

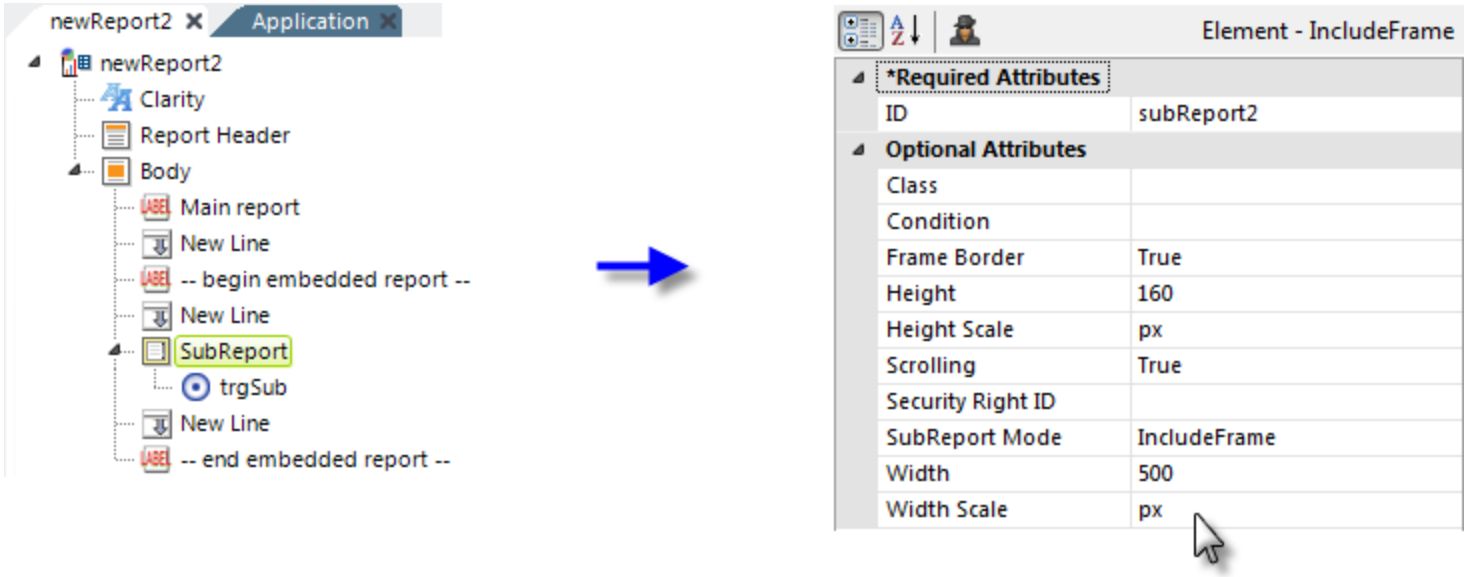
Keep these restrictions in mind when using this mode:

- Do not embed report definitions containing super-elements, such as the **Analysis Grid**, using this mode. Internal component naming and session variable conflicts will arise and the super-element will not work correctly.
- Do not embed report definitions that use the **Wait Panel** element when using this mode.
- Any **Sort** elements used beneath individual Data Table columns and **Interactive Paging** elements in the subreport definition will be *ignored* and will not be rendered.

- You can add a **Style** element to the subreport but it may be overridden by the main report's style.
- Drilling down using subreports in this mode is limited to two levels. In other words, you cannot drill down through one report to a subreport, which itself embeds another subreport, then drill down again using links in the second embedded subreport. Those links will not work correctly.

# Using IncludeFrame Mode

When the SubReport element's **SubReport Mode** attribute is set to *IncludeFrame*, the subreport may be a Logi report definition or an external HTML page. When the Logi Server Engine renders the main report, it will include <IFRAME> tags that reference the subreport. The subreport then runs in its own little "window" (which is an isolated, separate environment) inside the main report page. Super-elements, such as the Analysis Grid, *will* operate correctly when included using this mode.




In the example shown above, the SubReport element's other attributes are now usable and can be configured to constrain the subreport to a specific size, to add scrollbars, or to draw a border around the subreport. If the **Height** and **Width** attributes are left blank, the window will be automatically sized to include the entire subreport page.

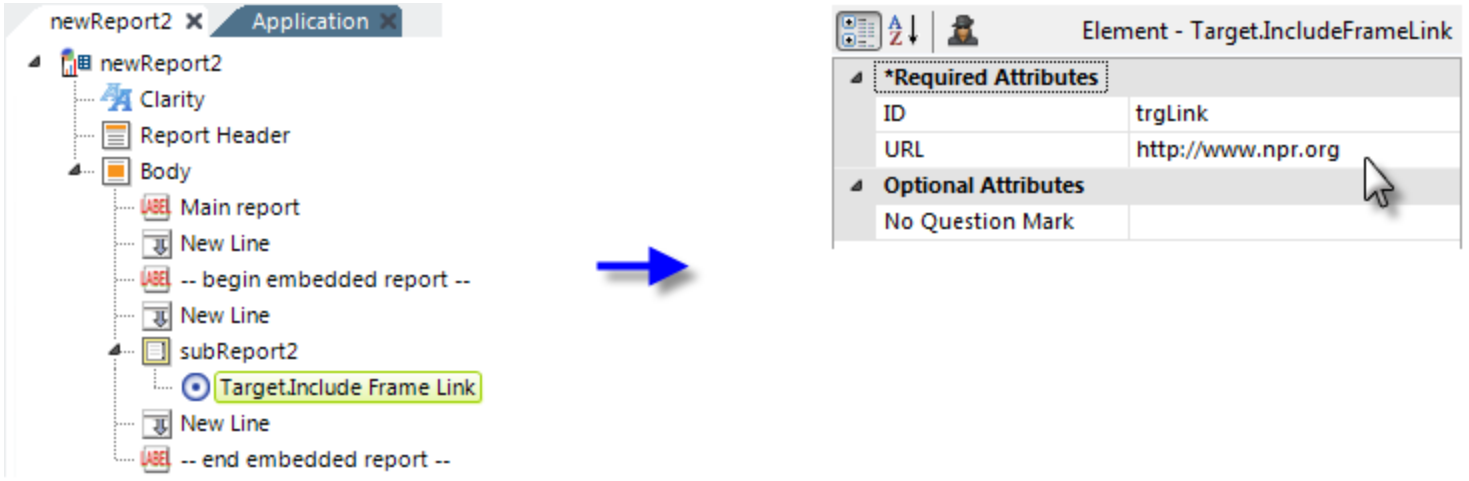
Main report

-- begin embedded report --

| Order ID | Product ID | Product Name | Unit Price | Quantity | Discount |
|---|---|---|---|---|---|
| 10248 | 11 | Queso Cabrales | $14.00 | 12 | 0.00% |
| 10249 | 14 | Tofu | $18.60 | 9 | 0.00% |
| 10248 | 42 | Singaporean Hokkien Fried Mee | $9.80 | 10 | 0.00% |
| 10249 | 51 | Manjimup Dried Apples | $42.40 | 40 | 0.00% |

-- end embedded report --

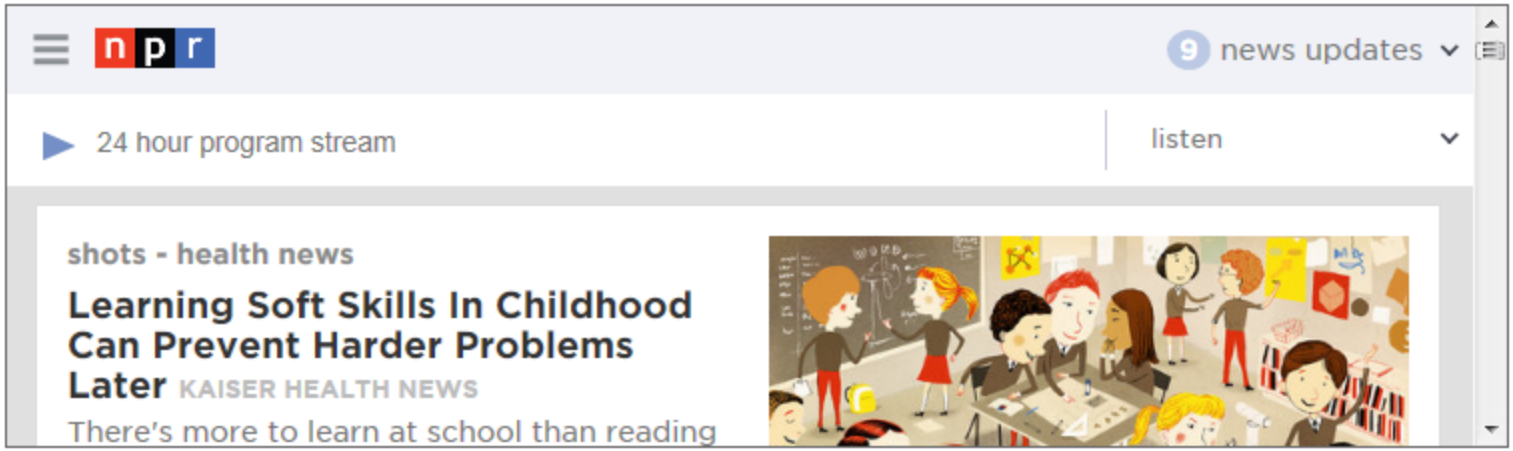
In the example shown above, the subreport is displayed in a bordered window inside the main report, restricted to a specific size, and with its own scrollbar.



If, instead of a Target.Report element, we use a **Target.Include Frame Link**, we can enter an actual URL and include an HTML page from an internal or external web site.

Main report

-- begin embedded report --



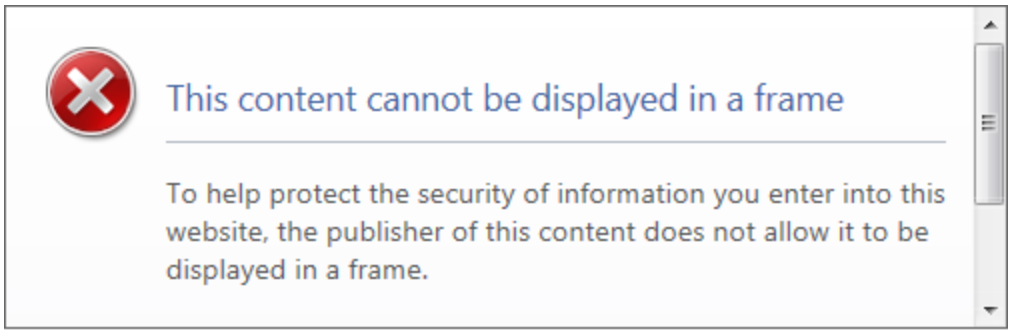

-- end embedded report --

And here we see the www.NPR.org home page displayed as a subreport.

You must always start the URL attribute value with "`http://`" to ensure that the Logi Server Engine recognizes it as a URL.

## Restrictions: X-Frame-Options

Creators of external content may choose to prevent you from embedding their pages in an iFrame by using the X-Frame-Options header in their HTTP Responses.




Browsers typically respond to this by displaying a message indicating that embedding is not allowed, like the one shown above, or by displaying nothing. If necessary, you can use online or built-in browser tools (often F12) to view the HTTP headers and confirm the presence of X-Frame-Options. Unfortunately, there is little you can do to overcome this restriction other than lobby the content provider for an exception.

## Style Considerations

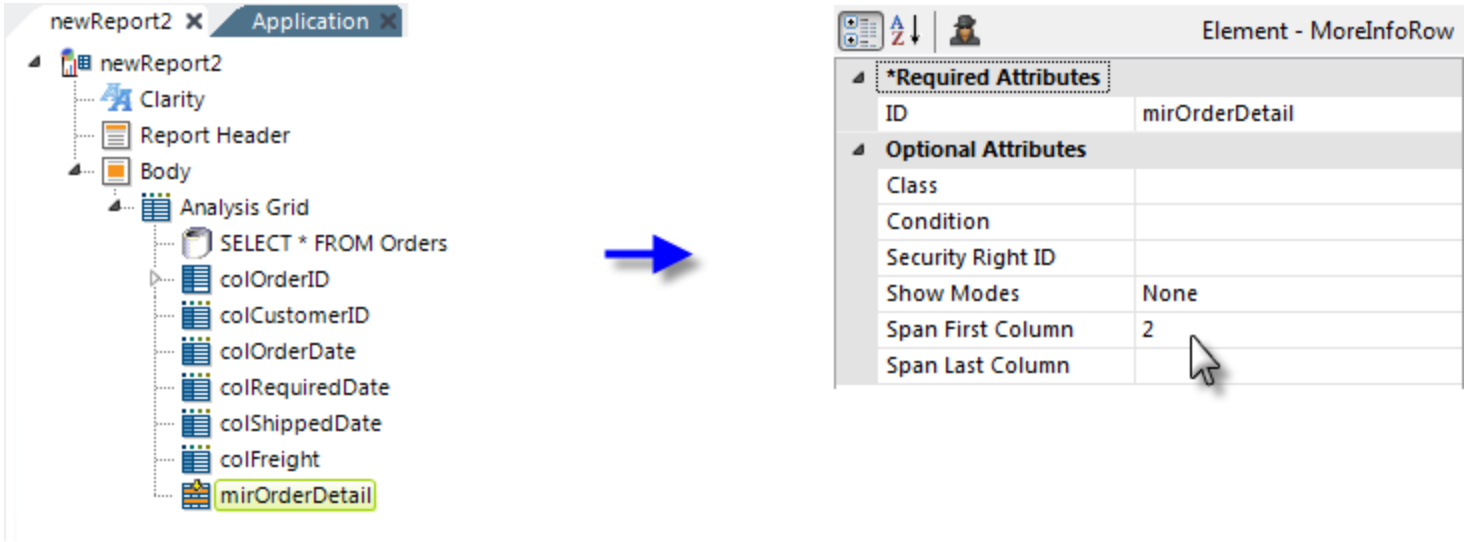
The style of your subreport may be affected by the style of your main report. Generally, *IncludeFrame* mode helps isolate the subreport from the main report's style, but you may need to work with your style sheets to iron out occasional conflicts.

# Using IncludeFrameAsynch Mode

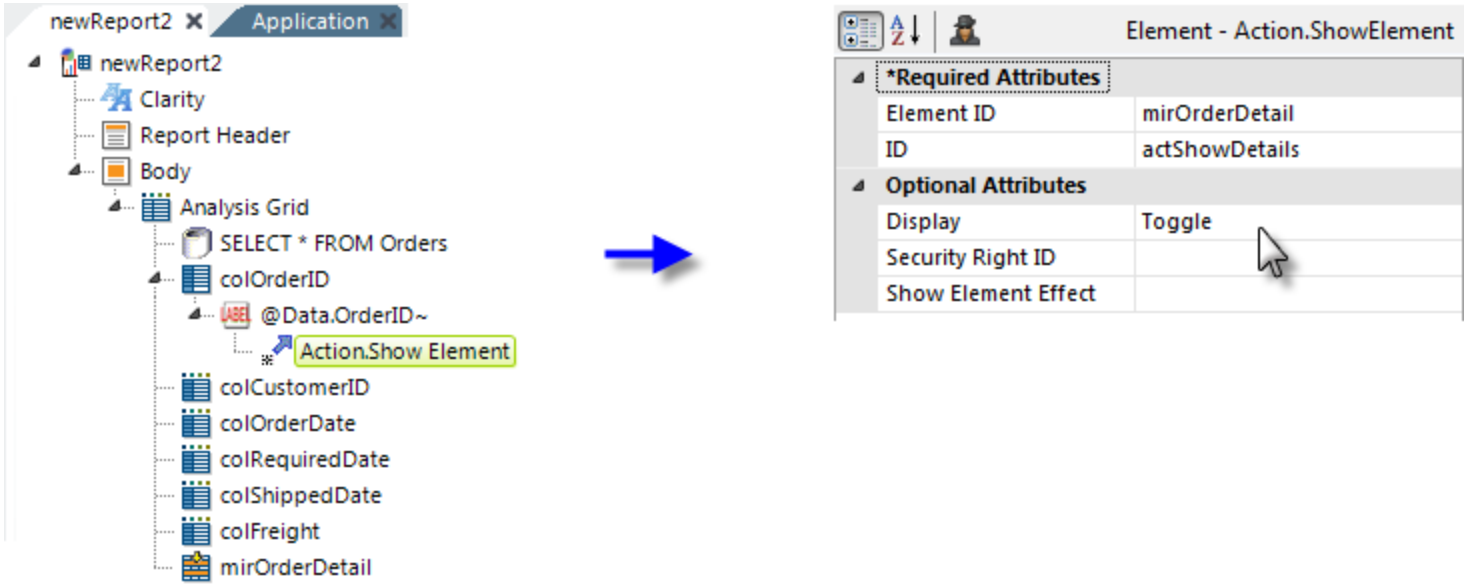
This SubReport Mode option is similar to *IncludeFrame* mode, except each request to the web server will be processed in its own thread. This may provide improved performance, for example when SubReports are used in multiple Dashboard panels, with the caveat that session variables *cannot* be saved for SubReports run in this mode.

# Using a SubReport with More Info Row

We've seen the elements and attributes used to place a subreport in the *body* of a main report, and now let's see how we can use a subreport to display information on demand, inside a Data Table row. In this topic, we'll see it in use with a More Info Row element and an Analysis Grid. For more information on More Info Row, see *Working with "More Info Rows"*.



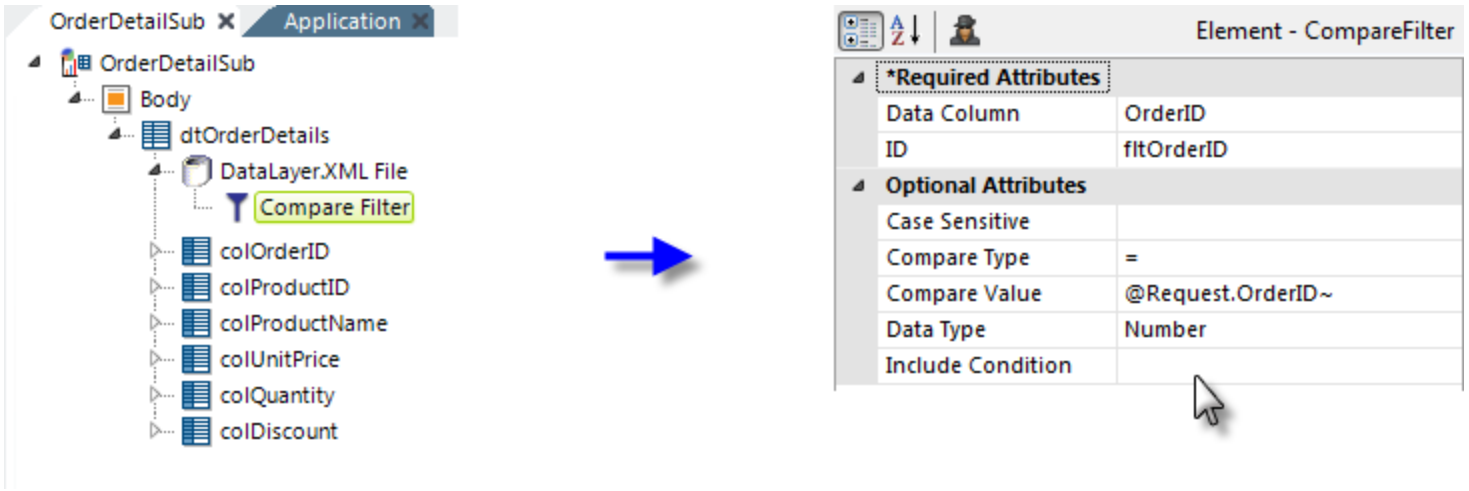

The example above shows an Analysis Grid (AG) and its columns. A **More Info Row** element has been added beneath the AG and configured, as shown above, to span all of the AG columns and to be hidden, using Show Modes, by default. By default, the width of a subreport embedded in a More Info Row will be limited by the width of the parent AG and will start under the first column of the AG. You can configured this differently by using the More Info Row element's **Span First Column** and **Span Last Column** attributes to adjust the placement of the subreport. In the example above, we're going to start the detail subreport in column 2.



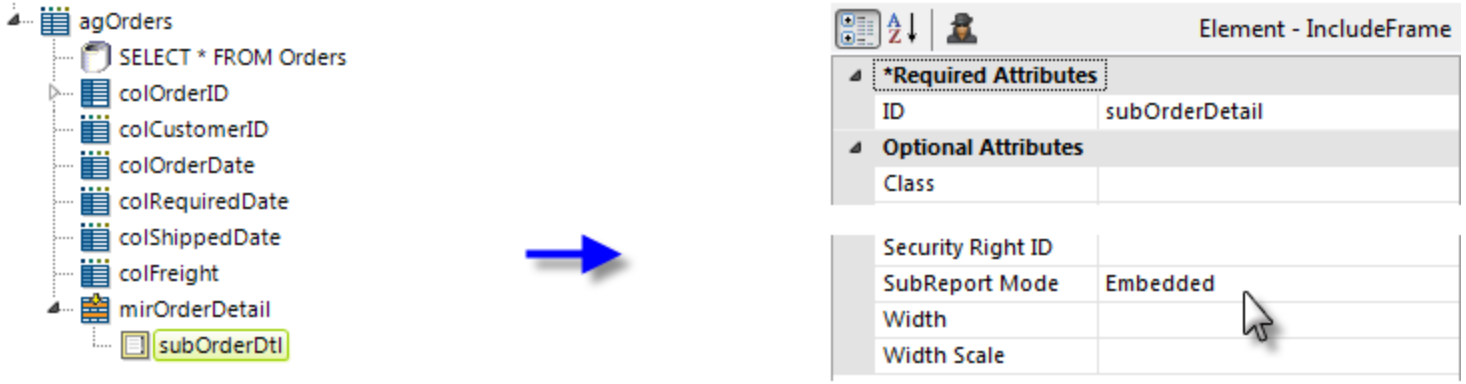

Next we need to add the trigger that will display and hide the More Info Row, by placing an **Action.Show Element** element beneath the Label element in the AG's first column, and configuring its attributes as shown.

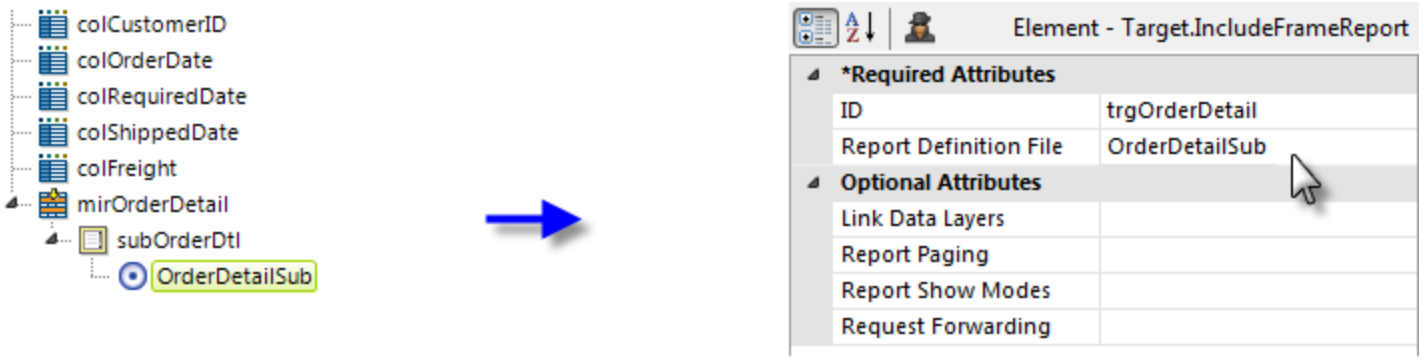
- Its **Element ID** attribute value matches the element ID given to the More Info Row element. With this in place, clicking the data value in the first AG column will cause the More Info Row to appear and disappear.
- You can't use Action.Show Element as the child of an Analysis Grid Column, so you need to use a regular **Data Table Column** to display the data that you want to use as a link. This column will therefore be unable to be selected in the AG's Formula, Filter, and other configuration panels.

Now let's create the subreport to be shown in the More Info Row:




In a new definition, we'll create the subreport content. As you can see in the example above, it does not need Report Header or Report Footer elements, and may not need a Style element. The example uses a **Data Table** and has its own datalayer and some kind of filter element that restricts the data to a unique identifier that will be passed to the report as a Request variable.
If we were using DataLayer.SQL, the query's SELECT statement could use the Request variable in its WHERE clause, and a filter element would not be necessary. The subreport can be tested independently of the rest of the application, by passing it test Request variables, to ensure that it's retrieving the correct data and displaying it properly. Now we're ready to put the subreport into the main report:

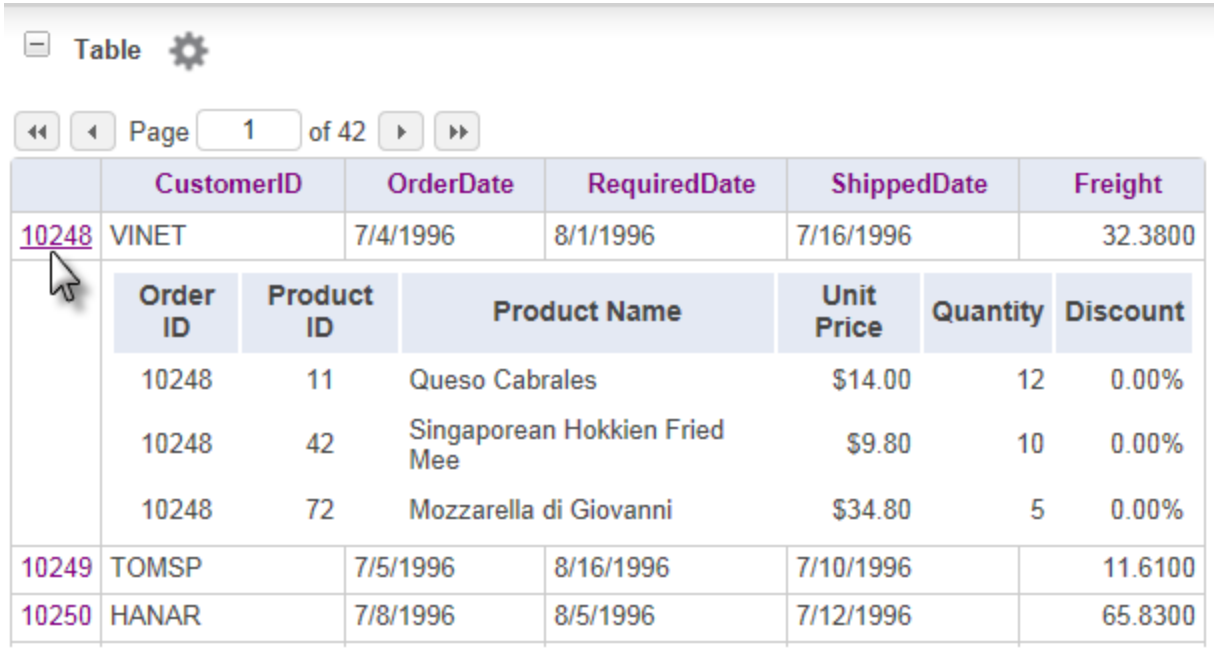

agOrders
SELECT * FROM Orders
colOrderID
colCustomerID
colOrderDate
colRequiredDate
colShippedDate
colFreight
mirOrderDetail
subOrderDtl
Element - IncludeFrame
*Required Attributes
ID
subOrderDetail
Optional Attributes
Class
Security Right ID
SubReport Mode
Embedded
Width
Width Scale


colCustomerID
colOrderDate
colRequiredDate
colShippedDate
colFreight
mirOrderDetail
subOrderDtl
OrderDetailSub
Element - Target.IncludeFrameReport
*Required Attributes
ID
trgOrderDetail
Report Definition File
OrderDetailSub
Optional Attributes
Link Data Layers
Report Paging
Report Show Modes
Request Forwarding


colShippedDate
colFreight
mirOrderDetail
subOrderDtl
trgSubRpt
Link Parameters
Parameters
OrderID
@Data.OrderID~

As shown above, a **SubReport** element is added beneath the More Info Row element. Beneath it, **Target.Report** and **Link Parameters** elements are added. Their attributes are configured as shown.

The SubReport and Target.Report elements have slightly different internal names, which appear at the top of the Attributes panel. The target report is, of course, the subreport definition that we created in previous step, and the parameter passed is the unique identifying value that the subreport is expecting.




And, the resulting Analysis Grid should look like the one shown above, when an Order ID column value is clicked.

# Linking Datalayers

When the Logi Server Engine renders a report with a More Info Row, it runs the subreport datalayer for *each row* of the AG before it displays the AG, which could produce an undesirable delay. If this is problematic, you may be able to avoid it: the Target.Report element has a **Link Data Layers** attribute which, when set, will cause the AG's datalayer's cached data to be used for the sub-report. The subreport then uses a **DataLayer.Linked** element in its definition to access the data.

 When a SubReport in *Embedded* mode is used beneath a More Info Row, any **Sort** elements used beneath individual Data Table columns and **Interactive Paging** elements in the subreport definition will be *ignored* and will not be rendered. There are a lot of ways subreports can be used, and the Action.Show Element element can be used to make them appear/disappear in a variety of places in a report, not just within a More Info Row.

# Paginate Data and Print Reports

The volume of data retrieved into Logi reports often requires that tables be presented as a series of pages; in addition, developers often need to offer the ability to print hard copy versions of reports with suitable headers.

The following topics introduce developers to techniques for satisfying both of these requirements:

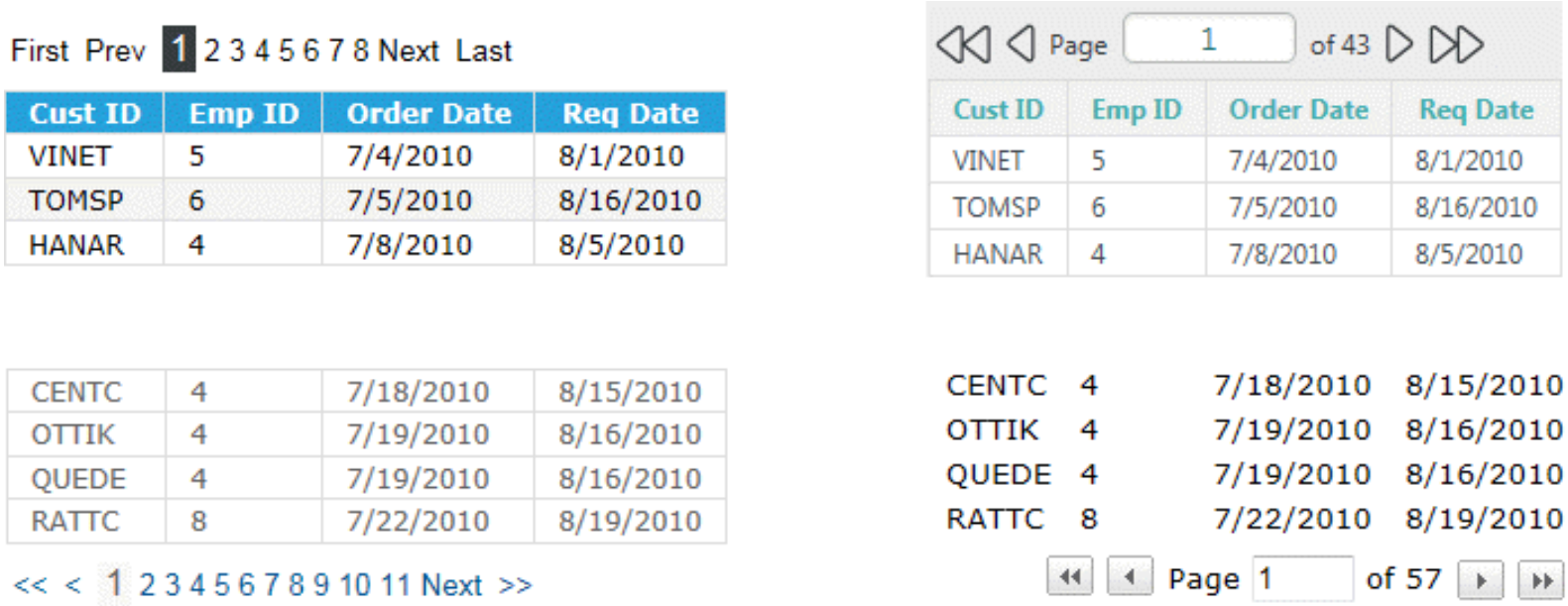
- Interactive Paging Element Attributes
- Pagination Usage Examples
- Page Navigation using URL
- Creating Printable Reports

## About Pagination

Large Data Tables, with hundreds of rows of data, are cumbersome to interact with in a browser-based environment. The concept of navigating through a lot of data using "pages", each with a small number of rows, is familiar to anyone who's used an Internet search engine and looked through the search results. Logi reporting tools include the **Interactive Paging** element which provides this functionality for Data Tables in Logi applications.

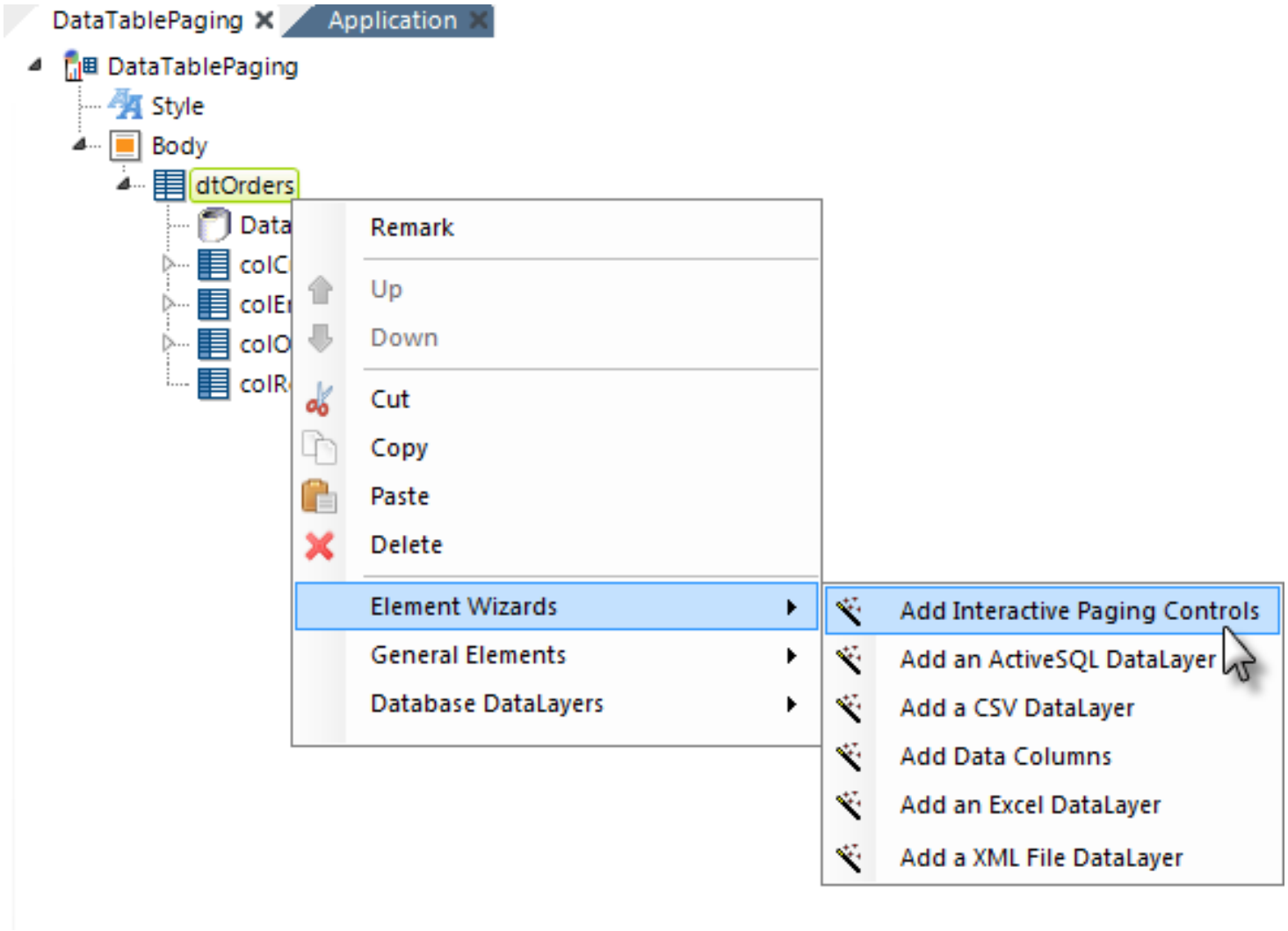
 We *do not* recommend using Interactive Paging with tables that contain user input elements in each row.

The examples below show four different styles and locations for page navigation links, all of which are created using the Interactive Paging element. The left two examples use Text-type links and the right two use Image-type links:

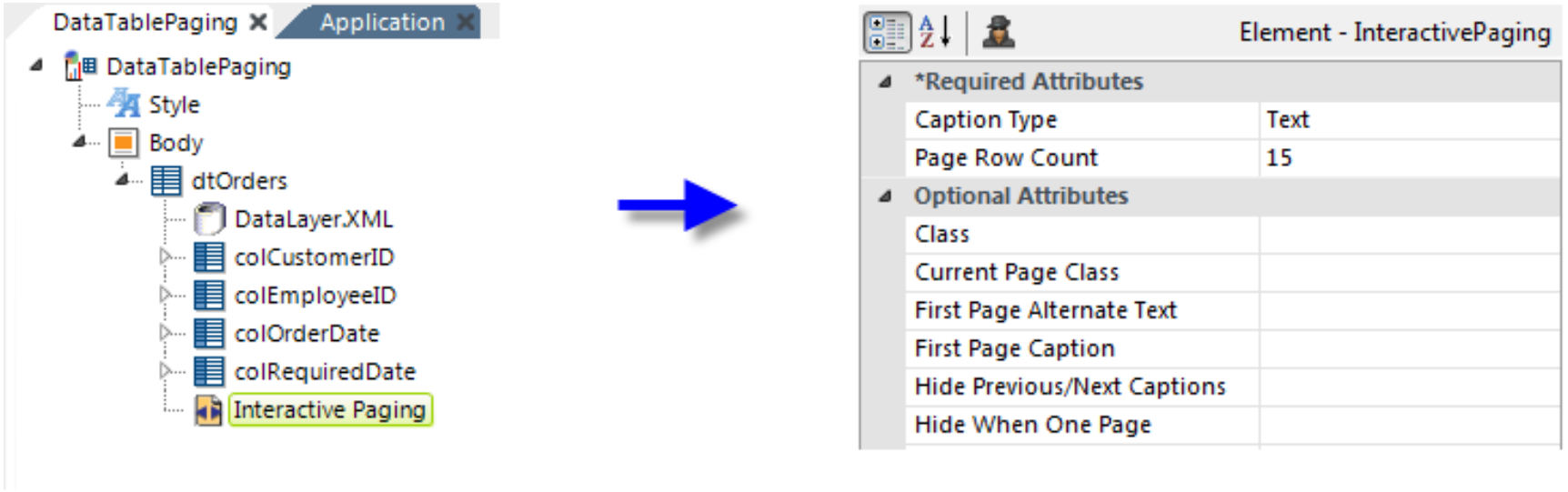



Studio includes a wizard that will add an Interactive Paging element to a report definition and configure it, using the Image-type links shown in the upper-right hand corner of the previous examples.

To use it, select the desired **Data Table element**, right-click it and select **Element Wizards.** Then, select **Add Interactive Paging Controls**, as shown below:




The Interactive Paging element can, of course, be added and configured manually as well.




The resulting element is inserted beneath the selected Data Table, as shown above, with its attributes configured by the wizard. For more information, see "Paginate Data and Print Reports" on page 205.

# Interactive Paging Element Attributes

Once an **Interactive Paging** element has been added, it can be customized. The element's attributes are:

| Identifier | Description |
|---|---|
| Caption Type | (Required) Specifies whether the navigation links will be Text, Images, or Scrollbar . |
| Page Row Count | (Required) Specifies the *number* of data **rows** that will appear per "page". |
| Class | Specifies the general style class to be applied to the element. |
| Current Page Class | Specifies the style class to be applied to the text that indicates the current page, overriding the previous Class. Use this, for example, to make the current page number appear in a larger font, with a colored background, etc. |
| First Page Alternate Text | Specifies the Alternate Text to be displayed as the "First Page" link when the browser options have images turned off. The text is also used by browsers that convert text to speech or Braille output. |
| First Page Caption | Specifies the text or the image file name that will appear as the "First Page" link. |
| Hide When One Page | Specifies if the navigation controls will be hidden when the datalayer returns only one page of data. The default value is *False*. |
| Hide Previous/Next Captions | Specifies if the non-numeric navigation links will be dynamic based on the current page. When set to *True*, the links for First, Previous, Next, and Last are hidden when the current page is the first or last page. The default value is *False.* |

| Identifier | Description |
|---|---|
| ID | Specifies a unique element ID. |
| Last Page Alternate Text | Specifies the Alternate Text to be displayed as the "Last Page" link when the browser options have images turned off. The text is also used by browsers that convert text to speech or Braille output. |
| Last Page Caption | Specifies the text or the image file name that will appear as the "Last Page" link. |
| Lazy Loading | Specifies whether the scrollbar utilizes lazy loading. The default value is *False*, all data loads at one time. When set to *True*, loading begins when the scrollbar reaches the bottom of the page. For this to apply, the **Caption Type** attribute must be set to *Scrollbar*.<br><br>Depending on your data results, lazy loading may expand your table width while scrolling. Once you reach the bottom of your results the table width is final. |
| Lazy Loading Multiple | Specifies the amount of data that fetches on the next load. The default value is empty or *1*, the next load fetches page count row data. If set to 3, the next load fetches 3x the page count row data. For this to apply, the **Caption Type** attribute must be set to *Scrollbar* and **Lazy Loading** must be set to *True*.<br><br>Depending on your data results, lazy loading may expand your table width while scrolling. Once you reach the bottom of your results the table width is final. |
| Location | Specifies where the navigation controls will appear: *Top*, *Bottom*, or *Both*. The default value is *Both*. To align the controls with the left or right side of the table, apply an appropriate style class to the Data Table's parent container (such as a Division or Column Cell element). |

| Identifier | Description |
|---|---|
| Next Page Alternate Text | Specifies the Alternate Text to be displayed as the "Next Page" link when the browser options have images turned off. The text is also used by browsers that convert text to speech or Braille output. |
| Next Page Caption | Specifies the text or the image file name that will appear as the "Next Page" link. |
| Numbered Page Count | Specifies how many separate numbers will be display, when using a Text-type caption, in a list of numbered page links. The default value is *10*. For this to apply, the **Show Page Number** attribute must be set to *Numbered*. |
| Page Number Caption | Specifies a *word* to use instead of "Page" in the navigation control caption "Page X of Y". The default is *Page*. |
| Page of Caption | Specifies the *word* to use instead of the word "of" in the navigation control caption "Page X of Y". The default is *of*. |
| Previous Page Alternate Text | Specifies the Alternate Text to be displayed as the "Previous Page" link when the browser options have images turned off. The text is also used by browsers that convert text to speech or Braille output. |
| Previous Page Caption | Specifies the text or the image file name that will appear as the "Previous Page" link. |
| Previous/Next Class | Specifies the style class to be applied to theFirst, Last, Next, and Previous parts of the paging control. |

| Identifier | Description |
| --- | --- |
| Show Page Number | Specify *True* or *False* here to show or hide the "Page X of Y" part of the navigation controls. Specify *Numbered* to create a list of numbered page links (works with the Numbered Page Caption attribute). |

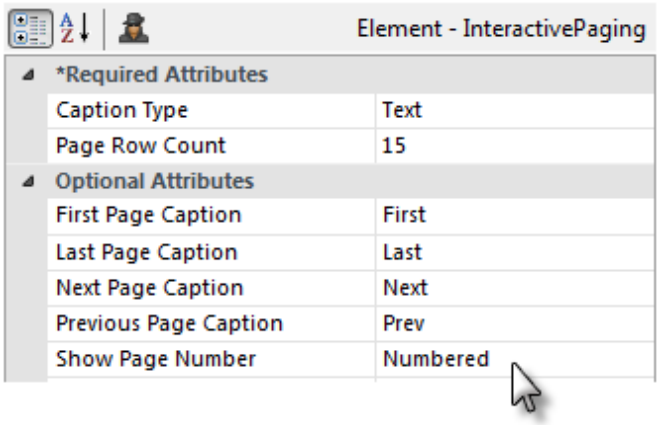
To prevent the entire page from being refreshed when a different table page is selected, set the parent Data Table element's **AJAX Paging and Sorting** attribute to *True*. AJAX is a technology that allows targeted portions of web pages to be updated, rather than the entire page. With this attribute enabled, when the user changes pages by clicking the table's navigation controls, only the table portion of the web page will be updated, preventing the page from "flickering". You should be aware that this alters the behavior of the browser's "Back" button because the page history is not updated when AJAX is used. If AJAX

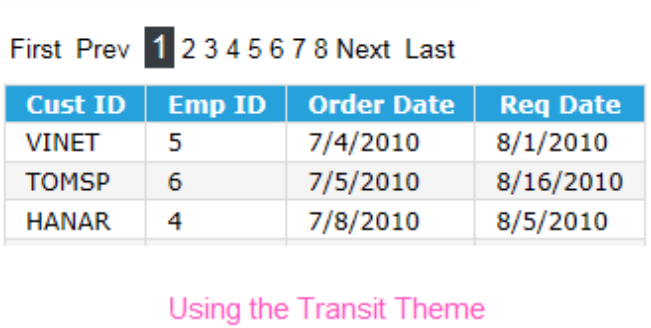
Paging and Sorting is enabled and a user changes table pages and there is more than a very brief delay, a spinning "Wait" icon will automatically be displayed to let the user know that processing is occurring.

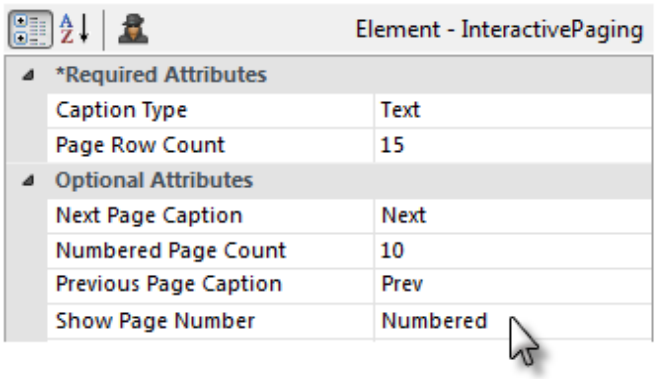
# Pagination Usage Examples

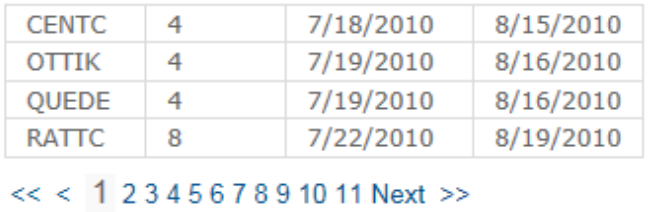
The following images show the attribute settings for the examples shown earlier. To conserve space, some blank attributes have been removed:

- **Text-type examples:**

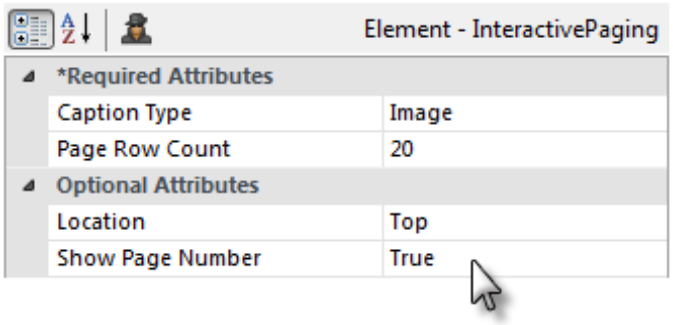


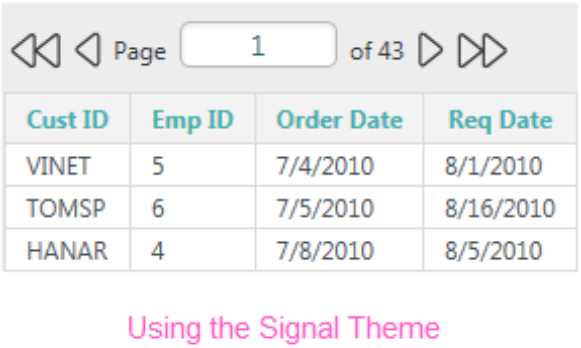



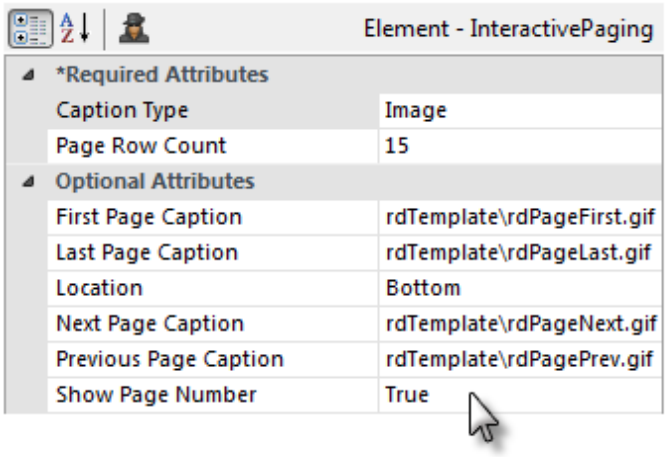



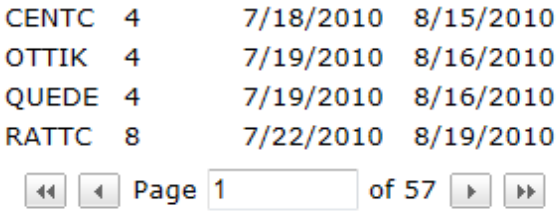
- **Image-type examples:**



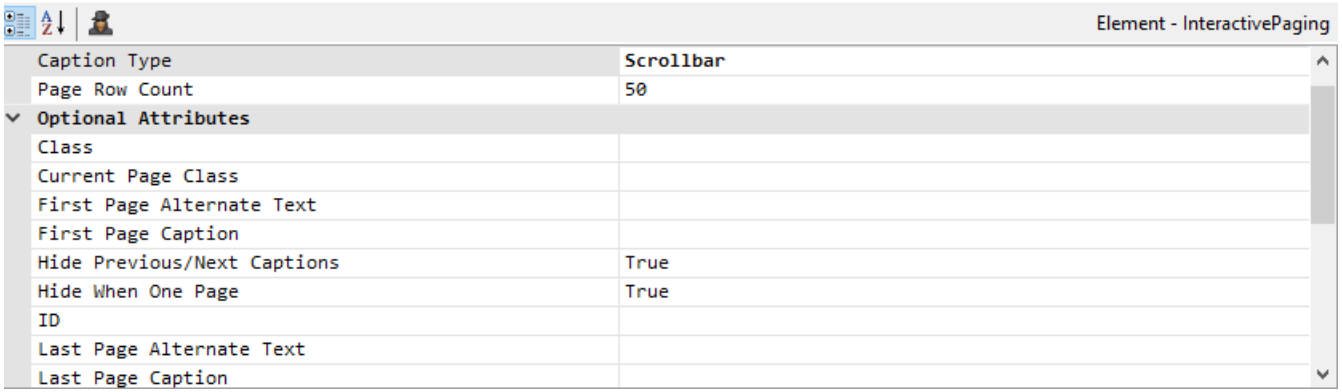

Using the Signal Theme




The special image files used in the last example, `rdPageFirst.gif`, `rdPageLast.gif`, etc. are distributed with Logi Info and can be found in your application's `rdTemplate` folder.

- **Scrollbar-type example:**


Element - InteractivePaging
Caption Type
Scrollbar
Page Row Count
50
Optional Attributes
Class
Current Page Class
First Page Alternate Text
First Page Caption
Hide Previous/Next Captions
True
Hide When One Page
True
ID
Last Page Alternate Text
Last Page Caption

| Order Id | Order Date | Ship Country |
|---|---|---|
| 10498 | 4/7/1997 12:00:00 AM | Venezuela |
| 10499 | 4/8/1997 12:00:00 AM | Venezuela |
| 10500 | 4/9/1997 12:00:00 AM | France |
| 10501 | 4/9/1997 12:00:00 AM | Germany |
| 10502 | 4/10/1997 12:00:00 AM | Mexico |
| 10503 | 4/11/1997 12:00:00 AM | Ireland |
| 10504 | 4/11/1997 12:00:00 AM | USA |
| 10505 | 4/14/1997 12:00:00 AM | Canada |
| 10506 | 4/15/1997 12:00:00 AM | Germany |
| 10507 | 4/15/1997 12:00:00 AM | Mexico |
| 10508 | 4/16/1997 12:00:00 AM | Germany |
| 10509 | 4/17/1997 12:00:00 AM | Germany |
| 10510 | 4/18/1997 12:00:00 AM | USA |
| 10511 | 4/18/1997 12:00:00 AM | France |
| 10512 | 4/21/1997 12:00:00 AM | Brazil |
| 10513 | 4/22/1997 12:00:00 AM | Germany |
| 10514 | 4/22/1997 12:00:00 AM | Austria |
| 10515 | 4/23/1997 12:00:00 AM | Germany |
| 10516 | 4/24/1997 12:00:00 AM | Ireland |
| 10517 | 4/24/1997 12:00:00 AM | UK |

# Page Navigation Using URL

If a Data Table has been paginated, there are scenarios in which you might want to navigate to a specific table page. For example, suppose your Data Table has drill-down links in it and you'd like to return to the same Data Table page after drilling-down to a detail report. Yes, the "Back" button on your browser will accomplish the same thing, but there may be circumstances in which you don't want to rely on it.

When you navigate away from the paginated Data Table, a special Request variable is included in the URL that calls the next report. You can use that variable in the link used to return to the Data Table report to ensure you return to the same Data Table page.

To do this, include these two Link Parameters in the return link:

| Parameter Name | Parameter Value |
|---|---|
| `<myTableID>-PageNr` | `@Request.<myTableID>-PageNr~` |
| `rdNewPageNr` | `True` |

where <*myTableID*> is the element ID of the paginated Data Table.

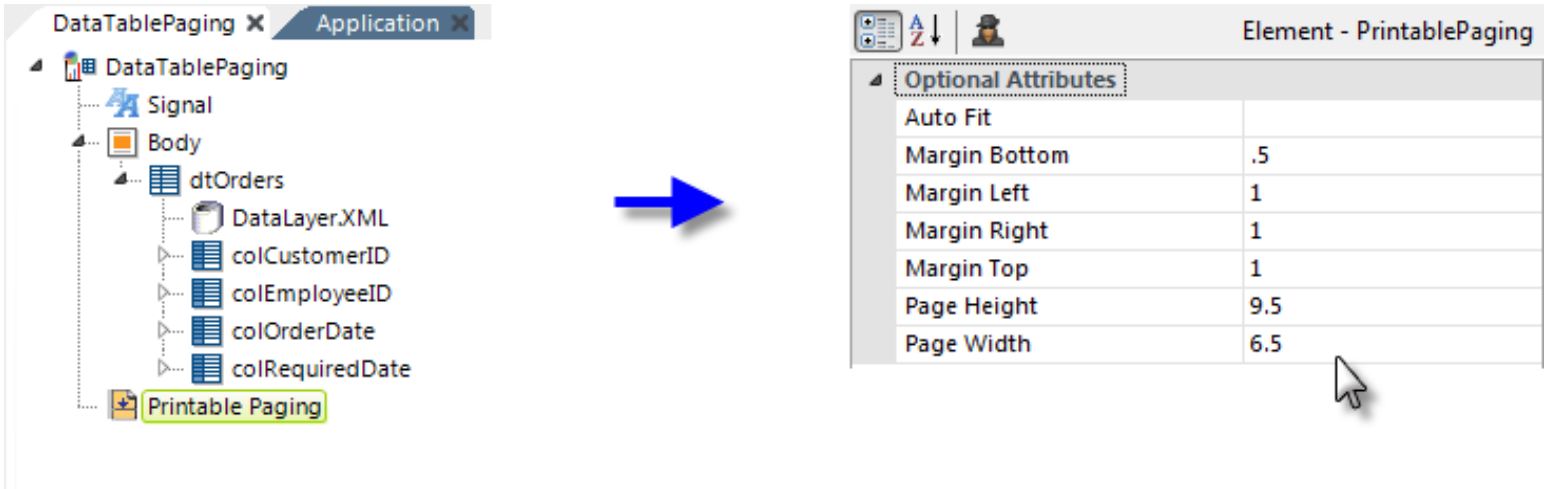
Logi Info developers may want to run a process task and then return to the same Data Table page. This can be done by passing `@Request.<myTableID>-PageNr~` to the task and then assigning its value to a Link Parameter named `rdNewPageNr` when calling the Data Table report.

# Creating Printable Reports

Report consumers often want to *print* hard copies of their web-based reports. To do this, the reports need to be paginated and formatted for printing. Logi reporting products make this easy to accomplish by providing the **Printable Paging** element.

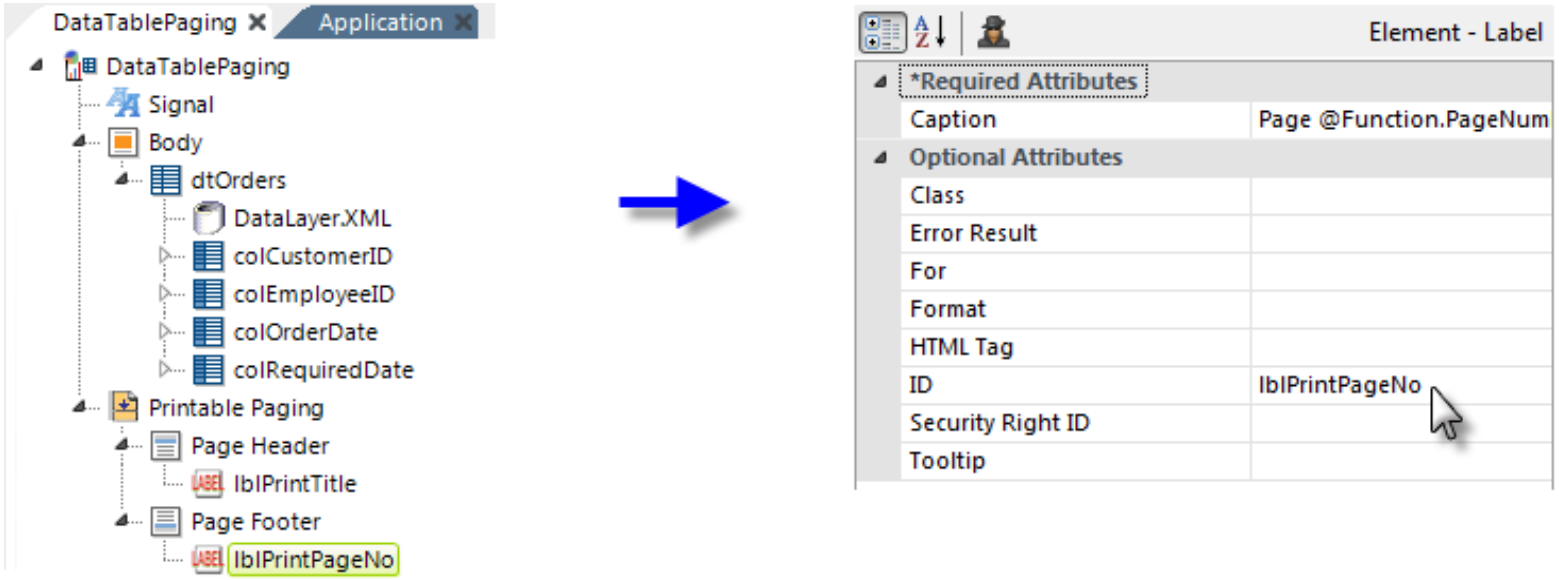
The process causes the report to be formatted for printing and then loaded into a browser window, after which it can be printed. The process automatically *hides* any **Interactive Paging** element, which ensures that the navigation controls do not appear on the printout and increases the number of records in tables so they fill the page.

There are two things that must be done to make a report "printable": the addition of a **Printable Paging** element, which governs page margins, size, and headers, and the addition of a link that's clicked to begin the formatting/printing process. The formatting/printing process formats the report for printing and then, optionally, outputs it to a printer or other formats, such as Adobe PDF or Microsoft Word.




Begin, as shown above, by adding a Printable Paging element beneath the root Report element (its location in the element tree does not matter). Set its **Margin** attributes as desired, in inches. To calculate the proper **Page Height** and **Page Width** attribute

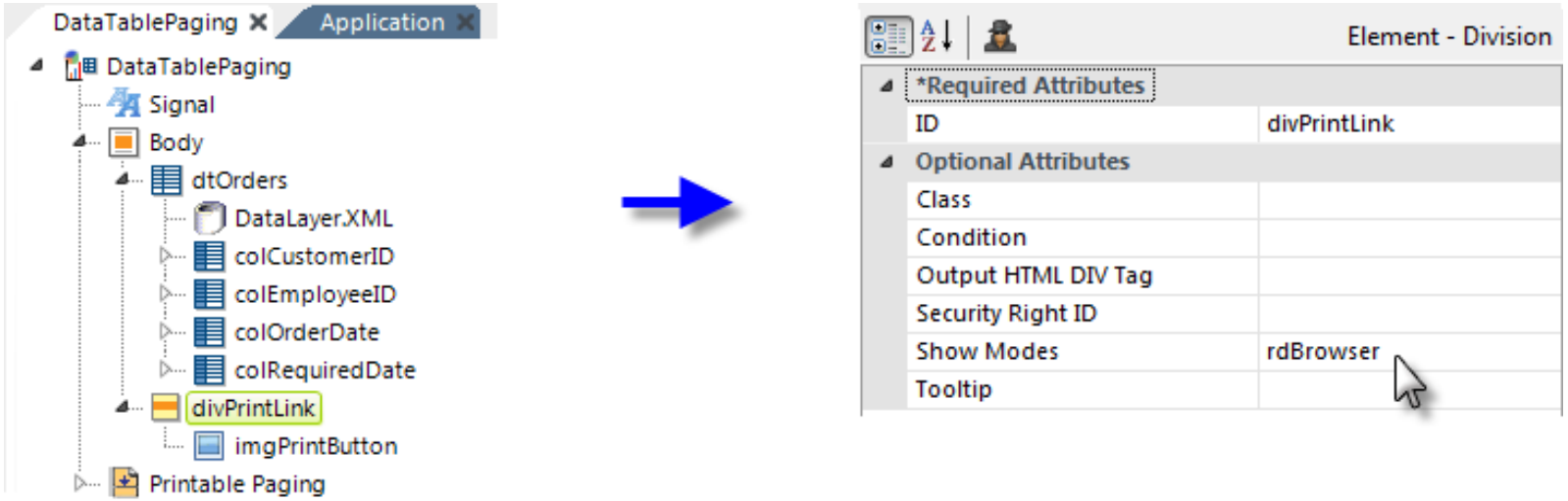
values, start with the physical paper size in inches and subtract the related margin values. So, in the example, working with a standard U.S. Letter size page, subtract from the physical paper height (11") the bottom margin (- .5") and the top margin (- 1") to arrive at the Page Height value ( =9.5").




If desired, a page header and footer that will only appear in the printed output can be included by adding **Page Header** and **Page Footer** elements as child elements beneath the Printable Paging element, as shown above. They, in turn, can contain other elements, including **Labels** and **Images**. In the example, the footer makes use of some *Function tokens* to present the current page number and total page number. The complete **Caption** attribute value for this is:

```
Page @Function.PageNumber~ of @Function.PageCount~
```

The `@Function.PageNumber~` token is only valid when Interactive Paging or Printable Paging elements are in use and the `@Function.PageCount~` token is only valid when Printable Paging is in use.

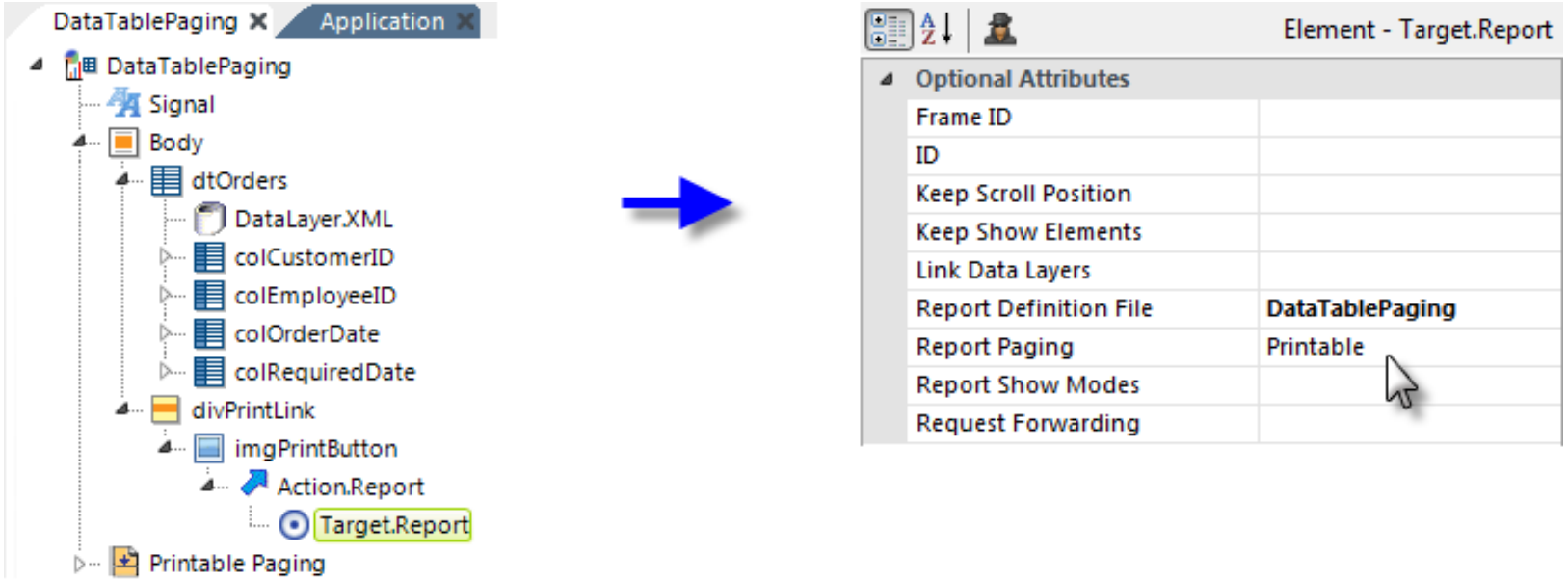



Next, add the element that users will click to print the report:

1. Add a **Division** element beneath the Body of your report and set its **Show Modes** attribute to the system value *rdBrowser*, as shown above.
2. Add an **Image**, **Label**, or **Button** element beneath the Division to use as the report's "print link".

The *rdBrowser* Show Modes value causes the Division element (and its child elements) to only be visible when the report is viewed in a browser; ensuring that it will *not* appear on the printed report. Next, select the appropriate actions depending on the type of output required.

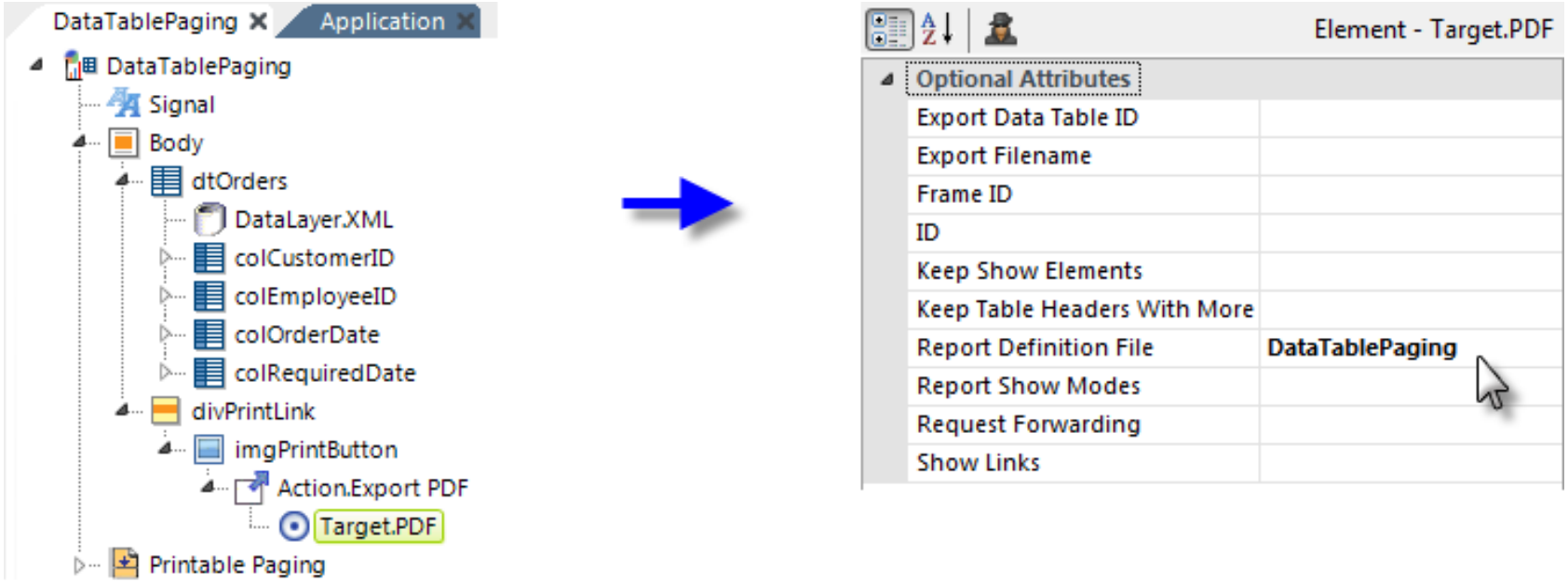
# Format Report for HTML Display and Printing




3. As shown above, add **Action.Report** and **Target.Report** elements beneath the print link element. Set the Target.Report element's attributes as shown.

When the print link is clicked, the report will be reformatted and displayed in the browser window. A dialog box will appear to allow printer selection and printing.

# Format Report and Display as a PDF




3. As shown above, add **Action.ExportPDF** and **Target.PDF** elements beneath the print link element. Set the Target.PDF element's attributes as shown.

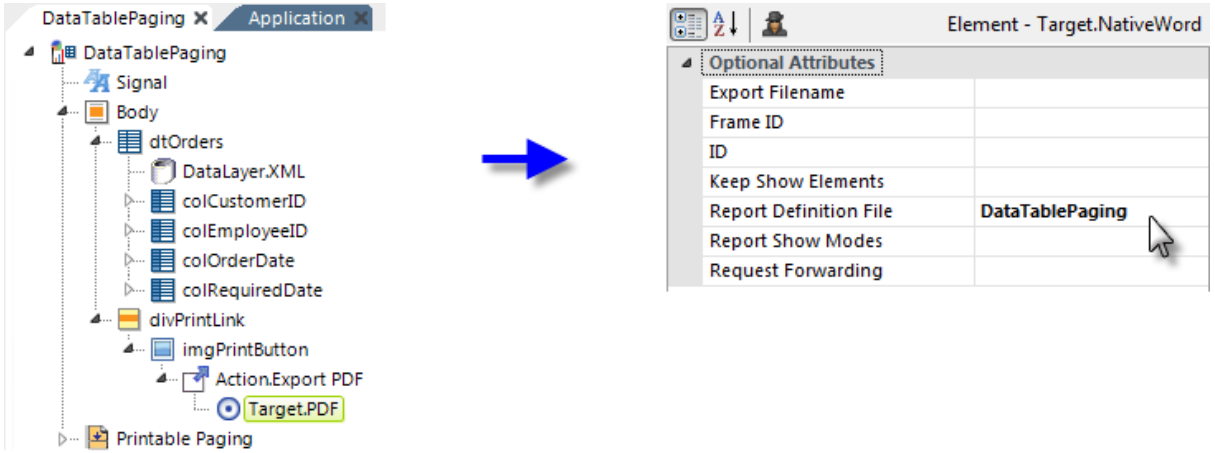
When the print link is clicked, the report will be reformatted, saved as a temporary .PDF file, and displayed using the Adobe Reader browser extension (which must be installed on the client machine). From there, the report can be saved as another PDF file or printed.

To specify the way PDFs exports are generated, set the Content Disposition attribute in the Target.PDF element to either **Attachment** or **Inline.** By default, the attribute is "empty", PDFs use browser settings. When the attribute is set to "Attachment", you

are prompted to download the PDF. When the attribute is set to "Inline", you are prompted to display the PDF inside the Web page.

Page headers and footers are rendered separately when exporting to PDF format. Since space is limited on printable pages, it's advisable to keep the **size** of page headers and footers small to allow room for the body of the report.

## Format Report and Display as a Word Document




3. As shown above, add **Action.Export Native Word** and **Target.Native Word** elements beneath the print link element. Set the Target.Native Word element's attributes as shown above.

When the print link is clicked, the report will be reformatted, saved as a temporary Word file, and opened using Word. From there, the report can be saved to another Word file or printed. Microsoft Office or the Word browser viewer must, of course, be installed on the client machine.

# Form-based Reporting

Form-based reporting delivers highly-formatted reports generated using Logi Info, via a web browser.

The following topics discuss form-based reporting:



## About Form-based Reporting

*Form-based reporting* is a special technique in which developers create *forms*, or *templates*, with specific regions designated in them for data. At runtime, the Logi Server Engine retrieves data and *fills-in* the specified data regions, leaving all the other report layout and formatting information unaffected. Examples of popular implementations of this technique include the generation of employment applications, tax forms, health care forms, and invoices. The final content generated must accurately render the form *and* the data without distortion or other formatting changes.

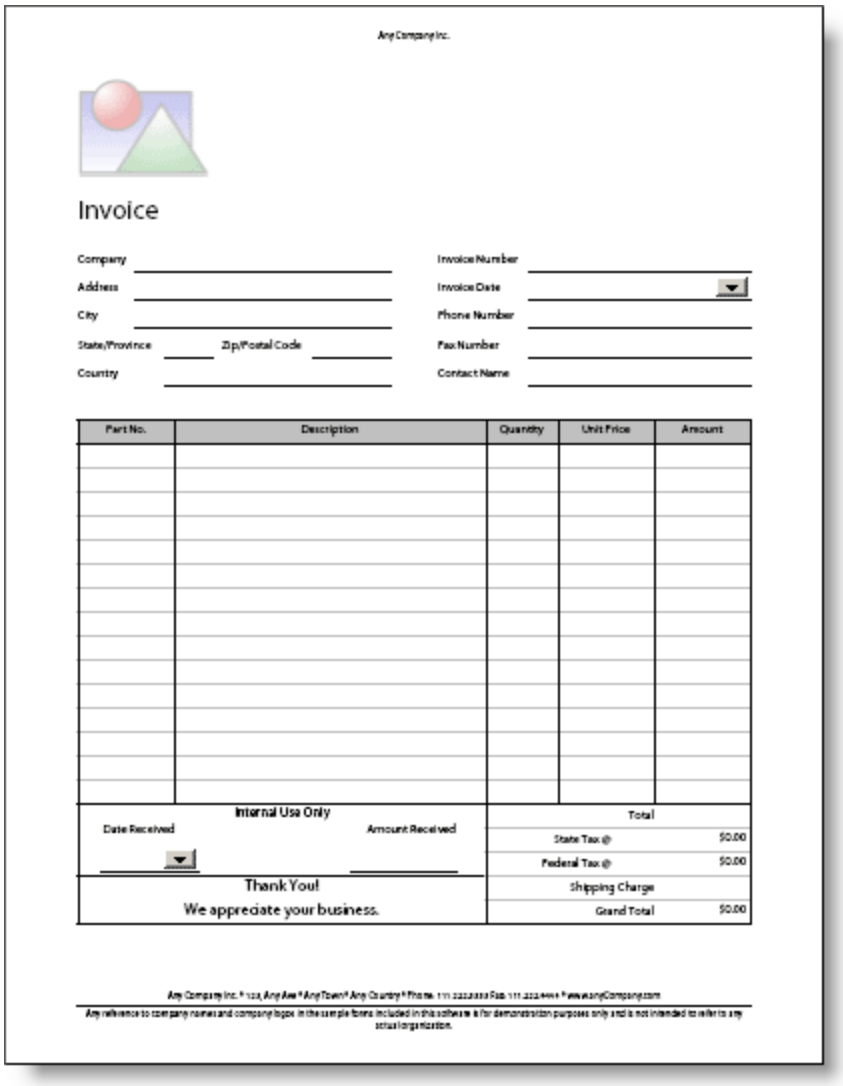
Form-based reporting differs from web-based reporting in the way developers use Logi Info. For web-based reports, developers use Logi Studio as the design front-end and the data and report layout are built using report definitions.

However, for form-based reporting, other vendors' tools are used to create the actual output document layout, or **template file**. Logi Studio is then used to develop the data retrieval and data population scheme. To facilitate this, Logi Info provides **template definitions** which are "blueprints" that indicate how data is mapped into the appropriate regions in a document.

Template *files* create with other vendor's tools are typically stored in the `_SupportFiles` folder but may be stored in any custom folder that you might want to create within your application root folder. Logi Template *definitions* are stored in the application's `_Definitions/_Templates` folder. In Logi Studio, these definitions appear in a separate Templates folder in the Application Panel.

# Adobe PDF Template

The Adobe Portable Document Format (PDF) is an industry standard for highly-formatted reports and documents. You can create PDF templates using any standard PDF editing tool. A PDF template includes editable form fields as well as static information such as graphics, text, bar-coding and boilerplate information.



Any Company Inc.

Invoice

Company _____ Invoice Number _____

Address _____ Invoice Date _____

City _____ Phone Number _____

State/Province _____ Zip/Postal Code _____ Fax Number _____

Country _____ Contact Name _____

| Part No. | Description | Quantity | Unit Price | Amount |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Internal Use Only

Date Received _____ Amount Received _____

| | |
|---|---|
| Total | |
| State Tax @ | $0.00 |
| Federal Tax @ | $0.00 |
| Shipping Charge | |
| Grand Total | $0.00 |

Thank You!

We appreciate your business.

Any Company Inc. * 123 Any Ave * AnyTown * Any Country * Phone: 111.222.3333 Fax: 111.222.4444 * www.anyCompany.com

Any reference to company names and company logos in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization.

The image above shows an empty PDF form template for an invoice. The locations of the *form fields* within the template determine the placement and appearance of the data in the final document. You can define and format text and numeric information on the form, as well as present data in a fixed tabular form.

# Microsoft Excel Templates

Business analysts work with Excel worksheets every day to build highly-formatted reports. You use Microsoft Excel as the design front-end for building Excel template files. The Logi server engine fills-in the empty Excel template with data from one or more data sources.

## Inputs for Synthetic Rating Estimation

| | | |
|---|---|---|
| Enter the type of firm = | 1 | *(Enter 1 if large manufacturing firm, 2 if* |
| Enter current Earnings before interest and taxes (EBIT) = | 10000 | *(Add back only long term interest expen* |
| Enter current interest expenses = | 2000 | *(Use only long term interest expense for* |
| Enter current long term government bond rate = | 6.00% | |

**Output**

| | |
|---|---|
| Interest coverage ratio = | **5.00** |
| Estimated Bond Rating = | **A** |
| Estimated Default Spread = | **1.00%** |
| Estimated Cost of Debt = | **7.00%** |

**For large manufacturing firms**

*If interest coverage ratio is*

| > | ≤ to | Rating is | Spread is |
|---|---|---|---|
| -100000 | 0.199999 | D | 10.00% |
| 0.2 | 0.649999 | C | 7.50% |
| 0.65 | 0.799999 | CC | 6.00% |
| 0.8 | 1.249999 | CCC | 5.00% |
| 1.25 | 1.499999 | B- | 4.25% |
| 1.5 | 1.749999 | B | 3.25% |
| 1.75 | 1.999999 | B+ | 2.50% |
| 2 | 2.499999 | BB | 2.00% |
| 2.5 | 2.999999 | BBB | 1.50% |
| 3 | 4.249999 | A- | 1.25% |
| 4.25 | 5.499999 | A | 1.00% |
| 5.5 | 6.499999 | A+ | 0.80% |
| 6.5 | 8.499999 | AA | 0.50% |
| 8.50 | 100000 | AAA | 0.20% |

**For financial service firms**

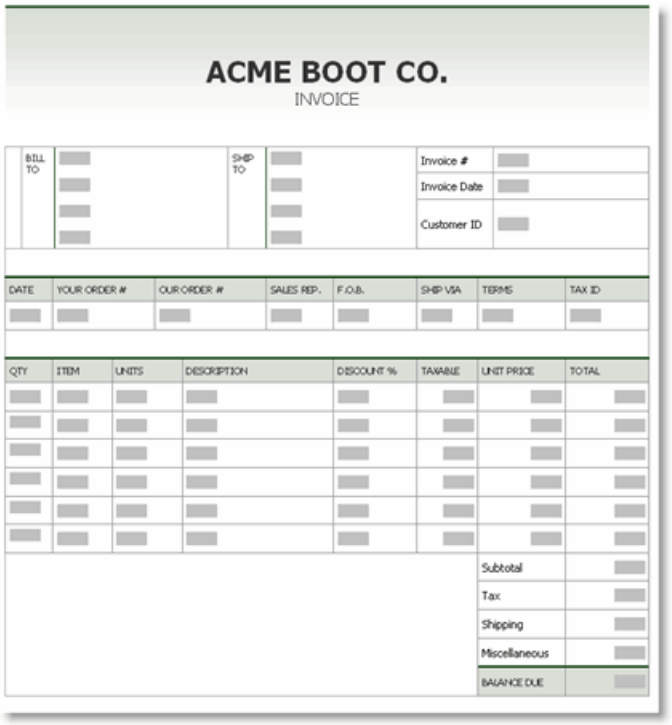
*If long term interest coverage ratio is*

| > | ≤ to | Rating is | Spread is |
|---|---|---|---|
| -100000 | 0.049999 | D | 10.00% |
| 0.05 | 0.099999 | C | 7.50% |
| 0.1 | 0.199999 | CC | 6.00% |
| 0.2 | 0.299999 | CCC | 5.00% |
| 0.3 | 0.399999 | B- | 4.25% |
| 0.4 | 0.499999 | B | 3.25% |
| 0.5 | 0.599999 | B+ | 2.50% |
| 0.6 | 0.799999 | BB | 2.00% |
| 0.8 | 0.999999 | BBB | 1.50% |
| 1 | 1.49999 | A- | 1.25% |
| 1.5 | 1.99999 | A | 1.00% |
| 2 | 2.49999 | A+ | 0.80% |
| 2.5 | 2.99999 | AA | 0.50% |
| 3 | 100000 | AAA | 0.20% |

The image above shows an Excel template for synthetic rating estimation, based on firm type, earnings and expenses. Charts, formulas, and pivot tables are dynamically updated based on the data contained in the worksheet. Templates are not restricted to *one* worksheet; a single report template can contain one or more worksheets within a workbook. You can also specify whether the final document contains one filled template or multiple filled templates by toggling an attribute within the Logi Template definition.

# Microsoft Word Templates

Microsoft Word can also be used to design document templates and, as with Excel templates, the Logi server engine fills-in the empty template with data from one or more data sources.




The image above shows part of a Word report template for adverse health care event information. The fields on the form template are filled-in by the Logi server engine and then the Word document can be processed like other templates: returned to the user's browser, saved, emailed, etc.

# The Template Definition

Logi Studio provides a special category of definition file, the Template definition, for use with form-based reporting. A Logi Template definition references a PDF, Excel, or Word template file (the target), specifies one or more datasource queries, and specifies how the target template is to be filled-in.

At runtime, the Logi Server Engine fills the data regions specified in the target template with data values and generates a new output document of the specified type. Developers control the delivery mechanism for the new document (e.g. return to the browser, save on the server, embed in an email, etc.).

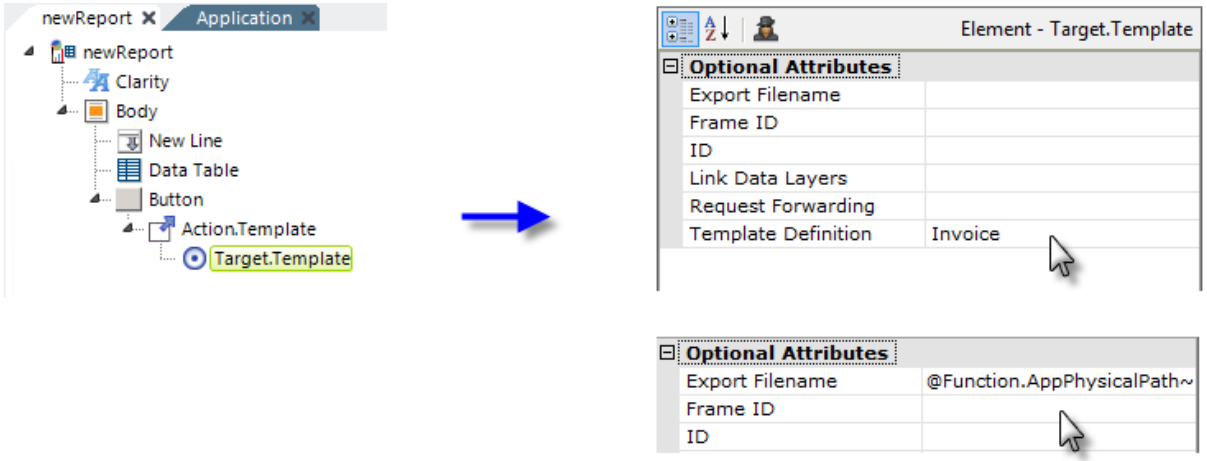
In a Logi application, template definitions are stored in the `_Definitions/_Templates` folder. In Studio, they appear beneath a Templates folder in the Application panel. Each template definition includes elements that are specific to the document type being generated and they're used to map the data into the native document template. The details of creating Template definitions are discussed in separate DevNet topics specific to each document type.

## Using Template Definitions

Two approaches are available for using Template definitions to product a document:

- In a Logi Info *Report* definition, the document is generated and returned to the user's web browser.
- In a Logi Info *Process* definition, a task saves the generated document as a file on the web server.

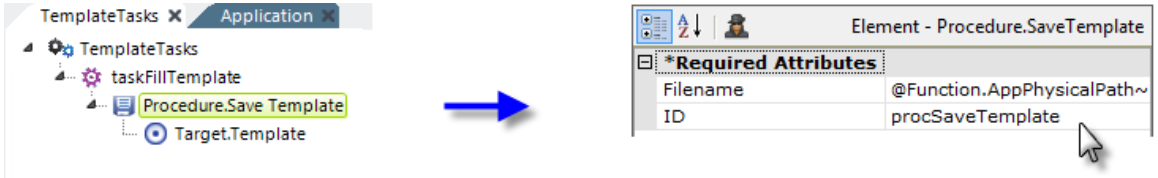
In the first approach, the **Action.Template** and **Target.Template** elements are used to reference template definitions:




The example above shows these elements in a Report Definition; they cause the "Invoice" template definition to be used to fill-in and generate the desired document. By default, the document is given a randomly-generated, GUID-based filename and is temporarily stored in the application's `rdDownload`folder, and then returned to the user's web browser to be viewed immediately, or saved locally. If a fully-qualified path and filename is specified for the **Export Filename** attribute, the document will be stored as that file on the web server, and then returned to the user's web browser to be viewed immediately, or saved locally. Examples of valid file names are:

```
@Function.AppPhysicalPath~\Exports\Invoices.xlsx
 C:\inetpub\wwwroot\MyLogiApp\MyTemplateDocs\FormLetters.xlsx
```

Naturally, the account used to run your Logi application must have full File Access permissions to any folder that will contain saved documents.




In the second approach, the **Procedure.SaveTemplate** and **Target.Template** elements are used in a Process task:

The example above shows a Process Definition task; it also causes the "Invoice" template definition to be used to fill-in and generate a document. However, in this case, the document is saved as a file, using the fully-qualified path and filename specified, and is *not* automatically returned to the user's web browser for viewing. Using other elements, you can then work with the saved file, perhaps linking it to the browser or sending it out in an email. Examples of valid file names are the same as those shown above and File Access permissions must be in place as discussed previously.

Templates provide a powerful reporting mechanism and Logi reporting products provide a number of ways of generating and distributing them. For additional information about specific type of templates, see our other Template documents.

# Debugging Templates

Special debugging features make it easier for you to determine what happens when a template is filled. These debugging features are enabled if you have the report's debugging style set to *Debugger Links*, either from Studio's toolbar or by manually configuring in it the _Settings definition.

| | A | B | C | D |
|---|---|---|---|---|
| 1 | **Customer ID** | **Order Date** | **Freight** | |
| 2 | | | | |
| 3 | VINET | 07/04/96 | $32.38 | |
| 4 | TOMSP | 07/05/96 | $11.61 | |
| 5 | HANAR | 07/08/96 | $65.83 | |
| 6 | VICTE | 07/08/96 | $41.34 | |
| 7 | SUPRD | 07/09/96 | $51.30 | |
| 8 | HANAR | 07/10/96 | $58.17 | |
| 9 | CHOPS | 07/11/96 | $22.98 | |
| 10 | RICSU | 07/12/96 | $148.33 | |
| 11 | WELLI | 07/15/96 | $13.97 | |
| 12 | HILAA | 07/16/96 | $81.91 | |
| 13 | | | | |
| 14 | | | | |
| 15 | Debug this page | | | |
| 16 | | | | |
| 17 | | | | |

For **Excel** templates, a debugging link, as shown above, will be displayed in the template. Clicking this link will display a typical Debugger Trace page, with details of the template operation.

For **Word** and **PDF** templates, no such link will be displayed. However, a file with the debug information will be saved, and you can browse it directly. The filename will be:

```
yourLogiAppfolder/rdDownload/GUID + templateName-rdDebug.htm
```

# Pass Information

The process of passing information between Logi reports and processes and external applications is essential to building **robust** and **flexible** Logi applications.

The following topics discuss techniques for passing information:

- Using Request Parameters
- Accessing User Input Values
- Working with @Request Tokens
- Query String Limits
- Accessing Global Data with Tokens
- Using HTML5 Local Storage

# Using Request Parameters

Logi reports and processes are called using URLs, which may include a Query String. The most common method of passing information in Logi applications is with **request parameters**, using the HTTP "GET" method. This is a example of a URL for a Logi report:

```
http://www.myReportSite.com/myApp/rdPage.aspx?rdReport=OrdersReport&addRecord=True
```

and for a Logi process:

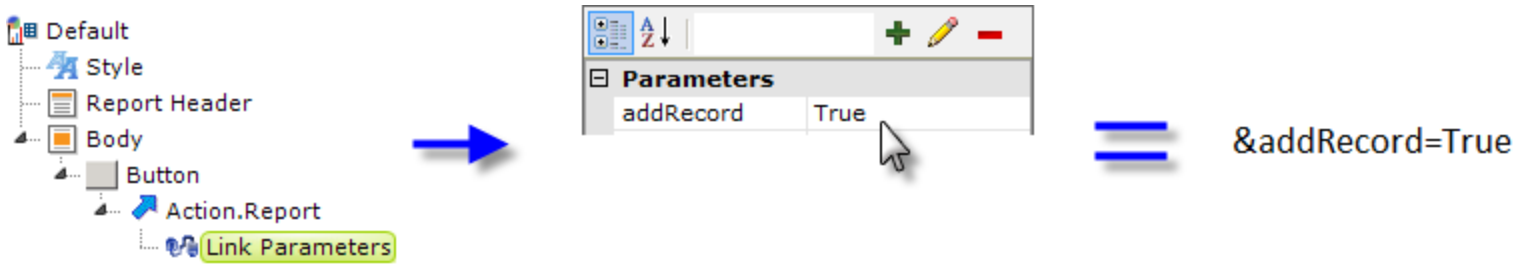
```
http://www.myReportSite.com/myApp/rdProcess.aspx?rdProcess=SiteTasks&rdTaskID=MemberLogin&UID=123
```

Both examples contain a **Query String** (the portion after the "?") which consists of multiple name and value pairs connected with an equals sign (e.g. UID=123) and separated by an ampersand. These name-value pairs are the request parameters.

Any application, whether web- or desktop-based, Logi application or not, that can redirect a user on the basis of a URL can make use of request parameters in the query string to pass information to a Logi report or process task definition.

## Using Link Parameters

Information can be passed from one report or process to another using request parameters. This can happen either automatically (see "Accessing User Input Values" on page 242) or explicitly through the use of the **Link Parameters** element:



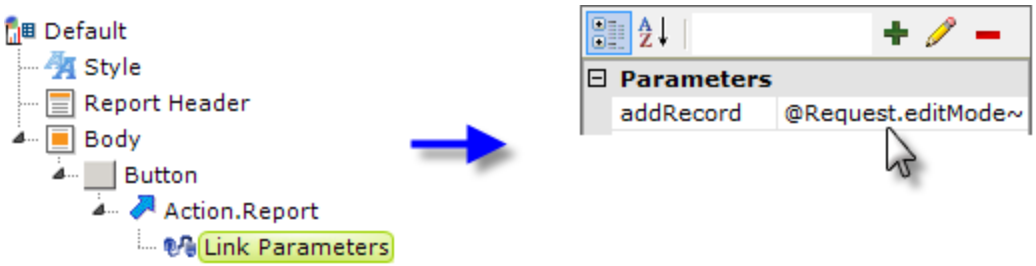

The example above shows a report definition with a **Button** element that calls another report when clicked. The Action.Report element has two child elements: **Target.Report**, which specifies the report being called, and a Link Parameters element.

Parameters and their attributes entered in the Link Parameters element wind up as request parameters in the **query string** used to call the next report or process task. Look at the earlier sample report URL and the example above; note how the Link Parameters attribute name and value becomes a request parameter key-value pair.

In the report called when the Button is clicked (the target report) the request parameter information can be accessed using @Request tokens. The format of the token is `@Request.<LinkParameterName>~`

In the case of the example above, the token `@Request.addRecord~` would have a value of *`True`*. Tokens are case-sensitive so careful spelling is required when using them.

*Token Reference* provides additional general information about tokens.




As shown in the example above, the value of request parameter **received in one report** (editMode) can be assigned to a Link Parameter with a different name (addRecord) and **passed to another report**.

Although Link Parameter values will resolve tokens, as shown above, they *will not* evaluate expressions. For example, if you create a parameter with the value of "=1+1" and submit the report, in the next report or process task its @Request token will have a value of the string *"=1+1"*, not a numeric value of *2*.

## Encoding Characters

Certain characters are invalid in query string data, such as "&" and "?", because they're used to delimit information within the URL itself. Whenever possible, the Logi Server Engine tries to handle these situations by **encoding** the characters before placing them in a URL. However, it's not always possible to interpret the developer's intentions using this mechanism, so you may need to manually encode the text or data you pass when using Link Parameters or request forwarding.

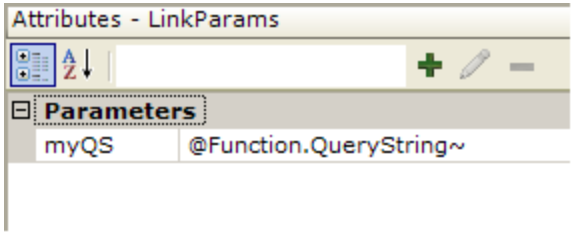
For example, you may want to pass a value in a Link Parameter that includes an ampersand, such as "Brooks&Dunn". This causes a problem because the ampersand is an invalid URL character. The solution is to pass the encoded value for the character, instead. The encoded value consists of the "%" symbol and the hex value of the character, in this case "%26", so that value in the URL looks like "Brooks%26Dunn".

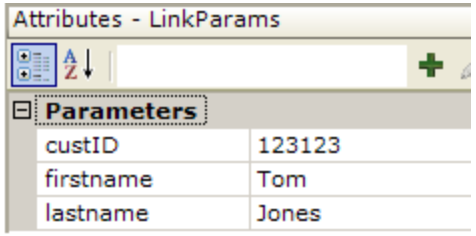
Characters that you may need to encode include:

| Character | Hex Value | Character | Hex Value |
|---|---|---|---|
| Dollar sign ("$") | 24 | Question mark ("?") | 3F |
| Ampersand ("&") | 26 | At sign ("@") | 40 |
| Plus ("+") | 2B | Space (" ") | 20 |
| Comma (",") | 2C | Quotation marks | 22 |
| Forward slash ("/") | 2F | Less than ("<") | 3C |
| Colon (":") | 3A | Greater than (">") | 3E |
| Semi-colon (";") | 3B | Pound sign ("#") | 23 |
| Equals ("=") | 3D | Percent sign ("%") | 25 |

# Avoid Passing Full Query String as a Parameter

Although the @Function.QueryString~ token makes it easy to do, we *do not recommend* that you pass an entire query string as a single Link Parameter, rather than passing its parts and then re-assembling them at the target definition.




NO!

Yes!

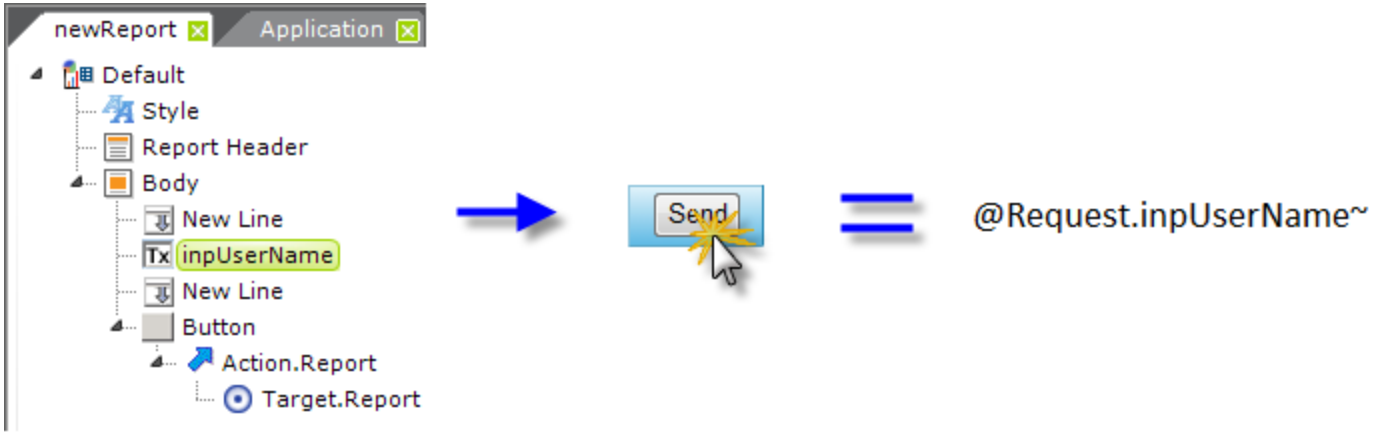
Passing the entire query string often results in double-encoding of the query string, raises cross-browser compatibility and URL length-limit issues, and can produce difficult-to-diagnose errors.

# Accessing User Input Values

Logi applications may include **User Input elements**, which *automatically* pass data to other definitions when the page is submitted.

You *do not* need to use **Link Parameters** to pass user input values. Doing so will cancel the automatic passing of them!

Accessing the data users enter is very easy:




As shown above, the text entered by a user is available in the target report by using the token `@Request.inpUserName~`. The **ID** attribute of a User Input element is appended to "@Request" to identify the data. Once again, using the correct spelling and case is essential.

Because an HTTP POST method is used when a Logi application page is submitted, data entered by users *does not* appear in the query string used to call the target report or process. Nonetheless, @Request tokens can still be used in the target report or process to access the input data.

A common use of User Input elements is to **dynamically** create **SQL queries** used in datalayers to retrieve data. For example, imagine a report page that includes an Input Select List named "inpState" that allows the user to select one of the states in the United States. When the page is submitted, the following query could be used to limit the SQL query on a subsequent page to the selected state:

```
SELECT * FROM SalesData WHERE Sales_State = '@Request.inpState~'
```

The entire request token is enclosed within the usual single-quotes required for a text value in a query. At runtime, the token will be replaced with the value selected in the Input Select List.

Be careful that you don't create a situation where a NULL value in the query causes an error, or be sure to use a **Default Request Parameter** (discussed below) to ensure that your query will have a default value when its page is first displayed.

### Receiving Data From External HTML Forms

Developers may need to call a Logi report *from* an external **HTMLpage** that's in another web application - not part of your Logi application. The following is an example of code used in that external page to create an HTML form that does that:

```
<body>
 <form name="form1" method="post" action="rdPage.aspx">
 <p>Enter Report: <input type="text" name="rdReport"></p>
 <p>Enter Category: <input type="text" name="Category"></p>
 <input type="submit" name="submitCategory" value="Submit">
 </form>
</body>
```

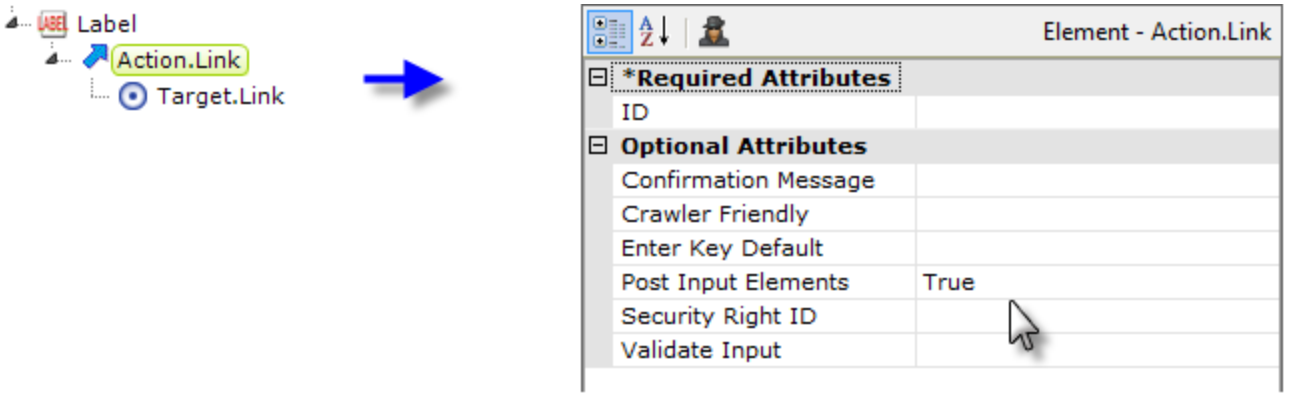
The *action* attribute for the <form> tag is set to **rdPage.aspx**, the default page for displaying report definitions within a Logi application.

The <input> tags are coded with *name* attributes which are used to identify the request parameters when the form is submitted. The rdPage.aspx page is expecting a request parameter named **rdReport** to identify the target report, so that's required, either as user input or hard-coded as a hidden input field.

In the target report, the data entered on the HTML page in the **Category** text input control will be accessible using the token `@Request.Category~`.

## Sending Data To External Web Application

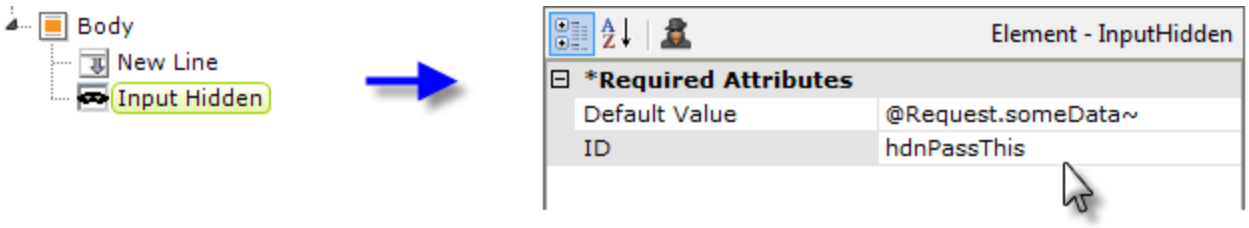
In the reverse case from the preceding section, you may need to have your Logi application link *to* a web page that is not part of your Logi application. This is easily done using the **Action.Link** and **Target.Link** elements:



Ordinarily, this kind of link *will not* pass the data entered into user input elements in your Logi report to the external web page. If you have user input data that you want to submit to the external page, set the Action.Link element's **Post Input Elements** attribute to *True*, as shown above. This will submit the data, using the HTTP POST method, to the external page (it will not appear in the browser in the URL).

## Passing Hidden Data

Developers may need to pass critical identifying data to another definition within their Logi application, but may not want to have it displayed in the browser in a URL. And for good reason: a user could simply edit the URL and refresh the page, perhaps accessing privileged information. The HTTP POST approach, discussed above, is one way to pass and hide such data.



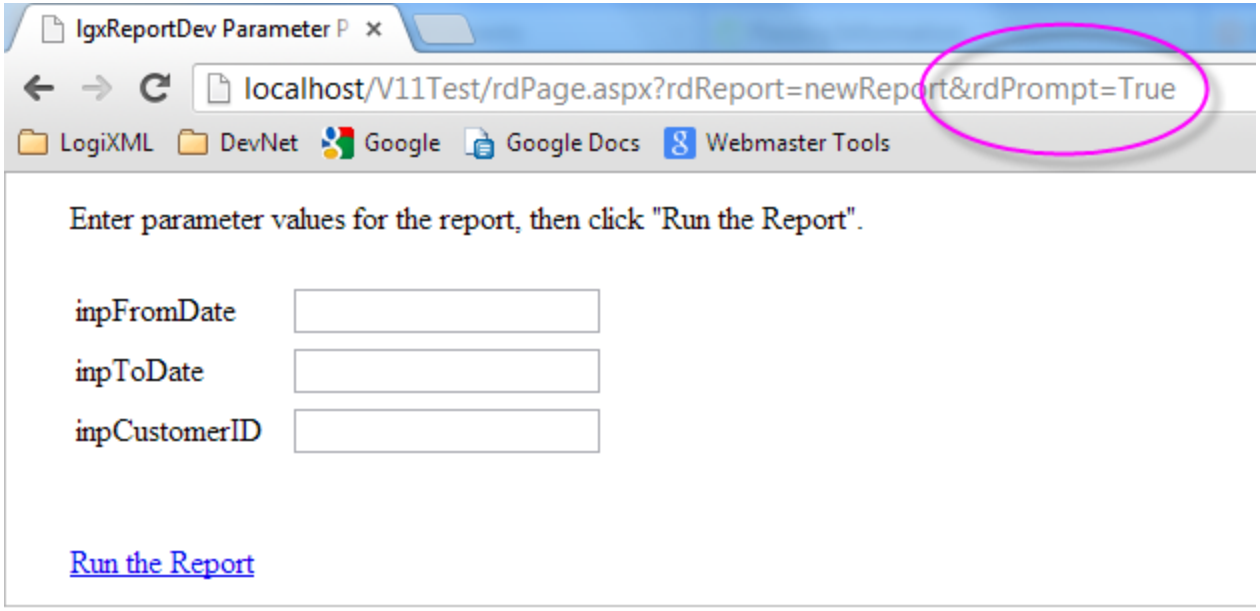

As shown in the example above, this can also be done by using an **Input Hidden** element. This element is not displayed on the web page but acts just like any other user input element: it passes its contents to the next Logi application definition using the HTTP POST method, so it doesn't appear in the browser in the displayed URL. You can use as many Input Hidden elements as you need in a report definition.

There are a variety of ways to put data into an Input Hidden element. Its **Default Value** attribute value can be a constant or you can use a token, as shown above. You can also use scripting to insert a value into it when the page is submitted or when some other event occurs. For examples of how this is done using JavaScript, see Passing Input Select List Captions.

In the target report for this example, the token `@Request.hdnPassThis~` would be used to get the value of the hidden input.

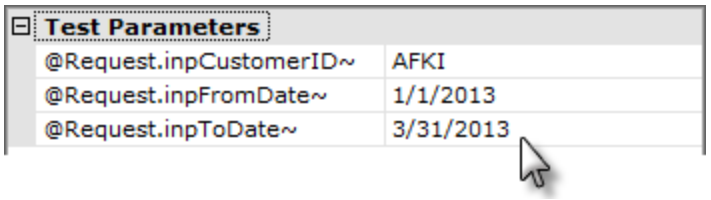
# Working With @Request Tokens

A Logi report or process that uses @Request tokens within its definition is "expecting" to receive these token from whatever calls it. Developers can **independently test** reports to see what Request variables are expected by adding the request parameter `&rdPrompt=True` to the end of the URL used to call it.

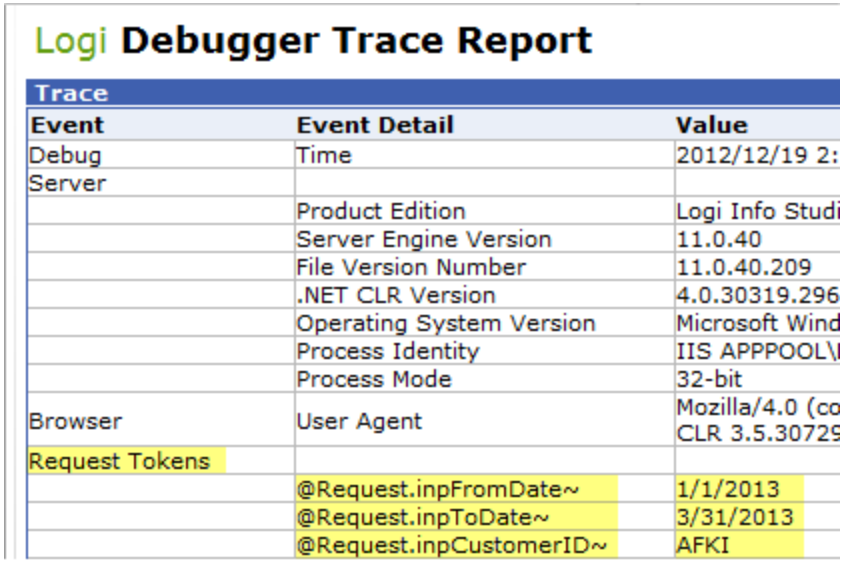



As shown above, when that request parameter is present in the URL, a special page with a form will be displayed with **text input fields** for every Request variable that the report or process is expecting. The developer can fill in the values and then run the report using them.

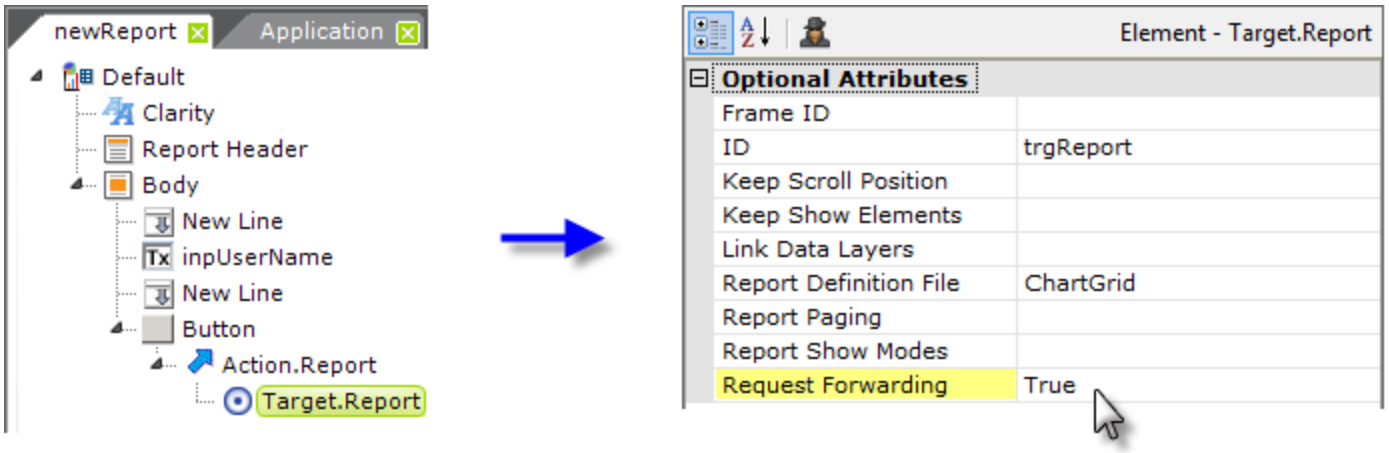
The above is also an example of how a request parameter can be used as a "switch" to toggle some feature on or off; this is often useful for **showing** or **hiding** elements in applications.




Similarly, if a report is expecting request parameters, the **Test Parameters** panel in Studio, shown above, will be visible beneath the Attributes panel and can be used to enter values for testing purposes.




As shown above, the diagnostic **Debugger Trace** page (accessible when debugging is turned on) shows the @Request tokens and their values.




**Target.Report** elements have a **Request Forwarding** attribute, as shown above. When set to **True**, all of the request parameters that were sent to the current report will be **forwarded** to the target report when the button is clicked.

This may sound like good thing, but it should be used with *caution*, because the application may pass **system request parameters** in addition to those that you specify and the **query string** may wind up getting very lengthy, especially if a sequence of reports keeps forwarding their parameters. This can be problematic because some browsers have a **length limit** on the URL they can handle (see "Query String Limits" on page 252).

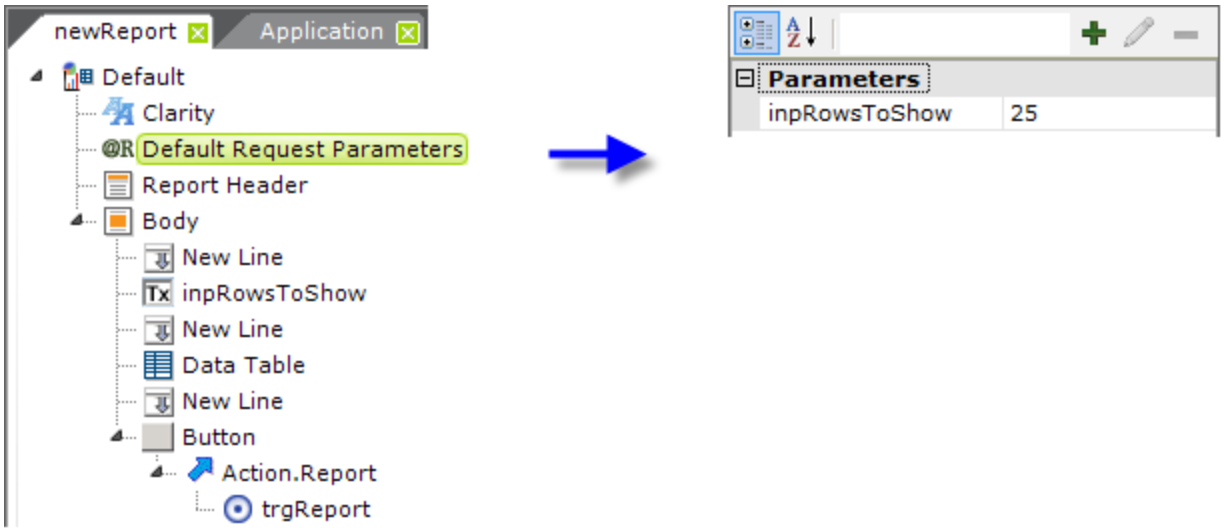
It's a good practice to **actively manage** your request parameters, rather than using Request Forwarding, to ensure that you know what's getting passed and when.

## Using Default Request Parameters

Imagine a report that lists salespeople based on their year-to-date sales volume and uses this SQL query:

```
SELECT TOP @Request.inpRowsToShow~ FROM SalesStaff ORDER BY YTDSales DESC
```

The report page includes an Input Text element, that has an ID of *inpRowsToShow*, so that at runtime the user can specify the number of rows to be shown in the report's Data Table. The report calls itself when a button is clicked, in order to refresh the data displayed.




In order to ensure that @Request.inpRowsToShow~ has a value in the SQL query, so that *some* data is retrieved the first time the report is displayed, a **Default Request Parameters** element can be included at the top of the report definition. Its attributes define an *inpRowsToShow* parameter with a value of 25, as shown above.

If **no request parameter** with the name *inpRowsToShow* is passed in the query string that calls this report or as a form field, then the value defined in the Default Request Parameters element will be used. If such a request parameter or form field does exist, then its value will be used and this element will be ignored. This ensures that, in any case, when the report is displayed at least the top 25 records in the query will be shown.

# Multi-Instance Dashboard Panels and Instance IDs

If Dashboard Panels are configured to be "multi-instance" panels, they're assigned individual "instance IDs" when they're added to a Dashboard. These IDs are used to ensure that they're uniquely identified - essential if a panel is used multiple times in the same Dashboard.

To avoid confusion, request variables generated by these Dashboard panels are passed with their instance ID appended to them, in the format *RequestVar_InstanceID*. Here's an example of what one of these request variables looks like in the Debugger Trace page:

| | | |
|---|---|---|
| Get DefaultRequestParams | | <DefaultRequestPa |
| | lstEmployeeName_3dc863a1868d4ef9a854ecfee18e8ee | Davolio,King |

In order to work with these request tokens, use the special token **@Function.InstanceID~** to get the ID for the panel. Then pass that value as a request parameter to your next process task or report and use it with your input element's request token. For example: `@Request.lstEmployeeName~_@Request.pnlInstanceID~`

An easy way to do that is to use a Default Request Parameters element in your dasboard panel to set a default request variable to something like `pnlInstanceID = @Function.InstanceID~`.

# Query String Limits

There *are* limits on the amount of data you can pass in a query string. The first limit is **length**: browsers do not accept an unlimited number of characters in a URL. Moreover, the limit varies by browser, as follows:

| Browser | Max URL Bytes |
|---|---|
| Microsoft IE | 2,048 |
| Microsoft Edge | ~80K |
| Firefox | 65K |
| Chrome & Safari | 80K |
| Opera | 190K |

Trying to pass very large amounts of data in the query string is generally *not* a good idea. For example, instead of querying data in Report 1 and then passing *all* of it to Report 2 in the query string, you might instead pass only the key data and re-execute the database query in Report 2. If it's not feasible to run the database query twice, another approach is to use linked datalayers that span the two reports, sharing the query results.

The second limit is in the Logi Server Engine itself. It uses a fixed-size **array** with **100rows** to process request parameters. If you try to pass 101 parameters in your query string, an "index out of range" error will occur.

In summary, when passing data in query strings, don't exceed the length limit of the target browsers and don't pass more than 100 items as request parameters.

# Accessing Global Data with Tokens

In a Logi application there are two sources of data that can be considered "global" in scope: **Session** variables and **Cookies**. These are values that, once set, are available to *all* report and process definitions in an application. As such, they're an unrestricted method of "passing" information between report and process definitions.

Logi application Session variables and Cookies can be set in both report and process definitions. They can also be set by entities *outside* a Logi application and used in the Logi application. In the case of a Logi application for Java, there are special circumstances for Session variable sharing with external Java applications, see *About Logi Apps and Java*.

Session variable and Cookie values can be read in a Logi app using the @Session and @Cookie tokens. For example, if a process task creates a session variable called UserName, it can be accessed in any report definition with the token `@Session.UserName~`.

## Cookie Limitations

While we don't recommend that you use cookies to manage large amounts of data, they are very useful, so you should understand their limitations. Modern browsers adhere to the cookie limits described in **RFC 2965**. They will manage:

- At least 300 cookies total
- At least 20 cookies per unique host or domain name
- At least 4,096 per cookie (name and value text)

Popular browsers extend the maximum number of cookies allowed per unique host or domain name as follows:

| Browser | Max Cookies | Max Byte/Cookie |
|---|---|---|
| IE 7*, 8, 9, 10, 11 | 50 | 4,096 |

| Browser | Max Cookies | Max Byte/Cookie |
|---|---|---|
| Firefox | 50 | 4,097 |
| Chrome & Safari | no limit | 4,097 |
| Opera 9 | 30 | 4,096 |

* after system patch. Without patch, limit is 20 cookies. See this MS Knowledgebase article.

Generally, if the you attempt to set a domain cookie that exceeds the number of cookies allowed, the oldest cookie is deleted and the new cookie is accepted.

# Using HTML5 Local Storage

If a browser supports the HTML5 **Local Storage** technology, Logi User Input elements can optionally store their data using it. Data stored this way is retained between sessions on the client machine and is *automatically* restored as an Input element's default value when the page is redisplayed. The Local Storage size limit is approximately 5 MB and all values are stored as strings. There is no explicit method or token currently available in Logi apps for storing and retrieving data in Local Storage other than with an Input Element, however, it is accessible using scripting.

# PDF Templates

Form-based reporting provides a powerful method for making data available in popular formats. This topic describes how to use Logi Info to generate and fill-in **Adobe PDFForm Templates**. Filling PDF forms with data starts with the creation of the PDF form template. Once the PDF form has been designed and created, four more steps are required:

1. Add the PDF form template to your project
2. Create a template definition
3. Retrieve the correct data
4. Map that data to the form fields

The rest of this topic assumes that you've already created your PDF form template.

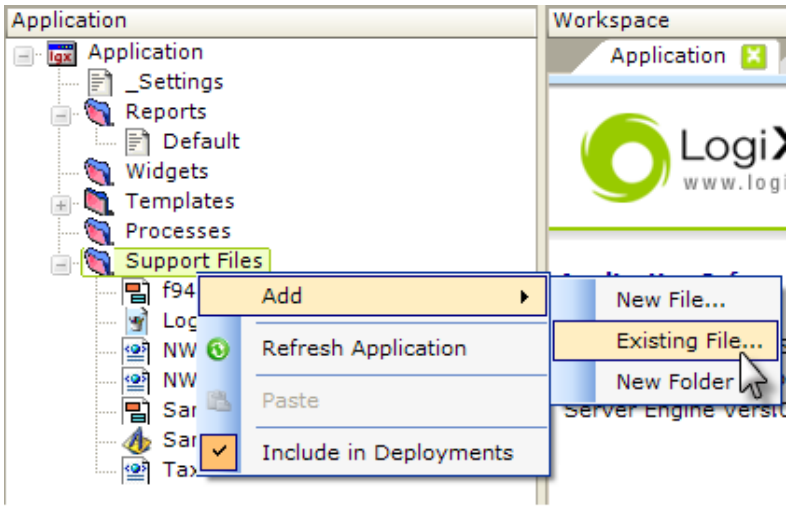
 In recent versions, the Firefox browser includes its own *built-in* PDF viewer which may not properly display all values, such as check box marks, in a filled template. If in doubt, use the real Adobe Acrobat browser plug-in to view filled templates.

## Add the PDF Form Template to Your Project

The term "template" is used frequently in this discussion, and to avoid confusion, let's clarify it:

- A **PDF form template** is a *.PDF file* created with a PDF editor such as Adobe Acrobat Professional. It looks like the finished output without any data and has form fields indicating where the data will go. This is the *target template*.
- A Logi Info**template definition** is an *.LGX file*, similar to a report or process definition, created with Logi Studio. It's a set of instructions to the Logi server engine to retrieve data and map it to the form fields in the target template.

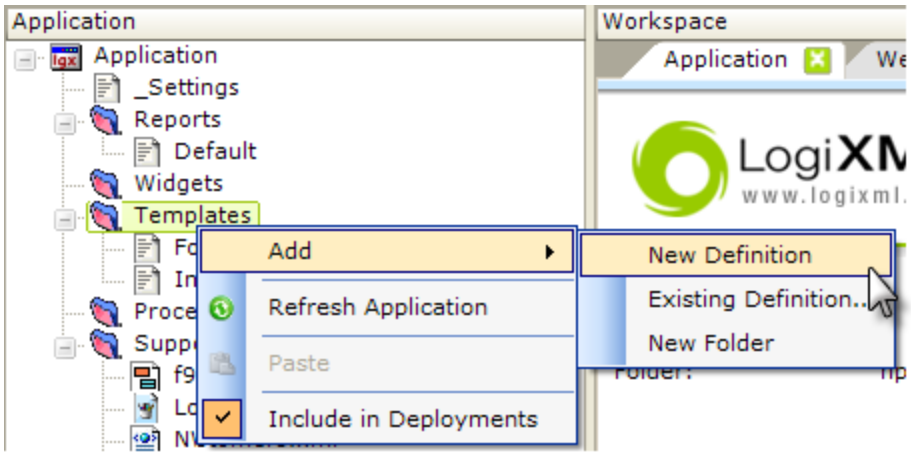
PDF form template files are managed within your Logi applicationas support files.



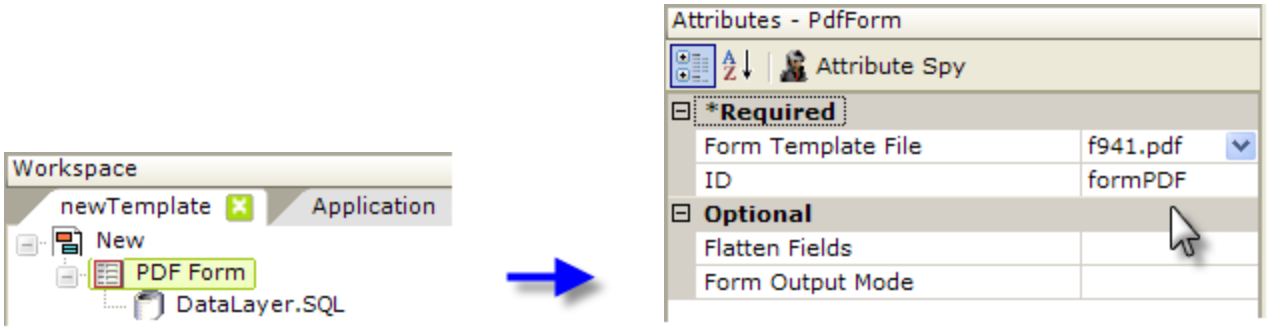

1. In Logi Studio's Application Panel, select and right-click the **Support Files** folder.
2. In the popup menus that appear, select the **Add** and **Existing File...** items.
3. Browse to your PDF form template (.PDF) file and select it. Click **OK** to add it to the project. The file will be *copied* to the **_SupportFiles** folder in your application project folder.

## Create a Template Definition

The next step is to create a new template definition that maps the data to the PDF form template fields:




1. In the Application Panel, select and right-click the **Templates** folder**.**
2. In the popup menus that appear, select the **Add** and **New Definition** items.
3. A new definition named "newTemplate" will appear beneath the Templates folder, ready to be renamed, and will open in the Workspace Panel for editing. The actual file will be created in the _Definitions/_Templates folder in your application project folder.

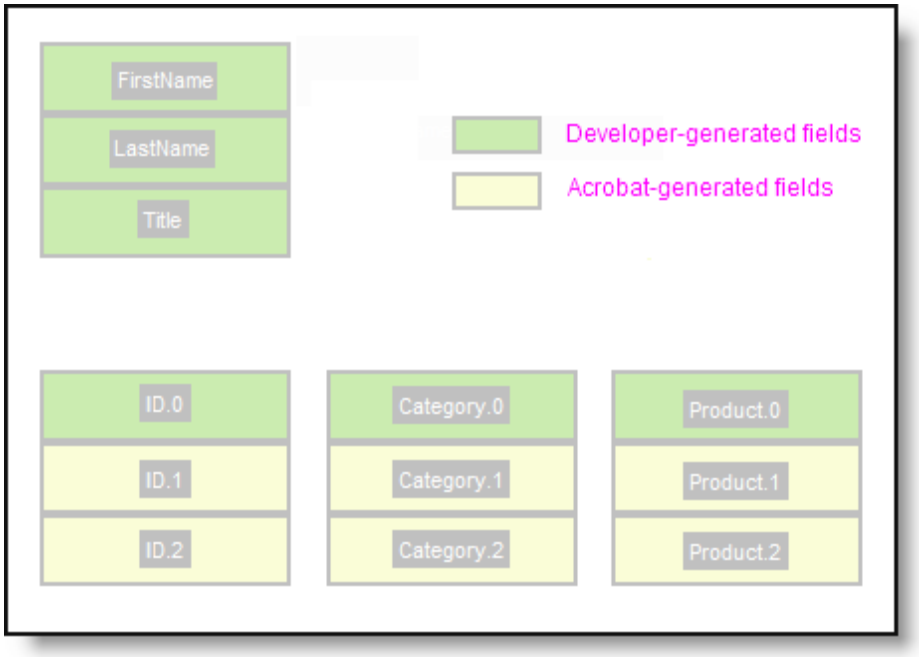


4. In the Workspace editor, add a **PDF Form** element to the new definition, as shown above.
5. Set its **Form Template File** attribute to the name of the PDF form template file you added as a Support File (it should be available in the attribute's drop-down list of options).
6. Finally, add a **datalayer** element to the definition and configure it to retrieve your report data (see the next section).

Template definitions cannot utilize more than **one** PDF Form element. The **Flatten Fields** attribute is optional but, when used, it applies to all fields within the form template file. Setting this attribute to *True* makes all fields in the output PDF document **read-only**. Its default value is *False*.

## Retrieve the Correct Data

It's important to understand exactly what data you will need. Consider the following:



PDF form template files can contain two types of form fields:

- **Developer-generated fields** - manually created by the developer when designing the PDF form template.
- **Acrobat-generatedfields** - automatically generated by the PDF editor when the developer *copiesandpastes* a field or group of fields multiple times. Acrobat-generated fields are often found in invoice-style form templates.

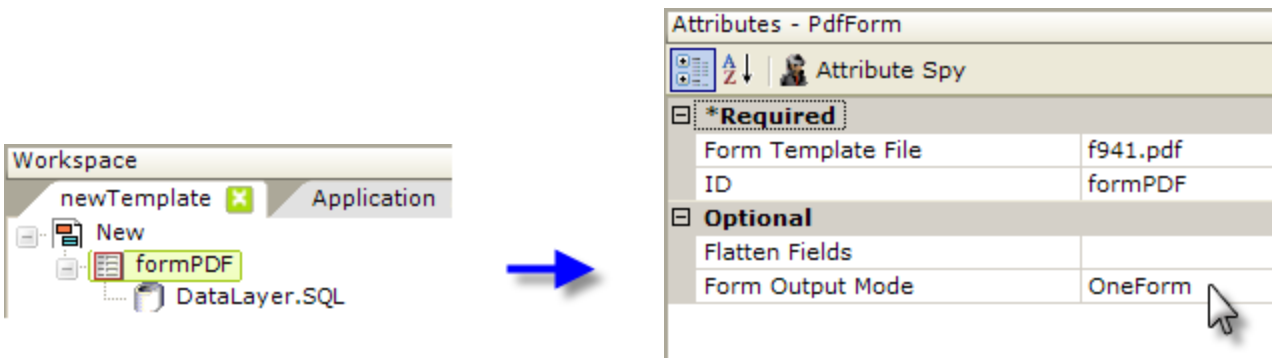
In order to correctly configure the datalayer element(s) in your Template definition, you need to understand how the Logi Engine processes the data (which is handled as XML data rows). You should ask these two questions:

1. Does the PDF Form template contain any **Acrobat-generated fields**? If so, is this form an "invoice-style" report?
2. Should the Template definition create a PDF document containing a single PDF Form ("One Form" output mode), or a PDF document with multiple PDF Forms merged into it ("One Form Per Data Row" output mode)?

Now you need to configure the proper output mode.

## Using *One Form* Output Mode

When filling a document with a single PDF form template,



in your template definition set the PDF Form element's **Form Output Mode** attribute to *OneForm*, as shown above. If the PDF form template has *no* Acrobat-generated fields, then only one row of data is needed to fill the form. The following example shows a SQL query and the resulting XML data needed to fill this kind of PDF form template:

```
SELECT FirstName, LastName, Title, SSN, DOB FROM Employee WHERE SSN = '111-22-3333
```

```
[XML]
<Sample>
       <Employee FirstName="Andrew" LastName="Webber" Title="Manager" SSN="111-22-3333" DOB="08-23-
1969" />
</Sample>
```

But, if the PDF form template *includes* Acrobat-generated fields, then multiple rows of data are needed to fill the form. The following example shows a SQL query and the resulting XML data needed to fill this "invoice-style" PDF form template:

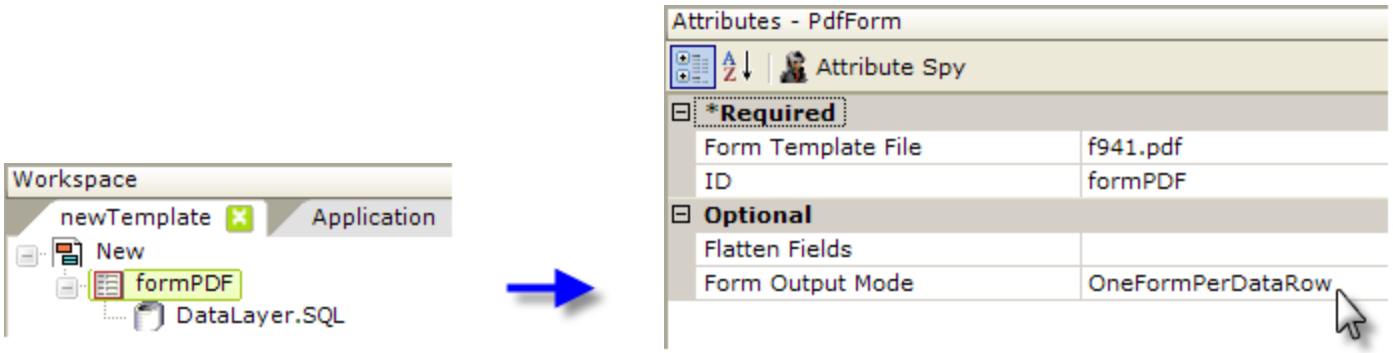
```
SELECT FName, LName, ProductID, Category, Product FROM EmployeeOrders WHERE SSN = '111-22-3333'
```

```
[XML]
<Sample>
       <EmployeeOrders FName="Andrew" LName="Webber" ProductID="33" Category="Fruit" Product="Apple" />
 <EmployeeOrders FName="Andrew" LName="Webber" ProductID="76" Category="Tools" Product="Hammer" />
 <EmployeeOrders FName="Andrew" LName="Webber" ProductID="54" Category="Clothing" Product="Shirt" />
</Sample
```

The *unique values* in the last three fields of the XML data above are used to fill-in the Acrobat-generated fields in the PDF form template.

### Using *One Form Per Data Row* Output Mode

When filling a document with a multiple PDF form templates,




in your template definition set the PDF Form element's **Form Output Mode** attribute to *OneFormPerDataRow*, as shown above. If the PDF form template has *no* Acrobat-generated fields, the one row of data is needed to fill each output form. The Logi server engine will merge each filled PDF form into a single document. The following example shows a SQL query and the resulting XML data needed to fill multiple PDF Forms of this type:

```
SELECT FirstName, LastName, Title, SSN, DOB FROM Employee
```

```
[XML]
<Sample>
 <Employee FirstName="Andrew" LastName="Webber" Title="Salesman" SSN="111-22-3333" DOB="08-23-1969" />
```

```
        <Employee FirstName="Charles" LastName="Hubbard" Title="HR Manager" SSN="444-55-6666" DOB="05-22-
1989" />
        <Employee FirstName="Meghan" LastName="Porter" Title="QA Manager" SSN="777-88-9999" DOB="07-21-1979"
/>
</Sample>
```

But, if this kind of PDF Form template *includes* Acrobat-generated fields, a **hierarchical** XML dataset is required to fill it. For example, a company might have 3 customers that place 3 orders each. In this case, an XML data row and all its child rows are necessary to fill-in a single form.

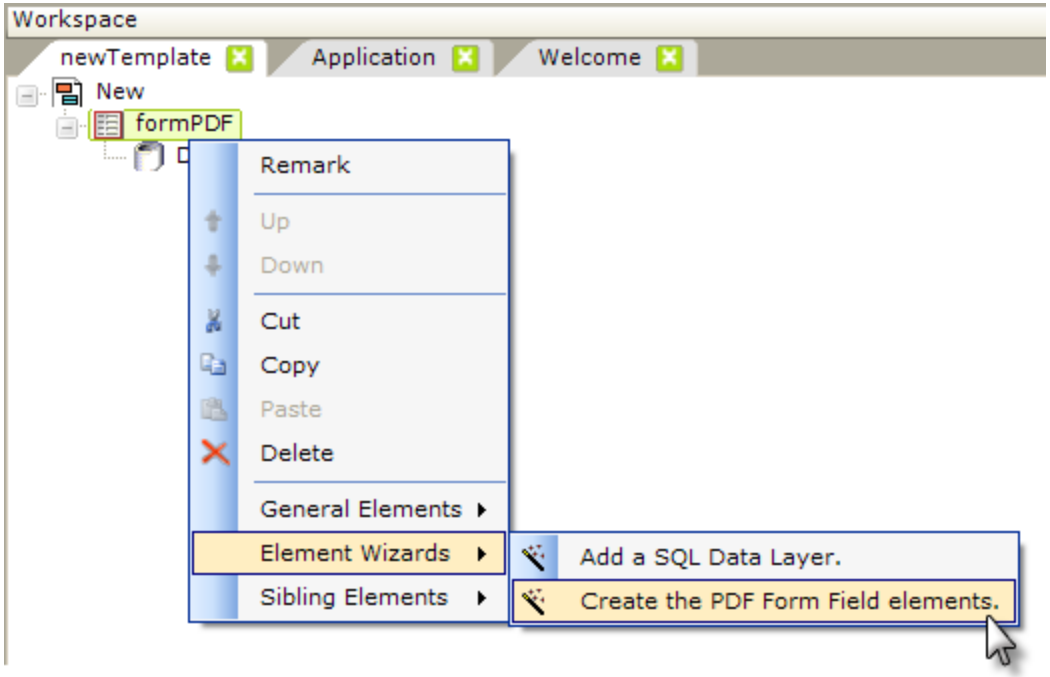
```
[XML]
<Sample>
 <Employee FirstName="Andrew" LastName="Webber" Title="Sales Manager" SSN="111-22-3333" DOB="08-23-1969" />
        <EmployeeOrders FirstName="Andrew" LastName="Webber" ProductID="33" Category="Fruit" Product="Apple" />
        <EmployeeOrders FirstName="Andrew" LastName="Webber" ProductID="76" Category="Tools" Product="Hammer" />
        <EmployeeOrders FirstName="Andrew" LastName="Webber" ProductID="54" Category="Clothing" Product="Shirt" />
 <Employee FirstName="Charles" LastName="Hubbard" Title="Product Manager" SSN="444-55-6666" DOB="05-22-1989" />
        <EmployeeOrders FirstName="Charles" LastName="Hubbard" ProductID="33" Category="Fruit" Product="Apple" />
        <EmployeeOrders FirstName="Charles" LastName="Hubbard" ProductID="76" Category="Tools" Product="Hammer" />
        <EmployeeOrders FirstName="Charles" LastName="Hubbard" ProductID="54" Category="Clothing" Product="Shirt" />
 <Employee FirstName="Meghan" LastName="Porter" Title="QA Manager" SSN="777-88-9999" DOB="07-21-1979" />
        <EmployeeOrders FirstName="Meghan" LastName="Porter" ProductID="33" Category="Fruit" Product="Apple" />
        <EmployeeOrders FirstName="Meghan" LastName="Porter" ProductID="76" Category="Tools" Product="Hammer" />
```

```
        <EmployeeOrders FirstName="Meghan" LastName="Porter" ProductID="54" Category="Clothing" Product="Shirt" />
</Sample>
```

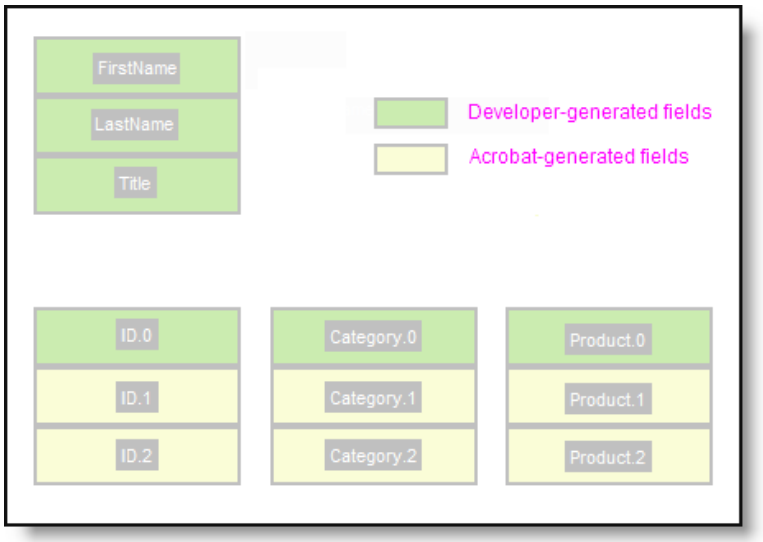
The XML data must be **hierarchically shaped** to produce multiple, invoice-style PDF documents from a single PDF form template. This could be done in a variety of ways, for example by using SQL query GROUP BY statements or grouping elements.

## Map Data to the Form Fields

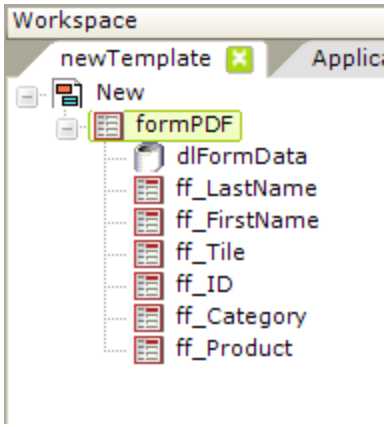
The template definition models the PDF form by utilizing a **PDF Form Field** element for each field or group of fields in the PDF form template file.



To speed the process, developers can use a Logi Studio **wizard**, shown above, to generate the PDF Form Field elements required to fill the PDF template. The wizard will add one PDF Form Field element to the Template definition for each user-generated field, and it will also add one (but *only* one) PDF Form Field element for each *group* of Acrobat-generated fields.



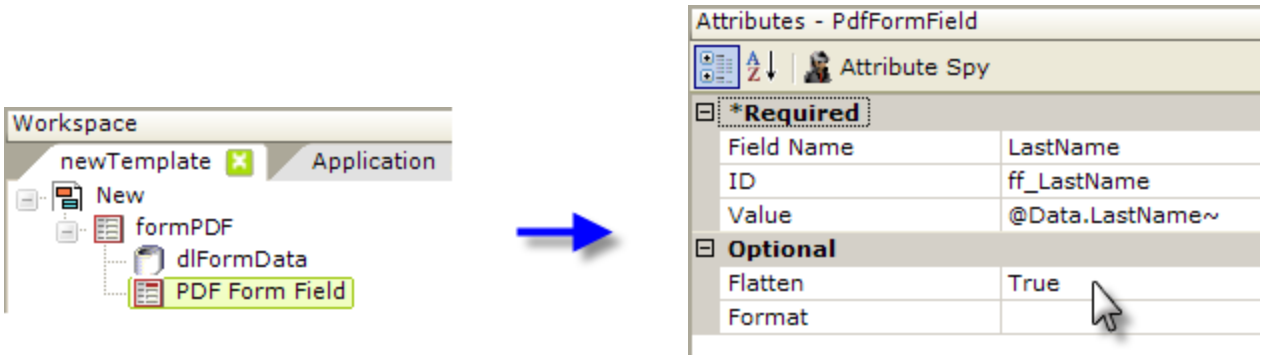

So, for the example form shown above, the wizard will add the following PDF Form Field elements in the definition:



The *ff_ID*, *ff_Category* and *ff_Product* elements each represent a *group* of Acrobat-generated fields. When the Logi server engine fills-in the data, it does so row-by-row to provide data for all the fields in each group.

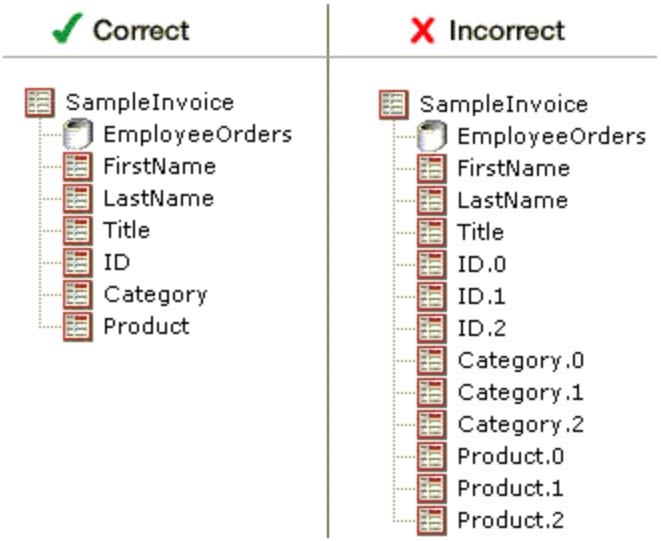
To manually add a form field to the definition:

Developers who don't want to use the wizard can add PDF Form Field elements to the definition manually, as follows:




1. In the Workspace editor, add a **PDF Form Field** element below the PDF Form element, as shown above.
2. In its **Field Name** attribute, enter the *exact* field name from the PDF Form Template. This is **case-sensitive**.
3. Assign a value for the **Value** attribute. The value can be a token, a static value, a script formula, or any combination of them.
4. Make the field read-only by setting the **Flatten** attribute to *True*. (This will be overridden, however, by the Flatten Fields attribute of the PDF Form element, if it's been configured).
5. (Optional) Select a format for the **Format** attribute.

Repeat for each field and group of fields in the PDF Form Template file.




*Do not* add an element for *each* field within an Acrobat-generated group fields. The PDF form template and the Logi template definition are now complete and you're ready to begin producing filled PDF documents.

# Exporting to Adobe PDF

The Adobe Portable Document Format (**PDF**) is a well-established file format standard and a widely-accepted method of distributing reports.

The following topics discuss how to export Logi reports to Adobe PDF:



## About Exporting Reports to PDF

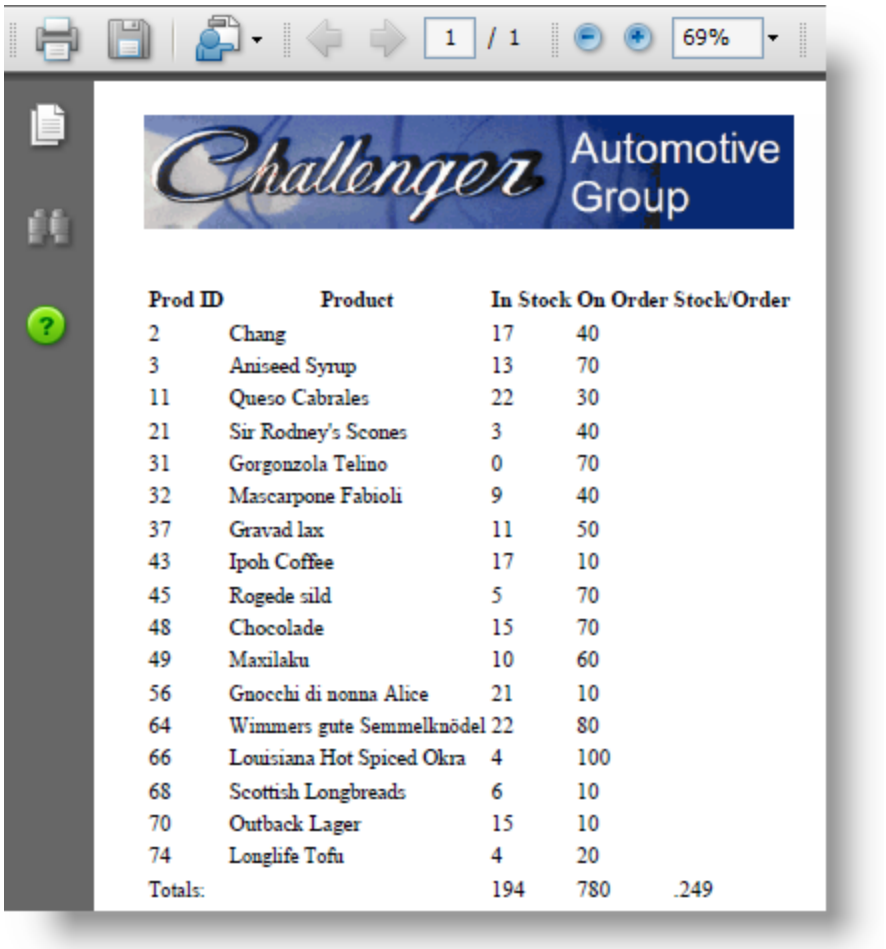
The PDF format is useful because of its ability to faithfully reproduce the appearance of a *document*, including graphics, images, and text, so that it can be easily viewed using the Adobe Reader program or browser plug-in. Notice the emphasis on "document" - PDF was *not* initially designed as a format for reproducing browser content and it doesn't always do so very well. Even though PDFs can be viewed within a browser, the two technologies are quite different.

For example, recent browser versions and Logi products support doctype declarations, which serve to tell the browser how to handle content layout, how to apply style, and more. Report pages can look quite different with different doctype selections. However, the PDF format *does not* support doctype declarations, and therefore can't represent everything exactly as the browser does. So, exported report pages may not look exactly like they do in the browser if a doctype has been applied.

What does this mean? It means developers and users should be realistic in their expectations about just how faithfully a Logi report can be reproduced when exported as a PDF document.

Chart Canvas charts are exported as SVG objects rather than as images. This results in Chart Canvas charts exported to PDF having extremely high resolution - they can be zoomed or printed with high-quality at any resolution.

Logi Studio provides the **Action.Export PDF** element so that Logi reports can be exported to PDF files.




| Prod ID | Product | In Stock | On Order | Stock/Order |
|---|---|---|---|---|
| 2 | Chang | 17 | 40 | |
| 3 | Aniseed Syrup | 13 | 70 | |
| 11 | Queso Cabrales | 22 | 30 | |
| 21 | Sir Rodney's Scones | 3 | 40 | |
| 31 | Gorgonzola Telino | 0 | 70 | |
| 32 | Mascarpone Fabioli | 9 | 40 | |
| 37 | Gravad lax | 11 | 50 | |
| 43 | Ipoh Coffee | 17 | 10 | |
| 45 | Rogede sild | 5 | 70 | |
| 48 | Chocolade | 15 | 70 | |
| 49 | Maxilaku | 10 | 60 | |
| 56 | Gnocchi di nonna Alice | 21 | 10 | |
| 64 | Wimmers gute Semmelknödel | 22 | 80 | |
| 66 | Louisiana Hot Spiced Okra | 4 | 100 | |
| 68 | Scottish Longbreads | 6 | 10 | |
| 70 | Outback Lager | 15 | 10 | |
| 74 | Longlife Tofu | 4 | 20 | |
| Totals: | | 194 | 780 | .249 |

Developers can give users the ability to export a report in two ways: **manually** (to a PDF viewed in their own browser) or **automatically** (to a PDF file) based on an event or schedule. Manual exports are configured within *report definitions* and automated exports are configured within *process definitions*.

The PDF export engine uses the Gecko-based rendering technology developed by Mozilla and found in the Firefox browser, resulting in Chart Canvas charts being exported as SVG objects rather than as images. This means that Chart Canvas charts exported to PDF have extremely high resolution - they can be zoomed or printed with high quality at any resolution.

### Exporting Links

By default, links in reports are *not* enabled when they're exported to PDF. However, "live links", which are those that do not use Action.Link, *can* be enabled in the PDF output. To enable them, set the **Target.PDF** element's **Show Links** attribute to *True*.

Live links are created by using a **Label** element, with its **Format** attribute set to *HTML*, and adding `<a>` tags around the text in its **Caption**. Here's an example of a valid Caption attribute value for this purpose:

```
<a href="https://www.logianalytics.com">Visit the Logi Analytics web site</a>
```

A *complete* URL must be specified, as shown, not a *relative* URL.

### Exporting Report Titles and Filter Information

By default, report titles and filter information are not included in PDF exports. To enable this functionality, set the global constants `rdIncludeReportTitleInExport` and `rdIncludeFilterInfoInExport`to *True*.

### Specifying a Custom Temp File Location

In .NET Info applications, the PDF rendering engine used during exports writes temporary files to the `C:\Windows\Temp` folder by default. However, some anti-virus tools will flag multiple file-writing operations to this folder as potentially dangerous behavior

and block them. Developers who encounter this situation can tell the PDF rendering engine to write temporary files elsewhere by adding a section to the Info application's `web.config` file:

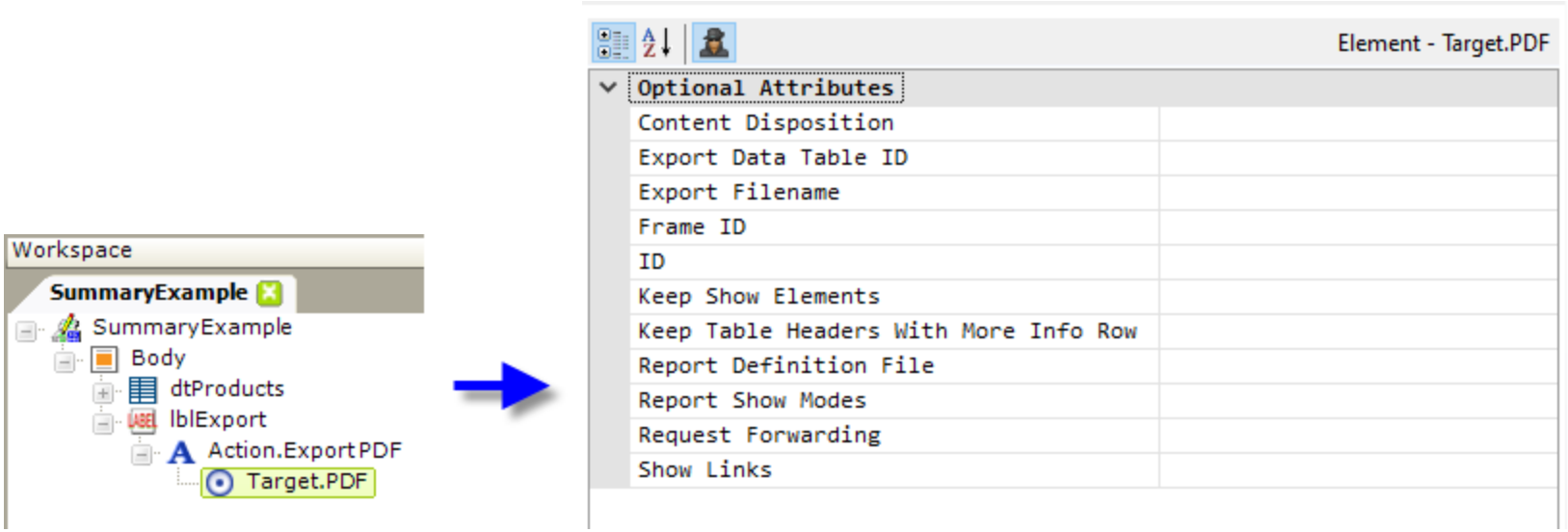
```
<configSections>
<section name="ABCpdf9.Section" type="WebSupergoo.ABCpdf9.ConfigSection, ABCpdf" allowLocation="true"
allowDefinition="Everywhere" allowExeDefinition="MachineToLocalUser" overrideModeDefault="Allow"
restartOnExternalChanges="true" requirePermission="true" />
</configSections>

<ABCpdf9.Section>
<Preferences>
 <clear />
 <add Key="TempDirectory" Value="C:\my_PDF_Temp_Folder" />
</Preferences>
</ABCpdf9.Section>
```

Add the code above that starts with `<ABCpdf9.Section>` just after the `</configSections>` section closing tag, as shown. Provide your own custom folder value for the TempDirectory key.

 Ensure that your custom temp folder already exists - the rendering engine will not create it.

# PDF - Exporting a Report Manually

Here's an example of how to create a report with a link that allows the report to be exported manually:




1. In your report definition, add an **Action.Export PDF** element as the child of a **Label**, **Image**, **Button** or **Chart** element, as shown above.
2. Beneath it, add the required **Target.PDF** element.
3. If the report to export *is* the current report, you don't need to do anything else. Just save your definition, preview or browse your report, and click the link to export it. It should open in the PDF viewer associated with your browser. It's that simple.
4. To specify the way PDFs exports are generated, set the Content Disposition attribute in the Target.PDF element to either **Attachment** or **Inline.** By default, the attribute is "empty", PDFs use browser settings. When the attribute is set to "Attachment", you are prompted to download the PDF. When the attribute is set to "Inline", you are prompted to display the PDF inside the Web page.
5. In order to export the current report while maintaining any **sort order** the user may have applied, set the Target.PDF's **Report Definition File** attribute to *CurrentReport*.

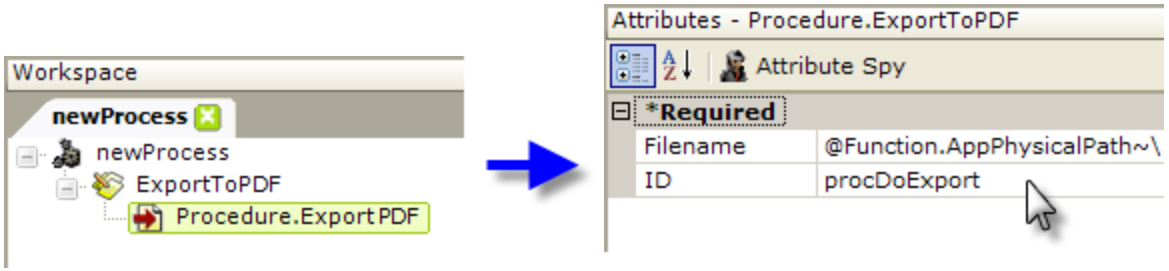
6. If the report to export *is not* the current report, specify that report by setting the Target.PDF's **Report Definition File** attribute appropriately.
7. If you wish to export *only* a specific Data Table, provide its ID in the **Export Data Table ID** attribute. This prevents anything else in the report, such as headers or footers, images, etc., from being exported.

What happens: the report is exported to a *temporary* PDF file which is created in your project folder's rdDownload folder on the web server. The temp file is then opened automatically in your browser. Temporary files are cleaned up automatically over time.

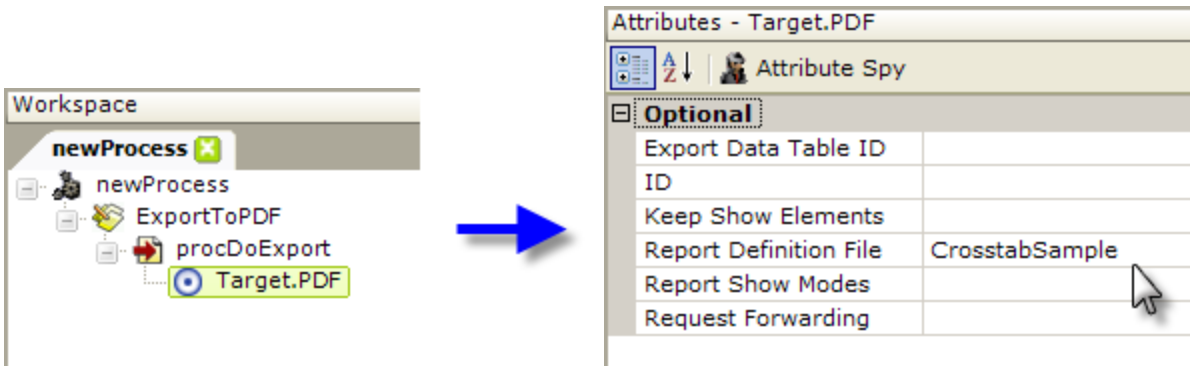
If you wish to export to PDF in "landscape" page orientation, you should use the **Printable Paging** element, as described in "Creating Printable Reports" on page 218.

# PDF - Exporting a Report Automatically

The following example, for Logi Info only, shows how to create a **process task** that exports data automatically:




1. In your **Process** definition, add a **Procedure.ExportPDF** element beneath your Task element.
2. In the element's **Filename** attribute, specify the output path and filename, on the web server, for the exported report. The filename should include the ".pdf" file extension. For example, this value uses a token to export the report to a folder called myExports within your project folder: `@Function.AppPhysicalPath~\myExports\myfile.pdf`




3. Add the required **Target.PDF** element.

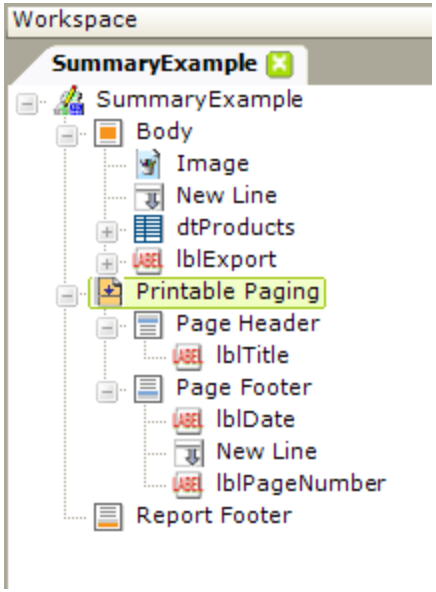
4. In the element's **Report Definition File** attribute, you must specify the report to be exported (a blank value is not allowed). The *CurrentReport* option in the suggested values **is not valid** in this case, which means that the report will be exported but will not reflect any sort order that the user may have applied before clicking the export link.

4. If you wish to export *only* a specific Data Table, provide its ID in the **Export Data Table ID** attribute. This prevents anything else in the report, such as headers or footers, images, etc., from being exported.

When your task runs, the specified report will be exported to a **temporary** file in your project folder's **rdDownload** folder on the web server; the temp file is then copied to your output file. The file is not opened in your browser automatically.

# PDF - Adding Exported Headers and Footers

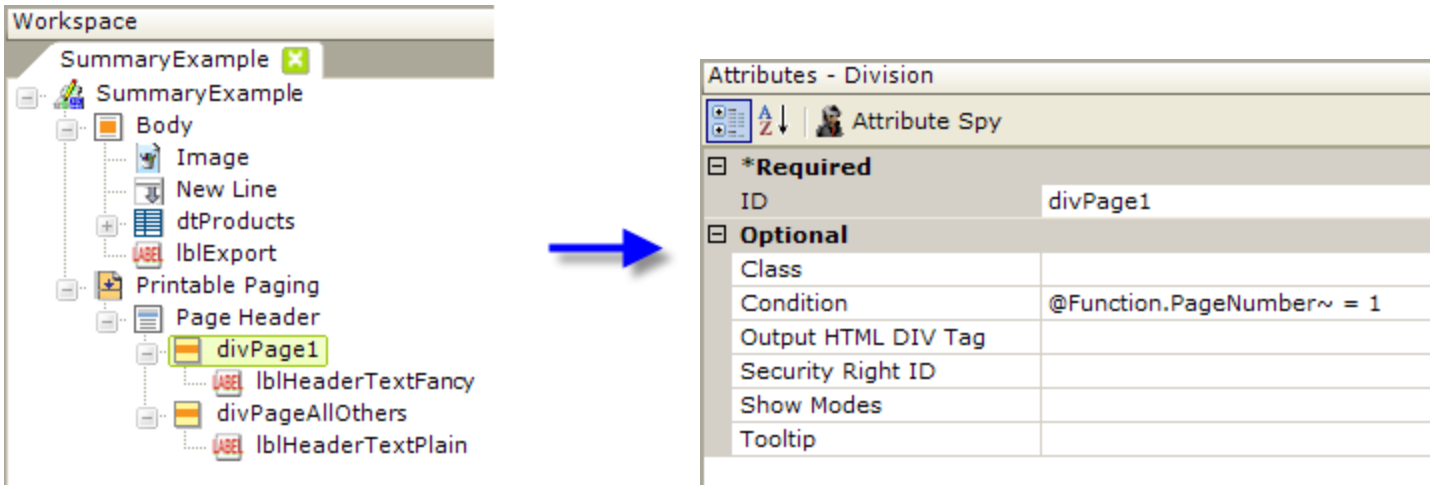
Reports exported to PDF are paginated and you may want to have a report header and/or footer appear on each exported page. This can be done, without affecting the normal appearance of the HTML report output in a browser, using the **Printable Paging** element.




As shown above, the **Printable Paging** element is added beneath the report's top-level Report element. It has its own **Page Header** and **Page Footer** child elements which are containers for **Label** and other elements. In the example above, the captions of the footer labels could use the `@Date.Today~`, `@Function.PageCount~`, and `@Function.PageNumber~` tokens.

When the report is run and viewed in a browser, the header and footer under the Printable Paging element will *not* be visible. When the report is exported to PDF, the header and footer *will be* visible on each exported page.

Suppose you want to have different-looking headers or footers on an exported page depending on the page number? For example, you might want Page 1 to have a fancy header and the following pages, a less-fancy header. Developers can do this using **Division** elements:

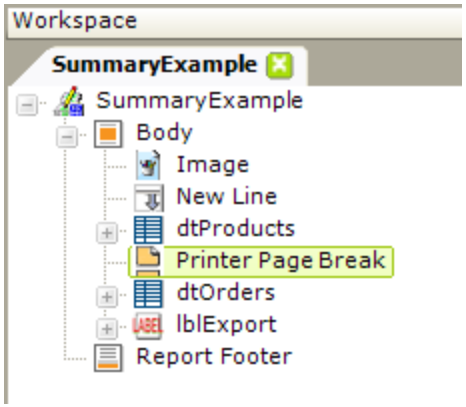



As shown in the example above, place Division elements beneath the Page Header element and then set their **Condition** attributes to evaluate the PageNumber token. The Condition attribute value for the first division is shown, and the value for the second division might be `@Function.PageNumber~ > 1`. Each division, and its child elements, will appear in the export then, depending on the page number. The same functionality is available in exported report footers.

Adding a header or footer beneath the Printable Paging element will **increase** rendering time. Their inclusion causes the export engine to have to make multiple passes through the entire report while generating the PDF, and the performance hit incurred may be acceptable for short reports but not for longer reports. The alternative is to use regular Report Header and Report Footer elements, which will appear in the report in a browser, rather than the Printable Paging element's Page Header and Page Footer child elements.

# PDF - Forcing Page Breaks in Exports

When a report is paginated during an export, it may be useful to be able to force a page break, for example, to ensure that sections of the report start on new pages. This can be done using the **Printer Page Break** element.




You can add one or more **Printer PageBreak** elements beneath the **Body** element to break the exported pages where desired. In the example above, these elements ensure that the "dtProducts" table and the "dtOrders" table start on new pages. When the report is viewed in the browser, however, like the Printable Paging element, the Printer Page Break element has no effect.

## Forcing a Data Table Row Break

When working with Data Tables with multi-line rows, developers may want to ensure that rows break cleanly across pages of their PDF exports. In other words, they want to prevent different lines of the same Data Table row from appearing on different PDF pages. This can be accomplished through CSS, using code like:

```
.noPageBreakCell { page-break-inside: avoid; }
```

Assigning the above class to a **Data Table Column** element will create the desired effect.

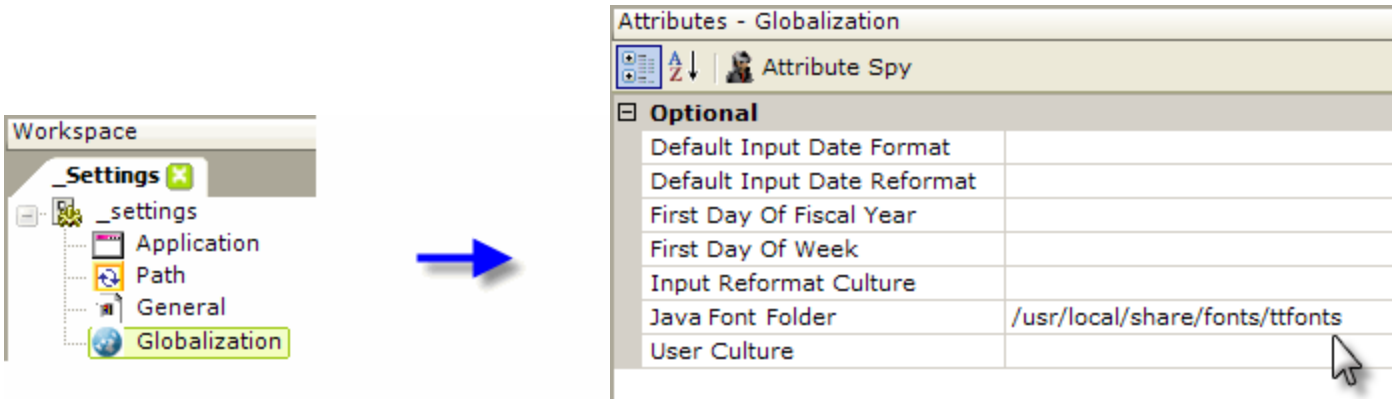
### Printer Page Break Attribute Restriction

The Gecko-based rendering engine used in PDF exports does not fully support printer page breaks within HTML tables. This means that enabling the Printer Page Break attribute for the **Group Header Row**, **Group Summary Row**, and **More Info Row Column** elements, which use tables internally, will produce unpredictable results in PDF exports.

As a "last resort" alternative you can make these page breaks work by switching to the older IE rendering engine; this is done by creating the constant `rdPdfRenderingType=`*MSHTML* in your _Settings definition. Unfortunately, this will also eliminate some of the benefits of the Gecko engine, such as the ability to export Chart Canvas Charts as SVG objects, and so it's not generally recommended.

# PDF - Special Considerations for Java Apps

Developers creating Logi apps that use the **Java** libraries and include less commonly-used fonts or special character sets may need to take steps to ensure that those character sets or fonts are also used in their PDF exports. This is done by adding the **Globalization** element in the _Settings definition, and setting its **Java Font Folder** attribute:



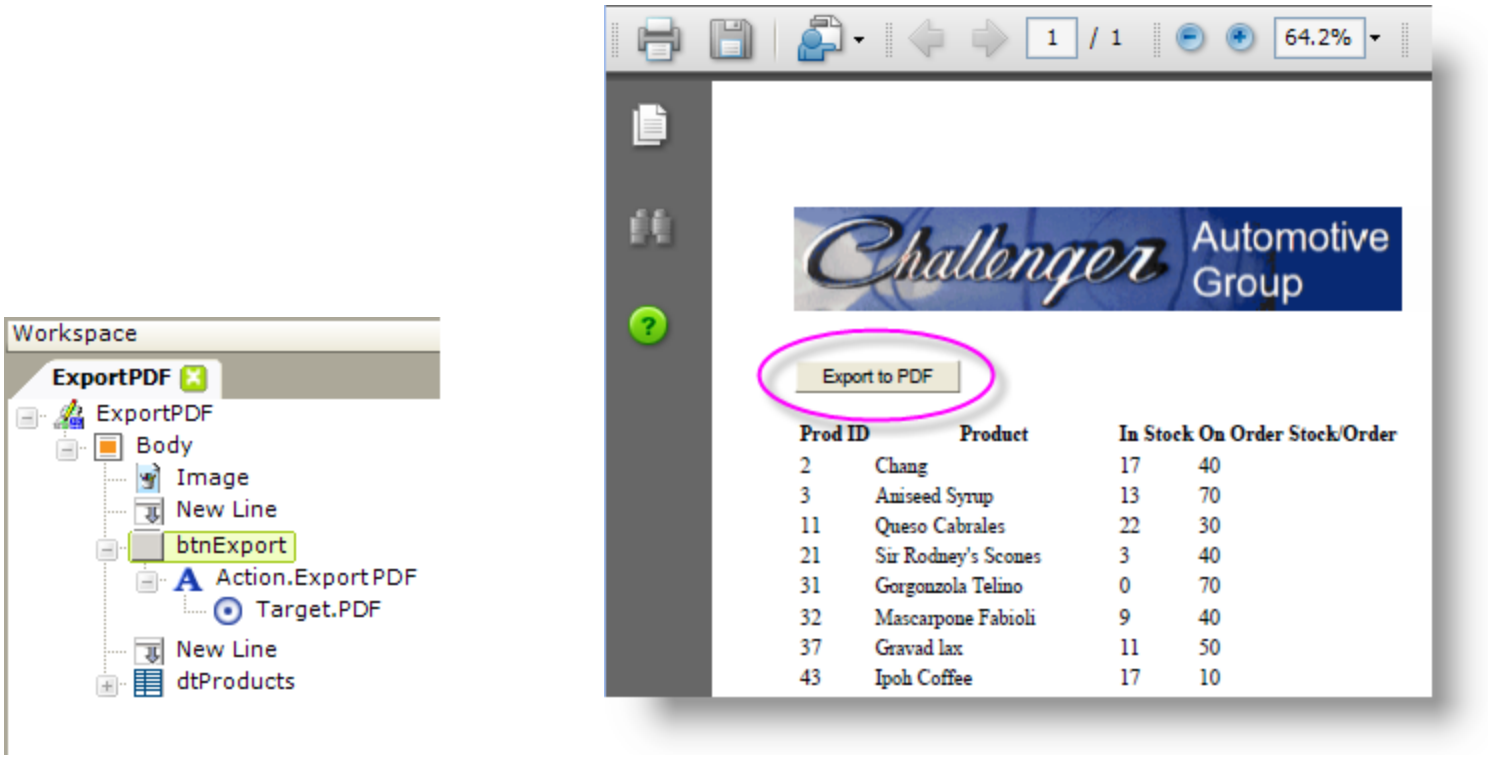

The value of this attribute identifies the folder where **TrueType** font files, for use with PDF exports, are located. These are required when PDF exports use fonts that contain characters that are not in the ISO 8859_1 character set, such as Arabic, Cyrillic, and Korean language characters. Depending on your Java environment, you may even need to point the Logi application to the location of your regular fonts in order to ensure that they'll be used.

A typical value for a Windows installation might be something like `C:\Windows\Fonts`. For a Linux installation, the value might be something like `/usr/local/share/fonts/ttfonts`. This attribute is only functional when creating a Java application.

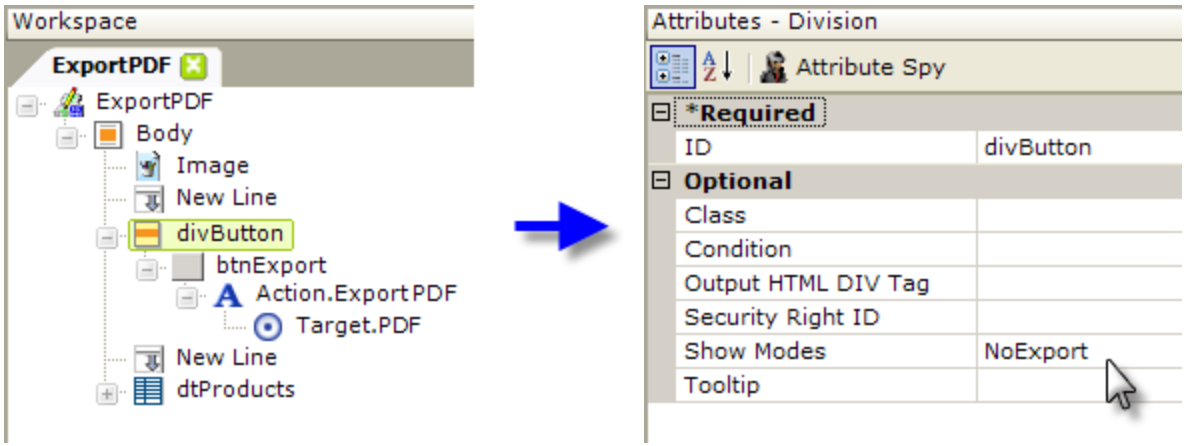
CSS classes are cached for Java application exports to PDF. If you need to stop this caching, add this case-sensitive constant in your _Settings definition: `rdJavaPdfClearCache = True`.

# PDF - Hiding Elements when Exporting

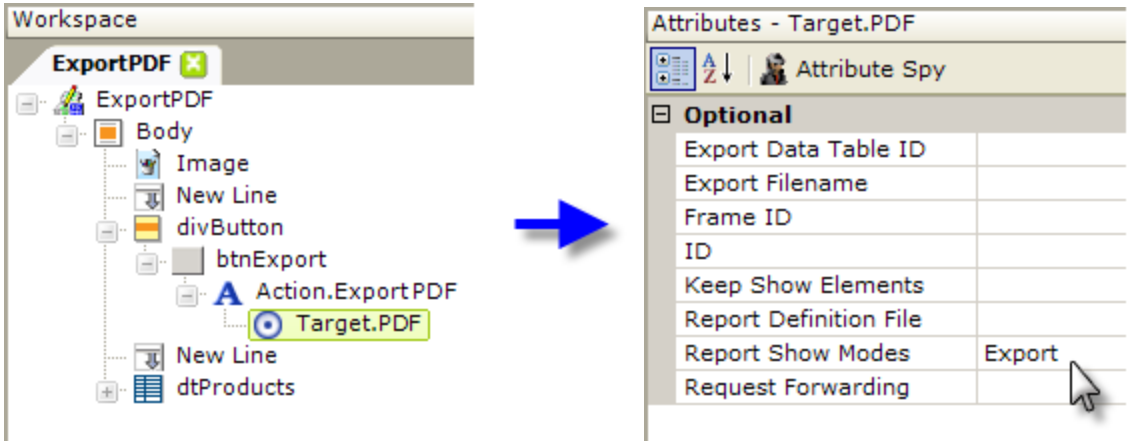
When exporting to PDF, developers commonly want to hide some of the elements in the report page, like the Export button or link. This can be done very easily using **Show Modes**.



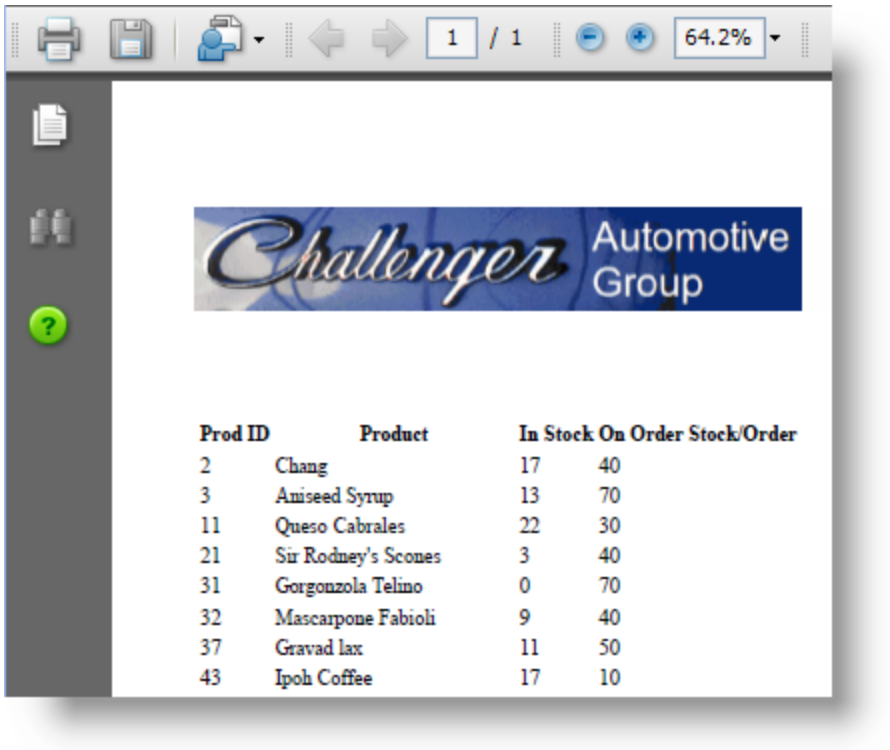

Consider the example shown above. The definition, on the left, includes a button that can be clicked to export the report to a PDF. However, when it's exported, shown on the right, the **Export to PDF** button appears in the PDF itself. We don't want that button to appear in the PDF.




The first step in "hiding" the button in the export is to place the button beneath a **Division** element. This element supports Show Modes, which the button, by itself, does not. Set the Division element's **Show Modes** attribute value to an arbitrary string, such as "NoExport", as shown above.



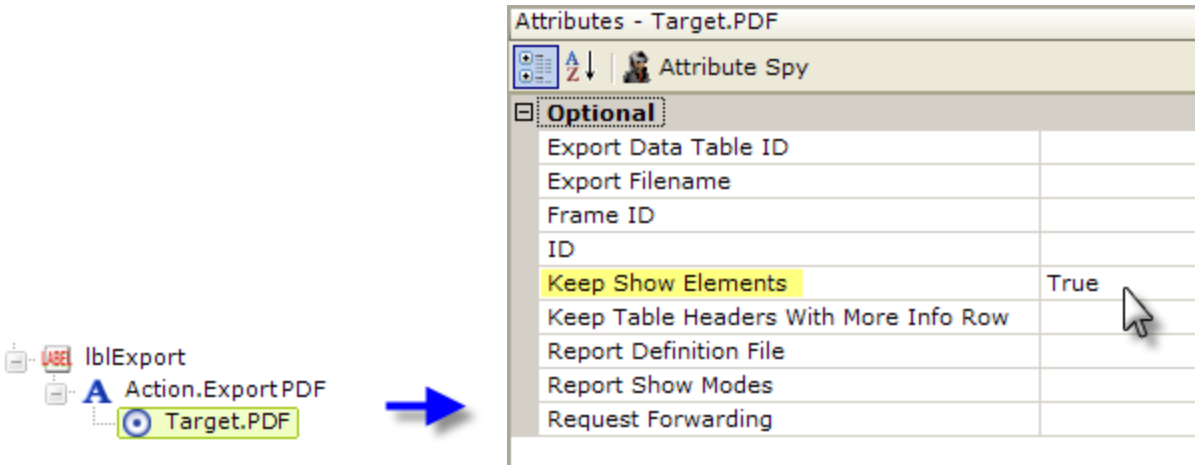

Then set the Target.PDF element's **Report Show Modes** attribute value to an arbitrary string that *does not match* the string used in the previous step, as shown above.




When the report is run and exported, as shown above, the PDF no longer includes the **Export to PDF** button. Any element whose Show Modes attribute value does not match the Target.PDF element's Report Show Modes attribute will not appear in the export.

# PDF - Exporting More Info Rows

If your report contains a Data Table and you're using the **More Info Row** element to show/hide additional detail data within the table, you may want the detail rows (when visible) to be exported along with the main table data.




To enable this, as shown above, set your Target.PDF element's **Keep Show Elements** attribute to *True*.

Keep Show Elements works with Action.Show Elements (used to show/hide the More Info Row) so that items that have been shown or hidden by the user remain that way when the report is re-displayed or exported.

In addition, you may want to set the **Keep Table Headers with More Info Row** attribute to *True* if you want to ensure that all the tables in the report have repeatable table headers, even if they have a More Info Row containing other reports with tables.

It is possible to "nest" More Info Row elements, so that you have one More Info Row element as a child of a table that's a child of another More Info Row. However, be aware that the lowest More Info Row contents may not export correctly, or at all, so we don't recommend this design arrangement if exporting of expanded rows is desired.

# PDF - Cascading Style Sheet Support

When a PDF export is being generated, the export engine processes any **style sheet information** along with the report data. The style sheet assigned to the report is used; you do not need to do anything additional to specify the style sheet. Because your browser is used to render the PDF, your browser's style sheet limitations or quirks apply to the exported PDF.

If desired, you can use **two different style sheets** in your report definition, one for the report and one for the PDF export:

1. Select the root element of your report definition and set its **Default Show Modes** attribute value to an arbitrary string, such as "Normal".
2. Add two Style Sheet elements to your report definition.
3. Assign one Style Sheet element's **Show Modes** attribute a value that matches the string used in Step 1, "Normal". This style sheet will be used for the regular HTML report.
4. Assign the other Style Sheet element's Show Modes attribute a value that matches the Target.PDF element's Report Show Modes attribute value (both should be an arbitrary string that does not match the string from Step 1, such as "Export".

Now when the report runs in a browser, the first style sheet will be applied; when it's exported to PDF, the second style sheet will be applied.

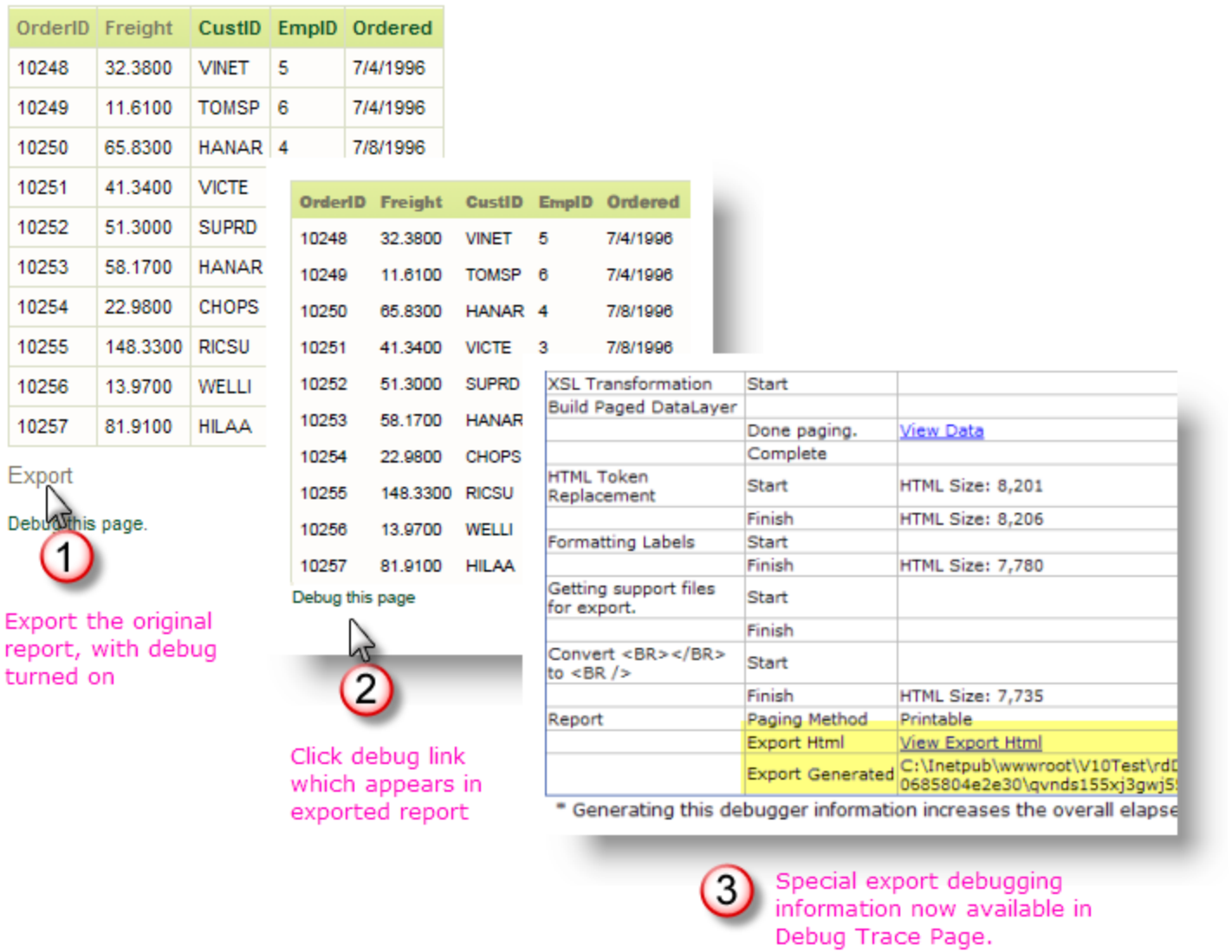
What if you want to combine the effects from this topic and "PDF - Hiding Elements when Exporting" on page 283, i.e. hide some elements and use different spreadsheets? Just remember that the Show Modes attribute can accept several values, separated by a comma. To combine the examples from these two sections, ensure that the root element's Default Show Modes attribute includes both Show Modes values: "Normal,NoExport" (do not enter the double quotes).

Our PDF export engine has **full support** for all symbols in the **Unicode** character set. Developers can also apply specific fonts in a CSS class. If you're having difficulty rendering to PDF in your native language, ensure that the fonts specified in your CSS classes support the language.

We *do not* recommend that you use absolute positioning of elements through CSS if you plan to export a report to PDF. The export engine will attempt to "fit" your report to its page size and elements so positioned may wind up in unexpected locations.

# PDF - Debugging Exports

Logi Studio includes a feature that allows developers to debug their exports. When the debugger has been set to *Debugger Links* and the report is run, and then exported, a debug link will be included at the bottom of the exported report:




This mechanism allows you to review the export process in more detail and can help to diagnose any problems that may arise.

# PDF - Export Considerations

The following apply when exporting to PDF:

- The Logi Info PDF export engine will not export Data Table captions in Java applications. Developers are advised to use a separate Label element, or a Data Table Header Row child element, set to *Top* position, instead of using the table's Caption attribute, to provide an exportable caption in Java apps.
- Chart color transparency is not available in exports to PDF. Non-animated charts displayed in Logi HTML reports are rendered as .png images and support transparency, however, when exported to PDF, these charts are rendered as .jpg images, which do not support transparency.
- It's not possible to support links that may use relative URLs in an exported document, as the document can be saved and relocated. As a result, links in general are *not* supported in exported PDFs. They will work when Studio's *Debugging Links* feature is turned on during development, but when that link is removed for a production release, they will no longer work.
- Data Table column headers will be repeated on each page of PDF export *only* if there are no sub-elements within the report. This includes Sub Report, Sub Data Table, and More Info Row.
- Data Table column headers will be repeated on each page of a PDF export *only* if there are one or more characters in the column header text. If you want a column that has no header text to be repeated, you must include a single space character in its Header attribute value.
- When exporting to a PDF from a web browser on different computers, the page dimensions may be affected by different screen resolutions. In addition, when using different browsers, exported reports may appear differently, as different browsers handle styling differently. Developers may care to use a Printable Paging element in their report definitions, to set absolute page dimensions. For more information, see "Creating Printable Reports" on page 218.
- Exporting reports that include characters that are read right-to-left, such as Arabic, from a Windows server requires that special OS files be installed, as follows: Go to Regional and Language Options in Control Panel; select "Install files for complex script and right-to-left languages (including Thai)" under the Languages tab; when prompted, insert the Windows OS CD and run the install; restart the computer once the files are completely copied and the restart prompt comes up.

# PDF - Export Errors

"Error Exporting to PDF - HTML render is blank."

- *Insufficient Process Identity Permissions* - Ensure that the account, application pool, or identity used by your web server to run your Logi app has *full* file access permissions to the Logi app directory or any folder outside it that you may have designated as an exported file destination.
- *Insufficient Temp File Space* - Ensure that you have enough space for temporary Logi application files, by *shortening* your temp file cleanup interval (60 minutes by default). For more information, see our document *Temporary Cache File Management*.
- *Insufficient Permissions for Windows/Temp* - Ensure that the account, application pool, or identity used by your web server to run your Logi app has Read/ Write file access permissions to C:\Windows\Temp (if using IIS) and the ability to create folders.
- *Web Server Unable to Resolve URLs to Itself* - This means that when testing using the IP address of another machine, your code will work but when you try and render a web page on the local server using a URL, you may get errors. This is a typically a DNS problem.

"The type initializer for WebSupergoo.ABCpdf7.Internal.NDoc threw an exception"

- This is usually the result of installing the Logi Server Engine on the wrong type of system. It will occur, for example, if you install the 64-bit version of the Logi engine on a 32-bit system, or vice versa, if you install the 32-bit engine on a 64-bit system. Logi products are available in 32-bit and 64-bit versions; please ensure that you have the correct version installed, based on the system's environment and the IIS configuration for running the Logi application (under Windows 7+, you can configure specific applications to run in 32- or 64-bit mode under a 64-bit version of the OS).

"Chart Canvas chart exported to PDF as a solid, colored block"

- When creating Chart Canvas charts that will be exported to PDF, the **Chart Canvas** element *must* have Height and Width attribute values set. In addition, in the web server settings, the application's rdDownload folder should have Anonymous Authentication enabled, and any other authentication disabled.
  Height and Width values are no longer required.

"Invalid character at..." or similar parsing error

- Exports may fail if XHTML reserved characters are included in the data. The application fails and produces an error message that looks like a parsing error. The most common culprit is the unencoded "&" character. Replacing the "&" character with "&" is often an easy solution.

# Excel Template

This topic guides developers in creating template definitions for use with **Microsoft Excel templates**. A Logi template definition acts as a "blueprint", mapping how the Excel template will be filled with data by the Logi Server Engine. Filling an Excel template with data starts with the creation of the Excel template file; once it has been created, four more steps are required:

- Adding the Excel Template to Your Logi Application
- Creating a Template Definition
- Configuring the Data Ranges
- Working with Hierarchical Data

For information about how to call templates from within Logi applications, see "Form-based Reporting" on page 224.

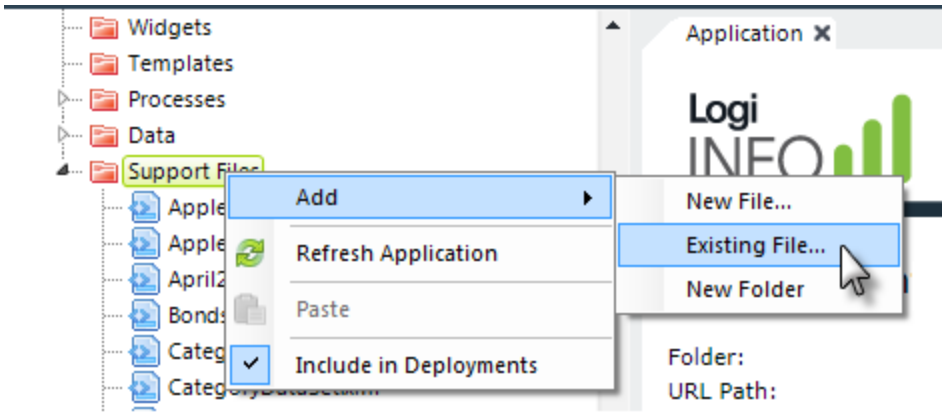
For information about creating template files, see "Create an Excel Template" on page 304.

## Adding the Excel Template to Your Logi Application

The term "template" is used frequently in this discussion, and to avoid confusion, let's clarify it:

- For this purpose, an *Excel template file* is an **.xltx**, **.xltm**, or **.xlt** file (depending on version) created with Microsoft Excel. It looks like the finished worksheet without any data and has cells reserved for the data. This is the *target template*. Use of other Excel file types are not recommended.
- A Logi Info *Template definition* is an **.lgx** file, similar to a report or process definition, created with Logi Studio. It's a set of instructions to the Logi Server Engine to retrieve data and map it into the cells in the Excel template file.

The rest of this topic assumes that you've already created your Excel template file. Excel template files are managed within your Logi application as support files.

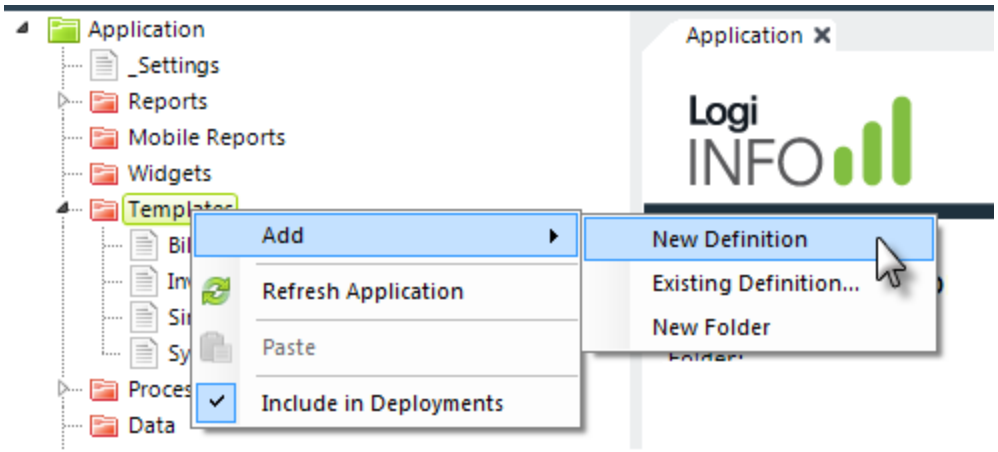



When you create your Excel template file either save it to your Logi application's _SupportFiles folder, or:

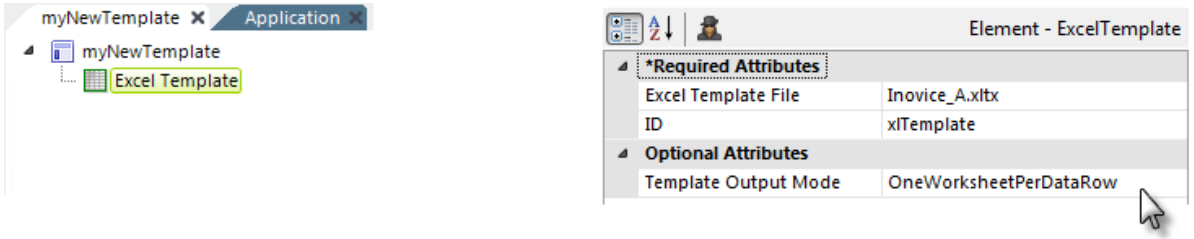
1. In Logi Studio's Application Panel, select and right-click the **Support Files** folder.
2. In the pop-up menus that appear, select the **Add** and **Existing File...** items.
3. Browse to your Excel template (.xltx, .xltm, or .xlt) file and select it. Click **OK** to add it to the project. The file will be *copied* to the **_SupportFiles** folder within your Logi application folder.

## Creating a Template Definition

The next step is to create a new Logi Template definition that maps the data into the Excel template file data ranges:




1. In the Application Panel, select and right-click the **Templates** folder.
2. In the popup menus that appear, select the **Add** and **New Definition** items.
3. A new definition named "newTemplate" will appear beneath the Templates folder, ready to be renamed, and will open in the Workspace Panel for editing. The actual file will be created in the `_Definitions/_Templates`folder in your application folder.




4. In the Workspace editor, add an **Excel Template** element to the definition, as shown above. Template definitions cannot utilize more than one Excel Template element.
5. Set the **Excel Template File** attribute to the full name of the .xltx, .xltm, or .xlt file you added as a Support File. You can choose .xltx and .xlt file names from the attribute value drop-down list but you will need to manually enter a .xltm file name.

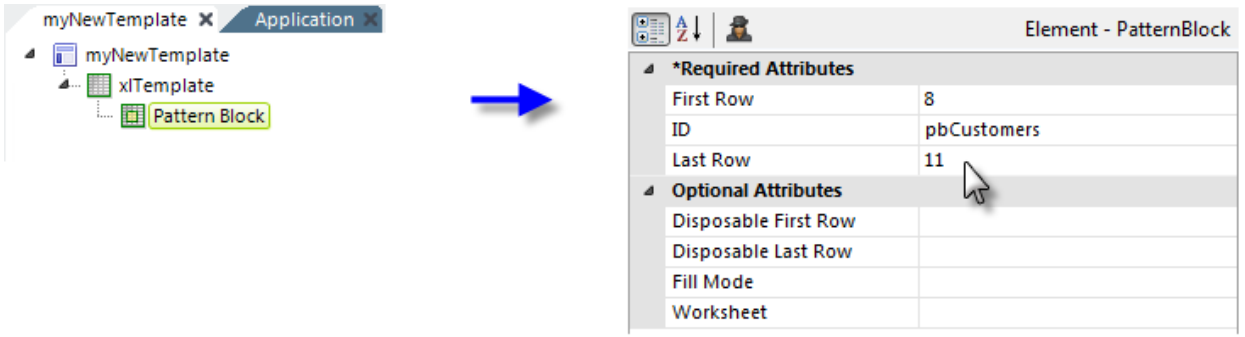
6. The **Template Output Mode** attribute is optional; its default value is *OneWorksheet*. If this attribute is set to *OneWorksheet*, all rows returned by the top-most data layers are processed in a single worksheet; each **Pattern Block** element (discussed below) can specify a different worksheet in the template. If the attribute is set to *OneWorksheetPerDataRow*, each row from the top-most data layers is treated as a *separate* worksheet. Developers will find this mode useful for invoice-style reports.

## Configuring the Data Ranges

The **Pattern Block** element gives developers the ability to define data ranges within the Excel template. A pattern block must correspond to one or more rows from the worksheet. Developers should keep the following guidelines in mind when creating template definitions:

- Two distinct pattern blocks cannot have overlapping ranges.
- Disposable rows must follow the corresponding pattern block.
- A sub-pattern block cannot extend beyond the boundaries of its parent block.
- A sub-pattern block's range must be less than its parent block range.

Now let's continue building the template definition:



1. Add a **Pattern Block** element to the template definition, beneath the Excel Template element, as shown above.
2. Set its **First Row** and **Last Row** attributes to values that correspond with row numbers from the Excel template.

   **Hint:** A Pattern Block *range* is equal to the difference of the last row number minus the first row number. Make the First Row and Last Row attributes equal to specify one row.

The **Disposable First Row** and **Disposable Last Row** attributes are optional. If dynamic charts and formulas are present in the Excel template, developers will need to set both attributes to values that correspond with row numbers from the worksheet. A *disposable range* consists of extra rows added to the template in order to test a formula or chart range.

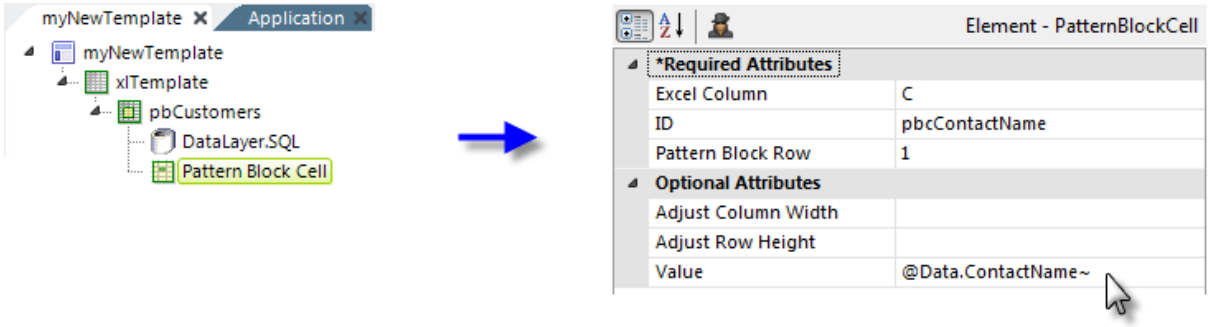
Test rows with dummy data cannot be part of the data range, because the Logi server engine extends the data range downward in certain scenarios. Without a method for declaring rows as "disposable", the Logi report server has no means of removing such rows from the final report. By specifying values for these attributes, developers can remove the unwanted, dummy rows from the final output.

If the Disposable First Row attribute is set but not the Disposable Last Row attribute, the Logi report server will use the Disposable First Row value for *both* attributes.

The **Worksheet** attribute is also optional and the default is the first worksheet. Each pattern block range resides on a single worksheet, however, you may specify any number of pattern block ranges using different worksheets.

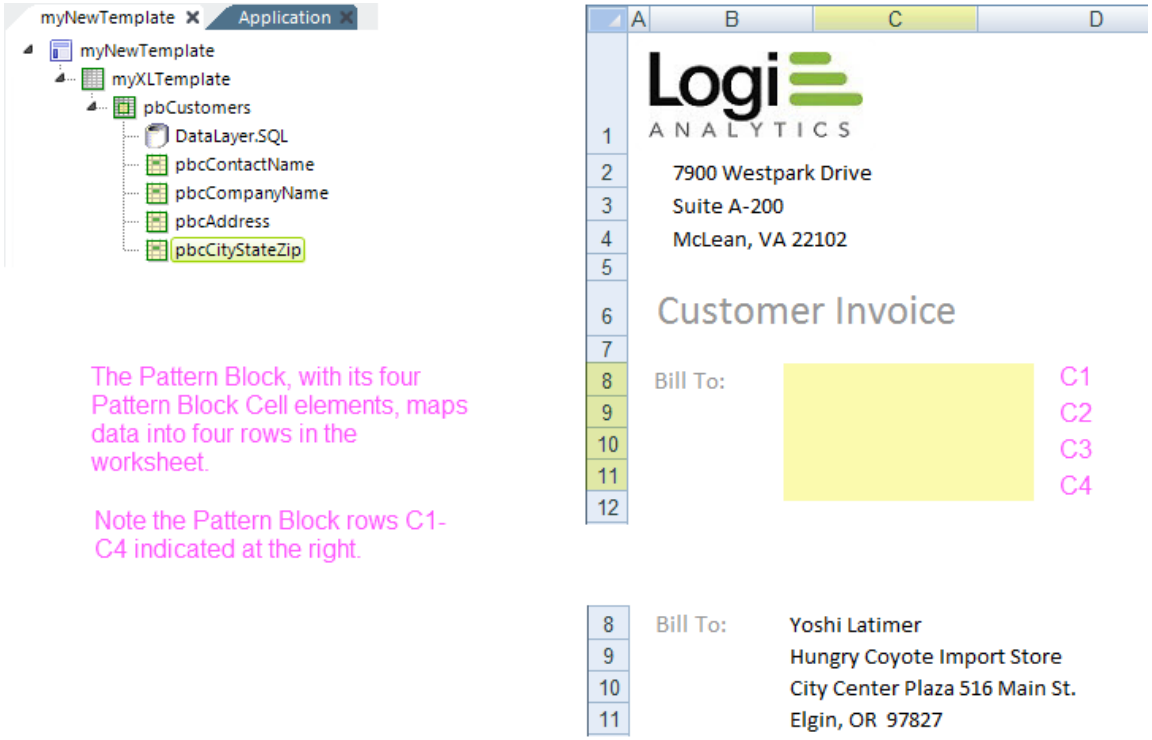
### The Pattern Block Cell Element

We've now specified the rows associated with the pattern block; next we need to specify the columns and map the data.




1. Beneath the Pattern Block element, add a **datalayer** element and configure its attributes appropriately, as shown above.
2. Beneath the Pattern Block element, add a **Pattern Block Cell** element.
3. Set its **Excel Column** attribute to a letter that corresponds with a column from the Excel template file.
4. Set the **Pattern Block Row** attribute to a row number within the pattern block range. This value is relative to the parent Pattern Block element, *not* to the worksheet row number.
5. The **Adjust Column Width** attribute is optional. Set this attribute to *True* to automatically adjust the column width to fit the data.
6. The **Value** attribute is optional and either tokens or static values can be used here.

Repeat the previous steps for each data item.



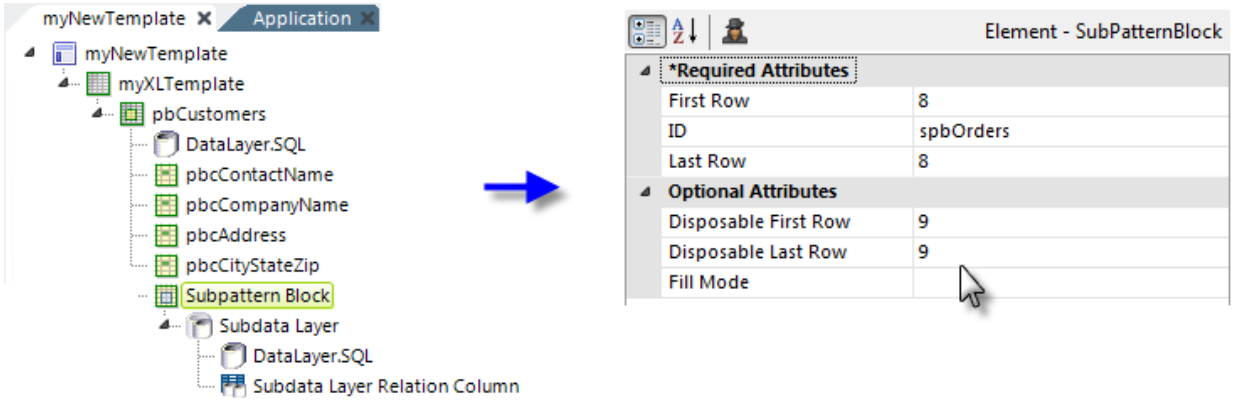

As shown above, left, we'll have three more Pattern Block Cell elements, one each for Company Name, Address, and City-State-Zip data. As shown above, right, the four Pattern Block Cell elements added in the sample definition above correspond to one cell within the block range. In this example, each cell in column *C* within the pattern block range will be filled with data. This is the technique that's used to map data to specific cells in the Excel template. Repeat as necessary to map all the data into the work-sheet.

# Working with Hierarchical Data

Logi Info supports hierarchical datasets within Excel templates. Hierarchical data presents parent-child relationships in the data and requires developers to specify **sub-data ranges** within an existing data range in the template definition.

Due to the dynamic nature of columns created by it, you *cannot* use a datalayer with a **Crosstab Filter** to export data through an Excel Template. If you'd like to create a Crosstab Table within an Excel Template, you may want to create a data sheet in the template file and then render an Excel Pivot table from the raw data. This will allow you to show crosstabbed data within your Excel file, using the Pivot Table from Excel.

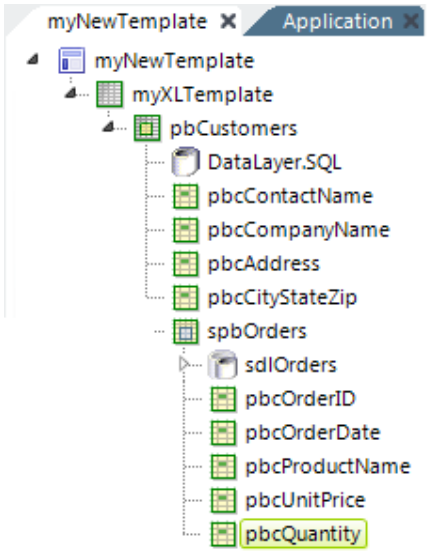
The **Subpattern Block** element is used to specify a repeatable sub-range within a parent Pattern Block or another Subpattern Block element. Developers must add a **Subdata Layer** element with at least one **Subdata Layer Relation Column** element to establish the parent-child relationships with the query and sub-query. Here's how: |



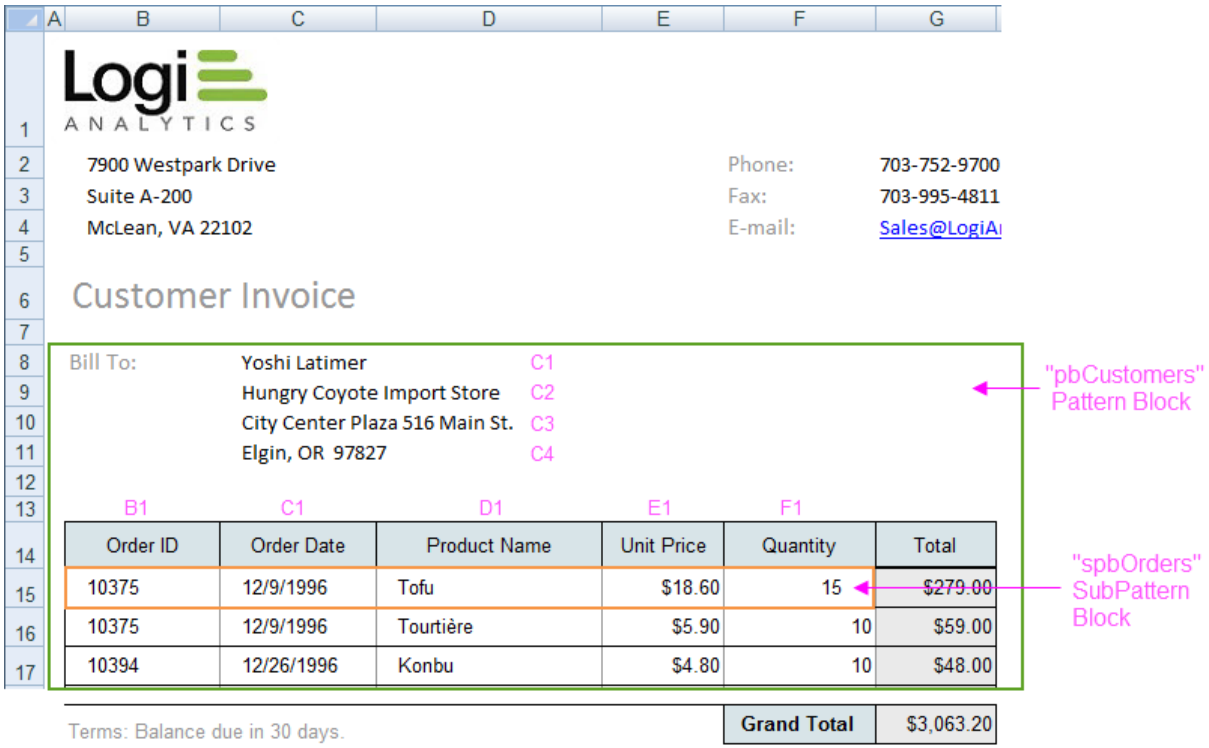

1. Beneath the Pattern Block element, add a **Subpattern Block** element, as shown above.
2. Set its **First Row** and **Last Row** attributes to row numbers from the parent pattern block range.

3. Leave its **Fill Mode** blank, which defaults to *Insert*. The Logi Server Engine will insert new rows into the Excel template for each row of data returned by the data layer. The number of rows inserted is equal to the total number of rows in the pattern block range.

   Subpattern Block attribute values are always relative to the parent block and must reside on the same worksheet.
4. Beneath it, add a **Subdata Layer** element and beneath it a **Subdata Layer Relation Column** element.
5. Set the Subdata Layer Relation Column's **Child Column** and **Parent Column** attributes to a column that exists in both data-sets, to establish the relationship.




6. Finally, add additional Pattern Block Cell elements, as shown above, to map the hierarchical child data into the Excel template where it's desired.




The Excel template shown above specifies the range of the *pbCustomers* Pattern Block as worksheet rows 8 - 17. The range of the *spbOrders* Subpattern Block is the last row of its parent block - in this case, its eighth row - through its 10th row. Any Pattern Block Cells that are children of the *spbOrders* Subpattern Block have row numbers relative to the parent sub-block. Since there is only one row in *spbOrders*, each Pattern Block Row attribute for any child Pattern Block Cell elements must have a value of **1**.

# Create an Excel Template

Form-based reporting provides a powerful method for making data available in popular formats.

The following topics provide guidance for creating a Microsoft Excel Template to use with form-based reporting and Logi Info applications:

- Creating the Template and Data Ranges
- Working with Charts and Formulas
- Preparing for Hierarchical Data

For basic information about templates and calling them from within Logi applications, see "Form-based Reporting" on page 224.

## About Excel Templates

Developers use Microsoft Excel to create Excel Templates, which are saved as one of these file types:

- **.xltx** - called an "Excel Template"
- **.xlt** - called an "Excel 97-2003 Template"

These templates become the "blueprint" for mapping the data from a datasource, retrieved using a Logi application, into Excel worksheets.

 These are the *only* file types supported as templates in a Logi application. Use of other file types may produce unpredictable results. In this scheme, one Excel template populates one Excel workbook. You can create Excel templates that use multiple work-sheets, if desired, to utilize 3-D formulas or to better organize template content. Before creating the template, developers should consider the following:

- Which areas in the template will be reserved for data?
- Will the template include any charts or formulas?
- Is the data itself hierarchical or flat?

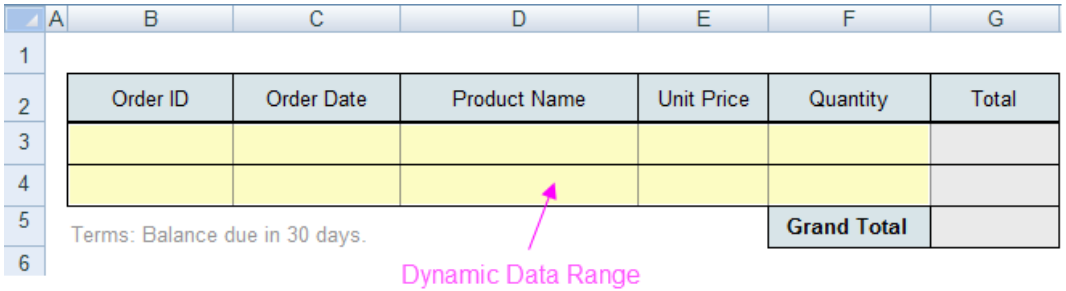
Once these questions have been answered, you're ready to begin creating a template.

# Creating the Template and Data Ranges

Microsoft Excel templates generally consist of both static content, called the *templatearea*, and landing areas for dynamic content, called the *data area* or *data range*. In order to preserve highly-formatted templates, when it generates output, the Logi server engine makes no modifications to the template area. The data range, however, is *filled-in* at runtime with the data retrieved by one or more data queries; developers specify the data style and format directly in the worksheet. If the template includes macros, they will run normally after the data is inserted. You specify how the template is to be filled-in by creating a special category of definition, the **Template definition**, in Logi Studio. The Logi server engine provides two template output modes, *OneWorksheet* and *OneWorksheetPerDataRow*.

## One Worksheet Mode

For flat data in **OneWorksheet** mode, data ranges are *dynamic* if more than one data row is returned by a query. For example, if ten rows are returned, the data range appears ten times on one worksheet.

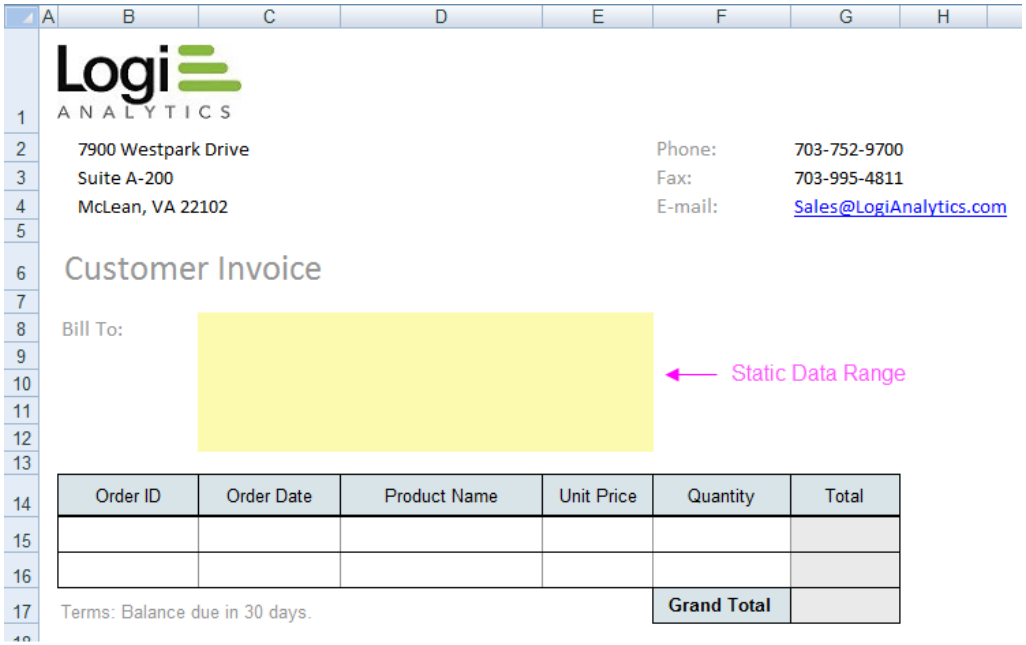



In the example above, the data range (the yellow cells) is meant for order information. To make the data range dynamic so that it uses a worksheet row for each data row, you must choose OneWorksheet mode. New *rows* are inserted into the worksheet for

dynamic data ranges as needed. Any information that lies beneath a dynamic data range is "pushed down" as new rows are inserted.

## One Worksheet Per Data Row Mode

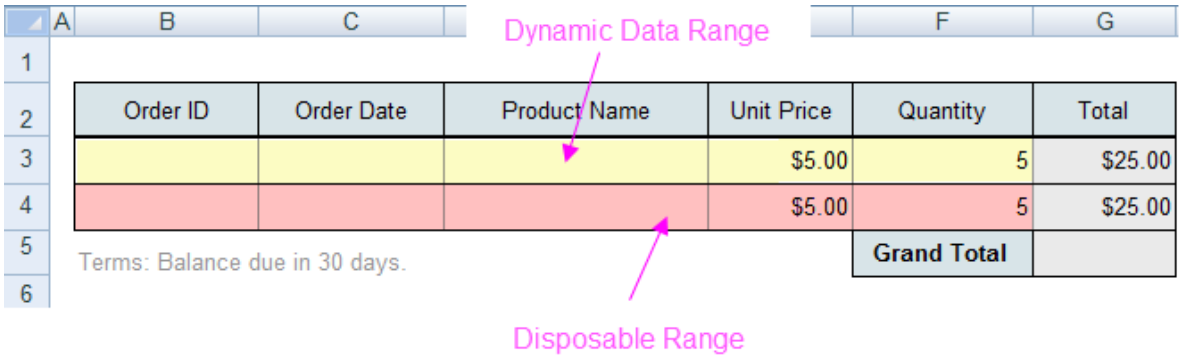
For flat data in **OneWorksheetPerDataRow** mode, data ranges are *static* if more than one data row is returned by a query. If ten rows are returned, then ten worksheets are produced, each containing the data from one data row.




In the example above, the data range shown is meant for a customer's billing address. In order to make the data range static and produce one worksheet for each customer returned by the query, you must choose OneWorksheetPerDataRow mode.

# Working with Charts and Formulas

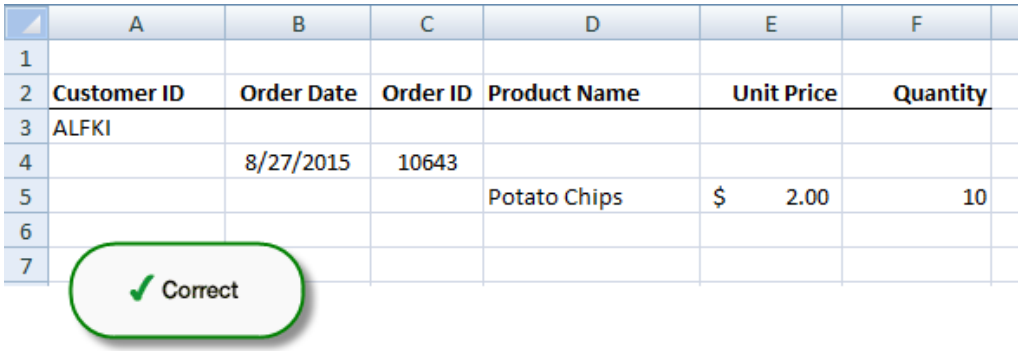
Charts and formulas in the template are automatically updated when the report is run. Microsoft Excel requires at least *two values* in a formula so that the formula is updated properly when new data rows are inserted into the worksheet. You're encouraged to use dummy values in order to test formulas, charts and other objects included in the report template.



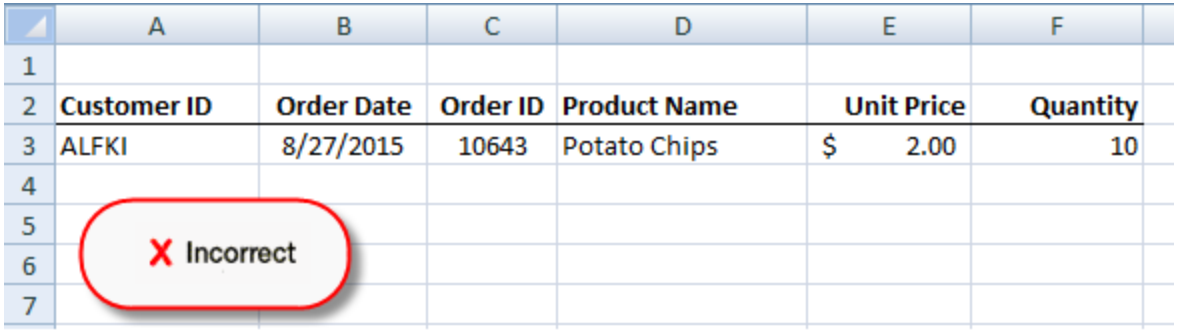

Extra rows can be added to the data range for the purposes of testing a formula or adding a chart. To prevent them from being included in the final report, developers can use special ranges called **disposable ranges**. Logi Studio provides a way to declare a range "disposable" and the Logi report server removes these ranges in the final report. See "Excel Template" on page 294for more details.

# Preparing for Hierarchical Data

Hierarchical data is *multi-layered* data that contains numerous parent-child relationships. For example, for each customer, an invoice-style report may contain customer contact information and data for multiple orders. If the data for several customers is returned from a query and each customer has placed one or more orders, then a *hierarchical dataset* exists. The only rule you need to follow when creating a template for hierarchical data is that each level in the hierarchy must have its own row.




For example, the report template shown above is configured properly for three levels of hierarchical data. If additional data at the third level is retrieved, data in the correct columns will be extended downward in additional rows.



However, the example shown above *is not* configured properly for multiple levels of data. If additional data at the second or third level is retrieved, data for all three levels will be extended downward in additional and meaningless rows.

# Exporting To Native Excel

Exporting reports into Microsoft Excel worksheets is a popular and useful method of applying additional analysis to data.

The following topics discuss techniques for exporting reports to Excel:



## About Exporting Reports to Excel

Exporting tabular data into a worksheet allows further computations and analysis to be performed. Microsoft's widely-used Excel spreadsheet program is ideal for this practice and Logi Studio provides appropriate elements for exporting data and reports.

Export features include the ability to export multiple Data Tables in the same operation (to the same or to separate worksheets), to apply style sheets to the exported data, to export report headers**,** images and charts or just raw table data, to maintain proper time and date formats, and to maintain Data Table internal cell linefeeds. The report title, grouping, summary, and filter information display on the export results, if your application has been configured for it.

Data Table cell background and text colors generated using *The Color Range Column* are exported to Excel worksheets.

**Group Header Row** and **Group Summary Row** elements now have a Show Modes attribute, which lets you hide them when exporting a report to Excel. For more information, see *Show Modes*. In addition, if using these two elements with their **Append Blank Rows** or **Prepend Blank Rows** attributes set to a number, blank rows will now *not* be appended/prepended when the report is exported to Excel.

In a Data Table report, when Append Blank Rows to the Group Summary Row is set, the width of the newly added blank rows will be the same as the largest width in the existing Table Row.

Developers interested in inserting data into pre-defined worksheet forms should see "Create an Excel Template" on page 304.

Excel imposes column and row limits per worksheet (the exact numbers vary based on the version of Excel). Exported data that exceeds these limits will overflow onto the next worksheet.

You can specify that the exported output be in Excel 2007 (.xlsx) format or Excel 2003 and earlier (.xls) format.

### Use "Export Native Excel"

Use the **Action.Export Native Excel** element, **Procedure.Export Native Excel** element, or **Rapid Excel Export** element for all of your Excel exports. Earlier versions of Logi Info included a now-deprecated Action.Export Word Or Excel element that has only been retained for backward compatibility.

For large data sets, we recommend using the rapid Excel export option. Below is a formal list of the differences between native Excel and rapid Excel exports:

| Native Excel Export | Rapid Excel Export |
|---|---|
| Can export a report of one or more specified | Can only export one specified Data Table to a file |

| Native Excel Export | Rapid Excel Export |
|---|---|
| Tables/Crosstabs to a file | |
| Can support Data type attribute in "Excel Column Format" element | Can support Data type attribute in "Excel Column Format" element |
| Can export a Crosstab | Cannot export a Crosstab |
| Can export a Data Table created with the "Auto Columns" element | Cannot export a Data Table with the "Auto Columns" element |
| Can use the "Excel Paper Size" attribute | Cannot use the "Excel Paper Size" attribute |
| Can use the "Show Gridlines" attribute | Cannot use the "Show Gridlines" attribute |
| Can use the "Caption" attribute | Cannot use the "Caption" attribute |
| Can use group or count info (like "Header Row" or "Summary Row" elements) | Cannot use group or count info (like "Header Row" or "Summary Row" elements) |
| Can support Style info (like background color or text color) | Cannot support Style info (like background color or text color) |
| Can support the "More Info Row" element | Cannot support the "More Info Row" element |
| Can export report title and filter information | Can export report title and filter information |
| Can remove empty rows if there is no detail data under a particular group | Cannot remove empty rows if there is no detail data under a particular group |
| Can support token resolution | Can support token resolution |

| Native Excel Export | Rapid Excel Export |
|---|---|
| Can export group or summary information | Can export group or summary information |
| Can support Yes/No, On/Off, and True/False formatted columns | v23.1 Can support Yes/No, On/Off, and True/False formatted columns |

For more information on the rapid export feature, see "Rapid Excel Export" on page 335.

### Manual vs Process Task Export

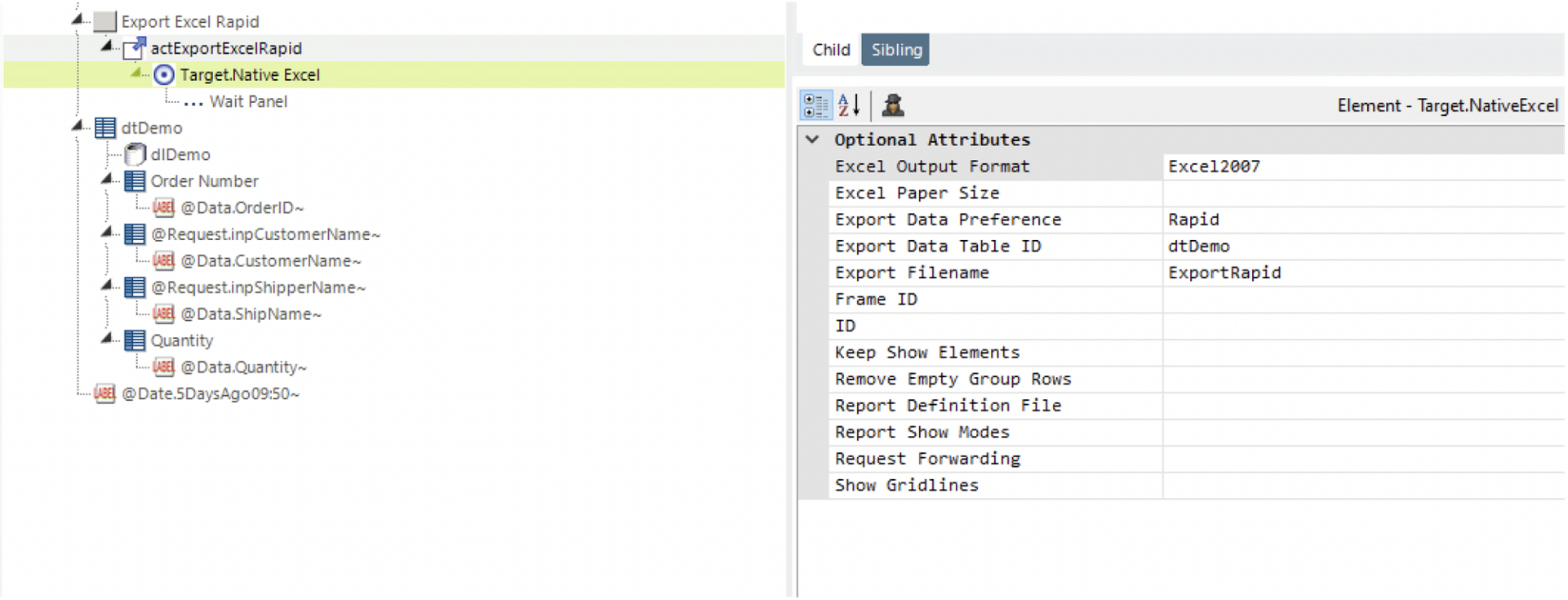
You can give your users the ability to export a report in two ways:

- Manual - An exported report is made available for download and opened using an Excel browser plug-in or Excel itself, or
- Process Task - An exported report is saved, as an .xlsx or .xls file, to the web server, as part of process task execution. This also allows you to distribute exported Excel files as email attachments.

Manual exports are configured within *report definitions* and process exports are configured within *process definitions*.

# Excel - Exporting a Report Manually

Here's an example of how to create a report, with a link, that exports itself manually:




1. In your report definition, add an **Action.Export Native Excel** element beneath a Label, Image, Button, or Chart element, as shown above.
2. Add the required **Target.Native Excel** element.
3. If the report to export *is* the current report, you need do nothing more. Just save your definition and browse your report. It's that simple.
4. If the report to export *is not* the current report, specify that report by setting the Target.Native Excel element's **Report**

**Definition File** attribute.

5. All Target.Native Excel element attributes are *optional*. The default settings will export the current report in its entirety.

What Happens: The report is exported to a temporary file created in your application's rdDownload folder on the web server. The temp file is then returned to the client and opened automatically in the Excel plug-in within your browser (you maybe prompted to Open or Save the Excel file) or in Excel itself. Temporary files on the server are cleaned up automatically over time.

The following table describes the attributes of the **Target.Native Excel** element, all of which are optional:

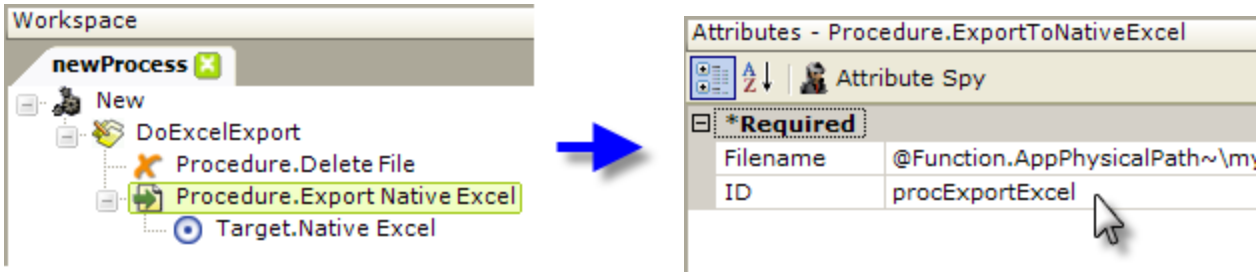
| Attribute | Description |
|---|---|
| Excel Output Format | Specifies the desired output format of the Excel file created. Set to *Excel2007* to produce an .xlsx file, which supports up to 16K columns and 1M rows per worksheet. Set to *Excel2003* to create an .xls file, with 255 columns and 64K rows per worksheet.<br><br>.xlsx files can be opened with Office 2003, but they will lose any data beyond the 255 columns and 64K rows that are supported.<br>Default: *Excel2003 (.xls)* |
| Excel Paper Size | Specifies the default paper size of the Excel workbook, used when it's being printed. Options include *A4*, *Letter*, *Legal*, etc. |
| Export Data Preference | Specifies the export preference. Options include Empty (default), Rapid, and Formatted. For more information, see "Rapid Excel Export" on page 335. |
| Export Data Table ID | Specifies the ID of a particular Data Table to be exported, and causes a *Data Table only* export. Only the Data Table caption, columns (with header text), and data rows will be output. No report headers, images, report |

| Attribute | Description |
|---|---|
| | footers, etc. will be output. If this attribute is left blank, the full report with all report headers, report footers, Data Tables, images, etc. will be output.<br><br>To export the data for an Analysis Grid element, set this value to *dtAnalysisGrid*. This is the ID of the Analysis Grid's internal Data Table. |
| Export File Name | Specifies a custom name for the export files. When left blank, file names consist of a random GUID number and extension. |
| Frame ID | Specifies the frame for the target HTML page. Leave blank for the current browser window, or enter NewWindow to open a new browser window. You can also specify an existing frame ID to re-use the same window for each request. |
| Keep Show Elements | Specifies whether page items that have been shown or hidden by the user, using Action.Show Element, remain that way when the report is exported. Set to *True* so that those elements that were originally hidden but then shown by the user get included in the export. Alternatively, any elements hidden by the user will not be exported. Default: *False.* |
| Remove Empty Group Rows | Specifies whether the Excel output displays empty rows when there is no detail data under a particular group. Default: Empty, or *False*.<br><br>This attribute only supports Native Excel and Formatted exports. |
| Report Definition File | Specifies the name of a the report definition file to be exported. Default: *the current report.* |

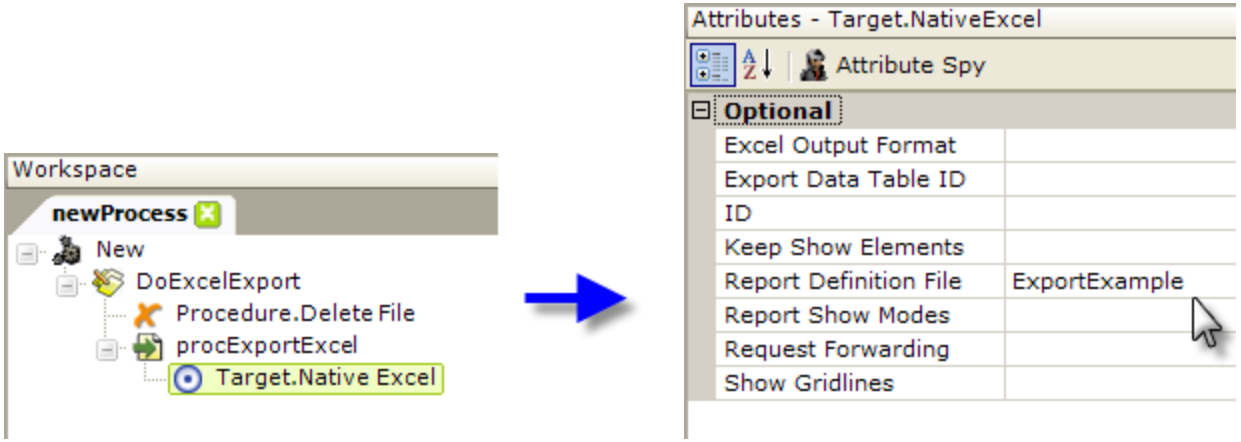
| Attribute | Description |
|---|---|
| Report Show Modes | Specifies the Show Modes strings to be passed to the report when it's rendered for export; makes it possible to hide elements in reports. See "Excel - Hiding Elements When Exporting" on page 326. |
| Request Forwarding | Specifies whether to pass all request parameters to the report when it's rendered for export. Default: *False.* |
| Show Gridlines | Specifies grid line behavior in the exported report. In Excel, if two consecutive rows or columns are set to the same style (font, background, foreground color, etc.) the gridlines between them will disappear and this may not provide the desired appearance. Set this attribute to *True* to show the gridlines in all cases. Default: *False.* |

# Excel - Exporting a Report using a Process Task

The following example, for Logi Info only, shows how to create a **process task** that exports data automatically:




1. In your Process definition, add a **Procedure.Export Native Excel** element to your Task, as shown above.
2. In the element's **Filename** attribute, specify the output path and filename, on the web server, for the exported report. The filename should include the ".xlsx" or ".xls" file extension. For example, this value uses a token to export the report to a folder called *myExports* within your application folder: `@Function.AppPhysicalPath~\myExports\myfile.xlsx`
3. If the folder you're exporting to is not within your Logi app root folder, ensure that the account or Application Pool used to run your application has been granted full control file access permissions on the folder.
4. In the example above, note the user of a **Procedure.Delete File** element before the Procedure.Export Native Excel element. The export *will not overwrite* an existing file, so this element is used to delete any older version before the export occurs; no error will occur if there's no existing file to delete. Set its **Filename** attribute value to match the Export Native Excel element's Export File Name attribute value.




4. Add the required **Target.Native Excel** element.
5. In the element's **Report Definition File** attribute, specify the report to be exported.

In a Process definition, the *CurrentReport* option that appears in the suggested values for this attribute cannot be used. You *must* specify the actual name of the report.

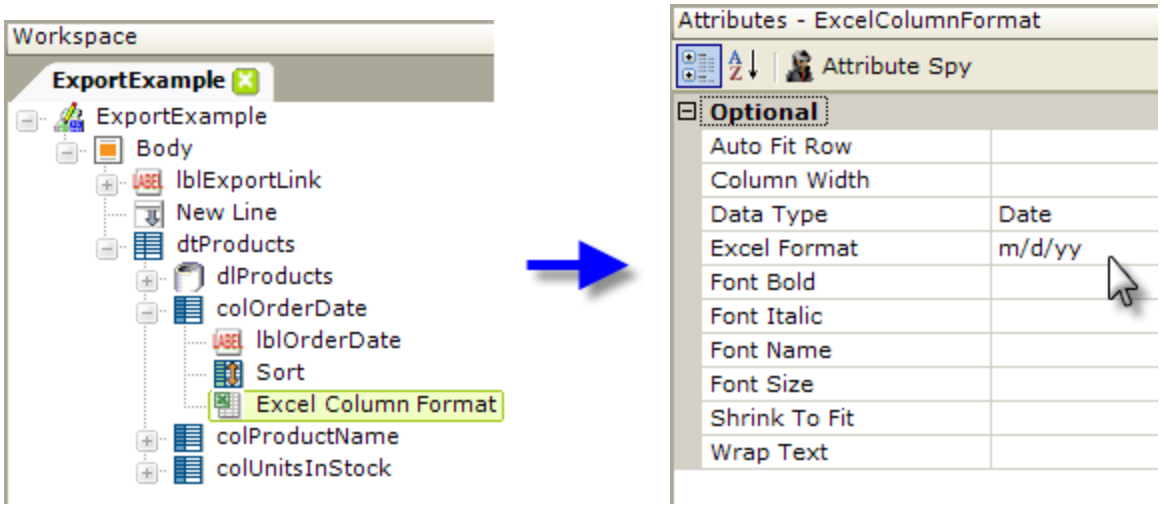
What Happens: When your task runs, the specified report will be exported to a temporary file in your application folder's rdDownload folder on the web server; when ready the temp file is then copied to your output file. Temporary files on the server are cleaned up automatically over time.

# Excel - Specifying Excel Column Formatting

Data exported to Excel generally arrives in the worksheet as text-type data, which may not be desirable. For example, Excel will store exported numbers as text, then "helpfully" flag their cells with a confusing little green triangle. And, of course, these values (as text) cannot then be used in calculations in the worksheet.

You can turn off this behavior in Excel using options in File → Options → Formulas → Error Checking or Error Checking Rules. Or, you can click the green triangle and select "Convert to Number". These and other remedies such as writing scripts in Excel to do the conversions automatically are not very convenient and may not be feasible.

However, the easiest way to handle the formatting of exported data on a column-by-column basis is to use **Excel Column Format** elements. These elements provide you with the opportunity to separately optimize the appearance of report and exported output and ensure that the exported data is seen as the correct data type.



The **Excel Column Format** element is added as a child of each Data Table Column element that you want to format, as shown above, and is configured to format the data exported to the worksheet.

The following table describes the attributes of the Excel Column Format element:

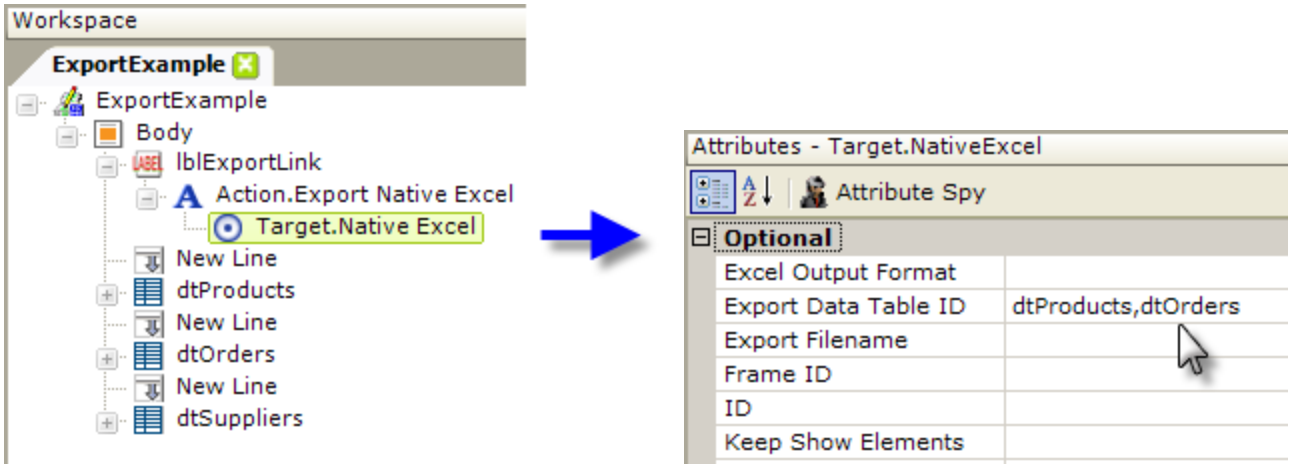
| Attribute | Description |
|---|---|
| Auto Fit Row | When set to *True* for any column, the height of the Excel rows is automatically adjusted to fit the data. |
| Column Width | Specifies the width of the Excel column, in units representing the average character width for the current font. For example, a value of 10 would indicate a column that is usually able to contain 10 characters. The column will always be wide enough to display the value of the Column Header attribute of the Data Table Column element. |
| Data Type | Specifies a data type for values in the column; also affects justification of displayed data. |
| Excel Format | Applies formatting to the data displayed in Excel. Works with the Data Type attribute, so setting the data type correctly is required for correct formatting.<br><br>In addition to the standard Logi format options you can also use custom Excel formatting strings in this attribute. For example, to right-align whole numbers and currency, use this string:<br><br>`_(* $#,##0.00_);_(* $(#,##0.00);_(* $"-"??_);_(@_)`<br><br>Refer to this document for information about custom number formats in Excel 2007+. |

| Attribute | Description |
|---|---|
| Font Bold<br>Font Italic | When set to *True*, displays the data in Bold or Italic fonts, respectively. |
| Font Name | Specifies the name of the font to be used for this data. Font options are presented in a drop-down list. |
| Font Size | Specifies the size, in points, of the font used for this data. |
| Shrink To Fit | When set to *True*, automatically reduces the font size of a value that is too long to be displayed in a single cell to a smaller size in order to make it fit. |
| Wrap Text | When set to *True*, causes text that is too long to be displayed in a single cell to be wrapped into additional rows. |

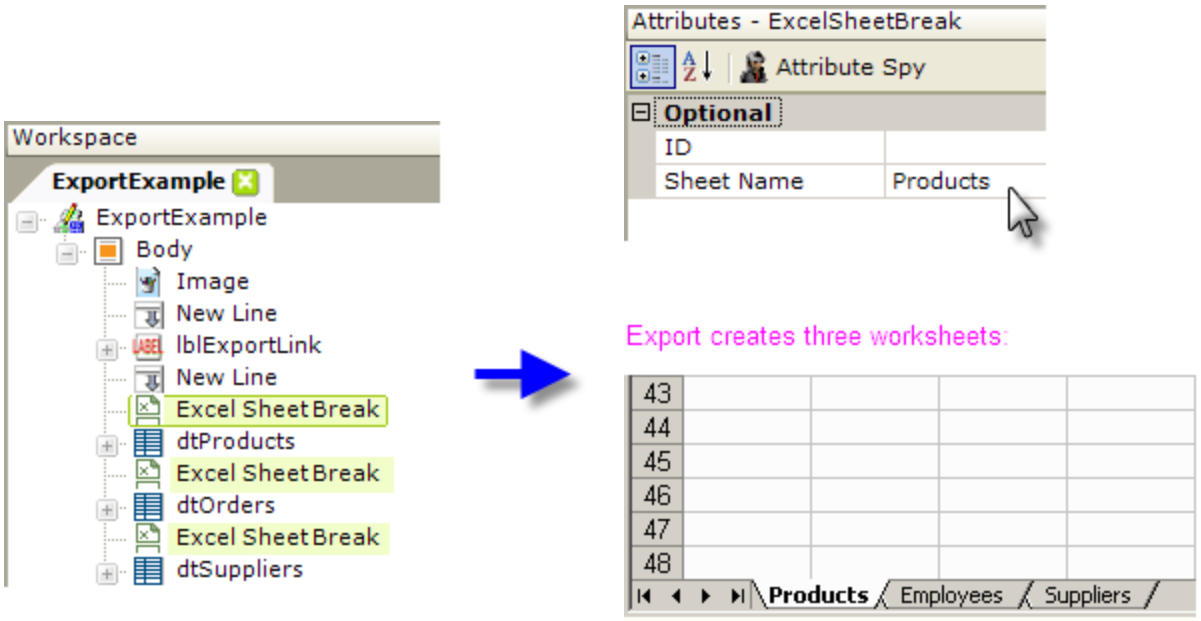
 Excel Column Format elements have no effect when used with a table within a **SubReport** and they are not available for use with data displayed in **More Info Rows**.

# Excel - Exporting Multiple Tables

If a report contains multiple tables, they can be exported into the *same* or *different* worksheets in a single Excel file.



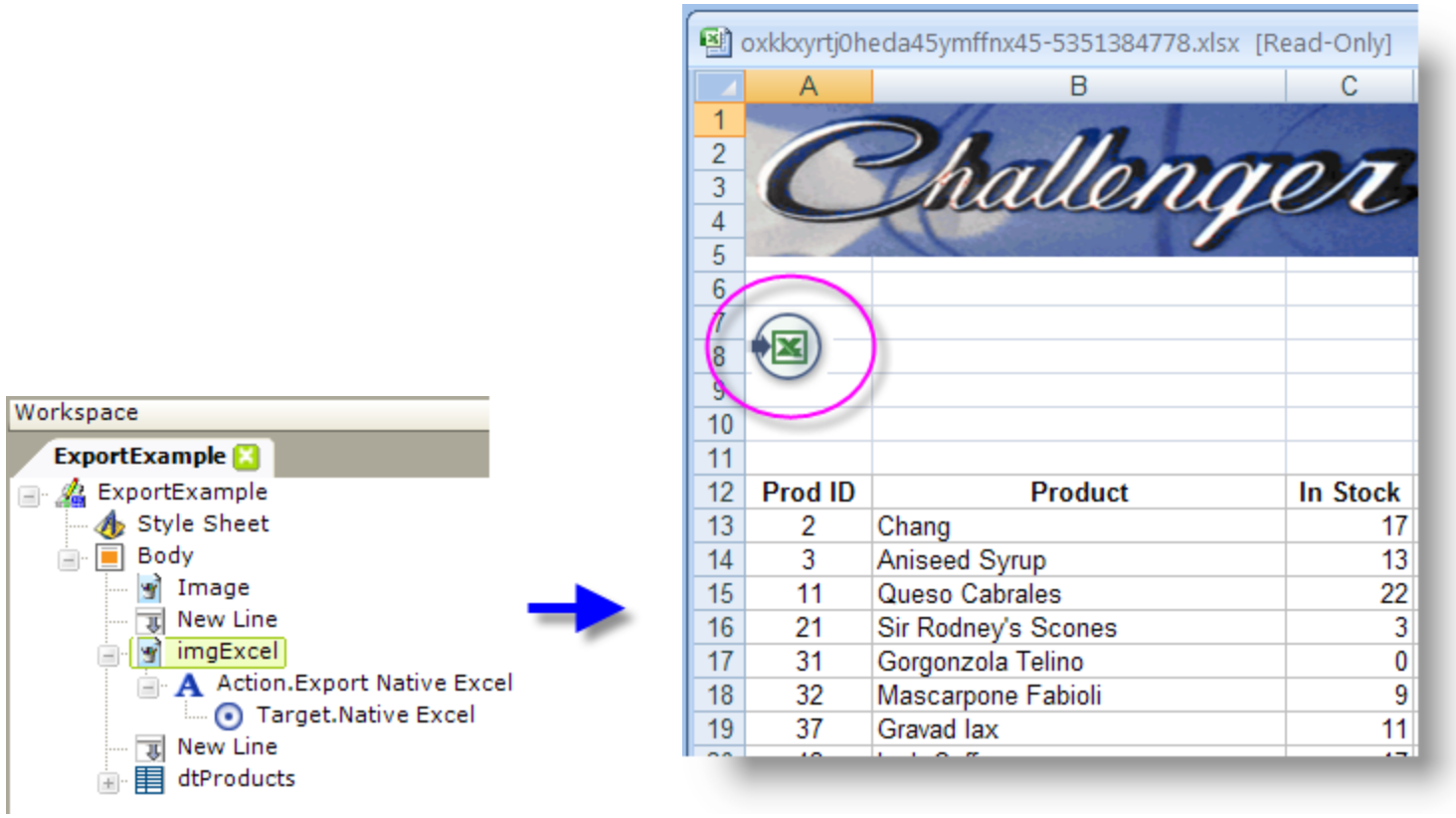

By default, all tables in a report definition will be exported into the same Excel worksheet during an export operation. However, you can instead include tables selectively by entering them as a comma-separated list in the **Export Data Table ID** attribute of the Target.Native Excel element. For example, the configuration shown above will only export two of the report's three tables to Excel.



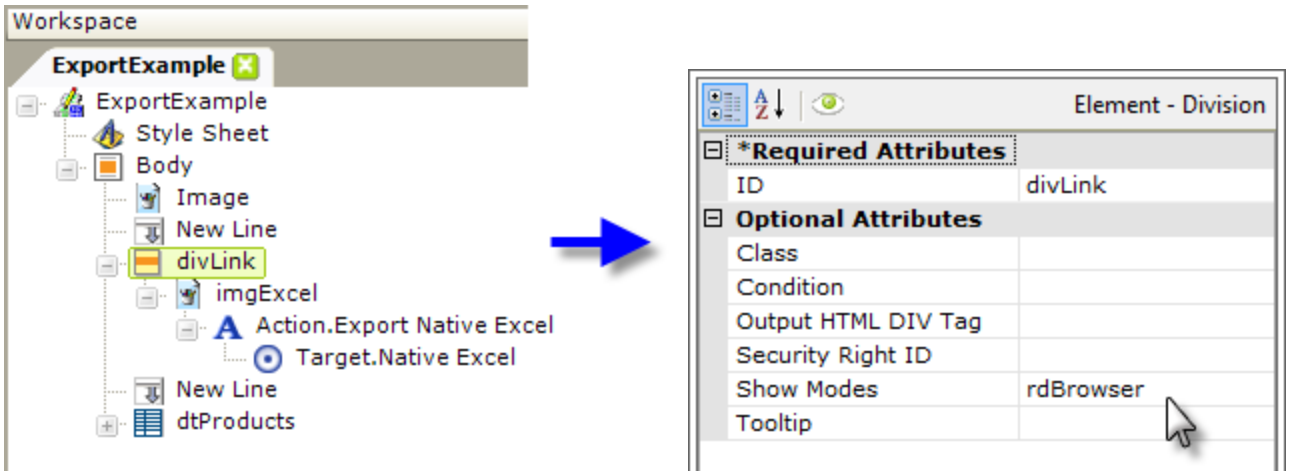

By adding **Excel Sheet Break** elements in the Body of your report definition, as shown above, you can cause multiple tables to be sent to separate worksheets. The order of the tables in the report definition dictates the order of the worksheets containing them in the export.

# Excel - Hiding Elements When Exporting

When exporting complete reports to Excel, developers commonly want to hide some of the elements in the report page, like the Export button or link. This can be done very easily using **Show Modes**.

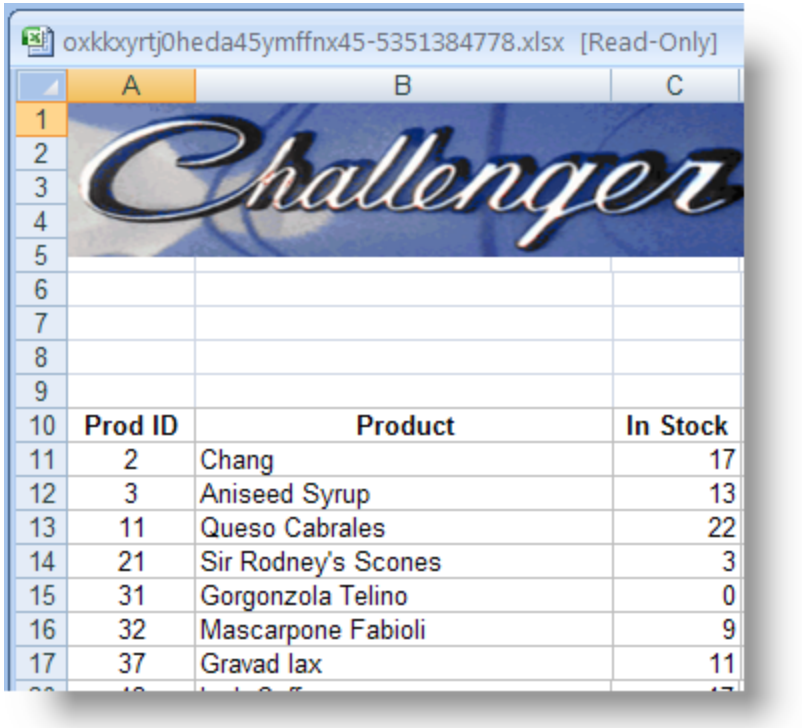



Consider the example shown above. The definition, on the left, includes an image that can be clicked to export the report to Excel. However, when it's exported, shown on the right, the Excel Export icon appears in the worksheet itself. We don't want that icon to appear in the worksheet.




To "hide" the icon image in the export, simply place it beneath a Division element. This element supports Show Modes, which the Image element does not.

Then set the Division element's Show Modes attribute value to the built-in Show Mode value *rdBrowser*, as shown above. Elements with this Show Mode value will only be visible in the browser.

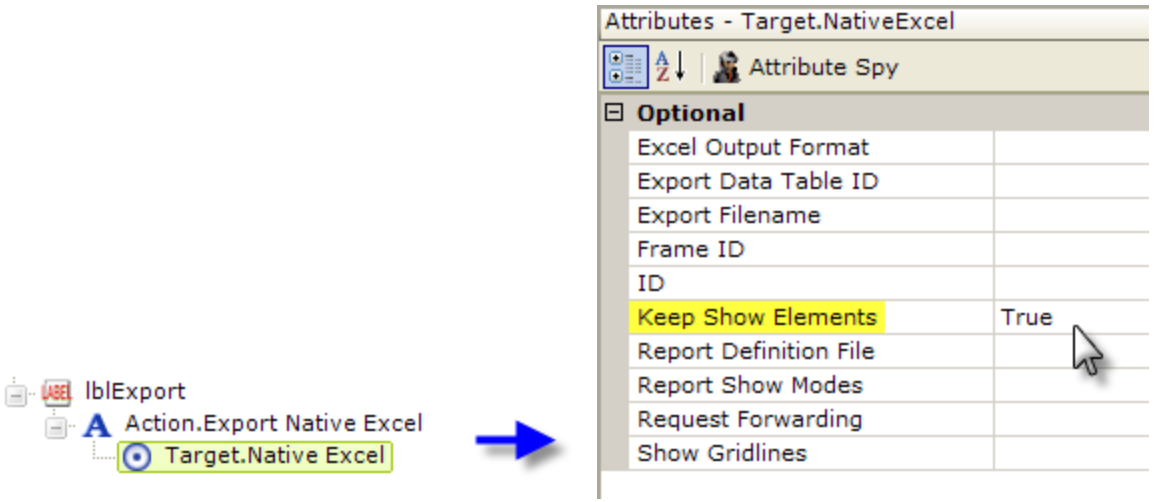



| Prod ID | Product | In Stock |
|---|---|---|
| 2 | Chang | 17 |
| 3 | Aniseed Syrup | 13 |
| 11 | Queso Cabrales | 22 |
| 21 | Sir Rodney's Scones | 3 |
| 31 | Gorgonzola Telino | 0 |
| 32 | Mascarpone Fabioli | 9 |
| 37 | Gravad lax | 11 |

When the report is run and exported, as shown above, the document no longer includes the Export Excel icon.

# Excel - Exporting More Info Rows

If your report contains a Data Table and you're using the **More Info Row** element to show/hide additional detail data within the table, you may want the detail rows (when visible) to be exported along with the main table data.




To enable this, as shown above, set your Target.Native Excel element's **Keep Show Elements** attribute to *True*, and leave the **Export Data Table ID** setting **blank**. If you place a value here, you risk not displaying all of the rows in the spreadsheet.

Keep Show Elements works with Action.Show Elements (used to show/hide the More Info Row) so that items that have been shown or hidden by the user remain that way when the report is re-displayed or exported.

# Excel - Cascading Style Sheet Support

When an Excel export is being generated, the export engine processes any style sheet information along with the report data. Style classes assigned ithin the report are used; you do not need to do anything additional to specify the style sheet for the export.

Issues related to CSS-based styling of exported data were resolved in Info v12.1.188 and v12.5 - check the Logi Info Release Notes for more information. Logi Info is also restrictive with CSS syntax; in order for the CSS code to be picked up by the Excel export engine, there must be a white space between the CSS selector and the curly bracket. For example:

`.yellowBackground{ /*` works for browsers but not for Excel exports `*/background-color: Yellow;`

`}`

In addition, *Excel itself does not fully support all CSS styles and colors*, so you may need to experiment to find those that are supported.

If desired, you can use two different style sheets in your report definition, one for the report and one for the Excel export:

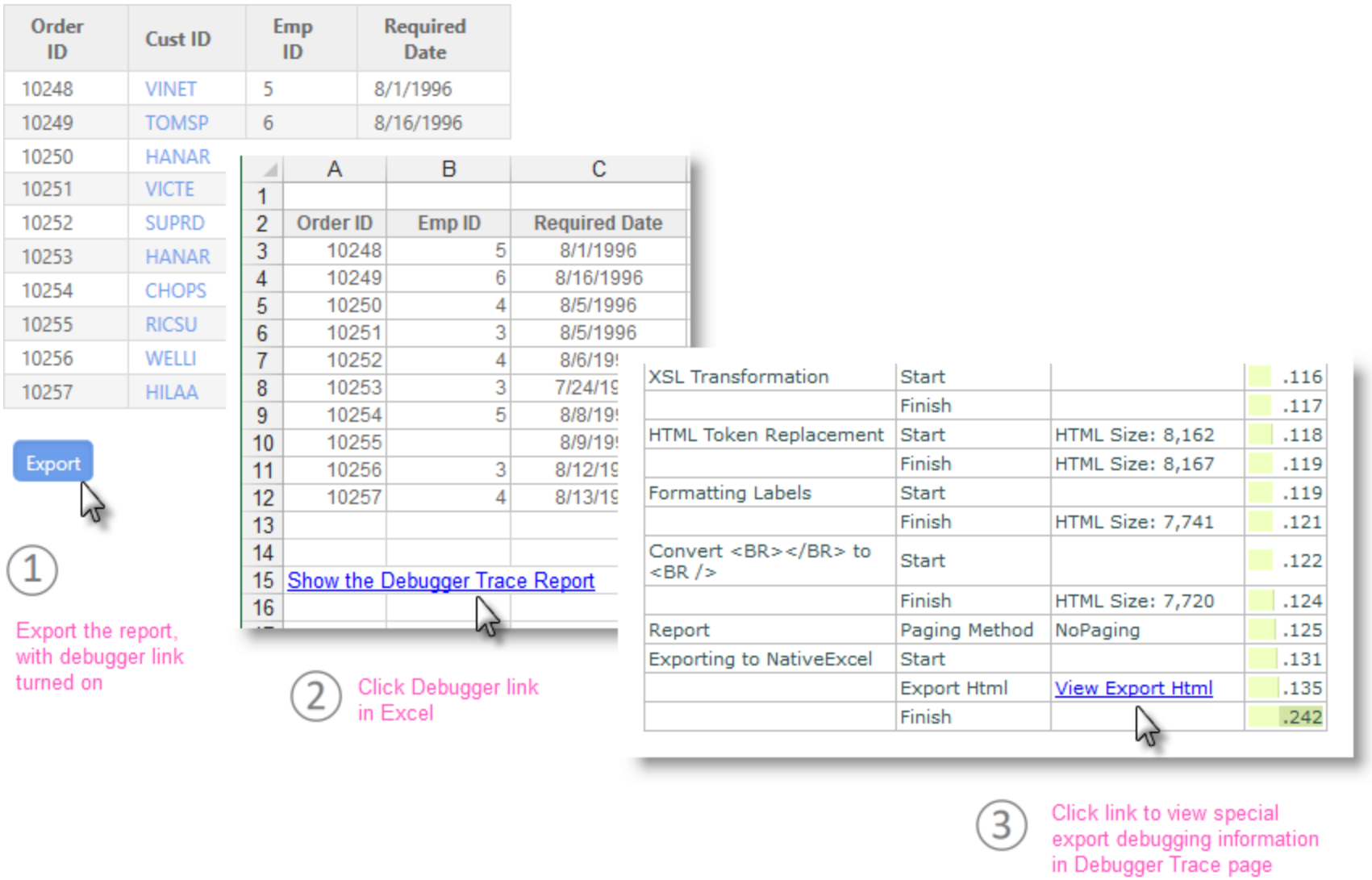
1. Select the root element of your report definition and set its **Default Show Modes** attribute value to an arbitrary string, such as "Normal".
2. Add two Style Sheet elements to your report definition.
3. Assign one Style Sheet element's **Show Modes** attribute a value that matches the string used in Step 1, "Normal". This style sheet will be used for the regular HTML report.
4. Assign the other Style Sheet element's Show Modes attribute a value that matches the Target.Native Excel element's Report Show Modes attribute value (both should be an arbitrary string that *does not match* the string from Step 1, such as "Export".

Now when the report runs in a browser, the first style sheet will be applied; when it's exported to Excel, the second style sheet will be applied.

What if you want to combine the effects from this topic and "Excel - Hiding Elements When Exporting" on page 326, i.e. hide some elements and use different spreadsheets? Just remember that the Show Modes attribute can accept several values, separated by a comma. To combine the examples from these two sections, ensure that the root element's Default Show Modes attribute includes both Show Modes values: "Normal,NoExport" (do not enter the double quotes).

# Excel - Debugging Exports

Developers can debug their exports. When the debugger has been set to *Debugger Links* and the report is run, and then exported, a debug link will be included at the bottom of the exported report:




This mechanism allows you to review the export process in more detail and can help youdiagnose any problems that may arise.

# Excel - Export Considerations

The following apply when exporting to Excel:

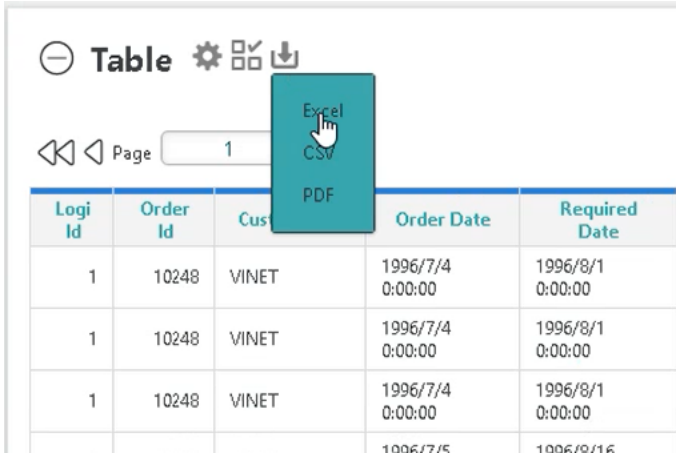
- Chart color "transparency" may not be available in exports to Excel.
- HTML files embedded in a report may not be exported correctly; success is dependent on the nature of the external HTML file, its adherence to proper XHTML syntax, and other factors.
- Summary Rows, if present in a Data Table, will be exported to Excel but the summarized data will appear in the worksheet as a text value, not a number.
- Bulleted lists are not supported (using the Bullet element)
- The Rectangle element cannot be exported.
- When a Multi-Column List is exported to Excel, its data is exported into a single Excel column, not multiple columns.

# Rapid Excel Export

The Rapid Excel Export option provides a built-in capability for both the Data Table and Analysis Grid, running off the datalayer. This feature is available for Consumer and Self-Service users who export tabular data in Excel format for further analysis outside of the context of the Info application. The Rapid Export option greatly improves the performance of exporting large data, as compared to the Native Excel export option. The Rapid Export feature honors basic formatting (bold, font, italics, etc), but it does not apply custom styling or group aggregation. To review a detailed list of the differences between the native Excel and rapid Excel export, see "Exporting To Native Excel" on page 311.

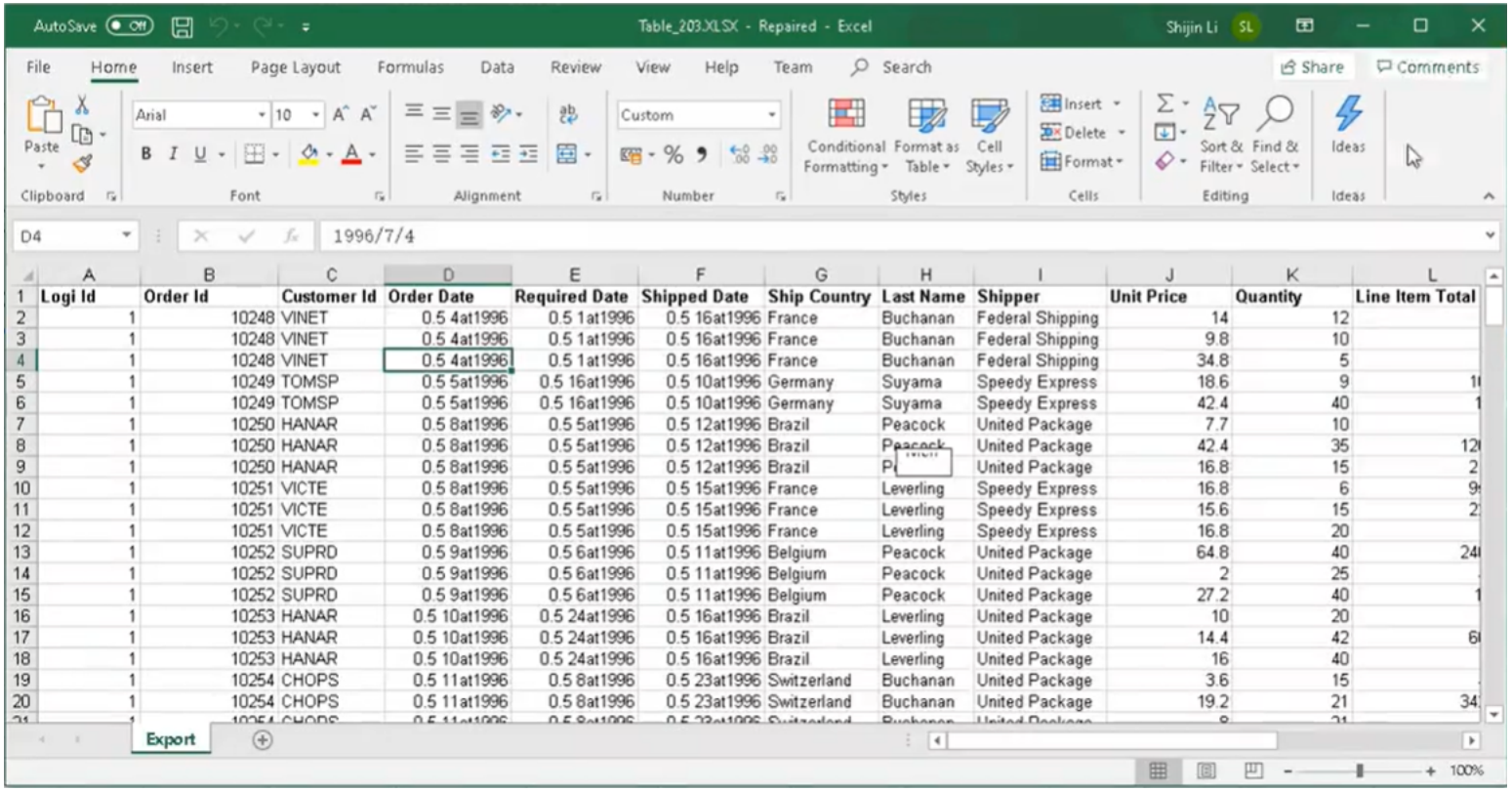
To export your data using the Rapid Excel Export, access your Data Table in the Analysis Grid and select the **Export** icon.

Then, select **Excel** from the drop-down list:




Your download will appear in the bottom of the web browser, or may open automatically, depending on your machine's configuration.

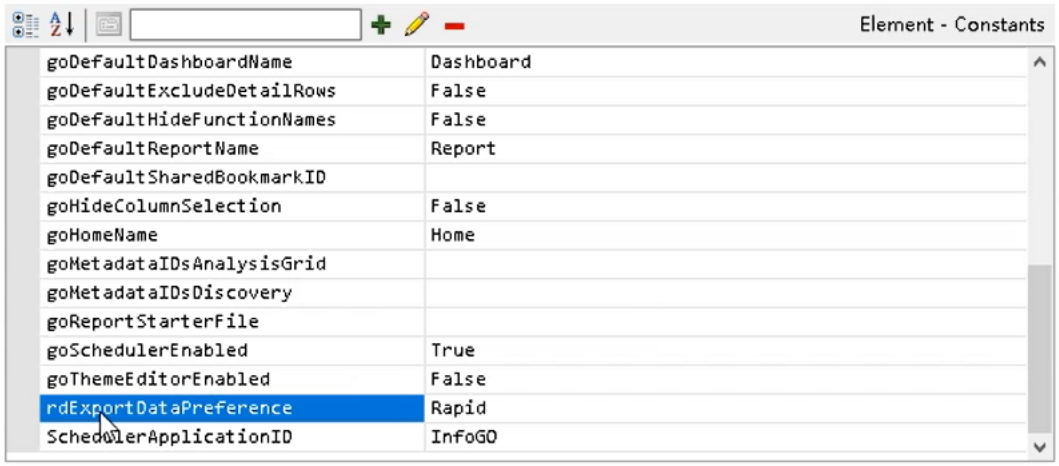
The Data Table displays in Excel:




| | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Logi Id | Order Id | Customer Id | Order Date | Required Date | Shipped Date | Ship Country | Last Name | Shipper | Unit Price | Quantity | Line Item Total |
| 2 | 1 | 10248 | VINET | 0.5 4at1996 | 0.5 1at1996 | 0.5 16at1996 | France | Buchanan | Federal Shipping | 14 | 12 | |
| 3 | 1 | 10248 | VINET | 0.5 4at1996 | 0.5 1at1996 | 0.5 16at1996 | France | Buchanan | Federal Shipping | 9.8 | 10 | |
| 4 | 1 | 10248 | VINET | 0.5 4at1996 | 0.5 1at1996 | 0.5 16at1996 | France | Buchanan | Federal Shipping | 34.8 | 5 | |
| 5 | 1 | 10249 | TOMSP | 0.5 5at1996 | 0.5 16at1996 | 0.5 10at1996 | Germany | Suyama | Speedy Express | 18.6 | 9 | 1 |
| 6 | 1 | 10249 | TOMSP | 0.5 5at1996 | 0.5 16at1996 | 0.5 10at1996 | Germany | Suyama | Speedy Express | 42.4 | 40 | 1 |
| 7 | 1 | 10250 | HANAR | 0.5 8at1996 | 0.5 5at1996 | 0.5 12at1996 | Brazil | Peacock | United Package | 7.7 | 10 | |
| 8 | 1 | 10250 | HANAR | 0.5 8at1996 | 0.5 5at1996 | 0.5 12at1996 | Brazil | Peacock | United Package | 42.4 | 35 | 12 |
| 9 | 1 | 10250 | HANAR | 0.5 8at1996 | 0.5 5at1996 | 0.5 12at1996 | Brazil | P | United Package | 16.8 | 15 | 2 |
| 10 | 1 | 10251 | VICTE | 0.5 8at1996 | 0.5 5at1996 | 0.5 15at1996 | France | Leverling | Speedy Express | 16.8 | 6 | 9 |
| 11 | 1 | 10251 | VICTE | 0.5 8at1996 | 0.5 5at1996 | 0.5 15at1996 | France | Leverling | Speedy Express | 15.6 | 15 | 2 |
| 12 | 1 | 10251 | VICTE | 0.5 8at1996 | 0.5 5at1996 | 0.5 15at1996 | France | Leverling | Speedy Express | 16.8 | 20 | |
| 13 | 1 | 10252 | SUPRD | 0.5 9at1996 | 0.5 6at1996 | 0.5 11at1996 | Belgium | Peacock | United Package | 64.8 | 40 | 24 |
| 14 | 1 | 10252 | SUPRD | 0.5 9at1996 | 0.5 6at1996 | 0.5 11at1996 | Belgium | Peacock | United Package | 2 | 25 | |
| 15 | 1 | 10252 | SUPRD | 0.5 9at1996 | 0.5 6at1996 | 0.5 11at1996 | Belgium | Peacock | United Package | 27.2 | 40 | 1 |
| 16 | 1 | 10253 | HANAR | 0.5 10at1996 | 0.5 24at1996 | 0.5 16at1996 | Brazil | Leverling | United Package | 10 | 20 | |
| 17 | 1 | 10253 | HANAR | 0.5 10at1996 | 0.5 24at1996 | 0.5 16at1996 | Brazil | Leverling | United Package | 14.4 | 42 | 6 |
| 18 | 1 | 10253 | HANAR | 0.5 10at1996 | 0.5 24at1996 | 0.5 16at1996 | Brazil | Leverling | United Package | 16 | 40 | |
| 19 | 1 | 10254 | CHOPS | 0.5 11at1996 | 0.5 8at1996 | 0.5 23at1996 | Switzerland | Buchanan | United Package | 3.6 | 15 | |
| 20 | 1 | 10254 | CHOPS | 0.5 11at1996 | 0.5 8at1996 | 0.5 23at1996 | Switzerland | Buchanan | United Package | 19.2 | 21 | 34 |

The report title, grouping, summary, and filter information display on the export results, if your application has been configured for it.

Developers control the availability of different types of exports. In Info Studio, you can set the new attribute on the Target.NativeExcel element called "ExportDataPreference" (Formatted, Empty {default}, or Rapid ). This preference can be set on a global application level by setting a restricted constant `rdExportDataPreference` to *Rapid*, like below:




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*.

v23.1 Also in Studio, developers can format boolean columns by adjusting the Excel Format values in the Excel Column Format element. Options include: Yes/No, On/Off, and True/False.

Additionally, Rapid Excel exports can now resolve tokens.

For more information about how to configure your Info application to export to Excel, see "Excel - Exporting a Report Manually" on page 315.

# Word Template

A Logi template definition acts as a "blueprint" showing how the Word template will be filled by the Logi Server Engine at runtime. Filling a Word template with data starts with the creation of the Word template; once it has been created, three more steps are required.

The following topics discuss how to create and use a Logi Template definition with Microsoft Word templates:

- Adding a Word Template to Your Application
- Creating a Logi Template Definition
- Working with Hierarchical Data

For information about how to *call* Template definitions from within Logi applications, see "Form-based Reporting" on page 224.

## About Templates

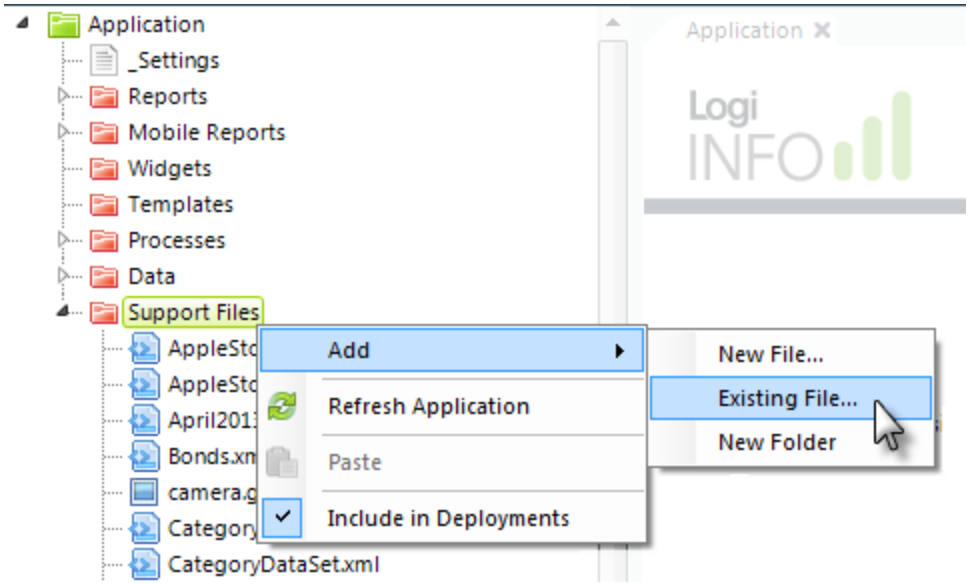
The term "template" applies here in several different ways, so let's clarify its use at the outset:

- A **Word template** is a special variety of Microsoft Word document, saved as either a *.dotx* or a *.dot* file, depending on your version of Word. It looks like the finished document but without any data; it has form fields reserved for the data. This is also referred to in the document as the "target template".
- A Logi Info **Template definition** is an *.lgx file*, similar to a Report or Process definition, created with Logi Studio. It's a set of instructions to the Logi Server Engine to retrieve data and it maps that data into the fields in the target template.
- Other uses of "template", such as Word document formatting templates or Avery Label templates, *do not* apply in this document.

The rest of this topic assumes that you've already created your Word template.

# Adding a Word Template to Your Application

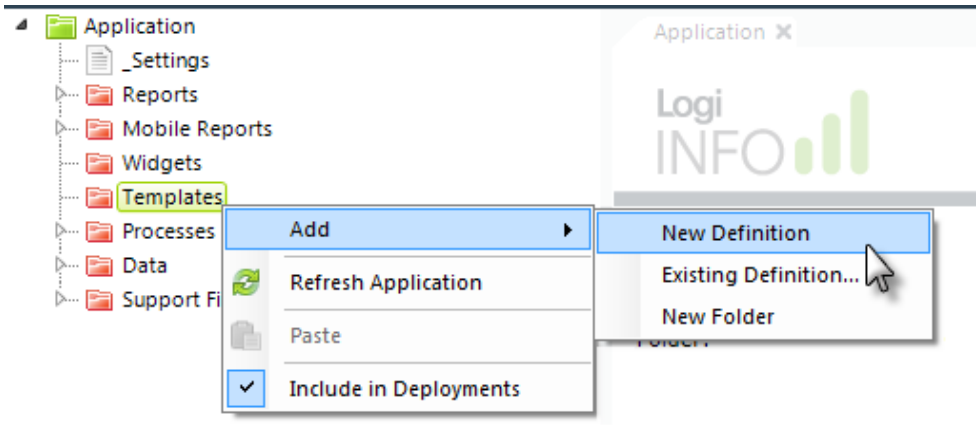
Word template files are managed within your Logi application as **Support Files**. Adding them is easy:



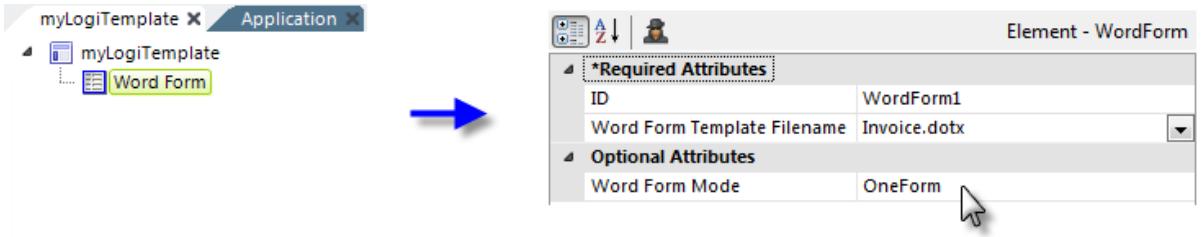

1. In Logi Studio's Application Panel, select and right-click the Support Files folder, as shown above.
2. Select **Add** and then **Existing File...** from the context menus.
3. Browse to your Word template (.dotx or .dot) file and select it. Click **OK** to add it to the application. The file will be *copied* to the _SupportFiles folder within your application folder.

# Creating a Logi Template Definition

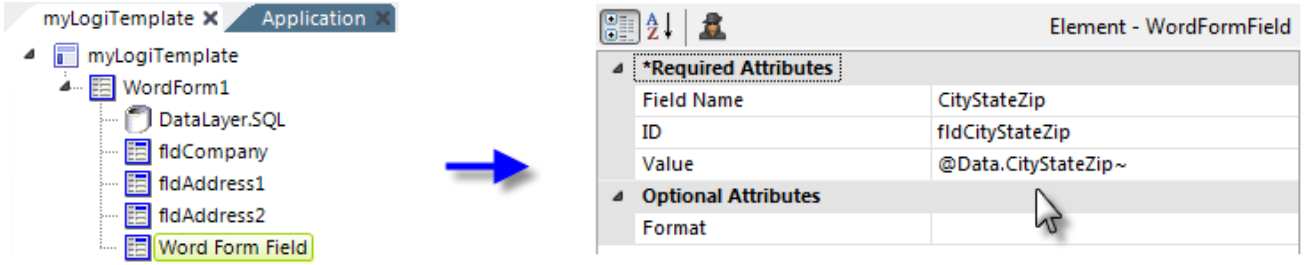
After you have added a Word template to your Logi application, the next step is to create a new Logi Template definition that maps the data into the Word template form fields:




1. In the Application Panel, select and right-click the **Templates** folder, as shown above.
2. Select **Add** and then **New Definition** from the context menus.
3. The definition will be created with the standard name "newTemplate". Rename it as you desire. Your new definition file has been created in the _Definitions/_Templates folder within your application folder and should open in the Workspace panel.

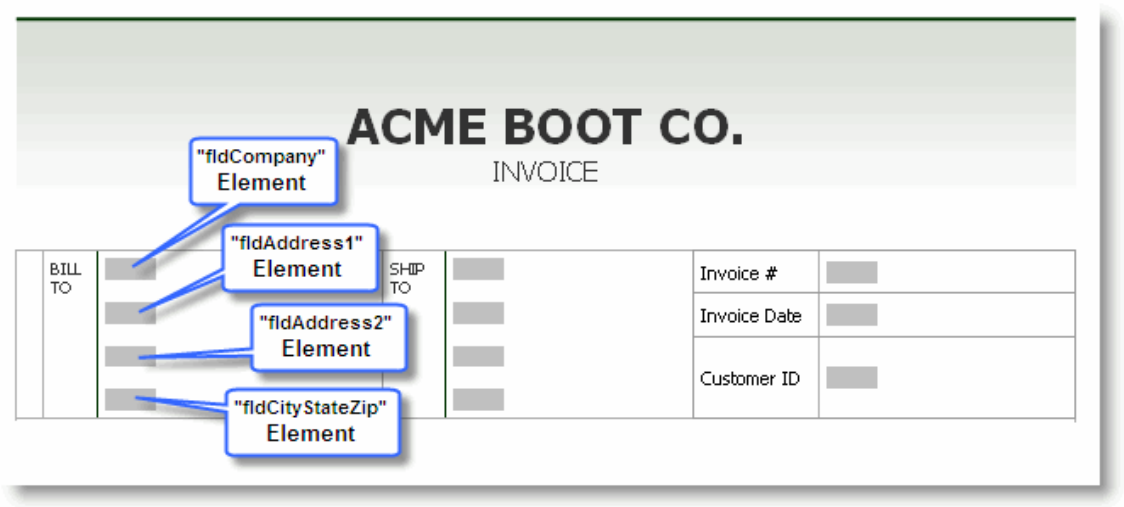


4. In the Workspace editor, as shown above, add a **Word Form** element to the definition.
5. Set its **Word Form Template Filename** attribute to the name of the .dotx or .dot file you added as a Support File (it should be available as an option in the value's drop-down list of suggestions).
6. Setting its **Word Form Mode** attribute is optional; its default value is *OneForm*. If this attribute is set to *OneForm*, all rows returned by the top-most data rows are processed in a single document page. If the attribute is set to *OneFormPerDataRow*, each row from the top-most data rows is treated as a separate document page. Developers will find this mode useful for invoice-style reports.




7. As shown above, beneath the Word Form element, add an appropriate **DataLayer** element and configure its attributes as needed.
8. Add a **Word Form Field** element.
9. Set its **Field Name** attribute to the corresponding field name in the Word template where this data should go. Field name spelling and case *must* match exactly!
10. Set its **Value** attribute to a token for the appropriate data column in the datalayer.

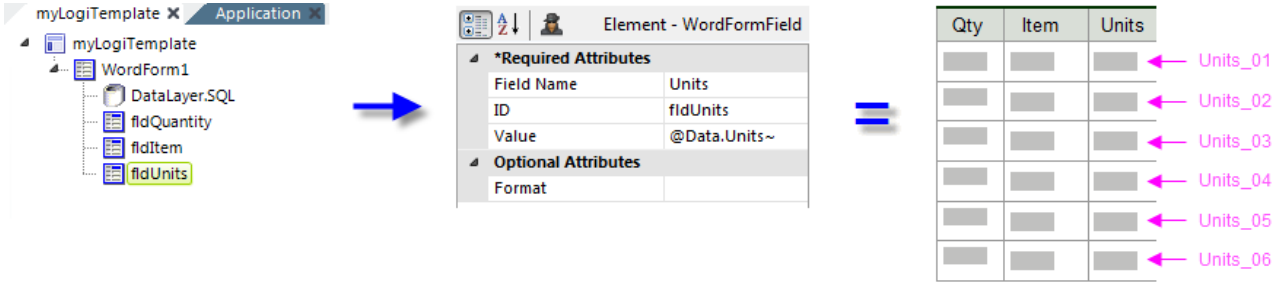
Repeat Steps 8-10 to add and configure additional Word Form Field elements as necessary to fill-in all the data in the template.




The Word template example above shows how four Word Form Field elements correspond to four form fields in the template. In this fashion, data is mapped to specific fields in the Word template.

# Working with Hierarchical Data

The Logi Engine also supports *hierarchical datasets* within Word templates. You can use elements such as **Group Filters** to shape the data appropriately before assigning columns to Word Form Field elements. The Logi server engine *does not* dynamically copy or add fields to the template based on the data returned in datalayer; at runtime, there must be one form field in the template for every possible data column. Developers must anticipate the maximum number of fields that could be required by the data and provide them, or otherwise apply limits in their queries.




When working with hierarchical data, only one Word Form Field element is required to fill repeating fields in a Word template. In the example above, the Word Form Field element "fldUnits" provides the data (from successive rows in the dataset) for all of the fields named Units_01 thru Units_06. You do not need one element for each form field.

# Exporting to Native Word

Exporting reports into **Microsoft Word** documents is a popular method of formatting and presenting data.

The following topics discuss techniques for exporting to Word:



## About Exporting Reports to Native Word

Exporting data into a Word document allows for additional formatting to be applied, and for narrative text to be included with visualizations. Logi Info includes specific elements for exporting data into documents that can be edited using Microsoft's widely-used Word program.

 Aways use the **Action.Export Native Word** element in your Logi applications. Earlier versions of Logi Info used the deprecated Action.Export Word Or Excel element and it's been retained in Logi Studio only for backward compatibility.

## Manual or Programmatic Export

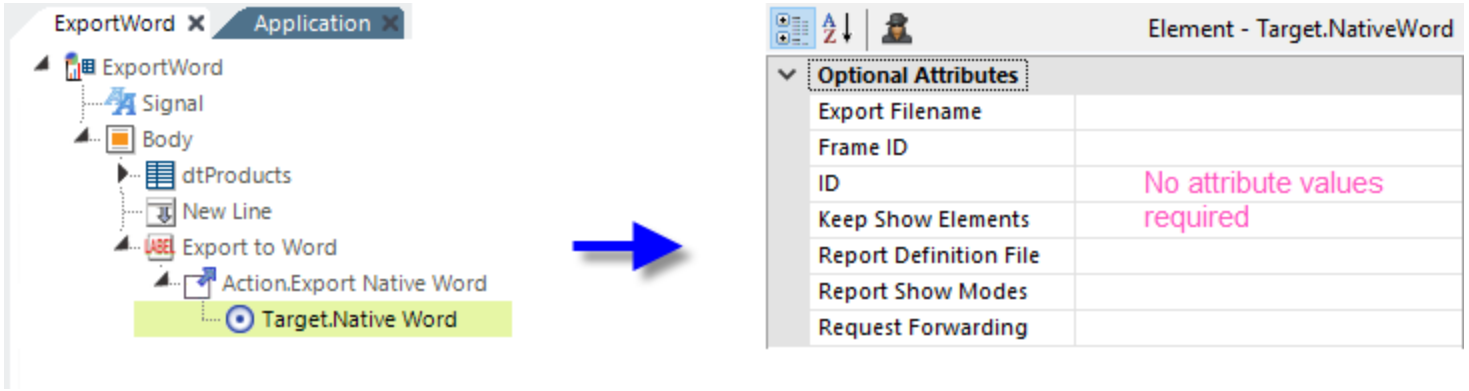
You can configure your application to export data to Word in two ways:

- **Manually** - The user clicks a link or button in a report to initiate the export and the result is made available for viewing in a browser or can be saved locally. The export elements are added into the report definition.
- **Programmatically** - The export occurs when a *Process Tasks* executes, triggered by an user-initiated event or scheduled event, and the result is written out as a Word file on the web server. The export elements are added into the process definition.

You may use both techniques in the same application.

# Word - Exporting a Report Manually

Here's an example of how to create a report with a link that exports manually:

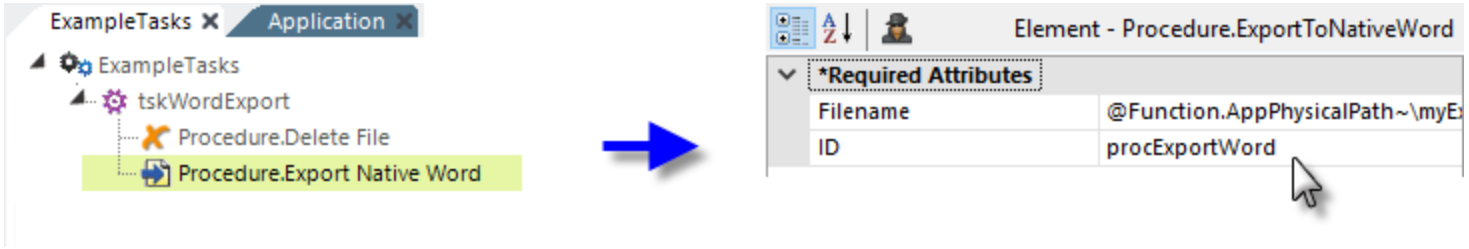



1. In your Report definition, add an **Action.Export Native Word** element to a Label, Image, Button or Chart element, as shown above.
2. Add the required **Target.Native Word** element.
3. If the report to export *is* the current report, you don't need to do anything else. Just save your definition and browse your report; it's that simple.
4. All Target.Native Word element attributes are *optional*. The default settings will export the current report in its entirety.
5. If the report to export *is not* the current report, specify that report by setting the Target.Native Word element's **Report Definition File** attribute.
6. The exported file will be given a random file name and a .doc file extension (Word 2003 and earlier) unless you specify a specific file name and the .docx extension (Word 2007+) in the **Export Filename** attribute.

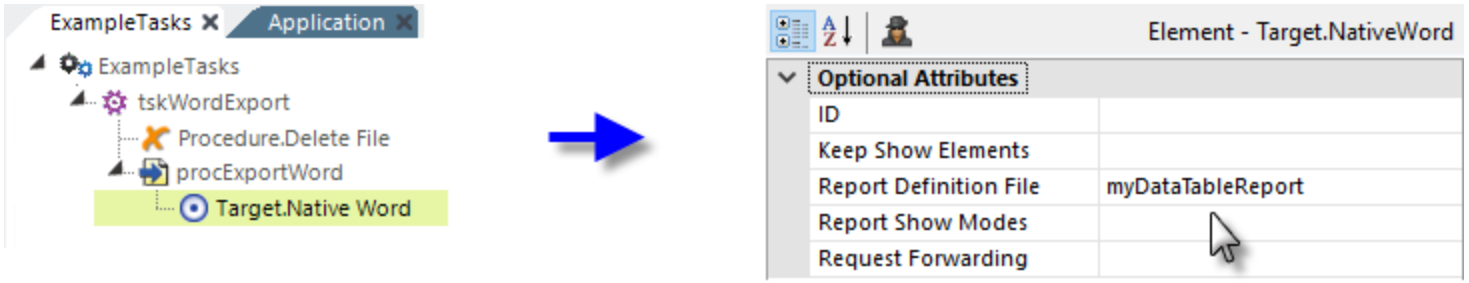
What Happens: The report is exported to a temporary file which is created in your project folder's rdDownload folder on the web server. The temp file is then returned to the client and opened in your browser. You may be prompted to *View* or *Save* the file. Temporary files on the server are cleaned up automatically over time.

# Word - Exporting a Report Programmatically

The following example, for Logi Info only, shows how to create a process task that exports data programmatically:




1. In your process definition, add a Task element.
2. Add a **Procedure.Delete File** element beneath the Task. The export *will not overwrite* an existing file so this element is used to delete any older version before the export occurs; no error will occur if there's no existing file to delete.
3. Add a **Procedure.Export Native Word** element beneath the task, as shown above.
4. In both elements' **Filename** attribute, specify the output path and filename on the web server for the exported document. The filename *should* include the .doc or .docx file extension. In the example above, this value uses a token to export the report to a folder called `myExports` within your project folder: `@Function.AppPhysicalPath~\myExports\myfile.docx`



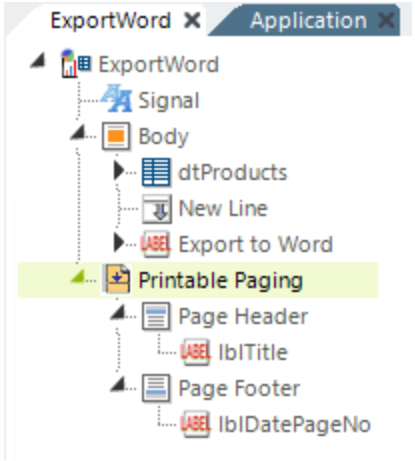
5. Add the required **Target.Native Word** element, as shown above.
6. In the element's **Report Definition File** attribute, specify the report definition to be exported.

In a Process definition, the "CurrentReport" option that appears in the suggested values for this attribute cannot be used. You *must* specify the actual name of the report.

What Happens: When your task runs, the specified report will be exported to a temporary file in your project folder's rdDownload folder on the web server; the temp file is then copied to your output file.

# Word - Adding Exported Headers and Footers

Exported reports are paginated and you may want to have a report header and/or footer appear on each exported page. This can be done, without affecting the normal appearance of the HTML report output in a browser, using the **Printable Paging** element.

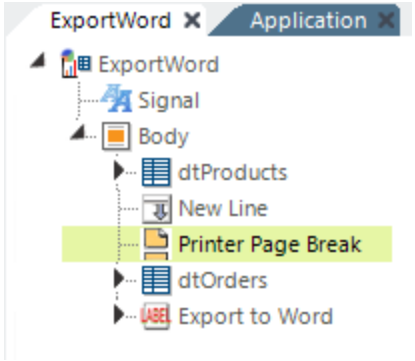



As shown above, the Printable Paging element has been added beneath the report's top-level Report element. It has its own **Page Header** and **Page Footer** child elements which are containers for **Label** elements. In the example above, the captions of the footer label could use the `@Date.Today~` and `@Function.PageNumber~` tokens to get their values.

When the report is run, the header and footer under the Printable Paging element will *not* be visible. When the report is exported, the header and footer *will be* visible on each exported page.

# Word - Forcing Page Breaks in Exports

When a report is paginated during an export, it may be useful to be able to force a page break, for example, to ensure that sections of the report start on new pages. This can be done using the **Printer Page Break** element.




You can add one or more **Printer PageBreak** elements beneath the **Body** element to break the exported pages where desired. In the example above, these elements ensure that the "dtProducts" table and the "dtOrders" table start on new pages. When the report is viewed in the browser, however, like the Printable Paging element, the Printer Page Break element has no effect.

## Forcing a Data Table Row Break

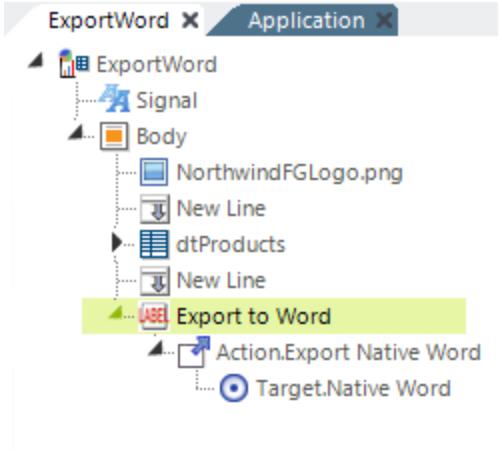
When working with Data Tables with multi-line rows, you may want to ensure that rows break cleanly across pages of their Word exports. In other words, they want to prevent different lines of the same Data Table row from appearing on different Word document pages. This can be accomplished through CSS, using code like:

```
.noPageBreakCell { page-break-inside: avoid; }
```

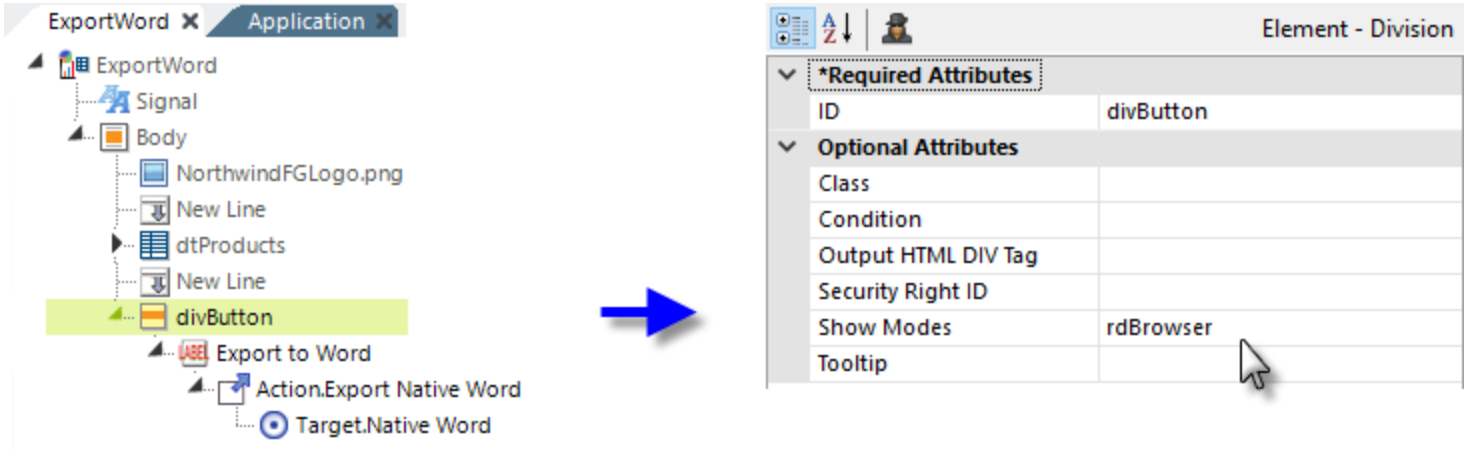
Assigning the above class to a **Data Table Column** element will create the desired effect.

# Word - Hiding Elements When Exporting

When exporting to Word, youcommonly want to hide some of the elements in the report page, like the Export button or link. This can be done very easily using Show Modes, see *Show Modes*.




In the example shown above, the definition on the left includes a Label (styled as a button) that can be clicked to export the report to Word. However, when it's exported, shown on the right, the Export button appears in the Word document. We don't want that text to appear in the document.



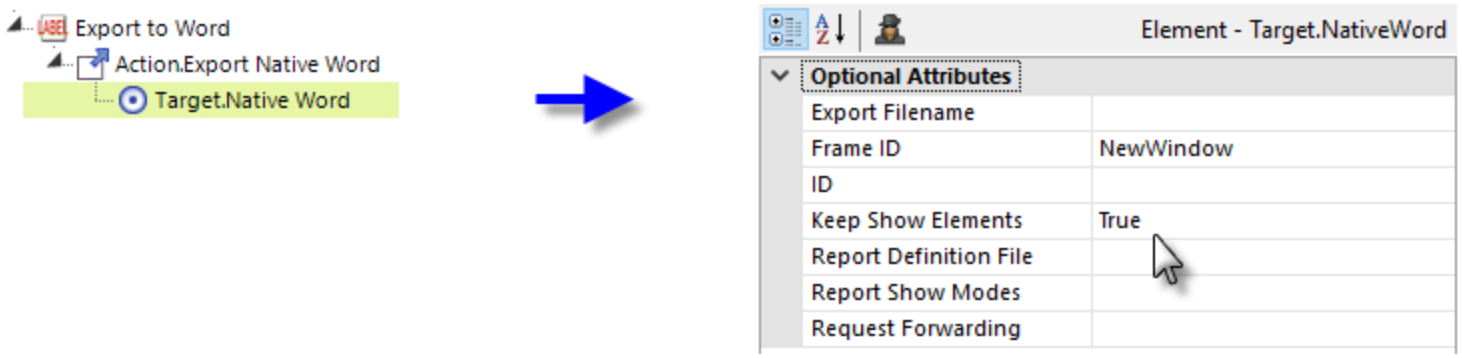

To "hide" the button in the exported document, simply place the elements beneath a **Division** element, as shown above. This element supports Show Modes, which the Label does not. Then set the Division element's **Show Modes** attribute value to the built-in *rdBrowser* value, as shown above. Elements with this ShowMode value will only be visible in the browser.


| ProductID | ProductName | SupplierID |
|---|---|---|
| 1 | Chai | 1 |
| 2 | Chang | 1 |
| 3 | Aniseed Syrup | 1 |
| 4 | Chef Antons Cajun Seasoning | 2 |
| 5 | Chef Antons Gumbo Mix | 2 |
| 6 | Grandmas Boysenberry Spread | 3 |
| 7 | Uncle Bobs Organic Dried Pears | 3 |
| 8 | Northwoods Cranberry Sauce | 3 |
| 9 | Mishi Kobe Niku | 4 |
| 10 | Ikura | 4 |

When the report is run and exported, as shown above, the document no longer includes the "Export to Word" button.

# Word - Exporting More Info Rows

If your report contains a Data Table and you're using the **More Info Row** element to show/hide additional detail data within the table, you may want the detail rows (when visible) to be exported along with the main table data.




To enable this, as shown above, set your Target.Native Word element's **Keep Show Elements** attribute to *True*. Keep Show Elements works with Action.Show Elements (used to show/hide the More Info Row) so that items that have been shown or hidden by the user remain that way when the report is re-displayed or

# Word - Cascading Style Sheet Support

When a Word export is being generated, the export engine processes any style sheet information along with the report data. The style sheet assigned to the report is used; you do not need to do anything additional to specify the style sheet. If desired, you can use two different style sheets in your report definition, one for the report and one for the Word export:

1. Select the root element of your report definition and set its **Default Show Modes** attribute value to an arbitrary string, such as "Normal".
2. Add two Style Sheet elements to your report definition.
3. Assign one Style Sheet element's **Show Modes** attribute a value that matches the string used in Step 1, "Normal". This style sheet will be used for the regular HTML report.
4. Assign the other Style Sheet element's Show Modes attribute a value that matches the Target.Native Word element's Report Show Modes attribute value (both should be an arbitrary string that *does not match* the string from Step 1, such as "Export".

Now when the report runs in a browser, the first style sheet will be applied; when it's exported to Word, the second style sheet will be applied. What if you want to combine the effects from this topic and "Word - Hiding Elements When Exporting" on page 351, i.e. hide some elements and use different spreadsheets? Just remember that the Show Modes attribute can accept several values, separated by a comma. To combine the examples from these two sections, ensure that the root element's Default Show Modes attribute includes both Show Modes values: "Normal,NoExport" (do not enter the double quotes).


The following considerations should be noted when working with style sheets for Word exports. Failure to follow these guidelines will result in style sheet information being ignored by the Word exporter, or, in some cases, failure of the export process.

1. When defining CSS classes in their style sheet(s), you *must* include a space or line feed between the name of a class and the opening curly-brace. These are **valid** classes:

```
.myClass1{                        color: Blue; }
.myClass2
              {
              color: Blue;
 }
```

but this is in *invalid* class:

```
.myBadClass{ color: Blue; }
```

2. You may combine classes using the CLASS attribute of elements in a report definition file:

   ```
   CLASS="class1 class2"
   ```

3. You *must* use the **Width** and **Width Scale** attributes when setting table and cell widths. The CSS "width" property is not supported.
4. If duplicate class definitions exist, only the first definition is used.
5. If duplicate properties appear inside a class, only the last occurrence is used.
6. The following CSS *Selectors* are supported by the Word export engine:

| Selector | Example |
|---|---|
| Type | TABLE |
| Class | .<*classname*> (begins with a period) |

| Selector | Example |
|---|---|
| ID | #second_table |
| Type-prefixed Class | TABLE.<*classname*> |

7. The following CSS Level 1 *Properties* are supported by the Word exporter. The *order* of the values for properties with multiple values is significant. Multiple values must be listed in the order defined in this table or the Word exporter will ignore the property altogether:

| Property | Value/Example |
|---|---|
| background | <background-color> |
| background-color | Web colors (examples: #00FF00 or Green) |
| border | <border-width> <border-style> <border-color><br><border-style> <border-color><br><border-color> |
| border-bottom | <border-width> <border-style> <border-color><br><border-style> <border-color><br><border-color> |
| border-color | Web colors (examples: #00FF00 or Green) |

| Property | Value/Example |
|---|---|
| border-style | Example: solid |
| border-top | <border-width> <border-style> <border-color><br><border-style> <border-color><br><border-color> |
| border-width | Pixels (example: 25px) |
| color | Web colors (examples: #00FF00 or Green) |
| filter | Examples: flipv, fliph |
| font | <font-style> <font-weight> <font-size> <font-family>; |
| font-family | Base-14 & TrueType fonts (example: Arial) |
| font-size | Points (example: 12pt) |
| font-style | Example: italic |
| font-weight | Example: bold |
| padding | <padding-top> <padding-right> <padding-bottom> <padding-left> |
| height | Pixels (example: 25px) |

| Property | Value/Example |
|---|---|
| padding-top | Pixels (example: 25px) |
| padding-right | Pixels (example: 25px) |
| padding-bottom | Pixels (example: 25px) |
| padding-left | Pixels (example: 25px) |
| text-align (see Note below) | Examples: left, right, center, justify |
| text-decoration | Examples: underline, overline |
| vertical-align | Examples: top, middle, bottom |
| writing-mode | Examples: lr-tb, tb-rl |

## Style Tips

- With a limited list of CSS properties, if Native Word exporting is going to be used in an application, it's a good idea to design the CSS classes used for elements to be exported with these limitations in mind. That way a single style sheet could be used for all exports and the HTML display of a report. Many of these property limitations stem from the fact that there isn't a good way to translate certain CSS properties into a formatting setting in a native Word document; the formatting options available in a native Word document are much more limited than those available with CSS.
- When it comes to specifying font-size, Word can't handle pixel values. All pixel values specified in CSS classes for font-size will be converted to point values. Therefore, it's highly recommended that points be used to specify the font-size.

- Word builds tables sequentially, cell-by-cell, left-to-right, top-to-bottom. Word will attempt to apply all the properties of the previous cell to the cell currently being created unless the properties are explicitly defined differently for the current cell using a different CSS class which overwrites all the properties of the previous cell. For instance, if the first cell has a "border-bottom: 1px solid black;" class applied and you want the next cell to appear without a bottom border, you would have to apply a class with the following property: "border-bottom: 1px solid white;".
- Use **text-align** to set the alignment of tables. This is not generally allowed by the rules of CSS, however, an exception is made because the **align** attribute is not supported.

# Word - Debugging Exports

When the debugger has been set to *Debugger Links* and the report is run, and then exported, a debug link will be included at the bottom of the exported report:

| ProductID | ProductName | SupplierID |
|---|---|---|
| 1 | Chai | 1 |
| 2 | Chang | 1 |
| 3 | Aniseed Syrup | 1 |
| 4 | Chef Antons Cajun Seasoning | 2 |
| 5 | Chef Antons Gumbo Mix | 2 |
| 6 | Grandmas Boysenberry Spread | 3 |
| 7 | Uncle Bobs Organic Dried Pears | 3 |
| 8 | Northwoods Cranberry Sauce | 3 |
| 9 | Mishi Kobe Niku | 4 |
| 10 | Ikura | 4 |

Export to Word

1

Export the original report, with debugging turned on

| ProductID | ProductName | SupplierID |
|---|---|---|
| 1 | Chai | 1 |
| 2 | Chang | 1 |
| 3 | Aniseed Syrup | 1 |
| 4 | Chef Antons Cajun Seasoning | 2 |
| 5 | Chef Antons Gumbo Mix | 2 |
| 6 | Grandmas Boysenberry Spread | 3 |
| 7 | Uncle Bobs Organic Dried Pears | 3 |
| 8 | Northwoods Cranberry Sauce | 3 |
| 9 | Mishi Kobe Niku | 4 |
| 10 | Ikura | 4 |

Click the debug link that appears in the exported document

2

Show the Debugger Trace Report

| | | | |
|---|---|---|---|
| | Finalize DataLayers | | .092 |
| | Data Engine | Data processing completed. | .092 |
| Generated DataLayers | File stream | View Data | .093 |
| | DataLayer Summary XML | View Data | .095 |
| XSL Transformation | Start | | .097 |
| | Finish | | .100 |
| HTML Token Replacement | Start | HTML Size: 15,518 | .101 |
| | Finish | HTML Size: 15,523 | .101 |
| Convert <BR></BR> to <BR /> | Start | | .103 |
| | Finish | HTML Size: 15,517 | .103 |
| Report | Paging Method | Printable | .104 |
| Exporting to NativeWord | Start | | .110 |
| Native Export Generation | Timeout seconds | 0 | .114 |
| | Page Width | 8.5 | .115 |
| | Page Height | 11 | .115 |
| | Export Html | View Export Html | .116 |
| | Finish | | .225 |


This mechanism allows you to review the export process in more detail and can help to diagnose any problems that may arise.

# Word - Export Considerations

The following apply when exporting to Word:

- Chart color transparency is not available in exports to Word. Non-animated charts displayed in Logi HTML reports are rendered as .png images and support transparency, however, when exported to Word, these charts are rendered as .jpg images, which do not support transparency.
- Embedded HTML files in a Logi report may not be exported correctly; success is dependent on the nature of the external HTML file, its adherence to proper XHTML syntax, and other factors.
- Background colors are only applied within tables
- Bulleted lists are not supported (using the Bullet element)
- Rectangle element is not supported
- Layout Positioning is not supported
- HideDuplicates element is not supported

keywords: MoreInfoRow

# Exporting to CSV File

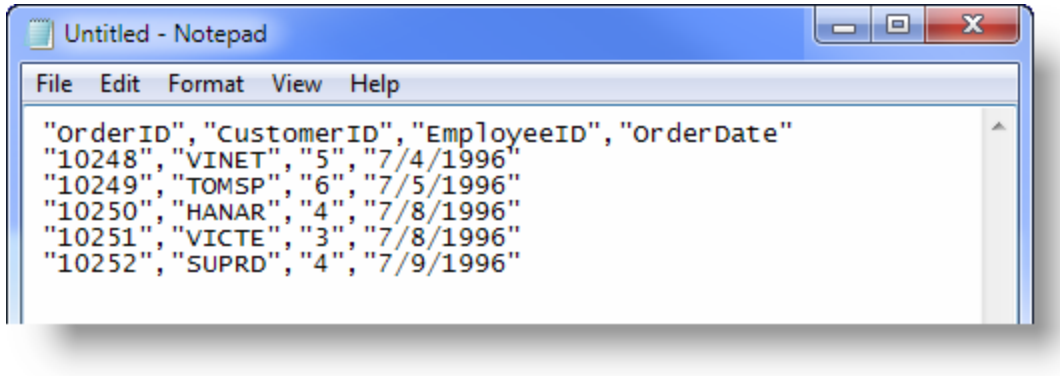
The **Comma-Separated Values** (CSV) format is a well-established data file format and a widely-accepted method of transferring data.

The following topics discuss how to export Logi report data to CSV files:

- Exporting Data Manually
- Exporting Data with Automation
- Adjusting the Encoding

## About Exporting Data to CSV

The CSV data format is useful because of its relative simplicity and universal cross-platform accessibility. CSV files are simple text files that are generally used to represent tabular data: each row of text in the file represents one data record and, in the typical format, each column of data is enclosed in double-quotes and delimited by a comma. The first row may represent the column names. Here's an example:



Logi Info provides **Export CSV** elements that make it easy to export report data into CSV files.

The data exported to CSV files is *only* the data in the report's datalayers. Summary Rows, for example, are *not* included in the export. Logi developers can give users the ability to export a data *manually* via user interaction at runtime or developers can *automate* data exports based on an event or a schedule. Manual exports are configured within Report definitions and automated exports are configured within Process definitions.

For large data sets, we recommend using the rapid CSV export option. Below is a formal list of the differences between native CSV and rapid CSV exports:

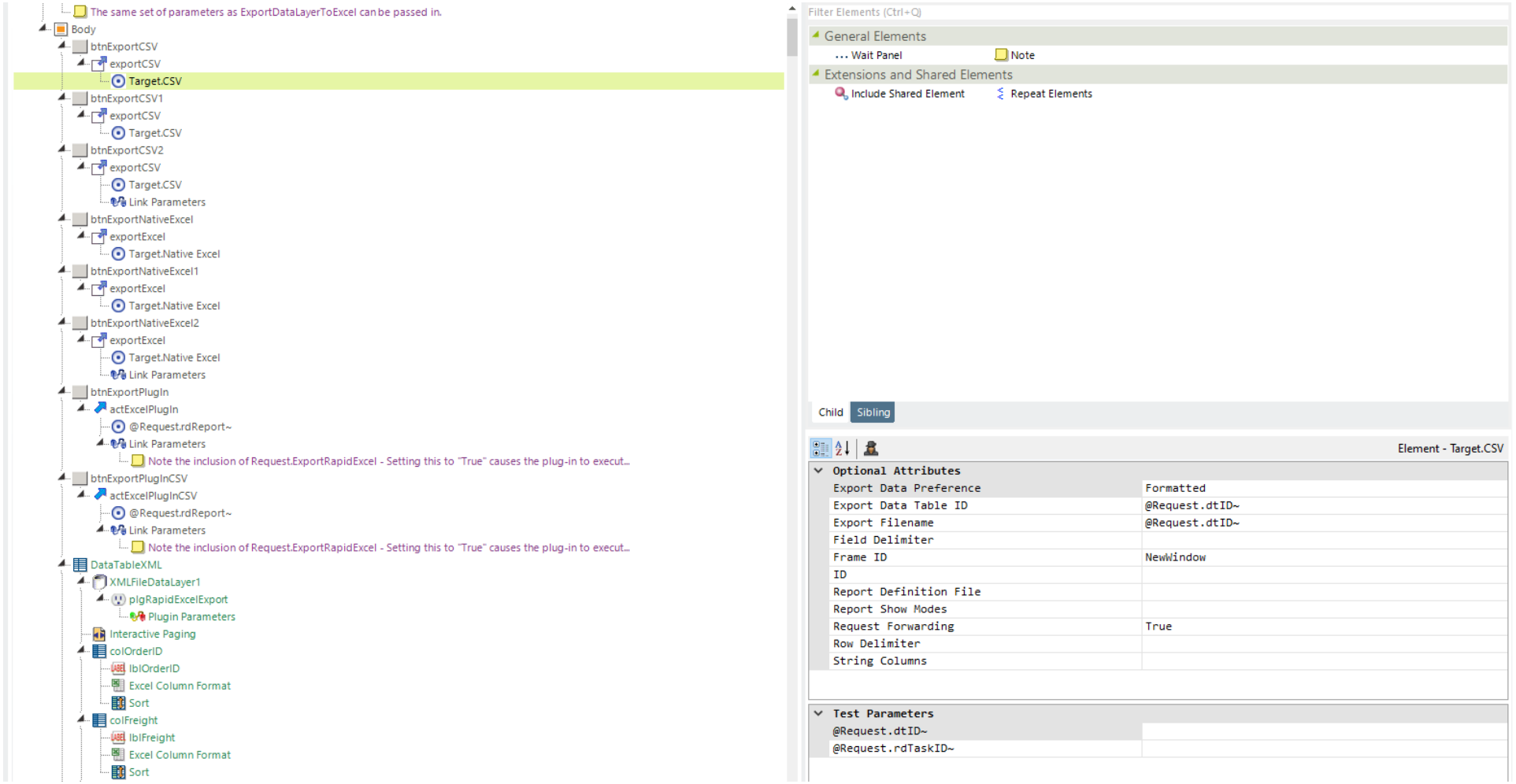
| Native CSV Export | Rapid CSV Export |
|---|---|
| Can export Crosstab Tables | Cannot export Crosstab Tables |
| Can use Field Delimiter, Row Delimiter, and String Column attributes | Cannot use Field Delimiter, Row Delimiter, and String Column attributes |
| Can have a null value for the "Export Data Table ID" | Cannot have a null value for the "Export Data Table ID" parameter |
| Can export a Data Table created with the "Auto Columns" element | Cannot export a Data Table created with the "Auto Columns" element |
| Can use group or count info (like "Header Row" or "Summary Row" elements) | Cannot use group or count info (like "Header Row" or "Summary Row" elements) |
| Can support Data type attribute in “Excel Column Format” element | Can support Data type attribute in “Excel Column Format” element |

| Native CSV Export | Rapid CSV Export |
|---|---|
| Cannot use format setting "Excel column format" | Can use format setting "Excel column format" |
| Cannot export report title and filter information | Can export report title and filter information |
| Can export Group or Summary information | Can export group or summary information |

For more information on the rapid export feature, see "Rapid CSV Export" on page 376.

# CSV - Exporting Data Manually

Here's an example of how to create a report with a link that allows data to be exported manually:




1. In your report definition, add an **Action.Export CSV** element as the child of a **Label**, **Image**, **Button** or **Chart** element, as shown above. Set its required ID attribute.

2. Below it, add the required **Target.CSV** child element.
3. If the data to be exported is part of the *current report*, you need don't need to do anything else. Just save your definition and browse your report. It's that simple. Notice that the example above shows that no attribute values need be set: the default settings will export the current reports' data.

What happens when the link is clicked? The data in the report's datalayer is exported to a temporary .csv file which is created in your application's `rdDownload` folder on the web server. The temp file is then opened automatically in your browser, usually in the Microsoft Excel plug-in, if it's installed (the .csv file extension gets associated with it when Excel is installed). With the exception of data column header text, *only* data is exported; report headers, footers, paging controls, images, summary rows, etc. from the source report are *not* exported. If the source Data Table has column header labels, these will be exported as the first row of the CSV file.

The following table provides explanations for the Target.CSV element's attributes, which are all optional:

| Identifier | Description |
|---|---|
| Export Data Preference | Specifies the export preference. Options include Empty (default), Rapid, and Formatted. For more information on Rapid CSV Export, see "Rapid CSV Export" on page 376. |
| Export Data Table ID | Specifies the **Data Table** from which data will be exported. This is useful if a report contains multiple Data Tables; if there is only one table in the report, this attribute can be left blank. To export the data in an **Analysis Grid** element, set this value to "dtAnalysisGrid", the ID of the Analysis Grid's internal Data Table. |
| Export Filename | Specifies an explicit file name for the CSV file that receives the exported data. This value should be a file name *only*, without a path, and may or may not include the .csv file extension. If left blank, a random GUID file name will be generated and used. |

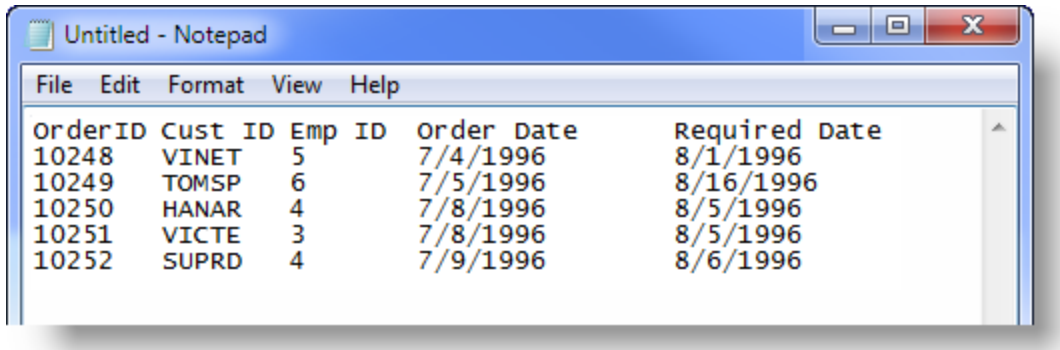
| Identifier | Description |
|---|---|
| Field Delimiter | Specifies the CSV file **datadelimiter**, typically a comma. Enter the desired character. You may use tokens in this attribute to set the delimiter dynamically. If left blank, the delimiter is determined first by inspecting the browser's language setting, and then by using the web server's List Separator character from its Regional Settings. This allows international users to get the appropriate field separator. If you use a Tab delimiter here, it will also remove the double-quotes around the values - see the following section for an example. |
| Frame ID | Specifies the **frame** for the target page. Leave blank for the current browser window, or select *NewWindow* to open a new browser window. You can also specify an existing frame identifier to re-use the same window for each request. |
| ID | Specifies a unique identifier for this element. |
| Report Definition File | Specifies the name of the report definition file that contains the data to be exported.<br>Default: *the current report* |
| Report Show Modes | Specifies the **Show Modes** to be used when rendering the data for export, making it possible to hide some of the data. When this attribute is blank, all of the elements and, therefore, all of the data is exported. To set Show Modes in the exported report, enter a comma-delimited list of Show Mode values in this attribute; the data from any elements in the report definition that have a Show Mode that matches on in this list will appear in the exported data. |
| Request Forwarding | Specifies whether **request variables** will be passed to the report definition file that contains the data to be exported. Has no effect if the current report is being exported. |
| Row Delimiter | Specifies the CSV file **row delimiter** character(s). Default: *standard CRLF characters* |

| Identifier | Description |
|---|---|
| String Columns | Specifies the **columns** of the exported data that should be interpreted as **string values**. The CSV format is frequently used to export data into Microsoft Excel. When Excel encounters a value that looks like a number, it removes any leading zeros. This is a problem for some values that should not be treated as numbers and should not have leading zeros removed. For example, "000123" might be an account number wherein the leading zeros are relevant. The removal of leading zeroes in Excel can be suppressed by supplying the appropriate Data Table Column element IDs, in a comma-separated list, in this attribute. |

The temporary files generated during the export are **cleaned up** automatically over time.

## Delimiting with the Tab Character

A common alternate CSV format uses the **Tab** character as the delimiter, which removes the double-quotes that usually enclose the field data. However, you can't type a Tab character in the **Field Delimiter** attribute so, to use this format, enter its ASCII value as an expression, `=CHR(9)`, in the attribute value instead.

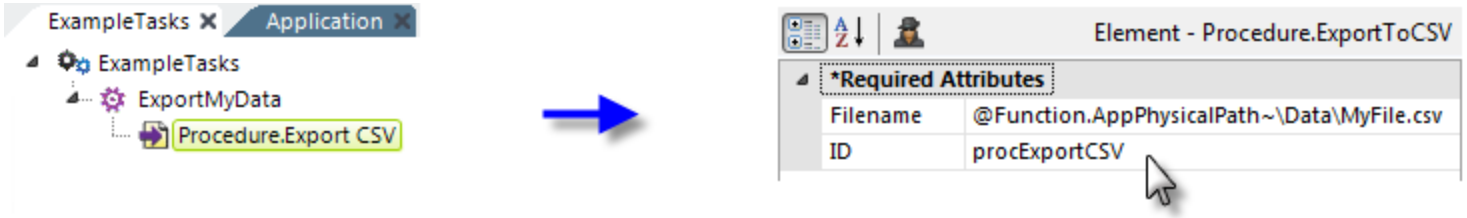



As shown above, when you run the report and export the data, it will be Tab delimited and have no double-quotes.

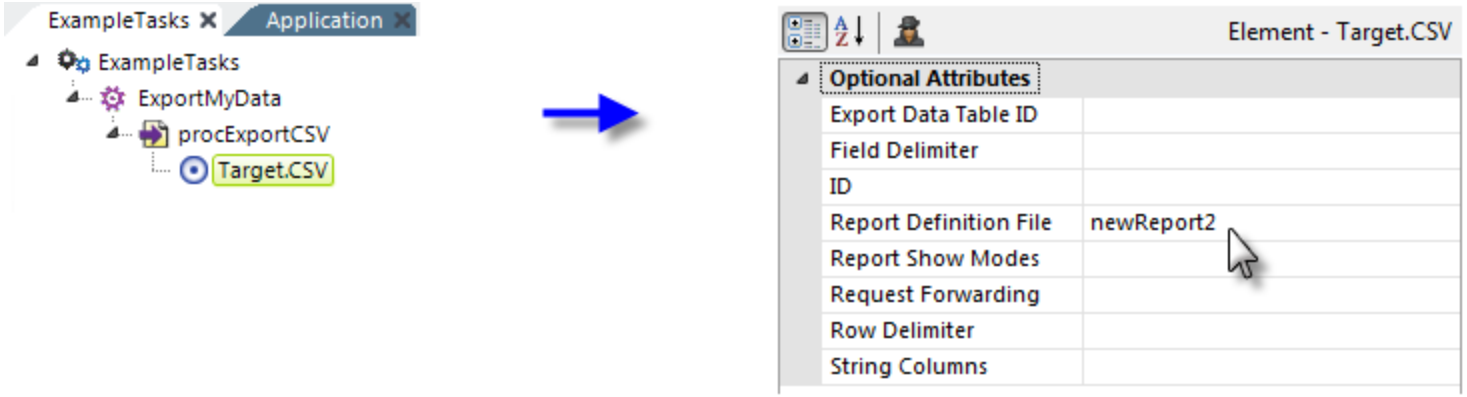
Using the ASCII code for a Tab delimiter is the *only* way to remove the double-quotes from the standard CSV format output. Entering other characters or ASCII codes in the Field Delimiter attribute will not remove them.

# CSV - Exporting Data with Automation

The following example, for Logi Info only, shows how to create a **process task** that exports data automatically:

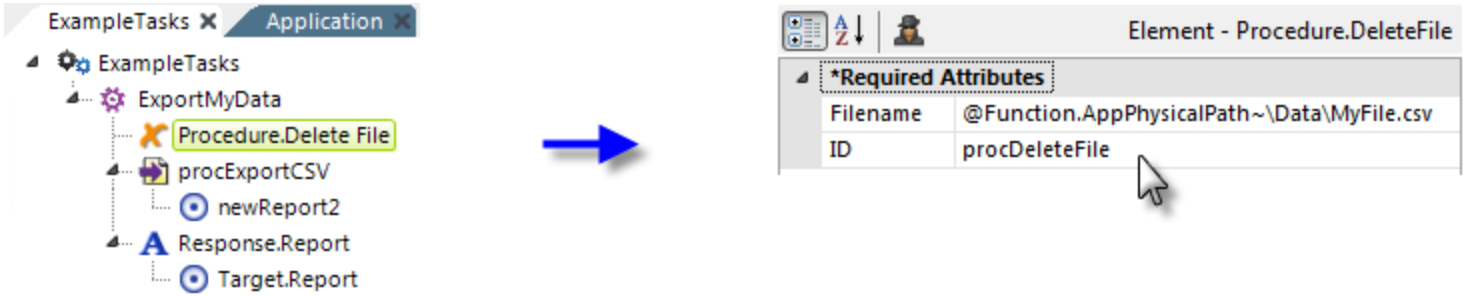



1. As shown above, in your Process definition, add a **Task** element and set its ID attribute.
2. Then add a**Procedure.Export CSV** element beneath it.
3. In the element's **Filename** attribute, specify the output path and filename, on the web server, for the exported report. The filename should include the ".csv" file extension. The example shown above uses the @Function.AppPhysicalPath~ token to provide the path to the application folder.



4. Next, add the required **Target.CSV** child element, as shown above.
5. In the element's **Report Definition File** attribute, specify the report that contains the data to be exported. Do not select the "CurrentReport" choice in the list of suggestions as it *will notwork* in this case.
6. Your task will need to be completed, of course, with a **Response** element to run properly.

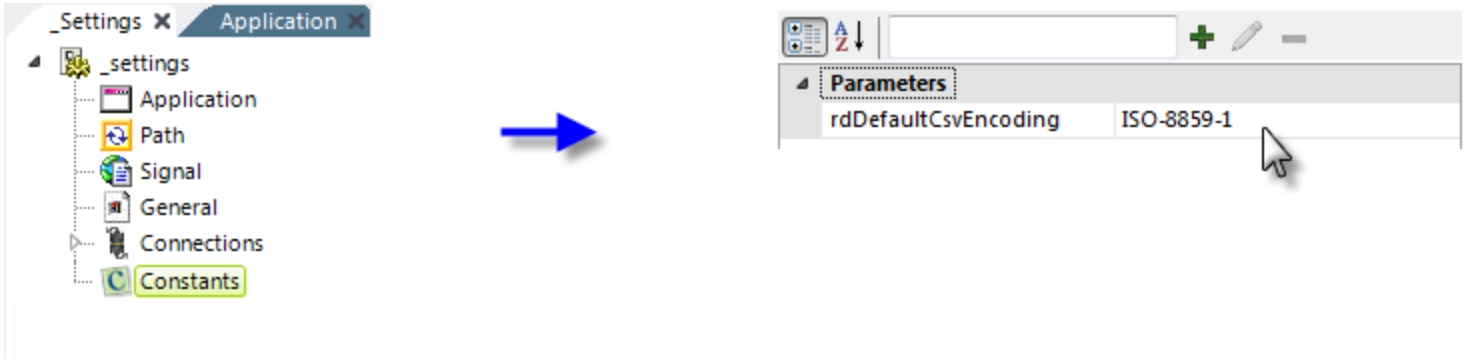
What happens when your task runs? The data from the specified report's datalayer will be exported to a temporary file in your project folder's `rdDownload` folder on the web server; then the temp file will be *copied* to the output file you specified. If you try to run this task more than once, you'll receive an error. This is due to the fact that **Procedure.Export CSV** will not *overwrite* an existing file. Adding a **Procedure.Delete File** element at the beginning of the task will solve this:




An example of a complete task is shown above. Note the use of the @Function again in the Procedure.Delete File element's **Filename** attribute.

# CSV - Adjusting the Encoding

By default, data exported to CSV format is output using **UTF-8** encoding.



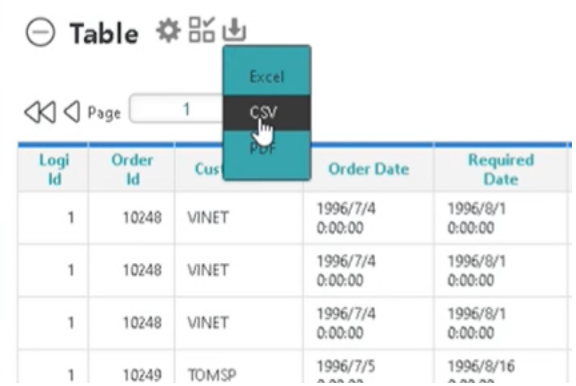

If you want to use **ISO-8859-1** encoding instead, you can do so by creating theconstant `rdDefaultCsvEncoding = ISO-8859-1` in your _Settings definition, as shown above.

# Rapid CSV Export

The Rapid CSV Export option provides a built-in capability for both the Data Table and Analysis Grid, running off the datalayer. This feature is available for Consumer and Self-Service users who export tabular data in CSV format for further analysis outside of the context of the Info application. The rapid export option greatly improves the performance of exporting large data, as compared to the regular CSV export option. To review a detailed list of the differences between the native CSV and rapid CSV export, see "Exporting to CSV File" on page 365.

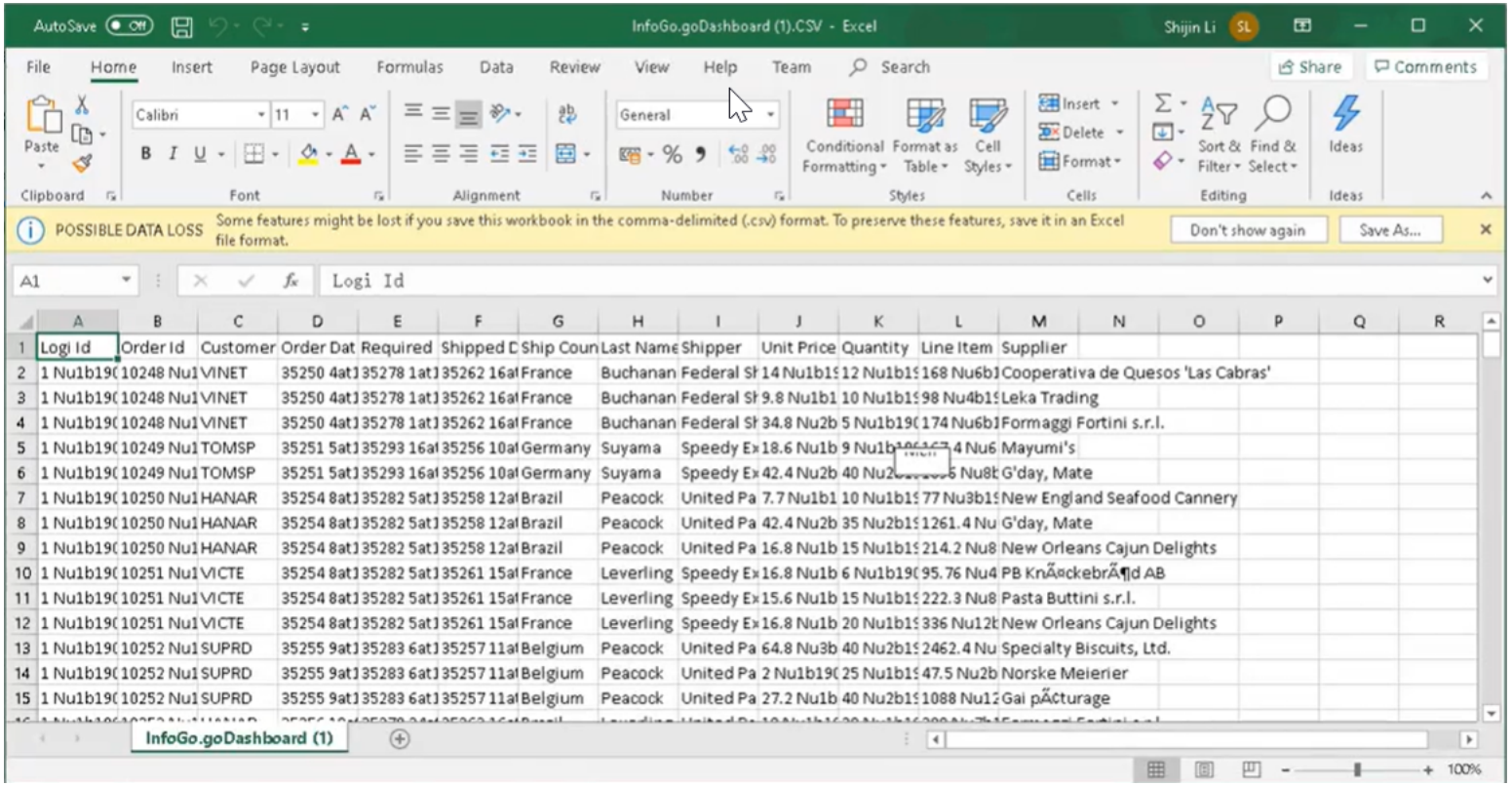
To export your data using the Rapid CSV Export, access your Data Table in the Analysis Grid and select the **export** icon.

Then, select **CSV** from the drop-down list:



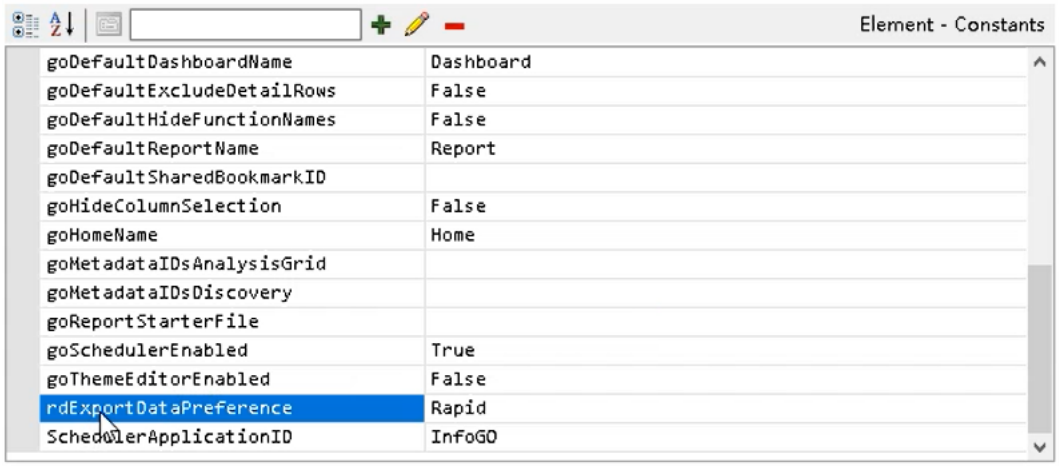

Your download will appear in the bottom of the web browser, or may open automatically, depending on your machine's configuration.

The Data Table displays in CSV format in whichever program you have set to open with the download. The example below shows the exported CSV file in Excel:




The report title, grouping, summary, and filter information display on the export results, if your application has been configured for it.

The availability of different types of exports is under developer control. In Info Studio, you can set the new attribute on the Target.CSV element called "ExportDataPreference" (Formatted, Empty {default}, or Rapid ). This preference can be set on a global application level by setting a restricted constant "rdExportDataPreference" to *Rapid*:




For more information about how to configure your Info application to export to CSV, see "CSV - Exporting Data Manually" on page 368.

# Exporting to XML

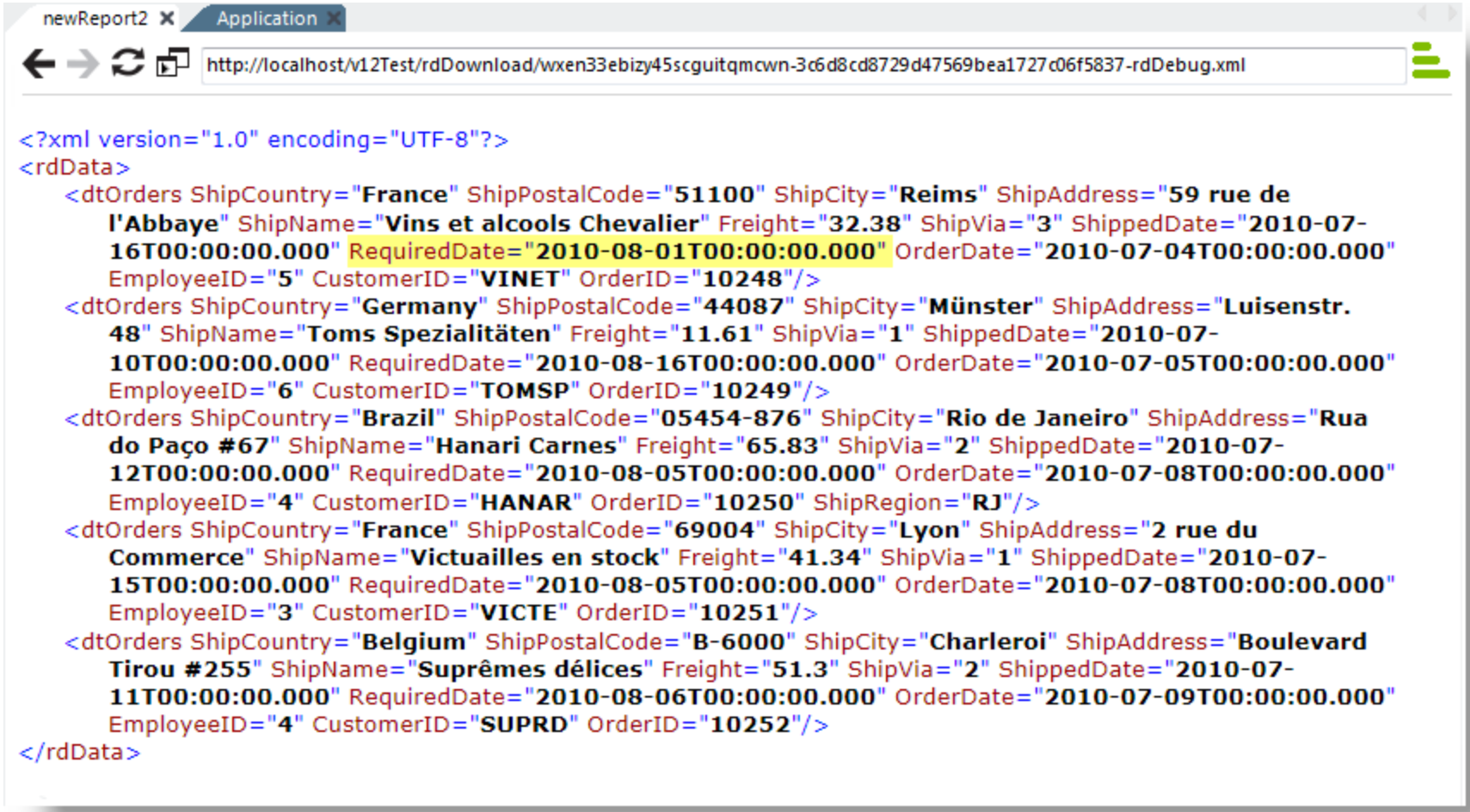
Data stored in the Extensible Markup Language (XML) format allows information systems to share structured data and is a widely-accepted method of storing and transferring data. Logi products use XML files as temporary caches for a variety of purposes and can export Logi report data to XML files.

The following topics discuss techniques for exporting to XML:



## About Exporting Data to XML

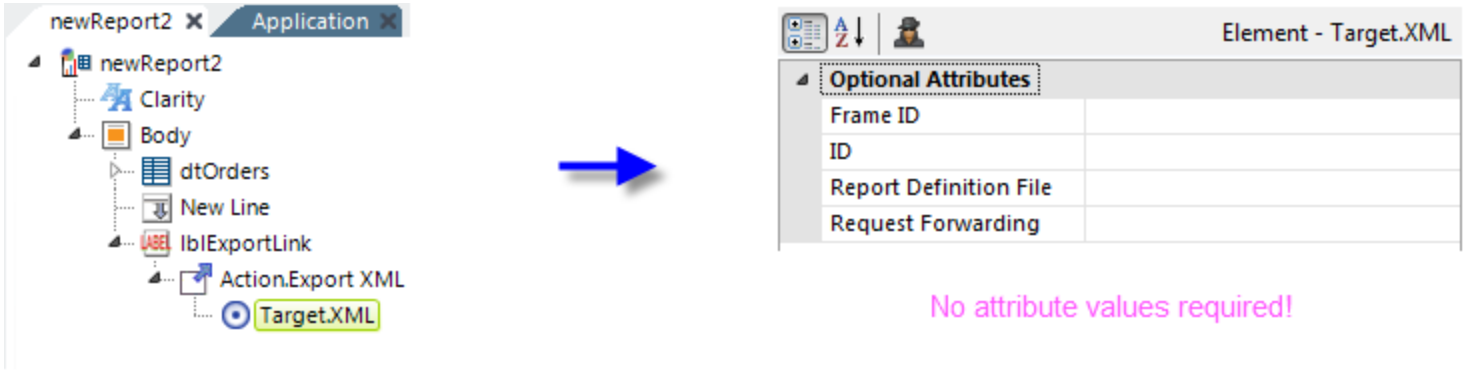
The XML data format is useful because of its relative simplicity and general cross-platform accessibility. XML files are simple text files that can be used to store a variety of data. In the schema used by Logi Info for tabular data, an XML element set represents one data record, each column is identified by an XML attribute name, and its data is the attribute value. Logi Studio provides **Export XML** elements that make it easy to export Logi report data into XML files.




The example above shows report data that has been exported into an XML data file. Note the formatting of the highlighted DateTime data: it's in the **ISO 8601** format. Record 1 of the dataset is everything between the first `<dtOrders` and `/>`. You can give users the ability to export a data *manually* via user interaction at runtime or choose to *automate* the export process based on an event or schedule. Manual exports are configured within **Report** definitions and automated exports are configured within **Process** definitions.

# XML - Exporting Data Manually

Here's an example of how to create a report with a link that allows data to be exported manually by the user:




1. In your report definition, add an **Action.Export XML** element as the child of a **Label**, **Image**, **Button** or **Chart** element, as shown above.
2. Below it, add the required **Target.XML** child element.
3. If the data to be exported is part of the *current report*, you don't need do anything else. Just save your definition and browse your report. It's that simple. The default settings will export the current reports' data.

What happens when the link is clicked? The report's data is exported to a temporary .XML file which is created in your application's `rdDownload` folder on the web server. The temp file is then opened automatically in your browser (or in whatever tool the .xml file extension has been associated with in the OS). From there, you can save it to a file, if desired.

XML files opened in a browser are usually formatted for easier viewing, which may introduce extraneous characters. If you want to save a "clean" copy of the XML (which is necessary if you want to use the file as a data source later), right-click the displayed XML and select your browser's "View source" option, then save that display to a file. Only the data from the datalayer is

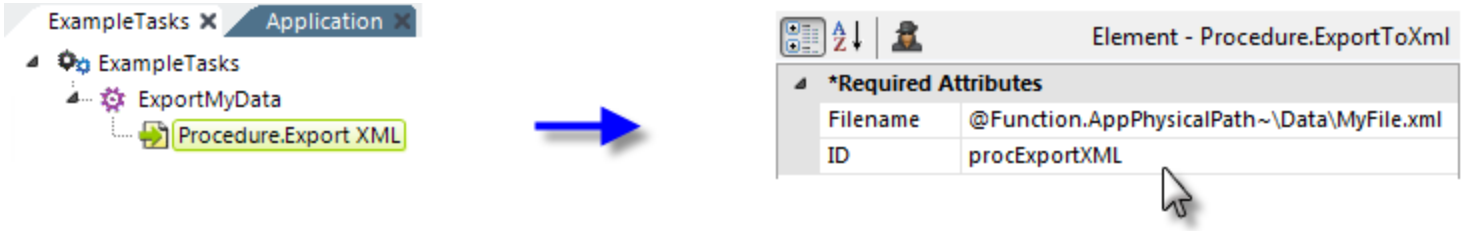
exported; headers, footers, paging, images, etc. from the source report are not exported. Any conditions, filters, aggregations, etc. that have been applied to the datalayer will be reflected in the exported data. The following table provides explanations for the Target.XML element's attributes, which are all optional:

| Attribute | Description |
|---|---|
| Frame ID | Specifies the **frame** for the target page. Leave blank for the current browser window, or select *NewWindow* to open a new browser window. You can also specify an existing frame identifier to re-use the same window for each request. |
| ID | Specifies a unique identifier for this element. |
| Report Definition File | Specifies the name of the report definition file that contains the data to be exported.<br>Default: *the current report* |
| Request Forwarding | Specifies whether request variables will be passed to the report definition file that contains the data to be exported. Has no effect if the current report is being exported. |

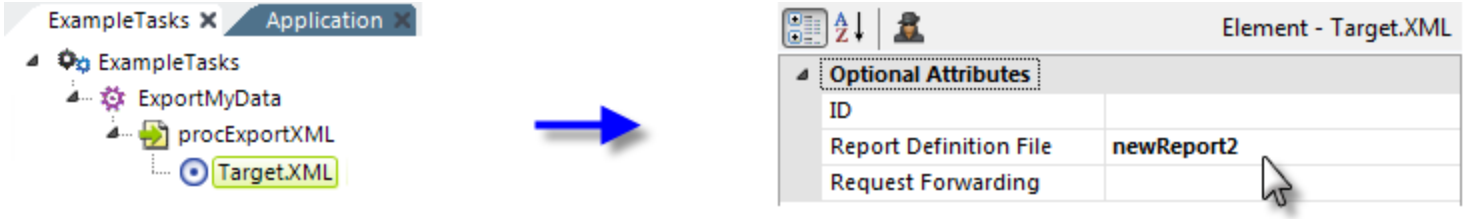
The temporary files generated during the export are **cleaned up** automatically over time.

# XML - Exporting Data Automatically

The following example, for Logi Info only, shows how to create a **Process task** that exports data automatically:

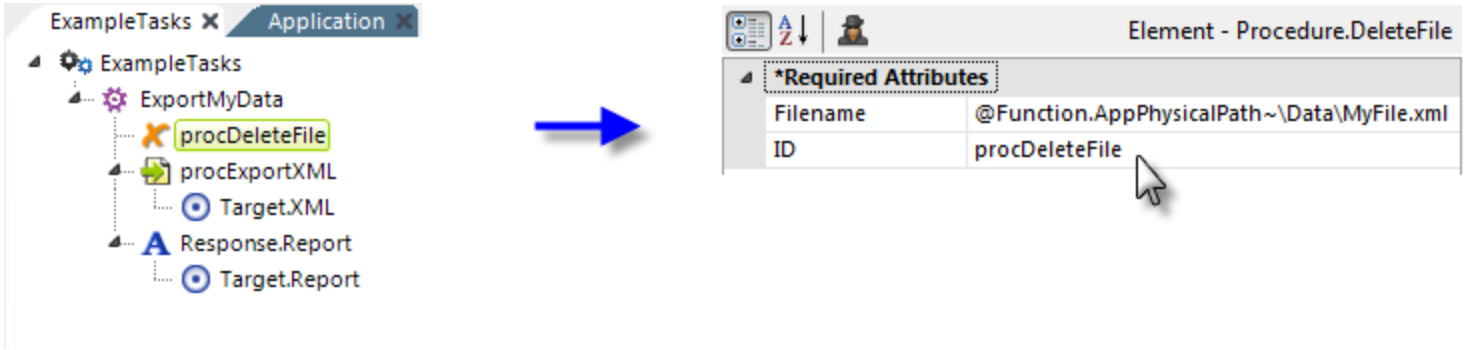



1. As shown above, in your **Process** definition, add a **Task** element and beneath it a **Procedure.Export XML** element.
2. In the element's **Filename** attribute, specify the output path and filename, on the web server, for the exported report. The filename should include the ".XML" file extension. The example shown above uses an @Function token to provide the path to the Logi application's application folder on the web server.




3. Next, add the required **Target.XML** element, as shown above.
4. In the element's **Report Definition File** attribute, specify the report that contains the data to be exported. Do not select the *CurrentReport* option in the list of suggestions as it *will notwork*.
5. Your task will need to be completed, of course, with a **Response** element to run properly.

What happens when your task runs? The specified report data will be exported to a temporary file in your project folder's `rdDownload` folder on the web server; then the temp file will be copied to the output file you specified. If you try to run this task more than once, you'll receive an error. This is due to the fact that **Procedure.Export XML** will not *overwrite* an existing file. Adding a **Procedure.Delete File** element at the beginning of the task will solve this.

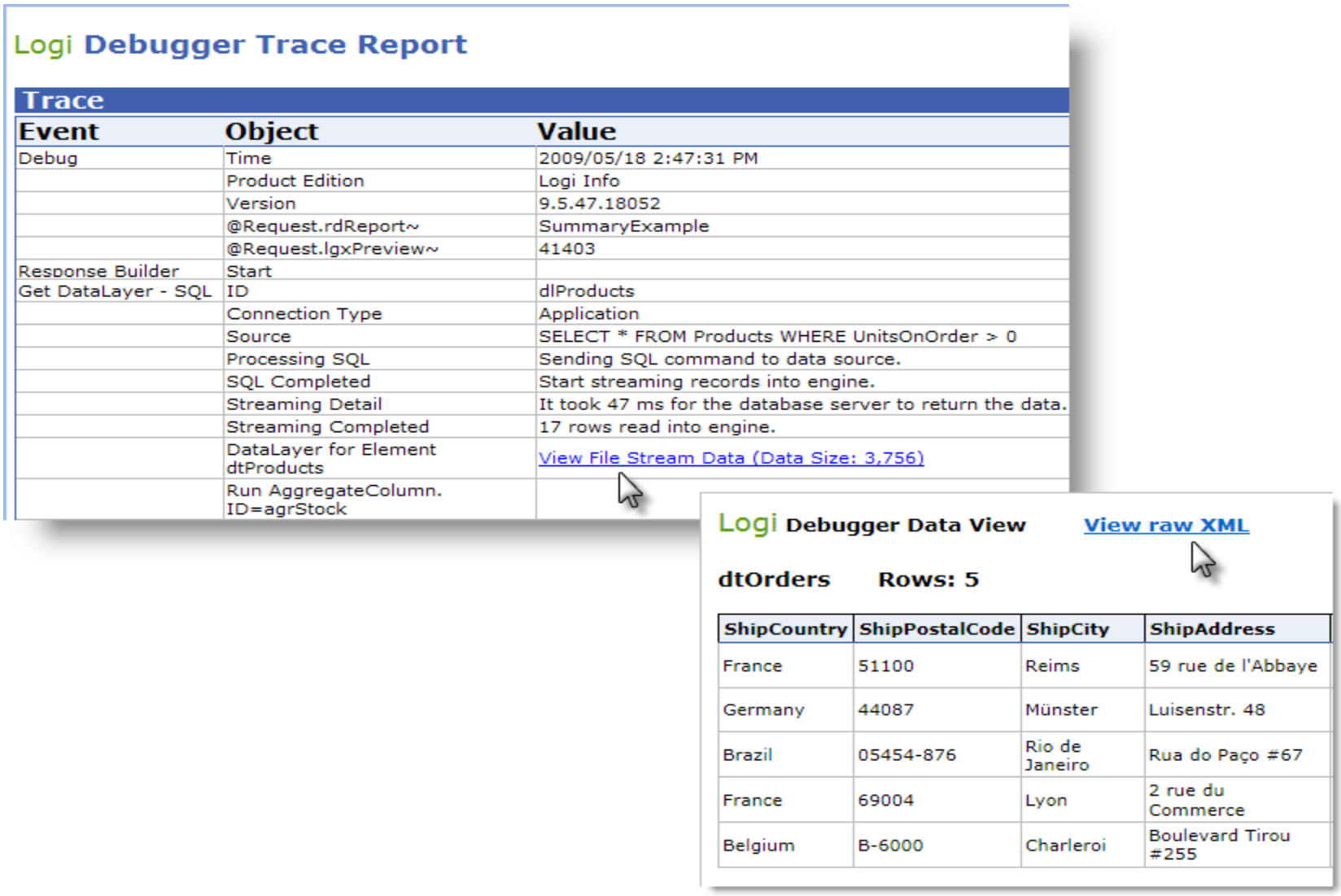



An example of a complete task is shown above. Note the use of the @Function again in the Procedure.Delete File element's **Filename** attribute. Logi products also include an **XSL Transform** element, which allows you to apply transformations to change the XML output format before the data is exported.

# XML - Saving Other Data as XML Data

There are situations in which you may, during development, want to *capture data* from a SQL database or other data source and save it as XML data. For example, it may be desirable to create an XML data file of country names, postal codes, or other static data to be used in an application. Or, it may be useful to work with XML data, without the need for a live database connection. Logi reporting products make it very easy to do this, following these steps:

1. Turn on debugging, using the Debug icon in Studio's toolbar or by configuring your application's _Settings definition, **General** element's **Debugger Style** attribute to *DebuggingLinks* and saving the definition.
2. Run the Logi report that retrieves the desired SQL data or data from another datasource.
3. Click the debug link or icon at the bottom of the report page.



Logi Debugger Trace Report

Trace

| Event | Object | Value |
|---|---|---|
| Debug | Time | 2009/05/18 2:47:31 PM |
| | Product Edition | Logi Info |
| | Version | 9.5.47.18052 |
| | @Request.rdReport~ | SummaryExample |
| | @Request.lgxPreview~ | 41403 |
| Response Builder | Start | |
| Get DataLayer - SQL | ID | dlProducts |
| | Connection Type | Application |
| | Source | SELECT * FROM Products WHERE UnitsOnOrder > 0 |
| | Processing SQL | Sending SQL command to data source. |
| | SQL Completed | Start streaming records into engine. |
| | Streaming Detail | It took 47 ms for the database server to return the data. |
| | Streaming Completed | 17 rows read into engine. |
| | DataLayer for Element dtProducts | View File Stream Data (Data Size: 3,756) |
| | Run AggregateColumn. ID=agrStock | |

Logi Debugger Data View View raw XML

dtOrders Rows: 5

| ShipCountry | ShipPostalCode | ShipCity | ShipAddress |
|---|---|---|---|
| France | 51100 | Reims | 59 rue de l'Abbaye |
| Germany | 44087 | Münster | Luisenstr. 48 |
| Brazil | 05454-876 | Rio de Janeiro | Rua do Paço #67 |
| France | 69004 | Lyon | 2 rue du Commerce |
| Belgium | B-6000 | Charleroi | Boulevard Tirou #255 |

4. As shown above, find the *View File Stream Data* link in the Debugger Trace Report page and click it. This should open the XML in the datalayer in your browser in a Data Table.

5. Click the *View raw XML* link to view the formatted XML data by itself.
6. Right-click in the browser and select *View Source*. This should open the data in **Notepad** or some other text editor.
7. Save the data in the text editor to a file with an .xml extension.

The data is now ready to be used with Logi applications and can be retrieved using DataLayer.XML.

# XML - Export to XML Considerations

The following considerations apply to exports to XML file:

- This is a *data only* export. Only the row-column data in the datalayer is exported. Report headers, footers, and titles, table summary rows and totals, images, etc. are *not* exported.
- Your computer may associate .xml files with Microsoft Excel, which means that Excel will be launched any time you double-click an .xml file. To view these files outside of Excel, try opening them with Notepad or some other text editor.
- The data exported is the data from the datalayer and the XML attributes will be the column names from the datalayer, not the Data Table Column header text.

# Logi Widgets

Logi Widgets are fully-functional **reports** that can be embedded within HTML pages that are **unrelated** to Logi applications. Developers can therefore make the **functionality** of Logi reports available within independent web pages, including blogs.

The following topic guides developers through the process of creating and using widgets:



## About Logi Widgets

The Logi Widget is a Logi report with special properties that uses a special type of definition. In Logi Studio, widget definitions reside in a special Widgets folder in the Application panel. Widgets represent a new approach to **integrating** Logi reports with other web pages and is an excellent alternative to using i-Frames or .NET program integration.

 Our JavaScript-based *Embedded Reports API* is now the recommended technology for embedding Logi reports into other web pages and applications. Logi Widgets continue to be supported for backward compatibility.

When a widget is constructed and then **previewed** in Studio, the report is displayed in the Workspace panel as usual but it's followed by the HTML and JavaScript code needed to embed it in another web page.

```
<FORM> <!-- The FORM element is required if there are INPUT elements in the Widget -->


<!-- Include this once for each page. -->
```

```
<SCRIPT src="http://localhost/testapp/rdTemplate/rdWidget/rdWidget.js"> </SCRIPT>

<!-- Include this for each widget on the page. -->
 <DIV ID="myWidgetDiv" >Loading...</DIV>
<SCRIPT>
var myWidget= new rdLogiWidget;
myWidget.definition="MyWidget";
myWidget.containerID="myWidgetDiv";
myWidget.setParameter("lgxPreview","70555");
myWidget.load();
</SCRIPT>

</FORM>
```

This script, shown in the example above, can be copied and pasted from Logi Studio into any HTML web page in order to display the widget there. The script consists of these four parts:

1. The **form** container. If the widget(s) include any **Input** elements, then the script must be within <FORM></FORM> tags.

2. The **script declaration** statement, within <SCRIPT></SCRIPT> tags. Only one of these statements is needed per HTML page, even if there are multiple widgets on a page.
   The **URL** specified in the script declaration statement must be **valid** and **available**, from a connectivity and security perspective, to any consumers of the widget. In this respect, a Logi application that contains widgets is deployed to a production web server in exactly the same way as any Logi application and this URL must point to the production server. The file specified, "rdWidget.js", contains JavaScript code for facilitating the widgets and is static, i.e. it is not generated or customized for any specific widget.

3. A **division container** for the actual script, one per widget occurrence, within <DIV></DIV> tags. If there are multiple widgets on a page, each division container must be given a **unique ID**.
4. The actual **script statement**, one per widget occurrence, within <SCRIPT></SCRIPT> tags. If there are multiple widgets on a page, the myWidget.containerID value must be set to the appropriate **division ID** value.

## Passing Parameters to the Widget

The myWidget.setParameter function is used to specify parameter **name-value** pairs, if any, to be passed to the widget, and you can add multiple function calls as desired. In the example above, the widget will be called with a **Request** variable named "lgxPreview" that has a value of "70555", appearing in the query string as "&lgxPreview=70555". This was included in the example because the widget was previewed in Studio but is not necessary in your production code. Parameters passed in this manner are available in the widget definition as **@Request** tokens and can be used to pass in security parameters, variables, etc.

The values passed in a parameter can be hard-coded:

```
myWidget.setParameter("UserName","JHarlan");
```

or set using a Token:

```
myWidget.setParameter("ReportDate","@Function.Date~");
```

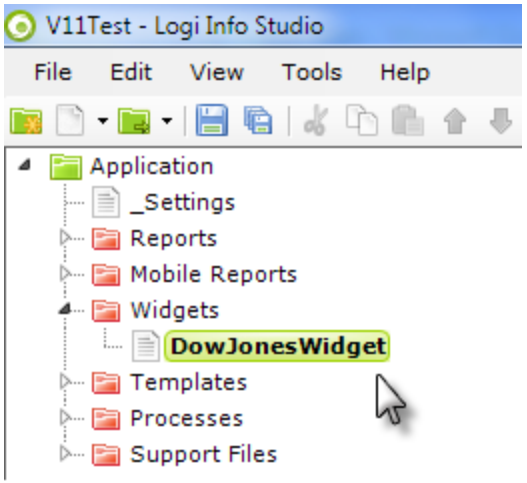
or returned from a separate JavaScript, PHP, or other script function in the HTML page:

```
myWidget.setParameter("UserName",<?php GetUser();?>");
```

As we have seen, the HTML and JavaScript required to **integrate** a widget into an HTML page is **simple** and **straightforward**. In "Creating a Widget Definition" on the next page, we'll explore how a widget definition is created.

# Creating a Widget Definition

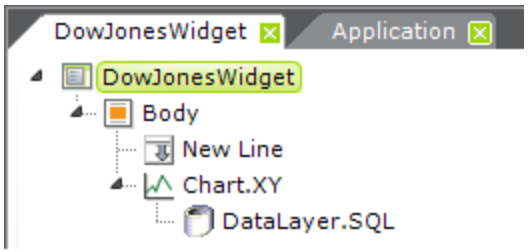
A widget definition is simply a different **category** of Logi report definition. In Logi Studio, these definitions appear in the **Application** panel in the Widgets folder.




To create a new widget, right-click the **Widgets** folder and add a new definition. The new file, with an .lgx extension, will be created in the file system in this folder (the folder is not created until you add your first widget to the application):

```
<LogiApplicationFolder>\_Definitions\_Widgets
```

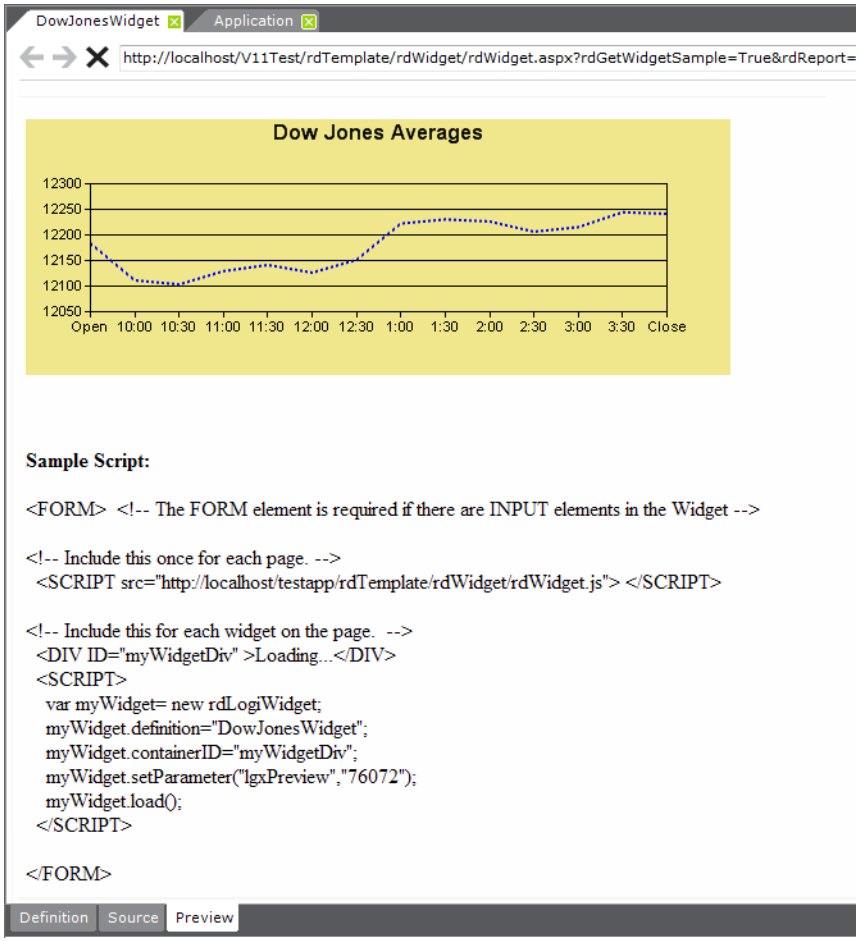
The widget definition can now be opened and edited in the **Workspace** panel, just like any report definition.




In the example above, a line chart that gets its data from an XML data file has been added to the widget. While there are some elements that are excluded from use in a widget, generally most **elements** can be used. Widgets can be fully dynamic and support drill-downs, drill-through's, and links to other pages or applications. Sorting and paging of tables is also supported. Some of the limitations in widgets themselves can be overcome by providing a **link** in the widget to a **Logi application** page that provides the desired functionality.

# Embedding the Widget Code in an HTML Page

Once your have your widget definition created within Studio, the next step is to **Preview** it:



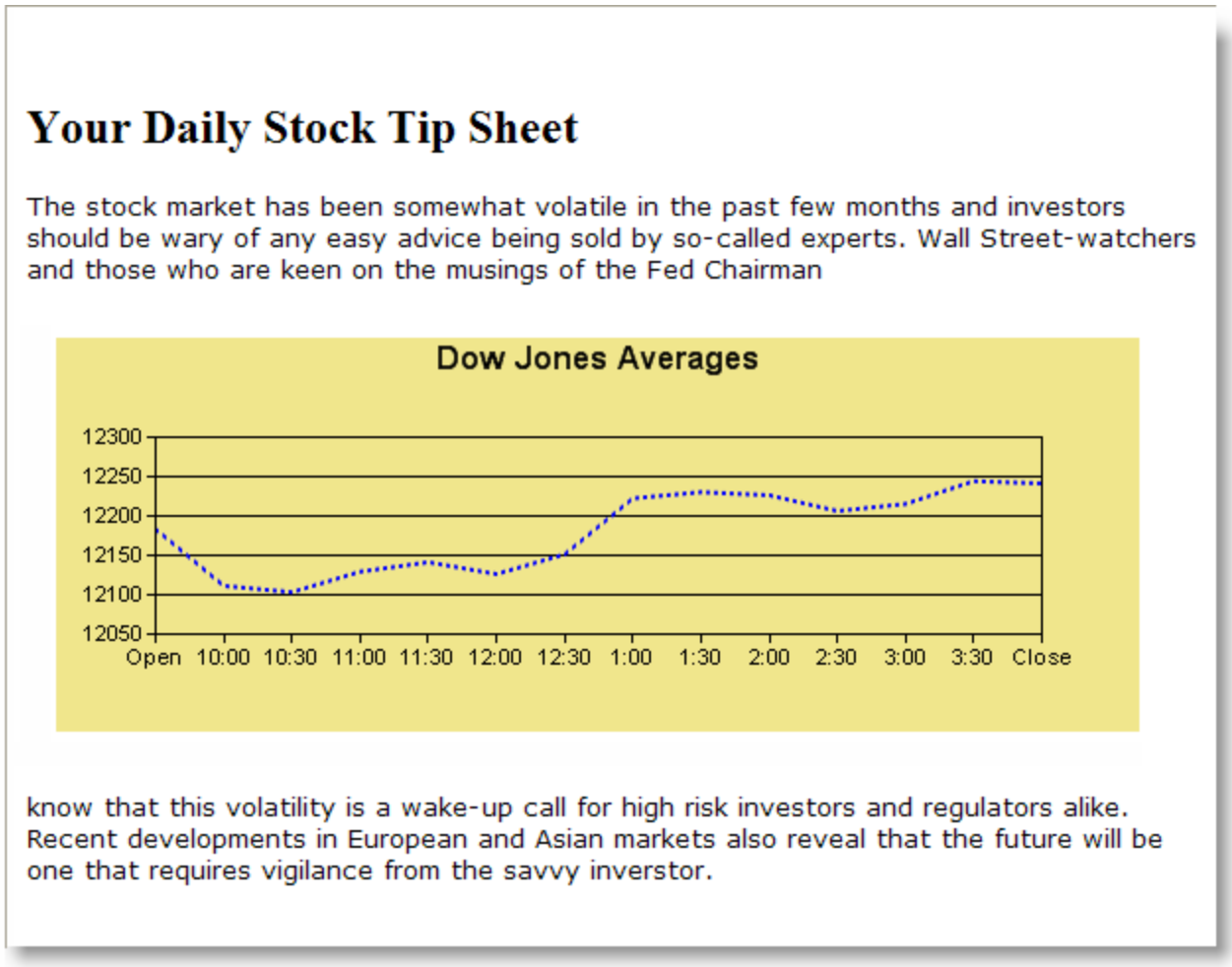

As shown above, the widget will be displayed in the Preview tab and, below it, the **HTML and JavaScript** necessary to implement it will be displayed.

You can work with the widget as you would with a regular Logi report, refining and previewing it until satisfied. **Debugging links** can be turned on and will function as usual. When the widget is ready, preview it and then, right in the Preview tab, select and copy the sample script and paste it into the target **external HTML** page.

```
<html>
<head>
<title>Your Daily Stock Tip Sheet</title>
</head><body bgcolor="white" text="black" link="blue" vlink="purple" alink="red">
<p> </p>
<h2>Your Daily Stock Tip Sheet</h2>
<p><font face="Verdana"><span style="font-size:10pt;">The stock market has been somewhat volatile in the past few
months and investors should be wary of any easy advice being sold by so-called experts. Wall Street-watchers and those
who are keen on the musings of the
 Fed Chairman</span></font><br><!-- Include this once for each page. -->
<SCRIPT src="http://localhost/testapp/rdTemplate/rdWidget/rdWidget.js"> </SCRIPT><!-- Include this for each widget on
the page. -->
<DIV ID="myWidgetDiv">Loading...</DIV>
<SCRIPT>
 var myWidget= new rdLogiWidget;
myWidget.definition="DowJonesWidget";
myWidget.containerID="myWidgetDiv";
myWidget.setParameter("lgxPreview","57952");
myWidget.load();
</SCRIPT><br><font face="Verdana"><span style="font-size:10pt;">know that this volatility is a
wake-up call for high risk investors and regulators alike. Recent developments in European
 and Asian markets also reveal that the future will be one that requires vigilance from the savvy inverstor.
```

```
</span></font><p> </p>
<p> </p>
</body>
</html>
```

As shown above, the code has been **pasted** into an HTML page. The example widget does not include any Input elements, so no <FORM> tags are required.




When displayed, as shown above, the widget is embedded in the HTML page and shows the data.

# Multiple Widgets on a Page

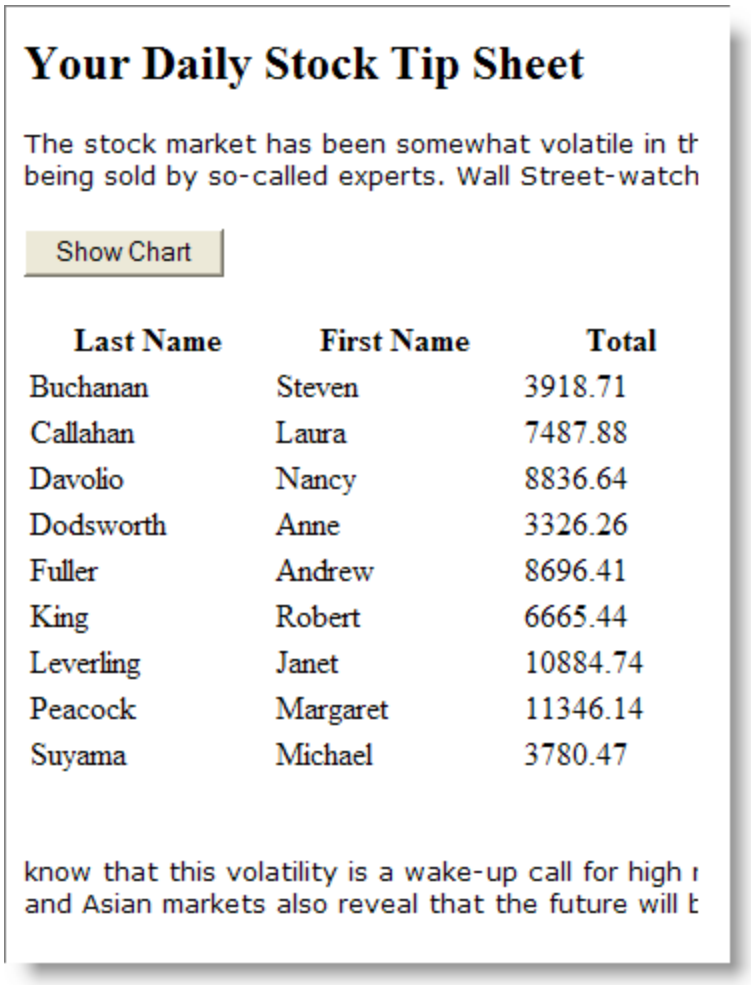
To add **multiple widgets** to the page and make them all visible at the same time, you will need separate <DIV> statements in your HTML for each widget. Their scripts should then be combined within the <SCRIPT> tags.

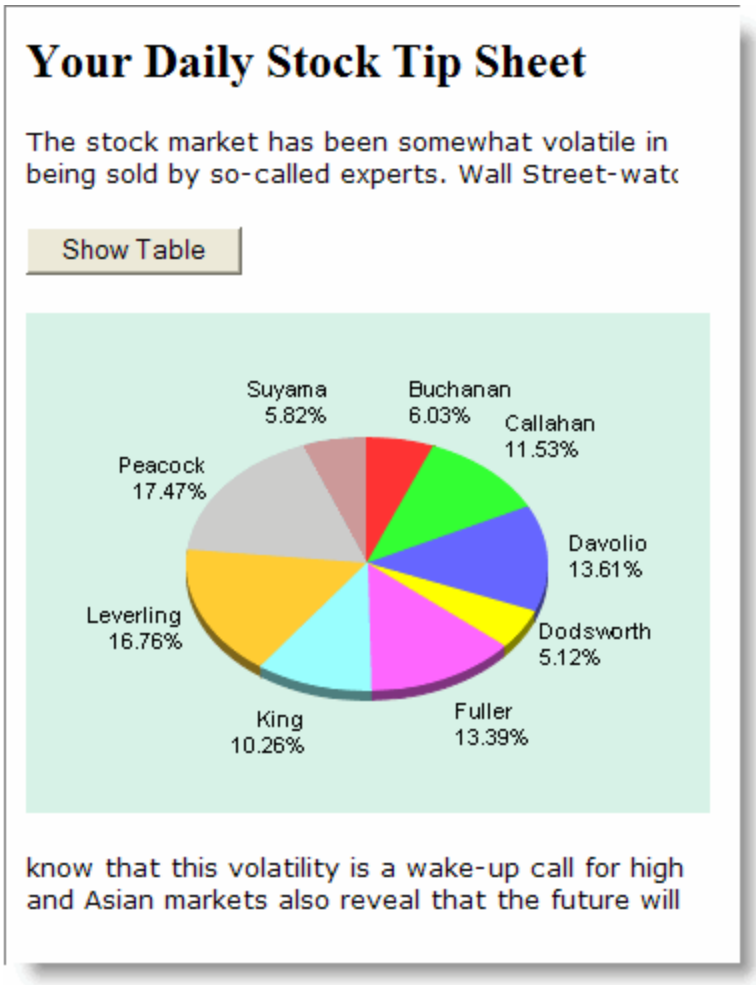
```
<html>
...(HTML removed for brevity in this exsample)
<!-- Include this once for each page. -->
<SCRIPT src="http://localhost/testapp/rdTemplate/rdWidget/rdWidget.js"> </SCRIPT><!-- Include this for each widget on the page. -->
<DIV ID="myWidgetDiv1">Loading...</DIV>
<BR></BR>
<DIV ID="myWidgetDiv2">Loading...</DIV>
<SCRIPT>
 var myWidget1= new rdLogiWidget;
myWidget1.definition="Table";
myWidget1.containerID="myWidgetDiv1";
myWidget1.load();
var myWidget2= new rdLogiWidget;
myWidget2.definition="Charts";
myWidget2.containerID="myWidgetDiv2";
myWidget2.load();
</SCRIPT>...(HTML removed for brevity in this exsample)
</html>
```

The example above shows the code to include two widgets, one below the other, at the same time on the HTML page (the resulting page is not shown here).

## Switching Between Widgets

It's also possible to add multiple widgets to a page and display them **one-at-a-time**, using **links** or **buttons** on the widgets themselves. To do this, use **Action.Widget** and **Target.Widget** elements in your widget definition, to call the next widget to be shown. You only need to add the script for the **first widget** to your HTML page; the Action.Widget element will cause other widgets to be displayed in the same <DIV> as the first one.




Only one of the two widgets shown above appears at a time. They alternate being visible when the button is clicked.

```
<SCRIPT>
 var myWidget1= new rdLogiWidget;
myWidget1.serverURL = "http://server1/widgetFarm";
```

```
myWidget1.definition="Table";
myWidget1.containerID="myWidgetDiv1";
myWidget1.load();
var myWidget2= new rdLogiWidget;
myWidget2.serverURL = "http://server2/widgetsRUS";
myWidget2.definition="Charts";
myWidget2.containerID="myWidgetDiv2";
myWidget2.load();
</SCRIPT>
```

If multiple widgets on the page are from *different* servers, be sure to include the server URLs, shown above in red, by setting the `serverURL` property.

# Current Widget Code in an HTML Page

There are certain **limitations** for widgets and not all Logi report elements are available for use with them. As our Logi Widget technology continues to evolve, the following limitations may be reduced. These elements *cannot* be used with widgets:

- Action.AppDev
- Action.Exit
- Action.Export (all varieties)
- Action.GoogleSpreadsheet
- Action.Process
- Action.RefreshElement
- Action.Template
- Analysis Chart
- Analysis Grid
- Dashboard
- Interactive Data View
- Google Map
- Input Date
- Input File Upload
- OLAP Grid
- Procedure (all varieties)
- Security Filter
- Shapes (all varieties)
- Sub-Report
- Tabs
- UserRoles (all varieties)

The unsupported "super elements", including Analysis Chart, Analysis Grid, Dashboard, IDV, and Google Map, can be used by placing them in separate Logi application pages and accessing them via a drill-through link on the widget.

Widgets *cannot* use **shared elements** that are defined in other definition types. For example, shared elements created in a report definition are not available for use in a widget definition.

# Widgets and Load Balancing

Because they're actually embeddable JavaScript code, Widget definitions behave slightly differently than standard report definitions: their images are temporarily stored on the web server in a browser-accessible directory, which is the rdDownload folder.

This can be tricky when used with load-balancing, since the location (rdDownload) is specific to the individual server. It can be problematic for round-robin load-balance scenarios and may depend on the type of load balancer you're using and what type of persistence there is, if any, across unique requests. The only certain way to ensure problem-free operation is to enable "sticky sessions", or session affinity, for the load balancer. In this case, there would never be a file/request conflict or missing file error.

If you don't use complete session affinity, it may be possible to configure the load balancer "sticky time", also called "persistence", so that, for any request/session duration, you attach to one server node to ensure the operations that take longer to happen, or that require multiple requests, don't automatically jump to the next node. That means the user is persistent or sticky for a given time duration, and then can be sent to a different server for requests outside of the time duration. This setup would allow for multiple sequential requests to be made to the same server.

More information about load-balancing in general can be found in *Load Balancing Configuration*.

# Sample Definition

The following sample widget definition has been included to get you started with your first Logi Widget.

1. Launch Logi Studio, create a new **Widget** definition file, and give it a new name.
2. Open the new file in the **Workspace** and click its **Source** tab at bottom of the panel.
3. Copy the text from this page to your clipboard and paste it into the Source tab in Studio, within the <Widget> </Widget> tags.
4. Preview the widget.

```
<Body>
<LineBreak LineCount="2" />
<Chart Type="XY" ChartDataColumn="Avg" ChartHeight="200" ChartWidth="550" XYChartType="Line" ChartLabelColumn="Time" LineStyle-
e="DotLine" LineWidth="2" ShowDataValues="False" BackgroundColor="Khaki" BorderBottom="50" BorderLeft="50" BorderRight="50"
BorderTop="50" Color="Blue" ChartTitle="Dow Jones Averages">
<DataLayer Type="Static" ID="dlDowJones">
<StaticDataRow Time="Open" Avg="12182" />
<StaticDataRow Time="10:00" Avg="12110" />
<StaticDataRow Time="10:30" Avg="12102" />
<StaticDataRow Time="11:00" Avg="12128" />
<StaticDataRow Time="11:30" Avg="12140" />
<StaticDataRow Time="12:00" Avg="12125" />
<StaticDataRow Time="12:30" Avg="12150" />
<StaticDataRow Time="1:00" Avg="12221" />
<StaticDataRow Time="1:30" Avg="12229" />
<StaticDataRow Time="2:00" Avg="12225" />
<StaticDataRow Time="2:30" Avg="12205" />
<StaticDataRow Time="3:00" Avg="12214" />
```

```
<StaticDataRow Time="3:30" Avg="12243" />
<StaticDataRow Time="Close" Avg="12240" />
</DataLayer>
</Chart>
</Body>
```

Now copy the widget script from the Preview panel into an HTML page created outside of Studio and experiment.

keywords: integrate, integration

# The Mobile Dashboard

The **Mobile Dashboard** element is used to provide many of the Dashboard features available in a regular Logi Info report to a Logi mobile report.

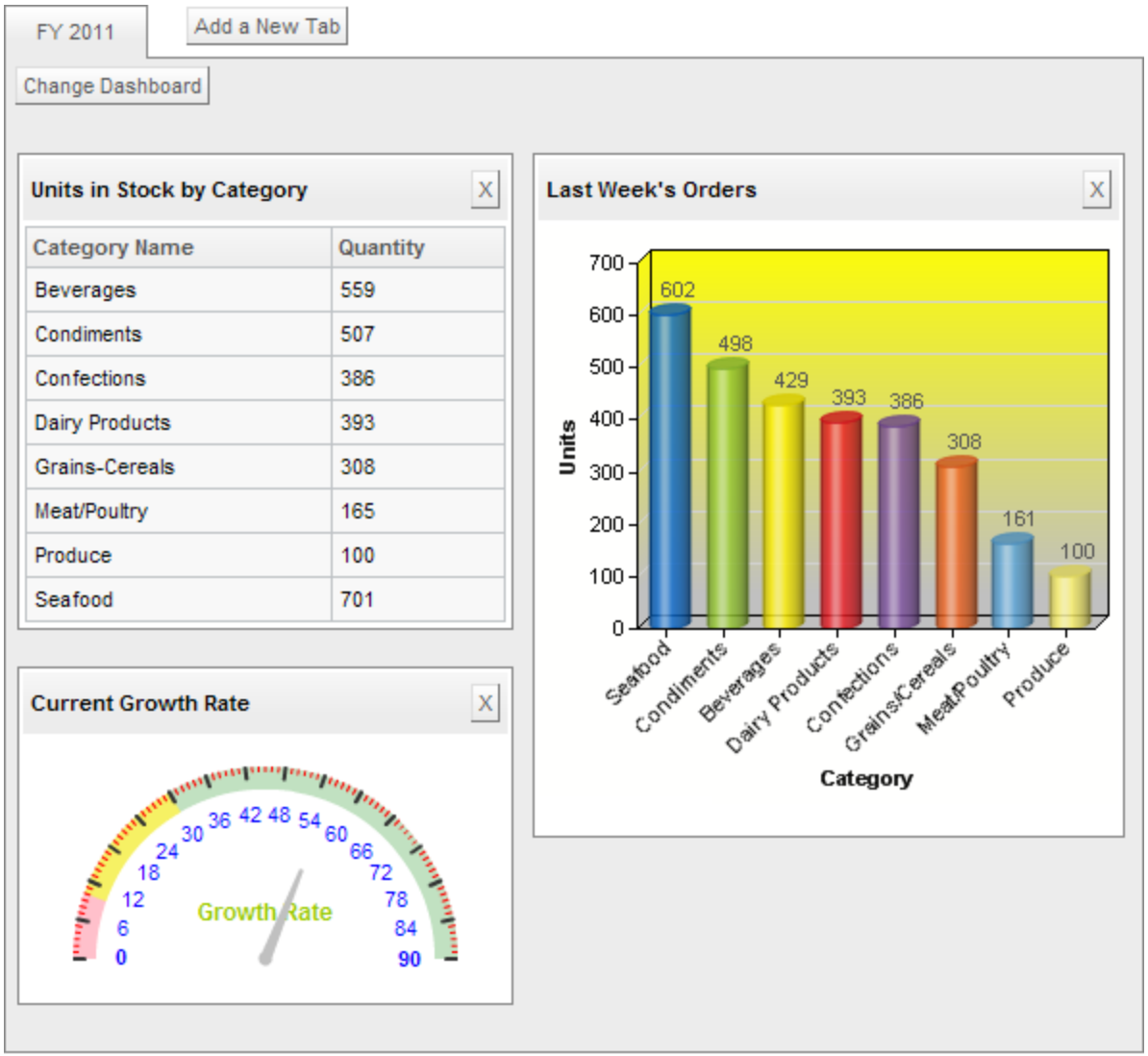


## About the Mobile Dashboard

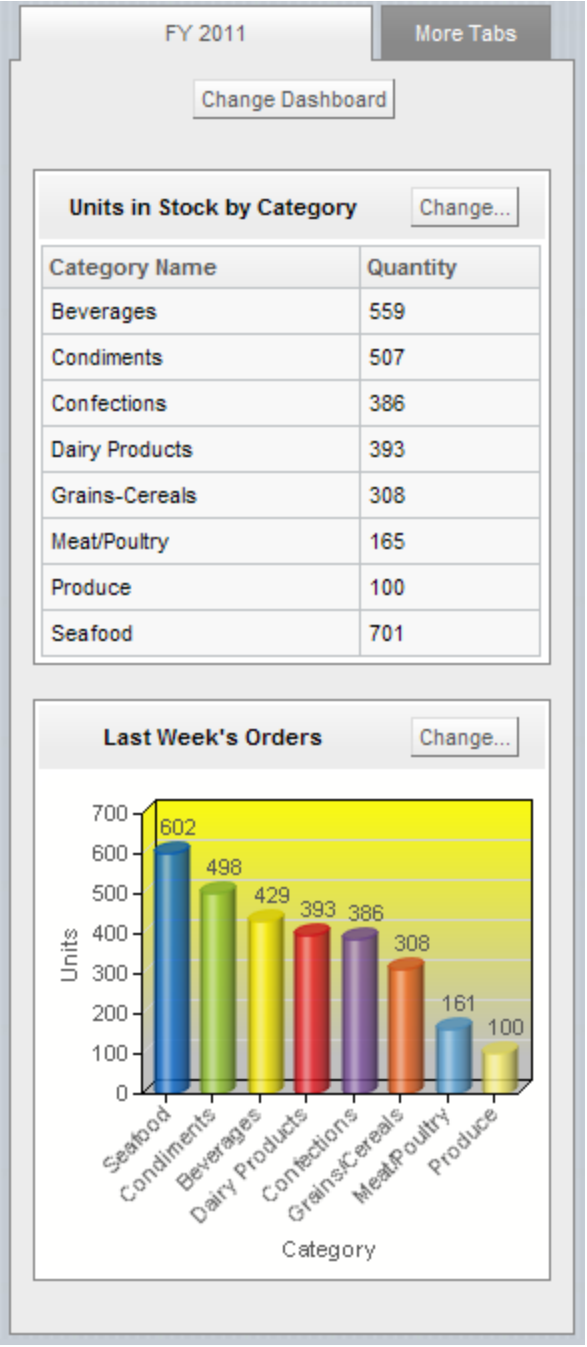
The **Mobile Dashboard** element allows mobile report developers to "share" a Dashboard created in a regular Logi report definition, using the **Dashboard** element. The Mobile Dashboard applies special formatting and functionality to the visualizations in the existing regular Dashboard. This approach allows you to build and test a Dashboard in Studio before bringing it into the mobile device realm. It also allows you to leverage existing reports that include Dashboards, without having to recreate them in a separate mobile report definition.

Dashboards are frequently used to make it easy for users to see a broad view of a variety of information. A Dashboard is a collection of panels, customizable by the user at runtime by adding, re-arranging, or deleting panels.

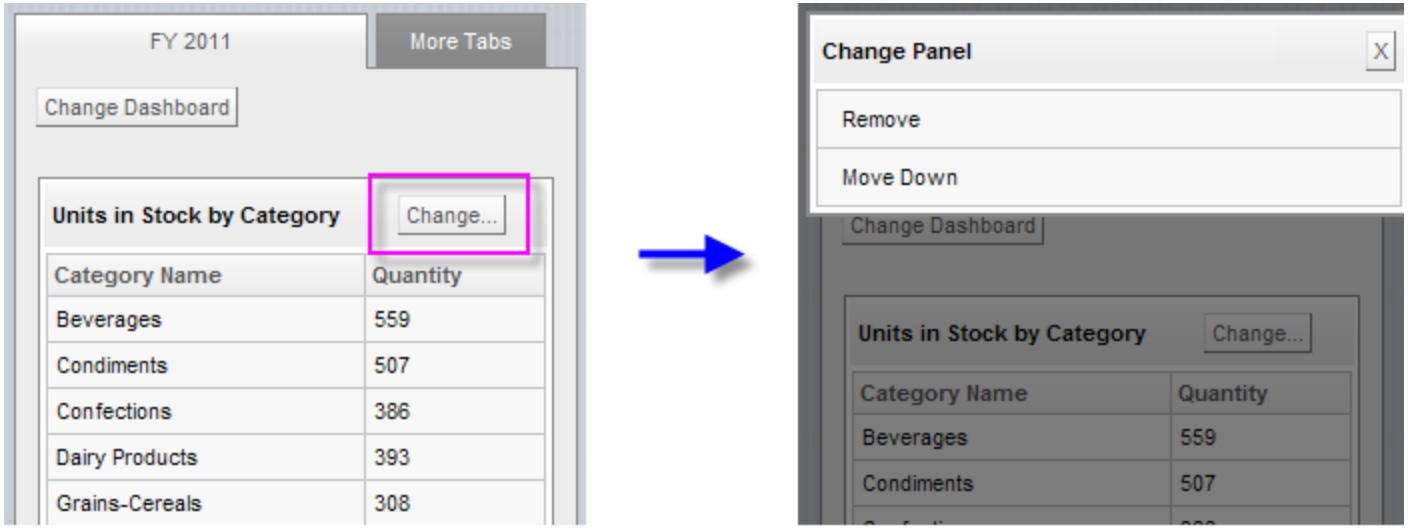
Let's look at some examples:




The example above shows a regular Dashboard, as it appears in a regular **Logi report**, with one tab and three panels.


FY 2011
More Tabs
Change Dashboard
Units in Stock by Category
Change...
Category Name
Quantity
Beverages
559
Condiments
507
Confections
386
Dairy Products
393
Grains-Cereals
308
Meat/Poultry
165
Produce
100
Seafood
701
Last Week's Orders
Change...
700
600
500
400
300
200
100
0
Units
602
498
429
393
386
308
161
100
Seafood
Condiments
Beverages
Dairy Products
Confections
Grains/Cereals
Meat/Poultry
Produce
Category

The example above shows that same Dashboard shared as a mobile Dashboard in a **Logi mobile report**. In a mobile Dashboard, panels are displayed in a single column. If the regular Dashboard being shared includes tabs, the mobile Dashboard will too, but their visibility will be restricted to a "current" tab and a navigation tab which is used to access other tabs.

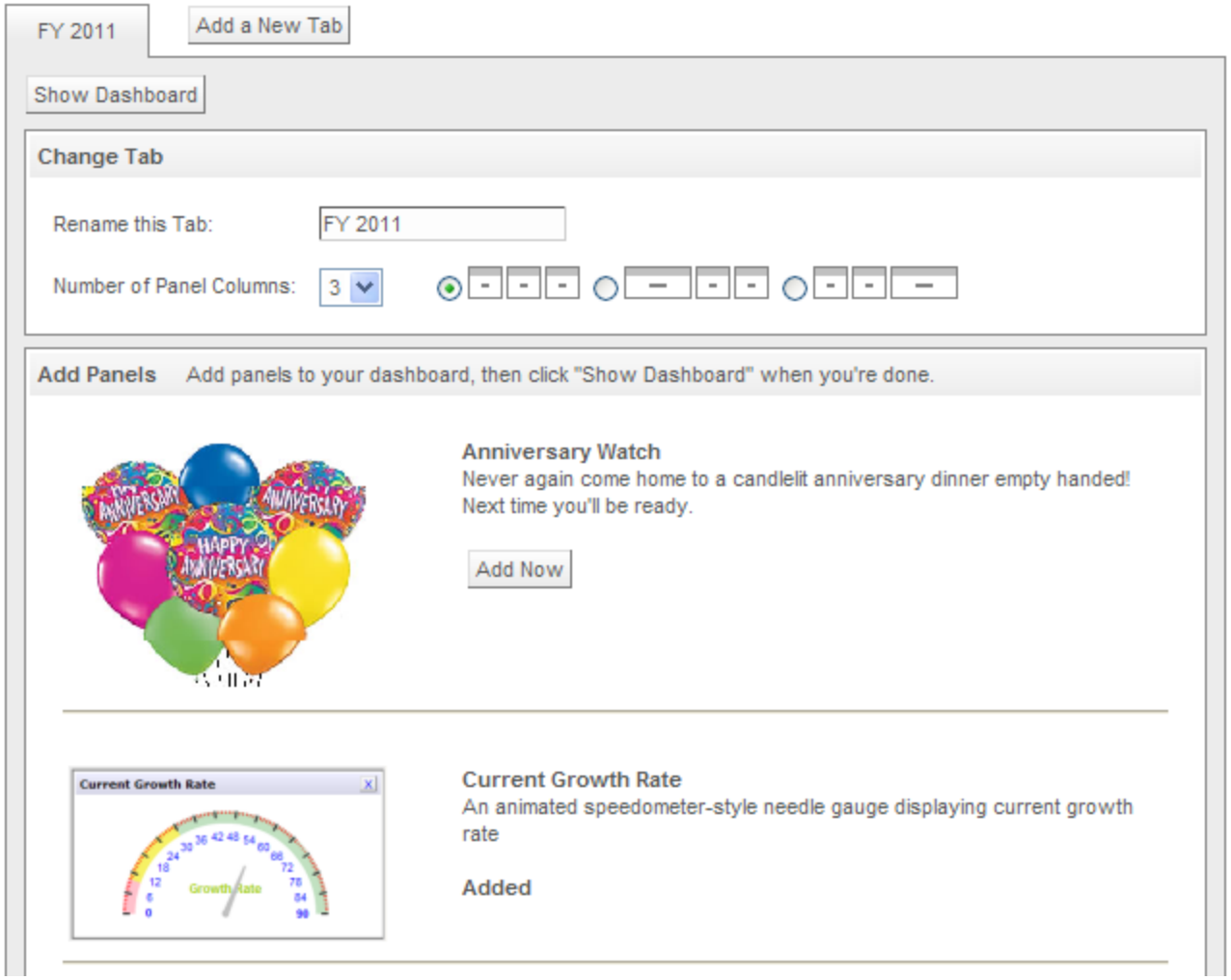



If the shared regular Dashboard's **Dashboard Adjustable** attribute has been set to *True*, then Dashboard panels can be removed or rearranged in the mobile Dashboard by tapping the Change... button in a panel. As shown above, a modal "change panel" will then be displayed with panel management options. Tap the desired option to effect a change. Tap the X button to close the change panel.

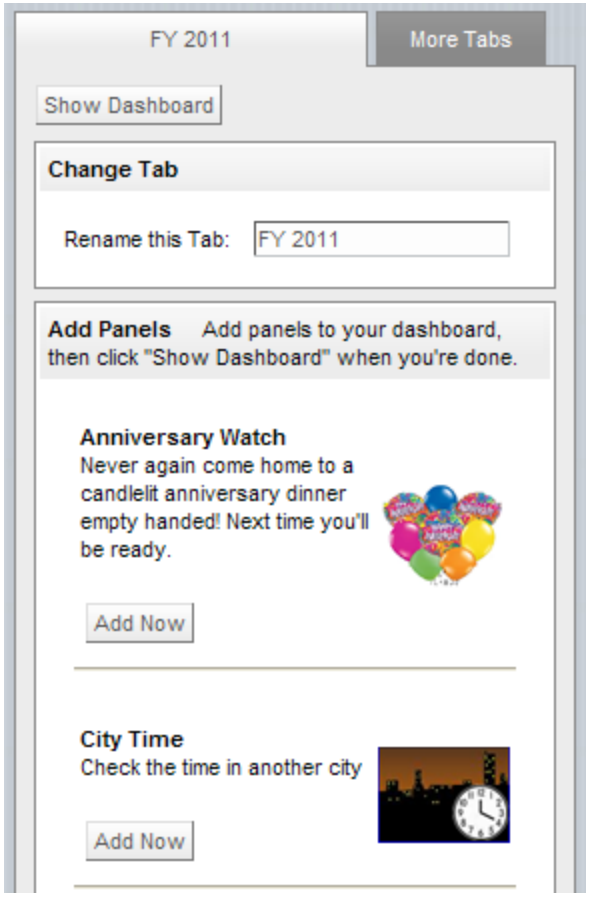
But wait... the original Dashboard had *three* panels but the mobile Dashboard is only showing *two* of them. Where's the "Current Growth Rate" panel?

| | | |
|---|---|---|
| Load DefinitionModifierFile | For Dashboard | E:\Samples\SampleMobileReporting\rdT |
| | Done | |
| | ** WARNING ** Dashboard panel automatically removed because of an element not supported in Mobile Reports. | Unsupported element: AnimatedGauge |

That panel includes an Animated Gauge, an element that's not supported in Mobile Reports in the mobile Dashboard in earlier releases. When this happens, the Debug Trace page (shown above) displays an explanation. At the current time, Animated Charts, Animated Gauges, Input Sliders and Heat Maps are *not* available in mobile Dashboards.




The example above shows the **Dashboard Configuration Page** for a regular Dashboard. If the Dashboard Adjustable attribute has been set to *True* by the developer, at runtime users can manage the Dashboard's tabs and panels here.




The same functionality is available in the mobile Dashboard, as shown above, in a slightly different format. Tapping the Change Dashboard and Show Dashboard buttons will show and hide the configuration page. Panels containing unsupported elements will not appear in the list of panels that can be added to the Dashboard.

Customizations for each user can be shared from session to session, using the SaveFile features of the shared Dashboard. Layout changes made in the regular Dashboard will not appear in a related Mobile Dashboard until the SaveFile is updated.

## Attributes

The Mobile Dashboard element has the following attributes:

| Attribute | Description |
|---|---|
| ID | (Required) The unique element name. |
| Dashboard Definition File | (Required) The name of the regular report definition file that contains the Dashboard element to be shared in the mobile report. |

# HTML Tables

Basic **HTML Table** structures are one method of organizing and laying out content in Logi application report pages. These tables provide a consistent arrangement of rows and columns that can contain text, data, charts, etc.

The following topics discuss the use of HTML tables:

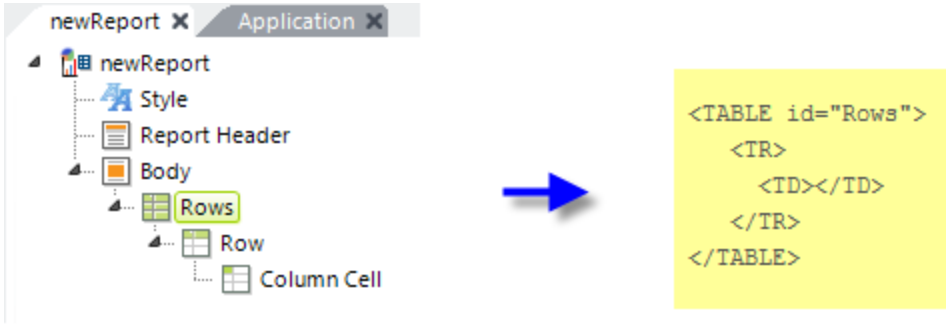
- Using Nested Tables
- Working with Automatic Table Layouts
- Merging Table Cells

## About HTML Tables

An HTML table consists of a collection of **rows** and **columns**; the intersection of a row and a column is called a "cell". A single row contains one cell for each column. HTML tables in Logi applications follow the HTML standard specifications: rows must contain at least one column cell to display content within the table.

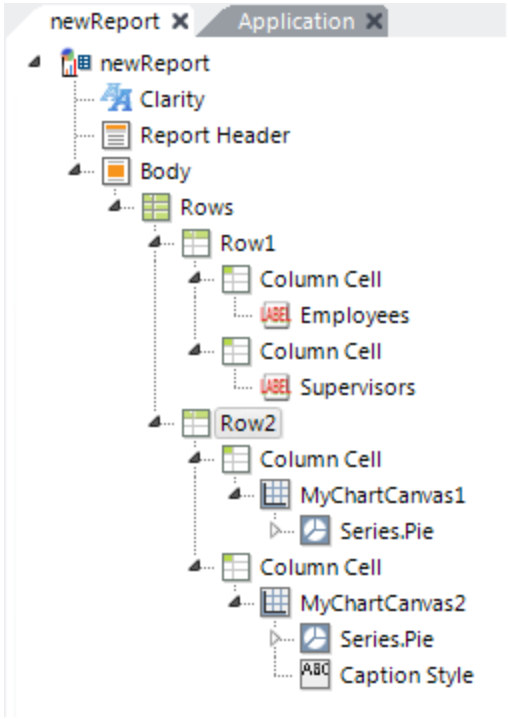
HTML tables in Logi Studio are created using one or more **Rows**, **Row**, and **Column Cell** elements.

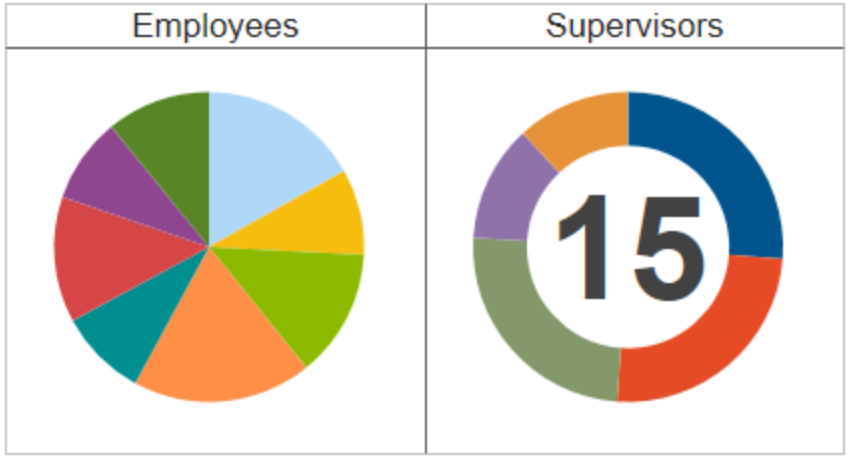
Don't confuse HTML tables with Logi *Data Tables*. They're different structures, created using different sets of elements.




At runtime, as shown above, the Rows element produces an HTML `<TABLE>` tag set in the report output, the Row element produces a child `<TR>` tag set, and the Column Cell element produces a child `<TD>` tag set.

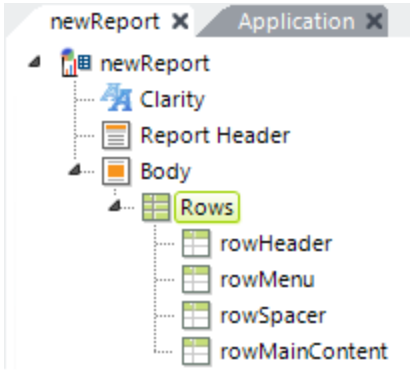
HTMLtables are very useful for spacing report content *horizontally*. Report content can be added to any **Column Cell** element and any number of Column Cell elements can exist in a single row.




In the example shown above, a basic table structure creates a 2 x 2 layout that aligns two charts with their titles.

Generally, all rows must have the same number of columns in them and the first, or top, row sets the number and sizes of columns for the rest of the table. For special situations about *merging* and *spanning* cells, see "Merging Table Cells" on page 426



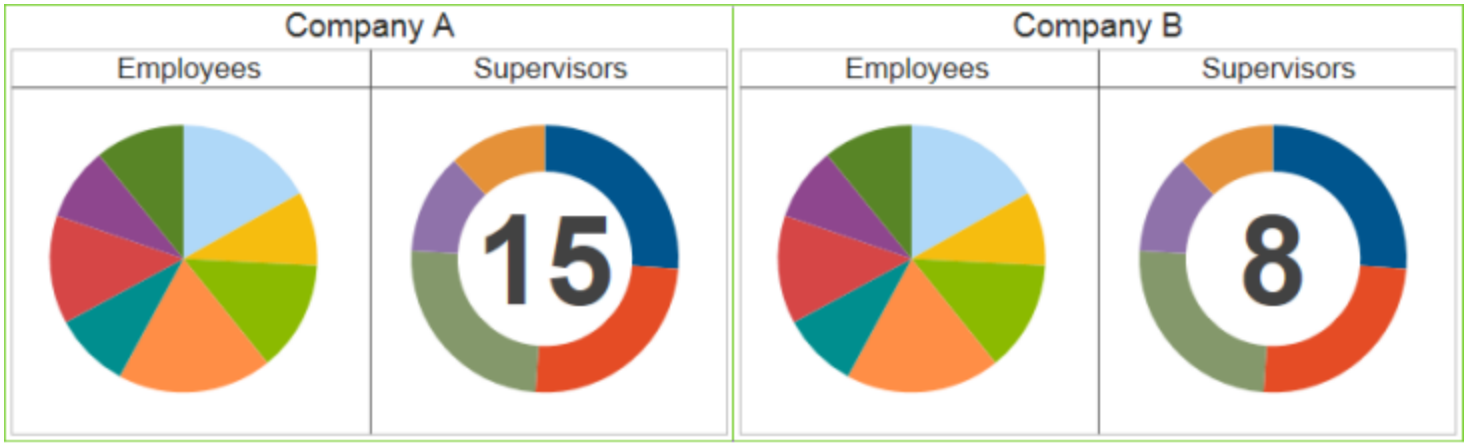

Tables can also be used to provide overall formatting for an *entire* report page. In the example shown above, the definition includes four rows: one each for header, menu, a spacer, and the main page content.

## Responsive Design Elements

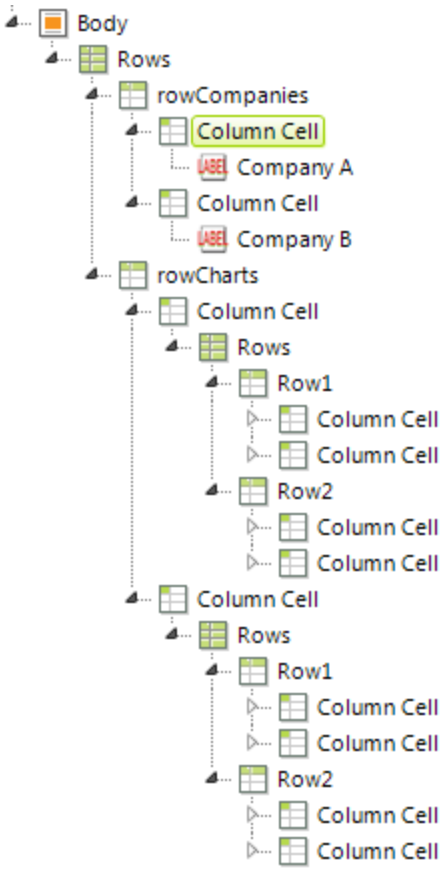
Responsive Design elements, including **Responsive Row** and **Responsive Column**, are available. These row and column elements may provide an improved viewing experience; they can dynamically re-arrange their change their size and arrangement depending on the viewport size. For more information, see *Responsive Design Elements*.

# Using Nested Tables

A *nested table* is a table that is completely enclosed by another table. Nested tables provide you with a means to structure report content even further.




Column Cell elements can contain their own *separate* HTML tables. In the example shown above, the parent table has two columns with green borders and the tables nested within each column have gray borders.



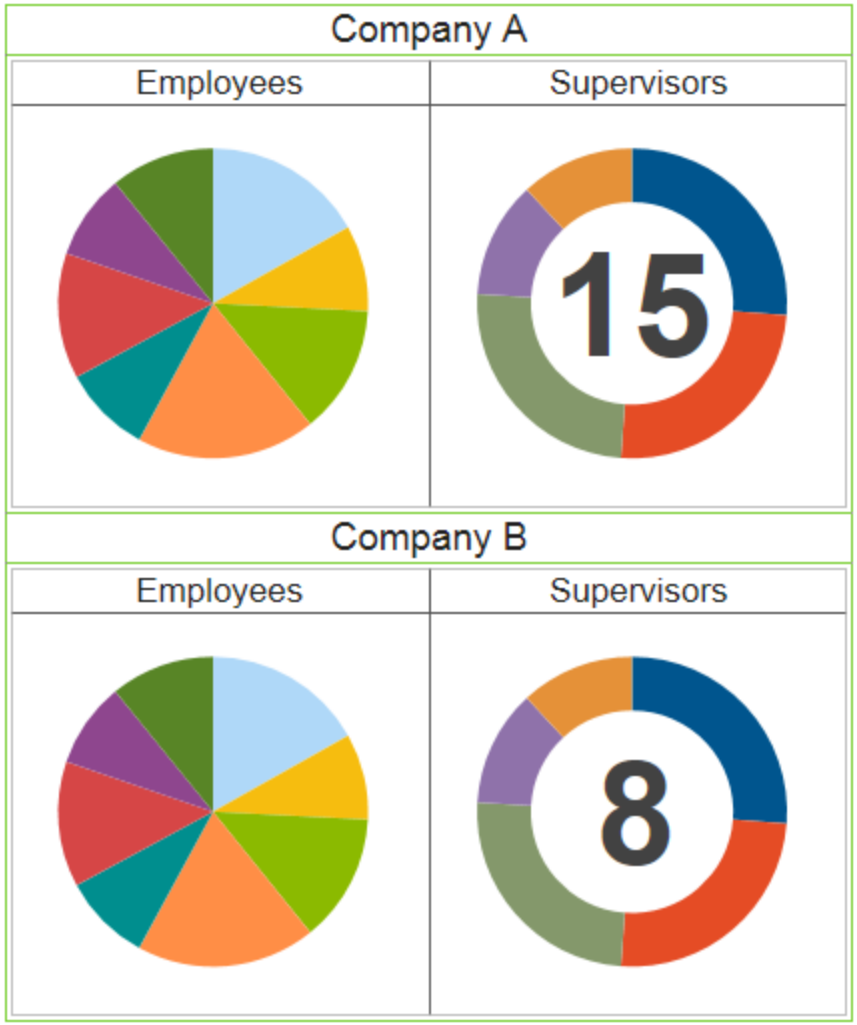

Here's the element arrangement used to create the example. Nested tables allow for a further degree of layout control on your report page.

# Work with Automatic Table Layouts

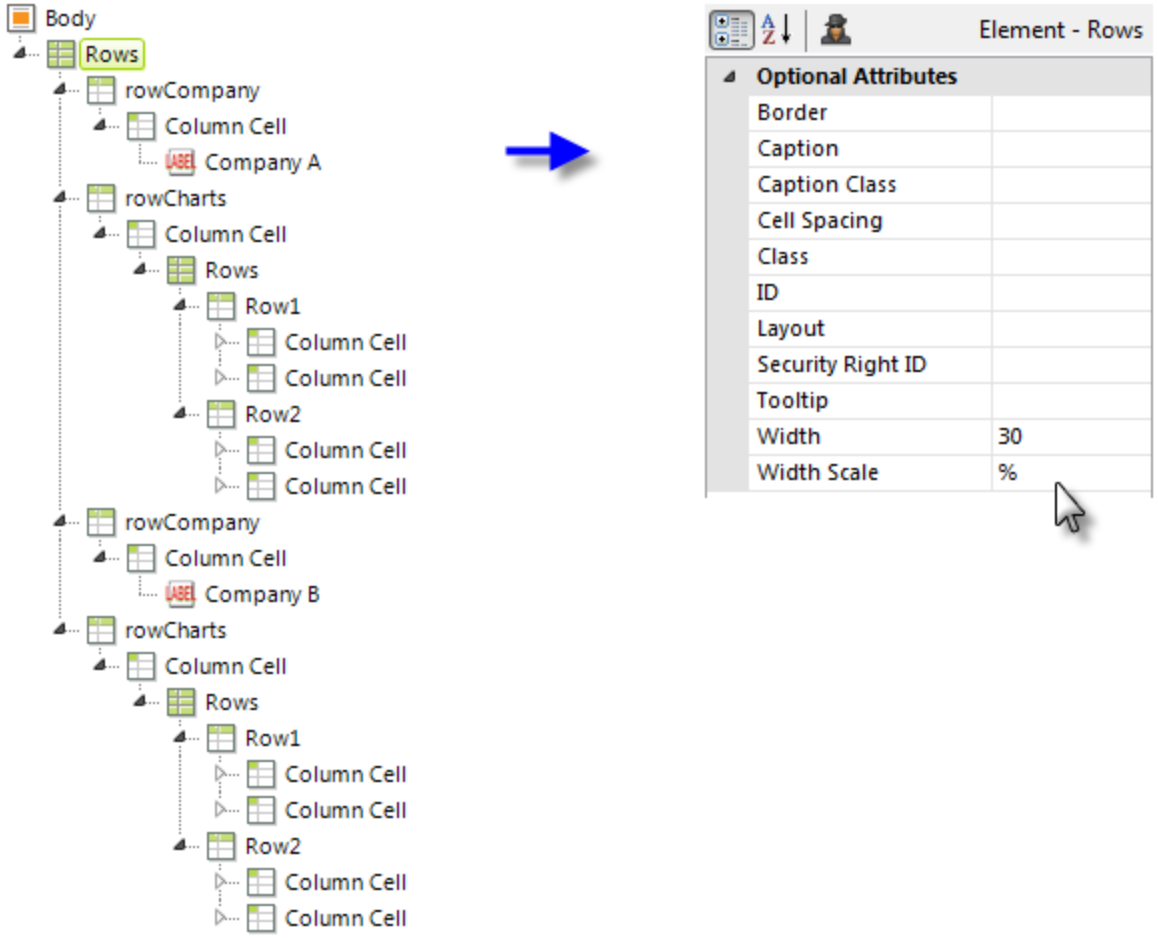
In general, HTML table layout is concerned with size, alignment, and spanning. The layout type is configured by setting the Rows element's **Layout** attribute to *Auto (*the default) or *Fixed.*

Automatic table layouts rely on the web browser's ability to automatically calculate the column widths required to fit its contents, while fixed layouts restrict column widths to the values you specify.

There's even a hybrid mode: when using automatic layouts you can specify the size of *some* columns and let the browser figure out the rest. Column widths, when specified, can be an exact number of pixels or percentages of the overall table width.




In the example above, the parent table's width is set to 30% of the total browser window's width. If the **Width** attribute value is too small, the browser will automatically calculate the amount of space needed to display the table's contents on a single line.




The element arrangement used to create the previous example is shown above.

In general, the three elements used with tables, **Rows**, **Row**, and **Column Cell**, are all "container" elements and all have a **Class** attribute that allows you to apply style classes to them (and to the elements they contain). This can very useful when formatting

content. For example, a table column can have style applied to it that centers everything within the column, sets a default font size, or creates padding that provides a small area of white space between the edges of the column and its content.

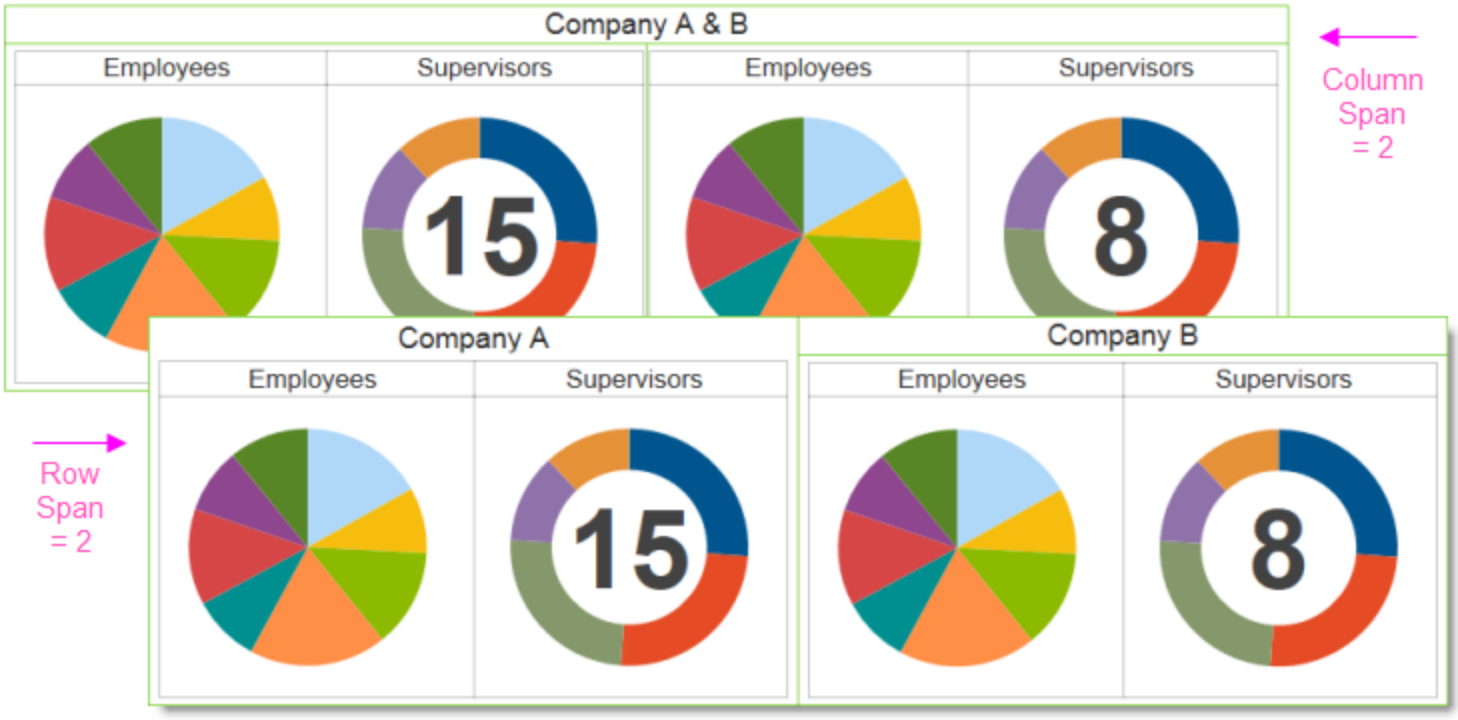
Table alignment is controlled by style sheet classes when automatic layouts are used. You can use the *text-align* and *vertical-align* style attributes to align table contents. Define a *class* selector or *ID* selector class to align a specific table. Logi reporting products support CSS and you're encouraged to use style classes to control the layout and appearance of report content when it makes sense to do so.

```
TABLE {
 vertical-align: middle;
 text-align: center;
}

TR {
 vertical-align: middle;
 text-align: center;
}

TD {
 vertical-align: middle;
 text-align: center;
}
```

When using element selector classes, you can define a class for any of the three HTML tags that comprise a table, as shown above. You can control overall table height by defining the *height* property in a style class.

# Merging Table Cells

Merging table cells, also known as *cell spanning*, is a great way to control table layout even further. Cell spanning merges multiple cells together horizontally or vertically. A vertical span is called *row span* and a horizontal span is called *column span*. The **Row Span** and **Column Span** attributes are only available for the Column Cell element. Set these attributes to a numeric value that specifies the number of rows or columns to be spanned.




Setting a column or row span value in excess of the total physical columns and rows present may cause the table to stretch horizontally or vertically. Remove extra Column Cell elements to fix alignment issues when using column and row span.

# Insert HTML into Reports

The Logi Server Engine generates HTML and JavaScript when it processes a Logi definition, but you may find that you need to insert your own HTML code directly into a Logi report.

The following topics discuss the elements available to insert HTML:



## About Inserting HTML

The "source code" of a Logi definition is an XML document, which is processed by the Logi Server Engine, and output as HTML. However, you may want to insert your own HTML code directly into a Logi report. There are many reasons for wanting to do this, including adding <META> information, adding <SCRIPT> content, and embedding external HTML documents into your report page. For example, the topic you're reading right now is actually a separate, external HTML file that has been embedded into the DevNet site page.

## Requirements

HTML that's included into a Logi definition must be have properly-formed tag structures, and include all closing tags. For example,

```
<p><strong>My Title</strong></p>
```

If a complete HTML page is embedded, it may include <HTML> and <BODY> tags, but they aren't required.

# Using the HTML Tag Element

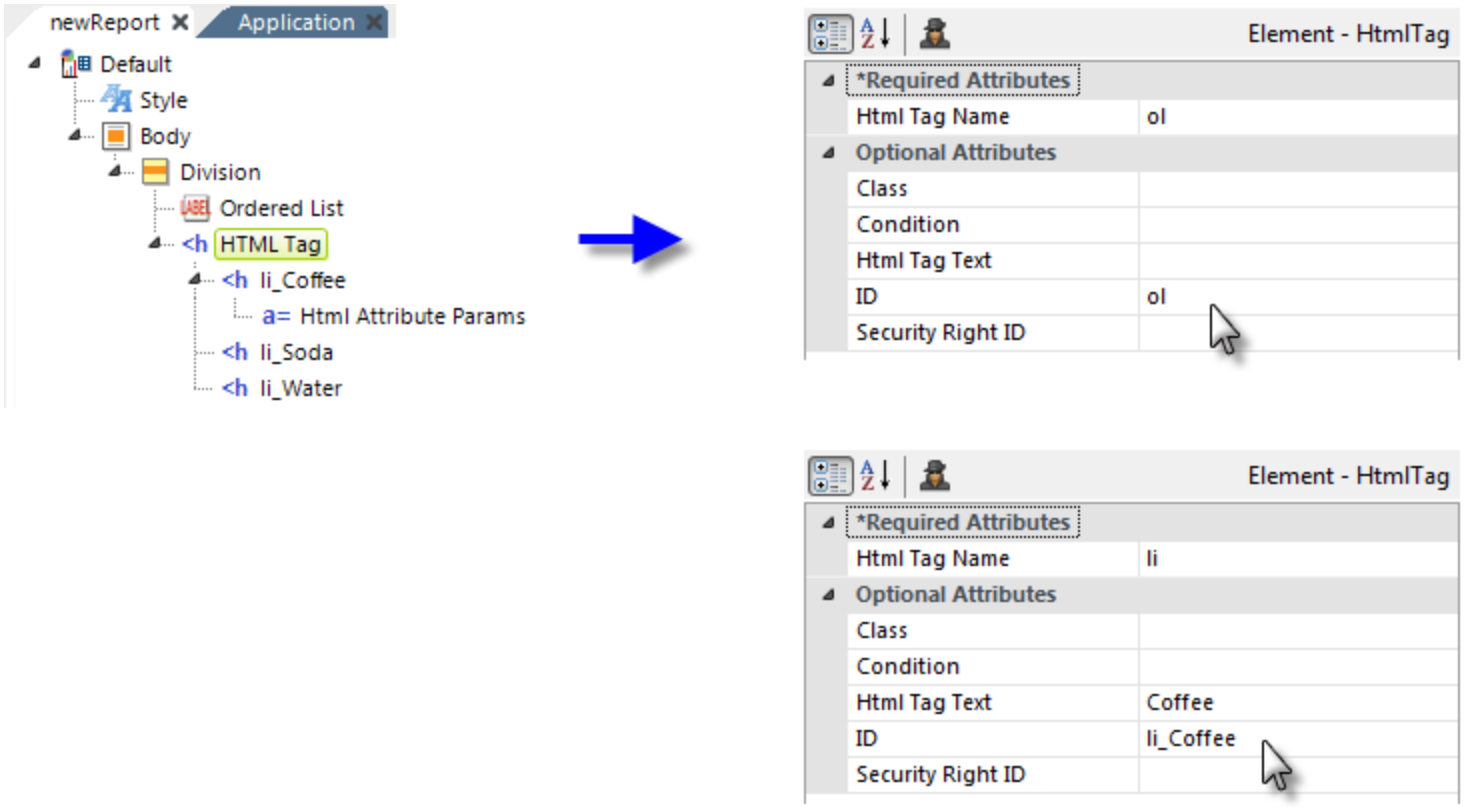
The **HTML Tag** element enables you to easily insert custom HTML into your definition. One popular usage is to add <UL> and <LI> tags, which are commonly used by libraries like JQuery to build a unique UI.

## Attributes

The **HTML Tag** element has the following attributes. Tokens are supported in all attribute values.

| Attribute | Description |
|---|---|
| Html Tag Name | (Required) Specfies the HTML tag to be inserted, such as *OL* or *LI* or *SPAN*. Opening and closing tags will be inserted. |
| Class | Specifies the CSS class to be used by the element. |
| Condition | Specifies an expression that evaluates to a value of *True* or *False*. If *True*, then the HTML is generated, otherwise it's not. Expressions should be in JavaScript or intrinsic function syntax. |
| Html Tag Text | Specifies the text content that will be included between the tags. For example, if Html Tag Name = *SPAN* and Html Tag Text = *My dog has fleas*, the resulting output is: <SPAN>My dog has fleas</SPAN>. |
| ID | Specifies a unique identifier for this element. |
| Security Right ID | When using Logi Security, specifies one or more Security Right IDs, separated by commas. Users with these Security Right IDs will be able to view the HTML tag, users without them will not. |

Here's a usage example:



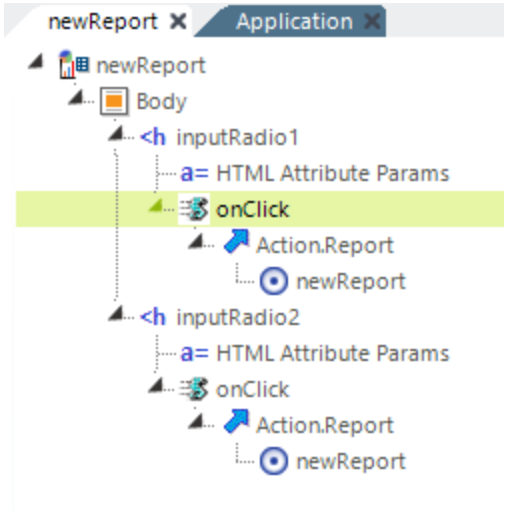

As shown above, you can create hierarchies of nested HTML Tag elements, allowing us to create an un-ordered list (ol) with several list items (li). Notice the **Html Attribute Params** child element being used - it allows you to supply additional attributes (shown highlighted below) for an HTML tag. The HTML generated by the example looks like this:

```
<ol>
 <li title="my title" id="li_Coffee">Coffee</li>
 <li id="li_Soda">Soda</li>
 <li id="li_Water">Water</li>
</ol>
```

Also notice that the HTML is *not* wrapped in <SPAN> tags.

The **Conditional Class** element is available as a child of HTML Tag, allowing conditional setting of style classes.

The **Event Handler** element is available as a child of HTML Tag:



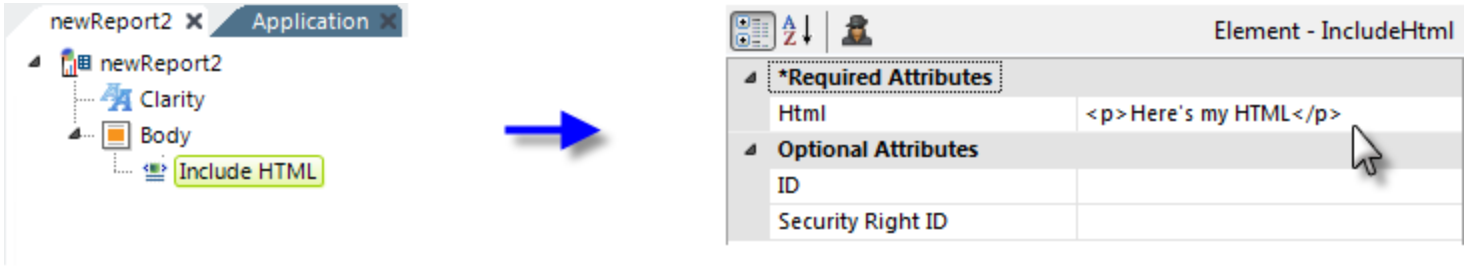
You can use the Event Handler as a child of HTML Tag to trigger actions related to HTML objects. In the example above, the HTML Tag element has been used to create two radio buttons that, unlike the Input Radio Buttons element, don't require a datalayer. When the buttons are clicked, the report is redisplayed. As in the case of a typical User Input element, the selected value is passed along as a Request variable that uses the HTM Attribute Params "name" and "value".

# Using the Include HTML Element

The **Include HTML** element allows you to insert HTML at selected points within your report definition.

This element can be placed in your definition as the child of a number of elements, such as Report Header, Body, Division, Data Table Column, Report Footer, and others. See the Element Reference entry for this element for a complete list.

In the past, developers have used a **Label** element, set to **HTML** format, for this purpose. However, feedback from developers indicated that this was a somewhat obscure usage, so the Include HTML element is now recommended for this purpose instead.




A simple example is shown above and the resulting HTML output looks like this:

```
<!DOCTYPE HTML>
<HTML xmlns:msxsl="urn:schemas-microsoft-com:xslt" xmlns:rdXslExtension="urn:rdXslExtension">
<HEAD>
(... various standard tags not shown for clarity )
</HEAD>
<BODY onload="rdBodyLoad()">
(... various standard tags not shown for clarity )
<div id="rdMainBody">
```

```
<SPAN><p>Here's my HTML</p></SPAN>
</div>
(... various standard tags not shown for clarity )
</BODY>
</HTML>
```

As you can, the HTML from the Include HTML element has been inserted between two <SPAN> tags in the code.

Scripts can be entered here in the same way, using <SCRIPT> tags. If you double-click the **HTML** attribute name and open the Attribute Zoom window, it's easy to enter multi-line scripts, like the following example:

```
<script type="text/javascript">
NagMsgTimer = function() {
setTimeout('ShowNagMsg()', 2*60*1000); /* delay 2 mins, in milliseconds, */
 }
function ShowNagMsg() {
alert('Register your product now!');
setTimeout('ShowNagMsg()', 2*60*1000); /* delay 2 mins, in milliseconds, */
 }
/* cause function NagMsgTimer to run when page is loaded */
 if (window.attachEvent) { /* IE */
window.attachEvent('onload', NagMsgTimer); /* event name is case sensitive */
 }
 else { /* Moz, Netscape, Firefox */
window.addEventListener('load', NagMsgTimer, false);
```

```
}
</script>
```

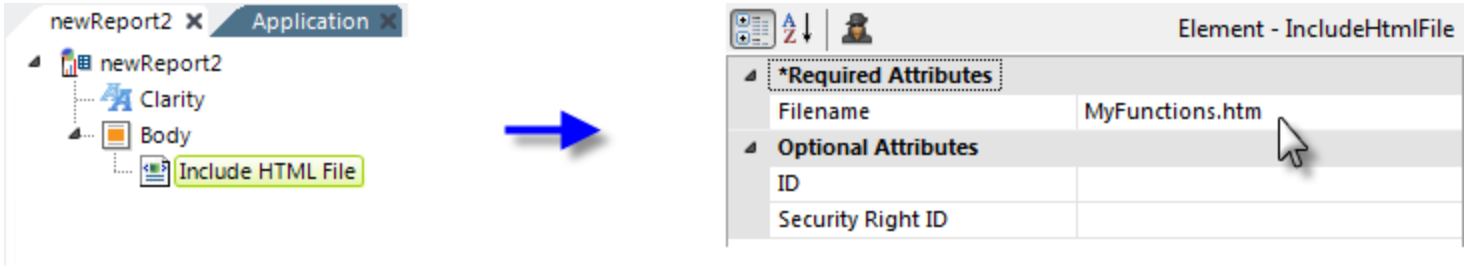
This script will be included in the HTML and, because it attaches itself to the page load event, will begin to execute automatically when the page is loaded. To get the script inserted in the <HEAD> section, see the next part of this topic and use the **Include HTML File** element instead.

# Using the Include HTML File Element

The **Include HTML File** element allows an HTML document to be included within your Logi report definition. HTML files stored in your application's _SupportFiles folder will be available for selection in the element's attributes from a drop-down list.

The Include HTML File element can be placed in your definition beneath a number of elements, such as **Report Header**, **Body**, **Division**, **Data Table Column**, **Report Footer**, and others. When it's placed beneath any of these lower level elements, the HTML it references will be inserted between the <BODY> tags of the generated report page.

It can also be a child of the **Report** (root) element in your definition, which places the HTML between the report page's <HEAD> tags. This can be very useful for including scripts, instructing the browser where to find external style sheets, providing meta information, and more. See the Element Reference entry for this element for a complete list of parent elements.




If your HTML file has been added to the application in the _SupportFiles folder, then, in the Include HTML File element's **Filename** attribute, you need only specify the actual file name (or select it from the drop-down list), as shown above. Otherwise, you'll need to specify a fully-qualified file path (on the web server), starting with a drive letter and the @Function.AppPhysicalPath~ can be used here for that purpose.

You can place script functions in an HTML file (with the proper <SCRIPT> tags) and use Include HTML File to include them in definitions; this can be useful if you want to include the same script functions in a number of Logi definitions but maintain them all in a single file. Or, you may want to use scripting in this manner to include an external JavaScript library.

```
<script type="text/javascript" src="_Scripts/tiny_mce/tiny_mce.js"></script>
```

For example, the **TinyMCE editor** used in DevNet's forums is added into DevNet using an Include HTML File element and an HTML file containing the script shown above.

Here are some key points to remember when working with HTML files with *visible content* and the Include HTML File element:

## Caching Confusion

Files embedded with the Include HTML File element are *cached* at the web server. If you forget about this, it can cause great frustration when you edit the HTML content in your file and then it doesn't appear to change in the report page! To abandon the cached content, you will need to delete the value in the Filename attribute, save and browse the page again, then re-enter the filename and continue testing. Or, if it doesn't affect anyone else, you can reset the web server.

## Width Coordination

When embedding your HTML file in a report definition that you must pay close attention to *widths*. Possible parent elements, such as Column and Data Table Column, need to be wide enough to accommodate your HTML file; conversely, your HTML file may need to constrain its own width to fit inside the Logi report page. Coordination of widths is essential for a good result.

## No Scrolling

A file displayed using the Include HTML File element will be shown in its *entirety*; its full length will be displayed, "pushing" anything below it down the page. There is no option to show a portion of it within a scrolling window. (To create a scrolling area for HTML content within a report, you need to use the **Sub-Report** element).

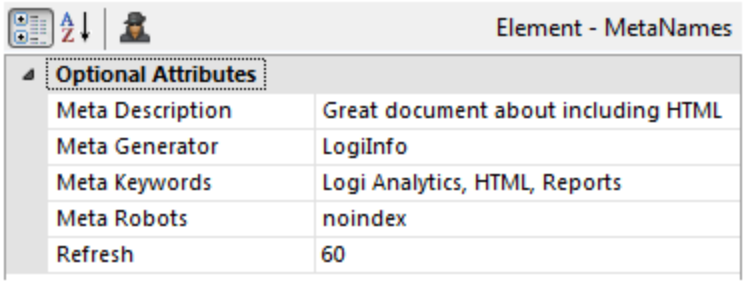
## Style sheet Interaction

The effects of style classes referenced in your embedded HTML document will be *combined* with those of the Logi report page containing the document. They will interact and this can have unexpected consequences as different classes vie for precedence. There is no hard-and-fast rule to follow concerning the use of style in this situation, but be prepared for the possibility that a little experimentation may be necessary to get things looking right.

# Using the Meta Names Element

The HTML <META> tag provides metadata about a Logi report. Metadata is not displayed on the page, but is "machine parsable" and is typically used to specify page description, keywords, modification dates, and other details. The <META> tag always goes between the <HEAD> tagsand can be used by browsers (to tell them how to display content or to reload page), search engines (keywords), and other web services.

Logi products provide the **Meta Names** element, which allows developers to set some of the more common <META> tag variations directly, without the use of an external HTML file and the Include HTML File element. The Meta Names element is child of the report's root element.




The example above shows a Meta Names element with every one of its attributes configured (they're all optional),

```
<META name="Description" content="Great document about including HTML" />
 <META name="Generator" content="LogiInfo" />
 <META name="Keywords" content="Logi Analytics, HTML, Reports" />
 <META name="Robots" content="noindex" />
 <META http-equiv="refresh" content="60" />
```

and the resulting <META> tag that each attribute generates in the report page are shown above. The element's attributes are described below:

| Attribute | Description |
|---|---|
| Meta Description | Specifies a description of the web page. Some search engines use this to provide the user with a document summary in the result of a search. |
| Meta Generator | Specifies the name of the application used to create the document. The default value is *LGXReporting*. |
| Meta Keywords | Specifies a comma-separated list of words that describe the document. Some search engines use this for keyword searches. |
| Meta Robots | Specifies whether the containing document should be indexed by search engines |
| Refresh | Specifies an interval, in seconds, for periodically refreshing the report. |

# Inserting <Meta> and Other Tags Using Javascript

The Meta Names element discussed in "Using the Meta Names Element" on page 439 provides one method of inserting common <META> tags into your report, but if you want to insert other tags, you can do so using JavaScript. Suppose, for example, that you want to insert a tag that specifies a character set, like this one:

```
<META charset="UTF-8">
```

To do this, you can use either an **Include Script** element, beneath the Report Header element, or an **Include Script File** element, beneath the definition's Report root element, to include the following JavaScript:

```
var headTag = document.getElementsByTagName('HEAD')[0];
 var charsetTag = document.createElement('META');
 charsetTag.setAttribute("charset", "utf-8");
 headTag.appendChild(charsetTag);
```

This will add the desired <META> tag to the code between the <HEAD> tags in the generated HTML page. Here's another example: suppose that you want to insert a tag that controls Internet Explorer compatibility settings, like this one:

```
<META http-equiv="X-UA-Compatible" content="IE=EmulateIE7">
```

But, to be effective, this needs to be the *first* tag in the <HEAD> section. The following code allows you to create this arrangement:

```
var headTag = document.getElementsByTagName('HEAD')[0];
 var emulationTag = document.createElement('META');
 emulationTag.setAttribute("http-equiv", "X-UA-Compatible");
```

```
 emulationTag.setAttribute("content", "IE=EmulateIE7");
 // next line ensures insert is first element under <HEAD> tag
 headTag.insertBefore(emulationTag, headTag.firstChild);
```

And, for completeness sake, here's code that demonstrates use of a custom "insertAfter" function, should you need to do that. This code inserts the <META> tag from the previous example into the <HEAD> section, *after* the <TITLE> tag:

```
// create a custom function, which takes two parameters:
 function insertAfter(newElement,targetElement) {
// target is what you want it to go after, look for this element's parent
var parent = targetElement.parentNode;
// if the parent's lastchild is the targetElement...
 if(parent.lastchild == targetElement) {
 // add the newElement after the target element
parent.appendChild(newElement);
 } else {
 // target has siblings, insert new element between the target and it's next sibling
parent.insertBefore(newElement, targetElement.nextSibling);
 }
 }

var titleTag = document.getElementsByTagName('TITLE')[0];
 var emulationTag = document.createElement('META');
 emulationTag.setAttribute("http-equiv", "X-UA-Compatible");
 emulationTag.setAttribute("content", "IE=EmulateIE7");
```

```
// call the custom function to do the insert
insertAfter(emulationTag, titleTag);
```

JavaScript can be used, with the Document Object Model, to manipulate the final HTML page in many ways that are otherwise unavailable through elements.

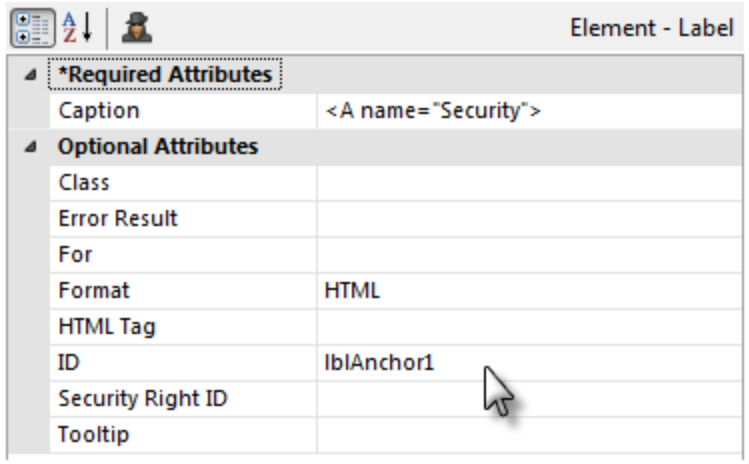
# Using the Label Element with HTML Format

The **Label** element can be used to insert HTML into your report. This is done by setting its **Format** attribute to *HTML* and entering the desired HTML code in its Caption attribute.

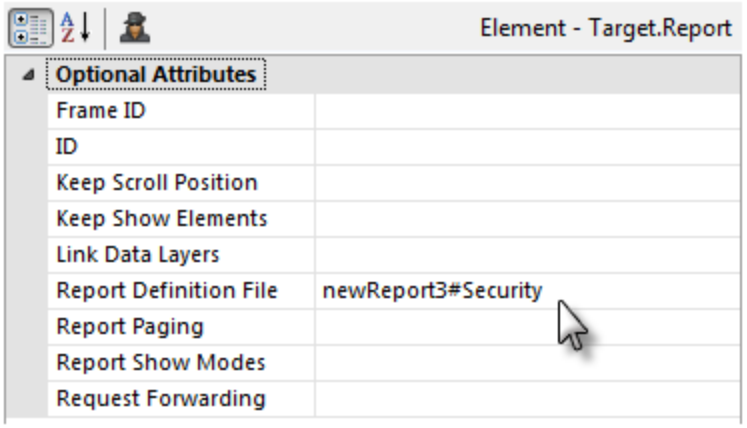
Because the code is embedded within an existing HTML page when the report is generated, you *do not* need to include <HTML> or <BODY> tags in the Caption.

Any valid HTML can be entered in the Caption and this is often a simple way to add **Bold** or *Italic* effects to text without the need for a style class. The HTML must be "well-formed"; that is, you need to ensure that you have entered all of the matching opening and closing HTML tags required.

For example, have you ever needed to call a Logi report and then automatically scroll to some specific location within it? In HTML, this is done using an **Anchor** tag (<A>) with a named bookmark in the destination page. Here's how to do it using a Label element:



1. In the *destination* report, use a **Label** element to create an HTML bookmark, as shown above. The Label element won't be visible, so include it in the appropriate location in your report (the location that the browser will scroll to when the report is displayed). Set its **Format** attribute to *HTML* and, in its **Caption** attribute, enter the HTML code for an anchor, with the anchor name ("Security" in the example).



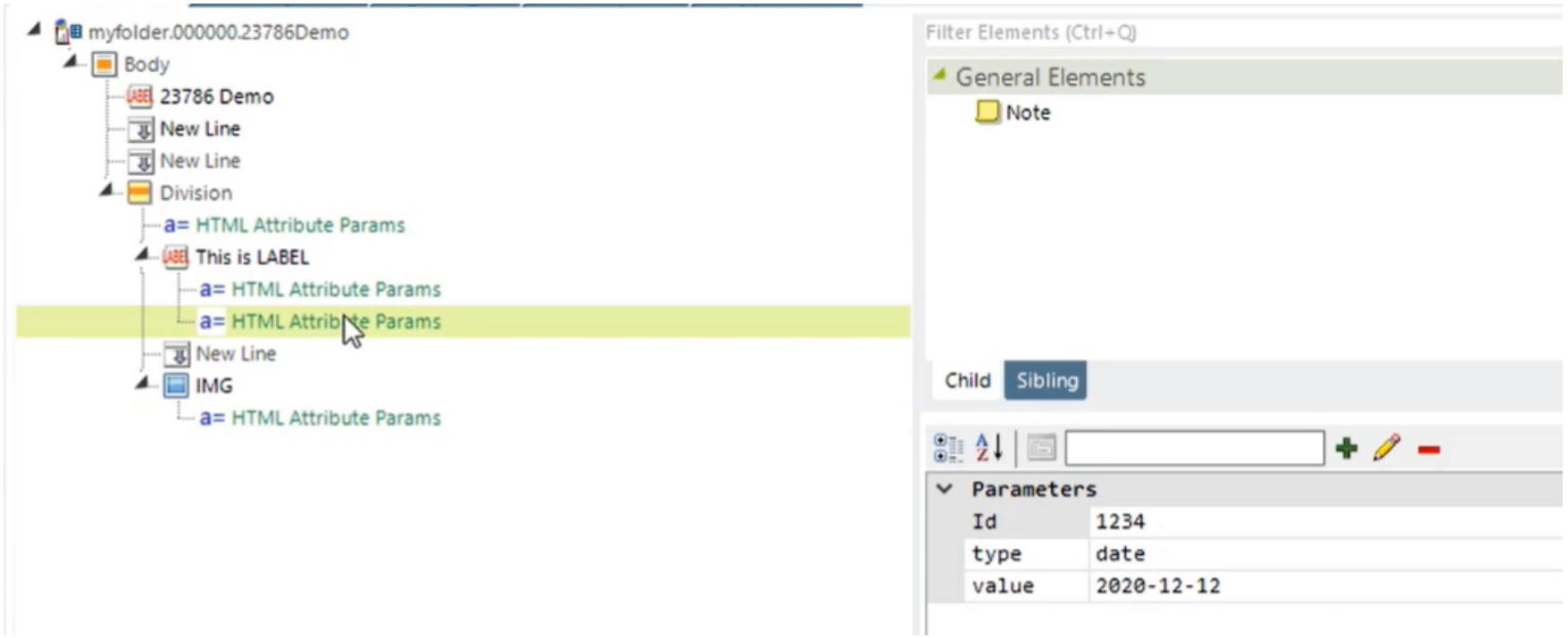

2. In the *calling* report, in the **Target.Report** element used to identify the destination Logi report, add a hash mark (#) and the anchor name after the report definition name, as shown above.

Now, when the link, image, etc. that calls the report definition is clicked, the browser will scroll to bring the Label element from Step 1 into view. The URL generated looks like this:

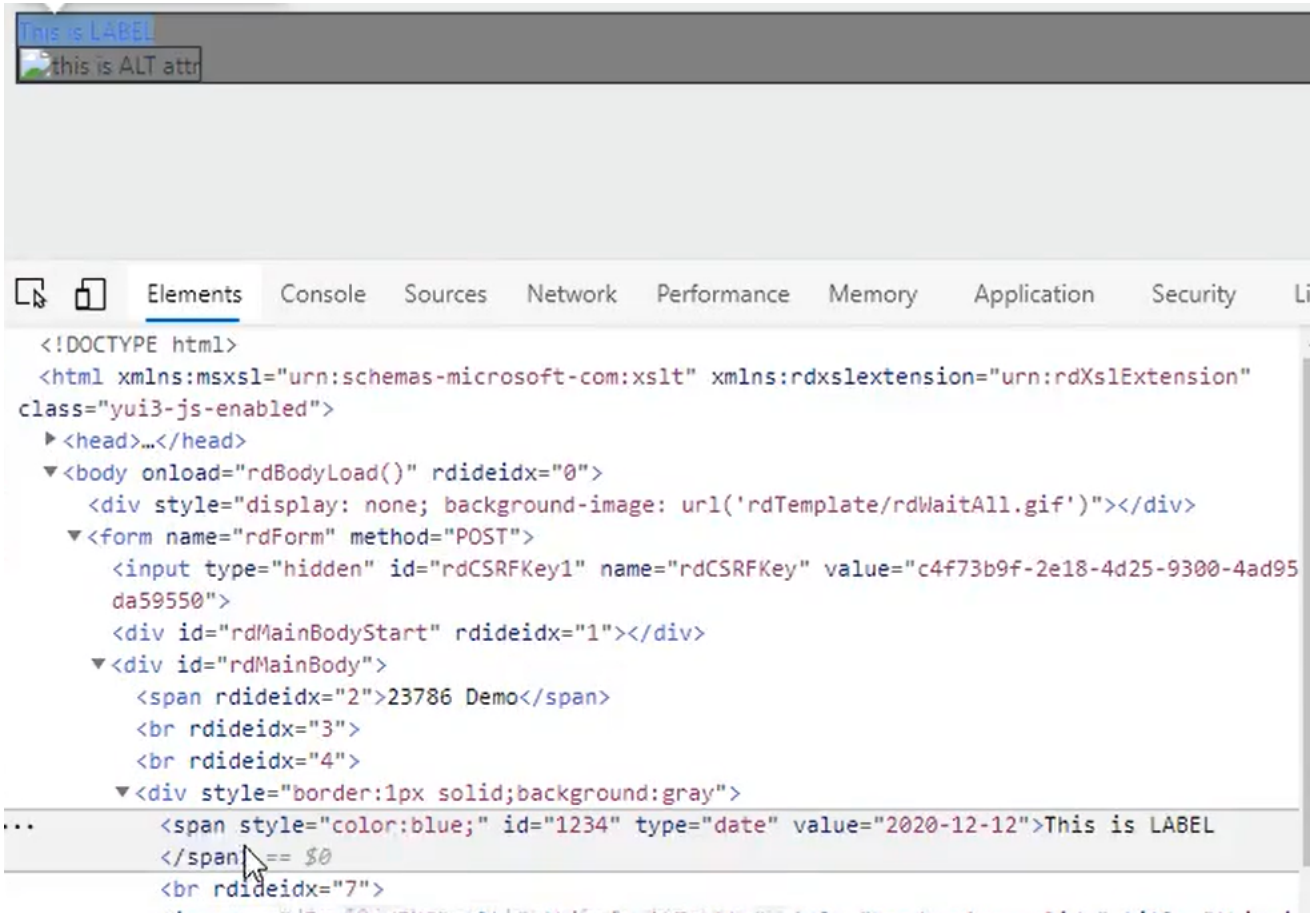
```
http://myServer/myLogiApp/rdPage.aspx?rdReport=newReport3#Security
```

For the anchor to work correctly, it has to be the very *last* thing in the query string. This means, for example, that you cannot use this technique with **Link Parameters** or the **Wait Page** element, as they will add items to the query string *after* the anchor name.

Label 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 "Id", "type", and "value" parameters:



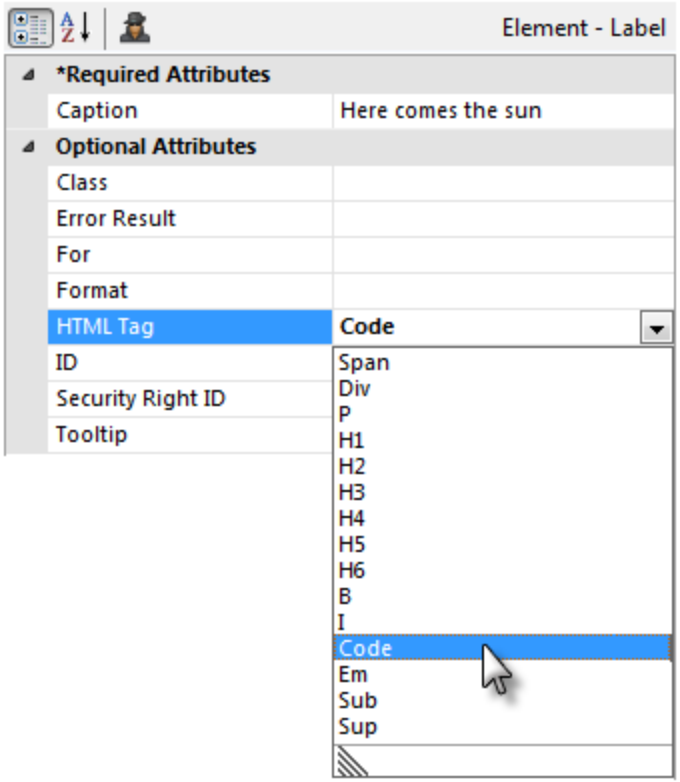

After the HTML Attribute Params are applied:




If an attribute was set in both Element and HTML attribute params, the one set in the HTML attribute params will be ignored.

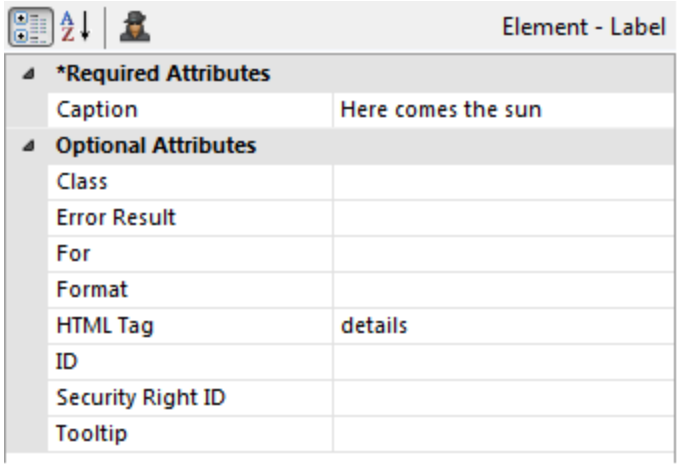
# Using the Label Element's HTML Tag Attribute

The **Label** element's **HTML Tag** attribute allows you to control which HTML tag set will surround the label text. The default is a `<SPAN>` tag set and the resulting HTML output looks like this: `<SPAN>Here comes the sun</SPAN>` However, the new attribute allows you to specify a range of other options:




Produces this output:
<code>Here comes the sun</code>

A selection of commonly-used tags is available from a pull-down list, as shown in the example above.




Produces this output:
<details>Here comes the sun</details>

You can also *type in* other values. HTML5, for example, supports a number of semantic tags including `<article>`, `<figure>`, and `<details>`, and you can type any of these directly into the attribute value (without brackets) to use a tag set not found in the pull-down list.

# Invalid HTML Characters

The Logi Server Engine may generate errors if it needs to read a string of characters as XML and that XML includes un-escaped, invalid XHTML characters.

A primary examples of this is when AJAX-style requests are used, such as Data Table sorting or paging (when **Ajax Paging** = *True* has been set), or use of an **Action.Refresh Element** that includes a Data Table or other data object. In these cases, the XML data may be indistinguishable from XML tags and reserved characters.

This can also happen when HTML is included in a report definition using the techniques discussed earlier.

The most common culprit in this scenario is the inclusion of unencoded "&" characters. In situations like those described above, the application will fail and produce an error message that looks like a parsing error.

Replacing the "&" character with "&"is often an easy solution.

# 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.