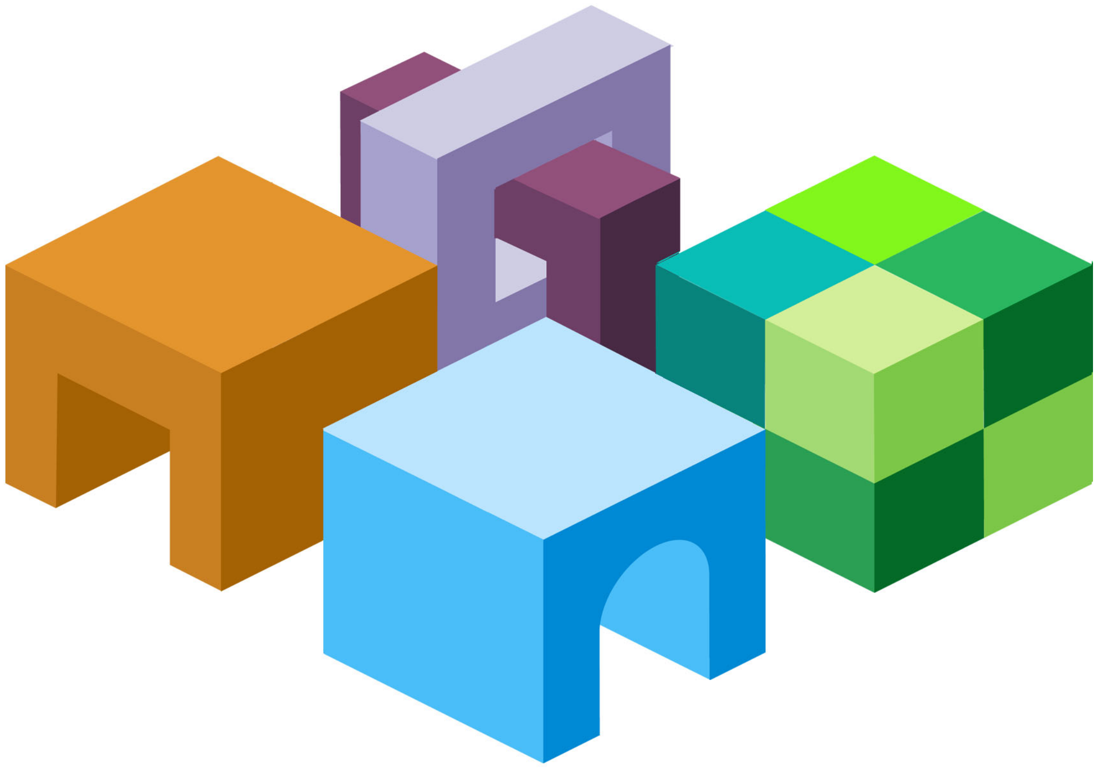


HYPERION® INTERACTIVE REPORTING

RELEASE 11.1.1

OBJECT MODEL AND DASHBOARD DEVELOPMENT SERVICES DEVELOPER'S GUIDE

VOLUME VI: DASHBOARD ARCHITECT

ORACLE®

ENTERPRISE PERFORMANCE MANAGEMENT SYSTEM

Contents

1 Introduction and Concepts

In This Chapter

JavaScript Concepts

This topic discusses the JavaScript and the Oracle's Hyperion® Interactive Reporting Studio relationship. It examines the differences between document-level and dashboard section-level customization.

Definition of JavaScript

JavaScript is an interpreted programming language, that enables executable content to be included in Oracle's Hyperion® Interactive Reporting documents. It is a scripting language that contains a small vocabulary and a simple programming model, enabling developers to create interactive and dynamic content with syntax loops, conditions, and operations. While JavaScript is based on the syntax for Java, it is not Java.

How Interactive Reporting Studio Supports JavaScript

Interactive Reporting Studio supports the Netscape JavaScript Interpreter 1.4. Netscape JavaScript is a superset of the ECMA–262/ISO–16262 standard scripting language, with minor differences from the published standard. The code in the tree supports JavaScript 1.1, 1.2, 1.3, and 1.4. JavaScript 1.4 includes support for some ECMAv2 features (for example, exception handling and new switch behavior).

JavaScript enables developers to customize Interactive Reporting documents at the document, dashboard section (including controls), computed items, and menu levels.

Document-Level Customization

Documents can include queries, tables, pivots, charts, reports, or dashboard sections. Dashboard sections can be inserted into documents or can be standalone. Scripts can be included within Interactive Reporting documents to initiate application functionality, to process queries, activate sessions, and create filters. A document-level action is initiated when the `OnStartUp` or `OnShutDown` events are fired in a document.

JavaScript enables the developer to define event handlers (code that is executed when an event occurs). Events are usually initiated by the user; for example, clicking a button, or events can be set to commence when a document is opened, or a section is activated.

Dashboard Section-Level Customization

Dashboards are interfaces for users to view data, create ad hoc queries, and print reports. Developers use templates, charts, pivots, and so on to create user-friendly sections. These sections are the hypertext, push-button interface to the entire query and reporting process. A dashboard section-level action is initiated when the `OnActivate` or `OnDeactivate` events are fired in a section.

In sections, JavaScript controls the action and behavior of objects such as command buttons, lists, and graphic controls. Typically, objects are scripted to perform actions based on events, such as `OnClick`, `OnSelection`, or `OnDoubleClick`.

Table 1 Dashboard Controls

Dashboard Controls	Event Supported
Command Buttons	`OnClick`
Option Buttons	`OnClick`
Check Boxes	`OnClick`
Lists	`OnClick, OnDoubleClick`
Drop-down Lists	`OnSelection`
Text Boxes	`OnChange, OnEnter, OnExit`

Table 2 Dashboard Graphics

Dashboard Graphics	Event Supported
Lines	`OnClick`
Rectangles	`OnClick`
Round Rectangles	`OnClick`
Ovals	`OnClick`
Text Labels	`OnClick`

Dashboard Graphics	Event Supported
Pictures	`OnClick`
Results	`OnClick, OnRowDoubleClick`
Tables	`OnClick, OnRowDoubleClick`
Pivots and Charts	`OnClick`

Object Oriented Concepts

This topic explains key Object Oriented (OO) concepts, provides examples, and discusses how to address objects, methods, and properties.

Objects

In Object Oriented Programming (OOP) objects represent the real world. An object is a software package that contains a collection of related characteristics and behaviors, or properties and methods.

JavaScript objects can package multiple properties and methods together, or provide built-in, dynamic objects, properties, and methods.

Properties are characteristics that describe data or other objects. For example, a door is a real world object and can be modeled in a software application. An object property example is a door in one of two states—open or closed. A door may have a latch. A latch is a property of the door but because a latch can be locked or unlocked, it is also an object in its own right.

Methods or event handlers are functions associated with objects that can alter the state of the object. For example, a door can be opened or closed. Open and close are examples of methods, a sequence of events must occur to move the door from one state to another. For example, pressure is applied to the door in a direction to close it. In Dashboard Architect, a routine animates the closing of the modeled door.

Object methods and properties define the characteristics and behavior of an object.

Examples of Interactive Reporting Studio Objects

In Interactive Reporting Studio, documents are accessed through the scripting environment as a hierarchy of objects. For example, the Interactive Reporting Studio button object contains these properties and methods.

Table 3 Button Object Properties and Methods

Characteristic	Type	Description
Alignment	Property	Align text on button face—left, right, center

Characteristic	Type	Description
Enabled	Property	Availability of objects to be clicked
Font	Property	Style of text displayed on a button face (an object in its own right that contains its own characteristics)
Name	Property	Programmatically addressed object name
OnClick	Method	Event handler called when button is clicked
Text	Property	Text displayed on button face
Type	Property	Type of object (for example, bqButton)
Vertical Alignment	Property	Align text on button face—top, bottom, middle
Visible	Property	Visibility of a button

The Font object described in the table also contains these properties.

Table 4 Font Properties

Property	Example
Color	Red, blue, black
Effect	Superscript, subscript
Name	Arial, Times New Roman
Size	10pt, 12pt
Style	Bold, italic

A collection is another type of object supported by Interactive Reporting Studio and JavaScript. Collections usually contain a *count* property to calculate how many elements are included as parts. A collection contains one or more methods, and provides a way of enabling access to the elements, to add elements, and other generic facilities to sanction change to the state of the collection. Different types of collections may behave differently.

The Interactive Reporting Studio application object is a collection that contains one or more documents including one or more sections. Each section contains other objects specific to it. For example, queries include a data model, a Request line, and a Filter line. Filters include show values and custom values lists, and custom SQL.

Each object contains unique methods and properties that enable you to leverage the object functionality.

Methods

Methods (functions and event handlers) are actions that change the state of the object. Programmers write JavaScript to determine the actions that are executed when an event is fired. Dashboard Architect is concerned with scripting these actions, which involves the execution of one or more methods. For example, when a button is pressed, the Save method in the current document is invoked.

Examples of Interactive Reporting Studio Methods

Examples of methods include Save in a document, Process in a query, and Recalculate in a results set.

Properties

Properties are characteristics that describe objects.

Examples of Interactive Reporting Studio Properties

Examples of properties include the visible state of a button or section, the connected state of a connection, or the size of a font.

Addressing Objects, Properties, and Methods

Properties and methods belong to objects, and as properties can themselves be objects, a recursive-nested structure evolves. Objects are referenced by including all names in the chain from the imaginary root object right down to the object, property, or method being addressed.

For example, the user property of a connection object in a data model of the query section of the active document is expressed as syntax:

```
ActiveDocument.Sections["Query"].DataModel.Connection.User
```

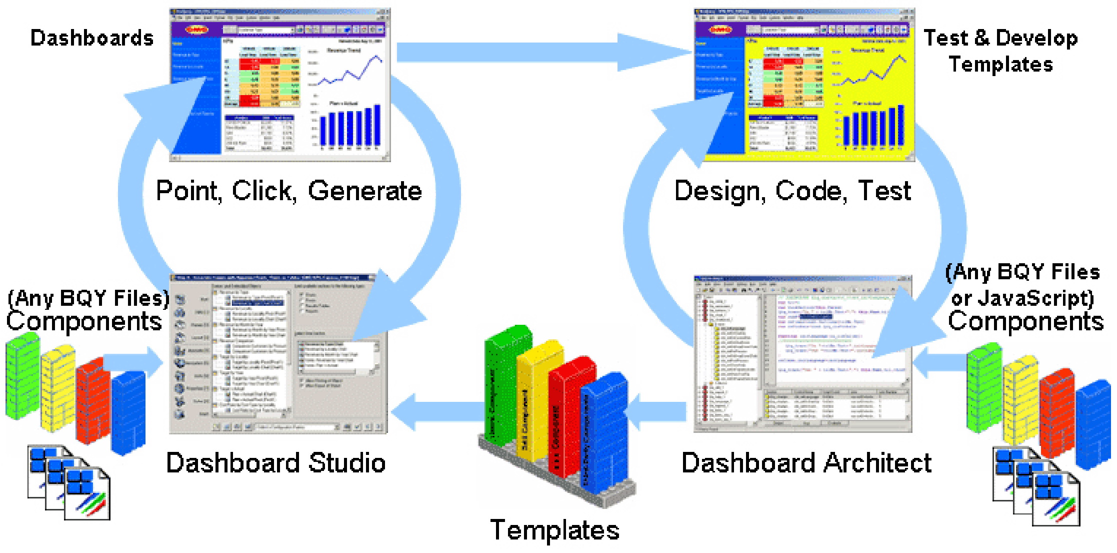
Interactive Reporting Studio and Dashboard Architect provide an object model tree view navigator that can generate the syntax on behalf of the user and ease the task of addressing objects, methods, and properties.

Dashboard Architect Concepts

This topic discusses the main Dashboard Architect concepts, including the operating environment, a description of the software, and the architecture and functions of the program.

The Dashboard Development Environment

The figure illustrates the dashboard development environment and the role of Dashboard Architect within the test and development cycle.



The test and development cycle of Dashboard Studio templates is displayed on the right side of the diagram. Dashboard Architect provides a 4GL-programming environment that uses JavaScript and leverages the Interactive Reporting Studio object model. On the left side of the diagram, the platform provides a 5GL-development environment that requires no programming at all. Application development is through a template-driven, point-and-click wizard whose templates can be extended with plug-and-play components developed by using Dashboard Architect. See the *Hyperion Interactive Reporting – Object Model and Dashboard Development Services Developer's Guide, Volume 7: Component Reference Guide.*

Dashboard Studio templates contain frameworks that provide most basic functions required by analytical applications or dashboards. These functions are implemented as discrete components that are plugged together in a variety of combinations to form templates. Templates are transformed into sophisticated applications with Dashboard Studio, a point-and-click dashboard constructor.

The JavaScript at the core of these templates was developed initially with the Interactive Reporting Studio JavaScript editor. A more powerful development environment was required to cope with the size and complexity of the template, therefore Dashboard Architect was developed.

Dashboard Architect is a JavaScript editor and debugger that is integrated with Interactive Reporting Studio. Dashboard Architect is used to create and maintain Interactive Reporting Studio JavaScript applications, and coupled with Dashboard Studio, it provides a great productivity advantage to users.

The right side of the dashboard development environment diagram is optional, and applies only if Dashboard Architect is used to create templates. The left side of the diagram represents a lifecycle of developing applications directly.

The dashboard development environment is founded on these principles:

- Users capable of surfing the web can use a dashboard

- Users capable of building Interactive Reporting Studio charts or pivots can build sophisticated dashboards with a point-and-click paradigm
- JavaScript programmers can build Interactive Reporting Studio applications and plug-and-play components in a productive manner using these facilities
- Interactive Reporting Studio users can increase quality and productivity by reusing data and code sections

About Dashboard Architect

Dashboard Architect is an integrated development environment (IDE) for Interactive Reporting Studio. It enables you to swiftly test, debug, and build Interactive Reporting Studio applications, freeing up development time and increasing productivity.

Dashboard Architect enables programmers to work on JavaScript alongside Interactive Reporting Studio in a fully interactive integrated manner. Dashboard Architect main functionality includes these concepts:

- Create a project by importing an Interactive Reporting document that contains JavaScript
- Edit the JavaScript in a project with these facilities:
  - Powerful search-and-replace
  - Individual and bulk formatting
  - Brace matching to create complete syntax
  - An object model browser to generate syntax
  - Undo and back-tracking
- Use Dashboard Architect to change an Interactive Reporting document or synchronize the state of the Interactive Reporting document with the state of the project, where Interactive Reporting document changes are made outside Dashboard Architect
- Test and debug code with breakpoints and stack traces to verify correct operations and fix faulty behavior
- Create an Interactive Reporting document from the project and deploy it in production environments

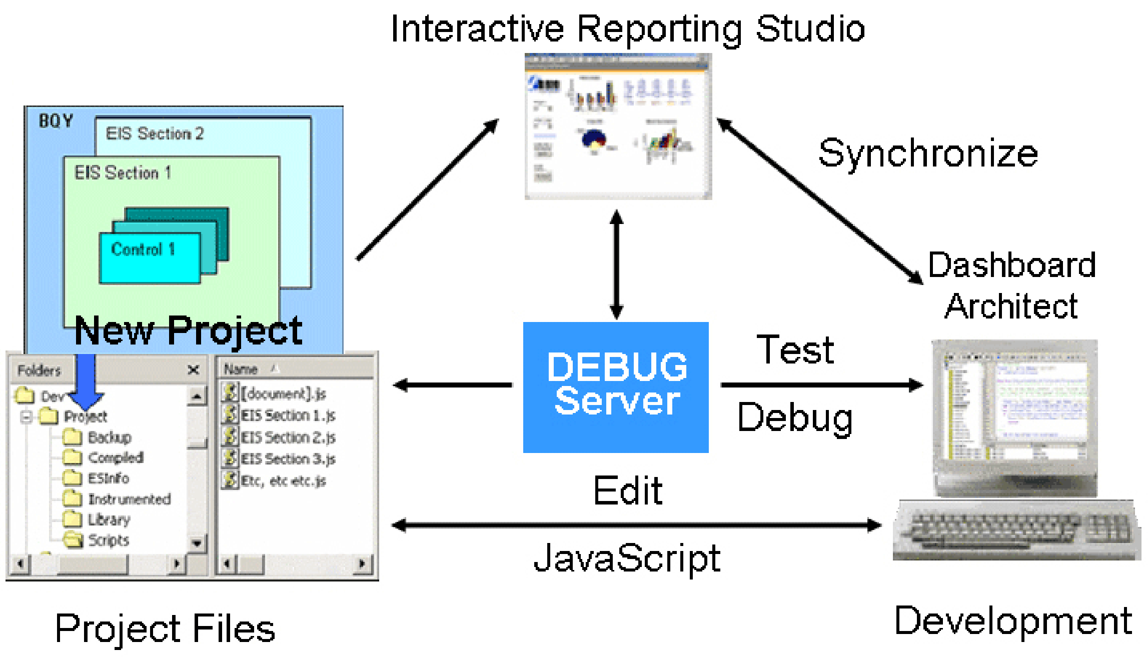
Note:

Source code can be managed and version-controlled like other application sources as it is stored externally in text files.

Architecture

Figure 1 depicts the Dashboard Architect structural design.

Figure 1 Product Architecture



These sub-topics expand the items identified in the architecture diagram.

Creation of Projects from Interactive Reporting Documents

As the starting point of a project, an Interactive Reporting document is composed of many sections, one or more of which may be a dashboard. Each section contains objects that expose scriptable events. Documents may contain JavaScript as part of the document and section events.

The regular Interactive Reporting document is disassembled, and all JavaScript is read from events within the document and stored in a Scripts folder as a text file—one for each section plus one for the scripts. Provision is made for every event in the Interactive Reporting document even if it is empty.

`Instrumented.bqy` is created from the regular Interactive Reporting document. The file contains a full set of unmodified sections including queries, results, tables, charts, pivots, and report sections. It also contains altered or instrumented releases of the original sections. All visual elements remain unchanged but a line of JavaScript is inserted into all object events in place of the original script. The JavaScript calls the Debug Server to request code. In response, the Debug Server sends JavaScript back to Interactive Reporting Studio in time for live execution of the code.

Note:

To successfully duplicate a project, the entire directory tree that contains the project must be copied.

Editing JavaScript

Interact with Dashboard Architect to view and edit the JavaScript that is outside the Interactive Reporting document. A window similar to Windows Explorer provides access to the JavaScript, including a tree view of all sections, objects, and events in each dashboard, as leaves of the tree.

Testing Capability

Code is available to the Interactive Reporting document and Interactive Reporting Studio in real time, even though it is held outside the Interactive Reporting document. The reason this occurs is because every Interactive Reporting document event is a call to the Debug Server requesting its code and passing itself as a parameter. Interactive Reporting Studio is used to execute the code. Buttons and items are clicked, events are fired, and event handlers call the Debug Server. The Debug Server examines the object, fetches the JavaScript, and returns it to the event handler as a string. The event handler completes the operation by passing the returned string through to the JavaScript `eval()` function and so the latest code is parsed, interpreted, and executed, all in real time. The Interactive Reporting Studio environment is thereby able to execute the code as if it were stored internally, ensuring that the document behaves as if it were running as usual, instead of being instrumented.

Debugging Capability

Debugging is closely related to testing. When debugging in Dashboard Architect, one or more breakpoints are typically placed in strategic positions in the code that is being edited. Execution is interrupted in Interactive Reporting Studio and the current state of the application is examined at a point within the application logic.

Adding a breakpoint modifies the way the regular JavaScript behaves. Instead of simply executing, code is inserted that makes calls back to the debugger to highlight the line that is about to be executed. When such a line is encountered, Dashboard Architect hands control back to you so the state of the properties and objects can be examined or modified, and the behavior of the code is understood.

Synchronization with Interactive Reporting Studio

Interactive Reporting Studio enables many powerful actions to be performed through its programmatic interface. However, it does not sanction the creation or deletion of objects on a dashboard, nor the dynamic addition of JavaScript into the event handlers of those objects.

To solve these problems, Dashboard Architect provides a synchronization mechanism. Actions that are performed through the Interactive Reporting Studio user interface, are detected and the Dashboard Architect structures are adjusted to reflect the current state of `instrumented.bqy`. While this is not a seamless way to create and remove objects, it is an easy-to-use mechanism.

Re-creation of an Interactive Reporting Document

The final step in the development sequence is the re-creation of a regular Interactive Reporting document. Merging the code that was edited inside Dashboard Architect with sections and controls within `instrumented.bqy`, creates a document that is no longer connected to Dashboard Architect and that can be deployed in any production environment. No trace of Dashboard Architect can be found in the compiled Interactive Reporting document. See "Making an Interactive Reporting Document" on page 33.

2 Dashboard Architect Features

In This Chapter

Opening Dashboard Architect

Upon opening Dashboard Architect, a window is presented that enables you to create a new project or select a project.

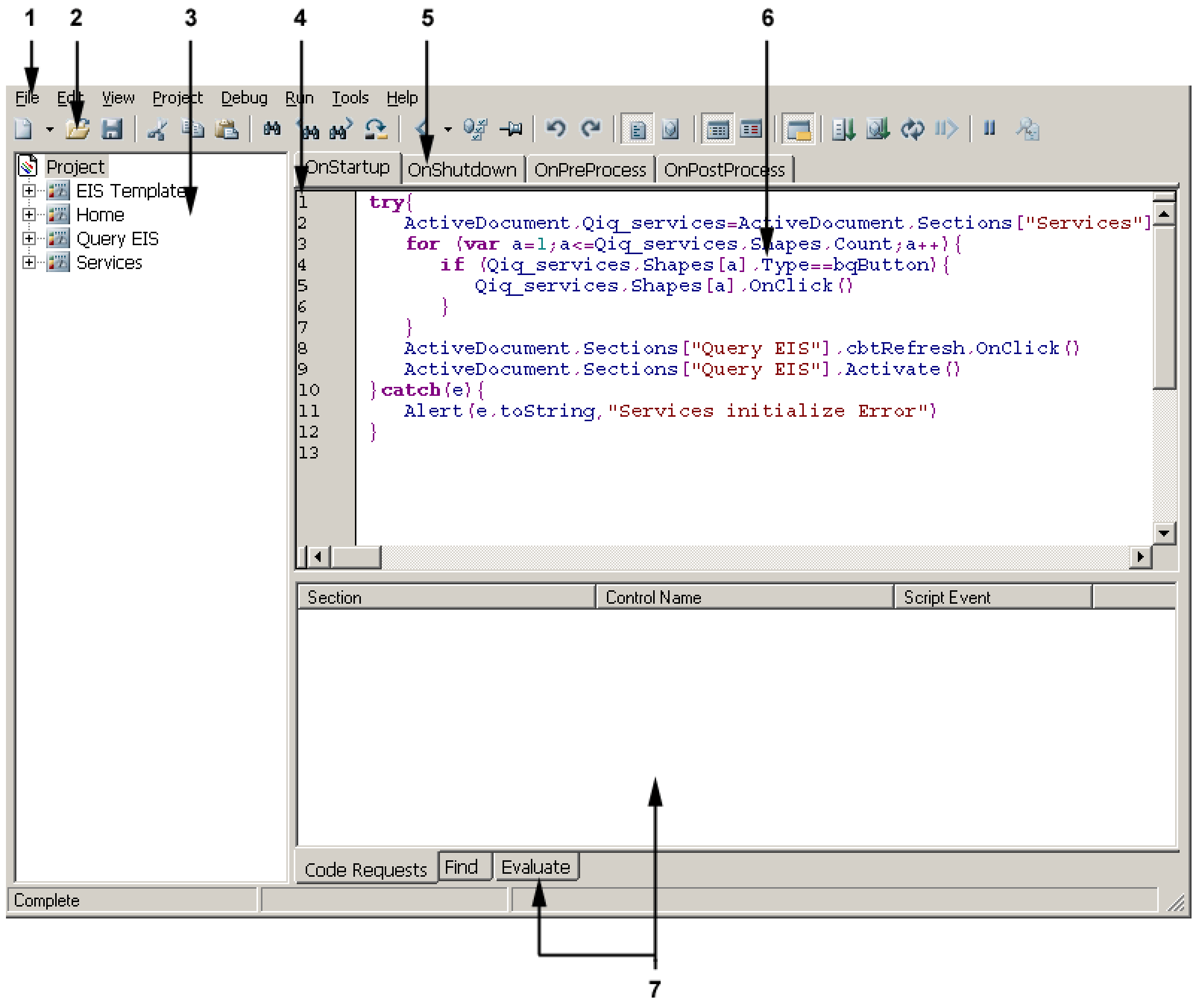
To create a new project, click New Project. See "Creating Projects" on page 29.

To select a project, perform an action:

- Click Browse, and locate the Dashboard Architect Project (*.qdp) file
- Select a project from Open Project, and click Open

The Dashboard Architect User Interface

The main Dashboard Architect screen is divided into numbered segments, as illustrated in the figure. Each segment is described in Table 5.



```
try{
   ActiveDocument.Qiq_services=ActiveDocument.Sections["Services"]
   for (var a=1;a<=Qiq_services.Shapes.Count;a++){
      if (Qiq_services.Shapes[a].Type==bqButton){
         Qiq_services.Shapes[a].OnClick()
      }
   }
   ActiveDocument.Sections["Query EIS"].cbtRefresh.OnClick()
   ActiveDocument.Sections["Query EIS"].Activate()
}catch(e){
   Alert(e.toString,"Services initialize Error")
}
```

Table 5 Interface Descriptions

Item Label	Description
1. Menu bar	Access to Dashboard Architect features (Some menu bar entries provide common keyboard shortcuts and use buttons to activate functionality)
2. Toolbar	One-click access to the most functionality
3. Navigation panel	The panel contains one of these modes: 1. Sections and the programmable elements within them in the form of an expanded or contracted tree view. The editing window (6) displays the JavaScript associated with the selected programmable element. 2. Interactive Reporting Studio objects available in the instrumented Interactive Reporting document in the form of an expandable tree view. Double-click an object, method, or property, and the associated syntax is added at the cursor position in the editing window (6).
4. Line number	Line number indicator
5. Event handlers	Displayed as content tabs for sections or controls at the top of the editing window (6)

Item Label	Description
6. Editing window	Edit JavaScript or examine executed code with breakpoints
7. Output window and content tabs	The output window contains three content tabs. 1. Code Requests—Track calls made by `instrumented.bqy` to the Debug Server for the JavaScript that each object must evaluate. If an error is detected, the last icon in the list is displayed with the red cog over the folder icon ( ). Clicking a line of code in the output window causes the editing window (6) to display the JavaScript for that event handler. 2. Find—Display the lines matching a find specification. Clicking a line in the find list causes the editing window (6) to display the JavaScript for that event handler and the cursor is placed where the match is made on the line. 3. Evaluate—Enable interaction with `instrumented.bqy` while execution is paused on a breakpoint. Two sub-panes are visible, one in which JavaScript expressions can be entered for evaluation and another in which the results of the evaluation are displayed. The window also contains buttons for debugging: (Evaluate) and (Stack Trace). Within the evaluate pane (Cut), (Copy), or (Paste) an expression to be evaluated, or click (Clear) to delete the expression from this window. **Note:** The output window is visible only if is clicked or Crtl+W is pressed.

Menu Commands, Shortcuts, and Buttons

This topic provides a comprehensive list and description of the Dashboard Architect menu commands, buttons, and shortcuts.

The File Menu

The commands available in the File menu are discussed in Table 6.

Table 6 File Menu Descriptions

Command	Button (where applicable)	Shortcut (if applicable)	Description
New Project		Ctrl+N	Create a project file from an Interactive Reporting document (The starting point of a project)
Open Project		Ctrl+O	Open a project to edit and test the code
Close Project			Close an open project

Command	Button (where applicable)	Shortcut (if applicable)	Description
Save Project	💾	Ctrl+S	Save an open project, including the JavaScript and associated `instrumented.bqy`
Print		Ctrl+P	Print a list of the JavaScript for the whole or part of the project
Print Setup			Configure printer settings
Make BQY			Create an Interactive Reporting document from `instrumented.bqy` and the JavaScript in the Dashboard Architect Scripts folder
<numbered items>			Open a recently used project
Exit		Ctrl+Q	Close the project and the application

The Edit Menu

The commands available in the Edit menu are discussed in Table 7.

Table 7 Edit Menu Descriptions

Command	Button (where applicable)	Shortcut (if applicable)	Description
Undo	↶	Ctrl+Z	Revert the document to a previous state (Operates only on actions in the current event. Moving to another event handler and using the Find, Replace, and Save operations causes discontinuity in the undo log)
Redo	↷	Ctrl+Y	Opposite to Undo
Cut	✂	Ctrl+X	Regular Windows behavior with regard to selected text
Copy	⎘	Ctrl+C	Regular Windows behavior with regard to selected text
Paste	📋	Ctrl+V	Regular Windows behavior with regard to selected text
Find	🔍	Ctrl+F	Activate Find and specify search criteria to match a text string or a pattern
Find Next	🔍	F3	Find the next instance of a matched text string or a pattern

Command	Button (where applicable)	Shortcut (if applicable)	Description
Find Previous		Ctrl+F3	Find the prior instance of a matched text string or a pattern
Replace		Ctrl+H	Activate Replace and specify the criteria to find one or more occurrences of a string (Each match is replaced with a different string)
Match Brace		Ctrl+B	Starting at the cursor position, locate the next opening parenthesis ((), brace ({), or bracket ([) and find the closing counterpart (If no opening ((), ({), or ([) exists, the function scans backwards until one is found and moves forward and finds the closing counterpart. If a match is found, the area between the opening and closing items is highlighted)
Go to Row		Ctrl+G	Move the cursor to the nominated row number in an event handler

The View Menu

The commands available in the View menu are discussed in Table 8.

Table 8 View Menu Descriptions

Command	Button (where applicable)	Shortcut (if applicable)	Description
BQY Script Outline			Turn Object Model mode off and display sections and objects in the navigation panel
Object Model			Turn BQY Script Outline mode off and replace the navigation panel with the Object Model browser
All Events			Display all objects with event handlers in the navigation panel (Applies in BQY Script Outline mode)
Scripted Events			Display objects with event handlers that are not empty in the navigation panel (Applies in BQY Script Outline mode)
Output Window		Ctrl+W	Hide or reveal the output window
Back			Return to where the cursor was last clicked within a script (Includes a drop-down arrow which displays a list of recently visited events)

The Project Menu

The commands available in the Project menu are discussed in Table 9.

Table 9 Project Menu Descriptions

Command	**Button (where applicable)**	**Description**
Import		Import one or more Interactive Reporting Studio sections from an external Interactive Reporting document (Sections are instrumented as part of the import)
Add Section		Create a section in `instrumented.bqy` and a matching set of event handlers in Dashboard Architect
Rename Section		Rename an Interactive Reporting Studio section (The corresponding event handlers in Dashboard Architect are renamed)
Delete Section		Delete a section (The corresponding event handlers are deleted in Dashboard Architect)
Add Control		Open a dialog box detailing the process required to create an object in a dashboard (Interactive Reporting Studio does not sanction the creation of this type of object programmatically so the operation involves a two-step process beginning in Dashboard Architect and concluding in Interactive Reporting Studio)
Resynchronize	🔄	Close `instrumented.bqy`, examine it, and instrument new objects (Synchronizes the state of the event handlers within Dashboard Architect with the current state of `instrumented.bqy`)
Include Control Documentation		Indicate whether documentation for variables and functions, not scoped outside a control, are included in the generated documentation
Generate Documentation		Generate the HTML documentation for the current Interactive Reporting document

The Debug Menu

The commands available in the Debug menu are discussed in Table 10.

Table 10 Debug Menu Descriptions

Command	**Button (where applicable)**	**Shortcut (if applicable)**	**Description**
Toggle Breakpoint	⏸	F9	Turn a breakpoint on or off (Select Run, then Fire Current Event before the breakpoint takes effect)

Command	Button (where applicable)	Shortcut (if applicable)	Description
Clear All Breakpoints			Turn off all breakpoints
Evaluate			Evaluate the expression in the left text box of the evaluate pane in the output window (Must be a valid JavaScript expression, otherwise a JavaScript error is generated and execution ceases at the breakpoint. Only available when paused on a breakpoint)
Stack Trace			Display the sequence of function calls that lead to the breakpoint (Only available when paused on a breakpoint)

The Run Menu

The commands available in the Run menu are discussed in Table 11.

Table 11 Run Menu Descriptions

Command	Button (where applicable)	Description
Fire Current Event		Cause the current event handler to be called, triggering the evaluation of JavaScript (Used to check for syntax errors, and to update definitions and breakpoints in functions)
Fire OnStartUp Event		Cause the `OnStartUp()` event of the document to be fired
Fire Document Load		Save, close, and reopen `instrumented.bqy`
Continue		Continue past the current breakpoint

The Tools Menu

The commands available in the Tools menu are discussed in Table 12.

Table 12 Tools Menu Descriptions

Command	Button (where applicable)	Description
Options		Display the Options dialog box (See "The Options Dialog Box" on page 27)
Reload Macro Definitions		Reload the macro definitions from the macros.txt file (Useful when developing, testing, and modifying macros)

Command	Button (where applicable)	Description
Interactive Reporting Studio Type Library Diagnostics		Open the Installation Diagnostics window (As part of the Installation Diagnostics, the Type Library file (TLB) describes the interface offered by a COM server such as Interactive Reporting Studio. Dashboard Architect uses the TLB to communicate with the services that Interactive Reporting Studio offers. If you are using a release earlier than Release 9.3.1 (except Release 9.2, Service Pack 3), and the version of the TLB and the release of Interactive Reporting Studio are not in synch, this can result in unexpected behaviors in Dashboard Architect. In a standard installation, the TLB is located in the Windows system32 folder and is called `brioqry.tlb`. The Interactive Reporting Studio installer keeps the application and the TLB in synch. However, when non-standard mechanisms for upgrading are used problems may arise.)
Line numbers		Show or hide line numbers in the editing window (A discrepancy exists between the line numbers shown in the Console window and the line numbers shown in Dashboard Architect when working with an instrumented Interactive Reporting document. To translate the line number reported in the Console window, subtract two for line numbers if no breakpoints exist in the current event, or the breakpoints come after the line being reported. Subtract a further five for each breakpoint)
Always On Top	📌	Dashboard Architect window is always be on top of other windows (Comments are unavailable when running in this mode)
	📌	Cause the application to behave as a typical window

The Help Menu

The commands available in the Help menu are discussed in Table 13.

Table 13 Help Menu Descriptions

Command	Shortcut (if applicable)	Description
Contents	F1	Display Dashboard Architect help
About Hyperion Dashboard Architect		Display product release information

The Options Dialog Box

Options enables you to change the look and feel of the Dashboard Architect interface. To display the Options dialog box, select Tools, then Options.

These settings can be modified with the Options dialog box:

- **Tab Indent**—Represents the number of spaces to add when the Tab key is pressed (The current line in the editing window is applied to the next line if Tab Indent is not altered)
- **History Entry Total**—Access up to 20 event handlers that were previously visited (Contains a limit of 20 entries)
- **Show Welcome dialog on startup**—Displays the Welcome dialog when Dashboard Architect is launched
- **Strip INCLUDES on Creation**—Removes references to all INCLUDES when a project is created, if selected
- **Enable automatic coding extensions**—Enables the Auto-Code feature, if selected (See "The Auto-Code Feature" on page 41)
- **Show warning for invalid documentation tags**—Enables the displayed warning dialog box when an invalid documentation tag is encountered, if selected (See"Documentation" on page 61)
- **Code Font**—Customizes the code font displayed in Dashboard Architect (Select a font, font style, and size from the Font dialog box, and apply it to the code)
- **User Interface Font**—Customizes the Dashboard Architect user interface font
- **Language**—Enables the user interface language to be set (A necessary option for localized or Unicode releases, as different languages display characters differently)

The remaining commands under Editor Format enable modification of the colors and font type used for syntax highlighting. Select a syntax format from the list, and apply color and font type.

Note:

Color is applied to text only as it is being viewed. It is not stored with the text in the JavaScript files.

3 Creating a Project

In This Chapter

Creating Projects

The process of creating or duplicating a project is the starting point for developing or maintaining code with Dashboard Architect. Dashboard Architect does not however, create Interactive Reporting documents, Interactive Reporting Studio sections, or dashboard objects, because there is no API facility enabling it to do so.

Dashboard Architect complements the strengths of Interactive Reporting Studio and provides value in scripting and debugging event handlers in sections.

➤ To create a project:

1 **Start Interactive Reporting Studio, and, on the Welcome dialog, click Cancel.**

Starting Interactive Reporting Studio before starting Dashboard Architect avoids the screen flicker caused by the intermittent display of menu bars as Dashboard Architect interacts with Interactive Reporting Studio.

2 **Start Dashboard Architect.**

3 **Alternatively, to open Dashboard Architect, start Interactive Reporting Studio, and select Tools, then Launch Dashboard Architect.**

4 **Select File, then New, or click .**

Create New Project is displayed.

5 **Click (next to Original BQY Location), and navigate to the Interactive Reporting document that contains the JavaScript to be to maintained or redeveloped.**

The Interactive Reporting document selected for this example is `Arch_tut.bqy`, located in the Samples directory.

Dashboard Architect Project Folder is filled with a project folder name (that matches the initial Interactive Reporting document), to be created in the folder that contains the original Interactive Reporting document.

Project Title is automatically filled to reflect the original selected Interactive Reporting document. Use the default title or specify another title.

6 **Optional: To override the default Dashboard Architect Project Folder, click 📂 to locate another folder.**

The folder must be empty because files and folders are added to create the directory structure.

7 **Click OK.**

Dashboard Architect examines the selected Interactive Reporting document, disassembles it, and creates a JS file for each section, in which it makes provision for event handlers for all dashboard objects. JavaScript found in the Interactive Reporting document is placed in the event handler locations within the JS files.

Dashboard Architect creates `instrumented.bqy`, which looks identical to the original Interactive Reporting document, except the JavaScript within it is replaced with instrumented debug code. The debug code calls the Debug Server, which returns the JavaScript code from the JS files to the Interactive Reporting document under test, in real time.

`Instrumented.bqy` behaves similarly to the original Interactive Reporting document in the presence of the Debug Server and Dashboard Architect. The original Interactive Reporting document is left unchanged and a copy is placed in the Backup folder. The file name adopts the form of *ORIGINAL-<date and time>*. For example, `ORIGINAL-2004-10-09-18-26-40.bqy`.

At the end of the development cycle, an Interactive Reporting document is re-created from the JavaScript in the JS files and `instrumented.bqy`.

The Project structure contains these folders:

- Backup—The original Interactive Reporting document that was used to create `instrumented.bqy`
- Compiled—The starting Interactive Reporting document (Additional Interactive Reporting document files are added each time the Make BQY command is selected)
- Instrumented—Contains `instrumented.bqy`
- Scripts—One JS script file for each section and one for the document
- Library, ESInfo, and Doc—Reserved for future use

After the instrumentation process is complete, Dashboard Architect displays the JavaScript for the created project. The project is ready for testing and editing.

Duplicating Projects

This procedure follows on from “Creating Projects” on page 29.

To duplicate a project, the directory tree that contains the project must be copied.

➤ To duplicate a project:

1 **Open Windows Explorer.**

2 **Locate the project to be duplicated.**

For example, locate the `Arch_tut` folder.

3. **Select the directory tree, including the project file, the instrumented Interactive Reporting document, and the JavaScript files.**

   In short, select everything that the project requires to work. Individual Dashboard Architect projects are self-contained and can be copied with no side-effects.

4. **Right-click, and select Copy.**

5. **Create a folder to paste the duplicate project directory into.**

   For example, create `Arch_tut_copy`.

6. **Paste the copied directory into the folder.**

   For example, paste the files into `Arch_tut_copy`.

7. **Open the duplicate project file in Dashboard Architect.**

   The duplicated project functions just like the original.

   Optional: An alternative duplication method for developers who use source code control systems, is to use a branch in the source code control system. This option is a more traditional software development technique.

4 Making an Interactive Reporting Document

At the end of the development process, Dashboard Architect re-creates a regular Interactive Reporting document from `instrumented.bqy` and the updated JavaScript from the JS files.

Instrumented debug code is replaced with the JavaScript from the JS file for each event in each dashboard. The regular Interactive Reporting document can now run in environments where Dashboard Architect is not available, as a regular Interactive Reporting document. For example, it can be deployed in these environments:

- Client; for example, Interactive Reporting Studio
- Plug-in; for example, Oracle's Hyperion® Interactive Reporting Web Client

➤ To create an Interactive Reporting document of the project that is open in Dashboard Architect:

1. **Select File, then Make BQY.**

   Save As is displayed. By default the file is saved into the folder which was specified when the project was created. A backup of the file is also created and stored in the Compiled folder of the project.

2. **Click Save.**

   No trace of Dashboard Architect instrumentation remains in the compiled Interactive Reporting document.

5

Editing

In This Chapter

General Notes About Editing

Editing JavaScript in Dashboard Architect is similar to most editing operations in programs such as UltraEdit, VisualBasic, and Notepad. Use the Dashboard Architect editing window to edit code in an Interactive Reporting document.

The changes made in the editing window are saved to the JS files in the Scripts folder when save is next performed, by clicking 💾, or reloading the document. Code changes are kept in memory and are available to the Interactive Reporting document when event handlers request code through the Debug Server.

The keyboard and buttons behave in familiar ways. These are special features to note.

- Syntax is color-coded—Dashboard Architect uses a dictionary of reserved words to control how it applies color to syntax (Color-coding is controlled in Options, select Tools, then Options. Color is applied to text only as it is viewed, and is not stored with text in the JavaScript files. See "The Options Dialog Box" on page 27)
- Indentation—Applied to the current and the next line (The number of spaces to add when Tab is pressed is controlled in Options. See "The Options Dialog Box" on page 27)
- Shift+Enter and Enter are not identical—Enter terminates a line of JavaScript, whereas Shift +Enter does not terminate a JavaScript statement; it continues it on the next line (Do not

use Shift+Enter with breakpoints, because these cannot be set across continuation lines. If breakpoints are added to these lines, JavaScript syntax errors are generated. See "Breakpoints" on page 52)

- Back operations—Return to previously visited event handlers (Up to 20 event handlers are kept in the execution stack for back-tracking purposes)
- Discontinuity in the undo log—Caused by operations such as Save, Find, Replace, Fire Current Event, Fire StartUp, Reload Document, and navigation to another event handler (After these operations are executed, undo cannot go past them. It is possible to Undo, Cut, Delete, and Paste up until, but not past, a point of discontinuity)

Navigation Using the Script Outliner

Use the script outliner view to navigate in Dashboard Architect.

Dashboard Architect uses a mechanism similar to Windows Explorer to assist in navigating through a project.

Click (BQY Script Outline) and select an object in the navigation panel. The JavaScript for the selected object is displayed in the editing window.

The contents of the editing window can be changed by clicking a row in the Code Requests or Find panes of the output window.

Code Generation Using the Object Browser

Click (BQY Object Model) to replace the navigation panel with an object browser. Use the object browser to explore the objects and properties that Interactive Reporting Studio exports through the COM interface. The set of objects and properties is similar to the object browser view available in Interactive Reporting Studio.

Expand and contract the nodes in the tree by clicking [+] or [-]. Or double-click the object or property to generate a reference in the editing window.

The object or property declaration is displayed beneath the object browser in the description information box. Click to access the Interactive Reporting Studio Help. Click (Reset Object Browser) to collapse the object tree and clear the local cache. Obtaining the object information is an expensive operation, so the retrieval is done in real time. Dashboard Architect caches the results. If the Interactive Reporting document changes, the cache may be out-of-date and must be reset. For example, if a filter or computed item is added in the part of the tree that is expanded, refreshing or resetting enables the object to be viewed by starting from the top.

The Find Dialog Box

Access Find by selecting Edit, then Find or clicking .

The Find dialog box is divided into three parts:

- Search conditions (The Search Feature)
- Extent of the Find operation (The Options Feature)
- Search preferences; that is, find the next instance (The Find Next Option) or find every instance (The Find All Option)

To use Find, in the Find What, enter the string or pattern to search, and click Find Next or Find All. Or select a previously entered Find criteria from the drop-down list.

The Search Feature

The scope of Find can be limited to these areas:

- Current Event—The event handler currently displayed
- Current Section—The section of which the event handler is a part
- Current Project—The entire project

The Options Feature

Several options are available to focus a Find process.

Find Whole Words Only

Whole elements of JavaScript syntax are searched, not sets of characters separated by spaces. The example contains seven words:

```
ActiveDocument.Sections["Query"].Limits["Cust_Id"].SelectedList.Count
```

Match Case

Locate instances of uppercase and lowercase letters, depending upon how the string or pattern is entered in Find What.

Use Pattern Matching

Pattern matching is implemented through regular expressions just as JavaScript recognizes them. Discussion of regular expressions is outside the scope of this guide, because regular expressions are a language unto themselves. See the resource on the internet called *WebReference.com*:

http://www.webreference.com/js/column5/

Strings and numbers usually represent themselves. Some characters have special meaning and are meta characters. To use these characters in a literal sense, escape them with a backslash (\).

Table 14 Character Examples

Character	Explanation
0–9a–zA–Z	The character
.	A character other than a new line
*	Zero or more occurrences of the previous item
+	One or more occurrences of the previous item
?	Zero or one occurrence of the previous item
\f	Form feed or new page
\n	New line
\r	Carriage return
\t	Tab
\/	/
\\	\
\.	.
\Xnn	An ASCII character specified by the hex nn
\w	A word character (a–zA–a0–9)
\W	A non-word character
\s	A white space character
\S	A non-white space character
\d	A digit 0–9
$	End of string (In this case, as the last character in an event handler)
^	Match start of string (In this case, the first character in an event handler)
|	Or = any of an alternate set
(...)	A grouping

Table 15 Pattern Examples

Pattern	Explanation
if|else	Match if or else
m.*(Parent|Name)	Match m followed by one or more characters and Parent or m followed by one or more characters and Name

Pattern	Explanation
Active.*Name	Match anything that contains the string Active, followed by one or more characters followed by Name
.*\}	Match a string that contains zero or more characters before a closing brace (}) (A brace is a special meta character in regular expressions, and it must be escaped with a backslash (\))
.+\}	Match a string that contains one or more characters before a closing brace (})
\{\r\n	Match a line that contains a brace ({) followed by a carriage return and a line feed
e\r\n|e$	Match a line that ends in an *e* or an event handler that ends in an *e*
\r\nf|^f	Match a line break that is followed by an *f*, or an event handler that starts with *f*

Dashboard Architect supports the Find and Replace utility for these symbols when pattern matching is *not* enabled.

Table 16 Symbols

Symbol	Function
^t	Matches a Tab character
^p	Matches a new line (CR/LF) (paragraph)
^^	Matches a "^" character

The Find Next Option

Use Find Next to move to the next instance of a search. The Find dialog box can be active while you are editing. To move to the next found item, click Find Next, , or the F3 key.

The Find All Option

If Find All is clicked, a Find operation is executed across the whole search context (event, section, or project) and one list of results is returned. To view the results click the Find tab in the output window.

The output window lists sets of items found in the project, and can be used as a navigation mechanism. Click a line of interest, and Dashboard Architect displays the event handler, the line in question, and highlights the found instance in the editing window.

Using the Floating Find Menu

The floating menu provides several Find operations. To find a simple string, highlight the string in the editing window. Right-click, and select the extent of the Find operation. For example, select Find In Event, Find In Section, or Find In Project. Dashboard Architect finds the specified string, and the results are presented in the output window.

Find Function is an example in floating Find menu that provides a quick way to find the definition of a function that is being called.

➤ To use Find Function:

1 **Highlight the text that corresponds to a function name.**

2 **Right-click, and select Find Function.**

Dashboard Architect prefixes the Find string with the word *function* and initiates a project-wide Find. The editing window displays the start of the function that is found, and the function is examined.

3 **After the examination is complete, return to the launch point of the find, by clicking .**

The Replace Feature

The Replace feature is similar to Find, it searches for text or patterns, and replaces the matched items with text in the Replace With drop-down list.

Access Replace by clicking , or selecting Edit, then Replace. The Replace dialog box is a subset of the Find dialog box.

Using the Printing Command

Several printing options are available in the Dashboard Architect Print command.

➤ To print:

1 **Select File, then Print to launch Select Print Range.**

2 **From Section to Print, select an option.**

3 **From Control to Print, select an option.**

4 **From Event to Print, select an option.**

5 **Optional: Select Print Line Numbers.**

6 **Click OK.**

7 **Optional: To configure the printer settings, click Setup.**

Using the Match Brace Feature

Use Match Brace to reduce the time spent identifying incomplete sets of braces. Complete sets of braces consist of these symbols:

- Braces {}
- Parentheses ()

- Brackets [ ]

➤ To use Match Brace:

1 **Place the cursor after an opening brace, press Ctrl+B, or select Edit, then Match Brace.**

Match Brace moves forward to find the opening brace from the current position. If no opening brace is found, the feature looks backwards until one is found. When an opening brace is located, the feature searches until it finds a matching closing brace, and highlights the extent of the brace pair.

2 **Repeat the operation by pressing Ctrl+B.**

If no closing brace is found, the operation ends without highlighting text.

The Auto-Code Feature

Dashboard Architect includes an Auto-Code feature that increase the speed of entering source code. To function correctly, Auto-Code must be enabled in Options. See "The Options Dialog Box" on page 27.

Auto-Code automatically adds closing quotation marks, brackets, and parentheses immediately after the opening character. The cursor waits in between the opening and closing characters for you to type. Entering the closing character overwrites the automatically added character.

Closing braces are added two lines down with a blank line in between the opening and closing braces. The cursor waits on the blank line, indented one tab.

Auto-Code enables a special class of macros that are triggered by the spacebar. See "Macros" on page 42. Table 17 lists and describes these Auto-Code macros that are available in Dashboard Architect. The caret position at the end of the expansion is substituted with **I**.

Tip:

To prevent Auto-Code macros from being expanded, press Ctrl+Space rather than space after the macro name. For example, `if` Ctrl+Space.

Table 17 Auto-Code Examples

Auto-Code	Expansion Description	Example

if	Expands to an `if` statement including braces, with the caret in parentheses	`if (`**`I`**`) {`
		
		`}`

else	Expands to an `else` statement including braces, with the caret on the blank line between the braces	`else {`
		    `I`
		`}`

while	Expands to a `while` loop including braces, with the caret in parentheses	`while (`**`I`**`) {`

Auto-Code	Expansion Description	Example
		`}`

for	Expands to a `for` loop including braces, with the caret in parentheses at the loop index initialization position	`for (`**`I`**`;;) {`
		
		`}`

function (or fn)	Expands to an empty function with the caret placed before the parentheses for the function, where the function name is placed	`function `**`I`**`() {`
		
		`}`

try	Expands to a `try and catch` block, with the caret on the first line of the `try` block indented one tab	`try {`
		
		    `I`
		
		`}`
		
		`catch(e) {`
		
		`}`

ad	Expands to `ActiveDocument` with the caret in the next character position	`ActiveDocument`**`I`**
cn	Expands to a `Console.Writeln` statement with the caret between the quotation marks of the message to be printed	`Console.Writeln("`**`I`**`")`

Macros

Macros are another means of entering source code quickly, and some features of Auto-Code are implemented with special macros.

Use macros to assign short character strings to commonly used snippets of code, so you enter the string and expand it into the code snippet. Macros can be simple, single line text substitutions such as *ad* and *cn,* or they can expand into complex multi-line code snippets, including replaceable parameters with default values.

Typical macros are invoked by entering the macro name and pressing Ctrl+P. The macro is expanded, only if the caret is immediately after the macro name, and the macro name is at the beginning of the line, or it contains at least one space before it.

Defining a Macro

Macros are defined in a text file called *macros.txt* that is in the config directory with the Dashboard Architect installation.

Each macro definition requires two lines. The first line contains the macro name, and the second contains the definition of the macro including parameters and caret control codes. To invoke the macro enter the macro name (the short character string) into the editing window. Each macro must have a unique name. Auto-Code macros must have names that begin with *internal.*

Caution!

The `macros.txt` file must end with a blank line.

Simple Macros

This topic defines simple macros that take no parameters, beginning with single-line macros and introducing caret control codes for macros whose output spans multiple lines.

The *ad* Macro

The *ad* macro is a simple macro that expands from *ad* to *ActiveDocument* and positions the caret at the end of the word.

The definition in the macros.txt file is:

```
ad
ActiveDocument
```

When *ad* is entered, and Ctrl+P is pressed, *ad* is replaced by *ActiveDocument.* The caret remains at the end of the text entered by the macro expansion.

The *cn* Macro

The *cn* macro expands *cn* to *Console.Writeln("")* and demonstrates simple caret control codes by positioning the caret inside the quotation marks.

The definition in the macros.txt file is:

```
cn
Console.Writeln{(}""{)}{LEFT 2}
```

The definition contains three places where characters are surrounded by braces.

Each parenthesis must be enclosed in braces, because each has a special meaning. The macro expansion feature is alerted that these characters are to be part of the expansion rather than using their special meaning.

Note:

Switching off the special meaning of a character, when it is enclosed in braces, is called *quoting*.

The `{LEFT 2}` expression at the end of the macro definition, means move the caret left two characters, as if the left-arrow had been pressed twice. These caret control codes may be placed anywhere within the macro definition and are executed as they are located.

See "Macro Control Codes" on page 46, for a complete list of caret control codes.

Multiple-Line Macros

This topic demonstrates how caret control codes are used to define macros whose output spans multiple lines. The *for* macro is an example.

The *for* macro expands into a simple *for* loop including braces.

The definition in the macros.txt file is:

```
for
for {(};;{)} {{}{ENTER 2}{}}{UP 2}{END}{LEFT 5}
```

As in the *cn* macro, the parentheses and braces are *quoted.*

The control code `{LEFT 5}` at the end of the macro means move the caret left five characters, and other control codes are introduced.

`{ENTER 2}` means press the Enter key twice, and leave a blank line between the two braces.

`{UP 2}` is similar to `{LEFT 2}`, press the up arrow key twice.

`{END}` press the End key to move the caret to the end of the current line.

Therefore, the overall effect of the macro is seen by reading it from left to right, imagining the special keys being pressed where the caret control codes are located.

The macro expands with the caret between the opening parenthesis and the first semicolon; that is, where the loop index variable is initialized.

```
for (;;) {

}
```

Macro Parameters

Macro definitions can include parameters. The text given with the macro name is used in the macro expansion enabling the same macro to expand differently each time it is invoked.

This topic discusses defining macros that take parameters, and how to invoke macros that require parameters.

A *for* Macro with a Loop Index Parameter

Most *for* statements start with "for (var *x* = 0;...)," where *x* is a variable identifier that is used within the *for* loop. If *for* loops are nested, the variable identifier must be unique for each loop inside another loop.

The *for* macro presented previously is improved to provide the loop index variable initialization, and to enable a different variable identifier to be used in nested loops:

```
for
for {(}var ${1} = 0;;{)} {{}{ENTER 2}{}}{UP 2}{END}{LEFT 4}
```

In the macro definition, the loop index variable identifier is given as `${1}`, which stands for the value of the first parameter supplied to the macro.

Macros can take up to nine parameters called *`${1}`* to *`${9}`*. Macro parameters are inserted anywhere in a macro definition except inside other special codes enclosed in braces, such as caret control codes. Each macro parameter is used as many times as necessary. For example, the macro definition can be modified to add the loop index variable increment code automatically, by placing `${1}{+}{+}` after the second semicolon.

```
for
for {(}var ${1} = 0;; ${1}{+}{+}{)} {{}{ENTER 2}{}} {UP 2}{END}{LEFT 8}
```

The plus signs are *quoted* because they have a special meaning within the macro definition. Macro definitions must be on one line.

Default Values *for* Parameters

The previous macro example requires the value for the parameter to be supplied each time the macro is invoked. It is only necessary to change the value of the parameter when *for* loops are nested. The macro definition can be improved further by giving the parameter a default value so it is not specified unless the default value cannot be used (such as in nested *for* loops).

A macro parameter is given a default value by specifying the value the first time the macro parameter is displayed in a macro definition:

```
for
for {(}var ${1=a} = 0;; ${1}{+}{+}{)} {{}{ENTER 2}{}} {UP 2}{END}{LEFT 8}
```

The syntax `${1=a}` assigns *`a`* as a default value to the parameter `${1}`. If the macro is invoked with no value for `${1}`, it expands to read:

```
for (var a = 0;; a++) {

}
```

Invoking Macros

In general, to invoke macros, enter the name as a stand-alone word (that is, at the beginning of a line or with at least one space in front of the macro name), and press Ctrl+P.

Invoking Macros without Giving Parameter Values

If a macro takes no parameters or all the parameters are given default values in the macro definition, a macro is invoked as described in the previous paragraph.

The macro name is removed and replaced with the expanded macro definition.

Invoking Macros Giving Parameter Values

If a macro contains parameters that contain no default value, or the default value is not sufficient, give the value of each parameter by appending it to the macro name separated by colons:

`for:b`—Sets the value of parameter 1 to *b* `for:in_startVal`—Sets the value of parameter 1 to *in_startVal*

When macros contain multiple parameters and some parameters contain default values, the parameters with default values need not be specified. For example, a macro *test* that takes three parameters, where the first and third parameter contain default values assigned in the definition, can be invoked in these ways:

`test:a:b:c`—Assigns the value *a* to parameter 1, *b* to parameter 2, and *c* to parameter 3

`test:a:b`—Assigns the value *a* to parameter 1, *b* to parameter 2, and parameter 3 takes the default value

`test::b:c`—Parameter 1 takes the default value, *b* is assigned to parameter 2, and *c* to parameter 3

`test::b`—Parameters 1 and 3 take the default values, and parameter 2 is assigned a value of *b*.

The parameters with default values that are followed by parameters *without* default values must include a colon as a placeholder. Parameters at the end of the list, which contain default values, need not be included.

Macro Control Codes

Macro control codes enable the use of non-printing keys, such as Enter or the arrow keys, in macro definitions.

The plus sign (+), caret (^), percent sign (%), tilde (~), and parentheses ( ) have special meanings in macro definitions. To specify one of these characters, enclose it with braces. For example, to specify the plus sign, use {+}. Brackets {[ ]} have no special meaning but must also be *quoted* because of the routines that are used to implement macros.

To specify brace characters, use {{} and {}}.

To specify characters that are not displayed when you press a key, such as Enter or Tab, and action keys use the codes displayed in Table 18.

Table 18 Macro Control Codes

Key	Code
Backspace	`{BACKSPACE}, {BS}, or {BKSP}`
DEL or Delete	`{DELETE} or {DEL}`
Down arrow	`{DOWN}`
End	`{END}`
Enter	`{ENTER} or ~`
Home	`{HOME}`
INS or Insert	`{INSERT} or {INS}`
Left arrow	`{LEFT}`

Key	Code
Right arrow	`{RIGHT}`
Tab	`{TAB}`
Up arrow	`{UP}`

To specify keys used in combination with the Shift, Ctrl, and Alt keys, precede the key code with one or more of the codes in Table 19.

Table 19 Combination Key Codes

Key	Code
Shift	+
Ctrl	^
Alt	%

To specify a combination of Shift, Ctrl, and Alt to be held down while several other keys are pressed, enclose the code for those keys in parentheses. For example, to specify that the user must hold down Shift while *E* and *C* are pressed, use `"+(EC)"`. To specify that the user must hold down Shift while *E* is pressed, followed by *C* without Shift, use `"+EC"`.

To specify repeating keys, use the form `{key number}`, and insert a space between *key* and *number*. For example, `{LEFT 42}` means press the left arrow key 42 times; `{h 10}` means press *h*, 10 times.

Importing Sections from other Interactive Reporting Documents

The Dashboard Architect Import Utility enables other Interactive Reporting documents to be imported into the current document. It eliminates the re-creation of identical information in other documents.

➤ To use the Dashboard Architect Import Utility:

1. **Select Project, then Import.**

   Import is displayed.

2. **Click (next to Secondary Document), and locate the Interactive Reporting document to be imported.**
3. **Click OK.**
4. **Optional: To specify one or more sections to import, select Select and Re-order Sections.**

   Select and Re-order Sections is displayed.

5. **If Select and Re-order Sections is displayed, perform an action to move sections between Available Sections and Selected Sections:**

- Select one or more sections, and click > or <
- Click » or « to move all sections
- Double-click a section

All dependent sections are also selected and moved.

6 **Optional:** Select **Remove Duplicate Images from the Final Document** to consolidate duplicate images into the Resource Manager.

All instances of an image are changed to reference the single copy of the image in the Resource Manager. The Interactive Reporting document file size and memory footprint are reduced, which improves the loading speed, and makes reuse of existing images from the Resource Manager possible when creating dashboards and reports. See “Resource Manager” in the *Hyperion Interactive Reporting Studio User's Guide.*

7 **Optional:** Reorder **Selected Sections** by clicking ⬆ or ⬇.

8 Click **OK**.

The sections are imported into the open project in Dashboard Architect.

A report is displayed detailing the imported sections, and if one or more of the sections from the secondary Interactive Reporting document required renaming.

9 **Optional:** Save the report for future reference.

The imported sections are instrumented and become part of the project.

Unicode Functionality

The Dashboard Architect Import Utility automatically converts old code page based Interactive Reporting documents to Unicode before use, enabling documents with different languages to be imported. Only the resulting documents are converted, originals remain unchanged.

6 The Testing and Debugging Processes

In This Chapter

Testing and Debugging

Testing and debugging activities are closely associated with editing. Testing is often accompanied by making corrections which involves editing. Editing is not complete until the code being edited is tested and performs as required. See "Editing" on page 35.

About Testing

Testing is accomplished by placing Dashboard Architect in the background and working with the `instrumented.bqy`. The user clicks the buttons, processes queries, selects items from lists, and selects options and check boxes. Each of these clicks causes events to be fired. The event handler that Interactive Reporting Studio calls, in turn calls the Debug Server and requests the JavaScript, which is returned as a string. The event handler executes the string, using the JavaScript `eval()` function, and so `instrumented.bqy` behaves similarly to an ordinary Interactive Reporting document.

Testing Rule

A simple rule must be remembered when working with Dashboard Architect.

JavaScript code is sent to an event handler when, and only when, that handler calls the Debug Server and requests it.

Procedural and Declarative JavaScript

Two distinct types of JavaScript exist, JavaScript that is declarative and JavaScript that is procedural. Declarative code acts to set up or declare shared or public resources such as global variables, properties, and functions that can be called later using the public handles from procedural code. Procedural code acts immediately on shared or public, or local or private resources.

The rule operates equally to the sets of code, but declarative code is usually associated with `OnStartUp()` events or events in hidden objects that the user does not typically see.

Changes to procedural code require that the event they are declared in be fired before the change is observed. Procedural code changes occur immediately in real time. The code associated with the `OnClick()` event of a button can be changed. In `instrumented.bqy,` click the button to run with the new code and observe the behavior.

Changes to declarative code also require the event they are declared in be fired before the change is observed. Causing an event that uses the results of a declaration is not the same as causing the event that creates the declaration. Consequently, changes to declarations require re-declaration before calls on the re-declarations can display the changes made in Dashboard Architect.

Consider example 1:

	MyEIS.Button_X
Instrumented.bqy	Dashboard Architect

eval(<QIQDebug_call>)	ListBox1.RemoveAll()
	ListBox1.Add(“select a Country”)
	ListBox1.Add(“Australia”)
	ListBox1.Add(“Britain”)
	ListBox1.Add(“France”)
	ListBox1.Add(“Germany”)
	ListBox1.Add(“United States”)

When the user clicks Button_X on the MyEIS section, the Debug Server is called and returns the JavaScript held by Dashboard Architect for the `OnClick()` event of Button_X.

As soon as the code is changed, the button can be clicked again. The Debug Server is called and finds the new code. It returns the code to `instrumented.bqy`. The code is passed to the JavaScript `eval()` function. The new behavior is immediately observed.

An example is illustrated in the section called *Query EIS,* in the `Arch_tut.bqy` that is provided as a sample with the Dashboard Architect installation.

Consider example 2:

	OnStartUp
Instrumented.bqy	Dashboard Architect
eval(<QIQDebug_call>)	var eis=ActiveDocument.Sections[“MyEIS”] eis.Shapes[“Button_X”].OnClick()
	MyEIS.Button_X

Instrumented.bqy	Dashboard Architect

eval(<QIQDebug_call>)	function fill_ListBox(lbx){
	lbx.RemoveAll()
	lbx.Add("select a Country")
	lbx.Add("Australia")
	lbx.Add("Britain")
	lbx.Add("France")
	lbx.Add("Germany")
	lbx.Add("United States")
	}
	ActiveDocument.fill_ListBox=fill_ListBox

	MyEIS.Button_Y
Instrumented.bqy	Dashboard Architect
eval(<QIQDebug_call>)	ActiveDocument.fill_ListBox(ListBox1)

1. Upon start up, the document calls the Debug Server for code. For example, the `OnClick()` event of Button_X is caused, creating a function called *fill_ListBox* available as a property of `ActiveDocument`.
2. When a user clicks Button_Y on the MyEIS section, the `fill_ListBox` function is called and identical behavior occurs as in example 1.
3. A change is made to the JavaScript in the `OnClick()` event of Button_X.
4. Button_Y is pressed.
5. Unlike example 1, the code is executed as if no changes were made, when in fact Button_X was just changed. The reason is because unless the `OnClick()` event of Button_X (or the `OnStartUp()` event of the document) is caused, the changes to the code of Button_X are not sent to `instrumented.bqy`. Therefore, Button_X must be clicked first and *then* Button_Y for the new code to be effectual.

The (Fire Current Event) is provided for the purpose of executing the event associated with the JavaScript currently in view.

It is vital to remember the distinction between simple procedural code (statements that perform operations directly as in the first example) and declarative code (code that creates functions to be called and used later) as in the second example.

It is highly recommended to open the Console window, when is clicked so errors are seen and reported. If an error occurs, the cog associated with the failed JavaScript event handler, changes to red () in the Code Requests pane of the output window.

About Debugging

Debugging is closely associated with testing and editing. Debugging involves determining why a behavior occurs and how to change it. Traditional debugging includes one of these options to determine what is occurring:

- `Console.Writeln()` statements are interspersed with regular code
- Debug functions are interspersed with regular code

In the first option, the code must be removed after the problem is located. In the second option, the code can stay, because debug functions can be turned on and off providing permanent instrumentation. While the second option is the better option, both have the disadvantage that manual instrumentation is not flexible, and is used only when identifying a specific situation that was anticipated by the programmer when the code was originally written.

Breakpoints

In Dashboard Architect, setting breakpoints makes debugging easier than the two traditional approaches.

Breakpoints are required when one or more areas of code come under suspicion. The cursor is positioned on a strategic line and a breakpoint is inserted by clicking or F9.

Internally, a breakpoint is implemented as a set of additional dynamic instrumentation instructions alongside the regular JavaScript. As a breakpoint is extra JavaScript, it behaves just like other code changes. If it is placed inside declarative code such as a function, you must click so the breakpoint can become part of the declaration for the function.

When all breakpoints are set, move to `instrumented.bqy`, and activate the event (for example, clicking a button). The code is executed and at some point it encounters the breakpoint. The `instrumented.bqy` calls the Debug Server and requests that execution is suspended and highlights the line associated with the breakpoint.

The Debug Server keeps Interactive Reporting Studio suspended and hands control to the user interface of Dashboard Architect, so the execution state of the application can be examined. The evaluate pane in the output window is activated, and JavaScript expressions can be entered for evaluation.

To see the value of a local variable, enter the name of the variable, and click . To see the value of an Interactive Reporting Studio property, enter it into the evaluate pane. As a shortcut, highlight text in the editing window, right-click, and click Evaluate, or copy and paste the text into the evaluate pane and modify the expression as required. To clear the evaluate pane, click .

If incorrect syntax is entered, the JavaScript engine generates a syntax error and aborts the current thread of execution. Return to `instrumented.bqy` and recommence the test by clicking the control to reactivate the breakpoint.

Limitations exist as to where breakpoints can be placed in code. Breakpoints are implemented, as JavaScript is inserted, so they must be syntactically valid. Dashboard Architect does not have direct access to the JavaScript engine and therefore cannot guarantee the correctness of breakpoints. These guidelines help with writing code that is straightforward to maintain, read, and set breakpoints upon.

Table 20 Breakpoints Working With JavaScript

JavaScript	Comment
Line 01 function f(param){	Break occurs when the function is being parsed, not when it is called
Line 02 var x	
Line 03 if (condition){	
Line 04 statement_1	
Line 05 }else{	
Line 06 statement_2	
Line 07 }	
Line 08 switch (x){	
Line 09 case 1:	Breakpoint is not OK as nothing can come before *case*
Line 10 statement_3	
Line 11 break	
Line 12 case 2:	Breakpoint is not OK as nothing can come before *case*
Line 13 statement_4	
Line 14 break	
Line 15 default:	Breakpoint is not OK as nothing can come before *default*
Line 16 }	
Line 17 // comment	
Line 18 }	

As seen in Table 20, only a few distinct and recognizable places exist where a breakpoint *cannot* be placed. If the code is not written in this manner, the options for breakpoints are restricted. Examples are provided in Table 21.

Table 21 Areas Where Breakpoints Do Not Work With JavaScript

JavaScript	Comment
Line 01 function f(param){	

JavaScript	Comment
Line 02 var x	
Line 03 if (condition)	
Line 04 {statement_1}	Breakpoint is not OK
Line 05 else	Breakpoint is not OK
Line 06 {statement_2}	Breakpoint is not OK
Line 07 switch (x){	
Line 08 case 1: statement_3;break	Breakpoint is not OK
Line 09 case 2: statement_3;break	Breakpoint is not OK
Line 10 default:	Breakpoint is not OK
Line 11 }	
Line 12 // comment	
Line 13 }	

7 Adding and Removing Objects

In This Chapter

Interaction with Interactive Reporting Studio

Dashboard Architect interacts with Interactive Reporting Studio through a COM programming interface. The interface defines the operations that Interactive Reporting Studio can perform on behalf of its clients.

The creation of objects (such as buttons, drop-down lists, check boxes, lists, and option buttons) and the addition of JavaScript into event handlers are operations that are not currently available through programmatic means. You must perform these operations inside Interactive Reporting Studio.

Operations that can be performed in only one environment are problematic, because two sets of structures are required. Interactive Reporting Studio contains one set of structures in `instrumented.bqy`, and Dashboard Architect holds the other. Unless these two structures are in synch, correct operation cannot be provided.

Operations that must be performed through the user interface of Interactive Reporting Studio must also be incorporated into the development environment of Dashboard Architect.

Resynchronizing

The simplest way for Dashboard Architect to incorporate objects is through resynchronizing, which can be initiated by selecting Project, then Resynchronize.

Resynchronize works in a similar way to the create project operation, see "Creating Projects" on page 29. It closes `instrumented.bqy`, and performs these actions:

1. `Instrumented.bqy` is disassembled.
2. `Instrumented.bqy` is analyzed.
3. Objects that exist in `instrumented.bqy` but not in Dashboard Architect are added. These are new objects created with the Interactive Reporting Studio user interface. The JS files are extended to include references to these objects. JavaScript added by the user is transferred to the JS files and is replaced by instrumentation code.
4. Objects that exist in Dashboard Architect but not in the `instrumented.bqy` are removed. Dashboard Architect removes all references to the deleted objects and they cease to exist.
5. `Instrumented.bqy` is reassembled.
6. `Instrumented.bqy` is reopened in Interactive Reporting Studio.

Resynchronization may take a few minutes to complete depending on the size of the project. It is the simplest operation from the viewpoint of the user, but it may not be the most convenient as `instrumented.bqy` must be closed, analyzed, and reopened.

If an object is renamed in Interactive Reporting Studio, the old object is deleted in Dashboard Architect (as it no longer exists in Interactive Reporting Studio), and a blank object is created. JavaScript associated with the old object is lost.

Adding Controls

Add a control by duplicating or creating a control.

Duplicating Controls

The quickest way to add a control is to duplicate one that exists and is instrumented.

➤ To duplicate a control:

1 **In Interactive Reporting Studio, and press Ctrl+D to enter Design mode.**

2 **Select a control and duplicate it.**

3 **Double-click the control.**

Properties is displayed.

4 **Enter an object name, and click OK.**

5 **Press Ctrl+D to exit Design mode.**

6 **Click the control.**

Dashboard Architect gets a call from the control for code. Dashboard Architect fails to find a reference to the control so it creates an event handler on the fly, and immediately synchronizes itself with the state of `instrumented.bqy`.

Creating Controls

If no controls are available to be duplicated, a control without instrumented code must be created.

➤ To create a control:

1. **Select Project, then Add Control.**

   Add Control is displayed.

2. **Click Copy to Clipboard to copy the instrumentation code.**
3. **Click Cancel.**
4. **In Interactive Reporting Studio, press Ctrl+D to enter Design mode.**
5. **Create the control and paste the instrumentation code into each of the event handlers.**
6. **Press Ctrl+D to exit Design mode.**
7. **Click the control.**

   Dashboard Architect gets a call from the control for code. Dashboard Architect fails to find a reference to the control so it creates one, and immediately synchronizes itself with the state of `instrumented.bqy`.

Deleting Controls

A control can be deleted in Interactive Reporting Studio. After deleting the control, resynchronize immediately, or wait until an Interactive Reporting document is next created, as Dashboard Architect performs a Resynchronize operation every time it makes an Interactive Reporting document.

Renaming Controls

Renaming a control is problematic, as Dashboard Architect perceives that a control is deleted and another control is added. JavaScript held for the control under the old name must be manually cut and pasted into the event handler for the new control.

Adding and Duplicating Sections

Adding and duplicating sections relates only to dashboard sections. Other sections do not contain JavaScript and are outside the domain of Dashboard Architect.

➤ To add or duplicate a section:

1. **Select Project, then Add Section.**

   Add Section is displayed.

a. To add a section, select **Create a New Section**.

b. To duplicate a section, perform an action:

- Select Duplicate an Existing Section
- From the navigation panel, select a section node, right-click, and select Duplicate

2 **Enter a Name for the new or duplicated section, and click OK.**

The section is added or duplicated.

Dashboard Architect provides for the creation of sections in these ways:

1. The new section is added programmatically (or copied from a section if duplicated).
2. `Instrumented.bqy` is closed.
3. `Instrumented.bqy` is disassembled.
4. Instrumented JavaScript is added to the `OnActivate()` and `OnDeactivate()` events.
5. `Instrumented.bqy` is reassembled.
6. `Instrumented.bqy` is reopened in Interactive Reporting Studio.

A matching section with all basic event handlers is added to Dashboard Architect. If the section is duplicated, a copy of all original event handlers is inserted into the node in the navigation panel. The operation is relatively fast.

Renaming Sections

Renaming sections relates only to dashboard sections. Other sections do not contain JavaScript and are outside the domain of Dashboard Architect.

➤ To rename a section:

1 **Select Project, then Rename Section.**

Alternatively, from the navigation panel, select a section node, right-click, and select Rename.

Rename Section is displayed.

2 **From Current Name, select a section.**

3 **Enter a New Name.**

4 **Click OK.**

The section is renamed.

Deleting Sections

Deleting sections relates only to dashboard sections. Other sections do not contain JavaScript and are outside the domain of Dashboard Architect.

➤ To delete a section:

1 **Select Project, then Delete Section.**

Alternatively, from the navigation panel, select a section node, right-click, and select Delete.

Delete Section is displayed.

2 **In Delete Section, highlight one or more sections.**

3 **Click OK.**

The sections are removed.

8 Documentation

In This Chapter

Documentation of Code

Good documentation enhances reuse of code and makes code easier to maintain.

Dashboard Architect provides features that enable you to document code easily and consistently, and to extract documentation from the code to be published in other forms.

Features are provided for documenting variables, functions, classes, and components. A special format of comment is used to recognize formal documentation, and a number of tags are supplied to define information such as the type of a variable (@type), parameter, or function, the formal parameters of a function (@param), and the visibility of a variable (@scope).

Documentation Comments

Comments to be published in the documentation must be entered in the format of block comments that start with /** and end with */.

Documentation comments follow the general form of one or more paragraphs of descriptive text, the first of which is used when a summary is required, followed by special tags. Paragraphs within the descriptive text are marked by a blank line. Documentation tags are placed after the general description, and descriptive text after a tag must be in one paragraph with no blank lines.

For example, a documentation comment to describe the function `multiply`, which takes two parameters, multiplies them, and returns the result, is structured as follows:

```
/**
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 */
function multiply(in_intA, in_intB) {...
```

The function is described in two paragraphs. The first is a summary of the functionality. Also described are the names and suggested parameters types, and how to interpret the return value.

Documentation of Variables

To document a variable, describe how and what the variable does, and provide a type so users know the acceptable data type for the variable.

The only documentation tags to be used in a variable comment are @type to specify the typical variable, and @scope to define the visibility of a variable. See "Namespace Scope of Entities and the @scope Tag" on page 65. The tags are optional, but it is recommended to use the @type tag. If the @scope tag is not used the documentation system assumes the variable is not visible outside the current event handler.

Variable documentation comments must be placed immediately before the variable declaration or immediately before the variable in an assignment statement, in which case the variable on the left side of the equals sign is assumed to be the variable that is being documented and the variable name may only have white space before it on the line.

A simple documentation comment for a variable.

```
/**
 * Points to the BQY Section that contains this shape.
 * objSection can be used to access other shapes within the
 * dashboard section.
 *
 * @type Section
 */
var objCurrentSection = this.Parent
```

Variable documentation is useful when variables are exposed at a section level or as a property of a class, so they are used outside the event handler in which they are defined.

Documentation of Functions

Documentation of functions enable programmers to determine which functions are available, the expected parameters, and what they will return.

The @param tag is used for variables, and to declare the parameters of the function, the suggested type, and a description. When used in a function documentation comment, the @type tag declares the typical return type of the function. A function documentation comment must be placed immediately before the function declaration. An example of a documentation comment for a function is provided in "Documentation Comments" on page 61.

Documentation of Classes

Classes are collections of related functions (member functions) that can share and expose a persistent state by means of properties (variables). Member functions and properties can be documented with the techniques described in the previous two topics.

A member function or variable of a class must be given an @scope tag with the name of the class as the value. Thus binding the function or variable to the class definition.

The function that defines the class is the class constructor. These are documented like other functions in the class, the @scope tag illustrates that they belong to the class, and the @type tag shows that they return an instance of the class.

The fact that the functions and variables are related to each other by membership to the class makes it necessary to add an extra level of documentation that binds them and gives an overview of what the class is used for.

The *class* comment provides this overview. The *class* comment can be anywhere in the source code, but it is useful when placed before the class constructor comment. A *class* comment is created using an @class tag with the class name as the first word after the tag. A *class* comment for a persistent property bag is illustrated with the constructor comment.

```
/**
 * The PropertyBag class is used to store, retrieve, and access
 * a set of persistent properties.
 *
 * Properties are stored in a Shape, which must be a text label
 * or text box.
 *
 * Properties are key/value pairs, where both the key and the
 * value must be Strings.
 *
 * @class PropertyBag
 */

/**
 * The PropertyBag constructor creates a new PropertyBag
 * instance.
 *
 *
 * If the in_objPersistenceShape parameter is not given the
 * PropertyBag is initialized to be empty.
 *
 * If the in_objPersistenceShape parameter is given, the
 * constructor attempts to load the property values from the
 * given shape.
 *
 * @param in_objPersistenceShape Shape an option parameter that
 * can be used to define the Shape to load the property values
 * from.
 * @type PropertyBag
 * @scope PropertyBag
 */
function PropertyBag(in_objPersistenceShape) {

   /**
```

```
     * Returns the value of the property identified by
     * in_strKey.
     *
     * If the PropertyBag does not contain a value for the
     * given key it will return null unless the optional
     * in_strDefaultValue parameter is supplied, in which
     * case that parameter value is returned.
     *
     * @param in_strKey String the key of the property
     * whose value is to be returned
     * @param in_strDefaultValue String an optional default
     * value to be returned if the PropertyBag does not
     * contain the property identified by in_strKey.
     * @type String
     * @scope PropertyBag
     */
   function getProperty(in_strKey, in_strDefaultValue) {
      ...
   }
}
```

In the example, the @class tag in the first comment marks it as the class overview comment; the @type and @scope tags in the function comment for the `PropertyBag` function reference back to the @class value of the class overview comment, and mark the function as the class constructor. The comment for the `getProperty` function contains an @scope tag that links it to the `PropertyBag` class, but the @type tag shows it returns a string, so the documentation system knows it is a member function and not a constructor.

Note:

Classes may expose member functions or properties that are not defined within the body of the class constructor, using an assignment statement that references a function or variable declared outside the constructor. In this case, the @scope tag still works to bind the externally declared function or variable to the class.

Documentation of Dashboard Development Services Components

The Dashboard Architect documentation system can provide extra information about Oracle's Hyperion® Dashboard Development Services components, and list them in another section of the documentation.

Each Dashboard Development Services component contains a number of text labels within sections that contain information about the component and the runtime requirements. These text labels are `txlName`, `txlReqBrio`, `txlReqTemplate`, `txlDescription`, and `VersionInfo`. See “Component Fingerprints” in the *Hyperion Interactive Reporting – Object Model and Dashboard Development Services Developer's Guide, Volume 7: Component Reference Guide.*

Dashboard Architect reads and formats the values of these text labels for use in the component documentation page, and provides a summary of the components at the top of the page.

Caution!

For the feature to work, `instrumented.bqy` must be loaded in Interactive Reporting Studio. If the Interactive Reporting document is not loaded, Dashboard Development Services components are treated like a standard dashboard.

Namespace Scope of Entities and the @scope Tag

The visibility of a function or variable and how to describe it are vital concepts to understand when using the Dashboard Architect documentation system.

In JavaScript, variables (including variables that are pointers to functions) may be visible in various scopes. In the context of JavaScript within Interactive Reporting Studio, the default scope for a variable is to be visible within the current event handler, anywhere after it is declared or first used.

It is common practice to expose a variable to other event handlers by attaching the variable to the parent section of the event handler or to the active document.

The @scope tag enables the visibility of a variable to be documented, making it clear in the generated documentation how the variable is accessed or how the function is called. Previous examples have illustrated one use of the @scope tag in associating functions and variables with classes.

For example, if a variable is defined and used in an event handler and not exposed, no other event handler can use it. If a variable is defined and attached to the parent section object, other event handlers can access that variable through the parent section. If a variable is defined and attached to the active document, other event handlers can access that variable by name because it is visible at the highest level. These examples illustrate different levels of visibility and how to use the @scope tag to document the levels.

```
/**
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 */
function multiply(in_intA, in_intB) {...
}
```

In the previous example, the function is not visible anywhere other than the current event handler. Because code in other event handlers cannot generally call the function. For this reason, no @scope tag is provided and the documentation system considers this to be a *local* function and it is not included in the documentation unless control level documentation is requested.

```
/**
```

```
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 * @scope Section
 */
function multiply(in_intA, in_intB) {...}
this.Parent.multiply = multiply
```

The previous example illustrates that the function is exposed through the parent section of the correct shape by the assignment after the closing brace of the function. Because code in other event handlers can easily call the function if they contain a reference to the section. The documentation comment explicitly provides an @scope tag with a scope of *Section.*

```
/**
 * Returns the result of in_intA * in_intB.
 *
 * If either parameter is not a number, the result is
 * undefined.
 *
 * @param in_intA Integer the number to be multiplied by
 * in_intB
 * @param in_intB Integer the number to be multiplied by
 * in_intA
 * @type Number the result of in_intA * in_intB
 * @scope Document
 */
function multiply(in_intA, in_intB) {...}
ActiveDocument.multiply = multiply
```

Finally, the preceding example shows a function that is available to an event handler in the current document by naming the function with no namespace qualifiers. The documentation comment explicitly provides an @scope tag with a scope of *Document.*

The @scope tag is necessary because JavaScript is type-less, and because the exposure of a function or variable through an assignment need not be done immediately after the function is declared. No reliable method exists for the Dashboard Architect documentation system to determine the visibility of a function or variable from the source code alone.

Documentation Grouping Using the @scope Tag

The @scope tag may be given a second parameter, which is used to define other groups for parts of the documentation. For example, the feature is used to differentiate the public and private parts of an API of an object—when the documentation is generated for use by in-house developers the private API calls are included in the documentation. When external developers generate documentation they can exclude the private API calls to ensure they do not call private methods by mistake.

The feature imparts no impact on the code visibility—the notions of public and private are a logical convenience. Other groupings are possible, with different parameters.

These are examples of the use of the parameter:

```
/**
 * A private API function that should not be called from
 * outside the class it is defined in.
 *
 * @type void
 * @scope SomeClass private
 */
function privateAPICall() {...
}

/**
 * This function does something useful and can be called
 * by any code with a reference to an object of this
 * class.
 *
 * @type Integer
 * @scope SomeClass public
 */
function publicAPICall() {...
}
```

See “Generating Documentation” on page 68.

Documentation Comment Tag Reference

This topic illustrates the information that is provided to each documentation tag.

@class classname [documentation_group]

The @class tag must be given a class name. The name that is used to bind the constructor, member functions, and properties back to the class.

The optional *documentation_group* is one word used to group sets of classes, functions, and variables so they can be included or excluded from the generated documentation as a group. See “Generating Documentation” on page 68.

@param param_name type_name [description]

The @param tag must be given the name of the parameter, and its expected type (such as number, section, object, shape, and so on). As JavaScript is type-less the parameter type can only be a suggestion but it enables callers to see what types of values are expected by the function.

The optional *description* may be given on multiple lines, but cannot contain blank lines.

@type type_name [description]

The @type tag must be given the type of the variable or the return type of the function being documented (such as number, section, object, or shape).

If documenting a function the optional *description* is given next to the return type of the function, so the caller can see how to interpret the return value.

@scope namespace [group]

The @scope tag may be given a namespace of *Document*, *Section*, *Control*, or the name of a class defined with an @class tag.

The optional *group* is one word used to group sets of classes, functions, and variables so they can be included or excluded from the generated documentation as a group. See "Generating Documentation" on page 68.

Dashboard Development Services Component-Specific Features

The documentation feature recognizes Dashboard Development Services components and includes additional information regarding those components.

Dashboard Development Services component-specific information is held in text labels on the component code section. All additional information is optional. The information in Table 22 may be specified for a Dashboard Development Services component.

Table 22 Dashboard Development Services Component-Specific Information in Text Labels

Text Label	Information
txlName	Component display name (For example, dynamic headings rather than `QIQ_fmtHeading`)
txlDescription	One or more paragraphs of descriptive text about the component

txlReqBrio	Interactive Reporting Studio release information in the format:
	"VersionMajor=*a*\nVersionMinor=*b*\nVersionRelease=*c*\nVersionPatch=*d*" where *a, b, c,* and *d* are numbers, and \n represents a new line sequence. For example,
	VersionMajor=8
	VersionMinor=3
	VersionRelease=0
	VersionPatch=647

txlReqTemplate	Dashboard Studio template release information in the format:
	"VersionMajor=*a*\nVersionMinor=*b*\nVersionRelease=*c*" where *a, b,* and *c* are numbers, and \n represents a new line sequence. For example,
	VersionMajor=8
	VersionMinor=3
	VersionRelease=45

Generating Documentation

To create HTML documentation, select Project, then Generate Documentation.

Inclusion and Exclusion of Documentation Groups

If documentation groups are defined using the second parameter of the @scope tag. The Select Documentation Groups dialog box is displayed that enables the selection of documentation groups to be included in the HTML output.

Select one or more of the documentation groups, and click OK. If no documentation groups are selected, or if Cancel is clicked, only documentation without a documentation group specifier (that is, an @scope tag with one parameter) is included in the HTML documentation.

Inclusion and Exclusion of Unscoped Documentation

Documentation blocks that exclude an @scope tag are not included in the generated HTML documentation, unless Project, then Include Control Documentation is selected. See "The Project Menu" on page 24.

If the command is selected, the unscoped documentation is included at the end of the section of the documentation, grouped by shape.

Dashboard Development Services Component-Specific Features in HTML Documentation

If `instrumented.bqy` cannot be located in the list of open Interactive Reporting documents, a warning is displayed.

The HTML documentation is generated, but the documentation generator cannot distinguish Dashboard Development Services components from other sections, so Oracle's Hyperion® Dashboard Development Services component-specific information is not included in the generated documentation.

9 Using the Dashboard Development Services Update Utility

In This Chapter

About Dashboard Development Services Update Utility

You use the Dashboard Development Services Update Utility to update JavaScript in Interactive Reporting documents, provided that the JavaScript is designed to create a layered architecture using dashboard sections.

- **Layer 1**—Data and views (queries, charts, pivots, and reports)
- **Layer 2**—Sections with which you interact; charts, pivots, and user interface controls (lists, buttons, drop-down lists, and so on that contain very simple, one-line calls to Library routines)
- **Layer 3**—Sections that contain reusable JavaScript functions, classes, and components

The Dashboard Development Services Update Utility updates only the contents of layer 3. It uses a `newsections.bqy` document (an Interactive Reporting document that contains the latest layer 3 sections) to *push* new layer 3 sections into *old* dashboards, thus converting *old* dashboards into *new* dashboards.

Within documents that are being updated, sections that are common to the current document and the `newsections.bqy` document are replaced by sections from the `newsections.bqy` document. As of Release 8.3.1, release information within sections is used to ensure that *earlier* sections do not replace *later* sections. Therefore, as of Release 8.3.1, a section is replaced only if its corresponding `newsections.bqy` section references a more recent release.

In summary, the update rests on these assumptions:

- JavaScript is developed in layers
- A layer is one or more Interactive Reporting document sections that together implement some discrete function or set of related functions

- Document scripts are treated as if they are sections with `onStartUp`, `onShutDown`, `onPreprocess`, `onPostProcess` events

Unicode Functionality

The Dashboard Development Services Update Utility automatically converts old code page based Interactive Reporting documents to Unicode before use, enabling documents with different languages to be updated.

Tip:

If running a non-Unicode Interactive Reporting document through the Dashboard Development Services Update Utility results in an error, open and save the non-Unicode document in Interactive Reporting Studio before an update.

Consolidate Images in Resource Manager

The utility automatically removes duplicate and unused images from the Resource Manager. See "Resource Manager" in the *Hyperion Interactive Reporting Studio User's Guide.*

Update Workflow

The workflow to update or transform sections is described in this topic.

1. Create or update a new sections file (see "New Sections File" on page 72).
2. Configure `JavaScriptUpdateConfig_dds.js`. A `JavaScriptUpdateConfig_dds.js` is provided with the installation (see "Configure Configuration Files" on page 73).
3. Update documents (see "Updating Documents" on page 73).

Note:

If templates are not and will not be customized, proceed directly to step 3 to update documents.

New Sections File

The Dashboard Development Services Update Utility uses a new sections file as its input. It is an Interactive Reporting document that contains the latest version of the infrastructure (the shared JavaScript function and object constructors). The utility opens each Interactive Reporting document in a nominated list and performs a compare operation which checks if section names from the list exist in the new sections file and the Interactive Reporting document to be updated. If a section exists in both, the section in the Interactive Reporting document is removed and replaced with the section from the new sections file.

Configure Configuration Files

In a pre-9.3 Release of the Dashboard Development Services Update Utility, an `Upgrade.ini` file was included with the installation. That file has been replaced by a less restrictive configuration file, that contains a set of JavaScript functions called by the update script at crucial points in the process. The configuration file enables the update process to be refined or customized to suit your requirements. Together with the new sections file, this configuration file updates Dashboard Studio templates or documents.

The Dashboard Development Services Update Utility enables you to move values from the contents of text labels, drop-down lists, and lists from the old dashboard sections that are to be discarded, and copy them into the equivalent shapes of the new sections when they are inserted into the Interactive Reporting document. The configuration file is used to identify the text labels, drop-down lists, and lists in the sections whose values are to be transferred when updating a file. Values to be maintained must be specified in this file.

Refer to the JavaDoc within the configuration file on how to customize the configuration file and the update process.

Two configuration files are provided with the installation: `JavaScriptUpdateConfig_dds.js` and `JavaScriptUpdateConfig_blank.js`. You can customize either file, however, `JavaScriptUpdateConfig_dds.js` works specifically with Dashboard Studio.

Modify the Configuration File

The configuration file can be modified to include custom components, and to add or remove sections.

The Dashboard Development Services Update Utility compares the available sections in the documents to update and the specified new sections file. If a section exists in the new sections file and the Interactive Reporting document to be updated, the corresponding section from the new sections file replaces the section in the Interactive Reporting document. The configuration file enables you to specify sections that are to be added to the documents to update even if these sections did not previously exist. Similarly, you can specify sections to be removed from the documents to update.

Updating Documents

The Dashboard Development Services Update Utility enables you to update documents in one of three ways:

- Using the Update One Method—Update one Interactive Reporting document at a time with a GUI
- Using the Update Many Method—Update a folder of Interactive Reporting documents with a GUI
- Selecting Documents to Update with Command Line Updates

Using the Update One Method

Update one Interactive Reporting document at a time using the utility GUI.

➤ To update one Interactive Reporting document:

1. **Open Dashboard Development Services Update Utility.**
2. **From Update Method, select Update One.**
3. **Select the preferred backup option.**
   - Selecting Place Updates in the Update Folder, creates an Update folder in the source path and the updated document is saved to the folder (If a document with an identical name exists in the Update folder, a timestamp is added to the document currently being updated. The original document in the Update folder is not overwritten)
   - Selecting Place Originals in Backup Folder, creates a Backup folder in the source path (The original document is saved to the Backup folder with a timestamp added to the file name. The updated document is saved to the source path and takes the name of the original document)

   When the option to create a backup of the document is selected, the original document must not be set to read-only.
4. **Click (next to JavaScript Configuration File), to locate the configuration file.**

   The JavaScript configuration file determines the sections to replace, add, delete, or preserve in the specified document to update. Use the default configuration file that is provided, unless you have a customized script to use.
5. **Click (next to New Sections File), and locate the new sections file.**

   A new sections file is provided with the Dashboard Studio installation, which contains the latest version of the infrastructure sections. If custom components have been added, you will need to use a different new sections file.
6. **Click (next to Document to Update), and locate the document to update.**

   The save path of the updated document is generated in Save Path of Updated Document(s).
7. **Click Update.**

   When the update process is complete, a report confirms a successful or unsuccessful update.
8. **Click View to launch the updated document in Interactive Reporting Studio.**

Using the Update Many Method

Update multiple Interactive Reporting documents using the utility GUI.

➤ To update a folder of Interactive Reporting documents:

1. **Open Dashboard Development Services Update Utility.**
2. **From Update Method, select Update Many.**

3 **Select the preferred backup option.**

- Selecting Place Updates in the Update Folder, creates an Update folder in the source path and the updated documents are saved to the folder (If a document with an identical name exists in the Update folder, a timestamp is added to the document you are currently being updated. The original document in the Update folder is not overwritten)
- Selecting Place Originals in Backup Folder, creates a Backup folder in the source path (The original documents are saved to the Backup folder with a timestamp added to the file name. The updated documents are saved to the source path and take the name of the original documents)

When the option to create a backup of the documents is selected, the original documents must not be set to read-only.

4 **Click (next to JavaScript Configuration File), to locate the configuration file.**

The JavaScript configuration file determines the sections to replace, add, delete, or preserve in the specified document to update. Use the default configuration file that is provided, unless you have a customized script to use.

5 **Click (next to New Sections File), and locate the new sections file.**

A new sections file is provided with the Dashboard Studio installation, which contains the latest version of the infrastructure sections. If custom components have been added, you will need to use a different new sections file.

6 **Click (next to Document Folder to Update), and locate the folder to update.**

The save path of the updated documents is generated in Save Path of Updated Document(s).

7 **Click Update.**

When the update process is complete, a report confirms a successful or unsuccessful update.

8 **Click View to open the updated folder.**

Tip:

If Interactive Reporting Studio does not launch properly after an Update Many operation in the Dashboard Development Services Update Utility, open the Windows Task Manager and end any `brioqry.exe` process before trying to launch the document.

Command Line Updates

The Dashboard Development Services Update Utility gives you the option of updating documents with a command line. The major advantages of this method include:

- Multiple documents from different locations can be updated simultaneously
- A permanent list of documents to update can be built

Dashboard Development Services Update Utility performs generic transformations based on a specified script. Generic transformations are only available using the command line. Any customized script can be run from the command line.

Selecting Documents to Update

Use scripts that are provided or customized scripts to perform the transformation.

➤ To select files or folders to update using the command line:

1 **Navigate to the bin folder.**

If the installation was not customized, the bin folder is under `C:\Hyperion\products\biplus\bin`. However, if the installation was customized, fix the path to be relative.

2 **Select `runTransformScript.bat`, right-click, and select Edit.**

This enables you to specify the script, parameters, and options to run.

The BAT file includes these lines:

```
set PARAM=%PARAM% -js ""
set PARAM=%PARAM% -param:name=""
```

The first line is the script to run and the second line is a parameter placeholder.

3 **Specify the script.**

The script to run must be entered after the `-js` option in the BAT file. For example, `-js "JavaScriptUpdate.js"`.

4 **Specify the script parameters using one of these two methods:**

A script can have many or no parameters.

a. Using the `-param` option.

All \ and " must be escaped.

The `-param` option has this syntax for single values:

```
-param:name="value"
```

The `-param` option has this syntax for an array with two items:

```
-param:name="[\"value\", \"C:\\Hyperion\\products\\biplus\"]"
```

The first item is value and the second item is `C:\Hyperion\products\biplus\`.

For example, JavaScript Update requires these parameters:

- The configuration file (parameter name: updateConfig)
- The new sections file (parameter name: newSectionFile)
- The document or file to update (parameter name: targetBQYs)
- The update folder (parameter name: updateFolder) OR the backup folder (parameter name: backupFolder)

An example:

```
set PARAM=%PARAM% -js "C:\\Hyperion\\products\\biplus\\DDS\\scripts\
\DDSUpdate\\JavaScriptUpdate.js"
set PARAM=%PARAM% -param:updateConfig="C:\\Hyperion\\products\\biplus\
\DDS\\scripts\\DDSUpdate\\JavaScriptUpdateConfig_dds.js"
set PARAM=%PARAM% -param:newSectionFile="C:\\Hyperion\\products\
\biplus\\DDS\\scripts\\DDSUpdate\\newsections.bqy"
set PARAM=%PARAM% -param:targetBQYs="C:\\Folder or Document to update"
set PARAM=%PARAM% -param:updateFolder="C:\\Update folder"
```

Optional: Replace the last line, if creating a backup folder:

```
set PARAM=%PARAM% -param:backupFolder="C:\\Backup folder"
```

b. Using the `-batch` option.

This option enables you to execute a script, multiple times with different parameters. Each line in the batch file represents one execution of a script. The parameters are comma-separated; for example, `name1="value1", name2="value2"`.

This example is an equivalent batch file for the JavaScript Update example in step 4a. It is an example of the contents of a BAT file:

```
updateConfig="C:\\Hyperion\products\biplus\\DDS\\scripts\\DDSUpdate\
\JavaScriptUpdateConfig_dds.js", newSectionFile="C:\\Hyperion\products
\biplus\\DDS\\scripts\\DDSUpdate\\newsections.bqy", targetBQYs="C:\
\Folder to update", updateFolder="C:\\Update folder"
```

Use this code to point to the BAT file:

```
set PARAM=%PARAM% -js "C:\\Hyperion\products\biplus\\DDS\\scripts\
\DDSUpdate\\JavaScriptUpdate.js"
set PARAM=%PARAM% -batch "C:\\Hyperion\products\biplus\\DDS\\scripts\
\DDSUpdate\\batchFile.txt
```

`BatchFile.txt` is used as an example name for the parameter file.

5 **Optional: To include an instruction in the BAT file, to fail on error (foe), add the parameter:**

```
set PARAM=%PARAM% -foe
```

If this parameter is included and the file encounters an error when running the transformation, the process will stop and an error message will be displayed in the log. If the line is not included, any errors that are encountered are skipped. The log alerts you to errors. This works only for JavaScript Update.

6 **Optional: To include an instruction in the BAT file, to create a report, enter this line:**

```
set PARAM=%PARAM% -rep "C:\\Hyperion\\products\\biplus\\DDS\\report"
```

A file path and folder must be specified; for example, `C:\Hyperion\products\biplus\DDS\report`.

7 **Save and close the BAT file.**

8 **In Interactive Reporting Studio, open a Command window.**

9 **Navigate to the bin folder.**

For example, navigate to `C:\Hyperion\products\biplus\bin`.

10 **Run `runTransformScript.bat`.**

The BAT file must be run from the bin directory.

The Command window displays output. If successful, a message is displayed.

10 Updating Documents with Advanced Scripting

In This Chapter

Customizing Scripts

This topic discusses customizing scripts to update documents in Oracle Enterprise Performance Management Workspace, Fusion Edition or on the desktop in Interactive Reporting Studio.

EPM Workspace Custom Scripting Environment

The custom scripting environment of the Oracle's Hyperion® Impact Management Services provides a mechanism for manipulating the content and structure of an Interactive Reporting document through a Document Object Model (DOM) and JavaScript. Although this environment is similar to the scripting environment in Interactive Reporting Studio, there are differences. For example, the custom scripting environment of the Impact Management Services:

- Does not work in the context of an active Interactive Reporting document
- Provides access to all properties in the document
- Does not perform logical system-level integrity checks
- Is not contained inside the Interactive Reporting document
- Executes a script over multiple documents

The custom scripting environment performs arbitrary, common transformations on one or more documents. This mechanism is used to implement the Update Data Models and Update JavaScript features of the Impact Management Services.

Scripts can be imported into EPM Workspace and then run using the Custom Update feature of the Impact Management Services to make changes to other imported documents. These scripts can also be run on a desktop by the Dashboard Development Services Update Utility. From the desktop, changes can be made only to files on disks visible from that desktop. The desktop is typically a development and test environment.

Scripts in EPM Workspace run under the control of the Impact Management Services and consequently can use the Undo feature. If a change made through scripts is unwanted, the task that used the script can be undone and the documents are returned to the pre-script state.

See "Using Impact Management Services" in the *Hyperion Workspace Administrator's Guide.*

Calling Scripts

The Impact Management Services scripts can be executed within EPM Workspace or on a client desktop in Dashboard Development Services Update Utility.

Calling Scripts in EPM Workspace

Within EPM Workspace, the Custom Update feature of the Impact Management Services is used. The feature presents three steps to execute scripts:

1. Browse for and select a script.
2. Enter parameters required by the script. The Impact Management Services builds a parameter form that is specific to that script. Or you can specify sets of parameter values by using a batch input file.
3. **Optional:** Schedule when to execute the script.

Calling Scripts in Dashboard Development Services Update Utility

Use the Dashboard Development Services Update Utility to execute scripts using batch files; for example, `runTransformScript.bat`. Each script requires a specific batch file containing the parameters required by the script. See "Selecting Documents to Update" on page 76.

The JavaScript Update transformation replaces earlier sections that contain JavaScript with later versions of sections. Property settings from the earlier section are transferred to the later section, so the later code can work with earlier properties. The Dashboard Development Services Update Utility supports both batch and interactive mode. See "Updating Documents" on page 73.

Monitoring Script Execution

The Show Task Status list in EPM Workspace enables progress monitoring of script execution. While awaiting execution, and during the running of a script, the status is displayed as Waiting (gray). Upon completion, the status changes to Success (green) or Fail (red).

When a task is complete, double-clicking the entry in the Show Task Status list displays generated log messages. Use logs to debug errant scripts.

In Interactive Reporting Studio, logging is written to the file called *`dds.log`*. In a standard installation, the logs folder is located under `C:\Hyperion\products\biplus\logs`.

Custom Scripts

These scripts are available to update documents in EPM Workspace or Interactive Reporting Studio.

JavaScriptUpdate.js

The JavaScriptUpdate script enables users to take advantage of the latest dashboard features without having to re-create documents from scratch. See "Using Impact Management Services" in the *Hyperion Workspace Administrator's Guide*.

UpdateDataModels.js

The UpdateDataModels script enables data models in documents to be updated to reflect changes in underlying databases. See "Using Impact Management Services" in the *Hyperion Workspace Administrator's Guide*.

SortDataModelTopics.js

The SortDataModelTopics script enables documents to be updated so the topics in data models are displayed in EPM Workspace in a user-defined order or alphabetically.

When an Interactive Reporting document is opened in EPM Workspace and a query is visible, a list of topics is displayed in the catalog pane under Tables. The topics are displayed in the order in which they were added to the Interactive Reporting document which makes locating topics difficult if there are many in the list.

The SortDataModelTopics script enables the user to specify the order in which the topics are displayed in these lists, using three parameters.

There are two ways to specify the sort order:

1. Use the first parameter to select a file containing a list of topic names, in the order preferred by the user.

2. Use the second parameter (true or false) to specify whether topics that are not included in the sorted file should be ordered alphabetically.

Topics that are not mentioned in the sorted file are placed after topics that are mentioned, and are ordered according to the second parameter. Therefore, if you provide an empty file and the second parameter is true, all topics will be ordered alphabetically, making it easy to locate a topic in the list.

Note:

The empty file should contain a blank line.

The third parameter enables selection from a set of files to be updated, through a multi-file picker.

A version is added for each successfully updated file. Therefore, double-clicking a file in EPM Workspace displays the updated content.

RevertImageResources.js

The RevertImageResources script is desktop-specific, and is not used in EPM Workspace.

In releases earlier than Interactive Reporting Studio Release 9.3, if an Interactive Reporting document contained identical images in multiple places, the image content was duplicated and the Interactive Reporting documents were larger than necessary, slower to load, and less efficient to maintain. Interactive Reporting Studio Release 9.3 and later implements a Resource Manager that centralizes image content storage. For Interactive Reporting documents created in earlier releases, duplicate images can be optionally merged for efficiency.

A side effect of this merging is that if a pre-9.3 release document is opened, optimized, and saved in the 9.3 or later release, the document loses its images when opened in an earlier release of Interactive Reporting Studio, since the relocation and rationalization of the images is not understood by earlier releases.

To retain compatibility with earlier releases in practical terms, the RevertImageResources script provides a facility to undo the relocation and image merging and return the Interactive Reporting document to the pre-9.3 format.

Caution!

The script is useful only with documents saved using Interactive Reporting Studio Release 9.3 and later, after the new image consolidation and translation features have been used. See "Consolidate Images in Resource Manager" on page 72.

Running the RevertImageResources Script

The RevertImageResources script can be run using `runRevertImageResourcesScript.bat`. If the installation was not customized, `runRevertImageResourcesScript.bat` is located in `C:\Hyperion\products\biplus\bin`.

This script accepts a path as a command line parameter:

- This can be the absolute path to an Interactive Reporting document to revert it to its pre-9.3 release format
- This can be the absolute path to a folder of Interactive Reporting documents to revert them all to their pre-9.3 release format

Note:

Any backslashes (\) in the path should be duplicated. For example, `C:\\docs\\mydocument.bqy`.

Using the RevertImageResources Script on a Folder

The example assumes that Oracle's Hyperion Reporting and Analysis is installed in `C:\Hyperion\products\biplus`, and that the folder of Interactive Reporting documents to be reverted is located in and called *`Q:\files_to_revert`*.

➤ To use `runRevertImageResourcesScript.bat` on a folder:

1. **In Windows, open a Command window by selecting Start, then Run, and enter** `cmd`**.**
2. **Click OK.**
3. **Enter** `cd C:\Hyperion\products\biplus\bin`**, and press Enter.**

   The path changes to the bin directory.
4. **Enter** `runRevertImageResourcesScript Q:\\files_to_revert`**, and press Enter.**

   The Interactive Reporting documents in the folder are examined, restructured, and new files are created with (images reverted) in their names. At the end of the process there are double the amount of files in the folder.

Using the RevertImageResources Script on a File

The example assumes that Reporting and Analysis is installed in `C:\Hyperion\products\biplus`, and that the Interactive Reporting document to be reverted is located in and called *`Q:\files_to_revert\my_revenue.bqy`*.

➤ To use `runRevertImageResourcesScript.bat` on a file:

1. **In Windows, open a Command window by selecting Start, then Run, and enter** `cmd`**.**
2. **Click OK.**
3. **Enter** `cd C:\Hyperion\products\biplus\bin`**, and press Enter.**

   The path changes to the bin directory.

4 Enter `runRevertImageResourcesScript Q:\\files_to_revert\\my_revenue.bqy`, and press **Enter**.

The Interactive Reporting document is examined, restructured, and a new file is created called *my_revenue(images reverted).bqy*. The document can be used with earlier releases of Interactive Reporting Studio.

Note:

New features cannot be converted to earlier features. For example, if a Release 9.3 or later Interactive Reporting document contains scatter and bubble charts, these are lost when the document is opened in an earlier release of Interactive Reporting Studio.

Script Parameters

The parameters required by a script are specified using comments in the header. These are similar in format to the JavaDoc comments used to document Java.

The minimum that can be specified to define a parameter is the name; for example, @param sourceLanguage.

This assumes that the input is a simple string and displays an (initially empty) text box on the UI.

Optional: An @inputType line enables more specific data input methods:

- text—Text
- password—Text displayed as asterisks (*)
- file_picker_single_value—Select one file from the repository
- file_picker_multi_values—Select multiple files from the repository, all of which constitute one value
- file_picker_multi_values_parallel_execution—Select multiple files from the repository, all of which can be processed in parallel by separate instances of the script
- dropdown—Select from a predefined set of fixed values

Input types can be given a default value using @defaultValue. The @defaultValue of file_picker type is the fully qualified path and name; for example,

`/Administration/Impact Manager/Script Repository/SortDataModelTopics.js.`

Note:

If this is not unique or the file does not exist, then a warning dialog is displayed and the parameter default value is not set. It has the same effect as not specifying the default value.

Drop-down lists require a separate @comboValues line that specifies possible choices, separated by commas.

Note:

For custom scripts, parameter values are validated only when the script is executed, not at submission time. For example, if an unacceptable value is specified for a script, the user is not informed at the time of task submission. If a script cannot recover from invalid data, it logs a message and throws an exception, causing the status to display as Fail (red) in Show Task Status, alerting the user to the problem.

Logging

Scripts use log messages to communicate with users. In EPM Workspace, logs are accessed through the Show Task Status list. On the desktop, in Interactive Reporting Studio, users view `dds.log`, which is the default name for script log files, and can be changed in the BAT file. In a standard installation, logs are located in the default folder `C:\Hyperion\products\biplus\logs`. Each execution of the script clears the log file. On the desktop, the log represents the entire set of tasks one for the folder and one for each file.

Note:

In the Dashboard Studio Inspector Utility Custom Update Script feature logging messages are also displayed in Task Options, in the Results window. See "Dashboard Studio Inspector Utility" in the *Hyperion Interactive Reporting – Object Model and Dashboard Development Services Developer's Guide, Volume 5: Dashboard Studio.*

The higher the level set, the more messages are displayed in the logs. The levels are explained in Table 23.

Table 23 Logging Levels

Level	Description
Debug	Determines what is happening during script development or to track down problems
Warn	Warns of recoverable problems that require correcting
Error	Indicates the inability to correctly perform the requested processing
Fatal	Indicates the script cannot continue
Always	Messages that are always displayed

There are env methods available to log messages at each of these levels. For example, env.logInfo(), env.logDebug(), and so on. See "ScriptEnvironment Object" on page 93.

There is also a default logging level associated with the script execution. The env.log() method logs messages at that level. The default level is initially debug, but can be changed by using env.setLogLevel().

The env.logClassName() method provides information on the type of an object, because values returned by methods are a combination of JavaScript and Java objects.

Writing Document Information into the Log

A simple script that works with an Interactive Reporting document or a folder of Interactive Reporting documents can be used to determine each file name and the number of sections it contains.

Example: Using a Windows command file for desktop execution

`Rhino.bat` is a general purpose command file that is used to launch most Impact Management Services scripts on the desktop. Parameters must be entered in the form: `<documentOrPath> param_1=value_1 param_2=value_2`.

Example: Using a script file for one file or a folder of files

Where the `document` parameter of the script is a folder, then the script instructs the environment to expand the folder into a list of files and calls the script once for each file in the list. Script execution is called a *task* and is monitored in EPM Workspace using the Show Task Status and Task Management lists. If the `document` parameter of the script is one file, then the script is called once.

Document Object Model Tree Structure

The Document Object Model (DOM) is a tree structure of nodes and properties; each node is made up of more nodes and properties. The DOM and JavaScript engine provide the ability to interrogate and update the state. In Impact Management Services, it is not necessary to expand the whole Interactive Reporting document, only those nodes with properties of interest. For example, when doing a data model update, only query and data model sections need to be expanded. However, this procedure requires no expansion, as the information is available at the top level of the DOM.

Expanding part of an Interactive Reporting document speeds up document loading and consumes less memory. The document loading routines expand only what is required as it is requested. Any scripts that make use of this optimization continue to work; the Document Conversion Strategy parameter is ignored.

Note:

You can include `bqReadWriteDom` and `bqReadOnlyDom` scripts; however, their values are ignored.

Each document manipulated by a script is stored in the form of a DOM, represented by a tree of nodes, each of which contains a set of associated properties.

The DOM for a document is acquired by retrieving the file and loading the content; for example,

```
var uuid = env.getParameterValue("document");
var file = repository.retrieveFile(uuid);
var dom = env.getBqyDocument(file, bqReadWriteDom,
bqDashboardReportStrategy)
```

The first line retrieves the parameter that contains the document UUID. The second line is used to copy the file from the repository to a temporary file which is deleted when the script ends. The third line loads the content of the file, providing a BqyDocument object that represents the DOM.

Note:

The second parameter, bqReadWriteDom, specifies that the document is to be rewritten. If it is not to be rewritten, specify bqReadOnlyDom to reduce the amount of memory required for the DOM. The third parameter is the document conversion strategy, bqDashboardReportStrategy. It determines how much of the underlying document structure is accessible to the script.

Using different strategies, the amount of memory required by a script can be reduced, as can the time spent loading the document.

Document Conversion and Loading Strategies

When loading documents, you can save memory by loading only those portions of the DOM that are required by a given script; for example, JavaScript Update uses only dashboard sections. If you want to log a list of section names in a document, you do not need to load the entire tree of nodes that lie beneath the sections. An example of the required syntax:

```
env.getBqyDocument(documentFile, bqReadWriteDom,
bqJavascriptUpdateStrategy);
env.getBqyDocument(documentFile, bqReadOnlyDom, bqDashboardReportStrategy)
env.getBqyDocument(documentFile, bqReadOnlyDom, bqDashboardReportStrategy)
```

These strategies are provided for loading sections:

- bqDashboardReportStrategy—Only dashboards and reports
- bqDatamodelUpgradeStrategy—All data models and queries
- bqJavascriptUpdateStrategy—Only dashboards
- bqTopLevelSectionsStrategy—All sections and section level properties (a minimal DOM is created)
- null—The whole document

Note:

A just-in-time approach to DOM building makes document loading strategies redundant. Any strategy parameter provided is ignored, and the bqReadWriteDom script is ignored.

Traversing the Document Object Model

To manipulate the content of a node in the DOM, you must locate the node.

The top-level nodes that represent the sections of a document can be accessed directly by using the Sections collection. The shapes within a dashboard are accessible through its Shapes collection. However, there is no collection for the children of a node.

Methods are provided to access the children of a node:

- getChildren()—Returns a complete list of children of a node
- getChildrenOfType()—Returns a list of children of a node that have a specific type
- addChild()—Adds a new child to the end of a list of children of a node
- removeChild()—Removes the specified node from a list of children of a node
- setChildren()—Replaces a list of children of a node with another list
- dump()—Dumps the DOM tree, starting at the given node, for debugging

To iterate over all subnodes of a given node, use getChildren() to retrieve a list that contains them. Use getChildrenOfType() to limit this to the children of a particular type. For example, the Root.MyDocument node contains a Rpt.DocComp node for each section in the document, which can be located using this line:

```
var sections = root.getChildrenOfType("Rpt.DocComp");
```

A node is added as a child of another by using addChild(). Use this to copy a node from one part of the DOM (or the DOM of another document) to another location.

To remove a child node, use removeChild().

Note:

The list of children returned by getChildren() and getChildrenOfType() is read-only. If you update the list by assigning a new value to an entry, this does not affect the node. However, the current list of nodes can be replaced using setChildren().

The content of a sub-tree of the document can be written to the log using dump(). By default, this dumps the tree to standard output, but by supplying parameters, it can be written to any print stream.

XPath-Style Searching

While obtaining lists of child nodes enables access to the entire DOM, you can search for nodes that satisfy a set of criteria.

For example, this code can be used to log the names of all shapes within a document:

```
for (var i = 0; i < dom.Sections.length; i++) {
    var section = dom.Sections[i];

    if (section.Type == bqDashboard) {
        env.log("Dashboard " + section.AnnotName + " has shapes");

        var shapes = section.Shapes;

        for (var j = 0; j < shapes.length; j++)
                env.log(shapes[j].Name);
    }
}
```

The DOM provides user-friendly collection names for both the Sections inside a document and the Shapes inside a dashboard. However, a complex search example that looks for all DataThreshold.DataThreshold nodes inside all ThreshFmt.ThreshFmt nodes, inside all ColColl.Item nodes, inside all table sections, results in multiple nested loops.

The Impact Management Services scripting provides an alternative approach, through XPath-style searches. For example, this is the code to use for the complex search example:

```
var items = dom.findNodesByPattern("/BQY/Root.MyDocument/Rpt.DocComp"
        + "/ColColl.Item/ThreshFmt.ThreshFmt"
        + "/DataThreshold.Threshold");
```

This single statement provides an array that contains the required nodes. Property matching requirements can be included to narrow down which nodes are to be returned.

For example, to limit the result to those nodes in the column named *Drawn* inside the table named *Rankings*.

```
var items = dom.findNodesByPattern("/BQY/Root.MyDocument"
        + "/Rpt.DocComp[AnnotName='Rankings']"
        + "/ColColl.Item[Label='Drawn']/ThreshFmt.ThreshFmt"
        + "/DataThreshold.Threshold");
```

Searches need not begin at the root of the DOM. If a variable that contains a section node is searched, use a relative path to find other nodes beneath that section; for example:

```
var table = dom.findNodesByPattern("/BQY/Root.MyDocument"
        + "/Rpt.DocComp[AnnotName='Rankings']");

var items = table.findNodesByPattern("ColColl.Item[Label='Drawn']"
        + "/ThreshFmt.ThreshFmt/DataThreshold.Threshold");
```

Use getNodesByPattern() if there is a possibility that a node may not exist (that is, if documents to be processed using a script may not contain this node) or where there can be many of these nodes. In these cases, the length of the returned array is used to determine the situation.

However, if one node matching the pattern is guaranteed, use getNodeByPattern() which returns one object, rather than an array.

The search mechanism provides two wildcard facilities. An asterisk (*) in place of a node name, represents any type of node. A pair of slashes (//) represents any number of intervening nodes.

For example, to find all images in a document, in dashboards or reports (in the body, header, footer, section header, or section footer), use this example code:

```
var pattern = "//Box.Item[RuntimeClassName='PictField']";
var pictures = dom.findNodesByPattern(pattern);
```

Differences Between the Impact Management Services and Interactive Reporting Studio Document Object Models

Impact Management Services provides the DOM to its scripts and Interactive Reporting Studio, including Interactive Reporting Web Client, provides the BQY object model (BOM) to scripts embedded within Interactive Reporting documents.

The DOMs available in the Impact Management Services scripting differ from those provided to event handlers in Interactive Reporting Studio scripting:

- All collection indices start at zero, rather than one
- The node names and properties match those stored in the underlying document, as displayed in the Dashboard Studio Inspector Utility
- The BOM provides user-friendly names to resemble the view through the Interactive Reporting Studio; whereas, the DOM provides fewer user-friendly names
- The BOM does not provide access to the majority of properties, however the DOM provides access to all properties
- Using the DOM, the BOM event handlers for sections and shapes cannot be called to effect changes to the document
- The BOM provides safety checks and restrictions, however the DOM provides only basic type checking
- Using the DOM, you can change and transform anything; for example, you can create files that are not recognized by other software

For example, display the SQL associated with all request line items in all queries by using this code:

```
var pattern = "//BQY/Qry.MyQry/QryCol.MyQryCol";
var nodes = dom.findNodesByPattern(pattern);

for (var i = 0; i < nodes.length; i++)
        env.log(nodes[i].SQL);
```

Note:

Both loop indices start at zero and access to the name of the section is through AnnotName, rather than Name.

Investigating the Impact Management Services DOM Structure

The Dashboard Studio Inspector Utility, included with the Interactive Reporting Studio installation, provides an explorer-style view of the DOM. See "Dashboard Studio Inspector

Utility" in the in the *Hyperion Interactive Reporting – Object Model and Dashboard Development Services Developer's Guide, Volume 5: Dashboard Studio.*

The left pane displays the nodes contained within the document as a tree. The right pane displays the names of all properties of the selected node, and their current values and associated data types.

Use the Inspector Utility when writing scripts, to determine where in the DOM the data resides that must be manipulated to achieve the intended change, and to generate the path that provides programmatic access to data of interest.

Accessing Properties

Access properties in Impact Management Services, as you do in Interactive Reporting Studio. The only difference is the DOM exported by Interactive Reporting Studio provides more user-friendly names for frequently used properties.

To learn the names of properties associated with a node in the DOM use the Dashboard Studio Inspector Utility. See "Dashboard Studio Inspector Utility" in the in the *Hyperion Interactive Reporting – Object Model and Dashboard Development Services Developer's Guide, Volume 5: Dashboard Studio.*

For example, this code accesses the IsHidden property of a dashboard, making the section visible if it is hidden.

```
var dashboard = dom.Sections["Dashboard"];

if (dashboard.IsHidden) {
        env.log("Making section " + dashboard.Name + " visible");

        dashboard.IsHidden = 0;
}
```

Collections

An important difference between the Interactive Reporting Studio scripting DOM and the Oracle's Hyperion® Impact Management Services DOM is that all collections are zero-based, not one-based. For example, a loop that would have been coded as:

```
for (var i = 1; i <= collection.Count; i++)
    // some processing on collection[i]
```

is now written as:

```
for (var i = 0; i < collection.length; i++)
    // some processing on collection[i]
```

Property Types

Every property of a DOM node has one of these data types:

- Byte
- DWord
- Long
- String
- Structure
- Word

Accessing the File System

To access the underlying file system from within a script; for example, where a large amount of configuration information is needed that does not change from execution to execution, use these methods.

- env.getFileSystem()—Retrieve an object that provides access to the underlying file system
- env.createTempFile()—Create a temporary file that is cleaned up when the script completes
- fs.getFile()—Retrieve a Java File object that refers to the file with a given path within EPM Workspace
- fs.writeBytesToStream()—Write the contents of a byte array to a file

General Java Code in Scripts

It can be necessary to construct Java objects as part of processing a script. For example, RevertImageResources creates a FileOutputStream using the call:

```
var fos = new Packages.java.io.FileOutputStream(imageFile);
```

The call is of the form:

```
var object = new Packages.java.some.package.ClassName(necessary,
parameters);
```

Using Batch Input Files

All parameters for a transformation script can be entered interactively by using the user interface, or you can request the processing of many sets of parameters by providing them as a batch input file.

Each line of a batch input file contains a complete set of parameters, as a comma-separated list of name="value" specifications.

For example, to use the SortDataModelTopics script to transform the three documents "/some.bqy", "/some/other.bqy" and "/yet/another/example.bqy", using the topic orderings in "/order.txt", and sorting unspecified topic names alphabetically, use this input file:

```
orderings="/order.txt",sortUnknownTopics="true",document="/some.bqy"
orderings="/order.txt",sortUnknownTopics="true",document="/some/other.bqy"
orderings="/order.txt",sortUnknownTopics="true",document="/yet/another/
example.bqy"
```

Note:

Each parameter value is quoted and all of them must be included on each line, even where the value does not change.

In EPM Workspace, the values of any parameters that represent files need to be UUIDs. The sample scripts are explicitly coded to enable batch files to specify file paths, by using code similar to this to convert them into UUIDs where necessary:

```
var document = env.getParameterValue("document");

if (document.substring(0, 1) == "/")
    document = repository.getFileUuid(document);
```

To enable annotation of batch input files, blank lines and any lines beginning with # are ignored.

Note:

The code also works on the desktop, because there the UUID of a file is identical to the file system path.

Scripting References

This topic includes scripting references and examples of objects, methods, and properties that are available to use on the desktop and in EPM Workspace.

ScriptEnvironment Object

Each script has a global variable called *env,* which provides access to the ScriptEnvironment object within which it runs and hosts the features that lead to granting access to documents and the document repository. A repository can be the Reporting and Analysis repository, if the script is running in EPM Workspace, or the file system, if the script is running on the desktop.

expandRequestAction()

Actions are added to the list for the task. Generally expandRequestAction() is used to generate multiple additional tasks to handle a collection of input files. For example, a user requests that an action occurs for a folder. The folder is expanded into an array of files and the script runs for each file in the array.

Example using expandRequestAction():

```
env.expandRequestAction(strParam, arrUuidValues)
```

Parameter	Description
strParam	Represents the file descriptor that is to be expanded
arrUuidValues	An array of the unique identifiers of the set of files on which to act. In EPM Workspace this is a set of real UUID values, on the desktop it is the list of paths of the files as expanded

getBqyDocument()

Used to retrieve a document from the repository, create the DOM, and provide access to the nodes and properties of the Interactive Reporting document.

Example using getBqyDocument():

```
var domBqy = env.getBqyDocument(filBqy, bqOpenMode, bqStrategy)
```

Parameter	Description
filBqy	Represents the Interactive Reporting document, generally retrieved using the retrieveFile method of the repository artifact
bqOpenMode	Defines the way the file is opened. For example, using bqReadOnlyDom the file is read-only, or using bqReadWriteDom the file has read/write properties. For scripts running in Release 9.3.1 or later this parameter is ignored because the Rhino engine loads nodes only if scripts attempt to reference them or the properties below them (see following table).
bqStrategy	Determines, as an efficiency measure, how much of the DOM is built. For scripts running in Release 9.3.1 or later this parameter is ignored because the Rhino engine loads nodes only if scripts attempt to reference them or the properties below them (see following table).

Property	Description
bqDashboardReportStrategy	Loads only dashboards and reports
bqDatamodelUpgradeStrategy	Loads only data models and queries
bqJavaScriptUpdateStrategy	Loads only dashboards
bqTopLevelSectionsStrategy	Loads all sections and the section level properties
null	Loads the whole document

getFileLines()

Used to retrieve file content from the repository as an array of strings, given the UUID.

Example using getFileLines():

```
var arrLines = env.getFileLines(filToRead)
```

Parameter	Description
filToRead	The text file from the repository to expand into the constituent lines. A file object consists of information about the file, but is not the content of the file.

getMimeTypeUuid()

Used to retrieve the UUID of the MIME type with the specified name.

Example using getMimeTypeUuid():

```
var uuiMimeType = env.getMimeTypeUuid(strMimeType)
```

Parameter	Description
strMimeType	The string representation of the MIME type to be returned; for example, application/x-brioquery

getParameterValue()

Values are obtained for a single-valued parameter, based on the name. If the named parameter value does not exist, return null.

The parameter value can be entered on the command line or through the Custom Update parameter gathering screen in EPM Workspace

Example using getParameterValue():

```
var strVal = env.getParameterValue(strName)
```

Parameter	Description
strName	The name of the parameter as supplied in the command file. For example, `script.js –param:document=c:\docs\myBqy –param:type=Query` In this case, strName is either document or type.

getParameterValues()

All values are obtained for a potentially multi-valued parameter as an array, based on the name. If the named parameter value does not exist, return null.

Example using getParameterValues():

```
var arrValues = env.getParameterValues(strName)
```

Parameter	Description
strName	The name of the parameter as supplied in the command file. For example, `script.js –param:document="[\"c:\\docs\\file1.bqy\", \"d:\\docs\\file2.bqy \"]"` In this case, strName is document.

getRepository()

Used to retrieve the repository artifact in whose context the script is running. If the script is running on the desktop, this is the file system.

Example using getRepository():

```
var repLocal = env.getRepository();
```

getScriptUuid()

The repository UUID of this script is retrieved.

Example using getScriptUuid():

```
var uuiScript = env.getScriptUuid();
```

Use this in EPM Workspace and on the desktop.

isDesktopMode()

Returns true if the script is running on the desktop.

Example using isDesktopMode():

```
var blnDesktop = env.isDesktopMode();
```

isServerMode()

Returns true if the script is running in EPM Workspace.

Example using isServerMode():

```
var blnWorkspace = env.isServerMode();
```

loadScript()

The JavaScript file is loaded and merged with the main script.

Example using loadScript():

```
env.loadScript(strPath, strDesc, uuiScript);
```

Example using loadScript():

```
env.loadScript(filScript, strDesc, uuiScript);
```

Parameter	Description
strPath	The string path that is used to locate the script file
filScript	The file object that references the JavaScript file
strDesc	**Optional:** The description that enables logging and debugging to identify whether an error occurred in the main script or a loaded script

Parameter	Description
uuiScript	**Optional:** The UUID of the script being loaded

Example 1: Use the string to search for the file in the same location as the script. If it fails to locate the file it searches the root of the script repository folder: in EPM Workspace the root is /Administration/Impact Manager/Script Repository, and on the desktop it is `C:\Hyperion\products\biplus\DDS\scripts`.

```
env.loadScript("lib_hysl_core.js");
```

Example 2: Use the file object (`env.loadScript(filScript, strDesc, uuiScript);`) to implement a similar mechanism to Example 1.

```
function _loadScript(in_strScript){
    var uuid, fil
    var folServer = "/Administration/Impact Manager/Script Repository/lib/"
    var folDesktop = "C:\\Hyperion\\products\\BIP\\\biplus\\DDS\\scripts\\"
    if (env.isServerMode()){
       uuid = cn_repLocal.getFileUuid(folServer + in_strScript)
    } else {
        uuid = folDesktop + in_strScript
    }
    fil = cn_repLocal.retrieveFile(uuid)
    env.loadScript(fil, in_strScript, uuid)
}
_loadScript("lib_hysl_core.js");
```

setParameterValues()

Used to set the value of named parameter. For example, if the same values are set up over and over, a separate script can be written that lists only a subset of the parameters, which are the only ones displayed in the parameter screen, setParameterValues() is used to set the others, and loadScript() is used to read in the original.

Example using setParameterValues()

```
setParameterValue(strName, strValue)
```

writeBqyDom()

A document is written to disk that is ready to import into the repository as a new version of an existing document or as a new document.

Example using writeBqyDom():

```
var filBqy = env.writeBqyDom(domBqy);
```

Parameter	Description
domBqy	The DOM that contains all the document nodes

Reporting and Analysis Repository: Repository Artifact

The repository artifact provides access to the features of the container that holds the documents. If the script is run in EPM Workspace it uses the Oracle's Hyperion Reporting and Analysis repository. Publications that are stored here include; documents, connection information, and the folder hierarchy. If the script is run on the desktop, the file system represents the repository artifact with reduced features.

A repository artifact is created by calling `env.getRepository()`, and the object has these methods.

addVersion Method

A new version of the file is added to the repository. This method applies only in EPM Workspace. If the number and names of sections are not changed by your script, then the Interactive Reporting database connection (oce) information is not required.

Example 1 using addVersion:

```
var intV = objRep.addVersion(strUuid, objFile, strDesc);
```

Parameter	Description
strUuid	The document UUID
objFile	The file that contains the new content, created by calling env.writeBqyDom() method
strDesc	The description to add to the repository

Example 2 using addVersion:

```
var intV = objRep.addVersion(strUuid, objFile, strDesc, objOce);
```

Use this method format if the Interactive Reporting database connection files associated with the sections require change, or if you have modified the query or data model. This is used by Data Model Update; see “Using Impact Management Services” in the *Hyperion Workspace Administrator's Guide.*

Parameter	Description
strUuid	The document UUID
objFile	The file that contains the new content, created by calling the env.writeBqyDom() method
strDesc	The description to add to the repository
objOce	An object that represents section information for the Interactive Reporting document, including the Interactive Reporting database connection information that is associated with each query and data model.

Example 3 using addVersion:

```
var intV = objRep.addVersion(strUuid, objFile, strDesc, objOceOld,
objOceNew);
```

This form of the method is tailored to a specific situation, where the updated document has the same query and data model sections as the original (one for one mapping, the same number of each as the original, with identical names to the original) and you want to retain the settings of the previous version.

The oce details are copied from objOceOld (the previous version in repository) to objOceNew, as is.

JavaScript Update uses this form of the method, because it retains the oce settings of the previous version. See “Using Impact Management Services” in the *Hyperion Workspace Administrator's Guide.*

Parameter	Description
strUuid	The document UUID
objFile	The file that contains the new content, created by calling the env.writeBqyDom() method
strDesc	The description to add to the repository
objOceOld	An object that represents section information for the earlier version of the Interactive Reporting document, including Interactive Reporting database connection information that is associated with each query or data model.

convertBqyFileToUnicode Method

The Interactive Reporting document file is converted from a code page based format to Unicode format. To convert the format the desktop calls to Interactive Reporting Studio through its COM interface. This operation requires both `brioqry.exe` and `brioqry.tlb`.

Use this method if you are trying to convert an Interactive Reporting document to the latest format. All images are updated to the Resource Manager format as well as converting to Unicode.

Example using convertBqyFileToUnicode:

```
var objFile = objRep.convertBqyFileToUnicode(objFileOld, intCodePage)
```

Parameter	Description
objFileOld	The original file in the earlier format
inCodePage	The code page of the original document which is accessible from the DOM of the original file; that is, the attribute StdCodePage2

findFiles Method

A list of the files is retrieved that are contained within the folder represented by a UUID. The UUIDs of the files are returned in an array. The call can return the files in the folder, or all the files in a hierarchy of folders under that folder.

Example 1 using findFiles:

```
var clcFiles = objRep.findFiles(uuiFolder, uuiMimeType, blnRecursive)
```

Parameter	Description
uuiFolder	The UUID of the folder. On the desktop, this is the path name.
uuiMimeType	The file type to search for; for example, an Interactive Reporting document
blnRecusrive	False: examine just the folder or True: expand all sub-folders

Example 2 using findFiles:

```
var repLocal = env.getRepository()
var uuiFolder = env.getParameterValue("document")
if (repLocal.isFolder(uuiFolder){
    var uuiMime = env.getMimeTypeUuid("application/x-brioquery")
var clsUuid = objRep.findFiles(uuiFolder, uuiMime, true)
var a = 1
for (var it = clcUuid.iterator(); it.hasNext(); a++) {
    env.log("clcUuid[" + a + "] = " + it.next());
}
    env.expandRequestAction("document", clsUuid)
    return
}
```

folderExists Method

Returns true if the specified folder path exists within the repository.

This method is only available in EPM Workspace and does not apply to the desktop.

Example using folderExists:

```
var blnExists = objRep.folderExists(strPath)
```

Parameter	Description
strPath	The path that represents the folder. In EPM Workspace, a folder is represented by a UUID, and on the desktop, the UUID is the same as a path.

getCurrentFolder Method

Returns the string that represents the current working folder.

Example using getCurrentFolder:

```
var uuiPath = objRep.getCurrentFolder()
```

Parameter	Description
uuiPath	The UUID of the folder path that is the current working folder

getFileUuid Method

The UUID that corresponds to the given absolute path of a file is returned. On the desktop, complete paths and the UUID are identical, but when writing scripts that are intended for the desktop or EPM Workspace, treat UUID values and paths as if they are different and make the extra calls that convert paths to UUID values.

This method is only available in EPM Workspace and does not apply to the desktop.

Example using getFileUuid:

```
var uuiFile = objRep.getFileUuid(strPath)
```

Parameter	Description
strPath	The complete path that represents the file

getFolderContentsFor Method

A list of the names of the files are retrieved that are contained within the folder represented by the given path. The call returns the names in just the folder, or in the entire hierarchy of folders under the folder.

This method is only available in EPM Workspace and does not apply to the desktop.

Example using getFolderContentsFor:

```
var arrNames = objRep.getFolderContentsFor(strPath, blnRecursive)
```

Parameter	Description
strPath	The complete path that represents the folder
blnRecursive	False: examine just the folder or True: expand all sub-folders

getFolderUuid Method

The UUID that corresponds to the given absolute path of a folder is returned. On the desktop, complete paths and the UUID are identical, but when writing scripts that are intended for the desktop or EPM Workspace, treat UUID values and paths as if they are different and make the extra calls that convert paths to UUID values.

Example using getFolderUuid:

```
var uuiFolder = repLocal.getFolderUuid(strPath)
```

Parameter	Description
strPath	The complete path that represents the file

getNameForUuid Method

The name that represents the leaf node of the path for the file referenced by the UUID is retrieved.

This method is only available in EPM Workspace and does not apply to the desktop.

Example using getNameForUuid:

```
var strName = objRep.getNameForUuid(uuiPath)
```

Parameter	Description
uuiPath	The UUID of the files whose name is required

getPathForUuid Method

The path of the file referenced by the UUID is retrieved.

This method is only available in EPM Workspace and does not apply to the desktop.

Example using getPathForUuid:

```
var strPath = objRep.getPathForUuid (uuiPath)
```

Parameter	Description
uuiPath	The UUID of the file whose path is required

getSubfolderPathsFor Method

A list of sub-folders for a folder is retrieved.

This method is only available in EPM Workspace and does not apply to the desktop.

Example using getSubfolderPathsFor:

```
var arrPaths = objRep.getSubfolderPathsFor(uuiPath, blnRecusrive)
```

Parameter	Description
uuiPath	The UUID of the files whose name is required
blnRecursive	False: examine just the folder or True: expand all sub-folders

isFile Method

Returns true if the UUID is a file.

Example using isFile:

```
var bln = objRep.isFile(uuiPath)
```

Parameter	Description
uuiPath	The UUID of the object type that is being tested

isFolder Method

Returns true if the UUID is a folder.

Example using isFolder:

```
var bln = objRep.isFolder(uuiPath)
```

Parameter	Description
uuiPath	The UUID of the object type that is being tested

makeFolder Method

One or more folders are created.

Optional: Creates a hierarchy of folders if they do not exist.

This method is only available in EPM Workspace and does not apply to the desktop.

Example using makeFolder:

```
objRep.makeFolder(strPath, strDesc, blnRecursive)
```

Parameter	Description
strPath	The path that describes the folder to be created
strDesc	**Optional:** The description to be added to the repository for the path
blnRecursive	**Optional:** If this is set to true and the full parent sub-folders do not exist above the lowest node in the path, then folder creation starts at the first missing node and continues until all sub-folders in the path are created

publishBqyFile Method

An Interactive Reporting document is published or imported into the repository, configures the Interactive Reporting database connection mappings, and identifies how the EPM Workspace server treats sections. This function performs the work also done by the publishing wizard when an Interactive Reporting document is imported into EPM Workspace.

Example using publishBqyFile:

```
var uuid = objRep.publishBqyFile(objF, strN, strD, uuiF, blnD, strH, oceP)
```

Parameter	Description
objF	The Interactive Reporting document that is being published
strN	The file name being published
strD	The description associated with the file being published
uuiF	The folder UUID under which this is to be published
blnD	True indicates that the Interactive Reporting document contains dashboard sections
stwH	The section name that is displayed when the Interactive Reporting document is activated on the thin client and when 🏠 (Home) is clicked

Parameter	Description
oceP	An object that represents section information for the Interactive Reporting document, including the Interactive Reporting database connection information is associated with each query and data model

The example illustrates publishing a copy of an Interactive Reporting document. For example, if the selected file is called *sales analysis,* it is published with a name provided by the user, or if no name is provided as *Copy of sales analysis,* into the same folder as the source document. The Interactive Reporting database connection mappings from the source file are also copied to the new file so it can be processed in the same way as the source file. The script works on the desktop and in EPM Workspace.

Example: Publishing a copy of an Interactive Reporting document.

```
/**
 *
 * @param document Select the source to copy.
 * @inputType file_picker_single_value
 *
 * @param target Provide a name to call the copied file
 *
 */

var uuiSrc = env.getParameterValue("document");
var repLocal = env.getRepository();
var filSrc = repLocal.retrieveFile(uuiSrc);
var vrsSrc = repLocal.retrieveVersionedDocument(uuiSrc);
var strSrc = vrsSrc.getName();
var strTrg = env.getParameterValue("target");
if (strTrg == null){
   strTrg = "Copy of " + strSrc;
}
var uuiFolder = vrsSrc.getParentIdentity();
var domSrc = env.getDocument(filSrc, bqReadWriteDom, null);
var oceMapOld = vrsSrc.getSectionOCEMapping();
var oceMapNew = domSrc.sectionOCEPairInfos(uuiFolder);
for (var a = 0; a < oceMapOld.length; a++) {
    if (oceMapOld[a].isOCEEnabled()) {
        oceMapNew[a].setOCEDocument(oceMapOld[a].getOCEDocument());
        oceMapNew[a].setOCEEnabled(true);
    }
}
var strDesc = "this file was copied by a Rhino script from " + strSrc
var blnD =domSrc.isDashboard()
var strH = domSrc.getInitialTCSection()
repLocal.publish(filSrc, strTrg, strDesc, uuiFolder, blnD , strH,
oceMapNew);
```

retrieveFile Method

The latest or a specific version of a file is retrieved from the repository.

Example using retrieveFile:

```
var filBqy = objRep.retrieveFile(uuiBqy, intVersion)
```

Parameter	Description
uuiBqy	The UUID of the specified file. On the desktop the UUID is identical to the full path of the file.
intVersion	**Optional:** If omitted, then the latest version is obtained

retrieveVersionedDocument Method

The latest version of a versioned document object identified by UUID is retrieved. This method provides access to the document description, the keywords, the display name of the published file, the Interactive Reporting database connection file, and how the database connection maps to the document.

Note:

retrieveVersionedDocument is not for use on the desktop.

Example using retrieveVersionedDocument:

```
var vrsBqy = objRep.retrieveVersionedDocument(uuiBqy)
```

Parameter	Description
uuiBqy	The UUID of the specified document

The Node Object

An Interactive Reporting document is composed of hierarchically arranged node sets. Most nodes have commonalities and share methods and properties that apply to most Interactive Reporting document node types.

addChild()

An existing node is added as a child of another node.

Tip:

Useful to copy a node from one location to another.

Example using addChild():

```
nodTrg = nodMyNode.addChild(nodSrc)
```

Parameter	Description
nodSrc	References the source node to be replicated

addProperty()

An existing property is added to another node.

Tip:

Useful to copy a property from one node to another.

Example using addProperty():

```
prpRef = nodMyNode.addProperty(prpSrc)
```

Parameter	Description
prpSrc	References the source property to be replicated

cloneNode()

The entire node and its subordinates are cloned.

Example using cloneNode():

```
var bsndNew = nodMyNode.bsndSrc.cloneNode()
```

findNodeByPattern()

A single node, if any, that matches the specified pattern is retrieved.

Note:

If a node is not found, an exception is thrown.

Example using findNodeByPattern():

```
nodFound = nodMyNode.findNodeByPattern(strPattern)
```

Parameter	Description
strPattern	The search pattern

findNodesByPattern()

Nodes that match the specified pattern are retrieved.

Note:

No exception is thrown if none are found.

Example using findNodesByPattern():

```
arrNodes = nodMyNode.findNodesByPattern(strPattern)
```

Parameter	Description
strPattern	The search pattern

getChildren()

Returns an array of nodes that are directly under this node.

Example using getChildren():

```
arrNodes = nodMyNode.getChildren()
```

getNodeType()

Returns the type of the node; for example, a string.

Example using getNodeType():

```
strType = nodMyNode.getNodeType()
```

getPathWithContext()

Returns the path, as a slash-separated list of node names, with name attribute values that remove ambiguity as to the node identity.

Example using getPathWithContext():

```
strPath = nodMyNode.getPathWithContext()
```

getProperty()

The object that represents a given property is retrieved.

Tip:

Useful for getting values of out arrays.

Example using getProperty():

```
prpResult = nodMyNode.getProperty(strName)
arrValues = prpResult.getValues()
```

Property Object	Description
prpResult	An array of values that require reading or modification.

Parameter	Description
strName	The property name

getProperty() may be accompanied by a getValues() call; for example:

```
arrScripts = nodDocScripts.getProperty("EScript").getValues()
var startUp = arrScripts[0]
var shutDown = arrScripts[1]
var preProc = arrScripts[2]
var postProc = arrScripts[3]
```

Use the EScript property to access multi-valued properties that correspond to simple arrays; for example:

```
var someScript = docAnnotation.EScript[i]
```

In you are using a release earlier than 9.3.1, use this code:

```
var eScripts = docAnnotation.EScript.getValues()

var someScript = eScripts[i]
```

hasProperty()

Returns true if the named property exists, otherwise returns false.

Example using hasProperty():

```
blnResult = nodMyNode.hasProperty(strName)
```

Parameter	Description
strName	The property name

Use hasProperty() rather than performing a Boolean test on the property name, as this returns false if the property is false or zero.

```
// this is not safe because if Offset is 0 it will return false
if (node.Offset){
    // do whatever is needed if the node has an Offset property
}

// this is safe
if (node.hasProperty("Offset"){
    // do whatever is needed if the node has an Offset property
}
```

removeChild()

The nominated child node is removed.

Example using removeChild():

```
nodMyNode.removeChild(nodChild)
```

Parameter	Description
nodChild	The child node to remove

removeProperties()

The properties identified by the array of names are removed.

Example using removeProperties():

```
nodMyNode.removeProperties(arrNames)
```

Parameter	Description
arrNames	The array of property names to delete

Note:

removeProperties() is useful to downgrade the format of an Interactive Reporting document to one that is generated by an earlier format or Interactive Reporting Studio Release. However, you are not required to do this because any properties or nodes not understood by Oracle's Hyperion® Interactive Reporting Studio are ignored when the document is loaded, and therefore lost when the document is saved.

replaceChildNode()

The old child node is replaced with a new node.

Example using replaceChildNode():

```
nodRef = nodMyNode.replaceChildNode(nodChild, nodNew)
```

Parameter	Description
nodChild	A node that exists as a child
nodNew	The new node to replace the child

The BqyDocument Object

The BQY Document Object Model (DOM) provides access to the features of Interactive Reporting documents. Use this object to modify the internal properties of documents.

Example: Retrieving a DOM for a document.

```
var uuiBqySrc = env.getParameterValue("document");
var objRep = env.getRepository();
var bqySrc = objRep.retrieveFile(uuiBqySrc);
var domSrc = env.getBqyDocument(bqySrc, bqReadWriteDom, null);
```

Note:

A DOM is a collection of BQYNode objects arranged in a hierarchy or tree structure.

close()

The document is closed.

Example using close():

```
domSrc.close()
```

Note:

It is important to use this if a single script processes many documents as it saves resources.

compressBqy()

The Interactive Reporting document is compressed into the specified file.

Note:

Returns true if the file is compressed, or false if the file does not require compression.

Example using compressBqy():

```
var bln = domSrc.compressBqy(strNameOld, strNameNew, intInsHdrLen)
```

Parameter	Description
strNameOld	The Interactive Reporting document name to be compressed
strNameNew	The name of the compressed Interactive Reporting document. If it is identical to strNameOld, then the old uncompressed file is removed
intInsHdrLen	The number of bytes of Oracle's Hyperion® Interactive Reporting Web Client (Insight) Header to skip to get to the main header

copy()

A copy is created on the specified section, and the copy is added to the DOM as a section node (Rpt.DocComp).

Example using copy():

```
var rdcTarget domSrc.copy(rdcSource)
```

Parameter	Description
rcdSource	The section to be copied, the Rpt.DocComp object

getInitialTCSection()

A string is returned that identifies the Home section of an Interactive Reporting document for publishing to the thin client.

Example using getInitialTCSection():

```
strName = domSrc.getInitialTCSection()
```

isBQYProcessable()

Determines whether the Interactive Reporting document contains at least one section to be processed.

Example using isBQYProcessable():

```
blnResult = domSrc.isBQYProcessable()
```

isCompressed()

Determines whether the document, from which the DOM derives, is compressed.

Note:

isCompressed() is useful if a requirement for the script is to change the compression status of an Interactive Reporting document.

Example using isCompressed():

```
blnResult = domSrc.isCompressed()
```

isDashboard()

Determines whether the Interactive Reporting document contains at least one dashboard section.

Example using isDashboard():

```
blnResult = domSrc.isDashboard()
```

sectionOCEPairInfos()

An array of Interactive Reporting database connection mappings is provided for the DOM.

Note:

These arrays are not the published Interactive Reporting database connection files associated with the document. However, you can associate the array of mappings with each query published Interactive Reporting database connection, and enable the document to access a data source defined in EPM Workspace.

Example using sectionOCEPairInfos():

```
oceMap = domSrc.sectionOCEPairInfos(uuiParentFolder)
```

Parameter	Description
uuiParentFolder	The folder UUID where the document is published

Example 1: Copying the Interactive Reporting database connection mappings from one DOM to another, and republishing the DOM as a new publication or a new version.

```
function copyBqy(in_repSrc, in_bqySrc, in_bqyTrg){
   var uuiFold = in_repTrg.getFolderUuid(in_bqyTrg.strFolder)
   var oceMapO = in_bqySrc.vrs.getSectionOCEMapping();
   var oceMapN = in_bqySrc.dom.sectionOCEPairInfos(uuiFold);
   for (var a = 0; a < oceMapO.length; a++) {
      if (oceMapO[a].isOCEEnabled()) {
         oceMapN[a].setOCEDocument(oceMapO[a].getOCEDocument());
         oceMapN[a].setOCEEnabled(true);
      }
   }
   var strD = "created by copyBqy"
   var blnD =in_bqyTrg.dom.isDashboard()
   var strH = in_bqyTrg.dom.getInitialTCSection()
   var uuiFound = hysl_getUuid(uuiFolder, in_bqyTrg.strName)
   var filBqy = in_bqyTrg.file
   var strN = in_bqyTrg.strName
   if (uuiFound != null){
      in_repSrc.addVersion(in_uuiToAdd, in_filBqy, in_strDesc)
   }else{
      in_repSrc.publishBqy(filBqy, strN, strD, uuiFold, blnD , strH,
oceMapN);
    }
}
```

Example 2: Publishing a new document and assigning a specific Interactive Reporting database connection to the queries of the new document.

```
var uuiFold = rep.getFolderUuid("/sales/monthly")
var oceMap = bqySrc.dom.sectionOCEPairInfos(uuiFold);
var uuiOCE = rep.getFileUuid("/OCE/salesInfo.oce")
for (var a = 0; a < oceMap.length; a++) {
    if (oceMap[a].isOCEEnabled()){
        oceMap[a].setOCEDocument(uuiOCE);
    }
}
var strD = "my description"
var blnD =bqySrc.dom.isDashboard()
var strH = bqySrc.dom.getInitialTCSection()
var filBqy = bqySrc.file
var strN = bqySrc.strName
in_repSrc.publishBqy(filBqy, strN, strD, uuiFold, blnD , strH, oceMap);
```

Method and Properties References

This topic includes reference tables for methods and properties.

Reference for env Methods

Method	Description
createTempFile()	Create a temporary file that is cleaned up when the script completes
expandRequestAction()	Add a new sub-task for each set of values
getBqyDocument()	Construct a DOM from the content of an Interactive Reporting document
getDescription()	Retrieve the description associated with the script
getFileLines()	Read the lines of a file and construct an array that contains one string per line
getLogLevel()	Retrieve the current default log level that is used when calling log()
getMimeTypeUuid()	Retrieve the UUID of the specified MIME type
getNullUuid()	Retrieve a null UUID constant
getParameterValue()	Retrieve the value of the specified script parameter
getParameterValues()	Retrieve all of the values assigned to a multi-value script parameter
getRepository()	Retrieve an object that can be used to access the content of the repository
isDesktopMode()	Determine whether the script is being run on the desktop
isServerMode()	Determine whether the script is being run in EPM Workspace
loadScript()	Load the content of another script into this script environment
log()	Post a message at the current default logging level
logAlways()	Post a message that is always written to the log
logClassName()	Post a message that contains the specified Java class name of the object
logDebug()	Post a message for debugging
logError()	Post a message associated with a detected error condition
logFatal()	Post a message associated with a detected error condition
logInfo()	Post an informational message
logWarn()	Post a warning message
md5Hash()	Generate an MD5 hash from the specified string
setLogLevel()	Set the default level at which logging is to be performed
setProgress()	Update the progress of the script
updateDescription()	Set a new description for this script invocation
writeBqyDom()	Write the specified DOM out to a file

Reference for Repository Methods

Method	Description
addVersion()	Add a version of a document
convertBqyFileToUnicode()	Convert the specified document from code page to Unicode
findFiles()	Find all files in a folder
getFileUuid()	Retrieve the UUID of the file with a specified path
getFolderUuid()	Retrieve the UUID of the folder with a specified path
isFile()	Determine whether the specified UUID represents a file
isFolder()	Determine whether the specified UUID represents a folder
publishBqyFile()	Import a file into the repository with the specified content
remapOCEs()	Remap the OCEs of the specified document to the provided set
retrieveFile()	Retrieve the document with the specified UUID as a temporary file
retrieveVersionedDocument()	Retrieve the versioned document associated with the specified UUID

EPM Workspace-Specific Repository Methods

Method	Description
changeToFolder()	Change the logical position within EPM Workspace to the specified folder path
folderExists()	Determine whether a folder with the specified path exists in EPM Workspace
getCurrentFolder()	Retrieve the path to the current folder where this script is located in EPM Workspace
getFolderContentsFor()	Retrieve the UUIDs of all files in the folder
getPathForUuid()	Get the path in Oracle Enterprise Performance Management Workspace, Fusion Edition represented by the specified UUID
getSubfolderPathsFor()	Retrieve the UUIDs of all subfolders of the folder
makeFolder()	Create a subfolder with the specified name

Reference for Node Methods

Method	Description
addChild()	Add a child under this node

Method	Description
addProperty()	Add the specified property to this node
dump()	Dump the content of the node and the children of the node to standard output
findNodeByPattern()	Find one node that matches the specified pattern
findNodesByPattern()	Find all nodes that match the specified pattern
getChildren()	Retrieve a list of all the children of this node
getChildrenOfType()	Retrieve a list of all the children of this node with the specified node type
getContextualName()	Retrieve the logical name of this node
getNodeType()	Retrieve the type of this node
getPathWithContext()	Retrieve a string that represents the location of this node in the document, including contextual information to make the path unique
getProperties()	Retrieve a list of properties for this node
getProperty()	Retrieve the property of this node with the specified name
getRoot()	Retrieve the root node of the DOM in which this node is stored
hasProperty()	Determine whether this node has a property with the specified name
newNode()	Construct a node
removeChild()	Remove the specified child node
removeProperties()	Remove the specified list of properties from this node
replaceChildNode()	Replace the specified child node with the node provided
setChildren()	Replace the list of children of this node with the provided list

Reference for document

A document retrieved by using env.getBqyDocument() contains these properties.

Property	Description
DesignPassword	Password required to enter Design mode
DocumentPassword	Password required to open the document
EncryptedScripts	Determines whether scripts in the document are encrypted
EventScripts	Document-level scripts
Name	Document name

Property	Description
Path	Path to the document
Root_MyDocument	Root.MyDocument node
Root_MyResources	Root.MyResources node (or null if the document does not include Resource Manager data)
Sections	All sections contained in the document
Type	Retrieve the runtime class name
Unicode	Determines whether the document string content is in Unicode or code page format

The same document also contains these methods.

Method	Description
copy()	Copy the specified section to the document, rename it, if necessary, to avoid duplicates
getChartSections()	Retrieve a list of all chart sections
getChildrenWithRuntimeClass()	Retrieve all child nodes with a specified RuntimeClassName
getCodePage()	Retrieve the code page used by the document
getDashboardSections()	Retrieve a list of all the dashboard sections
getInitialTCSection()	Retrieve the home section identifier
getPivotSections()	Retrieve a list of all pivot sections
getQuerySections()	Retrieve a list of all query sections
getResultsSections()	Retrieve a list of all results sections
getSource()	Get the path to the Interactive Reporting document from which this document was loaded
getTableSections()	Retrieve a list of all table sections
isBQYPasswordProtected()	Determine whether the document has a password
isBQYProcessable()	Determine whether the document has at least one processable section
load()	Load a document from an Interactive Reporting document on disk
optimizeImages()	Optimize all of the Resource Manager images to remove duplicates
save()	Save the document to an Oracle's Hyperion® Interactive Reporting document on disk
sectionOCEPairInfos()	Retrieve a list of all the document OCE mappings
setCodePage()	Set the document code page

Method	Description
setEndianness()	Set whether the document should be stored as big- or small-endian
setHeader()	Set the document header
setSource()	Set the path to the source from which this document was loaded

Glossary

access permissions A set of operations that a user can perform on a resource.

accountability map A visual, hierarchical representation of the responsibility, reporting, and dependency structure of the accountability teams (also known as critical business areas) in an organization.

active service A service whose Run Type is set to Start rather than Hold.

active user A user who is entitled to access the system.

active user/user group The user or user group identified as the current user by user preferences. Determines default user preferences, dynamic options, access, and file permissions. You can set the active user to your user name or any user group to which you belong.

adaptive states Interactive Reporting Web Client level of permission.

aggregate cell A cell comprising several cells. For example, a data cell that uses Children(Year) expands to four cells containing Quarter 1, Quarter 2, Quarter 3, and Quarter 4 data.

aggregate limit A limit placed on an aggregated request line item or aggregated metatopic item.

alias An alternative name. For example, for a more easily identifiable column descriptor you can display the alias instead of the member name.

appender A Log4j term for destination.

application (1) A software program designed to run a specific task or group of tasks such as a spreadsheet program or database management system. (2) A related set of dimensions and dimension members that are used to meet a specific set of analytical and/or reporting requirements.

artifact An individual application or repository item; for example, scripts, forms, rules files, Interactive Reporting documents, and financial reports. Also known as an object.

attribute Characteristics of a dimension member. For example, Employee dimension members may have attributes of Name, Age, or Address. Product dimension members can have several attributes, such as a size and flavor.

attribute dimension A type of dimension that enables analysis based on the attributes or qualities of dimension members.

authentication service A core service that manages one authentication system.

axis (1) A straight line that passes through a graphic used for measurement and categorization. (2) A report aspect used to arrange and relate multidimensional data, such as filters, pages, rows, and columns. For example, for a data query in Simple Basic, an axis can define columns for values for Qtr1, Qtr2, Qtr3, and Qtr4. Row data would be retrieved with totals in the following hierarchy: Market, Product.

bar chart A chart that can consist of one to 50 data sets, with any number of values assigned to each data set. Data sets are displayed as groups of corresponding bars, stacked bars, or individual bars in separate rows.

batch POV A collection of all dimensions on the user POV of every report and book in the batch. While scheduling the batch, you can set the members selected on the batch POV.

book A container that holds a group of similar Financial Reporting documents. Books may specify dimension sections or dimension changes.

book POV The dimension members for which a book is run.

bookmark A link to a reporting document or a Web site, displayed on a personal page of a user. The two types of bookmarks are My Bookmarks and image bookmarks.

bounding rectangle The required perimeter that encapsulates the Interactive Reporting document content when embedding Interactive Reporting document sections in a personal page, specified in pixels for height and width or row per page.

cache A buffer in memory that holds data temporarily.

calculation The process of aggregating data, or of running a calculation script on a database.

Catalog pane Displays a list of elements available to the active section. If Query is the active section, a list of database tables is displayed. If Pivot is the active section, a list of results columns is displayed. If Dashboard is the active section, a list of embeddable sections, graphic tools, and control tools are displayed.

categories Groupings by which data is organized. For example, Month

cause and effect map Depicts how the elements that form your corporate strategy relate and how they work together to meet your organization's strategic goals. A Cause and Effect map tab is automatically created for each Strategy map.

cell (1) The data value at the intersection of dimensions in a multidimensional database; the intersection of a row and a column in a worksheet. (2) A logical group of nodes belonging to one administrative domain.

chart A graphical representation of spreadsheet data. The visual nature expedites analysis, color-coding, and visual cues that aid comparisons.

chart template A template that defines the metrics to display in Workspace charts.

child A member with a parent above it in the database outline.

choice list A list of members that a report designer can specify for each dimension when defining the report's point of view. A user who wants to change the point of view for a dimension that uses a choice list can select only the members specified in that defined member list or those members that meet the criteria defined in the function for the dynamic list.

clustered bar charts Charts in which categories are viewed side-by-side; useful for side-by-side category analysis; used only with vertical bar charts.

column A vertical display of information in a grid or table. A column can contain data from one field, derived data from a calculation, or textual information.

computed item A virtual column (as opposed to a column that is physically stored in the database or cube) that can be calculated by the database during a query, or by Interactive Reporting Studio in the Results section. Computed items are calculations of data based on functions, data items, and operators provided in the dialog box and can be included in reports or reused to calculate other data.

connection file *See Interactive Reporting connection file (.oce).*

content Information stored in the repository for any type of file.

cookie A segment of data placed on your computer by a Web site.

correlated subqueries Subqueries that are evaluated once for every row in the parent query; created by joining a topic item in the subquery with a topic in the parent query.

critical business area (CBA) An individual or a group organized into a division, region, plant, cost center, profit center, project team, or process; also called accountability team or business area.

critical success factor (CSF) A capability that must be established and sustained to achieve a strategic objective; owned by a strategic objective or a critical process and is a parent to one or more actions.

cube A block of data that contains three or more dimensions. An Essbase database is a cube.

custom calendar Any calendar created by an administrator.

custom report A complex report from the Design Report module, composed of any combination of components.

dashboard A collection of metrics and indicators that provide an interactive summary of your business. Dashboards enable you to build and deploy analytic applications.

data function That computes aggregate values, including averages, maximums, counts, and other statistics, that summarize groupings of data.

data layout The data layout interface is used to edit a query, arrange dimensions, make alternative dimension member selections, or specify query options for the current section or data object.

data model A representation of a subset of database tables.

database connection File that stores definitions and properties used to connect to data sources and enables database references to be portable and widely used.

descendant Any member below a parent in the database outline. In a dimension that includes years, quarters, and months, the members Qtr2 and April are descendants of the member Year.

Design Report An interface in Web Analysis Studio for designing custom reports, from a library of components.

detail chart A chart that provides the detailed information that you see in a Summary chart. Detail charts appear in the Investigate Section in columns below the Summary charts. If the Summary chart shows a Pie chart, then the Detail charts below represent each piece of the pie.

dimension A data category used to organize business data for retrieval and preservation of values. Dimensions usually contain hierarchies of related members grouped within them. For example, a Year dimension often includes members for each time period, such as quarters and months.

dimension tab In the Pivot section, the tab that enables you to pivot data between rows and columns.

dimension table (1) A table that includes numerous attributes about a specific business process. (2) In Essbase Integration Services, a container in the OLAP model for one or more relational tables that define a potential dimension in Essbase.

display type One of three Web Analysis formats saved to the repository: spreadsheet, chart, and pinboard.

dog-ear The flipped page corner in the upper right corner of the chart header area.

drill-down Navigation through the query result set using the dimensional hierarchy. Drilling down moves the user perspective from aggregated data to detail. For example, drilling down can reveal hierarchical relationships between years and quarters or quarters and months.

drill-through The navigation from a value in one data source to corresponding data in another source.

dynamic report A report containing data that is updated when you run the report.

Edit Data An interface for changing values and sending edits to Essbase.

employee A user responsible for, or associated with, specific business objects. Employees need not work for an organization; for example, they can be consultants. Employees must be associated with user accounts for authorization purposes.

ending period A period enabling you to adjust the date range in a chart. For example, an ending period of "month", produces a chart showing information through the end of the current month.

exceptions Values that satisfy predefined conditions. You can define formatting indicators or notify subscribing users when exceptions are generated.

external authentication Logging on to Oracle's Hyperion applications with user information stored outside the applications, typically in a corporate directory such as MSAD or NTLM.

externally triggered events Non-time-based events for scheduling job runs.

Extract, Transform, and Load (ETL) Data source-specific programs for extracting data and migrating it to applications.

fact table The central table in a star join schema, characterized by a foreign key and elements drawn from a dimension table. This table typically contains numeric data that can be related to all other tables in the schema.

filter A constraint on data sets that restricts values to specific criteria; for example, to exclude certain tables, metadata, or values, or to control access.

folder A file containing other files for the purpose of structuring a hierarchy.

footer Text or images at the bottom of report pages, containing dynamic functions or static text such as page numbers, dates, logos, titles or file names, and author names.

format Visual characteristics of documents or report objects.

free-form grid An object for presenting, entering, and integrating data from different sources for dynamic calculations.

generic jobs Non-SQR Production Reporting or non-Interactive Reporting jobs.

grid POV A means for specifying dimension members on a grid without placing dimensions in rows, columns, or page intersections. A report designer can set POV values at the grid level, preventing user POVs from affecting the grid. If a dimension has one grid value, you put the dimension into the grid POV instead of the row, column, or page.

group A container for assigning similar access permissions to multiple users.

highlighting Depending on your configuration, chart cells or ZoomChart details may be highlighted, indicating value status: red (bad), yellow (warning), or green (good).

host A server on which applications and services are installed.

host properties Properties pertaining to a host, or if the host has multiple Install_Homes, to an Install_Home. The host properties are configured from the LSC.

hyperlink A link to a file, Web page, or an intranet HTML page.

Hypertext Markup Language (HTML) A programming language specifying how Web browsers display data.

image bookmarks Graphic links to Web pages or repository items.

implied share A member with one or more children, but only one is consolidated, so the parent and child share a value.

inactive group A group for which an administrator has deactivated system access.

inactive service A service suspended from operating.

inactive user A user whose account has been deactivated by an administrator.

Install_Home A variable for the directory where Oracle's Hyperion applications are installed. Refers to one instance of Oracle's Hyperion application when multiple applications are installed on the same computer.

Interactive Reporting connection file (.oce) Files encapsulating database connection information, including: the database API (ODBC, SQL*Net, etc.), database software, the database server network address, and database user name. Administrators create and publish Interactive Reporting connection files (.oce).

intersection A unit of data representing the intersection of dimensions in a multidimensional database; also, a worksheet cell.

Investigation *See drill-through.*

Java Database Connectivity (JDBC) A client-server communication protocol used by Java based clients and relational databases. The JDBC interface provides a call-level API for SQL-based database access.

job output Files or reports produced from running a job.

job parameters Reusable, named job parameters that are accessible only to the user who created them.

jobs Documents with special properties that can be launched to generate output. A job can contain Interactive Reporting, SQR Production Reporting, or generic documents.

join A link between two relational database tables or topics based on common content in a column or row. A join typically occurs between identical or similar items within different tables or topics. For example, a record in the Customer table is joined to a record in the Orders table because the Customer ID value is the same in each table.

JSP Java Server Pages.

layer (1) The horizontal location of members in a hierarchical structure, specified by generation (top down) or level (bottom up). (2) Position of objects relative to other objects. For example, in the Sample Basic database, Qtr1 and Qtr4 are in the same layer, so they are also in the same generation, but in a database with a ragged hierarchy, Qtr1 and Qtr4 might not be in same layer, though they are in the same generation.

legend box A box containing labels that identify the data categories of a dimension.

level A layer in a hierarchical tree structure that defines database member relationships. Levels are ordered from the bottom dimension member (level 0) up to the parent members.

line chart A chart that displays one to 50 data sets, each represented by a line. A line chart can display each line stacked on the preceding ones, as represented by an absolute value or a percent.

link (1) A reference to a repository object. Links can reference folders, files, shortcuts, and other links. (2) In a task flow, the point where the activity in one stage ends and another begins.

linked data model Documents that are linked to a master copy in a repository

linked reporting object (LRO) A cell-based link to an external file such as cell notes, URLs, or files with text, audio, video, or pictures. (Only cell notes are supported for Essbase LROs in Financial Reporting.)

local report object A report object that is not linked to a Financial Reporting report object in Explorer. *Contrast with linked reporting object (LRO).*

local results A data model's query results. Results can be used in local joins by dragging them into the data model. Local results are displayed in the catalog when requested.

locked data model Data models that cannot be modified by a user.

LSC services Services configured with the Local Service Configurator. They include Global Services Manager (GSM), Local Services Manager (LSM), Session Manager, Authentication Service, Authorization Service, Publisher Service, and sometimes, Data Access Service (DAS) and Interactive Reporting Service.

Map Navigator A feature that displays your current position on a Strategy, Accountability, or Cause and Effect map, indicated by a red outline.

master data model An independent data model that is referenced as a source by multiple queries. When used, "Locked Data Model" is displayed in the Query section's Content pane; the data model is linked to the master data model displayed in the Data Model section, which an administrator may hide.

MDX (multidimensional expression) The language that give instructions to OLE DB for OLAP- compliant databases, as SQL is used for relational databases. When you build the OLAPQuery section's Outliner, Interactive Reporting Clients translate requests into MDX instructions. When you process the query, MDX is sent to the database server, which returns records that answer your query. *See also SQL spreadsheet.*

measures Numeric values in an OLAP database cube that are available for analysis. Measures are margin, cost of goods sold, unit sales, budget amount, and so on. *See also fact table.*

member A discrete component within a dimension. A member identifies and differentiates the organization of similar units. For example, a time dimension might include such members as Jan, Feb, and Qtr1.

member list A named group, system- or user-defined, that references members, functions, or member lists within a dimension.

metadata A set of data that defines and describes the properties and attributes of the data stored in a database or used by an application. Examples of metadata are dimension names, member names, properties, time periods, and security.

metric A numeric measurement computed from business data to help assess business performance and analyze company trends.

MIME Type (Multipurpose Internet Mail Extension) An attribute that describes the data format of an item, so that the system knows which application should open the object. A file's mime type is determined by the file extension or HTTP header. Plug-ins tell browsers what mime types they support and what file extensions correspond to each mime type.

minireport A report component that includes layout, content, hyperlinks, and the query or queries to load the report. Each report can include one or more minireports.

missing data (#MISSING) A marker indicating that data in the labeled location does not exist, contains no value, or was never entered or loaded. For example, missing data exists when an account contains data for a previous or future period but not for the current period.

model (1) In data mining, a collection of an algorithm's findings about examined data. A model can be applied against a wider data set to generate useful information about that data. (2) A file or content string containing an application-specific representation of data. Models are the basic data managed by Shared Services, of two major types: dimensional and non-dimensional application objects. (3) In Business Modeling, a network of boxes connected to represent and calculate the operational and financial flow through the area being examined.

multidimensional database A method of organizing, storing, and referencing data through three or more dimensions. An individual value is the intersection point for a set of dimensions.

native authentication The process of authenticating a user name and password from within the server or application.

note Additional information associated with a box, measure, scorecard or map element.

null value A value that is absent of data. Null values are not equal to zero.

online analytical processing (OLAP) A multidimensional, multiuser, client-server computing environment for users who analyze consolidated enterprise data in real time. OLAP systems feature drill-down, data pivoting, complex calculations, trend analysis, and modeling.

origin The intersection of two axes.

page member A member that determines the page axis.

palette A JASC compliant file with a .PAL extension. Each palette contains 16 colors that complement each other and can be used to set the dashboard color elements.

performance indicator An image file used to represent measure and scorecard performance based on a range you specify; also called a status symbol. You can use the default performance indicators or create an unlimited number of your own.

personal pages A personal window to repository information. You select what information to display and its layout and colors.

personal recurring time events Reusable time events that are accessible only to the user who created them.

personal variable A named selection statement of complex member selections.

perspective A category used to group measures on a scorecard or strategic objectives within an application. A perspective can represent a key stakeholder (such as a customer, employee, or shareholder/financial) or a key competency area (such as time, cost, or quality).

pie chart A chart that shows one data set segmented in a pie formation.

pinboard One of the three data object display types. Pinboards are graphics, composed of backgrounds and interactive icons called pins. Pinboards require traffic lighting definitions.

pins Interactive icons placed on graphic reports called pinboards. Pins are dynamic. They can change images and traffic lighting color based on the underlying data values and analysis tools criteria.

plot area The area bounded by X, Y, and Z axes; for pie charts, the rectangular area surrounding the pie.

predefined drill paths Paths used to drill to the next level of detail, as defined in the data model.

presentation A playlist of Web Analysis documents, enabling reports to be grouped, organized, ordered, distributed, and reviewed. Includes pointers referencing reports in the repository.

primary measure A high-priority measure important to your company and business needs. Displayed in the Contents frame.

Production Reporting *See SQR Production Reporting.*

promotion The process of transferring artifacts from one environment or machine to another; for example, from a testing environment to a production environment.

property A characteristic of an artifact, such as size, type, or processing instructions.

proxy server A server acting as an intermediary between workstation users and the Internet to ensure security.

public job parameters Reusable, named job parameters created by administrators and accessible to users with requisite access privileges.

public recurring time events Reusable time events created by administrators and accessible through the access control system.

publish The process that enables a model owner to forward a model or model changes for inclusion in an enterprise model.

range A set of values including upper and lower limits, and values falling between limits. Can contain numbers, amounts, or dates.

reconfigure URL URL used to reload servlet configuration settings dynamically when users are already logged on to the Workspace.

recurring time event An event specifying a starting point and the frequency for running a job.

relational database A type of database that stores data in related two-dimensional tables. *Contrast with multidimensional database*.

report object In report designs, a basic element with properties defining behavior or appearance, such as text boxes, grids, images, and charts.

resources Objects or services managed by the system, such as roles, users, groups, files, and jobs.

result frequency The algorithm used to create a set of dates to collect and display results.

role The means by which access permissions are granted to users and groups for resources.

row heading A report heading that lists members down a report page. The members are listed under their respective row names.

RSC services Services that are configured with Remote Service Configurator, including Repository Service, Service Broker, Name Service, Event Service, and Job Service.

scale The range of values on the Y axis of a chart.

schedule Specify the job that you want to run and the time and job parameter list for running the job.

score The level at which targets are achieved, usually expressed as a percentage of the target.

scorecard Business Object that represents the progress of an employee, strategy element, or accountability element toward goals. Scorecards ascertain this progress based on data collected for each measure and child scorecard added to the scorecard.

scorecard report A report that presents the results and detailed information about scorecards attached to employees, strategy elements, and accountability elements.

secondary measure A low-priority measure, less important than primary measures. Secondary measures do not have Performance reports but can be used on scorecards and to create dimension measure templates.

Section pane Lists all sections that are available in the current Interactive Reporting Client document.

security agent A Web access management provider (for example, Netegrity SiteMinder) that protects corporate Web resources.

security platform A framework enabling Oracle's Hyperion applications to use external authentication and single sign-on.

services Resources that enable business items to be retrieved, changed, added, or deleted. Examples: Authorization and Authentication.

servlet A piece of compiled code executable by a Web server.

Servlet Configurator A utility for configuring all locally installed servlets.

sibling A child member at the same generation as another child member and having the same immediate parent. For example, the members Florida and New York are children of East and each other's siblings.

single sign-on Ability to access multiple Oracle's Hyperion products after a single login using external credentials.

SmartCut A link to a repository item, in URL form.

snapshot Read-only data from a specific time.

SPF files Printer-independent files created by a SQR Production Reporting server, containing a representation of the actual formatted report output, including fonts, spacing, headers, footers, and so on.

Spotlighter A tool that enables color coding based on selected conditions.

SQL spreadsheet A data object that displays the result set of a SQL query.

SQR Production Reporting A specialized programming language for data access, data manipulation, and creating SQR Production Reporting documents.

stacked charts A chart where the categories are viewed on top of one another for visual comparison. This type of chart is useful for subcategorizing within the current category. Stacking can be used from the Y and Z axis in all chart types except pie and line. When stacking charts the Z axis is used as the Fact/Values axis.

Start in Play The quickest method for creating a Web Analysis document. The Start in Play process requires you to specify a database connection, then assumes the use of a spreadsheet data object. Start in Play uses the highest aggregate members of the time and measures dimensions to automatically populate the rows and columns axes of the spreadsheet.

strategic objective (SO) A long-term goal defined by measurable results. Each strategic objective is associated with one perspective in the application, has one parent, the entity, and is a parent to critical success factors or other strategic objectives.

Strategy map Represents how the organization implements high-level mission and vision statements into lower-level, constituent strategic goals and objectives.

structure view Displays a topic as a simple list of component data items.

Structured Query Language A language used to process instructions to relational databases.

subscribe Flags an item or folder to receive automatic notification whenever the item or folder is updated.

Summary chart In the Investigates Section, rolls up detail charts shown below in the same column, plotting metrics at the summary level at the top of each chart column.

super service A special service used by the startCommonServices script to start the RSC services.

target Expected results of a measure for a specified period of time (day, quarter, etc.,)

time events Triggers for execution of jobs.

time scale Displays metrics by a specific period in time, such as monthly or quarterly.

token An encrypted identification of one valid user or group on an external authentication system.

top and side labels Column and row headings on the top and sides of a Pivot report.

top-level member A dimension member at the top of the tree in a dimension outline hierarchy, or the first member of the dimension in sort order if there is no hierarchical relationship among dimension members. The top-level member name is generally the same as the dimension name if a hierarchical relationship exists.

trace level Defines the level of detail captured in the log file.

traffic lighting Color-coding of report cells, or pins based on a comparison of two dimension members, or on fixed limits.

transformation (1) Transforms artifacts so that they function properly in the destination environment after application migration. (2) In data mining, modifies data (bidirectionally) flowing between the cells in the cube and the algorithm.

transparent login Logs in authenticated users without launching the login screen.

trusted password A password that enables users authenticated for one product to access other products without reentering their passwords.

trusted user Authenticated user

user directory A centralized location for user and group information. Also known as a repository or provider.

Web server Software or hardware hosting intranet or Internet Web pages or Web applications.

weight Value assigned to an item on a scorecard that indicates the relative importance of that item in the calculation of the overall scorecard score. The weighting of all items on a scorecard accumulates to 100%. For example, to recognize the importance of developing new features for a product, the measure for New Features Coded on a developer's scorecard would be assigned a higher weighting than a measure for Number of Minor Defect Fixes.

ws.conf A configuration file for Windows platforms.

wsconf_platform A configuration file for UNIX platforms.

Y axis scale Range of values on Y axis of charts displayed in Investigate Section. For example, use a unique Y axis scale for each chart, the same Y axis scale for all Detail charts, or the same Y axis scale for all charts in the column. Often, using a common Y axis improves your ability to compare charts at a glance.

Zero Administration Software tool that identifies version number of the most up-to-date plug-in on the server.

zoom Sets the magnification of a report. For example, magnify a report to fit whole page, page width, or percentage of magnification based on 100%.

ZoomChart Used to view detailed information by enlarging a chart. Enables you to see detailed numeric information on the metric that is displayed in the chart.

Index