Implementing InterSystems IRIS Business Intelligence

Version 2019.4
2020-01-28

InterSystems Corporation   1 Memorial Drive   Cambridge MA 02142   www.intersystems.com

Implementing InterSystems IRIS Business Intelligence
InterSystems IRIS Data Platform Version 2019.4 2020-01-28
Copyright © 2020 InterSystems Corporation
All rights reserved.

Table of Contents

About This Book

This book describes how to implement InterSystems IRIS Business Intelligence and discusses all implementation topics other than modeling. It includes the following chapters:

For a detailed outline, see the table of contents.

The other developer books for Business Intelligence are as follows:

- *Introduction to InterSystems Business Intelligence* briefly introduces Business Intelligence and the tools that it provides.
- *Developer Tutorial for InterSystems Business Intelligence* guides developers through the process of creating a sample that consists of a cube, subject areas, pivot tables, and dashboards.
- *Defining Models for InterSystems Business Intelligence* describes how to define the basic elements used in Business Intelligence queries: cubes and subject areas. It also describes how to define listing groups.
- *Advanced Modeling for InterSystems Business Intelligence* describes how to use the more advanced and less common Business Intelligence modeling features: computed dimensions, unstructured data in cubes, compound cubes, cube relationships, term lists, quality measures, KPIs, plug-ins, and other special options.
- *Using InterSystems MDX* introduces MDX and describes how to write MDX queries manually for use with cubes.
- *InterSystems MDX Reference* provides reference information on MDX as supported by Business Intelligence.

- *Client-Side APIs for InterSystems Business Intelligence* provides information on the Business Intelligence JavaScript and REST APIs, which you can use to create web clients for your Business Intelligence applications.

The following books are for both developers and users:

- *Using Dashboards and the User Portal* describes how to use the Business Intelligence User Portal and dashboards.
- *Creating Dashboards* describes how to create and modify dashboards in Business Intelligence.
- *Using the Analyzer* describes how to create and modify pivot tables, as well as use the Analyzer in general.

Also see the article *Using PMML Models in InterSystems IRIS®*.

1
Overview

This chapter provides an overview of InterSystems IRIS Business Intelligence and the implementation tools and process. It discusses the following topics:

- Purpose of Business Intelligence
- Components that Business Intelligence adds to your application
- Recommended architecture
- Implementation steps
- Implementation tools

Be sure to consult the online *InterSystems Supported Platforms* document for this release for information on system requirements for Business Intelligence.

1.1 Purpose of Business Intelligence

The purpose of InterSystems Business Intelligence is to enable you to embed business intelligence (BI) into your applications so that your users can ask and answer sophisticated questions of their data. Your application can include *dashboards*, which contain graphical widgets. The widgets display data and are driven by *pivot tables* and *KPIs* (key performance indicators). For a pivot table, a user can display a *listing*, which displays source values.

Pivot tables, KPIs, and listings are queries and are executed at runtime:

- A pivot table can respond to runtime input such as filter selections made by the user. Internally it uses an MDX (MultiDimensional eXpressions) query that communicates with a cube.

  A *cube* consists of a *fact table* and its indices. A fact table consists of a set of *facts* (rows), and each fact corresponds to a base record. For example, the facts could represent patients or departments. The system also generates a set of level tables. All the tables are maintained dynamically.

  Depending on your configuration and implementation, the system detects changes in your transactional tables and propagates them to the fact tables as appropriate.

  The system generates an MDX query automatically when a user creates the pivot table in the Analyzer.

- A KPI can also respond to runtime user input. Internally, it uses either an MDX query (with a cube) or an SQL query (with any table or tables).

  In either case, you create the query manually or copy it from elsewhere.

- A listing displays selected values from the source records used for the rows of the pivot table that the user has selected. Internally, a listing is an SQL query.

  You can specify the fields to use and let the system generate the actual query. Or you can specify the entire query.

Dashboards can include buttons and other controls that launch *actions*. Actions can apply or set filters, refresh the dashboard, open other dashboards or other URLs, run custom code, and so on. The system provides a set of standard actions, and you can define custom actions.

1.2 Business Intelligence Components to Add to Your Application

To add Business Intelligence to an application, you add some or all of the following components:

- Data connector classes. A data connector enables you to use an arbitrary SQL query as the source of a cube or a listing.
- Cube definition classes. A cube defines the elements used within Business Intelligence pivot tables, and controls the structure and contents of the corresponding fact table and indices.

  A cube definition points to the transactional class (or the data connector) that it uses as its basis.

  You can have any number of cubes, and you can use a given class as the basis of multiple cubes.

  For each cube, the system generates and populates a fact table class and other classes.
- Subject area classes.

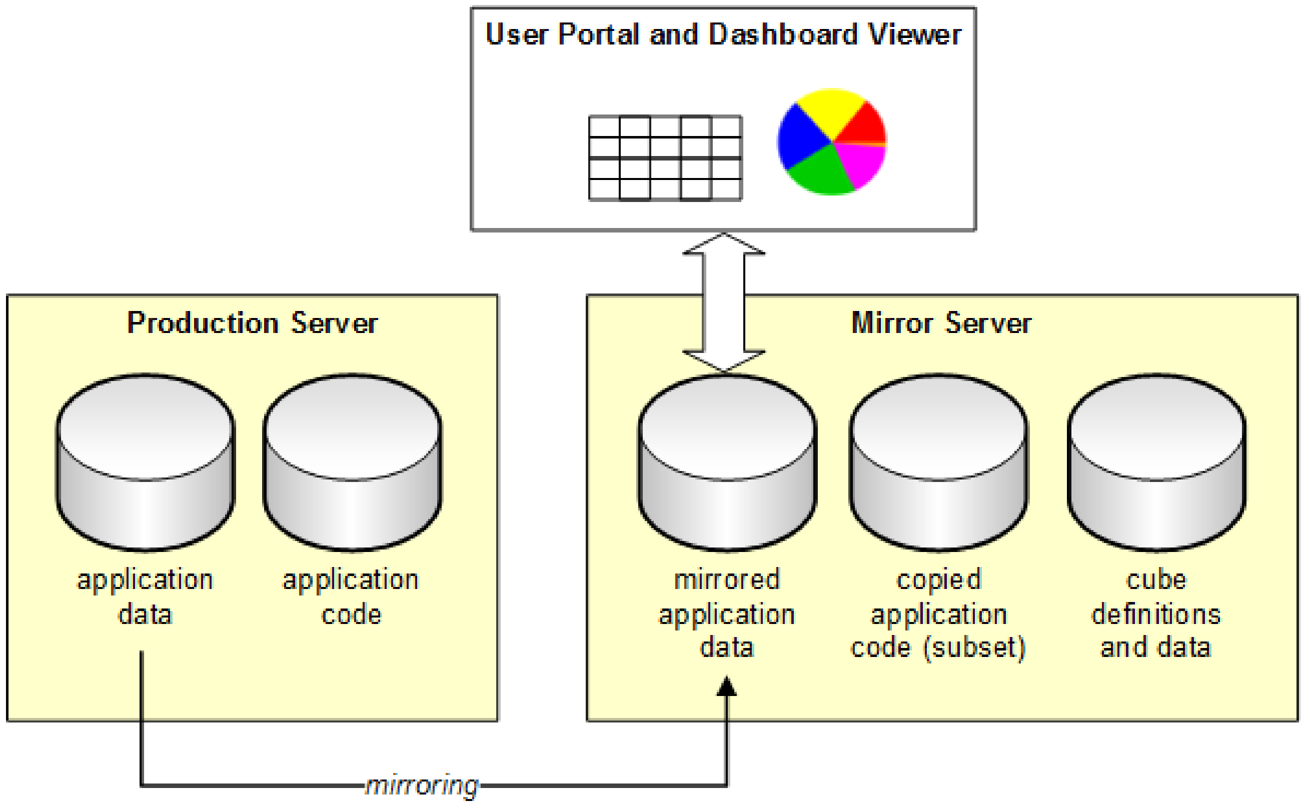
  A subject area is primarily a filtered cube. (It includes a filter and overrides for different parts of the cube definition, as wanted.) You can use cubes and subject areas interchangeably in Business Intelligence.
- KPI definition classes.

  You define KPIs when you need custom queries, particularly queries that are determined at runtime based on user input.

  You also define KPIs when you need custom actions, because actions are contained in KPI classes.
- Pivot tables, which you create by drag and drop. The system generates the underlying MDX queries.
- Dashboards, which display pivot tables and KPIs by running the underlying queries and displaying the results.
- The User Portal, which displays pivot tables and dashboards.

1.3 Recommended Architecture

As noted elsewhere, InterSystems generally recommends that you use mirroring as part of your high availability strategy. For any large-scale application, InterSystems recommends that you base your cubes on the application data that is on the mirror server, as shown in the following figure:


User Portal and Dashboard Viewer
Production Server
application data
application code
Mirror Server
mirrored application data
copied application code (subset)
cube definitions and data
mirroring

Specifically:

- Define your application so that the code and the data are in separate databases. This is not required, but is a typical architecture.
- Set up mirroring so that the application data is mirrored to the mirror server.
- So that the system can access the application data, copy some or all of the application classes and other code to the mirror server as well.

  It is not generally necessary to mirror the application code.
- On the mirror server, create a database to contain the cube definitions and (optionally) data.

  Optionally create another database to store the Business Intelligence fact table and other large-volume Business Intelligence data. The following chapter provides information on the globals that the system uses.
- On the mirror server, define a namespace in which to run Business Intelligence. In this namespace, define mappings to access the application data, application code, cube definitions, and Business Intelligence data on this server.

Note that for small-scale applications or demos, all the code and data can be in the same database.

For recommendations on Business Intelligence disaster recovery, see Business Intelligence and Disaster Recovery.

1.4 Main Implementation Steps

The implementation process includes the following steps:

1. If the namespace in which you want to use Business Intelligence does not yet define a web application, define a web application for it. See the chapter “Performing the Initial Setup.”

2. Optionally map the Business Intelligence globals from other databases, for performance.

   See the chapter "Performing the Initial Setup."

3. Create the cubes and optional subject areas. This process includes the following steps, which you iterate as needed:

   a. Define one or more cubes. In this step, you use either the Architect, Atelier, or both.

   b. Build the cubes. Here you use the Architect or the Terminal.

   c. Use the Analyzer to view the cubes and validate them.

   After the cubes are defined, define any subject areas based on those cubes.

   For information on creating cubes and subject areas, see *Defining Models for InterSystems Business Intelligence*.

   For information on using the Analyzer, see *Using the Analyzer*.

4. Optionally create KPIs. See the *Advanced Modeling for InterSystems Business Intelligence*.

5. Optionally create custom actions. See the chapter "Defining Custom Actions."

6. Make changes as needed to keep the cubes current. The way that you do this depends on how current the data must be, as well as any performance considerations.

   See the chapter "Keeping the Cubes Current."

7. Create pivot tables and dashboards. See *Using the Analyzer* and *Creating Dashboards*.

8. Package the pivot tables and dashboards into InterSystems IRIS® classes for easier deployment.

   See the chapter "Packaging Business Intelligence Elements into Classes."

9. Create links from your application to dashboards. See "Accessing Dashboards from Your Application."

At the appropriate points during this process, you may also have to do the following:

- Create data connectors — See "Defining Data Connectors."
- Configure settings — See "Configuring Settings."
- Perform localization — See "Performing Localization."
- Define custom portlets for use in dashboards — See "Creating Portlets for Use in Dashboards."
- Perform other development tasks — See "Other Development Work."
- Set up security – See "Setting Up Security for Business Intelligence."

1.5 Implementation Tools

You use the following tools during the implementation process:

- Tools available from the Business Intelligence section of the Management Portal:
  - Architect — Use this to define cubes and subject areas. Here you can also compile and build cubes (and compile subject areas).
  - Analyzer — Use this to examine cubes and subject areas when validating your model. Later you use it to create pivot tables.
  - User Portal — Use this to define dashboards.

  - MDX Query Tool — Use this to create MDX queries and view their query plans.
  - Folder Manager — Use this primarily to export pivot tables and dashboards so that you can package their definitions within an InterSystems IRIS class.

    You can also use it to associate resources with folders.
  - Settings option — Use this to specify the appearance and behavior of the User Portal, and to define variables that can be used in dashboards.
  - Business Intelligence Logs — Use this to see the Business Intelligence build log for this namespace.

- Atelier — Use this to define advanced cube features, any methods or routines used by cube elements, and any callback methods in the cube classes. You also use this to define KPIs.
- Terminal — You can use this to rebuild cubes and to test methods.
- MDX shell (running in the Terminal) — Use this to examine cubes and subject areas and to create custom MDX queries and see their results.
- Other sections of the Management Portal — Use these to map globals, define resources, roles, and users for use with Business Intelligence, and to examine the Business Intelligence fact tables if wanted.
- Utility methods:
  - %DeepSee.Utils includes methods that you can use to build cubes, synchronize cubes, clear the cell cache, and other tasks.
  - %DeepSee.UserLibrary.Utils includes methods that you can use to programmatically perform the tasks supported in the Folder Manager.
- The data connector class (%DeepSee.DataConnector) — Use this to make arbitrary SQL queries available for use in cubes and listings.
- The result set API (%DeepSee.ResultSet) — Use this to execute MDX queries programmatically and access the results.

1.6 Accessing the Samples Shown in This Book

Most of the samples in this book are part of the Samples-BI sample (https://github.com/intersystems/Samples-BI) or the Samples-Aviation sample (https://github.com/intersystems/Samples-Aviation).

InterSystems recommends that you create a dedicated namespace called SAMPLES (for example) and load samples into that namespace. For the general process, see *Downloading Samples for Use with InterSystems IRIS*.

These samples include cube definitions, subject areas, KPIs, data connectors, and plug-ins. They also include sample pivot tables and dashboards.

2

Performing the Initial Setup

This chapter describes setup activities to perform before you create cubes. It discusses the following topics:

- How to set up the web application
- How to place the Business Intelligence globals in a separate database
- Alternative mapping for the Business Intelligence globals
- How to set the web application session timeout period

2.1 Setting Up the Web Applications

In order to use InterSystems IRIS Business Intelligence in a web application, it is necessary to configure that web application so that it is *Analytics-enabled*. Specifically, a web application is Analytics-enabled if you select the **Enable Analytics** check box when you configure the application. For details on configuring web applications, see the chapter "Applications" in the *Security Administration Guide*.

The application name has an effect on how the application can be accessed; see the table below.

Web Application Configuration	In the Management Portal, the Business Intelligence menus link to this web application
• **Name** is /csp/namespace • **Namespace** is *namespace* • **Enable Analytics** is selected	YES (note that the Business Intelligence menus *always* try to access this web application — even if another web application is configured as the default, via the **Namespace Default Application** option)
• **Name** is any name other than /csp/namespace • **Namespace** is *namespace* • **Enable Analytics** is selected	NO (you can still access the web application by entering its URL in the browser)

2.2 Placing the Business Intelligence Globals in a Separate Database

When you use Business Intelligence in a given namespace, that increases the amount of data stored in the database (or databases) used by that namespace. If the source table is large, the system correspondingly stores a large amount of its own data. The Business Intelligence caches further increase the storage needs. As a consequence, it is generally a good idea to map some of the Business Intelligence globals to different databases. You can map all the Business Intelligence globals to a single database or you can define multiple mappings. As an example, the following steps describe how to place all the Business Intelligence globals in a single separate database:

1. Create the database.

   When you do so, you might consider pre-expanding the database (that is, setting its initial size), to avoid disk fragmentation created by runtime expansion.

2. Add a global mapping in the namespace that contains the classes that you plan to use with Business Intelligence. When you do so:

   - For **Globals Database Location**, select the database that you just created.
   - For **Global Name**, type `DeepSee.*`

   Also see the next section for more specific mappings you might use.

3. Recompile all cube, subject area, and KPI classes in this namespace.

   Also rebuild all cubes.

For details on creating databases and mapping globals, see the chapter “Configuring InterSystems IRIS®” in the *System Administration Guide*.

2.3 Alternative Mappings for the Globals

In some cases, you might want to separately map the Business Intelligence and related globals to separate databases. The following table lists the key globals:

Items	Globals	Comments
Fact tables and their indices	• **^DeepSee.Fact** • **^DeepSee.FactRelation** • **^DeepSee.Index**	When you initially build the cube, you might disable journaling for the database that contains these globals. After that, enable journaling for the databases.
Globals used to keep cube synchronized with the source table	• **^OBJ.DSTIME** • **^DeepSee.Update**	See the chapter “Keeping the Cubes Current.”
Cube internals	• **^DeepSee.Cubes** • **^DeepSee.Dimension** • **^DeepSee.DimensionI**	

Items	Globals	Comments
Cube Manager	• **^DeepSee.CubeManager** • **^DeepSee.CubeManager.CubeEventD** • **^DeepSee.CubeManager.CubeEventI** • **^DeepSee.CubeManager.CubeRegistr**	See "Using the Cube Manager" in "Keeping the Cubes Current."
Listing groups	**^DeepSee.ListingGroups**	See "Defining Listing Groups" in *Defining Models for InterSystems Business Intelligence*.
Result cache (for large data sets)	• **^DeepSee.BucketList** • **^DeepSee.Cache.*** • **^DeepSee.JoinIndex** • **^DeepSee.UpdateCounter** • **^DeepSee.Listing**	You can disable journaling for the database that contains these globals. For information on the result cache, see "Cube Updates and the Result Cache," later in this book.
Items created in the Analyzer and in the Dashboard Designer	• **^DeepSee.Filters** • **^DeepSee.Folder*** • **^DeepSee.FolderItem***	See *Using the Analyzer* and *Creating Dashboards*.
Term lists	• **^DeepSee.TermList**	See the *Advanced Modeling for InterSystems Business Intelligence*.
Quality measures	• **^DeepSee.QMsrs**	See the *Advanced Modeling for InterSystems Business Intelligence*.
Pivot variables	• **^DeepSee.Variables**	See "Defining and Using Pivot Variables" in *Using the Analyzer*.
Other portal options	• **^DeepSee.DashboardSettings** (user-specific dashboard settings) • **^DeepSee.User.SendTo** (user email addresses) • **^DeepSee.User.Settings** (runtime variables) • **^DeepSee.User.Icons** (custom icons) • **^DeepSee.UserPortalSettings** (general settings and worklist settings) • **^DeepSee.UserPreferences** (recent items, per user) • **^DeepSee.PaperSizes** (see "Adding Paper Sizes," later in this book)	For most of these, see the chapter "Configuring Settings."

Items	Globals	Comments
Custom code	• **^DeepSee.InitCode** • **^DeepSee.AuditCode**	See the chapter "Other Development Work."
Recent history and logs	• **^DeepSee.AgentLog** • **^DeepSee.Last*** • **^DeepSee.PivotError** • **^DeepSee.QueryLog** • **^DeepSee.Session** • **^DeepSee.SQLError**	
InterSystems IRIS NLP	• **^IRIS.IK.***	
Internals used for processing	• **^DeepSee.ActiveTasks** • **^DeepSee.Build** • **^DeepSee.Cancel** • **^DeepSee.ComputedSQL** • **^DeepSee.Functions** • **^DeepSee.IDList** • **^DeepSee.Pivot** • **^DeepSee.Shell** • **^DeepSee.TaskGroups** • **^DeepSee.Tasks** • **^DeepSee.UI.Charts**	

This is not a comprehensive list; the system uses additional globals with names that start `^DeepSee`. Globals not listed here typically contain only small amounts of data or are typically defined only briefly.

2.4 Adjusting the Web Session Timeout Period

The User Portal respects the web session timeout period for the namespace you are working in. The default session timeout period is 15 minutes, which might not be long enough.

To increase the web session timeout period:

1. Go to the Management Portal.
2. Click **System** > **System Administration**> **Security** > **Applications** > **Web Application**.
3. Click **Edit** in the row for the namespace in which you are using Business Intelligence.
4. Change the value of **Session Timeout**, which specifies the default timeout period for the web session, in seconds.

5. Click **Save**.

3

Configuring Settings

This chapter describes how to configure options that affect the appearance and behavior of InterSystems IRIS Business Intelligence. It discusses the following topics:

- How to access the Business Intelligence settings
- How to specify the basic settings
- How to configure Business Intelligence to support email
- How to customize worklists
- How to create runtime variables for use as default values for filters
- Allowed default values for filters
- How to create icons
- How to create custom color palettes

Also see "Accessing the Samples Shown in This Book," in the first chapter.

3.1 Accessing the Business Intelligence Settings

To access the Business Intelligence settings:

1. Click the InterSystems Launcher and then click **Management Portal**.

   Depending on your security, you may be prompted to log in with an InterSystems IRIS® username and password.

2. Switch to the appropriate namespace as follows:

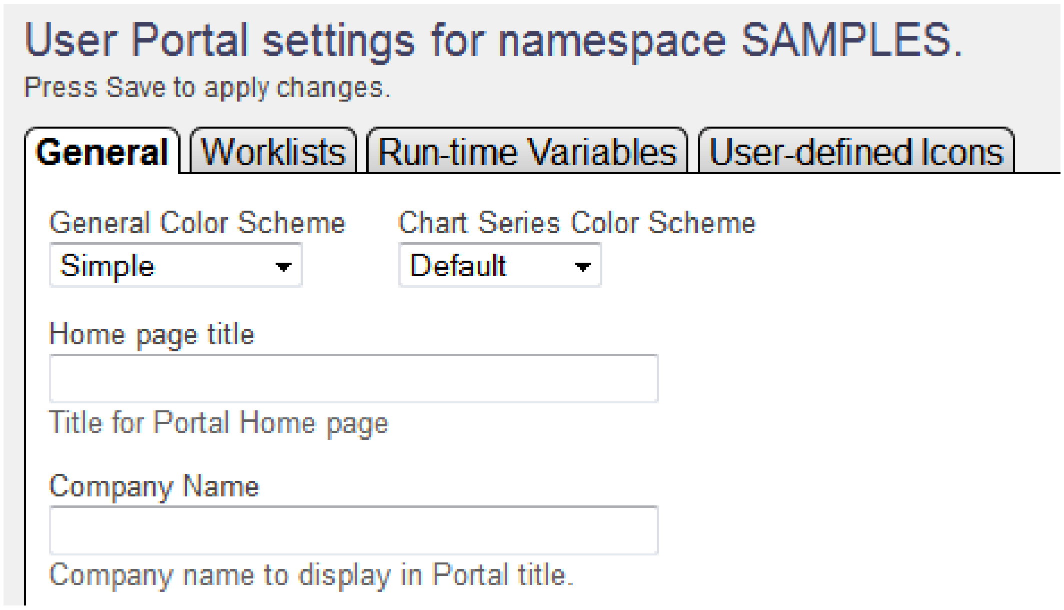
   a. Click **Switch**.

   b. Click the namespace.

   c. Click **OK**.

3. Click **Analytics** > **Admin** > **Settings**.

The system displays the following page:


User Portal settings for namespace SAMPLES.
Press Save to apply changes.

General Color Scheme: Simple
Chart Series Color Scheme: Default
Home page title
Title for Portal Home page
Company Name
Company name to display in Portal title.

3.2 Specifying Basic Settings

On the **General** tab, you can specify the following settings:

- **Color Scheme** — Select a color scheme for the User Portal.
- **Chart Series Color Scheme** — Select a color scheme for chart series. This is used as the default color scheme. Via the Dashboard Editor, users can apply a different color scheme to a given chart.
- **Home Page Title** — Specify the title for the browser page or tab.
- **Company Name** — Select a title to display in the upper right area of the User Portal.

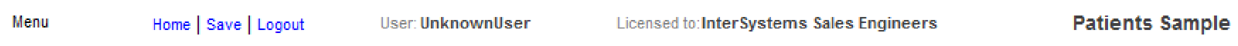
  If you specify this, do not specify **Company Logo**.
- **Company Logo** — Specify the URL of an image to display to the right of the company name.

  Specify either a complete URL, starting with `http://` or a URL relative to the web application defined for this namespace.

  If you specify this, **Company Name** is ignored.
- **Company Link** — Specify the URL to open when a user clicks the company logo or name in the upper right.

  Specify either a complete URL, starting with `http://` or a URL relative to the web application defined for this namespace.
- **Google Maps API Key** — Specify a key to use for the Google Maps API. Google has changed their policy regarding the use of the Google Maps libraries so that all new installations require an API key to function. See the Google Maps API Documentation for more information.
- **Dashboard email** — See the next topic.
- **Default Resource** — Default resource to use to secure pivot tables and dashboards.

  See "Adding Security for Business Intelligence Elements," later in this book.
- **No Dashboard Titles** — If this option is selected, the system hides the title area in the User Portal and in all dashboards. The title area is this area:



This option is equivalent to the NOTITLE URL parameter; see "Available URL Parameters," later in this book.

- **No Dashboard Borders** — If this option is selected, the system hides the border in the User Portal and in all dashboards. This option is equivalent to the NOBORDER URL parameter; see "Available URL Parameters," later in this book.
- **Show Calculated Members in Filters** — If this option is selected, calculated members that are part of existing cube dimensions will appear in filters. This setting does not affect calculated members that are part of special dimensions created by the definition of a calculated member.
- **Autosave** — These options enable or disable the autosave feature in this namespace. If the **Analyzer** check box is selected, the system automatically saves the state of the Analyzer for each user, for each pivot table. This means that when a given user opens a pivot table in the Analyzer, the system displays that pivot table as the user last saw it.

  Similarly, if the **User Portal Settings** check box is selected, the system automatically saves the state of the User Portal for each user, for each dashboard.

  In both the Analyzer and the User Portal, there is an option to clear the autosave state. (You can also remove all autosave data programmatically. See the **%KillAutosaveFolders()** method of %DeepSee.UserLibrary.Utils.)

Click **Save** after making any changes on this tab.

3.3 Configuring Business Intelligence to Support Email

On the **General** tab, you can configure Business Intelligence so that users can send email from within dashboards. To do so, use the **Dashboard email** setting. Select one of the following:

- **Use client-side email** — Enables email in Business Intelligence. When a user sends email, the system accesses the default client-side email system, which the user then uses to send a message. The message contains a link to the dashboard, and the user can edit the message.
- **Use server-side email** — Enables email in Business Intelligence. When a user sends email, the system displays a dialog box where the user types the email address and enters an optional comment, which the system adds to the message that it generates; this default message contains a link to the dashboard. The system then sends the email via an SMTP server.

  If you select this, you must also configure InterSystems IRIS to use an SMTP server. See "Configuring Task Manager Email Settings" in the chapter "Configuring InterSystems IRIS" in the *System Administration Guide*.
- **Disabled** — Disables support for email within Business Intelligence.

  This is the default.

3.4 Customizing Worklists

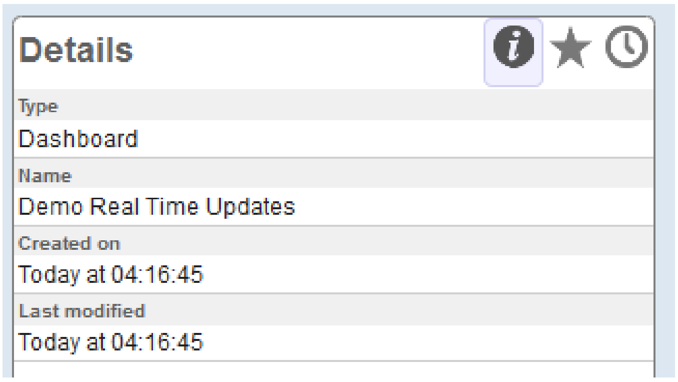
On the **Worklists** tab, you can customize how the system displays worklists. To do so, click **Customized worklists** and then select options in the following groups:

- The **Home Page Top Panel** and **Home Page Bottom Panel** options specify the worklists that are available in the User Portal, which always has two worklist areas on the left.

- The **Dashboard Page Top Panel** and **Dashboard Page Bottom Panel** options specify the worklists that are available in dashboards, which can have zero, one, or two worklist areas on the left, depending on their configuration.

In each section of this page, select the worklists to be available in the corresponding area. The available worklists are as follows:

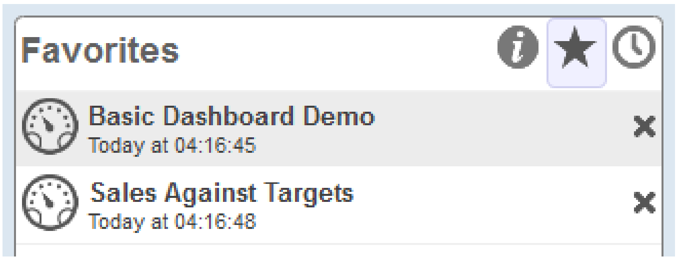
- The **Details** worklist displays details for the pivot table or dashboard that the user has selected. For example:



Details

Type
Dashboard
Name
Demo Real Time Updates
Created on
Today at 04:16:45
Last modified
Today at 04:16:45

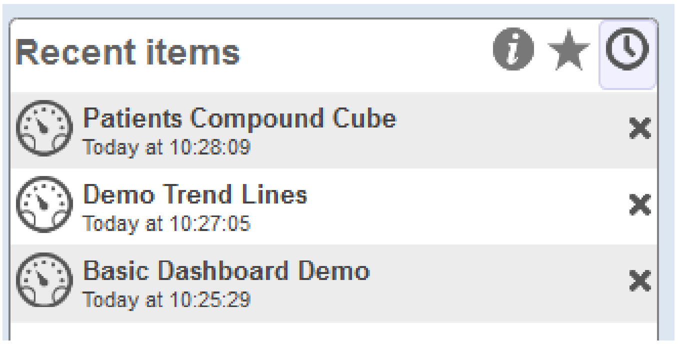
- The **Favorites** worklist displays any items that the user has marked as favorites. For example:



Favorites

Basic Dashboard Demo
Today at 04:16:45
Sales Against Targets
Today at 04:16:48

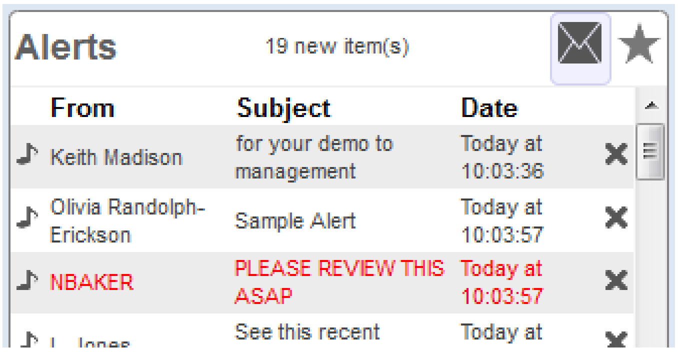
- The **Recent items** worklist displays items that the user has recently accessed. For example:



Recent items

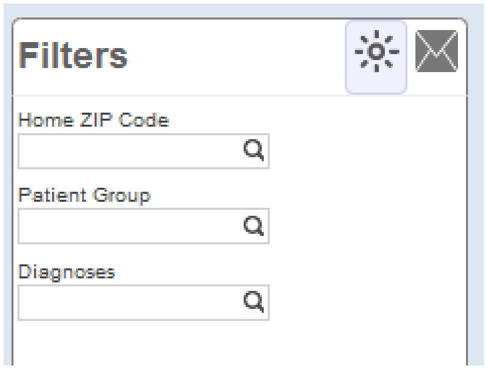
Patients Compound Cube
Today at 10:28:09
Demo Trend Lines
Today at 10:27:05
Basic Dashboard Demo
Today at 10:25:29

- The **Alerts** worklist displays recent alerts for the user. For example:



From	Subject	Date
Keith Madison	for your demo to management	Today at 10:03:36
Olivia Randolph-Erickson	Sample Alert	Today at 10:03:57
NBAKER	PLEASE REVIEW THIS ASAP	Today at 10:03:57
L. Jones	See this recent	Today at

- The **Filters** worklist displays filters and other controls in the dashboard. For example:



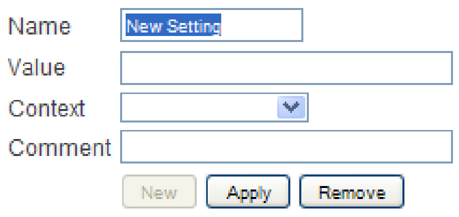
3.5 Creating Runtime Variables for Use as Default Values for Filters

On the **Run-time Variables** tab, you can define variables that have a logical name and a value that is an ObjectScript expression that is evaluated at runtime. You use these within dashboards for the default values of filters.

To add a setting:

1. Click **New**.

   The page then displays the following:



Name	New Setting
Value	
Context	
Comment	

2. Specify the following details:
   - **Name** — Specify the name of the variable.
   - **Value** — Specify an ObjectScript expression.

The value can be any valid ObjectScript expression. For example, it can be an invocation of a class method or routine; that method or routine can use special variables such as **$USERNAME** and **$ROLES**.

For details on the allowed values, see the section "Allowed Default Values for Filters."

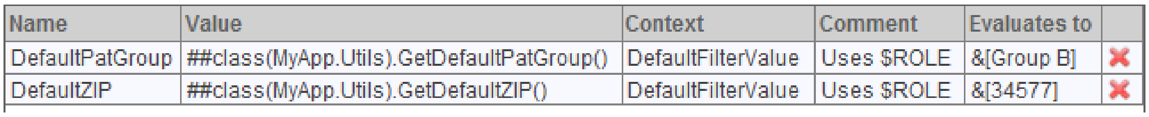
- **Context** — Select **DefaultFilterValue** to specify the context in which you will use this expression. Then the Widget Editor lists this setting as a possible default value for a filter, when you add a control to a widget.

  The value **Other** is currently not used.

- **Description** — Optionally specify a comment.

3. Click **Apply**.

   The variable is added to the table, which also shows its current value:



Name	Value	Context	Comment	Evaluates to	
DefaultPatGroup	##class(MyApp.Utils).GetDefaultPatGroup()	DefaultFilterValue	Uses $ROLE	&[Group B]	✗
DefaultZIP	##class(MyApp.Utils).GetDefaultZIP()	DefaultFilterValue	Uses $ROLE	&[34577]	✗

3.5.1 Editing Runtime Variables

To edit a runtime variable:

1. Click the variable in the table.
2. Edit the details in the area below the table.
3. Click **Apply**.

3.5.2 Removing Runtime Variables

To remove a runtime variable, click the X in the row for that variable.

The system immediately removes the variable.

3.6 Allowed Default Values for Filters

The following table lists the possible default values for filters, when used with an MDX-based data source. Use this information when you define runtime variables to use as filter defaults, or when you specify filters in other ways described in this book.

Scenario	Expression That Returns This Value
A single member	`"&[keyval]"` where *keyval* is the key for the member. See "Key Values" in the *InterSystems MDX Reference*.
A range of members	`"&[keyval1]:&[keyval2]"`
A set of members	`"{&[keyval1],&[keyval2],&[keyval3]}"`
All members of the level except for a specified single member	`"%NOT &[keyval]"`
All members of the level except for a specified subset	`"%NOT{&[keyval1],&[keyval2],&[keyval3]}"`

Note that for an MDX-based data source, the filter name and filter value are not case-sensitive (except for the optional `%NOT` string).

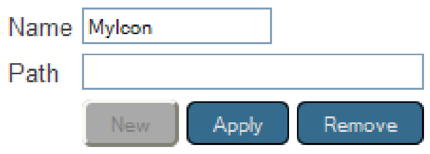
3.7 Creating Icons

On the **User-defined Icons** tab, you can define reusable icons with logical names. You can use these icons within pivot tables that have conditional formatting and within widget controls on dashboards.

To add an icon:

1. Click **New**.

   The bottom area of the page then displays the following:

   Name MyIcon
   Path

2. For **Name**, specify the name you will use to refer to this icon.
3. For **Path**, specify the location of the icon file. Do one of the following:
   - Specify a relative path that is relative to *install-dir*`/CSP/broker/`
   - Specify a complete URL.
4. Click **Apply**.

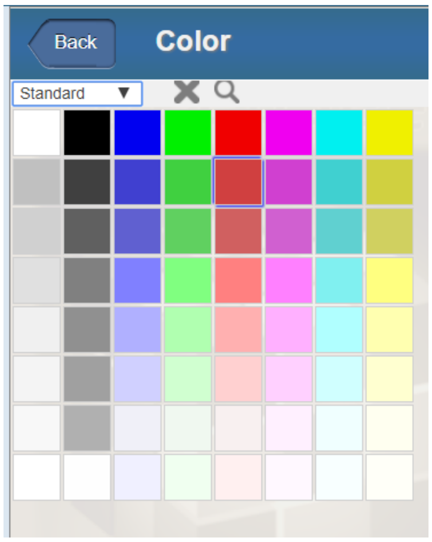
   The icon is added to the table, which also shows a preview.

You can edit or remove icons in the same way that you do with runtime variables. See the previous section for details.

For information on using icons in pivot tables with conditional formatting, see "Applying Conditional Formatting" in *Using the Analyzer*. For information on configuring widget controls, see "Adding a Control" in *Creating Dashboards*.

3.8 Creating Custom Color Palettes

You can also create custom color palettes, for use in the dashboard editor, which provides a color picker. The following shows this color picker with one of the default color palettes:



To add a custom color palette, add nodes to the `^DeepSee.UserPortalColorSets` global, as follows:

Node	Value
`^DeepSee.UserPortalColorSets(`*n*`)` where *n* is an integer, incremented from the previous node in the global.	A $LISTBUILD list that consists of the following items, in order: 1. Logical name of the color palette 2. Display name of the color palette. Optionally use $$$Text() to make this name localizable. 3. A list of CSS color names, separated by semi-colons.

For example:

```
set colorlist = "darkturquoise;greenyellow;hotpink;floralwhite;palevioletred;plum;"
set colorlist = colorlist _"powderblue;palegreen;plum;mediumaquamarine;linen;"
set colorlist = colorlist _"lightsteelblue;lightpink;oldlace;lightsalmon;gold;"
set mycolors=$LB("My Custom Colors","My Custom Colors",colorlist)
set ^DeepSee.UserPortalColorSets($I(^DeepSee.UserPortalColorSets)) = mycolors
```

When a user selects a color palette, the system displays a sample of each color in the grid. You can specify up to 64 colors.

4

Defining Data Connectors

This chapter describes how to define data connectors. It discusses the following topics:

- Introduction
- How to define basic data connectors
- How to preview the query results
- How to construct the data connector query at runtime
- How to use the data connector programmatically

Also see "Accessing the Samples Shown in This Book," in the first chapter.

4.1 Introduction to Data Connectors

A data connector maps the results of an arbitrary SQL SELECT query into an object that can be used as the source of a cube, for a detail listing, or both. (For information on defining cubes and listings, see *Defining Models for InterSystems Business Intelligence*.)

The SQL query can use a combination of the following:

- Local tables in the namespace in which you are using InterSystems IRIS Business Intelligence.
- Views in the same namespace.
- Linked tables in the same namespace. You define a linked table with the Link Table Wizard. The table has a class definition in your namespace but is linked to a table in an external database.

  Important: There are restrictions on queries when using linked tables. See "Restrictions on SQL Gateway Queries" in *Using the InterSystems IRIS SQL Gateway*.

You can define a data connector so that it supports updates to the cube. To update this cube, you must either rebuild the entire cube or use **ProcessFact()**; see "Keeping the Cubes Current."

4.2 Defining a Basic Data Connector

To define a data connector, create a class as follows:

- It must extend %DeepSee.DataConnector.
- It must specify a query. You can specify the query in an XData block, as described in the first subsection.

  Another possibility is to implement a callback to construct the query at runtime. This is described later in this chapter.
- It must define an output specification, which maps the query columns to properties, as described in the second subsection.
- If you need to use this data connector for a listing, the class must specify the *SUPPORTSIDLIST* class parameter as 1:

  ```
  Parameter SUPPORTSIDLIST = 1;
  ```

- If you need to use this data connector for a cube, and if you want to support cube updates, the class must specify the *SUPPORTSSINGLE* parameter as 1:

  ```
  Parameter SUPPORTSSINGLE = 1;
  ```

When you compile a data connector, the system generates a class with the name *packagename.classname*.ResultSet, where *packagename.classname* is the full name of the data connector class itself. Do not edit the generated class.

4.2.1 Defining the Query in an XData Block

To define the query in an XData block, add an element to the data connector class like the following:

```
XData SourceQuery [ XMLNamespace = "http://www.intersystems.com/deepsee/connector/query" ]
{
   <sql>SELECT %ID, DateOfSale, Product->Name AS ProductName FROM HoleFoods.SalesTransaction</sql>
}
```

Notes:

- You cannot use this technique if the data connector must support detail listings or updates. In such cases, instead see "Defining the Query at Runtime," later in this chapter.
- The name of this XData block must be `SourceQuery`
- The XMLNamespace parameter must equal `"http://www.intersystems.com/deepsee/connector/query"`
- The XData block must contain one `<sql>` element, which contains the SQL query to execute.
- The query must return the IDs of the records, in addition to other fields you need.
- To include the less than symbol (<) in the query, use `&lt;`

  For example:

  ```
  <sql>SELECT A,B,C FROM MyApp.MyTable WHERE A&lt;'50'</sql>
  ```

  Similarly, to include an ampersand (&) in the query, use `&amp;`
- If you are using this data connector for a listing or you need to support cube updates, the query must end with `WHERE $$$RESTRICT` token. For example:

  ```
  <sql>SELECT A,B,C FROM MyApp.MyTable WHERE $$$RESTRICT</sql>
  ```

  The `$$$RESTRICT` token is case-sensitive.

  Note: `$$$RESTRICT` is not a macro. It is replaced at runtime, not at compile time.

- If you use arrow syntax to access a field, it might be necessary to also supply an alias for the field. Specifically, an alias is required if you use the data connector as the basis of a cube and you want to use the field in the definition of a cube element.

  For example, consider the following query:

```
SELECT %ID, DateOfSale, Product->Name FROM HoleFoods.SalesTransaction
```

In this case, there is no way for a cube definition to refer to the `Product->Name` field; the build process throws an error if you use either `Product->Name` or `Product.Name`. As a consequence, you cannot use this field as the basis of a level or measure.

In contrast, consider this query:

```
SELECT %ID, DateOfSale, Product->Name AS ProductName FROM HoleFoods.SalesTransaction
```

In this case, you can treat `ProductName` as a property in the source class, so you can define a level or measure based on it.

4.2.2 Defining the Output Specification

Every data connector class must contain an XData block that maps the query columns to properties, as in the following example:

```
XData Output [ XMLNamespace = "http://www.intersystems.com/deepsee/connector/output" ]
{
<connector>
   <property name="Gender" sourceProperty="Gender" />
   <property name="Age" sourceProperty="Age" type="%ZEN.Datatype.integer"/>
   <property name="HomeCity" sourceProperty="HomeCity"/>
   <property name="PatientGroup" sourceProperty="PatientGroup"
             transform='$CASE(%val,"A":"Group A","B":"Group B",:%val)' />
   <property name="TestScore" sourceProperty="TestScore" type="%ZEN.Datatype.integer"/>
</connector>
}
```

Each `<property>` element is a *property* of the data connector and can be used by Business Intelligence.

Notes:

- The name of this XData block must be `Output`
- The XMLNamespace parameter must equal `"http://www.intersystems.com/deepsee/connector/output"`
- This XData block must contain one `<connector>` element.
- The `<connector>` element must include one or more `<property>` elements.
- Each `<property>` element must specify some or all of the following attributes:

Attribute	Purpose
name	Name of the property, for use as a source property in a cube, in a source expression in a cube, or as a field in a listing.
sourceProperty	Name of the corresponding column of the result set.
type	(Optional) Data type for the property. The default is %Library.String.
transform	(Optional) An expression that uses `%val` (the current column value) as input and returns a transformed value.

- If you are going to use this data connector for a listing, also specify the `idkey` attribute for the appropriate `<property>` element or elements. This attribute indicates that the given property or properties represent the IdKey of the data set.

  If you mark multiple fields with `idKey="true"`, the data connector combines these fields.

  Note: If you have a cube based on a data connector and listings in that cube that are also based on data connectors, all of these data connectors must have the same property (or properties) marked as `idkey="true"`, because the underlying mechanism uses the same ID values in all cases.

The following shows an example with `idkey`:

```
XData Output [ XMLNamespace = " http://www.intersystems.com/deepsee/connector/output" ]
{
<connector >
   <property name= "%ID" sourceProperty ="ID" displayName ="Record ID" idKey= "true"/>
   <property name= "Product" sourceProperty ="Product" displayName ="Product name"/>
   <property name= "AmountOfSale" sourceProperty ="AmountOfSale" displayName ="Amount of sale"/>
</connector >
}
```

4.3 Previewing the Query Results

To test a data connector, you can directly view the query results. To easily see the output for a data connector, use its **%Print()** class method in the Terminal. For example:

```
d ##class(BI.Model.PatientsQuery).%Print()
1     1         SUBJ_1003 M         27        Redwood
2     2         SUBJ_1003 M         41        Magnolia
3     3         SUBJ_1003 F         42        Elm Heigh
...
```

By default, this method prints the first 100 records of the output.

This method has the following signature:

```
classmethod %Print(ByRef pParameters, pMaxRows As %Integer = 100) as %Status
```

Where *pParameters* is currently not used, and *pMaxRows* is the maximum number of rows to display.

4.4 Defining the Query at Runtime

Instead of defining a hardcoded query in an XData block, you can construct the query at runtime. If the data connector must support detail listings or updates, you must use this technique.

To construct the query at runtime, implement the **%OnGetSourceResultSet()** method. This method has the following signature:

```
Method %OnGetSourceResultSet(ByRef pParameters, Output pResultSet) As %Status
```

Where *pParameters* is currently unused, and *pResultSet* is the result set.

In your implementation, do the following:

1. If you are using this data connector for multiple purposes, examine the %mode property of the data connector instance. The system automatically sets this property when it creates the data connector instance. This property has one of the following values:

   - `"all"` — Indicates that the cube is being built or that an All member is being shown.
   - `"idlist"` — Indicates that a listing is being requested.
   - `"single"` — Indicates that **%ProcessFact()** has been invoked.

2. Creates an instance of %SQL.Statement. The query must return the IDs of the records, in addition to other fields you need.

   The details of the query should be different, depending on the mode in which this data connector has been created. Typically:

- You define a basic query for use with the `"all"` mode.
- You add a restriction when the mode is `"single"`, to get the single record that is being updated. The first subsection provides details.
- You add a different restriction when the mode is `"idlist"`, to get a subset of the records. The second subsection provides details.

3. Execute that statement, optionally passing to it any runtime values as parameters. Certain runtime values are available as properties of the statement instance, as discussed in the following subsections.

   This step creates an instance of %SQL.StatementResult.

4. Return the instance of %SQL.StatementResult as an output parameter.

4.4.1 Restricting the Records When an Update Is Requested

When you update a cube with **ProcessFact()**, you indicate the ID of the record to update. When you create a data connector for use by a cube, you must add logic so that its query uses only the given ID. In this case, you can use the %singleId property of your data connector; it contains the ID of the record that is being updated. For example:

```
  //do this when constructing the SQL statement
  if (..%mode="single") {
      set sql = sql _ " where %ID = ?"
   }

...
  //do this when executing the SQL statement
  if (..%mode="single") {
             set pResultSet = tStatement.%Execute(..%singleId)
  }
```

For information on **ProcessFact()**, see the chapter "Keeping the Cubes Current."

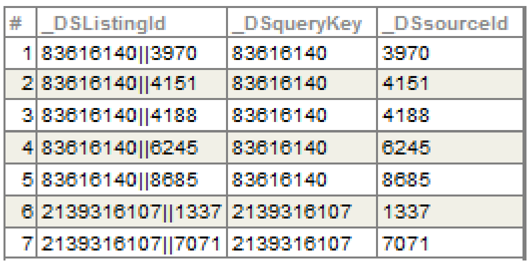
4.4.2 Restricting the Records When a Listing Is Requested

When a user requests a listing, the system retrieves the IDs of the records used in the given context and stores them for later use. For a default listing, the system automatically uses those IDs in the SQL query of the listing. When you create a data connector for use in a listing, you must add logic so that your query uses the IDs.

In this case, it is necessary to understand how the system stores the IDs for a listing. It writes these IDs to a table (the *listing table* for this cube), which includes the following columns:

- _DSqueryKey — Identifies a listing.
- _DSsourceId — An ID, as in the original source data.

The following shows an example:



#	_DSListingId	_DSqueryKey	_DSsourceId
1	83616140||3970	83616140	3970
2	83616140||4151	83616140	4151
3	83616140||4188	83616140	4188
4	83616140||6245	83616140	6245
5	83616140||8685	83616140	8685
6	2139316107||1337	2139316107	1337
7	2139316107||7071	2139316107	7071

Here, the first five rows are associated with the listing 83616140, which uses the IDs of five records, given in the _DSsourceId column. The next two rows are associated with the listing 2139316107, which uses the IDs of two records.

There are two ways to modify the data connector query to use the listing table:

- Add an IN clause to the query and use the applicable rows from the listing table in a subquery. The following shows an example:

  ```
  SELECT A,B,C FROM MyApp.MyTable
  WHERE (ID IN (SELECT _DSsourceId FROM listingtable WHERE
  _DSqueryKey=somekey))
  ```

  In this case:

  – `listingtable` is the name of the listing table for the cube. To get this table name, you use the %listingTable property of your data connector.

  – `somekey` is the unique key for the current listing. To get this key, you use the %listingKey property of your data connector.

  This approach can lead to <MAXSTRING> errors and other size-related issues.

- Perform a JOIN between the source table and the listing table with the correct WHERE clause.

The following shows an example, from a data connector that is used as the source for a cube and as the source for a listing. Notice that the listing key is passed to the query as a parameter.

```
Method %OnGetSourceResultSet(ByRef pParameters, Output pResultSet) As %Status
{
  set tSC = $$$OK
  set pResultSet = ""
  Try {
      set sql = "SELECT %ID, fdate, fname, ftimestamp FROM TestTD.TimeDimensions"
      //when we're using this for a listing, add WHERE clause to restrict to
      //the appropriate IDs (in the table given by the %listingTable property)

      if (..%mode="idlist") {
          set sql = sql _ " where %ID in (select _DSsourceId from "
                      _ ..%listingTable _ " where _DSqueryKey = ?)"
      }

      set tStatement = ##class(%SQL.Statement).%New()
      set tSC = tStatement.%Prepare(.sql)

      If $$$ISERR(tSC) {
          set ex = ##class(%Exception.StatusException).CreateFromStatus(tSC)
          throw ex
      }

      //if we're using this for a listing, pass in the listing key as a parameter
      if (..%mode="idlist") {
          set pResultSet = tStatement.%Execute(..%listingKey)
      } else {
          set pResultSet = tStatement.%Execute()
      }

      //check %SQLCODE and report if there's an error
      If pResultSet.%SQLCODE {
          set sqlcode=pResultSet.%SQLCODE
          set message=pResultSet.%Message
          set ex = ##class(%Exception.SQL).CreateFromSQLCODE(sqlcode, message)
          throw ex
          }
  }
  Catch(ex) {
      Set tSC = ex.AsStatus()
  }
  Quit tSC
}
```

4.4.3 Other Callbacks

The %DeepSee.DataConnector class provides additional callback methods that you can customize to handle errors, perform transformations on rows, perform filtering, and so on. These include **%OnNextRecord()** and **%OnProcessRecord()**. For details, see the *InterSystems Class Reference*.

4.5 Using a Data Connector Programmatically

To use a data connector programmatically, do the following:

1. Create an instance of it.
2. Invoke its **%Execute()** method, which returns a result set. This method also returns a status by reference.
3. Check the returned status.
4. If the status is not an error, you can now use the result set, which is an instance of %SQL.StatementResult.

For example:

```
Set tConnector=..%New()
Set tRS=tConnector.%Execute(,.tSC)
If $$$ISERR(tSC) {Quit}
//use tRS as needed ...
```

5 Performance Tips

This chapter contains performance tips for InterSystems IRIS Business Intelligence. The section "Placing the Business Intelligence Globals in a Separate Database," is related to this topic.

For more information on performance and troubleshooting options, see the InterSystems Developer Community. Also see "Accessing the Samples Shown in This Book," in the first chapter.

5.1 Result Caching and Cube Updates

For any cube that uses more than 64,000 records (by default), the system maintains and uses a result cache. When you update a cube in any way, parts of the result cache are considered invalid and are cleared. The details depend upon options in the cube definition (see "Cache Buckets and Fact Order," later in this chapter). Therefore, it is not generally desirable to update the cubes constantly.

The result cache works as follows: Each time a user executes a query (via the Analyzer for example), the system caches the results for that query. The next time any user runs that query, the system checks to see if the cache is still valid. If so, the system then uses the cached values. Otherwise, the system re-executes the query, uses the new values, and caches the new values. The net effect is that performance improves over time as more users run more queries.

5.2 Cache Buckets and Fact Order

As noted earlier, for large data sets, the system maintains and uses a result cache. In this case, it can be useful to control the order of rows in the fact table, because this affects how the system creates and uses the cache. To do this, you can specify the **Initial build order** option for the cube; see "Other Cube Options" in *Defining Models for InterSystems Business Intelligence*.

When users evaluate pivot tables, the system computes and caches aggregate values that it later reuses whenever possible. To determine whether the system can reuse a cache, the system uses the following logic:

1. It examines the IDs of the records used in a given scenario (for example, for a given pivot table cell).
2. It checks the buckets to which those IDs belong. A bucket is a large number of contiguous records in the fact table (details given later).
    - If the bucket has been updated (because there was a change for at least one ID in the bucket), the system discards any corresponding cache associated with that bucket and regenerates the result.

- If the bucket has not been updated, the system reuses the appropriate cache (if available) or generates the result (if not).

In some scenarios, changes to the source records (and the corresponding updates to any cubes) occur primarily in the most recent source records. In such scenarios, it is useful to make sure that you build the fact table in order by age of the records, with the oldest records first. This approach means that the caches for the older rows would not be made invalid by changes to the data. (In contrast, if the older rows and newer rows were mixed throughout the fact table, all the caches would potentially become invalid when changes occurred to newer records.)

For more information, see "How the Analytics Engine Works," later in this book.

5.3 Removing Inactive Cache Buckets

When a cache bucket is invalidated (as described in the previous section), it is marked as inactive but is not removed. To remove the inactive cache buckets, call the **%PurgeObsoleteCache()** method of %DeepSee.Utils. For example:

```
d ##class(%DeepSee.Utils).%PurgeObsoleteCache("patients")
```

5.4 Precomputing Cube Cells

As noted earlier, when users evaluate pivot tables, the system computes and caches aggregate values that it later reuses whenever possible. This caching means that the more users work with Business Intelligence, the more quickly the system runs. (For details, see "How the Analytics Engine Works," later in this book.)

To speed up *initial* performance as well, you can precompute and cache specific aggregate values that are used in your pivot tables, especially wherever performance is a concern. The feature works as follows:

- Within the cube class, you specify an additional XData block (`CellCache`) that specifies cube cells that should be precomputed and cached. For details, see the first subsection.
- You programmatically precompute these cube cells by using a utility method. See the second subsection.

  You must do this *after* building the cube.

Important: A simpler option is to simply run any queries ahead of time (that is, before any users work with them).

5.4.1 Defining the Cell Cache

Your cube class can contain an additional XData block (`CellCache`) that specifies cube cells that can be precomputed and cached, which speeds up the initial performance of Business Intelligence. The following shows an example:

```
/// This xml document defines aggregates to be precomputed.
XData CellCache [ XMLNamespace = " http://www.intersystems.com/deepsee/cellCache" ]
{
<cellCache xmlns= "http://www.intersystems.com/deepsee/cellCache" >
   <group name= "BS">
      <item>
         <element >[Measures].[Big Sale Count]</element >
      </item>
   </group>
   <group name= "G1">
      <item>
         <element >[UnitsPerTransaction].[H1].[UnitsSold]</ element>
         <element >[Measures].[Amount Sold]</element >
      </item>
      <item>
         <fact >DxUnitsSold</fact >
         <element >[Measures].[Amount Sold]</element >
      </item>
   </group>
</cellCache >
}
```

The `<cellCache>` element is as follows:

- It must be in the namespace `"http://www.intersystems.com/deepsee/cellCache"`
- It contains zero or more `<group>` elements.

Each `<group>` element is as follows:

- It has a `name` attribute, which you use later when specifying which groups of cells to precompute.
- It contains one or more `<item>` elements.

Each `<item>` element represents a combination of cube indices and corresponds to the information returned by %SHOWPLAN. An `<item>` element consists of one or more `<element>` elements.

An `<element>` can include one or more of either of the following structures, in any combination:

```
<fact>fact_table_field_name</fact>
```

Or:

```
<element>mdx_member_expression</element >
```

Where:

- *fact_table_field_name* is the field name in the fact table for a level or measure, as given by the `factName` attribute for that level or measure.
- *mdx_member_expression* is an MDX expression that evaluates to a member. This can be either a member of a level or it can be a measure name (each measure is a member of the special MEASURES dimension).

  This expression cannot be a calculated member.

Note: Each group defines a set of intersections. The number of intersections in a group affects the processing speed when you precompute the cube cells.

5.4.2 Precomputing the Cube Cells

To precompute the aggregate values specified by a `<group>`, use the **%ComputeAggregateGroup()** method of %DeepSee.Utils. This method is as follows:

```
classmethod %ComputeAggregateGroup(pCubeName As %String,
                                   pGroupName As %String,
                                   pVerbose As %Boolean  = 1) as %Status
```

Where *pCubeName* is the name of the cube, *pGroupName* is the name of the cube, and *pVerbose* specifies whether to write progress information while the method is running. For *pGroupName*, you can use `"*"` to precompute all groups for this cube.

If you use this method, you must first build the cube.

The method processes each group by looping over the fact table and computing the intersections defined by the items within the group. Processing is faster with fewer intersections in a group. The processing is single-threaded, which allows querying in the foreground.

6 Defining Custom Actions

This chapter describes how to define custom actions for use in InterSystems IRIS Business Intelligence dashboards. It discusses the following topics:

- Introduction to actions
- How to define actions
- Context information you can use on the server
- How to execute client-side commands
- How to display a different dashboard
- How to generate a table from cube context

Also see "Accessing the Samples Shown in This Book," in the first chapter.

6.1 Introduction

You define custom actions within KPI classes. Then:

- When you display a given KPI in a widget, you can add controls to that widget that invoke the custom actions. See "Adding Widget Controls" in *Creating Dashboards*.
- If you specify a KPI class as the `actionClass` attribute of the `<cube>` element, all actions within this class are available to pivot tables that use this cube, which means they can be added as controls to widgets that display these pivot tables.
- If you specify a KPI class as the `actionClass` attribute of another `<kpi>` element, all actions within this class are available to that KPI, in addition to any actions defined within that KPI.
- You can execute actions from within the Analyzer. Note that in this case, only a subset of the client-side commands are supported: **alert**, **navigate**, and **newWindow**. Other commands are ignored.

For details on defining KPIs, see the *Advanced Modeling for InterSystems Business Intelligence*.

You can perform many of the same operations with either a standard action or a custom action:

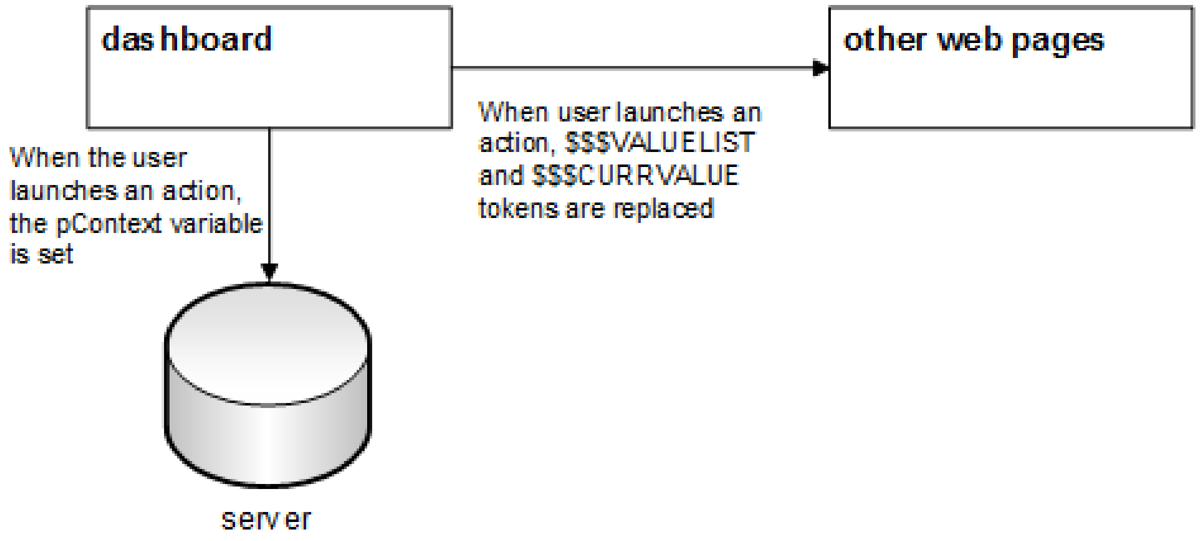
Operation	Available As Standard Action?	Can Be Performed in Custom Action?
Setting a filter	*Yes*	*Yes*

Operation	Available As Standard Action?	Can Be Performed in Custom Action?
Setting a filter and refreshing the display	*Yes*	*Yes*
Refreshing the display of a widget	*Yes*	*Yes*
Refreshing the display of the entire dashboard	*Yes*	No
Specifying the row or column sort for a pivot table	*Yes*	No
Specifying the row or column count for a pivot table	*Yes*	No
Displaying a listing for a pivot table	*Yes*	No
Displaying another dashboard	*Yes*	*Yes*
Displaying a URL in the same page	*Yes*	*Yes*
Displaying a URL in a new page	No	*Yes*
Executing code on the server	No	*Yes*
Changing the data source of the widget	*Yes*	No
Changing the row or column specification of the widget	*Yes*	No

For details on the standard actions, see “Adding Widget Controls” in *Creating Dashboards*.

6.1.1 Context Information

The system makes context information available to actions, by two different mechanisms. When a user launches a custom action, the system writes context information into the `pContext` variable, which is available in your custom code on the server. When a custom action opens a URL, the system replaces the `$$$VALUELIST` and `$$$CURRVALUE` tokens, if these are included in the URL. The following figure illustrates these mechanisms:


dashboard
other web pages
When user launches an action, $$$VALUELIST and $$$CURRVALUE tokens are replaced
When the user launches an action, the pContext variable is set
server

6.2 Defining the Behavior of Actions

To define custom actions, you must both declare the actions and define their behavior.

6.2.1 Declaring Actions

To declare actions, do either or both of the following tasks in a KPI class:

- Within the `<kpi>` element, include one `<action>` element for each action.

  This element specifies the name of an action available within this KPI class; the user interfaces use this information to create lists of available actions for the users. For example:

  ```
  <kpi xmlns="http://www.intersystems.com/deepsee/kpi"
   name="Holefoods Actions">

  <action name="ActionA"/>
  <action name="ActionB"/>
  <action name="ActionC"/>
  </kpi>
  ```

  For information on `<action>`, see the appendix "Reference Information for KPI and Plug-in Classes" in *Advanced Modeling for InterSystems Business Intelligence*.

- Override the **%OnGetActionList()** callback method of your KPI class. This method has the following signature:

  ```
  ClassMethod %OnGetActionList(ByRef pActions As %List, pDataSourceName As %String = "") As %Status
  ```

  Where *pActions* is an array with the following nodes:

Node	Value
pActions	Number of actions
pActions(*n*)	Details for the *n*th action. This is a $LISTBUILD list that consists of the following items: – A string that equals the logical action name – A string that equals the corresponding display name

  And *pDataSourceName* is for future use.

  For example:

  ```
  ClassMethod %OnGetActionList(ByRef pActions As %List, pDataSourceName As %String = "") As %Status
  {
      set newaction=$LB("New Action","New Action Display Name")
      set pActions($I(pFilters))=newaction
      quit $$$OK
  }
  ```

6.2.2 Defining the Behavior of the Actions

To define the behavior of the actions, override the **%OnDashboardAction()** callback method of your KPI class. This method has the following signature:

```
classmethod %OnDashboardAction(pAction As %String, pContext As %ZEN.proxyObject) as %Status
```

The system executes this callback when a user invokes an action on a dashboard. *pAction* is the logical name of the action. *pContext* is an object that contains information about the currently selected scorecard row and that provides a way for the method to return commands to the dashboard; the next sections give the details.

A simple example is as follows:

```
ClassMethod %OnDashboardAction(pAction As %String, pContext As %ZEN.proxyObject) As %Status
{
 Set sc = $$$OK
 Try {
      If (pAction = "Action 1") {
              //this part defines Action 1
              //perform server-side actions
         }
         Elseif (pAction="Action 2")
         {
                //this part defines Action 2
                //perform other server-side actions
                 }
        }
        Catch(ex) {
                Set sc = ex.AsStatus()
        }
 Quit sc
}
```

This method defines two actions, `Action 1` and `Action 2`.

Note: Because **%OnDashboardAction()** is a class method, you do not have access to %seriesNames or other properties of the KPI class from within this method (no class instance is available from the method).

6.3 Available Context Information

An action can use context information — values from the dashboard, based on the row or rows that the user selected before launching the action. These values are useful if you want to cause changes in the database that are dependent on context.

Because **%OnDashboardAction()** is a class method, you do not have access to %seriesNames or other properties of the KPI class from within this method. Instead, the system provides the *pContext* variable, which is an object whose properties provide information for use in the action. The details are different in the following scenarios:

- Pivot table that uses a pivot table as the data source
- Pivot table that uses a listing as the data source
- Pivot table that uses a KPI as the data source
- Scorecard that uses a KPI as the data source

6.3.1 Scenario: Pivot Table Widget with Pivot Table as Data Source

In this scenario, within the **%OnDashboardAction()** method, the *pContext* variable has the following properties:

Property Name	Property Contents
`currValue`	Value of first selected cell
`currSeriesNo`	Column number
`currItemNo`	Row number
`currFilterSpec`	MDX %FILTER clause or clauses that represent the filtering applied to the current cell context. This includes values of any filter controls, as well as the row and column context.
`valueList`	*Null*
`cubeName`	Name of the cube queried by this pivot table
`mdx`	MDX query defined by this pivot table
`pivotVariables`	A proxy object that contains one property for each pivot variable. Specifically, *pContext.pivotVariables.varname* contains the value of the pivot variable *varname*. In this proxy object, all pivot variable names are in lowercase. For example, if the server defines a pivot variable named `MyVar`, this pivot variable is available as `pContext.pivotVariables.myvar`
`filters`	An array that indicates the current values of all filter controls. For each node in this array, the subscript is the name of a filter defined by the KPI. The value of the node is the corresponding key or keys, in the form described at "Allowed Default Values for Filters".
`dataSource`	Name of the current data source

6.3.2 Scenario: Pivot Table Widget with Listing as Data Source

In this scenario, within the **%OnDashboardAction()** method, the *pContext* variable has the following properties:

Property Name	Property Contents
`currValue`	Value of the first selected cell that was displayed before the listing was shown
`currSeriesNo`	Column number of the first selected cell that was displayed before the listing was shown
`valueList`	Comma-separated list of values from the first column of the listing (these values must not *contain* commas)
`pivotVariables`	A proxy object that contains one property for each pivot variable. Specifically, *pContext.pivotVariables.varname* contains the value of the pivot variable *varname*. In this proxy object, all pivot variable names are in lowercase. For example, if the server defines a pivot variable named `MyVar`, this pivot variable is available as `pContext.pivotVariables.myvar`
`filters`	An array that indicates the current values of all filter controls. For each node in this array, the subscript is the name of a filter defined by the KPI. The value of the node is the corresponding key or keys, in the form described at "Allowed Default Values for Filters".
`dataSource`	Name of the current data source

6.3.3 Scenario: Pivot Table Widget with KPI as Data Source

In this scenario, within the **%OnDashboardAction()** method, the *pContext* variable has the following properties:

Property Name	Property Contents
`currValue`	Value of first selected cell
`currSeriesNo`	Column number of first selected cell
`valueList`	*Null*
`pivotVariables`	A proxy object that contains one property for each pivot variable. Specifically, *pContext.pivotVariables.varname* contains the value of the pivot variable *varname*. In this proxy object, all pivot variable names are in lowercase. For example, if the server defines a pivot variable named `MyVar`, this pivot variable is available as `pContext.pivotVariables.myvar`
`filters`	An array that indicates the current values of all filter controls. For each node in this array, the subscript is the name of a filter defined by the KPI. The value of the node is the corresponding key or keys, in the form described at "Allowed Default Values for Filters".
`dataSource`	Name of the current data source

6.3.4 Scenario: Scorecard with KPI as Data Source

In this scenario, within the **%OnDashboardAction()** method, the *pContext* variable has the following properties:

Property Name	Property Contents
`currValue`	Value of the KPI property that is marked as **Value Column** in this scorecard
`currSeriesNo`	Row number
`valueList`	Value of the KPI property that is marked as **Value Column** in this scorecard
`pivotVariables`	A proxy object that contains one property for each pivot variable. Specifically, *pContext.pivotVariables.varname* contains the value of the pivot variable *varname*. In this proxy object, all pivot variable names are in lowercase. For example, if the server defines a pivot variable named `MyVar`, this pivot variable is available as `pContext.pivotVariables.myvar`
`filters`	An array that indicates the current values of all filter controls. For each node in this array, the subscript is the name of a filter defined by the KPI. The value of the node is the corresponding key or keys, in the form described at "Allowed Default Values for Filters".
`dataSource`	Name of the current data source

6.4 Executing Client-Side Commands

An action can contain commands to execute when the control returns to the dashboard. To include such commands, set the *pContext*`.command` property within the definition of the action. For example:

```
//this part defines Action 1
//perform server-side actions
//on returning, refresh the widget that is using this KPI
Set pContext.command="refresh;"
```

For *pContext*`.command`, specify a string of the following form to execute a single command:

```
command1
```

For *pContext*`.command`, specify a semicolon-delimited list of commands:

```
command1;command2;command3;...;
```

The final semicolon is optional.

Some commands accept one or more arguments. For these, use `command:arg1:arg2:...` instead of `command`.

6.4.1 Available Commands

Within *pContext*`.command`, you can use the following commands:

alert

```
alert:message
```

Displays the message in a dialog box. *message* is the message to display

For example:

```
 Set pContext.command = "alert:hello world!"
```

applyFilter

```
applyFilter:target:filterSpec
```

For information on the arguments, see "Details for applyFilter and setFilter."

This command sets the given filter and refreshes the browser window.

For example, the following applies a filter to a pivot table:

```
 Set pContext.command = "applyFilter:samplepivot:[DateOfSale].[Actual].[YearSold]:&[2008]"
```

navigate

```
navigate:url
```

Where *url* is the URL to display.

This command opens the given URL in the same browser window.

For example:

```
 Set pContext.command = "navigate:http://www.google.com"
```

newWindow

```
newWindow:url
```

Where *url* is the URL to display.

This command opens the given URL in a new browser window.

For example:

```
 Set pContext.command = "newWindow:http://www.google.com"
```

popup

```
popup:popupurl
```

Where *popupurl* is the relative URL of a popup window.

This command displays the given popup window. For example:

```
Set pContext.command = "popup:%ZEN.Dialog.fileSelect.cls"
```

refresh

```
refresh:widgetname
```

Where *widgetname* is the optional name of a widget to refresh; if you omit this argument, the command refreshes the widget from which the user launched the action.

This refreshes the browser window, using any current settings for filters.

For example:

```
// Refresh the widget that fired this action and another named samplepivot.
Set pContext.command = "refresh;refresh:samplepivot"
```

Note that this example includes multiple commands, separated by a semicolon.

setFilter

```
setFilter:target:filterSpec
```

For information on the arguments, see "Details for applyFilter and setFilter."

This command sets the given filter, but does not refresh the browser window.

6.4.2 Details for applyFilter and setFilter

The applyFilter and setFilter commands are as follows, respectively:

```
applyFilter:target:filterSpec
```

And:

```
setFilter:target:filterSpec
```

Where:

- *target* is the widget to filter. You can use an asterisk (`*`) to apply the filter to all widgets.
- *filterSpec* specifies the filter value or values to apply to the given target. This must have the following form:

  ```
  filter_name:filter_values
  ```

  Where the arguments depend upon the details of the target widget as follows:

Scenario	*filter_name*	*filter_values*
Target widget displays a pivot table	`[dimension].[hierarchy].[level]`	See "Allowed Default Values for Filters" in the chapter "Configuring Settings."
Target widget displays a KPI	Name of a filter defined in that KPI	One of the allowed values for this filter, as defined in the KPI

Notes:

- You can use multiple filter specifications; to do so, separate them with a tilde (~). For example:

  ```
  FILTER:filterspec1~filterspec2
  ```

- The filter name and filter values are not case-sensitive for pivot tables or for KPIs that use MDX queries.
- The filter can affect only widgets that have been configured with a filter control (possibly hidden) that uses the same filter. For example, suppose that a widget includes a Cities filter control, and has no other filter controls. If the action filters to a city, the widget is updated. If the action filters to a ZIP code, the widget is not updated.

6.5 Displaying a Different Dashboard

In your custom action, you can use `navigate` or `newWindow` to display a different dashboard. Use the dashboard URL as described in the chapter "Accessing Dashboards from Your Application." The URL can include the SETTINGS keyword, which initializes the state of the dashboard.

6.6 Generating a SQL Table from Cube Context

In your custom action, you can use the **%CreateTable** API to create a SQL table from cube context. The table may be created from either:

1. A field list
2. The name of a listing defined in the cube, either as a field list or a custom SQL query.

See the class reference for more details.

7
Accessing Dashboards from Your Application

This chapter describes how to access InterSystems IRIS Business Intelligence dashboards and the Business Intelligence User Portal from your application. It discusses the following topics:

- How to access a dashboard from your application
- Parameters you can use in a dashboard URL
- Available options for the SETTINGS parameter
- Accessing other Business Intelligence pages from your application

Also see "Accessing the Samples Shown in This Book," in the first chapter.

7.1 Accessing a Dashboard

To access a dashboard, use a URL of the following form:

```
http://localhost:52773/csp/samples/_DeepSee.UserPortal.DashboardViewer.zen?DASHBOARD=dashbdname.dashboard
```

Where *localhost:52773* is the server and port on which InterSystems IRIS® is running, *samples* is the namespace in which the dashboard is defined, and *dashbdname* is the name of the dashboard, including the folder to which it belongs, if any.

More generally, use a URL of the following form:

```
http://localhost:52773/csp/samples/_DeepSee.UserPortal.DashboardViewer.zen?parmstring&parmstring&parmstring...
```

Where *parmstring* is a parameter, followed by an equals sign, followed by a value. For example:

```
DASHBOARD=Drill%20Options.dashboard
```

As shown previously, use an ampersand (&) to combine multiple parameter strings. For example:

```
DASHBOARD=Drill%20Options.dashboard&NOMODIFY=1
```

7.1.1 URL Encoding

Certain characters have reserved meanings in a URL and others are disallowed. To include such a character in *parmstring*, replace the character with the URL-encoded version (also called percent-encoded). The easiest way to do this is as follows:

1. Identify all the strings that could potentially include reserved or disallowed characters.
2. For each such string, do the following in sequence:
   a. Convert to UTF-8 encoding
   b. Create a URL-encoded version of the string.

   If you are performing these transformations on the server, you can use an ObjectScript function such as $ZCONVERT or $TRANSLATE. For example:

   ```
   set UTF8db=$ZCONVERT(dashboardname,"O","UTF8")
   set escapeddb=$ZCONVERT(UTF8db,"O","URL")
   set url=baseurl_"DASHBOARD="_escapeddb
   ```

   If you are performing these transformations on the client, use a suitable client function. For example, if the client uses JavaScript, use the encodeURI() function.

Or use other logic such as the $TRANSLATE function. Some of the most commonly used characters are these:

Character	URL-Encoded Version
space character	%20
&	%26
,	%2C

You can find lists of URL-encoded characters on the Internet; one resource is Wikipedia (http://en.wikipedia.org/wiki/Percent-encoding).

7.2 Available URL Parameters

You can use the following case-sensitive parameters within the dashboard URL. Note that for some parameters, you can use either a plain-text version or an encrypted version. For example, the dashboard URL can include an encrypted version of the dashboard name.

DASHBOARD

```
DASHBOARD=dashbdname.dashboard
```

This parameter specifies the dashboard to display. You must specify either this parameter or the XDASHBOARD parameter.

dashbdname is the name of the dashboard, including the folder to which it belongs, if any. For example:

```
DASHBOARD=Dashboards/Dashboard%20with%20Filters%20and%20Listing%20Button.dashboard
```

Here `%20` represents a space character; see "URL Encoding," earlier in this chapter.

XDASHBOARD

```
XDASHBOARD=encryptedvalue
```

Encrypted version of the DASHBOARD parameter. You can use parameter only within the context of a web session. You must specify either this parameter or the DASHBOARD parameter.

To create *encryptedvalue*, start with the name of the dashboard, including the folder to which it belongs, if any. For example:

```
Dashboards/Dashboard with Filters and Listing Button.dashboard
```

Do not include URL escaping; for example, leave a space as a space character.

Then use the **Encrypt()** class method of %CSP.Page to encrypt this value. Use the value returned by **Encrypt()** as the value of the XDASHBOARD parameter.

EMBED

```
EMBED=1
```

If this parameter is 1, the dashboard is displayed in embedded mode. This is equivalent to setting NOTITLE=1, NOMODIFY=1, NOBORDER=1, and WORKLISTS=0.

XEMBED

```
XEMBED=encryptedvalue
```

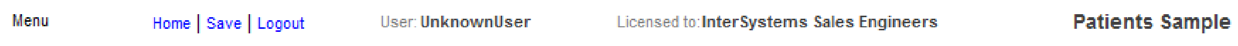
Encrypted version of the EMBED parameter. You can use parameter only within the context of a web session.

To create *encryptedvalue*, start with the value you would use for EMBED. Then use the **Encrypt()** class method of %CSP.Page to encrypt this value. Use the value returned by **Encrypt()** as the value of the XEMBED parameter.

NOTITLE

```
NOTITLE=1
```

If this parameter is 1, the dashboard is displayed without a title area. The title area is the top area, as in the following example:



NOMODIFY

```
NOMODIFY=1
```

If this parameter is 1, the dashboard cannot be modified. This option removes items from **Menu**. It also suppresses the edit options on widgets, so that a widget includes only minimize, maximize, and remove options in the upper right.

NOBORDER

```
NOBORDER=1
```

If this parameter is 1, the dashboard is displayed without the border.

RESIZE

```
RESIZE=boolean
```

Specifies whether the widgets can be resized and moved. If *boolean* is 1 (the default), the widgets can be resized and moved. If *boolean* is 0, they cannot.

WORKLISTS

```
WORKLISTS=n
```

Where *n* is 0, 1, or 2. This parameter specifies the number of worklist areas to display on the left.

XWORKLISTS

```
XWORKLISTS=encryptedvalue
```

Encrypted version of the WORKLISTS parameter. You can use parameter only within the context of a web session.

To create *encryptedvalue*, start with the value you would use for WORKLISTS. Then use the **Encrypt()** class method of %CSP.Page to encrypt this value. Use the value returned by **Encrypt()** as the value of the XWORKLISTS parameter.

SCHEME

```
SCHEME=schemename
```

Specifies the color scheme for the dashboard (if you do not want to use the default). For *schemename*, specify a scheme as listed in the **General** tab of the **Settings** page. See "Specifying Basic Settings," earlier in this book.

SETTINGS

```
SETTINGS=name1:value1;name2:value2;name3:value3;...;
```

Where *name1*, *name2*, *name3*, and so on are names of dashboard settings, as described in the next section, and *value1*, *value2*, *value3*, and so on are the values for the settings.

You can include this parameter multiple times in the URL.

For example, to pass values to all widgets in a dashboard, use a URL of the following form:

```
basic_dashboard_url&SETTINGS=name:value;name:value;name:value;...;
```

To pass values to a specific widget in a dashboard, use the following variation:

```
basic_dashboard_url&SETTINGS=TARGET:widgetname;name:value;name:value;name:value;...;
```

To pass values to multiple widgets in a dashboard, use the following variation:

```
basic_dashboard_url&SETTINGS=...;&SETTINGS=...;&SETTINGS=...;...;
```

A setting for a specific widget always takes precedence over settings for all widgets. Otherwise, the settings are applied in the order in which they are specified; if one setting is inconsistent with another setting, the later setting takes effect. These settings do not take precedence over any user settings.

XSETTINGS

```
XSETTINGS=encryptedvalue
```

Encrypted version of the SETTINGS parameter. You can use parameter only within the context of a web session.

To create *encryptedvalue*, start with the value that you would use with SETTINGS. Then use the **Encrypt()** class method of %CSP.Page to encrypt this value. Use the value returned by **Encrypt()** as the value of the XSETTINGS parameter.

IRISUsername and IRISPassword

```
IRISUsername=myuser&IRISPassword=mypass
```

Where *myuser* is an InterSystems IRIS username and *mypass* is the corresponding password. Include these parameters if the user has not yet logged in to InterSystems IRIS.

AUTOSAVE

```
AUTOSAVE
```

Requests the autosaved version of the dashboard. For information on the autosave feature, see "Specifying Basic Settings," earlier in this book.

7.3 Options for the SETTINGS Parameter

For the SETTINGS URL parameter, you can use settings given in the following list. Any SETTINGS string either applies to all widgets or applies to a specific widget. Include as many SETTINGS strings as you need. For example:

```
basic_dashboard_url&SETTINGS=...;&SETTINGS=...;&SETTINGS=...;...;
```

Note: When InterSystems IRIS parses a SETTINGS parameter, it assumes that any semicolon is a delimiter between two different settings strings. To include a semicolon and not have it treated as a delimiter, you must replace it with `%3B%3B` (this sequence is two URL-encoded semicolons; it is necessary to use *two* URL-encoded semicolons because of how the parsing is performed).

TARGET

```
TARGET:widgetname
```

Specifies the widget to which this set of settings applies. If you want the settings to apply to all widgets, omit this parameter from the SETTINGS string.

FILTER

```
FILTER:filter_name.filter_values
```

Specifies how to filter the given widgets. The arguments depend upon the details of the target widget as follows:

Scenario	*filter_name*	*filter_values*
Target widget displays a pivot table	URL-encoded version of `[dimension].[hierarchy].[level]`	URL-encoded version of the allowed filter values that are shown in "Allowed Default Values for Filters" in the chapter "Configuring Settings"
Target widget displays a KPI	URL-encoded version of the name of a filter defined in that KPI	URL-encoded version of an allowed value for this filter, as defined in the KPI

For information on creating URL-encoded strings, see "URL Encoding," earlier in this chapter.

Notes:

- You can use the special token `$$$FILTERS` in place of *filter_name.filter_value*. This is useful if you use the URL in a custom navigate action (which accesses another dashboard from a given dashboard; see "Displaying a Different Dashboard," elsewhere in this book). In this scenario, `$$$FILTERS` is replaced with the current filter values of the original dashboard. For example:

  ```
  FILTER:$$$FILTERS
  ```

  The target dashboard should include the same filters.

- You can use multiple filter specifications; to do so, separate them with a tilde (~). For example:

  ```
  FILTER:filterspec1~filterspec2
  ```

  Where each *filterspec* is *filter_name*`.`*filter_values*

- Each target widget must also be configured to include a filter control (possibly hidden) that uses the same filter. For example, suppose that a widget includes a Cities filter control, and has no other filter controls. You can include `&SETTINGS=FILTER:[Cities].%26[Centerville]` in the dashboard URL and that will filter this widget. If you include `&SETTINGS=FILTER:[ZIP].%26[34577]`, that does not affect the widget.

- To use multiple members of the same filter together, use a set expression that lists those members; see "Allowed Default Values for Filters" in the chapter "Configuring Settings." (If you include the same filter multiple times within the SETTINGS string, the system uses the last value that you provide; this is probably not the behavior that you want.)

VARIABLE

```
VARIABLE:variable_name.variable_value
```

Specifies the value of the given pivot variable. For information on pivot variables, see "Defining and Using Pivot Variables" in *Using the Analyzer*.

You can use the special token `$$$VARIABLES` in place of *variable_name.variable_value*. This is useful if you use the URL in a custom navigate action (which accesses another dashboard from a given dashboard; see "Displaying a Different Dashboard," elsewhere in this book). In this scenario, `$$$VARIABLES` is replaced with the current values of the given pivot variables, as specified in the original dashboard. For example:

```
VARIABLE:$$$VARIABLES
```

ROWCOUNT

```
ROWCOUNT:n
```

Specifies the maximum number (*n*) of rows to display; this applies only when members are displayed as rows.

COLCOUNT

```
COLCOUNT:n
```

Specifies the maximum number (*n*) of columns to display; this applies only when members are displayed as columns.

ROWSORT

```
ROWSORT:measure
```

Specifies the measure by which to sort the rows. Here *measure* is the MDX identifier for the measure. For example:

```
ROWSORT:[MEASURES].[mymeasure]
```

Note that you cannot omit the square brackets of these identifiers (in contrast to other uses of MDX in Business Intelligence).

COLSORT

```
COLSORT:[MEASURES].[my measure]
```

Specifies the measure by which to sort the columns. Here *measure* is the MDX identifier for the measure; see ROWSORT.

ROWSORTDIR

```
ROWSORTDIR:sortkeyword
```

Specifies how to sort the rows. Here *sortkeyword* is one of the following:

- `ASC` — Sort in ascending order but preserve any hierarchies.
- `DESC` — Sort in descending order but preserve any hierarchies.
- `BASC` — Sort in ascending order and break any hierarchies.
- `BDESC` — Sort in descending order and break any hierarchies.

COLSORTDIR

```
COLSORTDIR:sortkeyword
```

Specifies how to sort the columns. See ROWSORTDIR.

PORTLET

```
PORTLET:portlet_setting.value
```

Specifies the value for a portlet setting, to override any configured value for that setting. As with the other SETTINGS options, this setting is applied to all widgets listed by the TARGET parameter (or all portlet widgets if TARGET is not specified).

Here *portlet_setting* must be the name of the setting as defined in the portlet, and *value* must be the URL-encoded version of an allowed value for this setting. For information on creating URL-encoded strings, see “URL Encoding,” earlier in this chapter.

You can use multiple portlet specifications; to do so, separate them with a tilde (~). For example:

```
PORTLET:portletspec1~portletspec2
```

Where each *portletspec* is *portlet_setting*`.`*value*

For information on defining portlets, see the chapter “Creating Portlets for Use on Dashboards.”

To see an example, display the dashboard Widget Examples/Custom Portlet, which displays a round clock, and then add the following to the end of the URL in the browser:

```
&SETTINGS=PORTLET:CIRCLE.0~SIZE.200
```

Then press **Enter**. You should see the clock change into a square, slightly larger than it had previously been.

For example, the following limits the column count to 3 for most widgets but limits the column count to 1 for the widget `RegionVsYear`.

```
&SETTINGS=TARGET:RegionVsYear;COLCOUNT:1;&SETTINGS=COLCOUNT:3;
```

Note: These settings are not supported for custom widgets or custom controls.

7.4 Accessing Other Business Intelligence Pages from Your Application

Your application can also provide direct links to other Business Intelligence web pages, such as the Analyzer and User Portal.

The URLs for the Business Intelligence web pages have the following general structure.

```
http://localhost:52773/csp/samples/_Package.Class
```

Where *localhost:52773* is the server and port on which InterSystems IRIS is running, *samples* is the namespace in which you are running Business Intelligence, and *_Package.Class* is the name of the package and class that defines the page, with an underscore instead of a percent sign at the start of the package name. When you access the Analyzer or other Business Intelligence pages, this URL is shown in the toolbar or your browser.

You can use any of the applicable URL parameters with these pages; see "Available URL Parameters," earlier in this chapter. When you use the URL for the Analyzer, you can also specify the PIVOT URL parameter, which indicates the pivot table to display. For example:

```
http://localhost:52773/csp/samples/_DeepSee.UI.Analyzer.zen?PIVOT=Pivot%20Features%2FConditional%20Formatting.pivot
```

Note that if you use the URL for the Analyzer, and you specify the AUTOSAVE URL parameter but not the PIVOT parameter, the Analyzer displays the most recently viewed item.

8
Keeping the Cubes Current

This chapter discusses how to use the Cube Manager and other tools for updating the cubes. It discusses the following topics:

- Overview of the options
- How cube synchronization works
- How to enable cube synchronization
- How to clear the ^OBJ.DSTIME global
- How to use the Cube Manager
- How to use %SynchronizeCube()
- How to purge the DSTIME index
- How to update cubes manually
- Other options
- Examples that you can execute from a dashboard

Also see the appendix "Using Cube Versions."

Also see "Accessing the Samples Shown in This Book," in the first chapter.

8.1 Overview

The generic phrase *updating a cube* refers to the process of causing a cube to reflect the current contents of the source table and related tables. The system provides three techniques:

- Rebuild the cube, using the **Rebuild** option in the Architect, for example. This process can be time-consuming, and queries cannot be executed while a cube is being rebuilt.
- *Synchronize the cube*. The cube synchronization feature (also known as the DSTIME feature) enables InterSystems IRIS Business Intelligence to keep track of changes to the data. You periodically synchronize the cube to include those changes.

  It *is* possible to execute queries during synchronization.

  Depending on the cube implementation and depending on which data changes, it may not be possible to use this feature; see "When Cube Synchronization Is Possible," later in this chapter.

- *Update the cube manually*. This process uses the %ProcessFact() and %DeleteFact() methods. Unlike with the other options, in this case, it is necessary for your code to know which records of the fact table to update or delete.

  It *is* possible to execute queries during the manual updates.

You can use any suitable combination of these techniques. The following table compares them:

	Rebuilding	**Synchronizing**	**Updating Manually**
Comparative duration of process	long	short	short
Able to execute queries during this process	no	yes	yes
Technique is available in all scenarios	yes	no	yes
Technique requires you to know which records were changed	no	no	yes
Technique invalidates parts of the result cache	yes	yes	yes
User interfaces that provide this option	Cube Manager and Architect	Cube Manager	none

For information on the Cube Manager, see "Using the Cube Manager," later in this chapter.

8.1.1 Cube Updates and Related Cubes

For any kind of update, whenever you have cube-to-cube relationships, it is necessary to update the cubes in a specific order. In particular, update the independent cube first. Then update any cubes that depend on it. To do this, you can use the Cube Manager, which traverses the relationships and determines the correct update order.

Or you can write and use a utility method or routine that builds your cubes in the appropriate order.

8.1.2 Cube Updates and the Result Cache

For any cube that uses more than 512,000 records (by default), the system maintains and uses a result cache. For any combination of update techniques and tools, you should also carefully consider the *frequency* of cube updates, because any update could invalidate parts of the result cache.

For large data sets, the system maintains and uses a result cache for each cube as follows: Each time a user executes a query (via the Analyzer for example), the system caches the results for that query. The next time any user runs that query, the system checks to see if the cache is still valid. If so, the system then uses the cached values. Otherwise, the system re-executes the query, uses the new values, and caches the new values. The net effect is that performance improves over time as more users run more queries.

When you update a cube in any way, parts of the result cache are considered invalid and are cleared. The details depend upon options in the cube definition (see "Cache Buckets and Fact Order," earlier in this book). Therefore, it is not generally desirable to update constantly.

8.2 How Cube Synchronization Works

This section describes briefly how cube synchronization works. Internally, this feature uses two globals: *^OBJ.DSTIME* and *^DeepSee.Update*.

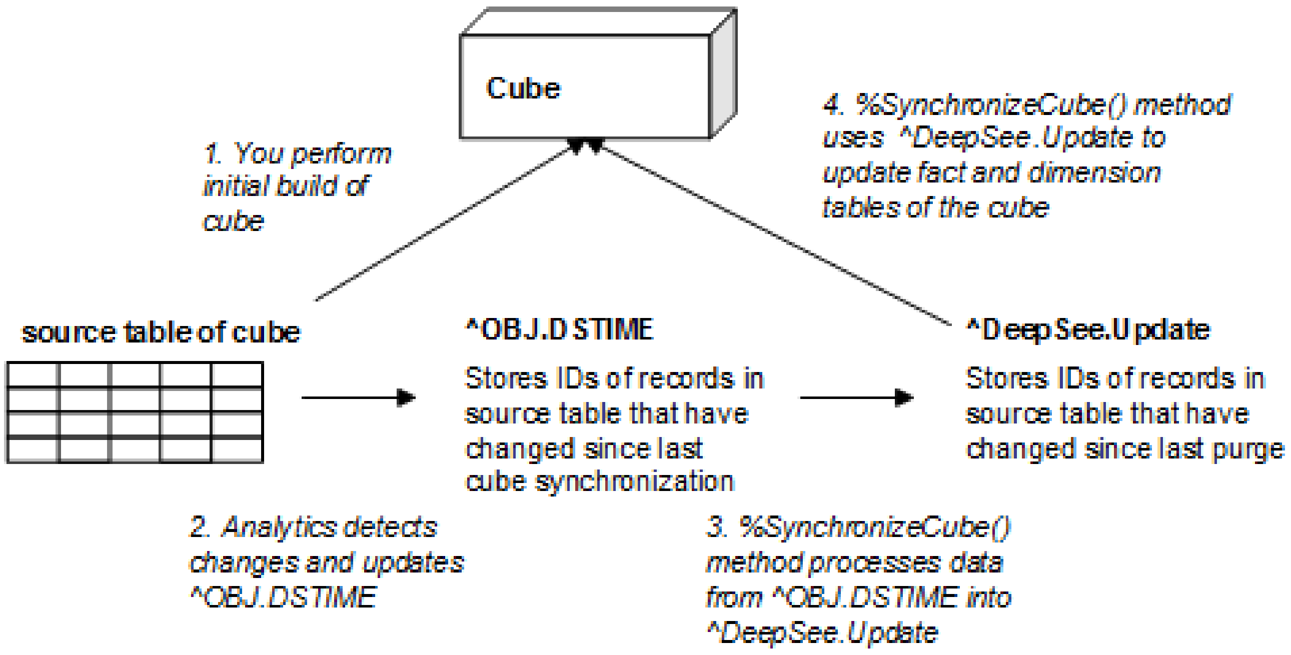
First, it is necessary to perform an initial build of the cube.

When InterSystems IRIS® detects a change within the source table used by a cube, it adds entries to the *^OBJ.DSTIME* global. These entries are to indicate which IDs have been added, changed, or deleted.

When you synchronize the cube (via **%SynchronizeCube()**, described later in this chapter), InterSystems IRIS first reads the *^OBJ.DSTIME* global and uses it to update the *^DeepSee.Update* global. After it adds an ID to the *^DeepSee.Update* global, InterSystems IRIS removes the same ID from the *^OBJ.DSTIME* global. (Note that in previous versions, the cube synchronization feature used only one global; the newer system prevents a race condition.)

Then InterSystems IRIS uses the *^DeepSee.Update* global and updates the fact and dimension tables of the cube, thus bringing the cube up to date.

The following figure shows the overall flow:


Cube
1. You perform initial build of cube
4. %SynchronizeCube() method uses ^DeepSee.Update to update fact and dimension tables of the cube
source table of cube
^OBJ.DSTIME
Stores IDs of records in source table that have changed since last cube synchronization
^DeepSee.Update
Stores IDs of records in source table that have changed since last purge
2. Analytics detects changes and updates ^OBJ.DSTIME
3. %SynchronizeCube() method processes data from ^OBJ.DSTIME into ^DeepSee.Update

The subsections discuss the following details:

- When the cube synchronization feature can be used
- When the cube synchronization feature cannot be used
- Cube synchronization in a mirrored environment
- Structure of the cube synchronization globals

8.2.1 When Cube Synchronization Is Possible

You can use the cube synchronization feature in scenarios where all the following items are true:

- The base class for the cube is a persistent class (but is not a linked table).
- The changed record is a record in that class.

8.2.2 When Cube Synchronization Is Not Possible

You cannot use the cube synchronization feature in the following scenarios:

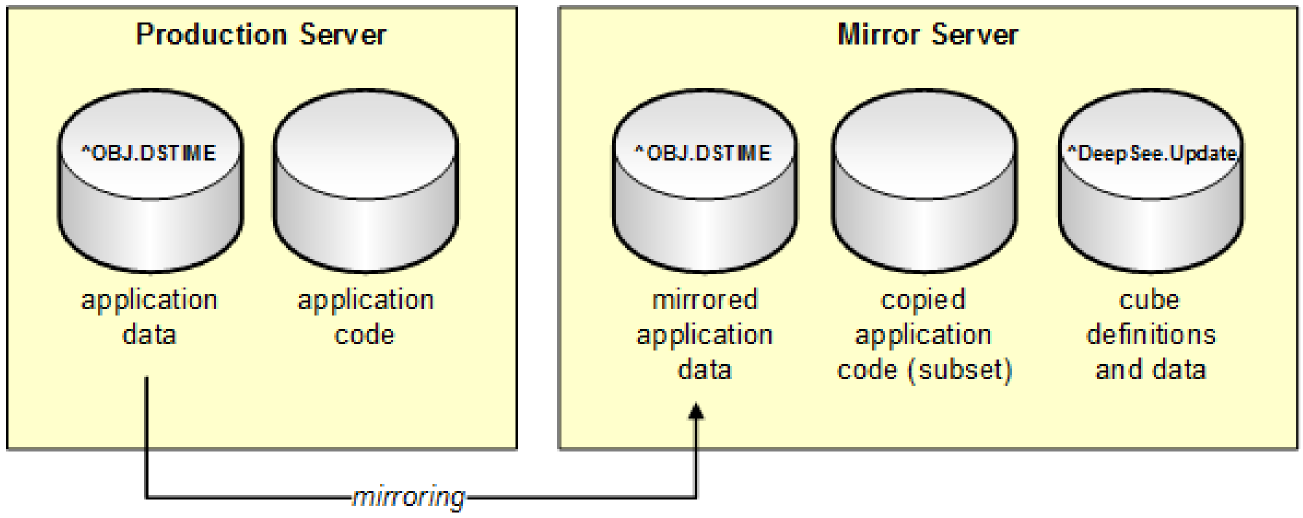
- The base class for the cube is a data connector. (See the chapter "Defining Data Connectors.")
- The base class for the cube is a linked table. (See "The Link Table Wizard" in *Using the InterSystems IRIS SQL Gateway*).
- The changed record is not in the extent of the base class used by the cube. That is, the changed record belongs to another table.

In these scenarios, the cube synchronization feature cannot detect the change, and your application must update the cube manually as described in "Updating Cubes Manually."

Also, cube synchronization does not affect age dimensions (that is, dimensions whose **Dimension type** is **age**).

8.2.3 Cube Synchronization in a Mirrored Environment

If you use Business Intelligence on a mirror server, note that the *^OBJ.DSTIME* global is part of the application data and should be mirrored (if it mapped to a different database, for example, that database should be mirrored). The *^DeepSee.Update* global is generated by Business Intelligence code and thus is present only in the database that contains the cube definitions and data.



Important: On the mirror server, the databases that store the *^OBJ.DSTIME* and *^DeepSee.Update* globals must be read/write. Note that you can store both of these globals in the same database, although the above figure shows them in separate databases.

For a discussion of using Business Intelligence on a mirror server, see "Recommended Architecture" in the first chapter of this book.

8.2.4 Structure of the Cube Synchronization Globals

This section describes the structure of the cube synchronization globals. You do not need this information to use cube synchronization; this information is provided in case you wish to use these globals for other purposes.

8.2.4.1 ^OBJ.DSTIME

The *^OBJ.DSTIME* global has a different form depending on whether *DSINTERVAL* is set.

If *DSINTERVAL* is not set, this global has nodes like the following:

Node	Value
`^OBJ.DSTIME(`*class*`,`*increment*`,`*ID*`)` where *class* is the full package and class name of the source class, *increment* is 0, and *ID* is the ID of the new, changed, or deleted record in the given class	One of the following values: • 0 (which means that the record was changed) • 1 (which means that the record was added) • 2 (which means that the record was deleted)

Note that it is possible to manually delete a fact from a fact table without deleting the corresponding record from the source class by using the **%SetDSTimeIndex()** method.

If *DSINTERVAL* is set, this global has nodes like the following:

Node	Value
`^OBJ.DSTIME(`*class*`,`*timestamp*`,`*ID*`)` where *class* and *ID* are the same as in the other scenario, and *timestamp* is the number of seconds since midnight on December 31st, 1840	Same as in the other scenario

The system removes unneeded entries from the *^OBJ.DSTIME* global when you synchronize or rebuild a cube.

8.2.4.2 ^DeepSee.Update

The *^DeepSee.Update* global has nodes as follows:

Node	Value
`^DeepSee.Update`	Integer that indicates the next value of *increment* to use
`^DeepSee.Update(`*class*`,`*increment*`,`*ID*`)` where *class* is the full package and class name of the source class, *increment* is 0 or a positive integer, and *ID* is the ID of the new, changed, or deleted record in the given class. Each time you synchronize cubes, the system new nodes to this global, using the next highest integer for *increment*. See the example.	Same as in the *^OBJ.DSTIME* global
`^DeepSee.Update("cubes",`*cube*`,"dstime")` where *cube* is the logical name of a cube	Integer that indicates the next value of *increment* to use when creating nodes in this global to record changes for the given cube.
`^DeepSee.Update("cubes",`*cube*`,"lastDataUpdate")` where *cube* is the logical name of a cube	The date and time (in $H format) when this cube was last synchronized.

Here is an example:

```
^DeepSee.Update=3
^DeepSee.Update("DeepSee.Study.Patient",0,1)=0
^DeepSee.Update("DeepSee.Study.Patient",0,2)=0
^DeepSee.Update("DeepSee.Study.Patient",0,100)=0
^DeepSee.Update("DeepSee.Study.Patient",1,1)=2
^DeepSee.Update("DeepSee.Study.Patient",1,120)=0
^DeepSee.Update("DeepSee.Study.Patient",2,42)=0
^DeepSee.Update("DeepSee.Study.Patient",2,43)=0
^DeepSee.Update("DeepSee.Study.Patient",2,50)=0
^DeepSee.Update("DeepSee.Study.Patient",2,57)=0
^DeepSee.Update("cubes","PATIENTS","dstime")=3
^DeepSee.Update("cubes","PATIENTS","lastDataUpdate")="64211,63222.68"
```

The nodes under `^DeepSee.Update("DeepSee.Study.Patient",0)` represent the first set of changes, the nodes under `^DeepSee.Update("DeepSee.Study.Patient",1` represent the second set of changes, and so on.

InterSystems IRIS does not automatically remove nodes from *^DeepSee.Update* global. For information on purging this global; see "Purging DSTIME."

8.3 Enabling Cube Synchronization

Before you can synchronize a cube, you must enable the cube synchronization feature for that cube. To do so:

1. Make sure that cube synchronization is possible in your scenario. See "When Cube Synchronization Is Possible," earlier in this chapter.
2. Add the *DSTIME* parameter to the *base class used by that cube*, as follows:

   ```
   Parameter DSTIME="AUTO";
   ```

   The parameter *value* is not case-sensitive.
3. Also optionally add the following parameter to the base class:

   ```
   Parameter DSINTERVAL = 5;
   ```

   This parameter primarily affects how entries are stored in the *^OBJ.DSTIME* global; see "Structure of the Cube Synchronization Globals." The form of the *^OBJ.DSTIME* global has no effect on the behavior of the cube synchronization mechanism.
4. Recompile the base class and all cube classes that use it.
5. Rebuild these cubes.

8.4 Clearing the ^OBJ.DSTIME Global

This section describes how to clear the *^OBJ.DSTIME* global. In some cases, you might want to periodically clear the *^OBJ.DSTIME* global. For example, if you are not using cubes in Business Intelligence, you may want to clear the *^OBJ.DSTIME* global to free up space.

You can set up a task in the Task Manager to periodically clear the *^OBJ.DSTIME* global. To do so, create a new task with an **OnTask()** method such as the following:

```
Method OnTask() As %Status
{
  set classname=$ORDER(^OBJ.DSTIME(""))
  while (classname="") {

    //check to see if this classname is contained in ^DeepSee.Cubes("classes")
    set test=$DATA(^DeepSee.Cubes("classes",classname))

    if (test'=1) {
        kill ^OBJ.DSTIME(classname)

    }
    set classname=$ORDER(^OBJ.DSTIME(classname))

  }

  q $$$OK
}
```

This task clears *^OBJ.DSTIME* entries if they aren't being used by Business Intelligence cubes. Use the **Task Schedule Wizard** to schedule the task to run as often as necessary.

8.5 Using the Cube Manager

This section describes how to access and use the Cube Manager, which is designed to help you manage cube updates. You use it to determine how and when to update cubes. It adds tasks that rebuild or synchronize cubes at the scheduled dates and times that you choose. This section discusses the following topics:

- Introduction to the Cube Manager
- Introduction to update plans
- How to access the Cube Manager
- How to modify the registry details
- How to register a cube group
- How to specify an update plan
- How to build all registered cubes
- How to perform an on-demand build
- How to unregister a cube group
- How to see past cube events
- How to restrict access to the Cube Manager UI

Note: The Cube Manager tasks are visible in the Task Manager, which is discussed in "Using the Task Manager" in the *System Administration Guide*. InterSystems recommends that you do not modify these tasks in any way.

8.5.1 Introduction to the Cube Manager

The Cube Manager enables you to define the *cube registry*, which contains information about the cubes in the current namespace. In particular, it contains information about how they are to be built, synchronized, or both.

The cube registry defines a set of cube groups. A *cube group* is a collection of cubes that need to be updated together, either because they are related or because you have chosen to update them together. When you first access the Cube Manager, it displays an initial set of cube groups. Each initial cube group is either a single cube or a set of cubes that are related to each other (and thus must be updated as a group). You can merge these initial cube groups together as wanted. You cannot, however, break up any of the initial cube groups.

Each cube group is initially unregistered, which means that it is not included in the cube registry. After you *register* a cube group (thus placing it into the registry), you define an update plan for it. The Cube Manager creates automatic tasks that use these update plans. See the next section for details.

8.5.2 Introduction to Update Plans

The *update plan* for a cube group specifies how and when the cubes are to be updated. Each group has a default plan, which you can modify. You can also specify different update plans for specific cubes in the group. In both cases, the plan choices are as follows:

- **Build and Synch** — Rebuild periodically, once a week by default. Also synchronize periodically, once a day by default.

  This option is not supported for a given cube unless that cube supports synchronization (as described earlier in this chapter).

- **Build Only** — Rebuild periodically, once a week by default.
- **Synch Only** — Synchronize the cubes periodically, once a day by default.

  This option is not supported for a given cube unless that cube supports synchronization (as described earlier in this chapter).

  Important: Before you synchronize cubes from the Cube Manager, it is necessary to build the cubes at least once from the Cube Manager.

- **Manual** — Do not rebuild or synchronize from the Cube Manager.

  Instead, use any suitable combination of other tools: the **Build** option in the Architect and the **%BuildCube()**, **%SynchronizeCube()**, **%ProcessFact()**, and **%DeleteFact()** methods. For details on the latter three methods, see later sections of this chapter.

For each plan (other than **Manual**), you can customize the schedule details.

For any namespace, the Cube Manager defines two tasks: one performs all requested cube build activity in this namespace, and one performs all requested cube synchronization activity in this namespace. Both of these tasks follow the instructions provided in the cube registry. Both tasks also automatically process cubes in the correct order required by any relationships.

The Cube Manager provides an **Exclude** check box for each registered group and cube, which you can use to exclude that group or cube from any activity by the Cube Manager. Specifically, the Cube Manager tasks ignore any excluded groups and cubes. Initially these check boxes are selected, because it is generally best to not to perform updates until you are ready to do so.

8.5.3 Accessing the Cube Manager

To access the Cube Manager, do the following in the Management Portal:

1. Switch to the appropriate namespace as follows:
   a. Click **Switch**.
   b. Click the namespace.
   c. Click **OK**.
2. Click **Analytics** > **Admin** > **Registry**.

3. If you have not used the Cube Manager in this namespace, it prompts you for information about the cube registry. In this case, specify the following information:

   - **Cube Registry Class Name** — Specify a complete class name, including package. This class definition will be the cube registry for this namespace.
   - **Disable** — Optionally click this to disable the registry. If the registry is disabled, the Cube Manager tasks are suspended. (Because there are no Cube Manager tasks yet, it would be redundant to disable the registry at this point.)
   - **Update Groups** — Specify how to update groups with respect to each other. If you select **Serially**, the tasks update one group at a time. If you select **In Parallel**, the tasks update the groups in parallel.
   - **Allow build to start after this time** — Specify the earliest possible build time.

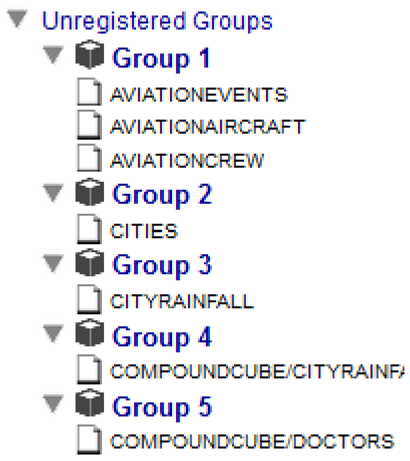
   You can change all these details later, apart from the class name.

   Then click **OK**.

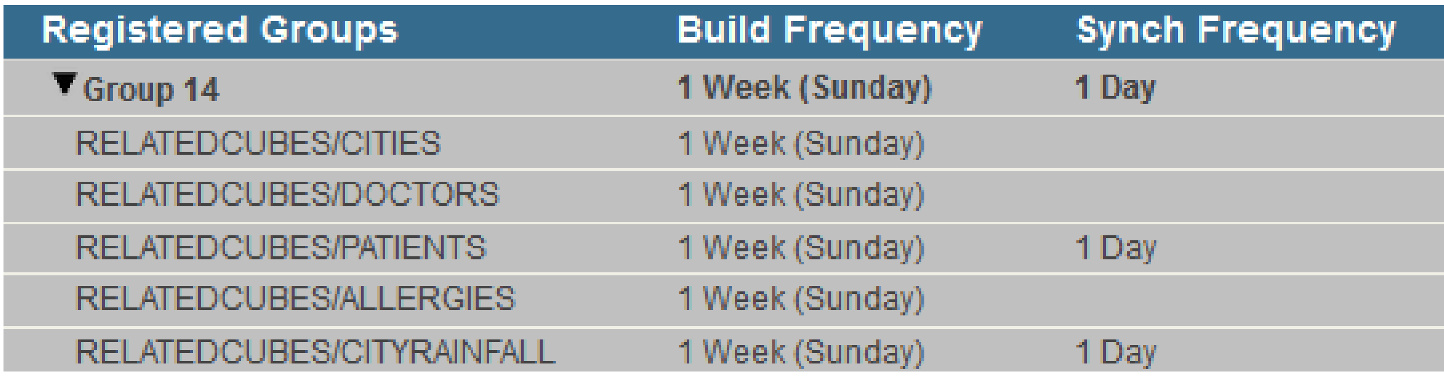
The system displays the Cube Registry page. You can view this page in two modes (via the **View** buttons). Click the left **View** button for tree view or click the right **View** button for table view.

8.5.3.1 Tree View

In tree view, the left area of the Cube Manager displays a tree of unregistered cube groups. For example:



The middle area displays a table (initially empty) with information for the registered groups. The following example shows what this table looks like after you have registered a group:

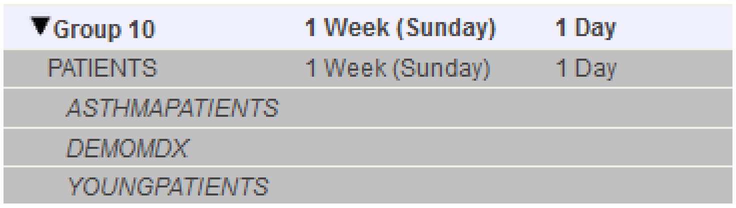


Registered Groups	Build Frequency	Synch Frequency
▼Group 14	1 Week (Sunday)	1 Day
RELATEDCUBES/CITIES	1 Week (Sunday)	
RELATEDCUBES/DOCTORS	1 Week (Sunday)	
RELATEDCUBES/PATIENTS	1 Week (Sunday)	1 Day
RELATEDCUBES/ALLERGIES	1 Week (Sunday)	
RELATEDCUBES/CITYRAINFALL	1 Week (Sunday)	1 Day

This area is color-coded as follows:

- White background — The group or cube is included, which means that the Cube Manager tasks update it. See the **Exclude** option in "Specifying an Update Plan," later in this chapter.
- Gray background — The group or cube is excluded, which means that the Cube Manager tasks ignore it.

This area also lists (in italics) any subject areas based on a given cube, for example:



▼Group 10	1 Week (Sunday)	1 Day
PATIENTS	1 Week (Sunday)	1 Day
ASTHMAPATIENTS		
DEMOMDX		
YOUNGPATIENTS		

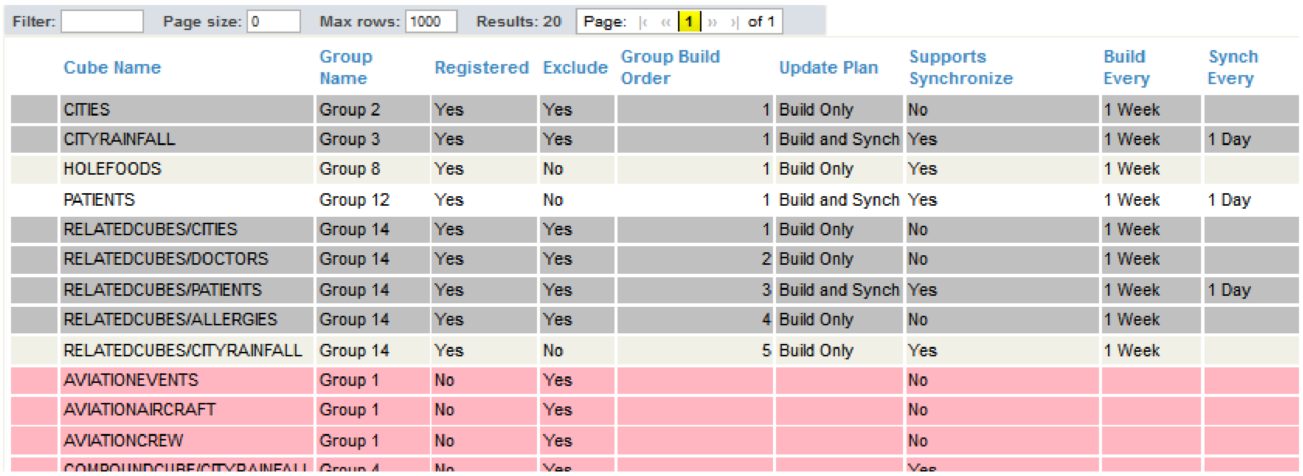
Note that you cannot specify update plans for the subject areas, because updates in a cube are automatically available in any subject area based on that cube. (So there is no need and no way to update a subject area independently from the cube on which it is based.)

In the right area, the **Details** tab (not shown) displays details for the current selection. You can make edits in this tab. The **Tools** tab provides links to other tools.

Note: When the Cube Manager is in tree view, you can expand or collapse the display of all registered groups, which are shown in the middle area. To do so, use the **Expand All** or **Collapse All** button, as applicable, at the top of the middle area. These buttons do not affect the left area of the page, which displays the unregistered groups.

8.5.3.2 Table View

In table view, the Cube Manager lists all cubes in the current namespace, with their update plans. For example:



Filter: Page size: 0 Max rows: 1000 Results: 20 Page: 1 of 1

Cube Name	Group Name	Registered	Exclude	Group Build Order	Update Plan	Supports Synchronize	Build Every	Synch Every
CITIES	Group 2	Yes	Yes	1	Build Only	No	1 Week	
CITYRAINFALL	Group 3	Yes	Yes	1	Build and Synch	Yes	1 Week	1 Day
HOLEFOODS	Group 8	Yes	No	1	Build Only	Yes	1 Week	
PATIENTS	Group 12	Yes	No	1	Build and Synch	Yes	1 Week	1 Day
RELATEDCUBES/CITIES	Group 14	Yes	Yes	1	Build Only	No	1 Week	
RELATEDCUBES/DOCTORS	Group 14	Yes	Yes	2	Build Only	No	1 Week	
RELATEDCUBES/PATIENTS	Group 14	Yes	Yes	3	Build and Synch	Yes	1 Week	1 Day
RELATEDCUBES/ALLERGIES	Group 14	Yes	Yes	4	Build Only	No	1 Week	
RELATEDCUBES/CITYRAINFALL	Group 14	Yes	No	5	Build Only	Yes	1 Week	
AVIATIONEVENTS	Group 1	No	Yes			No		
AVIATIONAIRCRAFT	Group 1	No	Yes			No		
AVIATIONCREW	Group 1	No	Yes			No		
COMPOUNDCUBE/CITYRAINFALL	Group 4	No	Yes			Yes		

This table is color-coded as follows:

- White background — The cube is included, which means that the Cube Manager tasks update it. See the **Exclude** option in "Specifying an Update Plan," later in this chapter.
- Gray background — The cube is excluded, which means that the Cube Manager tasks ignore it.
- Pink background — The cube is not registered and therefore has no update plan.

The **Group Name** field indicates the group to which each cube belongs, and the **Group Build Order** field indicates the order in which each cube is to be built or synchronized within its group. The Cube Manager computes this order only for cubes in registered groups.

In the right area, the **Details** tab (not shown) displays details for the current selection. You can make edits in this tab. The **Tools** tab provides links to other tools.

8.5.4 Modifying the Registry Details

When you first access the Cube Manager, it prompts you for initial information. To modify these details later (other than the registry class name, which cannot be changed):

1. Display the Cube Manager in tree view.
2. In the middle area, click the heading that starts **Registered Groups**.
3. Edit the details on the right.

   For information on the options, see the previous section.
4. Click **Save**.

8.5.5 Registering a Cube Group

To register a cube group:

1. Display the Cube Manager in tree view.
2. Expand the list of unregistered cubes on the left.
3. Drag the group from that area and drop it onto the **Registered Groups** heading in the middle area.

Or display the Cube Manager in table view, click the row for any cube in the group, and click **Register Group** in the right area.

In either case, the change is automatically saved.

8.5.6 Specifying an Update Plan

To specify the update plan for a cube group and its cubes:

1. Display the Cube Manager in tree view.
2. Click the group in the middle area.
3. In the **Details** pane on the right, specify the following information:
   - **Name** — Unique name of this group.
   - **Exclude** — Controls whether the generated tasks perform update activities for cubes in this group. Initially this option is selected, and the group is excluded.

     The Cube Manager displays any excluded groups or cubes with a gray background.
   - **Update Plan** — Select an update plan.

     Note that the Cube Manager does not permit you to use synchronization unless that cube supports it (as described earlier in this chapter). For example, you can choose the **Build and Synch** plan for the group, but the Cube Manager automatically sets the update plan to **Build** for any cube that does not support synchronization.

     Important: Before you synchronize cubes from the Cube Manager, it is necessary to build the cubes at least once from the Cube Manager.
   - **Build every** — Use these fields to specify the schedule for the build task (if applicable).

- **Synch every** — Use these fields to specify the schedule for the synchronization task (if applicable).
- **Build Cubes Synchronously** — Select this to cause the system to build these cubes synchronously (if applicable). If this option is clear, the system builds them asynchronously.

   Initially, these details apply to all cubes in the group. If you edit details for a specific cube and then later want to reapply the group defaults, click **Apply to All Cubes in Group**.

4. Optionally click a cube within this group (in the middle area) and edit information for that cube in the **Details** pane on the right.

   The options are similar to those for the entire group, but include the following additional options, depending on whether the cube supports synchronization:

   - **Post-Build Code** — Specify a single line of ObjectScript to be executed immediately after building this cube. For example:

     ```
     do ##class(MyApp.Utils).MyPostBuildMethod("transactionscube")
     ```

   - **Pre-Synchronize Code** — Specify a single line of ObjectScript to be executed immediately before synchronizing this cube. For example:

     ```
     do ##class(MyApp.Utils).MyPresynchMethod("transactionscube")
     ```

     If needed, to abort the synchronization, do the following in your code:

     ```
     set $$$ABORTSYNCH=1
     ```

   - **Post-Synchronize Code** — Specify a single line of ObjectScript to be executed immediately after synchronizing this cube. For example:

     ```
     do ##class(MyApp.Utils).MyPostsynchMethod("transactionscube")
     ```

   In all cases, your code can perform any processing required.

   Modify each cube as needed.

5. Click **Save**.

   When you do so, the Cube Manager creates or updates the cube registry in this namespace. If the Task Manager does not yet include the necessary tasks, the Cube Manager creates them.

8.5.7 Merging Groups

You can merge one group (group A) into another (group B). Specifically this moves all the cubes from group A into the group B and then removes the now-empty group A.

To merge one group into another, use the following procedure. In this procedure, group A must not yet be registered, and group B must be registered.

1. Display the Cube Manager in tree view.
2. Drag group A (the group that contains the cubes that you want to move) from the left area and drop it into the group heading of group B (the target group) in the middle area.

   The system prompts you to confirm the action.

3. Click **OK**.

   If group B currently has an update plan that cannot be used for some of the newly moved cubes, the system displays a dialog box to indicate this. Click **OK**. For any such cubes, the Cube Manager selects an update plan that *can* be used.

4. Review the update plan for each newly moved cube and modify it as needed.
5. Click **Save**.

Or use the following alternative procedure. In this procedure, both groups must already be registered.

1. Display the Cube Manager in table view.
2. In the middle area, click the row for any cube in group A (the group that contains the cubes that you want to move).
3. On the right, click **Merge to another group** and then select group B (the target group) from the drop-down list.
4. Click **Merge**.

   The system prompts you to confirm the action.
5. Click **OK**.

   If group B currently has an update plan that cannot be used for some of the newly moved cubes, the system displays a dialog box to indicate this. Click **OK**. For any such cubes, the Cube Manager selects an update plan that *can* be used.
6. Review the update plan for each newly moved cube and modify it as needed.
7. Click **Save**.

8.5.8 Building All the Registered Cubes

The system provides a utility method that you can use to build all the registered cubes, in the correct order. The method is **BuildAllRegisteredGroups()** in the class %DeepSee.CubeManager.Utils. This method ignores the schedule specified in the registry but uses the build order specified in the registry.

Important: Before you synchronize cubes from the Cube Manager, it is necessary to build the cubes at least once from the Cube Manager user interface.

8.5.9 Performing On-Demand Builds

The Cube Manager also provides options to build cubes on demand (that is, ignoring the schedule). In this kind of build, the Cube Manager rebuilds the requested cube as well as any cubes that depend on it.

To perform an on-demand build:

1. Save any changes to the cube registry.

   Important: The build options are disabled if there are any unsaved changes.
2. Select a registered cube. To do so, either:
   - Display the Cube Manager in tree view and then click a cube in the middle area.
   - Display the Cube Manager in table view and click a cube that shows **Yes** in the **Registered** column.
3. On the right, clear the **Exclude** option.
4. Click **Build Dependency List**.

   The Cube Manager then displays the build dialog box.
5. Click **Build List**.

   The dialog box displays progress of the build.
6. When the build is done, click **OK**.

There are other ways to perform on-demand builds:

- Display the Cube Manager in tree view. Click the *header* of the table in the middle area. Then click **Build All Registered Groups**. Continue as described previously.
- Display the Cube Manager in tree view. Click a cube *group* in the middle area. Then click **Build This Group**. Continue as described previously.

8.5.10 Unregistering a Cube Group

To unregister a cube group:

1. Display the Cube Manager in tree view.
2. In the middle area, click the X in the row for the cube group.
3. Click **OK**.

8.5.11 Viewing Cube Manager Events

For certain events, the Cube Manager writes log entries to a table, which you can query via SQL. The table name is %DeepSee_CubeManager.CubeEvent. The CubeEvent field indicates the type of cube event. Possible logical values for this field include the following:

CubeEvent Value	When the Cube Manager Writes This Log Entry
`register`	Immediately after registering a cube group.
`update`	Immediately after saving changes to a cube group.
`unregister`	Immediately after unregistering a cube group.
`build`	When building a cube. The Cube Manager generates an initial log just before starting the build, and then updates that entry after the build is complete.
`synch`	When synchronizing a cube. The Cube Manager generates an initial log just before starting the synchronization is started, and then updates that entry after the synchronization is complete.
`presynch`	Immediately after executing any code specified by the **Pre-Synchronize Code** option.
`postsynch`	Immediately after executing any code specified by the **Post-Synchronize Code** option.
`postbuild`	Immediately after executing any code specified by the **Post-Build Code** option.
`repair`	When you use the **Build Dependency List** option (which performs an on-demand build of a given cube and any related cubes). The Cube Manager generates an initial log just before starting the build, and then updates that entry after the build is complete.

For information on other fields in this table, see the class reference for %DeepSee.CubeManager.CubeEvent.

8.5.12 Restricting Access to the Cube Manager

You may want to manage the cube update schedule without allowing users to change that schedule through the Cube Registry page. To restrict access to the Cube Registry page, set the `UserUpdatesLocked` attribute to `"true"` in either the RegistryMap or RegistryMapGroup objects within your saved cube registry. For example:

```
<RegistryMap Disabled="false" IndependentSync="false" SerialUpdates="false" UserUpdatesLocked="true">
```

When `UserUpdatesLocked` is set to `"true"` for a RegistryMap:

- The registry's **Disable** setting cannot be changed through the **Details** tab. For information on accessing this tab, see Modifying the Registry Details.

When `UserUpdatesLocked` is set to `"true"` for a RegistryMapGroup:

- Each registered group's **Exclude** check box is displayed but disabled
- Each registered cube's **Exclude** check box is hidden
- Each registered group's **Update Plan** is hidden
- Each registered cube's **Update Plan** is hidden
- The red X button for removing registered groups is removed
- The **Build Frequency** and **Synch Frequency** columns are left blank
- The **Build Dependency List** is available for cubes, but the **Build This Group** button is disabled.

8.6 Using %SynchronizeCube()

Note: Before you can synchronize a cube, follow the steps in "Enabling Cube Synchronization," earlier in this chapter.

To synchronize a cube programmatically (that is, without the Cube Manager), call the **%SynchronizeCube()** method of the %DeepSee.Utils class, which has the following signature:

```
classmethod %SynchronizeCube(pCubeName As %String, pVerbose As %Boolean = 1) as %Status
```

For the specified cube (*pCubeName*), this method finds and applies all changes from the source data that have been made since the last call to this method.

If *pVerbose* is true, the method writes status information to the console. For additional arguments for this method, see the class reference.

You can call **%SynchronizeCube()** in either of the following ways:

- Call the method from the part of your code that changes the data in the base class.

  This is the approach used in the Patients sample.
- Periodically call **%SynchronizeCube()** as a recurring task.

If **%SynchronizeCube()** displays the message `No changes detected`, this can indicate that you had not previously rebuilt the cube.

8.7 Purging DSTIME

For historical reasons and for convenience, the phrase *purging DSTIME* refers to purging the older entries from the *^OBJ.DSTIME* global. It is necessary to purge this global periodically because it can become quite large.

To purge DSTIME for a given cube, do the following:

1. Call the REST API /Data/GetDSTIME. See "GET /Data/GetDSTIME" in *Client-Side APIs for InterSystems Business Intelligence*. Pass, as an argument, the full name of the source class of the cube.

This REST call returns the last ^OBJ.DSTIME timestamp processed for that source class on a given server. In the case of an async mirror setup, the timestamp retrieved from this REST service will be the most recent timestamp that can safely be purged on the primary production server.

2. Using the returned timestamp as an argument, call the **%PurgeUpdateBuffer()** method of %DeepSee.Utils so that you purge ^OBJ.DSTIME up to but not including the timestamp processed on the remote server. The default behavior for this method is to increment the top node of the local *^OBJ.DSTIME* so that every purge will provide a new sync point to be propagated to the Business Intelligence server.

8.8 Updating Cubes Manually

As described in "When Cube Synchronization Is Not Possible," it is sometimes necessary to update a cube manually. In these situations, your application must do the following:

1. Determine the IDs of the affected records in the base class.
2. Update the cube for those records by calling the **%ProcessFact()** and **%DeleteFact()** methods of %DeepSee.Utils.

   As input, these methods require the ID of the affected row or rows.

The following list provides information on these methods:

%ProcessFact()

```
classmethod %ProcessFact(pCubeName As %String,
                         pSourceId As %String = "",
                         pVerbose As %Boolean = 0) as %Status
```

Where *pCubeName* is the logical name of a cube, and *pSourceID* is the ID of a record in the base class used by that cube. For the given cube, this method updates the corresponding row of the fact table, the associated indices, and any level tables if affected.

If *pVerbose* is true, the method writes status information to the console.

%DeleteFact()

```
classmethod %DeleteFact(pCubeName As %String,
                        pSourceId As %String = "",
                        pVerbose As %Boolean = 0) as %Status
```

Where *pCubeName* is the logical name of a cube, and *pSourceID* is the ID of a record in the base class used by that cube. For the given cube, this method deletes the corresponding row of the fact table and updates the indices correspondingly.

If *pVerbose* is true, the method writes status information to the console.

8.9 Other Options

This section discusses other options that are more advanced or less common:

- How to use DSTIME=MANUAL
- How to inject a record into the fact table
- How to prebuild dimension tables

- How to update a dimension table manually

8.9.1 Using DSTIME=MANUAL

Instead of letting the system automatically update the *^OBJ.DSTIME* global, you can update this global at times that *you* choose. To do so:

1. Specify *DSTIME* as `"MANUAL"` rather than `"AUTO"`.
2. Then within your application, call the method **%SetDSTimeIndex()** of the class %DeepSee.Utils whenever you add, change, or delete objects of the class, or when you want to update the *^OBJ.DSTIME* global.

   This method has the following signature:

   ```
   ClassMethod %SetDSTimeIndex(pClassName As %String,
                               pObjectId As %String,
                               pAction As %Integer,
                               pInterval As %Integer = 0)
   ```

   Where:

   - *pClassName* is the full package and class name of the object that you have added, changed, or deleted.
   - *pObjectId* is the object ID for that object.
   - *pAction* is 0 if you updated the object, 1 if you added it, or 2 if you deleted it or want to delete the corresponding fact from the fact table without deleting the object. The value of *pAction* is used as the value of the resulting *^OBJ.DSTIME* node. Note that facts are removed from a cube during synchronization if the corresponding record does not exist in the source class, or if a value of 2 is specified for *pAction*.
   - *pInterval* is an optional integer. If you specify this as a positive integer, the system uses time stamp subscripts in the *^OBJ.DSTIME* and *^DeepSee.Update* globals. See the discussion of the *DSINTERVAL* parameter in "Enable Cube Synchronization."

Then, when you want to update a given cube, call the **%SynchronizeCube()** method of the %DeepSee.Utils class, as described previously.

8.9.2 Injecting Facts into the Fact Table

In rare cases, you might need the fact table to include records that do not correspond to any source records. In such cases, use the **%InjectFact()** method of the cube class.

This method has the following signature:

```
classmethod %InjectFact(ByRef pFactId As %String,
                        ByRef pValues As %String,
                        pDimensionsOnly As %Boolean = 0)
                        as %Status
```

Where:

- *pFactId* is the ID of the fact. Set this to "" for an insert. On return, this argument contains the ID used for the fact.
- *pValues* is a multidimensional array of fact values. In this array, the subscript is the sourceProperty name (case-sensitive).
- *pDimensionsOnly* controls whether the method affects both the fact table and dimension tables or just the dimension tables. If this argument is true, the method affects only the dimension tables. You use this argument if you prebuild the dimension tables as described in the next section.

8.9.3 Prebuilding Dimension Tables

By default, the system populates the dimension tables at the same time that it builds the fact table. It is possible to prebuild one or more dimension tables so that they are populated before the fact table, if this is necessary for some reason.

To prebuild one or more dimension tables, do the following:

- Implement the **%OnBuildCube()** callback in the cube definition class. This method has the following signature:

  ```
  classmethod %OnBuildCube() as %Status
  ```

  The **%BuildCube()** method calls this method just after it removes the old cube contents and before it starts processing the new contents.

- In this implementation, invoke the **%InjectFact()** method of the cube class and specify the *pDimensionsOnly* argument as true.

  For details on this method, see the previous section.

For example, the following partial implementation predefines the Cities dimension in the HoleFoods sample:

```
ClassMethod %OnBuildCube() As %Status
{
    // pre-build City dimension
    Set tVar("Outlet.Country.Region.Name") = "N. America"
    Set tVar("Outlet.Country.Name") = "USA"

    Set tVar("Outlet") = 1000
    Set tVar("Outlet.City") = "Cambridge"
    Do ..%InjectFact("",.tVar,1)

    Set tVar("Outlet") = 1001
    Set tVar("Outlet.City") = "Somerville"
    Do ..%InjectFact("",.tVar,1)

    Set tVar("Outlet") = 1002
    Set tVar("Outlet.City") = "Chelsea"
    Do ..%InjectFact("",.tVar,1)

    Quit $$$OK
}
```

Notes:

- It is necessary to provide a unique ID as well as a name for a member.
- For completeness, this code should also provide the city population, longitude, and latitude, because the corresponding dimension table contains these values.
- It is also necessary to provide values for any higher level members.

8.9.4 Updating a Dimension Table Manually

In some cases, there is no change to your base class, but there is a change to a lookup table that is used as a level. In these cases, you can update the cube in any of the ways described earlier in this chapter. If the only change is to a single dimension table, however, it is quicker to update the level table directly. You can do so via the **%UpdateDimensionProperty()** method of %DeepSee.Utils.

This method has the following signature:

```
classmethod %UpdateDimensionProperty(pCubeName As %String,
                                     pSpec As %String,
                                     pValue As %String,
                                     pKey As %String)
                                     as %Status
```

Where:

- *pCubeName* is the name of the cube.
- *pSpec* is the MDX member expression that refers to the level member to update. You must use the dimension, hierarchy, and level identifiers in this expression. For example: `"[docd].[h1].[doctor].&[61]"`

  As a variation, *pSpec* can be a reference to a member property. For example: `"[homed].[h1].[city].&[Magnolia].Properties(""Principal Export"")"`

  The system uses this argument and the *pCubeName* argument to determine the table and row to update.
- *pValue* is the new name for this member, if any.

  Or, if you specified a member property, *pValue* is used as the new value of the property.
- *pKey* is the new key for this member, if any.

  Specify this argument only if you specify a member for *pSpec*.

You can make three kinds of changes with this method:

- Specify a new key for a member. For example:

  ```
  Set tSC =
  ##class(%DeepSee.Utils).%UpdateDimensionProperty("patients","[docd].[h1].[doctor].&[186]",,"100000")
  ```

  By default, the key is also used as the name, so this action might also change the name.
- Specify a new name for a member. For example:

  ```
  Set tSC =
  ##class(%DeepSee.Utils).%UpdateDimensionProperty("patients","[docd].[doctor].&[186]","Psmith,
  Alvin")
  ```

  By default, the name is the key, so this action might change the key.
- Specify a new value for some other property (both Name and Key are properties). For example:

  ```
  Set memberprop="homed.h1.city.Pine.Properties(""Principal Export"")"
  ```

  ```
  Set tSC = ##class(%DeepSee.Utils).%UpdateDimensionProperty("patients",memberprop,"Sandwiches")
  ```

8.10 Examples

The Patients sample includes utility methods that change data and that use either synchronization or manual updates as appropriate. To try these methods, you can use a dashboard provided with this sample:

1. Open the User Portal in the namespace where you installed the samples.
2. Click the dashboard Real Time Updates.
3. Click the buttons in the upper left area. Each of these executes a KPI action that executes a method to randomly change data in this sample. The action launches the method via JOB, which starts a background process.
   - **Add Patients** adds patients.

     This action calls a method that adds 100 patients and calls **%SynchronizeCube()** after adding each patient.
   - **Change Patient Groups** changes the patient group assignment for some patients.

     This action calls a method that randomly changes the patient group assignment for some percentage of patients and calls **%SynchronizeCube()** after each change.

- **Delete Some Patients** deletes some patients.

  This action calls a method that deletes 1 percent of the patients and calls **%SynchronizeCube()** after each deletion.

- **Change Favorite Colors** changes the favorite color for some patients.

  This action calls a method that randomly changes the favorite color for some percentage of the patients. In this case, the changed data is stored in the BI_Study.PatientDetails table, which is not the base table for the Patients cube. Hence it is necessary to use **%ProcessFact()** instead of **%SynchronizeCube()**.

  This block of code executes an SQL query to return all patients who are affected by the change to the data. It then iterates through those patients and updates the Patients cube for each of them.

- **Add Encounters** adds encounters for some patients.

  This action calls a method that includes logic similar to that for BI.Study.PatientDetails; see the previous item.

- **Change Doctor Groups** changes the doctor group assignment for some of the primary care physicians.

  This action calls a method that includes logic similar to that for BI.Study.PatientDetails.

Tip: These methods write log details to the global **^DeepSee.Study.Log**. For example:

```
^DeepSee.Study.Log(1)="13 May 2011 05:29:37PM Adding patients..."
^DeepSee.Study.Log(2)="13 May 2011 05:29:38PM Current patient count is 10200"
```

9
Executing Business Intelligence Queries Programmatically

This chapter describes how to use the InterSystems IRIS Business Intelligence result set API, as well as how to execute files that contain MDX files. It discusses the following topics:

- How to use the result set API in general
- Basic examples
- How to prepare and execute a query
- How to print the query results
- How to examine the query results
- How to examine the query results for a DRILLTHROUGH query
- How to examine the query metadata
- Other methods you can use
- How to execute query files

For information on **%ShowPlan()** and **%PrintStatistics()**, see “How the Analytics Engine Works,” later in this book.

Also see “Accessing the Samples Shown in This Book,” in the first chapter.

9.1 Using the Result Set API

The class %DeepSee.ResultSet enables you to execute MDX queries against cubes and to view and examine the results. To use this class, do the following:

1. Create an instance of %DeepSee.ResultSet.

   For example:

   ```
   set rset=##class(%DeepSee.ResultSet).%New()
   ```

2. Optionally disable use of the cache. To do so, set the %UseCache property of that instance equal to 0. For example:

   ```
   set rset.%UseCache=0
   ```

By default, caching is enabled.

3. Optionally enabling tracing. To enable detailed tracing during the prepare phrase, set the %Trace property of the result set instance equal. To enable tracing for all phases of the query, set the *%dstrace* variable equal to 1. For example:

```
set rset.%Trace=1
set %dstrace=1
```

   By default, tracing is disabled.

4. Create an MDX query, as a string. For example:

```
set query="SELECT MEASURES.[%COUNT] ON 0, diagd.MEMBERS ON 1 FROM patients"
```

   For details on the MDX syntax and functions supported in Business Intelligence, see *Using InterSystems MDX* and *InterSystems MDX Reference*.

5. Prepare and execute the query. Typically you do this as follows:

   a. Call the **%PrepareMDX()** method of your instance, using your query string as the argument.

   b. Call **%Execute()** or **%ExecuteAsynch()**.

   Each of these methods returns a status, which your code should check before proceeding.

   Or you can call **%ExecuteDirect()**, which prepares and executes the query.

   Or you can call lower-level methods of the %DeepSee.ResultSet; these are not discussed here.

   Note: If the query uses any plug-ins, note that **%Execute()** and **%ExecuteDirect()** do not return until all pending results are complete. Specifically they do not return until the analytics engine has finished executing any plug-ins used in the query.

6. If you used **%ExecuteAsynch()**, periodically check to see whether the query has completed. If the query uses any plug-ins, make sure that any pending results are also complete; pending results are the results from the plug-ins, which are executed separately from the query.

   To determine the status of the query, call the **%GetStatus()** method of your instance. Or call the **%GetQueryStatus()** class method of %DeepSee.ResultSet. These methods return the status of the query and also (separately) the status of any pending results; see the class documentation for details.

   Optionally, to cancel a query that has not yet completed, call the **%CancelQuery()** class method.

7. Your instance of %DeepSee.ResultSet now contains the query results. Now you can use methods of this instance to perform tasks such as the following:

   - Print the results.
   - Get cell values, get the number of cells or axes in the result set, and otherwise examine the results.
   - Get the metadata for the query itself, such as the query plan, the SQL used for the listing, the MDX used for a range of cells in the query, and so on.
   - Get the query statistics.

9.2 Basic Example

The following example creates and prepares a query, executes it, returns the result set as output, and displays the results:

```
ClassMethod RunQuery1(Output result As %DeepSee.ResultSet) As %Status
{
 Set rset=##class(%DeepSee.ResultSet).%New()
 Set query="SELECT MEASURES.[%COUNT] ON 0, diagd.MEMBERS ON 1 FROM patients"
 Set status=rset.%PrepareMDX(query)
 If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}

 Set status=rset.%Execute()
 If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}

 Write !, "Full results are as follows ***************",!
 Do rset.%Print()
 Quit $$$OK
}
```

When you run this method in the Terminal, you see results like the following:

```
SAMPLES>do ##class(BI.APISamples).RunQuery1()

Full results are as follows ***************
                             Patient Count
1 None                                8,394
2 asthma                                671
3 CHD                                   357
4 diabetes                              563
5 osteoporosis                          212
```

9.3 Preparing and Executing a Query

When you prepare and execute a query, you typically use the following methods:

%PrepareMDX()

```
method %PrepareMDX(pMDX As %String) as %Status
```

Parses the query, converts it to a runtime query object, and prepares it for execution.

%Execute()

```
method %Execute(ByRef pParms) as %Status
```

Executes the query synchronously; the *pParms* argument is discussed after this list. Use this only after you have prepared the query.

%ExecuteAsynch()

```
method %ExecuteAsynch(Output pQueryKey As %String,
                      ByRef pParms,
                      pWait As %Boolean = 0) as %Status
```

Executes the query asynchronously (or synchronously depending on the value of *pWait*). The arguments are discussed after this list. Use this only after you have prepared the query.

%ExecuteDirect()

```
classmethod %ExecuteDirect(pMDX As %String,
                           ByRef pParms,
                           Output pSC As %Status) as %DeepSee.ResultSet
```

Prepares and executes the query and then returns the result set. *pSC* is the status, which you should check. For the other arguments, see the discussion after this list.

Where:

- *pParms*— Specifies the values of any named parameters to use in this query. This is a multidimensional array with one or more nodes as follows:

Node	Value
Parameter name, not case-sensitive	Value of this parameter

  These values override any values for the same parameters given within the body of the query itself.
- *pQueryKey* — Returns the unique key for this query, for use when later referring to the query (to cancel it, get the cell count, or for other uses).
- *pWait* — Specifies whether to wait until the query has completed, before returning from this method call.

  If *pWait* is true, **%ExecuteAsynch()** runs synchronously.

The following sample uses a query that contains a named parameter; this is an InterSystems extension to MDX:

```
ClassMethod RunQuery2(city as %String = "Magnolia",Output result As %DeepSee.ResultSet) As %Status
{
 Set rset=##class(%DeepSee.ResultSet).%New()
 Set query="WITH %PARM c as 'value:Pine' "
            _"SELECT homed.[city].@c ON 0 FROM patients"
 Set status=rset.%PrepareMDX(query)
 If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}

 Set myparms("c")=city
 Set status=rset.%Execute(.myparms)
 If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}

 Write !, "Full results are as follows ***************",!
 Do rset.%Print()
 Quit $$$OK
}
```

The following shows an example Terminal session:

```
d ##class(BI.APISamples).RunQuery2("Centerville")

Full results are as follows ***************
                               Centerville
                                      1,124
```

9.4 Printing the Query Results

To display the query results for diagnostic purposes, use one of the following methods:

%Print()

Prints the query results and returns a status. For an example, see “Basic Example” and “Preparing and Executing a Query, ”earlier in this chapter.

%PrintListing()

If the query uses the MDX DRILLTHROUGH clause, this method performs the drillthrough for the first cell of the query, and prints the results to the current device. Otherwise, it prints an error.

This method does not return anything.

Important: Both methods include a line number at the start of each line of data (that is, after any column headings). The line number is *not* part of the results.

The following example demonstrates **%PrintListing()**:

```
ClassMethod RunQuery3()
{
    Set rset=##class(%DeepSee.ResultSet).%New()

    Set query="DRILLTHROUGH SELECT gend.female ON 0,birthd.[1913] ON 1 "
             _"FROM patients RETURN PatientID,PrimaryCarePhysician->LastName"

    Set status=rset.%PrepareMDX(query)
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit}

    Set status=rset.%Execute()
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit}

    Write !, "Listing details for the first cell are as follows ***************",!
    Do rset.%PrintListing()
}
```

You can use this in the Terminal as follows:

```
SAMPLES>d ##class(BI.APISamples).RunQuery3()

Listing details for the first cell are as follows ***************
   #  PatientID      LastName
   1: SUBJ_101317    Xiang
   2: SUBJ_104971    North
   3: SUBJ_105093    Klausner
   4: SUBJ_109070    Quine
```

9.5 Examining the Query Results

To work with the query results programmatically, you first need to understand their organization. The result set is a set of cells organized by a set of axes. Unless you are sure of the organization of the result set, use **%GetRowCount()** and **%GetColumnCount()** to get information about the number of rows and columns.

Then to access the value in a given cell, use the **%GetOrdinalValue()** method. Or to access the column and row header labels, use the **%GetOrdinalLabel()** method. Or to get detailed information about members used in a cell, use the **%GetAxisMembers()** method. The following subsections give the details.

Note: There are different methods to examine the results of a DRILLTHROUGH query. See the next section.

9.5.1 Getting the Number of Columns and Rows

To get the number of columns in the result set, use **%GetColumnCount()**.

Similarly, to get the number of rows, use **%GetRowCount()**.

For example, the following method prints a given result set and then uses the preceding methods to report on the axes of this result set:

```
ClassMethod ShowRowAndColInfo(rset As %DeepSee.ResultSet)
{
    //print query results
    write !, "Result set for comparison",!
    do rset.%Print()

    set colCount=rset.%GetColumnCount()
    set rowCount=rset.%GetRowCount()
    write !, "This result set has ",colCount, " column(s)"
    write !, "This result set has ",rowCount, " row(s)"
}
```

The following shows example output from this method:

```
Result set for comparison
                             Patient Count
1 None                                844
2 asthma                               55
3 CHD                                  38
4 diabetes                             55
5 osteoporosis                         26

This result set has 1 column(s)
This result set has 5 row(s)
```

The following shows output based on a different result set:

```
Result set for comparison

1 0 to 29->Female                     207
2 0 to 29->Male                       192
3 30 to 59->Female                    205
4 30 to 59->Male                      209
5 60+->Female                         115
6 60+->Male                            72

This result set has 1 column(s)
This result set has 6 row(s)
```

As noted earlier, remember that **%Print()** includes a line number at the start of each line of data, and this line number is not part of the results.

9.5.2 Getting the Value of a Given Cell

To get the value of a given cell, use **%GetOrdinalValue()**. This method has the following signature:

```
method %GetOrdinalValue(colNumber,rowNumber) as %String
```

Where *colNumber* is the column number (and 1 represents the first column). Similarly, *rowNumber* is the row number (and 1 represents the first row). If there is no such cell within the result set, the method returns null.

9.5.3 Getting the Column or Row Labels

To get the labels used for a column or a row, call the **%GetOrdinalLabel()** method of your instance. This method has the following signature:

```
method %GetOrdinalLabel(Output pLabel As %String,
                        pAxis As %Integer,
                        pPosition As %Integer,
                        Output pFormat As %String) as %Integer
```

Where:

- *pLabel* is a multidimensional array with one node for each label as follows:

Node	Value
Integer that represents the label number; the first label is 1, and so on.	Label

  In this array, the first label is the most specific (innermost) label, the second label is the next most specific, and so on. See the example.

  This array is returned as an output parameter.

- *pAxis* is the axis to examine. Use 1 to get the column labels or use 2 to get the row labels.
- *pPosition* is the position along the axis to examine. The first position is 1.

This method returns the number of labels at the given position on the given axis. The following shows an example. It executes a CROSSJOIN query (so that an axis has multiple labels), displays the results so that you can compare them to the labels, and then it iterates through the members on that axis, printing the labels for each:

```
ClassMethod ShowRowLabels() As %Status
{
    Set rset=##class(%DeepSee.ResultSet).%New()
    Set query="SELECT CROSSJOIN(aged.[age group].MEMBERS,"
    Set query=query_"gend.gender.MEMBERS) ON 1 FROM patients"
    Set status=rset.%PrepareMDX(query)
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}

    Set status=rset.%Execute()
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}

    Write !, "Full results are as follows ***************",!
    Do rset.%Print()

    Write !, "Labels used on the rows are as follows ***************",!
    For j=1:1:rset.%GetRowCount() {
        Write !, "Row ",j
        Set labelcount=rset.%GetOrdinalLabel(.pLabel,2,j)
        For i=1:1:labelcount {
            Write !, "    label("_i_") is "_pLabel(i)
            }
       }

    Quit $$$OK
}
```

When executed in the Terminal, this method gives output like the following:

```
SAMPLES>d ##class(BI.APISamples).ShowRowLabels()

Full results are as follows ***************

1 0 to 29->Female                      207
2 0 to 29->Male                        192
3 30 to 59->Female                     205
4 30 to 59->Male                       209
5 60+->Female                          115
6 60+->Male                             72

Labels used on the rows are as follows ***************

Row 1
    label(1) is Female
    label(2) is 0 to 29
Row 2
    label(1) is Male
    label(2) is 0 to 29
Row 3
    label(1) is Female
    label(2) is 30 to 59
Row 4
    label(1) is Male
    label(2) is 30 to 59
Row 5
    label(1) is Female
    label(2) is 60 +
Row 6
    label(1) is Male
    label(2) is 60 +
SAMPLES>
```

9.5.4 Getting Details for Cell Contents

So far, this chapter has provided instructions only on obtaining labels and cell values. In some cases, you might need more specific information about the contents of a given cell.

First, it is useful to review the concepts, with some example queries for reference. Consider the following query results, as seen in the Business Intelligence shell:

```
                             Patient Count
1 None                                 844
2 asthma                                55
3 CHD                                   38
4 diabetes                              55
5 osteoporosis                          26
```

In this example, each row corresponds to one member of the diagnosis dimension. The column corresponds to one member (`Patient Count`) of the Measures dimension. The following shows another example:

```
                             Patient Count
1 0 to 29->Female                      207
2 0 to 29->Male                        192
3 30 to 59->Female                     205
4 30 to 59->Male                       209
5 60+->Female                          115
6 60+->Male                             72
```

In this example, each row corresponds to a *tuple* that combines one member of the age group dimension with one member of the gender dimension. (A tuple is a intersection of members.)

In general, in an MDX result set, each row corresponds to a tuple and each column corresponds to a tuple. Each of these tuples might be a simple member as in the first example, or might be a combination of multiple members as shown in the second example. A tuple may or may not include a measure.

For any given cell, you might need to find information about the tuple of the column to which it belongs and the tuple of the row to which it belongs. To get information about these tuples, do the following:

1. Invoke the **%GetAxisMembers()** method of your result set:

   ```
   method %GetAxisMembers(pAxis As %Integer,
                          Output pKey,
                          pItemNo As %Integer = "") as %Status
   ```

   Finds information for the requested axis (and the optional requested item on that axis), writes that to a process-private global and returns, by reference, a key that you can use to retrieve information from that global. (The system writes this information to a process-private global because potentially there can be a large amount of information, and it is impossible to determine its structure ahead of time.)

   pAxis optionally specifies the axis you are interested in:

   - Use 0 to return information about the slicer axis (the WHERE clause).
   - Use 1 to return information about the columns (this is axis 0 in MDX).
   - Use 2 to return information about the rows.

   pKey, which is returned as an output parameter, is a key that you use later to access the information.

   pItemNo optionally specifies the tuple on that axis for which you want information. If you specify this argument, the method writes data only for that tuple; if you omit it, the method writes data for all tuples. Use 1 for the first tuple on an axis.

2. Use *pKey* to retrieve the appropriate node or nodes from the process-private global ^||*DeepSee.AxisMembers*. The **%GetAxisMembers()** method writes data to the nodes ^||*DeepSee.AxisMembers(pKey,pAxis,j,k)* where:

   - *pKey* is the key returned by the **%GetAxisMembers()** method.
   - *pAxis* is an integer that specifies the axis.
   - *j* is an integer that specifies the tuple in which you are interested. Use 0 for the first tuple on an axis.
   - *k* is an integer that specifies the member of the tuple in which you are interested. Use 1 for the first member of a tuple.

3. Retrieve the appropriate list items from each of those nodes. Each node of ^||*DeepSee.AxisMembers* has a value of the following form:

   ```
   $LB(nodeno,text,dimName,hierName,levelName,memberKey,dimNo,hierNo,levelNo)
   ```

   Where:

   - *nodeno* is the node number of this part of the axis.
   - *text* is the text for this part of the axis.
   - *dimName*, *hierName*, and *levelName* are the names of the dimension, hierarchy, and level used for this part of the axis.
   - *memberKey* is the key for the member used for this part of the axis.
   - *dimNo*, *hierNo*, and *levelNo* are the numbers of the dimension, hierarchy, and level used for this part of the axis.

4. Kill the generated nodes of the process-private global ^||*DeepSee.AxisMembers*.

   Or, if you are certain that no other processes are using the **%GetAxisMembers()** method, kill the entire global.

   The system does not automatically kill this global.

The following example method prints a description of the column and row tuples for a given cell, given a result set and a cell position:

```
ClassMethod ShowCellDetails(rset As %DeepSee.ResultSet, col As %Integer = 1, row As %Integer = 1)
{
    //print query results
    write !, "Result set for comparison",!
    do rset.%Print()

    //call %GetAxisMembers to build process-private global with info
    //for given result set and axis; return key of node that has this info
    Set status=rset.%GetAxisMembers(1,.columnkey)
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit}
    Set status=rset.%GetAxisMembers(2,.rowkey)
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit}

    write !, "We are looking at the cell ("_col_","_row_")"
    write !, "The value in this cell is ", rset.%GetOrdinalValue(col,row)
    write !, "For this cell, the column is a tuple that combines the following member(s):"
    set i=0
    while (i '= "") {
        write !, "   Member ",i
        set infolist=^||DeepSee.AxisMembers(columnkey,1,col,i)
        write:$LI(infolist,3)'="" !, "      Dimension name: ",$LI(infolist,3)
        write:$LI(infolist,4)'="" !, "      Hierarchy name: ",$LI(infolist,4)
        write:$LI(infolist,5)'="" !, "      Level name: ",$LI(infolist,5)
        write:$LI(infolist,6)'="" !, "      Member key: ",$LI(infolist,6)
        set i=$ORDER( ^||DeepSee.AxisMembers(columnkey,1,col,i) )
    }

    write !, "For this cell, the row is a tuple that combines the following member(s):"
    set i=0
    while (i '= "") {
        write !, "   Member ",i
        set infolist=^||DeepSee.AxisMembers(rowkey,2,row,i)
        write:$LI(infolist,3)'="" !, "      Dimension name: ",$LI(infolist,3)
        write:$LI(infolist,4)'="" !, "      Hierarchy name: ",$LI(infolist,4)
        write:$LI(infolist,5)'="" !, "      Level name: ",$LI(infolist,5)
        write:$LI(infolist,6)'="" !, "      Member key: ",$LI(infolist,6)
        set i=$ORDER( ^||DeepSee.AxisMembers(rowkey,2,row,i) )
    }
    Kill ^||DeepSee.AxisMembers(columnkey)
    Kill ^||DeepSee.AxisMembers(rowkey)
}
```

The following shows example output for this method:

```
Result set for comparison
                           0 to 29          30 to 59               60+
 1 Female->None                 189               184                 62
```

```
 2 Female->asthma                18                 7                 7
 3 Female->CHD                    *                 4                14
 4 Female->diabetes               *                11                23
 5 Female->osteopor               *                 *                23
 6 Male->None                   178               186                45
 7 Male->asthma                  14                 7                 2
 8 Male->CHD                      *                 5                15
 9 Male->diabetes                 *                11                10
10 Male->osteoporos               *                 *                 3

We are looking at the cell (2,6)
The value in this cell is 186
For this cell, the column is a tuple that combines the following member(s):
   Member 0
      Dimension name: AgeD
      Hierarchy name: H1
      Level name: Age Group
      Member key: 30 to 59
For this cell, the row is a tuple that combines the following member(s):
   Member 0
      Dimension name: GenD
      Hierarchy name: H1
      Level name: Gender
      Member key: Male
   Member 1
      Dimension name: DiagD
      Hierarchy name: H1
      Level name: Diagnoses
      Member key: <null>
```

9.6 Examining the Query Results for a DRILLTHROUGH Query

If the query uses the MDX DRILLTHROUGH statement, then you use a different technique to examine the results.

In this case, use the following method of your instance of %DeepSee.ResultSet:

```
method %GetListingResultSet(Output pRS As %SQL.StatementResult, Output pFieldList As %List) as %Status
```

This method returns the following as output parameters:

- *pRS* is an instance of %SQL.StatementResult that contains the results from the DRILLTHROUGH query.
- *pFieldList* is a list (in **$LIST** format) of the fields in this result set.

Use *pRS* in the same way that you use any other instance of %SQL.StatementResult; see the class reference for details.

9.7 Examining the Query Metadata

You can use the following methods to get the cube name, query text, and other metadata for any instance of %DeepSee.ResultSet. (For information on accessing the query plan, see the next section.)

%GetCubeName()

```
method %GetCubeName() as %String
```

Returns the name of the cube that the query uses. The query must be prepared before you can use this method.

%GetListingSQL()

```
method %GetListingSQL() as %String
```

Returns the SQL statement used to display the source data, if the query is a DRILLTHROUGH query.

%GetParameterInfo()

```
method %GetParameterInfo(Output pParms) as %Status
```

Returns a multidimensional array that contains the parameters used in the query, along with the values used for them. This array has the structure described earlier in this chapter.

%GetQueryText()

```
method %GetQueryText() as %String
```

Returns a string that contains the MDX query that was used to create this result set.

%GetSlicerForCellRange()

```
method %GetSlicerForCellRange(Output pSlicer As %String,
                              pStartRow As %Integer, pStartCol As %Integer,
                              pEndRow As %Integer, pEndCol As %Integer)
                              as %Status
```

Returns, by reference, a string that contains the MDX slicer statement for the given range of cells. You specify a range of cells by indicating a rectangle that consists of a starting row and column and an ending row and column. The first cell position on any axis is 1.

%IsDrillThrough()

```
method %IsDrillThrough() as %Boolean
```

Returns true if the query is a DRILLTHROUGH query; returns false otherwise.

For example, the following method generates a report on the basic metadata:

```
ClassMethod ShowQueryMetadata(rset As %DeepSee.ResultSet) As %Status
{
    Set cubename=rset.%GetCubeName()
    Write !, "This result set comes from the following cube: ",cubename,!

    Set status=rset.%GetParameterInfo(.pParms)
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}
    If $DATA(pParms) {
        Write "The query uses the following parameters:",!
        Set p = $ORDER(pParms(""))
            While (p '= "") {
                Write $$$UPPER(p), " = " ,$GET(pParms(p,"VALUE")),!
                Set p = $ORDER(pParms(p))
        }
    }
    Set query=rset.%GetQueryText()
    Write "The query is as follows:",!, query,!

    Set isdrill=rset.%IsDrillThrough()
    If isdrill {
        Set listingsql=rset.%GetListingSQL()
        Write !!, "It uses the following SQL to drill into the source table:"
        Write !, listingsql
        }
}
```

The following examples (with line breaks added for readability) show output from this method, using several sample result sets. In the first case, we use **GetResultSet1()** of the sample class BI.APISamples:

```
SAMPLES>set rs1=##class(BI.APISamples).GetResultSet1()

SAMPLES>d ##class(BI.APISamples).ShowQueryMetadata(rs1)

This result set comes from the following cube: patients
The query is as follows:
SELECT {[MEASURES].[AVG TEST SCORE],[MEASURES].[%COUNT]} ON 0,
[DIAGD].[H1].[DIAGNOSES].MEMBERS ON 1 FROM [PATIENTS]
```

In the next example, we use **GetResultSet2()**, which uses a query that contains named parameters:

```
SAMPLES>set rs2=##class(BI.APISamples).GetResultSet2()

SAMPLES>d ##class(BI.APISamples).ShowQueryMetadata(rs2)

This result set comes from the following cube: patients
The query uses the following parameters:
C = Magnolia
The query is as follows:
SELECT [HOMED].[H1].[CITY].MAGNOLIA ON 0,%SEARCH ON 1 FROM [PATIENTS]
```

In the next example, we use **GetResultSet3()**, which uses a query that does a drillthrough:

```
SAMPLES>set rs3=##class(BI.APISamples).GetResultSet3()

SAMPLES>d ##class(BI.APISamples).ShowQueryMetadata(rs3)

This result set comes from the following cube: patients
The query is as follows:
DRILLTHROUGH SELECT [GEND].[H1].[GENDER].[FEMALE] ON 0,[BIRTHD].[H1].[YEAR].[1913] ON 1
FROM [PATIENTS] RETURN  PatientID, PrimaryCarePhysician-> LastName


It uses the following SQL to drill into the source table:
SELECT TOP 1000 PatientID,PrimaryCarePhysician-> LastName FROM
BI_Study.Patient source WHERE source.%ID IN (SELECT _DSsourceId FROM
BI_Model_PatientsCube.Listing WHERE _DSqueryKey = '1858160995')
```

The following example method generates a report that shows the MDX slicer for a given range of cells, in a given result set:

```
ClassMethod ShowSlicerStatement(rset As %DeepSee.ResultSet, Row1 As %Integer = 1,
Col1 As %Integer = 1, Row2 As %Integer, Col2 As %Integer) As %Status
{
    If '$DATA(Row2) {Set Row2=Row1}
    If '$DATA(Col2) {Set Col2=Col1}

    Set status=rset.%GetSlicerForCellRange(.slicer,Row1,Col1,Row2,Col2)
    If $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit status}

    Write !, "The requested cell range:"
    Write !, "   Columns ",Col1, " through ", Col2
    Write !, "   Rows    ",Row1, " through ", Row2

    Write !, "The slicer statement for the given cell range is as follows:"
    Write !, slicer

    If 'rset.%IsDrillThrough(){
        Write !!, "For comparison, the query results are as follows:",!
        Do rset.%Print()
    }
    Else {
           Write !!, "This is a drillthrough query and %Print "
            _"does not provide a useful basis of comparison"
           }
}
```

To try this method, we use **GetResultSet4()** of BI.APISamples, which uses a query that has different levels for rows and columns:

```
SAMPLES>d ##class(BI.APISamples).ShowSlicerStatement(rs4)

The requested cell range:
   Columns 1 through 1
   Rows    1 through 1
The slicer statement for the given cell range is as follows:
CROSSJOIN({[AgeD].[H1].[Age Bucket].&[0 to 9]},{[GenD].[H1].[Gender].&[Female]})

For comparison, the query results are as follows:
                              Female                Male
1 0 to 9                          689                 724
2 10 to 19                        672                 722
3 20 to 29                        654                 699
4 30 to 39                        837                 778
5 40 to 49                        742                 788
6 50 to 59                        551                 515
7 60 to 69                        384                 322
8 70 to 79                        338                 268
9 80+                             204                 113
```

9.8 Other Methods

The class %DeepSee.ResultSet also provides additional methods like the following:

- **%GetCellCount()**
- **%FormatNumber()**
- **%GetOrdinalLabel()**
- **%GetOrdinalKey()**
- **%GetQueryKey()**
- **%GetRowTotal()**
- **%GetColumnTotal()**
- **%GetGrandTotal()**

For a full list and details, see the class reference.

9.9 Executing Query Files

The system provides a tool for executing MDX queries that have been saved in files. The output can be written to the current device or to a file. The output results include statistics on the query run.

This tool can be useful for simple testing.

9.9.1 About Query Files

A query file must be an ASCII file as follows:

- Any line breaks in the file are ignored.
- Two or more blank spaces in a row are treated as a single blank space.
- The file can contain any number of MDX queries (zero or more).
- The queries can contain comments, but comments cannot be nested. An MDX comment has the following form:

  ```
  /* comment here */
  ```

A comment may or may not be on its own line.

- Use the command GO on a line by itself to execute a query. The query consists of all text from the previous GO (or the start of the file) up to, but not including, the GO command.

  There must be no spaces before GO on this line.

For example:

```
/* First query in this file*/
SELECT MEASURES.%COUNT ON 0,
homed.[home zip].[34577].CHILDREN
ON 1 FROM patients
GO


/* Second query in the file*/
SELECT MEASURES.%COUNT ON 0,
homed.[home city].MEMBERS ON 1 /*ignore this comment*/FROM patients
GO
```

9.9.2 Executing a Query File

To execute a query file, use the following class method of %DeepSee.Shell:

```
ClassMethod %RunQueryFile(pQueryFile As %String, pResultFile As %String = "") As %Status
```

Where:

- *pQueryFile* is the name of the query file.
- *pResultFile* is the name of the file into which to write the query statistics.

  If this argument is null, the method writes the query statistics to the current device.

In all cases, the method writes the query results to the current device.

For example:

```
d ##class(%DeepSee.Shell).%RunQueryFile("c:\mdxtest.txt")
------------------------------------------------------
Query 1:
/* First query in this file*/SELECT MEASURES.%COUNT ON 0, homed.[home zip].[34577].CHILDREN ON 1 FROM
 patients
                                     Count
1 Cypress                             1,091
2 Magnolia                            1,087
3 Pine                                1,121
Query Statistics:
 Results Cache:                         1
 Computations:                          0
 Cache Hits:                            0
 Cells:                                 0
 Expressions:                           0

 Prepare:                           0.261 ms
 Execute Axes:                      0.026 ms
 Execute Cells:                     0.000 ms
 Consolidate:                       0.000 ms
 Total Time:                        0.287 ms

ResultSet Statistics:
 Cells:                                 3
 Parse:                             3.553 ms
 Display:                           0.361 ms
 Total Time:                        3.914 ms
------------------------------------------------------
Query 2:
/* Query 2*/SELECT MEASURES.%COUNT ON 0, homed.[home city].MEMBERS ON 1 /*ignore this comment*/FROM
patients
```

```
                                     Count
1 Cedar Falls                         1,119
...
```

For information on query statistics, see “How the Analytics Engine Works,” later in this book.

10
Performing Localization

This chapter describes how to localize strings in InterSystems IRIS Business Intelligence. It discusses the following topics:

- Overview
- How to prepare for localization of the model
- How to prepare for localization of the folder items
- How to localize the strings

Also see "Accessing the Samples Shown in This Book," in the first chapter.

10.1 Overview of Localization in Business Intelligence

This section provides an overview of how InterSystems IRIS Business Intelligence supports localization of strings.

10.1.1 Model Localization

The system provides a simple mechanism for localizing the names of level, measures, and other model elements.

Every element in the Business Intelligence model has a logical value and a display value. You specify the logical value, the original display value, and alternative display values for use with other language locales. Then:

- In MDX queries, you always use the logical value.
- The user interfaces use the appropriate display value, if available. The user configures the browser to use a preferred language, and when the browser sends requests to a server, those requests indicate the preferred language to use, if available. The server sends a reply that includes the appropriate set of strings, based on that language preference.

10.1.2 Folder Item Localization

In a similar manner, you can localize a specific set of following strings within dashboards, pivot tables, and other folder items. For these strings, you specify the original display value and alternative display values for use with other language locales.

The User Portal and the dashboard viewer use the appropriate display value, if available. The user configures the browser to use a preferred language, and when the browser sends requests to a server, those requests indicate the preferred language to use, if available. The server sends a reply that includes the appropriate set of strings, based on that language preference.

10.2 Preparing for Model Localization

To prepare for localization of strings in the Business Intelligence models, do the following:

- Specify the *DOMAIN* class parameter in each cube, subject area, and KPI class.

  For example:

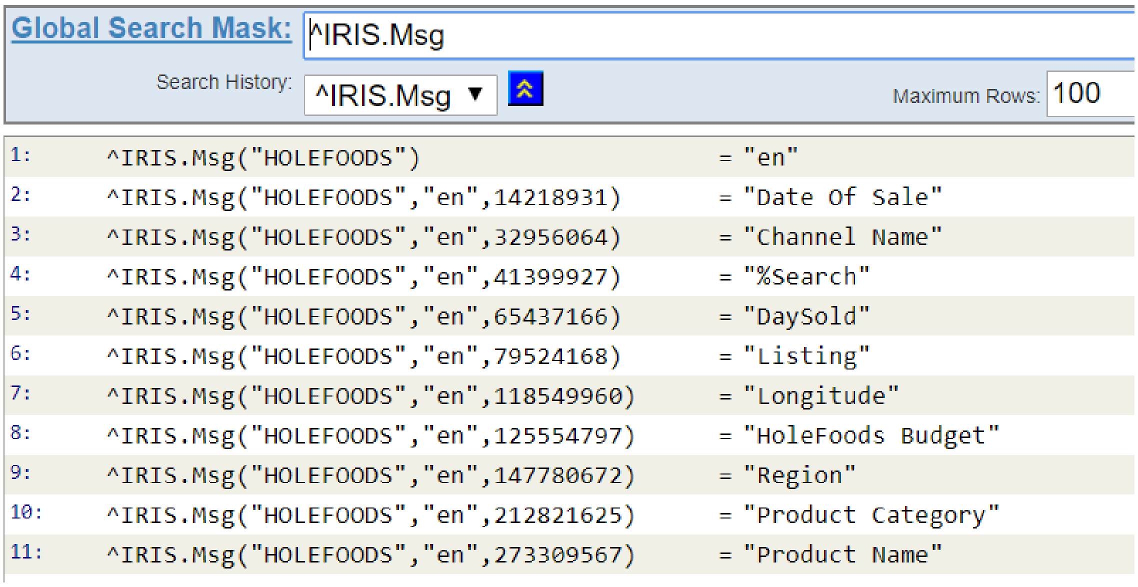
  ```
  Parameter DOMAIN = "PATIENTSAMPLE";
  ```

  The classes in the Patients sample all use the same value for *DOMAIN*, but this practice is not required. You can specify a different value for each class.

- Specify a value for the `displayName` attribute for every Business Intelligence element.

  In the Architect, when you specify a name, the system initializes the **Display name** field with the same value. When you work in Atelier, however, you must remember to specify the `displayName` attribute (which is optional), in addition to the `name` attribute (which is required).

When you compile the classes, the system adds values to the **^IRIS.Msg** global in this namespace. These values may look like this:



```
^IRIS.Msg
^IRIS.Msg
100
1:   ^IRIS.Msg("HOLEFOODS")                        = "en"
2:   ^IRIS.Msg("HOLEFOODS","en",14218931)          = "Date Of Sale"
3:   ^IRIS.Msg("HOLEFOODS","en",32956064)          = "Channel Name"
4:   ^IRIS.Msg("HOLEFOODS","en",41399927)          = "%Search"
5:   ^IRIS.Msg("HOLEFOODS","en",65437166)          = "DaySold"
6:   ^IRIS.Msg("HOLEFOODS","en",79524168)          = "Listing"
7:   ^IRIS.Msg("HOLEFOODS","en",118549960)         = "Longitude"
8:   ^IRIS.Msg("HOLEFOODS","en",125554797)         = "HoleFoods Budget"
9:   ^IRIS.Msg("HOLEFOODS","en",147780672)         = "Region"
10:  ^IRIS.Msg("HOLEFOODS","en",212821625)         = "Product Category"
11:  ^IRIS.Msg("HOLEFOODS","en",273309567)         = "Product Name"
```

This global (which is known as the *Message Dictionary*) contains the *messages* defined in this namespace; for Business Intelligence, each message corresponds to the name of a model element.

When you compile a cube, subject area, or KPI class that defines the *DOMAIN* parameter, the system updates this global to include the messages defined in that class, in your default language. Each message uses a numeric identifier and has a string value that applies to the default language.

If you do not see the expected set of strings, make sure that the class defines the *DOMAIN* parameter, that you have specified values for `displayName`, and that you have compiled the class.

10.3 Preparing for Folder Item Localization

This section describes how to prepare for localization of strings in the dashboards, pivot tables, and other folder items.

10.3.1 Default Domain

`DeepSeeUser` is the domain that the system uses by default when it looks for a localized string in a dashboard. For details, see the following sections.

10.3.2 Adding Strings to the Message Dictionary

Create a class that, when compiled, generates a set of entries in the Message Dictionary. In this class:

- Extend %RegisteredObject or any other class that provides access to the standard system macros.
- Specify the *DOMAIN* class parameter. For example:

  ```
  Parameter DOMAIN = "DeepSeeUser";
  ```

  The `DeepSeeUser` domain is the most convenient choice, because this is the default domain.

- Define a method that uses `$$$Text(`*Localizable String*`)` to refer to each string that the given domain should contain. *Localizable String* is an expression that evaluates to a string in this domain.

  You can specify any name for the method. It does not need to take any arguments or return any values. The following shows an example:

  ```
  ClassMethod DefineL18N()
  {
     set x=$$$Text("Dashboard Title")
     set x=$$$Text("Dashboard Description")
     set x=$$$Text("KeywordA")
     set x=$$$Text("KeywordB")

     set x=$$$Text("Control Label")
     set x=$$$Text("Tooltip")
     set x=$$$Text("Widget Title")
     set x=$$$Text("Chart Title")
  }
  ```

  Or, instead of `$$$Text(`*Localizable String*}, use `$$$Text(@`*MessageID*`@)` where *MessageID* is a numeric ID that is unique within the given domain.

When you compile this class, the compiler finds each instance of the `$$$Text` macro and adds values to the **^IRIS.Msg** global in this namespace.

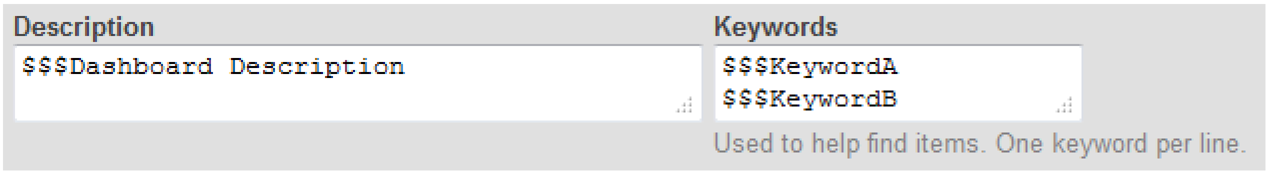
10.3.3 Using Localizable Strings in a Dashboard, Pivot Table, or Other Folder Item

In the definition of a dashboard, pivot table, or other folder item, use one of the following values instead of the exact string that you want to see:

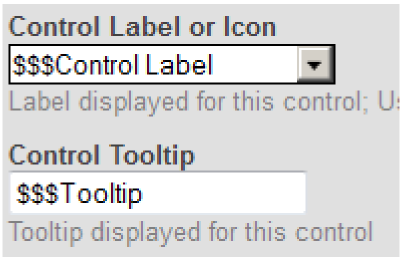
- `$$$`*Localizable String*

  Where *Localizable String* is a string defined in the default domain.

  For example:


Description
$$$Dashboard Description
Keywords
$$$KeywordA
$$$KeywordB
Used to help find items. One keyword per line.

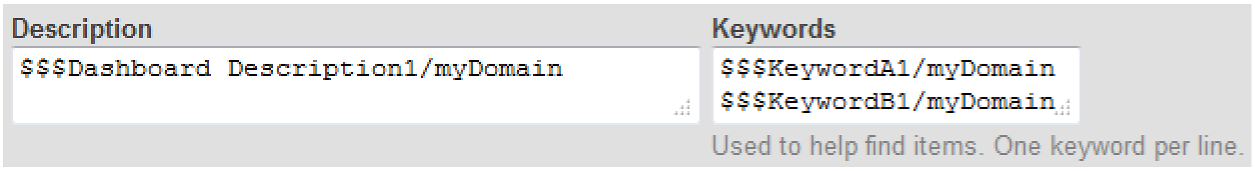
For another example:


Control Label or Icon
$$$Control Label
Label displayed for this control; U
Control Tooltip
$$$Tooltip
Tooltip displayed for this control

- $$$*Localizable String*/*OtherDomain*

  Where *Localizable String* is a string defined in the domain given by *OtherDomain*.

  For example:


Description
$$$Dashboard Description1/myDomain
Keywords
$$$KeywordA1/myDomain
$$$KeywordB1/myDomain
Used to help find items. One keyword per line.

  If you do not include the /*OtherDomain* part, the system looks for this string in the default domain.

  Important: For the name of a folder or of a folder item, use the following variation: $$$*Localizable String*#*OtherDomain*

  For example: use the following as a folder name: `$$$My Folder#MyDeepSeeDomain`

- $$$@*MessageID*

  Where *MessageID* is a numeric message ID defined in the default domain.

- $$$@*MessageID*/*OtherDomain*

  Where *MessageID* is a numeric message ID defined in the domain given by *OtherDomain*.

  If you do not include the /*OtherDomain* part, the system looks for this string in the default domain.

Use these values for any of the following strings in the folder item definition:

- Folder name
- Folder item name
- (For dashboards) Dashboard title (if specified, this is shown instead of the dashboard name)
- Item description
- Keywords
- Labels for dashboard controls
- Tooltips for dashboard controls
- Titles of widgets (but not their logical names)

- Chart titles within dashboard widgets that display charts

10.4 Localizing the Strings

To localize the strings:

1. Export the Message Dictionary to one or more XML files. To do so, do the following in the Terminal:
   a. Change to the namespace in which you are using Business Intelligence.
   b. Identify the output file and its location:

      ```
      SET file="C:\myLocation\Messages.xml"
      ```

      The specified directory must already exist; the system does not create it.
   c. Run the export command:
      - It may be practical to export only those messages in a particular domain:

        ```
        DO ##class(%Library.MessageDictionary).ExportDomainList(file,"myDomain")
        ```

        The domain names are case-sensitive.
      - Or, to export all the messages in the namespace:

        ```
        DO ##class(%Library.MessageDictionary).Export(file)
        ```

2. For each desired language, make a copy of the message file.
3. Edit each message file as follows:
   a. Edit the `Language` attribute of the root element:

      ```
      <MsgFile Language="en">
      ```

      Change this to the language name of the desired language.

      This must be an all-lowercase language tag that conforms to RFC1766 (so that a user can choose the preferred language in the browser from the standard set). This tag consists of one or more parts: a primary language tag (such as `en` or `ja`) optionally followed by a hyphen (-) and a secondary language tag (so that the result has the form `en-gb` or `ja-jp`).

      For example:

      ```
      <MsgFile Language="es">
      ```

   b. Scan the file to find the `<MsgDomain>` element that corresponds to the appropriate domain:

      ```
      <MsgDomain Domain="myDomain">
      ```

      If you exported only one domain, the file contains only one `<MsgDomain>` element.
   c. Within this section, edit the value of each message. For example, change this:

      ```
          <Message Id="2372513034">City</Message>
      ```

      To this:

      ```
          <Message Id="2372513034">Ciudad</Message>
      ```

4. Import the edited message file or files. To do so:

   - To import a single file:

     ```
     SET file="C:\myLocation\myfile.xml"
     DO ##class(%Library.MessageDictionary).Import(file)
     ```

   - To import all the files in the same directory:

     ```
     SET myFiles="C:\myLocation"
     DO ##class(%Library.MessageDictionary).ImportDir(myFiles,"d")
     ```

5. Optionally use the Management Portal to verify that the message dictionary has been updated. To do so, switch to the appropriate namespace, select **System Explorer** > **Globals**, and then click **View Globals** for the **^IRIS.Msg** global.

   Within this global, you should see a new set of subscripts that correspond to the language you have added.

6. In your browser, find the setting that controls the language that it requests for use on localized pages. Change this setting to the language that you specified in the edited message file.

   Depending on the browser, you might need to clear the browser cache, restart the browser, or both.

7. Access the Analyzer and validate that you see translated strings.

For more information on the utility methods in %Library.MessageDictionary, see the class reference for that class or see the article *String Localization and Message Dictionaries*.

11 Packaging Business Intelligence Elements into Classes

In most cases, you develop your application elements on a test system and then copy them to a production system. This chapter describes how to package the InterSystems IRIS Business Intelligence elements and copy them to another system. It discusses the following topics:

- Overview
- How to export Business Intelligence folder items to container classes
- When and how to edit the folder item definitions
- How to import exported container classes
- How to use the Folder Manager

Note: This chapter assumes that you are familiar with the process of exporting from and importing into Atelier.

Also see the appendix "Other Export/Import Options."

Also see "Accessing the Samples Shown in This Book," in the first chapter.

11.1 Overview

Your Business Intelligence implementation may include some or all of the following elements:

- Cube class definitions
- Subject area class definitions
- KPI class definitions
- Business Intelligence folder items, which include all the items that are not defined as classes. These include pivot tables, dashboards, pivot variables, and so on.

To move all these items to another system (here called the *target system*), do the following:

1. Export all the folder items to one or more Business Intelligence container classes, as described in the next section.

   A Business Intelligence container class contains an XML representation of any number of Business Intelligence folder items.

2. Export the cube, subject area, and KPI class definitions.

   You can create a project that contains all your Business Intelligence class definitions and folder items. Then you can export this project from InterSystems IRIS® and import it into another InterSystems IRIS instance, where needed. You can use the Atelier export/import options or you can use the usual class methods in %SYSTEM.OBJ.

3. Examine the exported folder item definitions to make edits for portability.

4. Import all the class definitions to the target system.

   When you compile the container classes, the system iterates over all the folder items contained in those classes and creates or overwrites each of those items in the target system.

11.2 Exporting Folder Items to a Container Class

To export Business Intelligence folder items to container classes, you use a method that generates a file that defines a container class that includes the items. The method is **%ExportContainer()**, which is in the class %DeepSee.UserLibrary.Utils. This method is as follows:

```
classmethod %ExportContainer(ByRef pItemList As %String,
                             pFileName As %String,
                             pContainerClassName As %String = "") as %Status
```

Where:

- *pItemList* is a multidimensional array that has nodes of the following form:

Node	Node Value
pItemList(*itemidentifier*)	`""`

  For each *itemidentifier*, use one of the following strings:

  – *dashboardname*`.dashboard` where *dashboardname* is the name of a dashboard. You can use the wildcard `*` to represent all dashboards; you can use the wildcard with the other types of items as well.

  – *pivotname*`.pivot` where *pivotname* is the name of a pivot table (or use `*`).

    Note that the **%ExportContainer()** method identifies all the pivot tables used by any dashboard you export. The only pivot tables you need to export explicitly are the pivot tables that are not used by any dashboard.

  – *namedfiltername*`.namedFilter` where *namedfiltername* is the name of a named filter (or use `*`).

  – *sharedcalcmembername*`.sharedCalcMember` where *sharedcalcmembername* is the name of a shared calculated member (or use `*`).

  – *listinggroupname*`.listingGroup` where *listinggroupname* is the name of a listing group (or use `*`).

  – *pivotvarname*`.pivotVariable` where *pivotvarname* is the name of a pivot variable (or use `*`).

  – *settingname*`.userSetting` where *settingname* is the name of a user setting (or use `*`).

  – *termlistname*`.termList` where *termlistname* is the name of a term list (or use `*`).

  – *themename*`.theme` where *themename* is the name of a dashboard theme (or use `*`).

  – *widgettemplatename*`.widgetTemplate` where *widgettemplatename* is the name of a widget template (or use `*`).

  – *linkname*`.link` where *linkname* is the name of a dashboard link (or use `*`).

   – *reportname*`.report` where *reportname* is the name of a dashboard report (or use `*`).

- *pFileName* is the name of the file to generate.
- *pContainerClassName* is the full name of the container class to generate, including package.

11.3 Editing the Business Intelligence Folder Items for Portability

If you intend to copy a Business Intelligence folder item to another system, it is worthwhile to examine the exported XML and make any necessary edits, discussed in the following subsections.

Also note the following points:

- When you export a dashboard, the system does not automatically export any pivot tables that it uses. It is your responsibility to identify and export the pivot tables as well.
- References between Business Intelligence elements (such as from a dashboard to any pivot tables) are made by name.

11.3.1 Removing <filterState> Elements

If it was saved in a previous release, a folder item definition might contain `<filterState>` elements, which are no longer supported. If so, you should remove these — that is, remove both the starting tag `<filterState>` and the matching ending tag `</filterState>`.

11.3.2 Stripping Out Local Data

A folder item definition might also contain information that is local to your system and not available on another system (depending on what elements you package and share between systems). Check the XML for the following items:

`localDataSource` attribute

Where found: `<widget>` elements in exported dashboards.

This attribute contains any local overrides performed in the Mini Analyzer. You should always clear this when you use the exported XML in another system. For example, change this:

```
localDataSource="$LOCAL/Basic Dashboard Demo/SamSmith/590125613.pivot"
```

To this:

```
localDataSource=""
```

Or remove the `localDataSource` attribute.

`owner` attribute

Where found: All folder items.

This element contains the name of the user who owns this item. If the given user does not exist on the target system, edit this attribute. You can set the attribute to null. For example, change this:

```
owner="DevUser"
```

To this:

```
owner=""
```

Or you can remove the attribute.

`resource` attribute

Where found: All folder items.

This element contains the name of the resource used to secure this item, if any. If this resource does not exist on the target system, edit this attribute. You can set the attribute to null or even remove the attribute.

`createdBy` attribute

Where found: All folder items.

This element contains the name of the user who created this item. You can set the attribute to null or even remove the attribute. If you do so, when the XML is imported (or the container class is compiled), `createdBy` is set to the current user.

`timeCreated` attribute

Where found: All folder items.

This element contains the name of the user who created this item. You can set the attribute to null or even remove the attribute. If you do so, when the XML is imported (or the container class is compiled), `timeCreated` is set to the current time stamp.

11.4 Importing an Exported Container Class

To import an exported container class, use the **%ImportContainer()** method, which is in the class %DeepSee.UserLibrary.Utils. This method is as follows:

```
ClassMethod %ImportContainer(pFileName As %String = "", pReplace As %Boolean = 1) As %Status
```

Where:

- *pFileName* is the name of the file to generate.
- *pReplace* specifies whether to replace the existing class.

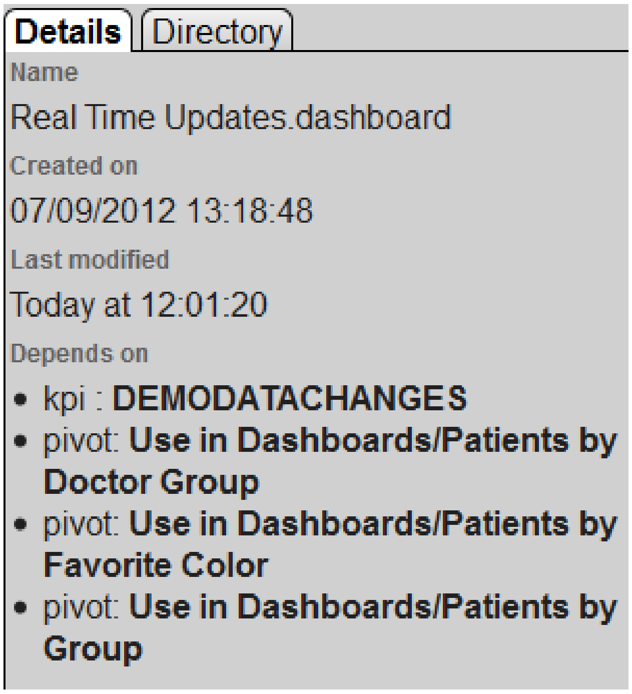
Note that **%ImportContainer()** automatically calls the **%OnLoad()** method if it is defined in the container class.

11.5 Using the Folder Manager

This section describes how to use the Folder Manager to see the dependencies of an item, export items, and import items. You can also export and import in Atelier, as described later in this chapter.

11.5.1 Seeing the Dependencies of a Folder Item

If you click the check box for a single item, the left area of the Folder Manager displays details for that item, including a list of the items that it depends on:



Name

Real Time Updates.dashboard

Created on

07/09/2012 13:18:48

Last modified

Today at 12:01:20

Depends on

- kpi : **DEMODATACHANGES**
- pivot: **Use in Dashboards/Patients by Doctor Group**
- pivot: **Use in Dashboards/Patients by Favorite Color**
- pivot: **Use in Dashboards/Patients by Group**

11.5.2 Exporting Business Intelligence Folder Items to the Server

To export Business Intelligence folder items to files on the server:

1. Click the InterSystems Launcher and then click **Management Portal**.

   Depending on your security, you may be prompted to log in with an InterSystems IRIS username and password.

2. Switch to the appropriate namespace as follows:

   a. Click **Switch**.

   b. Click the namespace.

   c. Click **OK**.

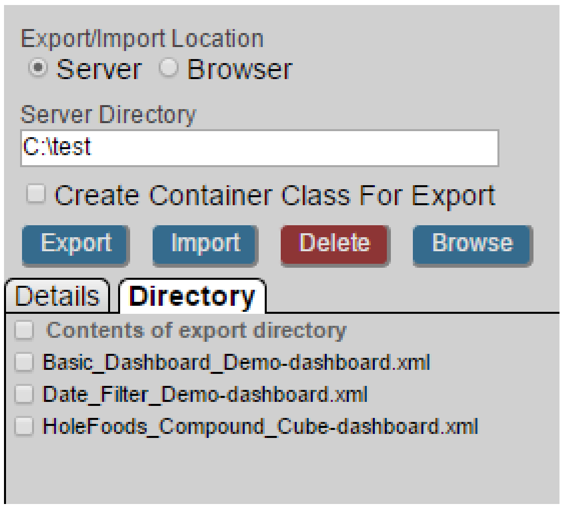
3. Select **Analytics** > **Admin** > **Folder Manager**.
4. Select **Server**.
5. For **Server Directory**, type the full path of the directory in which to export the items. Or type the name of a directory relative to the directory that contains the default database for this namespace. Or use the **Browse** button.

   The directory must already exist.

6. Click the check box next to each item that you want to export.

   Or to select all items, click the check box at the top of the column of check boxes.

7. Click **Export**.
8. Optionally click the **Directory** tab, which shows the files in the given directory.


Export/Import Location
● Server ○ Browser
Server Directory
C:\test
☐ Create Container Class For Export
Export
Import
Delete
Browse
Details
Directory
☐ Contents of export directory
☐ Basic_Dashboard_Demo-dashboard.xml
☐ Date_Filter_Demo-dashboard.xml
☐ HoleFoods_Compound_Cube-dashboard.xml

11.5.2.1 Variation: Exporting a Container Class

To instead export a single file that consists of a container class that contains the given folder items, do the following:

1. Specify **Server** and **Server Directory** as in the preceding steps.
2. Select the items to export.
3. Select the option **Create Container Class For Export**.
4. Optionally select **Export Related Supporting Items** to export all supporting items that might be needed to deploy the selected folder items. Examples of supporting items include pivot variables, named filters, and shared calculated members.
5. For **Container Class Name**, optionally specify a fully qualified class name (package and class). If no **Container Class Name** is specified, both the container class and the export file will use generated names.
6. Click **Export**.

For information on container classes, see “Packaging Business Intelligence Folder Items into Classes.”

11.5.3 Exporting Business Intelligence Folder Items to the Browser

To export Business Intelligence folder items to the browser’s download directory:

1. Click the InterSystems Launcher and then click **Management Portal**.

   Depending on your security, you may be prompted to log in with an InterSystems IRIS username and password.
2. Switch to the appropriate namespace as follows:
   a. Click **Switch**.
   b. Click the namespace.
   c. Click **OK**.
3. Select **Analytics** > **Admin** > **Folder Manager**.
4. Select **Browser**.
5. Select the items to export.
6. Optionally select **Export Related Supporting Items**.

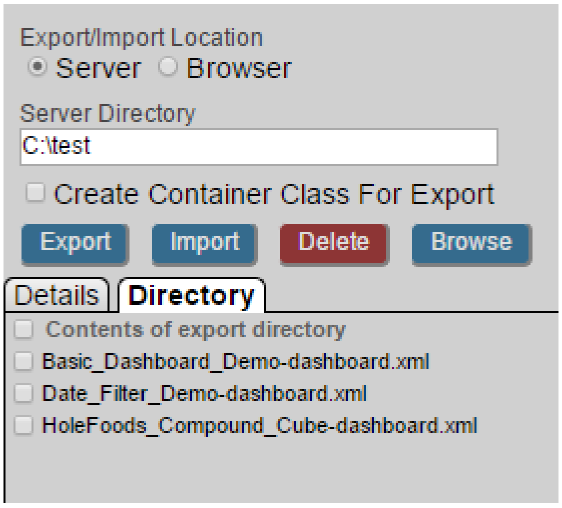
7. For **Container Class Name**, optionally specify a fully qualified class name (package and class). If no **Container Class Name** is specified, both the container class and the export file will use generated names.

8. Click **Export**.

11.5.4 Importing Business Intelligence Folder Items

To import a folder item that has previously been exported:

1. Click **Analytics**, **Admin**, and then click **Folder Manager**.
2. For **Server Directory**, type the full path of the directory that contains the exported items. Or type the name of a directory relative to the directory that contains the default database for this namespace.
3. Click the **Directory** tab, which shows the filenames for items in the given directory.



C:\test
Create Container Class For Export
Export
Import
Delete
Browse
Details
Directory
Contents of export directory
Basic_Dashboard_Demo-dashboard.xml
Date_Filter_Demo-dashboard.xml
HoleFoods_Compound_Cube-dashboard.xml

4. Click the check box next to each file that you want to import.

   Or to select all items, click the check box at the top of the column of check boxes.

5. Click **Import**.
6. Click **OK** at the prompt to continue. Or click **Cancel**.

Note: For the items created when you import the files, the owner is the username under which the InterSystems service runs, for example _SYSTEM.

11.5.4.1 Variation: Importing Local Files to the Server

To import a local file to the server:

1. Click **Analytics**, **Admin**, and then click **Folder Manager**.
2. Select **Browser**.
3. Click the **Directory** tab, and then click **Choose File**.
4. Select the file that you want to import.
5. Click **Import**.
6. Click **Ok** at the prompt to continue. Or click **Cancel**.

12
Creating Portlets for Use in Dashboards

This chapter describes how to create portlets that users can add to dashboards, as widgets. It discusses the following topics:

- Basics of defining portlets
- How to define a portlet that provides settings
- Simple examples

Also see "Accessing the Samples Shown in This Book," in the first chapter.

12.1 Portlet Basics

To define a portlet, create and compile a class as follows:

- Use %DeepSee.Component.Portlet.abstractPortlet as a superclass.
- Implement the **%DrawHTML()** method, which should draw the body of the portlet as HTML.

  This method has the following signature:

  ```
  method %DrawHTML()
  ```

  Also see "Using Settings" for additional options.
- Optionally implement the **%OnGetPortletName()** method, which returns the localized name of the portlet, to display in the Widget Builder dialog box.

  Otherwise, the short class name becomes the portlet name.

  This method has the following signature:

  ```
  classmethod %OnGetPortletName() as %String
  ```
- Optionally implement the **%OnGetPortletIcon()** method, which returns the URL of the icon for the portlet, to display in the Widget Builder dialog box.

  Otherwise, the system uses a generic icon.

  This method has the following signature:

  ```
  classmethod %OnGetPortletIcon() as %String
  ```
- Optionally implement the **%OnGetPortletSettings()** method, which returns one or more configurable settings. See "Defining Settings."

Otherwise, the portlet has no settings.

- Optionally implement the **adjustContentSize()** method, which the system calls whenever the widget containing the portlet is loaded or resized. This method has the following signature:

  ```
  ClientMethod adjustContentSize(load, width, height) [ Language = javascript ]
  ```

- Optionally implement the **onApplyFilters()** method, which the system calls whenever a filter change is sent to the widget. This method has the following signature:

  ```
  ClientMethod onApplyFilters(refresh) [ Language = javascript ]
  ```

12.2 Defining and Using Settings

It is fairly simple to define a portlet that provides configurable settings. To do this, implement the **%OnGetPortletSettings()** method in the portlet class. This method has two purposes:

- To define settings to be listed in the **Settings** menu for this widget, in the Dashboard Designer.
- To receive values for these settings via the dashboard URL. For information on passing the values via the URL, see the chapter "Accessing Dashboards from Your Application."

The **%OnGetPortletSettings()** method has the following signature:

```
classmethod %OnGetPortletSettings(Output pInfo As %List, ByRef pSettings) as %Status
```

pInfo should be a multidimensional array that contains the following nodes:

Node	Value
pInfo(integer)	List returned by **$LISTBUILD** as follows: **$LB**(*name*,*default*,*type*,*caption*,*tooltip*) • *name* is the logical name of the setting • *default* is the default value of the setting • *type* is the type of the setting. See the following subsection. • *caption* is the localized caption of the setting • *tooltip* is an optional tooltip

pSettings is a multidimensional array that is passed to this method; it contains the values of any settings passed via the URL. For details, see the second subsection.

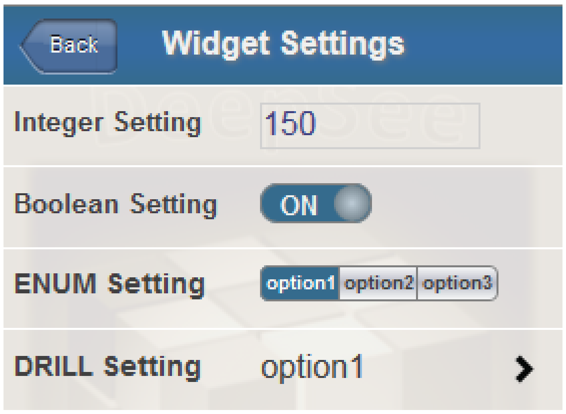
12.2.1 Types of Settings

In the *pInfo* argument of **%OnGetPortletSettings()**, you can specify the type of each setting; this controls how the Dashboard Designer displays that setting. Use one of the following:

- `"%Integer"`
- `"%Boolean"`

- `"ENUM^caption1:value1,caption2:value2"` or a similar form. In this string, *caption1* and *caption2* are labels to display in the Dashboard Designer, and *value1* and *value2* are the corresponding values that are actually used. In practice, a setting of this type can provide only a few options, before the Dashboard Designer runs out of space to display them. See the next item.
- `"DRILL^caption1:value1,caption2:value2"` or a similar form. In this string, *caption1* and *caption2* are labels to display in the Dashboard Designer, and *value1* and *value2* are the corresponding values that are actually used.

The following figure shows a sample of each of these types of setting:



The following implementation of **%OnGetPortletSettings()** shows how these settings were defined:

```
ClassMethod %OnGetPortletSettings(Output pInfo As %List, ByRef pSettings) As %Status
{
 Kill pInfo
 set pInfo($I(pInfo)) = $LB("INTEGERSETTING","150","%Integer","Integer Setting","Sample integer setting")


 set pInfo($I(pInfo)) = $LB("BOOLEANSETTING","1","%Boolean","Boolean Setting","Sample boolean setting")


 set pInfo($I(pInfo)) = $LB("ENUMSETTING","150","ENUM^option1:150,option2:200,option3:200",
 "ENUM Setting","Sample ENUM setting")

 set pInfo($I(pInfo)) = $LB("DRILLSETTING","150",
 "DRILL^option1:150,option2:200,option3:200,option4:200,option5:200,option6:200,option7:200",
 "DRILL Setting","Sample DRILL setting")

 Quit pInfo
}
```

12.2.2 Receiving Settings Passed Via URL

The URL of a dashboard can pass values to some or all widgets on that dashboard, including values for any portlet settings. To accept these values, when you implement **%OnGetPortletSettings()**, use the *pSettings* argument, which is a multidimensional array that contains values for any settings that were provided in the URL. The structure of this array is as follows:

Node	Value
pSettings`("`*setting*`")` where *setting* is the name of a setting	Value of that setting

One approach is to use **`$GET`**`(`*pSettings*`("setting")` as the default value for each setting. For example:

```
ClassMethod %OnGetPortletSettings(Output pInfo As %List, ByRef pSettings) As %Status
{
  Kill pInfo
  Set pInfo($I(pInfo)) = $LB("LOGO",$G(pSettings("LOGO")),"","Clock logo","Logo displayed on top of
clock")

  Set pInfo($I(pInfo)) = $LB("STEP",$G(pSettings("STEP"),"10"),"%Integer",
 "Second hand redraw interval (msec)","milliseconds steps of second hand")

  Set pInfo($I(pInfo)) = $LB("OFFSET",$G(pSettings("OFFSET"),"0"),"%Integer",
  "Offset from base time (min)","minutes difference from base time (Local or UTC)")

  Set pInfo($I(pInfo)) = $LB("UTC",$G(pSettings("UTC"),"0"),"%Boolean","UTC","Time Base: local (default)
 or UTC")

  Set pInfo($I(pInfo)) = $LB("CIRCLE",$G(pSettings("CIRCLE"),"1"),"%Boolean",
  "Circle","Shape: square (default) or circle")

  Set pInfo($I(pInfo)) = $LB("SIZE",$G(pSettings("SIZE"),"150"),"%Integer","Size","Size of the clock")


  Quit pInfo
}
```

12.2.3 Using Settings

To use the settings in the portlet, define the **%DrawHTML()** method so that it extracts the values of the settings from the settings property of the portlet and then uses those values in whatever manner is suitable for your needs. The settings property of the portlet is a multidimensional array of the following form:

Node	Value
settings("*setting*") where *setting* is the name of a setting	Value of that setting

For a simple example, **%DrawHTML()** could contain extract a setting called `SIZE`:

```
 set size=$G(..settings("SIZE"))
```

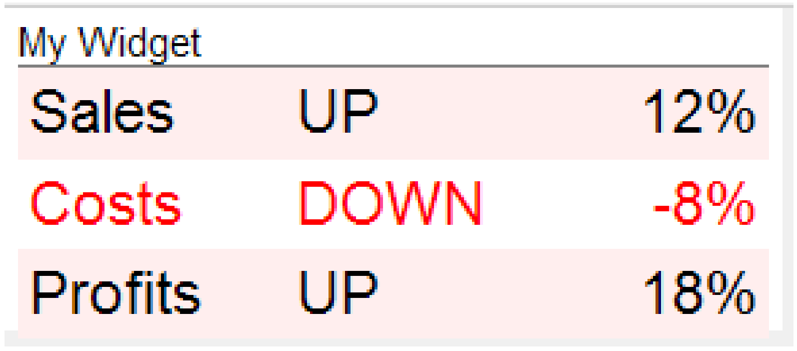
And the method could use this value to set the size of the portlet.

12.3 Examples

The following shows a simple example:

```
Class BI.Model.Custom.MyPortlet Extends %DeepSee.Component.Portlet.abstractPortlet
{

/// Static HTML display method: draw the BODY of this component as HTML.
Method %DrawHTML()
{
  &html<<div class="portletDiv" style="overflow:hidden;">>
  &html<<div style="font-size:16px; border-bottom:1px solid gray;">My Widget</div>>

  Set tInfo(1) = $LB("Sales","UP","12")
  Set tInfo(2) = $LB("Costs","DOWN","-8")
  Set tInfo(3) = $LB("Profits","UP","18")

  &html<<table width="100%" cellspacing="0" border="0">>
  Set n = $O(tInfo(""))
  While (n'="") {
    Set tName = $LG(tInfo(n),1)
    Set tDir = $LG(tInfo(n),2)
    Set tPct = $LG(tInfo(n),3)
    Set clr = $S(tPct<0:"red",1:"black")
    Set bg = $S(n#2:"#FFEEEE",1:"white")
    Set tPct = tPct _ "%"
    &html<<tr style="font-size:24px; background:#(bg)#;color:#(clr)#;">
      <td style="padding:4px;">#(tName)#</td>
      <td style="padding:4px;">#(tDir)#</td>
      <td style="padding:4px;text-align:right;">#(tPct)#</td></tr>>
    Set n = $O(tInfo(n))
  }
  &html<</table>>
  &html<</div>>
}

}
```

When used as a widget, the widget has the following contents:



Sales	UP	12%
Costs	DOWN	-8%
Profits	UP	18%

This example displays static data, but your portlet could display real-time data.

For a more complex example that also defines settings, see the sample class `BI.Model.PortletDemo.ClockPortlet`.

13
Other Development Work

Depending on the users' needs and the business requirements, you may have to do some or all of the following additional development work:

- Adding paper sizes
- Adding audit code
- Creating initialization code, to initialize the server environment

Also see "Accessing the Samples Shown in This Book," in the first chapter.

13.1 Adding Paper Sizes

When users print a dashboard widget to a PDF file, the system provides a default set of paper sizes, and the user can choose among them. To extend this set of sizes, add nodes as needed to the **^DeepSee.PaperSizes** global, as follows:

Node	Value
^DeepSee.PaperSizes(*n***)** where *n* is an integer	**$LISTBUILD(***sizename***,***dimensions***)** where *sizename* is the name of the size and *dimensions* specifies the dimensions. *dimensions* must have one of the following forms: *width*x*height* `in` *width*x*height* `mm` There must be exactly one space between *height* and the unit name.

For example:

```
Set ^DeepSee.PaperSizes(1) = $LB("My Sticker","100x100 mm")
```

The new size is immediately available.

13.2 Auditing User Activity

You can execute custom code, such as writing to an audit log, every time a user executes a query or accesses a dashboard.

To add custom code to execute when users execute a query, perform the following one-time setup steps:

- Write a class method, routine, or subroutine that contains the custom code. The first subsection provides details on the requirements and options; the second subsection provides an example.
- Set **`^DeepSee.AuditQueryCode`** equal to a string containing a valid ObjectScript statement that executes that method, routine, or subroutine.

  For example, do the following in the Terminal:

  ```
  set ^DeepSee.AuditCode="do ^MyBIAuditCode"
  ```

  Every time a query is executed in this namespace, the system executes the code specified in **`^DeepSee.AuditQueryCode`**, thus invoking your routine or class method.

Similarly, to add custom code to execute when users access a dashboard:

- Write a class method, routine, or subroutine that contains the custom code.
- Set **`^DeepSee.AuditCode`** equal to a string containing a valid ObjectScript statement that executes that method, routine, or subroutine.

  Every time a dashboard is accessed in this namespace, the system executes the code specified in **`^DeepSee.AuditCode`**.

13.2.1 Audit Code Requirements and Options

When you define audit code for either scenario, make sure that the code does not write any output to the current device. Also make sure that it does not kill any `%` variables required by InterSystems IRIS®.

Your code can use the following variables:

- **$USERNAME** — name of the current user.
- **$ROLES** — roles of the current user.
- *%dsQueryText* — text of the current query.
- *%dsCubeName* — logical name of the cube used in the current query.
- *%dsResultSet* — current instance of %DeepSee.ResultSet, which you can use to access other information, if needed. For details on working with %DeepSee.ResultSet, see "Executing Business Intelligence Queries Programmatically," earlier in this book.
- *%dsDashboard* — name of the dashboard that is being accessed, if any.

Typically, audit code writes output to a file or to a global.

Note that *%dsQueryText*, *%dsCubeName*, and *%dsResultSet* are only available to audit routines using **`^DeepSee.AuditQueryCode`**, while *%dsDashboard* is only available to routines using **`^DeepSee.AuditCode`**.

13.2.2 Example

The following shows a simple example audit routine. It has one subroutine for use with **`^DeepSee.AuditQueryCode`** and another subroutine for use with **`^DeepSee.AuditCode`**:

```
 ; this is the routine DeepSeeAudit
 quit

dashboard
 set auditentry="At "_$ZDT($H,3)_", " _$USERNAME_" accessed dashboard: "_%dsDashboard
 set ^MyBIAuditLog($INCREMENT(^MyBIAuditLog))=auditentry
 quit

query
 set auditentry="At "_$ZDT($H,3)_", " _$USERNAME_" ran query: "_%dsQueryText
 set ^MyBIAuditLog($INCREMENT(^MyBIAuditLog))=auditentry
 quit
```

To use this routine, we would enter the following two lines in the Terminal:

```
SAMPLES>set ^DeepSee.AuditQueryCode="do query^DeepSeeAudit"

SAMPLES>set ^DeepSee.AuditCode="do dashboard^DeepSeeAudit"
```

To see the audit log, we can use ZWRITE. The following shows example results (with line breaks added for readability):

```
SAMPLES>zw ^MyBIAuditLog
^MyBIAuditLog=2
^MyBIAuditLog(1)="At 2014-06-20 16:26:38, SamSmith accessed dashboard: User Defined Listing.dashboard"
^MyBIAuditLog(2)="At 2014-06-20 16:26:38, SamSmith ran query: SELECT NON EMPTY {[MEASURES].[AMOUNT 
SOLD],
[MEASURES].[UNITS SOLD]} ON 0,NON EMPTY [DATEOFSALE].[ACTUAL].[YEARSOLD].MEMBERS ON 1 FROM [HOLEFOODS]"
```

13.3 Defining Server Initialization Code

To define server initialization code:

- Place a valid ObjectScript statement in the **^DeepSee.InitCode** global.

  For example, do the following in the Terminal:

  ```
  set ^DeepSee.InitCode="do ^myroutine"
  ```

- Make sure that the code does not write any output to the current device.
- Also make sure that it does not kill any `%` variables required by InterSystems IRIS.

This code is called by the **%RunServerInitCode()** method of %DeepSee.Utils. This method is called whenever an InterSystems IRIS Business Intelligence session is created.

14 Setting Up Security

InterSystems IRIS Business Intelligence has a formal mechanism for managing access to functionality and Business Intelligence items. This mechanism is based on the underlying InterSystems security framework. This chapter discusses the following topics:

- Overview of security in Business Intelligence
- Basic requirements for using Business Intelligence
- Requirements for performing common tasks
- How to add security for model elements
- How to specify the resource for a pivot table or dashboard
- How to specify the resource for a folder

This chapter assumes that you are familiar with InterSystems security as described in the *Security Administration Guide*. In particular, it assumes that you understand the relationships between *resources*, *roles*, and *users*.

Note: If you install InterSystems IRIS® with the **Minimal Security** option (and if you do not tighten security after that), the user UnknownUser belongs to the **`%All`** role and has access to all parts of Business Intelligence. In this case, ignore this chapter.

Important: Also note that you use Business Intelligence from within a web application. By default, a web application can access a subset of InterSystems classes, which does not include the %DeepSee classes. To use Business Intelligence in your web application, you must explicitly enable access to Analytics. For details, see Setting Up the Web Applications.

14.1 Overview of Security

The following table summarizes how elements in Business Intelligence are secured:

Element	How Secured
Business Intelligence User Portal	**`%DeepSee_Portal`** and **`%DeepSee_PortalEdit`** resources
Analyzer	**`%DeepSee_Portal`**, **`%DeepSee_Analyzer`**, and **`%DeepSee_AnalyzerEdit`** resources
Architect	**`%DeepSee_Portal`**, **`%DeepSee_Architect`** and **`%DeepSee_ArchitectEdit`** resources
Folder Manager and Cube Manager	**`%DeepSee_Portal`** and **`%DeepSee_Admin`** resources
MDX Query Tool and Settings pages	**`%DeepSee_Portal`**, **`%DeepSee_Admin`**, and **`%Development`** resources
Term List Manager and Quality Measure Manager pages	**`%DeepSee_Portal`** and **`%DeepSee_PortalEdit`** resources
Listing Group Manager	**`%DeepSee_ListingGroup`**, **`%DeepSee_ListingGroupEdit`**, and **`%DeepSee_ListingGroupSQL`** resources
Cubes, subject areas, listings, listing fields, listing groups, KPIs, folders, and folder items (such as dashboards and pivot tables)	Custom resources (optional)
Quality measures	*Accessible only to users of any cubes to which the quality measures are published; no additional security*
Term lists	*No security options*

For details, see "Requirements for Common Business Intelligence Tasks," later in this chapter.

14.2 Basic Requirements

For a user to use Business Intelligence, the following must be true, in addition to the other requirements listed in the rest of this chapter:

- The user must have access to the database or databases in which Business Intelligence is used.

  By default, when you create a database, InterSystems IRIS does the following:

  - Creates a resource with a name based on the database name (%DB_*database_name*).
  - Establishes that this resource controls access to the new database.
  - Creates a role with the same name as the resource. This role has read and write privileges on the resource.

    You can specify whether the read and write privileges are public. These privileges are not public by default.

  For example, suppose that you create a database called MyApp for use with Business Intelligence, and you let InterSystems IRIS create the resource and role as described here, and suppose that the read and write privileges are *not* public. In this case, a Business Intelligence user must belong to the **`%DB_MyApp`** role, which has read and write privileges on the **`%DB_MyApp`** resource.

- If the **^DeepSee** globals are mapped from another database, the user must also have access to the database that contains these globals.

14.3 Requirements for Common Business Intelligence Tasks

The following table lists the security requirements for common tasks, in addition to the items in the previous section.

Task	Privileges That the User Must Have for This Task*
Viewing the User Portal (apart from the Analyzer or the mini Analyzer) with no ability to create dashboards	USE permission for the **`%DeepSee_Portal`** resource
Viewing the User Portal (apart from the Analyzer or the mini Analyzer) with the ability to create new dashboards	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_PortalEdit`** resource
Viewing a dashboard (including exporting to Excel and printing to PDF)	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the resource (if any) associated with the dashboard; see "Adding Security for Model Elements" • USE permission for the resources (if any) associated with the pivot tables used in the dashboard • USE permission for the resources (if any) associated with the folders that contain the dashboard and the pivot tables • USE permission for the resources (if any) associated with the cubes or subject areas** used in the pivot tables • USE permission for the resources (if any) associated with the KPIs used in the dashboard • SQL SELECT privilege for all tables used by the queries of the KPIs Note that the system displays all widgets to which the user has permission. That is, the dashboard is displayed even though the user cannot see all of it.
Read-only access to the Analyzer or Mini Analyzer	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_Analyzer`** resource
Full access to the Analyzer or Mini Analyzer	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_AnalyzerEdit`** resource
Viewing a listing	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the resource (if any) associated with the listing • SQL SELECT privilege for all tables used by the listing

Task	Privileges That the User Must Have for This Task*
Modifying an existing pivot table in the Analyzer	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_AnalyzerEdit`** resource • USE and WRITE permissions for the resource (if any) associated with the given pivot table • USE permission for the resources (if any) associated with the folders that contain the pivot table • USE permission for the resources (if any) associated with the cube** or subject area used in the pivot table
Creating a new dashboard	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_PortalEdit`** resource • USE permission for the resource (if any) associated with the folder that contains the dashboard
Modifying an existing dashboard	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_PortalEdit`** resource • USE and WRITE permissions for the resource (if any) associated with the given dashboard • USE permission for the resource (if any) associated with the folder that contains the dashboard
Read-only access to the Architect	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_Architect`** resource
Creating a new cube or subject area in the Architect	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_ArchitectEdit`** resource
Modifying an existing cube or subject area in the Architect	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_ArchitectEdit`** resource • USE and WRITE permissions for the resource (if any) associated with the given cube or subject area; see "Adding Security for Model Elements"
• Folder Manager page • MDX Query Tool page • Settings pages	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_Admin`** resource *or* USE permission for the **`%Development`** resource
• Term List Manager page • Quality Measures page	• USE permission for the **`%DeepSee_Portal`** resource • USE permission for the **`%DeepSee_PortalEdit`** resource

Task	Privileges That the User Must Have for This Task[*]
Listing Group Manager (read only access)	USE permission for the **`%DeepSee_ListingGroup`** resource
Listing Group Manager (edit access, except for custom SQL query options)	USE permission for the **`%DeepSee_ListingGroupEdit`** resource
Listing Group Manager (edit access, *including* custom SQL query options)	• USE permission for the **`%DeepSee_ListingGroupEdit`** resource • USE permission for the **`%DeepSee_ListingGroupSQL`** resource

[*]Also see the previous section. Note that in your resource definitions, some of the permissions might be public. For example, in a minimal security installation, by default, the USE permission is public for all the Business Intelligence resources.

[**]If a cube contains relationships to other cubes, those cubes are secured separately. A user must have USE permission for all of them in order to use the relationships. Similarly, a compound cube consists of multiple cubes, which are secured separately.

14.4 Adding Security for Model Elements

To add security for a cube, subject area, KPI, pivot table, dashboard, listing, or listing field:

1. Create a resource in the Management Portal. Use the **Resources** page (select **System Administration** > **Security** > **Resources**).
2. Create a role in the Management Portal. Use the **Roles** page (select **System Administration** > **Security** > **Roles**). This role should have USE and WRITE permissions on the resource you just created.

   Or you could create one role with USE and WRITE permissions and another role with only USE permission.
3. Associate the resource with the Business Intelligence item as follows:
   - For a dashboard or pivot table, when you save the item, type the name of the applicable resource into the **Access Resource** field.

     See also "Specifying the Resource for a Dashboard or Pivot Table."

     To save a dashboard or pivot table, you must also have the USE and WRITE privileges for the appropriate Business Intelligence user interface component, as described in the previous heading.
   - For a cube, subject area, or listing field, use the Architect to specify the resource that secures that item.
   - For a listing defined in a cube definition, use the Architect to specify the resource that secures that item.
   - For a listing group or for a listing defined in a listing group, use the Listing Group Manager to specify the resource that secures that item.
   - For a KPI, edit the class definition in Atelier. Use the name of the applicable resource as the value of the *RESOURCE* class parameter.
4. Assign users to roles as needed.

14.5 Specifying the Resource for a Dashboard or Pivot Table

To specify the resource for a dashboard or pivot table, specify the **Access Resource** field when you save the item. You can do this in any of the following cases:

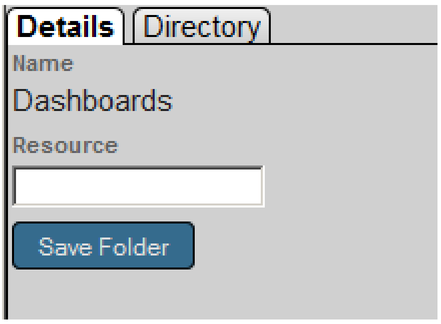
- The item has no owner (specified as the **Owner** field).
- You are the owner of the item.
- You have USE permission on the **`%DeepSee_Admin`** resource.

14.6 Specifying the Resource for a Folder

To specify the resource for a folder:

1. Click the InterSystems Launcher and then click **Management Portal**.

   Depending on your security, you may be prompted to log in with an InterSystems IRIS username and password.
2. Switch to the appropriate namespace as follows:
   a. Click **Switch**.
   b. Click the namespace.
   c. Click **OK**.
3. Click **Analytics** > **Admin** > **Folder Manager**.
4. Click the check box next to a folder.
5. In the left area, click the **Details** tab.



Name
Dashboards
Resource

6. Type the name of the resource.
7. Click **Save Folder**.

A
Using Cube Versions

This appendix describes how to use the cube version feature, which enables you to modify a cube definition, build it, and provide it to users, with only a short disruption of running queries. This appendix discusses the following topics:

- Introduction to this feature
- How to modify a cube to use this feature
- How to update the cube version
- Options for working with a specific cube version
- Additional options specific to the cube version feature

This feature requires twice the amount of disk space, per cube. Also, this feature requires editing the cube class in Atelier.

Note: The cube version feature is not supported for a cube that defines a formally shared dimension. It is also not supported for a cube that defines a one-way relationship; it *can* be used with cubes that define two-way relationships.

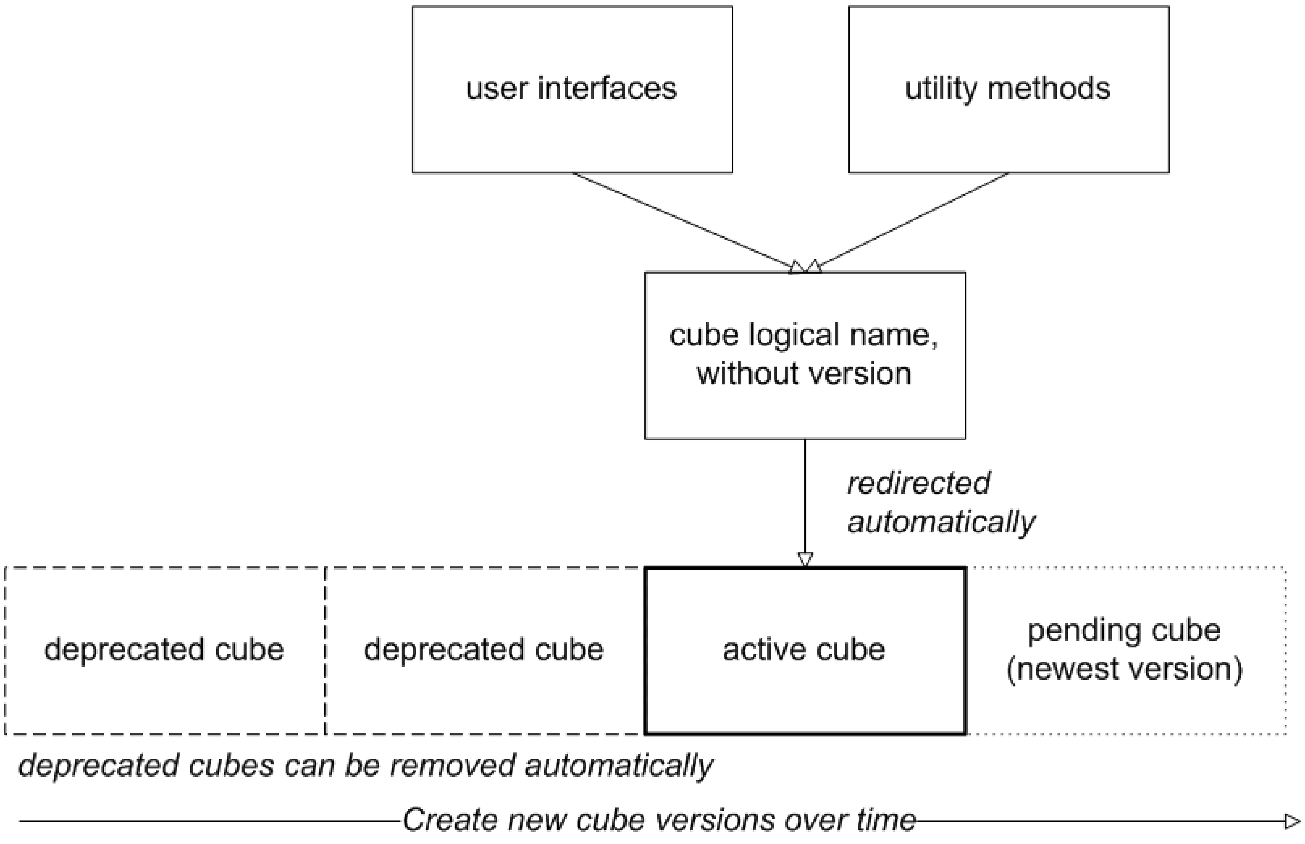
Also see "Accessing the Samples Shown in This Book," in the first chapter.

A.1 Introduction to the Cube Version Feature

The cube version feature enables you to modify a cube definition, build it, and provide it to users, with only a short disruption of running queries. The feature works as follows:

- A given cube definition can have versions.
- The system generates a version-specific fact table and dimension tables for each cube version.
- At any given time, only one cube version is active. The user interfaces and all generated queries use this version.
- To make the newest cube version available, it must be *activated*. At this point, the system momentarily blocks any queries from being run and then switches to the newest version.

The following figure shows the overall process:


user interfaces
utility methods
cube logical name, without version
redirected automatically
deprecated cube
deprecated cube
active cube
pending cube (newest version)
deprecated cubes can be removed automatically
Create new cube versions over time

The cube logical name is redirected automatically to the active cube. The Analyzer and other user interfaces use only the cube logical name and thus see only the active cube. Similarly, if you use methods in %DeepSee.Utils and you specify the cube logical name without a version number, the system runs the method against the active cube.

When you update the cube version number (in Atelier) and recompile, that creates a pending cube, which you can then build. When you are ready, you use a utility method to *activate* the cube, which causes the pending cube to become active and causes the previously active cube to become deprecated.

By default, the activation process automatically deletes the deprecated cube. The cube version feature is not intended to support switching back and forth between versions.

The best practice is to use source control. The cube version feature is not a replacement for source control, but can be helpful in conjunction with it.

A.1.1 Keeping the Cube Current

If a cube uses the cube version feature, you cannot *build* the active version of the cube. That is, the method **$SYSTEM.DeepSee.BuildCube()** does not affect the active version; instead an error is returned. The **Build** option in the Architect behaves the same way. These actions are blocked because they would disrupt running queries for a long time, and the goal of this feature is to prevent that disruption.

You can *synchronize* the cube.

A.1.2 Model Changes Can Break Queries

The cube version feature does not check to ensure that queries that function correctly on the active cube will function correctly on the pending cube. For example, if the pending cube no longer includes a model element that is defined in the active cube, any queries that use that element will not work when you activate the pending cube. It is the customer’s responsibility to identify model changes that could cause disruption and to handle such changes appropriately.

A.2 Modifying a Cube to Support Versions

To modify a cube so that it supports the cube version feature (and to create and activate the initial version):

Important: Read this note if you are making a transition to cube versions *and* you have existing cubes that do not use this feature *and* you do not want any queries to be disrupted.

When you make the transition to cube versions, the process is different for the *first* cube version. Specifically, the first cube version should be runtime-compatible with the cube currently in use (the unversioned cube definition). This means that the first cube version should not remove or redefine any measures or levels, compared to the non-versioned cube definition. It can add elements; that has no effect on existing queries.

1. Add the following parameter to the cube class:

   ```
   Parameter USECUBEVERSIONS=1;
   ```

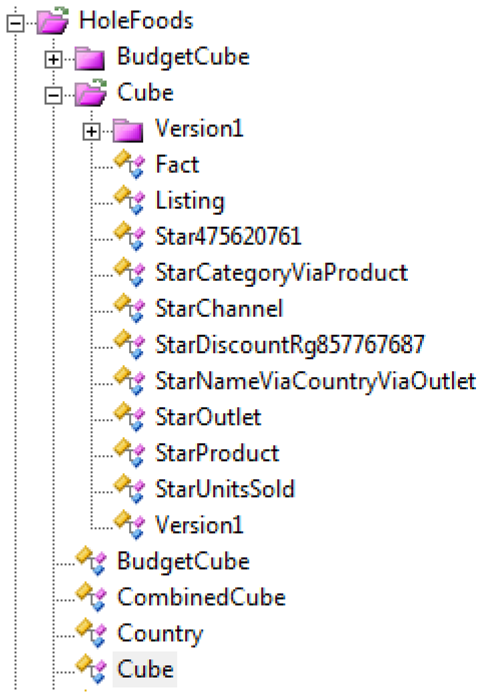
   To make this change and the next, it is necessary to use Atelier.

2. Add the following attribute to the `<cube>` element and then save the class:

   ```
   version="versionnum"
   ```

   Where *versionnum* is an integer.

3. Compile the class. Within the package generated by the system for this cube, there is now a new subpackage (named Version*versionnum*). For example:



   In this example, the new package is HoleFoods.Cube.Version1.

   The classes HoleFoods.Cube.Fact, HoleFoods.Cube.Listing, HoleFoods.Cube.Star475620761, and so on existed previously; these were generated for the cube before *USECUBEVERSIONS* was added. The cube version utilities do not touch these class definitions.

4. Optionally make changes to the cube definition. Read the important note at the start of this section to decide which changes to make. Save your changes.

5. Build the cube. This step does not affect any running queries (nor do the preceding steps, provided that you follow the guidelines in the important note at the start of this section).

   If you build the cube in the Terminal, the system displays slightly different output, to indicate that it is building a specific cube version. For example:

   ```
    Building cube [HOLEFOODS:1]
   ```

6. In the Terminal, execute the **%ActivatePendingCubeVersion()** method of the class %DeepSee.CubeVersion.Utils. This method takes one argument, the name of the cube to build (without any version number). For example:

   ```
    d ##class(%DeepSee.CubeVersion.Utils).%ActivatePendingCubeVersion("holefoods")
   ```

   This method displays output like the following:

   ```
   Pending version for holefoods: 1
   Pending version synchronized: HOLEFOODS:1
   Queries locked for cube: holefoods
   Killing active tasks for cube: holefoods
   Cube version activated: HOLEFOODS:1
   Removing non-versioned cube data
   ```

   One step of this method does briefly prevent queries from being executed against the cube; however, it is likely that users would not experience any actual delay.

   Now all users see the new version of the cube.

7. If you are using the Cube Manager to update this cube, make sure that the update plan for the cube is either **Synch Only** or **Manual**. See "Keeping the Cube Current."

A.2.1 Cube Versions and Relationships

You can use the cube version feature with cubes that are part of relationships. The rules are as follows:

- All relationships must be two-way, rather than one-way.
- Each of the related cubes must also specify a cube version.
- When you update the version, build the new version, and activate the new version for any of the cubes, you must do the same for all the related cubes.
- Activate the related cubes in the same order in which you build them. See "Determining the Build Order for Related Cubes" in *Advanced Modeling for InterSystems Business Intelligence*.

A.2.2 Details for %ActivatePendingCubeVersion()

The **%ActivatePendingCubeVersion()** method has the following signature:

```
ClassMethod %ActivatePendingCubeVersion(pCubeGenericName As %String,
                                        pRemoveDeprecated As %Boolean = 1,
                                        pVerbose As %Boolean = 1) As %Status
```

Where:

- *pCubeGenericName* is the name of the cube, without version number. This argument is not case-sensitive.
- *pRemoveDeprecated* specifies whether the method should also remove the cube version that is now being deprecated. If this argument is 1, the method removes the fact table and its data, dimension tables and their data, any cached data, and any internally used metadata for the cube version that is now being deprecated.

When you use this method for the first time, in the transition from a non-versioned cube, it removes the data stored in the fact table and so on for the non-versioned cube. It does not remove the non-versioned generated classes, which the system needs.

- *pVerbose* specifies whether to display messages indicating the stage of processing of this method.

A.3 Updating a Cube Version

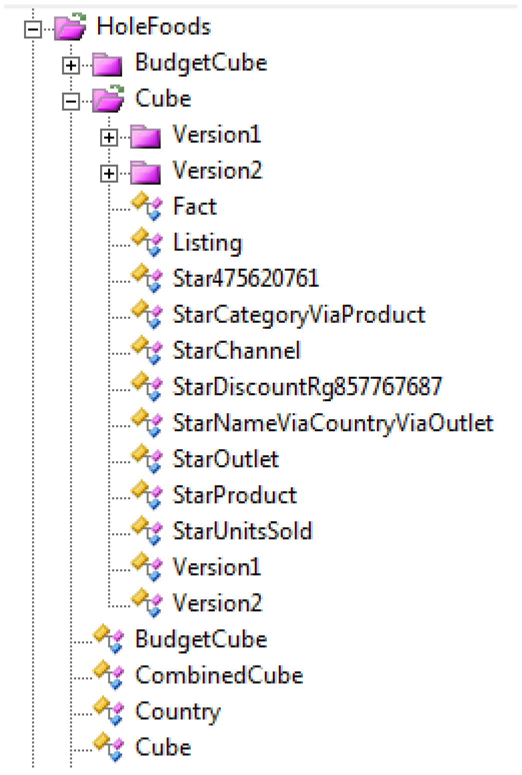
Important: If you have not yet activated the first cube version, see the previous section. When you compile the *first* cube version, any changes to the cube would affect running queries, even before you activate the cube. Therefore it is necessary to compile, build, and activate one version of the cube that is runtime-compatible with the non-versioned cube; see the previous section for what this means.

If you have already modified a cube and created an initial version, use the following process to update the cube:

1. First modify the cube class so that it uses a new version number, in the `<cube>`element. This precaution prevents any cube changes from being visible too early. (Recall that some cube changes, such as to display names, take effect as soon as you compile a cube. See "When to Recompile and Rebuild" in *Defining Models for InterSystems Business Intelligence*.)
2. Save the cube class.
3. Make changes to the cube as wanted and save them.

   Note that for a live system, you should test these changes on a different system first.
4. Compile the cube.

   Within the package generated by the system for this cube, there is now another new subpackage with the new version number. For example:

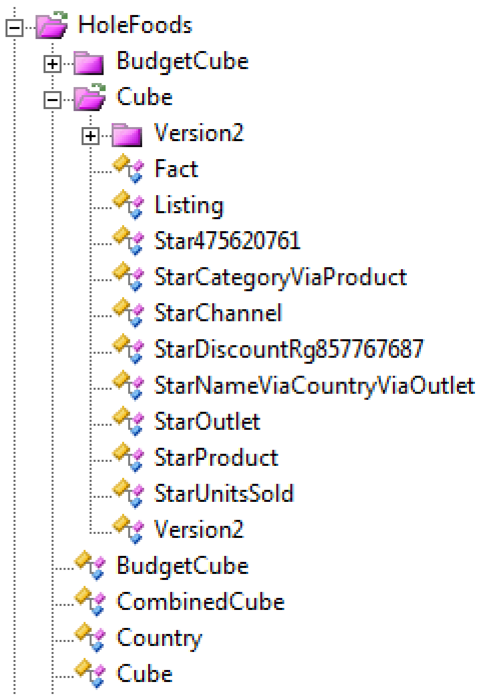


```
HoleFoods
  BudgetCube
  Cube
    Version1
    Version2
    Fact
    Listing
    Star475620761
    StarCategoryViaProduct
    StarChannel
    StarDiscountRg857767687
    StarNameViaCountryViaOutlet
    StarOutlet
    StarProduct
    StarUnitsSold
    Version1
    Version2
  BudgetCube
  CombinedCube
  Country
  Cube
```

5. Build the cube.
6. In the Terminal, execute the **%ActivatePendingCubeVersion()** method of the class %DeepSee.CubeVersion.Utils. In this case, this method displays output like the following:

```
Pending version for holefoods: 2
Pending version synchronized: HOLEFOODS:2
Queries locked for cube: holefoods
Killing active tasks for cube: holefoods
Cube version activated: HOLEFOODS:2
Deprecating previously active version: HOLEFOODS:1
Removing previously active version: HOLEFOODS:1
```

   Within the package generated by the system for this cube, there is now only the subpackage with the new version number. For example:



```
HoleFoods
    BudgetCube
    Cube
        Version2
        Fact
        Listing
        Star475620761
        StarCategoryViaProduct
        StarChannel
        StarDiscountRg857767687
        StarNameViaCountryViaOutlet
        StarOutlet
        StarProduct
        StarUnitsSold
        Version2
    BudgetCube
    CombinedCube
    Country
    Cube
```

Now all users see the new cube.

Note: You can define subject areas based on a cube that uses the versioning feature. As with any change in a base cube, when you change a cube version, you must also recompile the subject area so it will function properly.

A.4 Specifying the Cube to Work With

When you use cube versions, you have the following options for specifying which cube to work with:

- When creating a manual query in the Analyzer or in the MDX Query Tool, you can use either of the following forms of cube name:
  - The logical cube name. In this case, the query uses the active version of the cube.
  - The form *cubename*:*versionnum* where *cubename* is the logical cube name, and *versionnum* is the version number. In this case, the query uses the specified version.

- In the Analyzer, Cube Manager, and other user interfaces, you can work only with the active version, with the exceptions noted in the previous bullet.

  The user interfaces display the cube caption, which contains no information about the version.

  Also, when you save changes, the saved data contains only the logical cube name (that is, without the version number), unless you typed a version number into a manual query. By default, definitions of pivot tables and listing groups do not contain version numbers.

- When you use methods in %DeepSee.Utils that accept a cube name as an argument, you can use either the logical cube name or the form *cubename*:*versionnum*.

- In the MDX shell, you can use either the logical cube name or the form *cubename*:*versionnum*. If tracing is enabled in the shell, the shell displays the cube version number.

A.5 Additional Options

The class %DeepSee.CubeVersion.Utils provides additional methods that you can use for debugging purposes. These *include*:

- **%GetVersionedCubeName()**
- **%DeprecateCubeVersion()**
- **%SetPendingCubeVersion()**
- **%RemoveCubeVersion()**

For details, see the class reference for %DeepSee.CubeVersion.Utils.

Also, the **%BuildCube()** of %DeepSee.Utils can return, by reference, the cube name with the active version number. For example:

```
SAMPLES>set cubename="patients"

SAMPLES>set status=##class(%DeepSee.Utils).%BuildCube(.cubename)

Building cube [PATIENTS:1]
Existing cube deleted.
Fact table built:             1,000 fact(s) (2 core(s) used)
Fact indices built:           1,000 fact(s) (2 core(s) used)

Complete
Elapsed time:                  0.461454s
Source expression time:        0.298187s

SAMPLES>w cubename
PATIENTS:1
```

The method **$SYSTEM.DeepSee.BuildCube()** does not provide this option.

A.5.1 Disabling the Cube Version Feature

To disable versions for a given cube:

1. Modify the cube class and specify *USECUBEVERSIONS* as 0.
2. Save and compile the class.
3. Build the cube.
4. Optionally delete the cube versions that are no longer needed. Execute the following command in the Terminal:

   ```
   set status=##class(%DeepSee.CubeVersion.Utils).%RemoveCubeVersion(cubename,version)
   ```

Where *cubename* is the logical cube name, and *versionnum* is the version number.

This method returns an error if you attempt to remove the active version.

From this point on, the cube behaves the same as a non-versioned cube.

B
How the Analytics Engine Works

This appendix explains how the Analytics Engine executes MDX queries. You may find this information useful when you are viewing query plans or diagnosing problems. This appendix discusses the following topics:

- Introduction
- Steps that the query engine performs
- Details about axis folding
- Query plans
- Query statistics

Important: This appendix provides some information on globals used internally. This information is provided for demonstration purposes; direct use of these globals is not supported. The organization of these globals is subject to change without notice.

Also see "Accessing the Samples Shown in This Book," in the first chapter.

B.1 Introduction

This section introduces the basic concepts. The next section provides a more detailed description.

B.1.1 Use of Bitmap Indices

When you compile a cube class, the Analytics Engine creates the fact table class that the engine uses. This class defines all bitmap indices as needed by the engine; these are stored in the global `^DeepSee.Index`. When you build or synchronize a cube, the engine updates these indices as appropriate. When it is necessary to find records in the fact table, the engine combines and uses these bitmap indices as appropriate.

As an example, one bitmap index provides access to all the records that contribute to the `Snack` member of the `Product Category` level. Another bitmap index provides access to all the records that contribute to the `Madrid` member of the `City` level. Yet another provides access to all the records that contribute to the `2012` member of the `YearSold` level. To find all the records that contribute to `Snack`, `Madrid`, and `2012`, the engine combines those bitmap indices and then uses the resulting index to retrieve the records.

B.1.2 Caching

For any cube that uses more than 512,000 records (by default), the Analytics Engine maintains and uses a result cache. In this case, whenever the engine executes MDX queries, it updates the *result cache*, which it later uses wherever possible. The result cache includes the following globals:

- `^DeepSee.Cache.Results`, which contains values for each query previously executed for a given cube. This global also contains meta-information about those queries that can be used to quickly rerun them. To retrieve information for a query, the engine uses the cube name and the *query key*, which is a hash of the normalized query text.

  For a given cube name and query key, this global includes a set of subnodes that contain final and intermediate values. These subnodes are organized by bucket number and then by result cell. (A bucket is a contiguous set of records in the source table; see the next subsection.)

  The following shows an example:

  ```
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",-1,2,3)=67693.46
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",-1,2,4)=425998.02
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",-1,2,5)=212148.68
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",0,2,3)=301083.77
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",0,2,4)=1815190.08
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",0,2,5)=910314.95
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",1,2,3)=78219.74
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",1,2,4)=463165.12
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",1,2,5)=233031.39
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",2,2,3)=79153.44
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",2,2,4)=461472.97
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",2,2,5)=233584.42
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",3,2,3)=76017.13
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",3,2,4)=464553.97
  ^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",3,2,5)=231550.46
  ```

  In this example, the first subscript after `"data"` indicates the bucket number. Buckets –1 and 0 are special: the –1 bucket is the *active bucket* (representing the most recent records), and the 0 bucket is the consolidated result across all buckets.

  The final subscripts indicate the result cell by position. The value of the node is the value of the given result cell.

  For example, `^DeepSee.Cache.Results("HOLEFOODS","en2475861404","data",0,2,3)` contains the consolidated value for cell (2,3) across all buckets. Notice that this number equals the sum of the intermediate values for this cell, as contained in the other nodes.

- `^DeepSee.Cache.Axis`, which contains metadata about the axes of previously run queries. the engine uses this information whenever it needs to iterate through the axes of a given query. It does not contain cached data.

- `^DeepSee.Cache.Cells`, which contains cached values of measures for cells returned by previously executed queries. A cell is an intersection of any number of non-measure members (such as the intersection of Madrid, Snack, and 2012). In this global, each cell is represented by a *cell specification*, which is a specialized compact internal-use expression. The following shows a partial example:

  ```
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::1:1",1)=$lb(1460.05)
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::1:2",1)=$lb(606.22)
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::1:3",1)=$lb(40.17)
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::1:4",1)=$lb(63.72)
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::2:1",1)=$lb(3778)
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::2:2",1)=$lb(1406.08)
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::2:3",1)=$lb(117.31)
  ^DeepSee.Cache.Cells("HOLEFOODS",1,":::2012:::::1:1::2:4",1)=$lb(412.24)
  ```

  The first subscript is the cube name, the second is the bucket number, the third is the cell specification (`":::2012:::::1:1::1:1"` for example), and the last indicates the measure. The value of a given node is the aggregate value of the given measure for the given cube, cell, and bucket. In this case, the results are expressed in $LISTBUILD form for convenience in internal processing. Notice that this global does not use the query key; this is because the same cell could easily be produced by multiple, quite different queries.

This global is known as the *cell cache* and is populated only when the cache uses buckets.

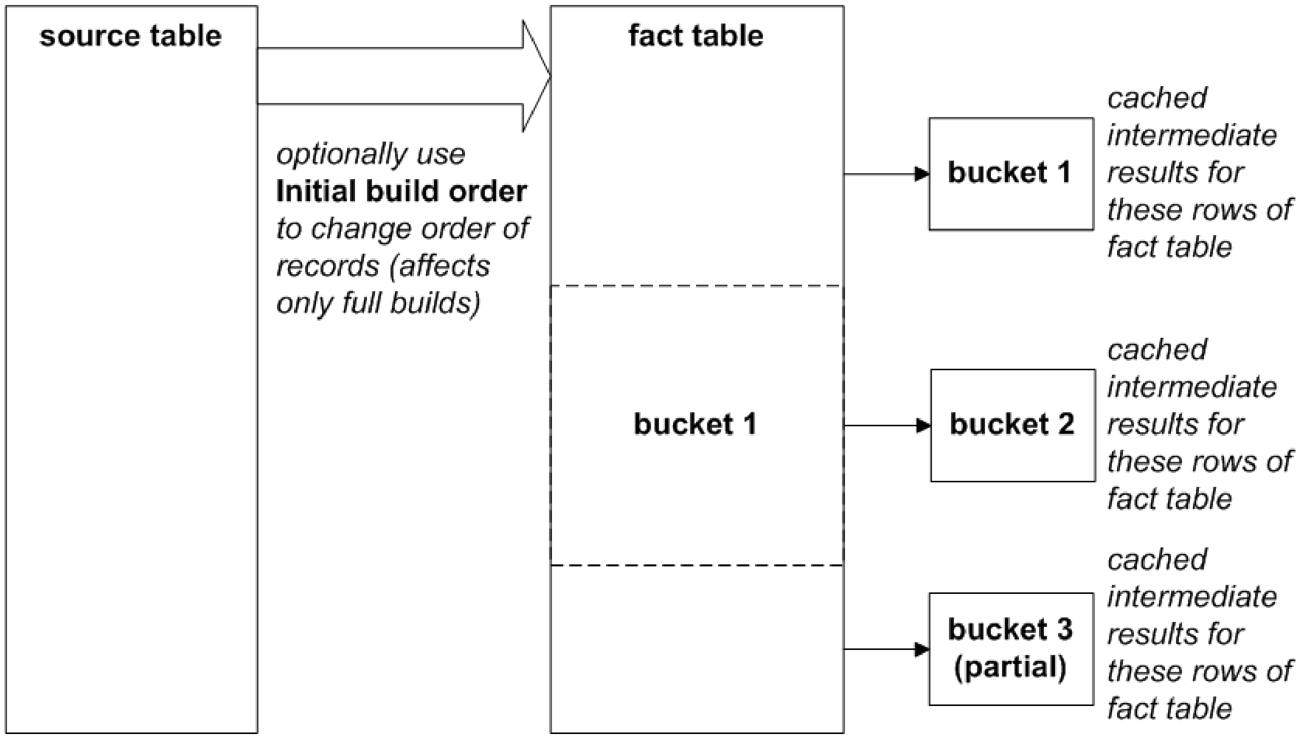
The cell cache does not include values for the active bucket. Nor does it include values for the 0 bucket (consolidated across all buckets).

These globals are not populated until users execute queries. The cache grows in size as more queries are executed, resulting in faster performance because the engine can use the cache rather than re-executing queries.

Note that the cache does not include values for any properties defined with `isReference="true"`. These values are always obtained at runtime.

B.1.3 Buckets

For any cube that uses more than 512,000 records (by default), the engine organizes the cache into *buckets*. Each bucket corresponds to a large number of contiguous records in the fact table, as shown in the following figure:


source table
optionally use Initial build order to change order of records (affects only full builds)
fact table
bucket 1
bucket 1
cached intermediate results for these rows of fact table
bucket 2
cached intermediate results for these rows of fact table
bucket 3 (partial)
cached intermediate results for these rows of fact table

The final bucket (or partial bucket) is the active bucket and is not represented in the cell cache.

By default, the fact table contains records in the same order as the source table. You can specify **Initial build order** for the cube to control the order in which the engine examines the source table records when it performs a full build of the cube; see "Other Cube Options" in *Defining Models for InterSystems Business Intelligence*.

Whenever the fact table is updated, the engine discards parts of the cache as appropriate. More specifically, the engine invalidates any buckets that use records from the affected part or parts of the fact table. Other buckets are left alone. When it executes a query, the engine uses cached data only for the valid buckets. For records that do not have valid cached results, the engine uses the bitmap indices and recomputes the needed intermediate values. As the last phase of query execution, the engine consolidates the results. Thus the engine can provide results that come from a combination of cached data and new or changed data. Also, because some of the engine work can be split by bucket, the engine can (and does) perform some processing in parallel.

B.1.3.1 Default Bucket Size

By default, a bucket is 512,000 records. The bucket size is controlled by the `bucketSize` option, which expresses the bucket size as an integer number of *groups* of records, where a group is 64,000 contiguous records. The default `bucketSize` is 8, so that the default bucket is 8 x 64,000 records or 512,000 records. For information on `bucketSize`, see "<cube>" in *Defining Models for InterSystems Business Intelligence*.

B.2 Engine Steps

To process an MDX query, the Analytics Engine performs the following steps:

1. *Preparation*, which occurs in process (that is, this step is not launched as a background process). In this phase:

   a. The engine parses the query and converts it to an object representation, the *parse tree*.

      In the parse tree, each axis of the query is represented separately. One axis represents the overall filtering of the query.

   b. The engine converts the parse tree to a normalized version of the query text.

      In this normalized version, for example, all %FILTER clauses have been combined into a single, equivalent WHERE clause.

   c. The engine generates a hash that is based on the normalized query text. the engine uses this hash value as the *query key*. The query key enables the engine to look up results for this query in the globals discussed in this appendix.

   d. If the engine finds that it is possible to reuse previous results for this query (from `^DeepSee.Cache.Results`), the engine does so and skips the following steps.

2. *Execute axes*, which also occurs in process. In this phase:

   a. The engine executes any subqueries.

   b. The engine examines the slicer axis (the WHERE clause), merges in any relevant filtering (such as from a subject area filter), and updates `^DeepSee.Cache.Axis` with information about this axis.

   c. The engine examines each of the remaining axes and updates `^DeepSee.Cache.Axis`.

3. *Execute cells*, which occurs in the background (in multiple parallel processes). In this phase, the engine obtains intermediate values for each cell of the results, separately for each bucket, as follows:

   a. First the engine checks to see if `^DeepSee.Cache.Cell` contains a value for the cell for the given bucket.

      If so, the engine uses that value.

   b. Otherwise, the engine uses the applicable nodes of `^DeepSee.Index` to obtain the bitmap indices that it needs. The engine combines these bitmap indices and then uses the result to find the applicable records in the source table.

      If the cache uses buckets, the engine adds nodes to `^DeepSee.Cache.Cell` for use by later queries.

4. *Consolidation*, which occurs in process. In this phase:

   a. For each slicer axis, the engine examines each result cell for that axis.

      For each result cell, the engine finds all the nodes in `^DeepSee.Cache.Cell` that contain values for this cell.

      It then combines those values.

   b. For each result cell, the engine then combines the results across the slicer axes and obtains a single value.

For information, see the next section.

The engine evaluates the CURRENTMEMBER function during the consolidation phase. In contrast, it evaluates other functions earlier in the processing.

B.3 Axis Folding

In the consolidation phase, if there are multiple slicer axes, the Analytics Engine combines results across these axes, for each result cell. This step is known as *axis folding*.

Important: Axis folding means that if a given source record has a non-null result for each slicer axis, that record is counted multiple times.

To determine whether axis folding is required, the engine considers all the filters applied to the query, from all sources: the subject area, the pivot table, and the dashboard. The net combination of these filters determines whether axis folding is needed, as follows. The following table lists the main possibilities:

Form of Filter	Axis Folding Performed?
Single member. Example: `[PRODUCT].[P1].[PRODUCT CATEGORY].&[Candy]`	No
Single measure. Example: `[MEASURES].[Units Sold]`	No
A tuple (combination of members or of members and a measure). Example: `([Outlet].[H1].[City].&[7],[PRODUCT].[P1].[PRODUCT CATEGORY].&[Candy])`	No
Cross joins that use members wrapped in **%TIMERANGE** functions `CROSSJOIN(%TIMERANGE([BirthD].[H1].[Date].&[1980],[BirthD].[H1].[Date].&[5980]),%TIMERANGE([BirthD].[H1].[Date].&[4980],[BirthD].[H1].[Date].&[NOW]))`	Yes
Other cross joins. Example: `NONEMPTYCROSSJOIN([Outlet].[H1].[City].&[7],[PRODUCT].[P1].[PRODUCT CATEGORY].&[Candy])`	No
The %OR function, wrapped around a set expression that lists multiple members. Example: `%OR({[Product].[P1].[Product Category].&[Candy],[Product].[P1].[Product Category].&[Snack]})`	No
A set expression that lists multiple members but does not use %OR. Example: `{[Product].[P1].[Product Category].&[Candy],[Channel].[H1].[Channel Name].&[2]}`	Yes

To create these expressions (as filters) in the Analyzer, you generally drag and drop items to the **Filters** box. To create the set expressions in the last two rows, you must use the Advanced Filter editor. Note that the engine automatically uses the %OR function when possible; the Advanced Filter editor does not display it as an option.

B.4 Query Plans

If you execute a query in the MDX Query Tool, you can see the query plan. Similarly, if you execute a query programmatically (as described earlier in this book), you can call the **%ShowPlan()** method of your result set. For example:

```
SAMPLES>do rs1.%ShowPlan()
-------------- Query Plan ---------------------
**SELECT {[MEASURES].[AVG TEST SCORE],[MEASURES].[%COUNT]} ON 0,[AGED].[AGE
BUCKET].MEMBERS ON 1,[GEND].[GENDER].MEMBERS ON 2 FROM [PATIENTS]****
DIMENSION QUERY (%GetMembers): SELECT %ID,DxAgeBucket MKEY, DxAgeBucket
FROM BI_Model_PatientsCube.DxAgeBucket ORDER BY DxAgeBucket**
**DIMENSION QUERY (%GetMembers): SELECT %ID,DxGender MKEY, DxGender
FROM BI_Model_PatientsCube.DxGender ORDER BY DxGender**
**EXECUTE: 1x1 task(s) **
**CONSOLIDATE**
-------------- End of Plan -----------------
```

Note that line breaks and spaces have been added here to format the documentation properly for its PDF version.

B.5 Query Statistics

If you execute a query programmatically (as described earlier in this book), you can call the **%PrintStatistics()** method of your result set. For example:

```
SAMPLES>do rs1.%PrintStatistics()
Query Statistics:
 Results Cache:                     0
 Query Tasks:                       1
 Computations:                     15
 Cache Hits:                        0
 Cells:                            10
 Slices:                            0
 Expressions:                       0

 Prepare:                       0.874 ms
 Execute Axes:                145.762 ms
  Columns:                      0.385 ms
  Rows:                       144.768 ms
   Members:                   134.157 ms
 Execute Cells:                 6.600 ms
 Consolidate:                   1.625 ms
 Total Time:                  154.861 ms

ResultSet Statistics:
 Cells:                             0
 Parse:                         3.652 ms
 Display:                       0.000 ms
 Total Time:                    3.652 ms
```

The values shown here are as follows:

- `Query Statistics` — This group of statistics gives information about the query, which returned a result set. It does not include information on what was done to use that result set.
  - `Results Cache` is 1 if the results cache was used or is 0 otherwise.
  - `Query Tasks` counts the number of tasks into which this query was divided.
  - `Computations` indicates how much time was spent performing intermediate computations such as aggregating a measure according to its aggregation option. It does not include evaluating MDX expressions.
  - `Cache Hits` counts the number of times an intermediate cache was used.
  - `Cells` counts all the cells of the result set as well as any intermediate cells that were computed.
  - `Slices` counts the number of cube slices in the query. This count indicates the number of items on the WHERE clause.
  - `Expressions` indicates how much time was spent evaluating MDX expressions.

    When the cache is used, `Computations`, `Cache Hits`, `Cells`, and `Expressions` are all zero.
  - `Prepare`, `Execute Axes`, `Execute Cells`, and `Consolidate` indicate how long different parts of the query processing took place. These parts are listed in order.

  - `Total Time` is the sum of those parts.

    When the cache is used, `Execute Cells` and `Consolidate` are both zero, because those parts of the processing are not performed.

- `ResultSet Statistics` — This group of statistics gives information about what was done to use the result set after it was returned by the result set. The values are as follows:
  - `Cells` counts the number of cells in the result set.
  - `Parse` indicates how long it took to parse the result set.
  - `Display` indicates how long it took to display it.
  - `Total Time` is the sum of those times.

C

Using the MDX Performance Utility

The system provides a tool, the %DeepSee.Diagnostic.MDXUtils class, to enable you to gather query statistics and lower-level performance statistics at the same time. This class provides the **%Run()** method:

```
classmethod %Run(pMDX As %String = "",
                 pBaseDir As %String = "",
                 pVerbose As %Boolean = 0,
                 ByRef pParms="",
                 Output pOutFile="") as %Status
```

Given an MDX query, this method prepares and runs the query and generates files that contain diagnostic information about that query. The arguments are as follows:

- *pMDX* — Specifies the MDX query.
- *pBaseDir* — Specifies the base directory to which the output directory (`MDXPerf`) is written. The default base directory is the installation directory.
- *pVerbose* — Specifies whether to invoke routines in verbose mode. Use 1 for yes, or 0 (the default) for no.
- *pParms* — Specifies a multidimensional array of parameters. This array can have the following nodes:
  - *pParms*`("CubeStats")` — Specifies whether to generate cube statistics. Use 1 (the default) for yes, or 0 for no.
  - *pParms*`("TimePERFMON")` — Specifies how long, in seconds, to collect data via **^PERFMON**. Specify a positive integer; the default is 15. For details, see "Monitoring Performance Using ^PERFMON" in the *Monitoring Guide*.
  - *pParms*`("pButtonsOn")` — Specifies whether to also generate a **^SystemPerformance** report. Use 1 for yes, or 0 (the default) for no.
  - *pParms*`("pButtonsProfile")` — Specifies the name of the **^SystemPerformance** profile to use. For details, see "Monitoring Performance Using ^SystemPerformance" in the *Monitoring Guide*.
- *pOutFile* — Returned as an output parameter, this argument specifies the name of the main report HTML file generated by this method.

The **%Run()** method generates the following files:

- MDXPerf_*nnnnn_nnnnn*.html — Main HTML report file. This contains query statistics, the query plan, and so on.
- *cubename*.xml — Definition of the given cube.
- Cached_MDXPerf_*cubename_nnnnn_nnnnn*.html — **^PERFMON** timed collection report for running the query when using the result cache.

  For details, see "Monitoring Performance Using ^PERFMON" in the *Monitoring Guide*.

- Uncached_MDXPerf_*cubename_nnnnn_nnnnn*.html — **^PERFMON** timed collection report for running the query when not using the result cache.

  Note that the engine creates a result cache only for a cube that uses more than 512,000 records (by default), so this report could have the same numbers as Cached_MDXPerf_*cubename_nnnnn_nnnnn*.html.

- *hostname_date_time*.html — **^SystemPerformance** report.

  For details, see "Monitoring Performance Using ^SystemPerformance" in the *Monitoring Guide*.

- Other files generated by **^SystemPerformance**. These vary by operating system.

D

Other Export/Import Options

This appendix describes additional options for exporting and importing Business Intelligence elements, as a supplement to the chapter "Packaging Business Intelligence Elements into Classes." It discusses the following topics:

- How to create a container class in Atelier
- How to export and import folder items via the older API

Note: This appendix assumes that you are familiar with the process of exporting from and importing into Atelier.

Also see "Accessing the Samples Shown in This Book," in the first chapter.

D.1 Creating a Business Intelligence Container Class

As noted in the chapter "Packaging Business Intelligence Elements into Classes,"you can package pivot tables and other folder items into InterSystems IRIS® classes. You can package as many elements as needed into a single class, which is easier to export and import than many separate files.

To create such a class:

- The class must extend %DeepSee.UserLibrary.Container.
- The class must include an XData block named `Contents`. For this XData block, you must specify the XML namespace as follows:

  ```
  [ XMLNamespace = "http://www.intersystems.com/deepsee/library" ]
  ```

- The top-level element within the XData block must be `<items>`.

Include as many XML definitions as needed within `<items>`. You can copy the definitions in Atelier or from exported XML files. Also see the next section, which describes edits you should make.

Also be sure to copy and paste only the definition, not the XML declarations at the top of the file. That is, do not copy the following line into the XData block:

```
<?xml version="1.0" encoding="UTF-8"?>
```

For example:

```
Class BI.Model.DashboardItems Extends %DeepSee.UserLibrary.Container
{

XData Contents [ XMLNamespace = "http://www.intersystems.com/deepsee/library" ]
{
<items>
<dashboard dashboard definition here ...
</dashboard>
<dashboard another dashboard definition here ...
</dashboard>
<pivot pivot definition here ...
</pivot>
<pivot another pivot definition here ...
</pivot>
<pivot yet another pivot definition here ...
</pivot>
</items>
}

}
```

When you compile this class or when you call its **%Process()** instance method, the system creates the items defined in the XData block. Specifically, it imports these definitions into the internal global that the User Portal uses.

The same class can also define the **%OnLoad()** callback, which can execute any additional code needed when these items are set up.

For samples of pivot tables and dashboards that are packaged into class definitions, see the sample classes BI.DashboardsEtc and HoleFoods.DashboardsEtc.

If you delete a container class, that has no effect on the pivots and dashboards that currently exist.

D.2 Exporting and Importing Folder Items

This section describes the older API for exporting and importing folder items.

D.2.1 Exporting Folder Items Programmatically

To export folder items programmatically, use the following command:

```
Do ##class(%DeepSee.UserLibrary.Utils).%Export(itemname,filename)
```

Where:

- *itemname* is the full name of the item, including the folder in which it belongs.
  - For a pivot table, append the extension `.pivot`
  - For a dashboard, append the extension `.dashboard`
  - For a widget, append the extension `.widget`
  - For a theme, append the extension `.theme`
- *filename* is the full path and file name of the file to create. InterSystems suggests that you end the file name with `.xml`, because the file is an XML file.

For example:

```
set DFIname="Chart Demos/Area Chart.pivot"
set filename="c:/test/Chart-Demos-Area-Chart-pivot.xml"
do ##class(%DeepSee.UserLibrary.Utils).%Export(DFIname,filename)

set DFIname="KPIs & Plugins/KPI with Listing.dashboard"
set filename="c:/test/KPIs-Plugins-KPI-with-Listing-dashboard.xml"
do ##class(%DeepSee.UserLibrary.Utils).%Export(DFIname,filename)
```

D.2.1.1 Alternative Technique (for Exporting Multiple Items)

To export multiple items programmatically into a single XML file, use the **$system.OBJ.Export()** method. The first and second arguments for this method are as follows:

- *items* is a multidimensional array as follows:

Array Node	Node Value
items("*full-folder-item-name.DFI*") where *items* is the name of the array and *full-folder-item-name.DFI* is the full name of the folder item, exactly as seen in Atelier, including case.	""

  Note that because this argument is a multidimensional array, you must precede it with a period when you use the **$system.OBJ.Export()** method.

- *filename* is the full path and file name of the file to create. InterSystems suggests that you end the file name with `.xml`, because the file is an XML file.

For example:

```
set items("Chart Demos-Area Chart.pivot.DFI")=""
set items("Chart Demos-Bar Chart.pivot.DFI")=""
set items("Chart Demos-Bubble Chart.pivot.DFI")=""
set filename="c:/test/Chart-Samples.xml"
do $system.OBJ.Export(.items,filename)
```

You can also use this method to export other items such as classes; for details, see the Class Reference for %SYSTEM.OBJ.

D.2.2 Importing Folder Items Programmatically

To import folder items programmatically:

```
Do ##class(%DeepSee.UserLibrary.Utils).%Import(pFile, pReplace, pVerbose)
```

Where:

- *pFile* is the full path and file name of the file to import.
- If *pReplace* is true, replace an existing item with the same name. The default is false.
- If *pVerbose* is true, write status to the console. The default is true.

For example:

```
set filename="c:/test/Chart-Demos-Area-Chart-pivot.xml"
do ##class(%DeepSee.UserLibrary.Utils).%Import(filename,1,1)
```

E

Business Intelligence and Disaster Recovery

This appendix describes the recommended procedure for write-protecting copied source data on an async mirror member using Business Intelligence. It discusses the following topics:

- Configuration
- Disaster Recovery

E.1 Configuration

This section describes the necessary initial configuration tasks.

1. Set up the async mirror as a disaster recovery (DR) async with all source data databases and the newly-mapped database for *^OBJ.DSTIME*. This will perform more validation of the system and push any issues with the ISCAgent and so on to configuration time instead of recovery time. Note that this mode does not allow for a read-write database.
2. Once configured, switch the DR to a read-only async member.
3. On a read-only async, each specific database has a `ReadOnly` flag that can be cleared, allowing writes. Do this for the database containing *^OBJ.DSTIME*.

The source data is now write-protected and the cubes can be synchronized properly.

E.2 Disaster Recovery

This section describes the steps to take during disaster recovery.

1. Remove the database containing *^OBJ.DSTIME* from the mirror configuration. Note that the database is still available.
2. Switch the async member back to a DR member.
3. Promote the member to primary.
4. Synchronize cubes.

The *^OBJ.DSTIME* buffer needs to be treated as out-of-date on any other systems that may now be relying on this one, as there will be no attempt to synchronize that data with other async members. The database containing *^OBJ.DSTIME* needs to be added back into the mirror set as part of the recovery procedure.