# Content

1. Document History .......................................................... 4
2. Preface ........................................................................... 5
   2.1 What can you learn from this guide? .......................... 5
   2.2 Who should use this guide? ...................................... 5
   2.3 What should you know? ........................................... 5
   2.4 Related documents .................................................. 5
   2.5 List of APIs not functional from the SAP BusinessObject XI 4.0 release onwards .............................................. 6
      Classes and interfaces no longer functional .................. 6
3. Introduction to the ReportEngine SDK ................................ 9
   3.1 Packages for customizing SAP BusinessObjects Enterprise ................................................................. 9
   3.2 ReportEngine SDK in SAP BusinessObjects Enterprise environment .................................................. 9
4. Workflow of ReportEngine SDK application .......................... 10
   4.1 Application essentials ............................................. 10
   4.2 Viewing reports .................................................... 10
   4.3 Open document workflow ...................................... 10
   4.4 Refresh document workflow ................................... 10
5. Setting up development environment ................................... 12
   5.1 Setting up the Java environment ................................ 12
      Setting up the Eclipse environment ......................... 12
      Project Setup ..................................................... 12
   6.1 Overview ............................................................. 15
   6.2 Working with the ReportEngine SDK ......................... 15
      Referencing ReportEngine SDK ............................. 16
      Closing a ReportEngines object ........................... 16
   6.3 Document Management - Overview .......................... 16
      Document types ................................................. 16
      Document state: storage tokens ............................. 17
      Working with documents ..................................... 18
      Document properties ......................................... 21
   6.4 Viewing reports - Overview .................................... 22
      Viewing reports ................................................ 22
      Handling prompts ............................................. 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Drilling in Web Intelligence Reports - Overview</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Introduction to drilling</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Drilling in reports</td>
<td>42</td>
</tr>
<tr>
<td>6.6</td>
<td>Working with Recordsets - Overview</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Anatomy of a recordset</td>
<td>47</td>
</tr>
<tr>
<td>7</td>
<td>Best Practices to improve ReportEngine performance</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>Best Practices</td>
<td>49</td>
</tr>
<tr>
<td>7.2</td>
<td>Developer tips</td>
<td>50</td>
</tr>
</tbody>
</table>
### 1 Document History

The following table provides an overview of the most important document changes:

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>November 2010</td>
<td>First release of the document.</td>
</tr>
<tr>
<td></td>
<td>April 2012</td>
<td>The following chapters have been removed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Organizing documents in categories and folders</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Displaying documents lists</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Refreshing a document list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Scheduling documents</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Sending documents to users, groups and categories</td>
</tr>
<tr>
<td></td>
<td></td>
<td>see Business Intelligence platform Java SDK developer guide at the link reference below.</td>
</tr>
</tbody>
</table>

### Related Information

2  Preface

The ReportEngine Java developer guide helps you to customize web application using Java SDK.

2.1  What can you learn from this guide?

This guide describes the processes and procedures for creating web applications using SAP BusinessObjects Business Intelligence platform SDK and the ReportEngine SDK.

2.2  Who should use this guide?

This guide is intended for developers who are interested in developing web applications using SAP BusinessObjects Business Intelligence platform SDK and the ReportEngine SDK.

2.3  What should you know?

To use the SAP BusinessObjects Business Intelligence platform SDK and ReportEngine SDK, you need a working knowledge of the concepts of distributed computing systems, HTTP server technology, and JSP development. You also need a basic knowledge of the products in the SAP BusinessObjects product line. Familiarity with SAP BusinessObjects Business Intelligence is assumed.

2.4  Related documents

For more information on classes and interfaces of SAP BusinessObjects Business Intelligence platform SDK and ReportEngine SDK, refer to the following guides:

- Business Intelligence platform Java API Reference
- Business Intelligence platform Java SDK Developer Guide
- Web Intelligence ReportEngine API Reference guide
2.5 List of APIs not functional from the SAP BusinessObject XI 4.0 release onwards

In SAP BusinessObjects XI 4.0 release, the Report Engine SDK does not support the following features:

- A subset of drilling features in Web Intelligence documents
- Building queries
- Document and report editing

2.5.1 Classes and interfaces no longer functional

The following tables list the set of classes/interfaces that are no longer functional from the SAP BusinessObjects XI 4.0 release onwards

Table 2: Classes and Interfaces from A to D

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdvancedCondition</td>
<td>BarType</td>
<td>Calculation</td>
<td>DataProvider (partially supported)</td>
</tr>
<tr>
<td>AggregationFunction</td>
<td>Block</td>
<td>CalculationElement</td>
<td>DataProviders (partially supported)</td>
</tr>
<tr>
<td>Alertable</td>
<td>BlockAxis</td>
<td>Cell</td>
<td>DataSource (partially supported)</td>
</tr>
<tr>
<td>Alerter</td>
<td>BlockBreak</td>
<td>CellContentType</td>
<td>DataSourceObject</td>
</tr>
<tr>
<td>Alerters</td>
<td>BlockCalculation</td>
<td>CellMatrix</td>
<td>DataSourceObjectPropertiesType</td>
</tr>
<tr>
<td>AlertersDictionary</td>
<td>BlockInfo</td>
<td>ChangeDataSourceMapping</td>
<td>DataSourceObjects</td>
</tr>
<tr>
<td>Alignment</td>
<td>Blocks</td>
<td>ChangeDataSourceMappingDetail</td>
<td>DataSourceObjectUsage</td>
</tr>
<tr>
<td>AllAvailableFunctionsIdentifiers</td>
<td>BlockSort</td>
<td>ChangeDataSourceObjectMapping</td>
<td>Decoration (partially supported)</td>
</tr>
<tr>
<td>AllAvailableOperatorsIdentifiers</td>
<td>BlockType</td>
<td>CombinedQueryOperator</td>
<td>DefaultUnit</td>
</tr>
<tr>
<td>AreaType</td>
<td>Border</td>
<td>ComparisonOperator</td>
<td>DocumentInstance (partially supported)</td>
</tr>
<tr>
<td>Attachable</td>
<td>BorderSize</td>
<td>ComplexBorder</td>
<td>DExpression</td>
</tr>
<tr>
<td>Attributes</td>
<td>BreakElement</td>
<td>ConditionContainer</td>
<td>DrillBar</td>
</tr>
<tr>
<td>AxisType</td>
<td>ConditionObject</td>
<td>DrillBarObject</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConditionObjectValue</td>
<td>DrillBlock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConditionOperator</td>
<td>DrillDetail</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConditionOperatorFeature</td>
<td>DrillDetails</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3: Classes and Interfaces from F to P

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>G - H - I - L</td>
<td>M - P</td>
<td></td>
</tr>
<tr>
<td>Filterable</td>
<td>Graph</td>
<td>Margins</td>
<td></td>
</tr>
<tr>
<td>FilterCondition</td>
<td>GraphAxis</td>
<td>ObjectQualification</td>
<td></td>
</tr>
<tr>
<td>FilterConditionConstant</td>
<td>GraphAxisProperties</td>
<td>ObjectType</td>
<td></td>
</tr>
<tr>
<td>FilterConditionContainer</td>
<td>GraphAxisValues</td>
<td>OpenDocumentInfo</td>
<td></td>
</tr>
<tr>
<td>FilterConditionNode</td>
<td>GraphData</td>
<td>Operator</td>
<td></td>
</tr>
<tr>
<td>FilterConditionObject</td>
<td>GraphGrid</td>
<td>Orientation</td>
<td></td>
</tr>
<tr>
<td>FilterConditionOperand</td>
<td>GraphTitle</td>
<td>OutputCacheEntry</td>
<td></td>
</tr>
<tr>
<td>FilterConditionValue</td>
<td>GraphType</td>
<td>PageHeaderFooter</td>
<td></td>
</tr>
<tr>
<td>FilterContainer</td>
<td>HAlignmentType</td>
<td>PageInfo</td>
<td></td>
</tr>
<tr>
<td>FilterObject</td>
<td>HAnchorType</td>
<td>PageLayout</td>
<td></td>
</tr>
<tr>
<td>Font (partially supported)</td>
<td>HTMLView (partially supported)</td>
<td>PaperSize</td>
<td></td>
</tr>
<tr>
<td>FontOrientation</td>
<td>HZoneType</td>
<td>PersonalDataProvider</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>ImageDisplayMode</td>
<td>PieType</td>
<td></td>
</tr>
<tr>
<td>FormulaExpression</td>
<td>Legend</td>
<td>Podium</td>
<td></td>
</tr>
<tr>
<td>FormulaOperator</td>
<td>LegendPosition</td>
<td>Position</td>
<td></td>
</tr>
<tr>
<td>FreeCell</td>
<td>LineType</td>
<td>PromptOrder</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FunctionCategory</td>
<td>LogicalOperator</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4: Classes and Interfaces from Q to V

<table>
<thead>
<tr>
<th>Q - R</th>
<th>S</th>
<th>T - U - V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>SamplingMode</td>
<td>Table</td>
</tr>
<tr>
<td>QueryContainer</td>
<td>Scope</td>
<td>TableAxis</td>
</tr>
<tr>
<td>QueryException</td>
<td>ScopeLevel</td>
<td>TableCell</td>
</tr>
<tr>
<td>QueryFeature</td>
<td>SectionContainer</td>
<td>TableFormBase</td>
</tr>
<tr>
<td>QueryNode</td>
<td>SimpleBorder</td>
<td>TableType</td>
</tr>
<tr>
<td>QuerySort</td>
<td>SimpleTable</td>
<td>Unit</td>
</tr>
<tr>
<td>Q - R</td>
<td>S</td>
<td>T - U - V</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>RadarType</td>
<td>Skin</td>
<td>UnitType</td>
</tr>
<tr>
<td>RankCondition</td>
<td>SkinReportElementType</td>
<td>VAlignmentType</td>
</tr>
<tr>
<td>RankConditionPrompt</td>
<td>SortElement</td>
<td>VAnchorType</td>
</tr>
<tr>
<td>Report (partially supported)</td>
<td>SortInfo</td>
<td>VariableExpression</td>
</tr>
<tr>
<td>ReportBlock</td>
<td>SortInfoType</td>
<td>VZoneType</td>
</tr>
<tr>
<td>ReportBody</td>
<td>SortType</td>
<td></td>
</tr>
<tr>
<td>ReportCell</td>
<td>SQLContainer</td>
<td></td>
</tr>
<tr>
<td>ReportContainer</td>
<td>SQLDataProvider</td>
<td></td>
</tr>
<tr>
<td>ReportDictionary</td>
<td>SQLNode</td>
<td></td>
</tr>
<tr>
<td>ReportElement</td>
<td>SQLSelectStatement</td>
<td></td>
</tr>
<tr>
<td>ReportElementContainer</td>
<td>SynchroManager</td>
<td></td>
</tr>
<tr>
<td>ReportEngine (partially sup-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ported)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReportExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReportInfo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReportStructure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Representation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3 Introduction to the ReportEngine SDK

The ReportEngine SDK is used to view Web Intelligence. The ReportEngine SDK is considered as a component of SAP BusinessObjects Business Intelligence SDK as they are designed to work together.

You can use Java API interfaces to create web portals that access the back-end SAP BusinessObjects BI server.

3.1 Packages for customizing SAP BusinessObjects Enterprise

There are two packages for customizing SAP BusinessObjects Business Intelligence platform:

- SAP BusinessObjects BI platform SDK
  This package is called the platform package and provides platform functions of the SAP BusinessObjects BI servers and document scheduling.

- ReportEngine SDK
  This package plugs into SAP BusinessObjects BI platform SDK. It provides functionalities of Web Intelligence ReportEngine.

3.2 ReportEngine SDK in SAP BusinessObjects Enterprise environment

SAP BusinessObjects Business Intelligence Framework is a framework of servers that facilitates enterprise-wide, web-based management and distribution of reports. Web applications can access the functionality of SAP BusinessObjects BI through APIs provided by the SAP BusinessObjects Business Intelligence SDK and ReportEngine SDK. For detailed information on SAP BusinessObjects BI, see the Business Intelligence Java SDK Developer Guide.

The diagram below illustrates the relationship between the ReportEngine SDK and other SAP BusinessObjects BI platform components.
4 Workflow of ReportEngine SDK application

All applications that use ReportEngine SDK do not have the same features. But all typically provide features for listing and viewing documents contained in the SAP BusinessObjects BI Central Management System (CMS).

4.1 Application essentials

Every application that uses ReportEngine SDK has to perform common tasks such as importing the appropriate packages, and creating and managing the user’s SAP BusinessObjects BI session (login).

Every application has to take user profiles into account and ensure that errors are handled properly.

4.2 Viewing reports

An application that allows to view documents can display them using the default DHTML format. It is easy to extend the functions to handle documents that contain prompts, and to provide drilling functions. These extended functions are handled when a document is refreshed.

The viewing features depends on the type of document opened (Web Intelligence, Adobe Acrobat, Microsoft Excel, XML).

4.3 Open document workflow

The following diagram shows the workflow for opening a document.

The document contains data that is retrieved from the data source (universe). Each document is identified by an unique ID which is required to open a document.

4.4 Refresh document workflow

The following diagram shows the workflow for refreshing a document.
The document must be refreshed to get the latest information from the data source. The refresh action requires prompt information if the document contains prompts, and context information if the document contains contexts.
Setting up development environment

This section describes how to set up a development environment for web application based on ReportEngine SDK in a Java environment.

5.1 Setting up the Java environment

To setup the ReportEngine Java environment, refer the following sections:

- Setting up the Eclipse environment
- Project Setup

5.1.1 Setting up the Eclipse environment

Eclipse is an Integrated Development Environment (IDE) which provides a universal toolset for application development. Eclipse is an open development platform and can be downloaded from the eclipse web site for free of cost.

To create a Java web-based application, you must have an application server and Java Development Kit (JDK) installed on the machine. By default, Apache Tomcat application server is installed along with SAP BusinessObjects Business Intelligence installation. Hence, an example on creating a dynamic web project is explained considering Apache Tomcat as an application server.

5.1.2 Project Setup

It is recommended to create a dynamic web project using Eclipse. It has the following advantages:

- External jars and other Eclipse projects can be included in the working project.
- Easy to export a war file which can be deployed in any application server.
- Easy to run the application code.

The Web project contains the JSP files, images and various web components in a structured manner. The WEB-INF, lib, classes and other directories which are required to create a web application will be predefined. This makes the coding simpler.
5.1.2.1 Creating a dynamic web project

Prerequisites

The following are the pre-requisites for creating a web application:

- Eclipse IDE for Java EE Developers must be installed and configured.
- Apache Tomcat application server must be installed.
- Java Development Kit (JDK) must be installed and configured.

Procedure

1. Launch Eclipse by selecting a workspace.
2. In the Package Explorer panel, right-click, select New Runtime and click Project.
   Select a Wizard window appears.
3. In the list box, expand Web and select Dynamic Web Project.
4. Click Next.
   Dynamic Web Project window appears.
5. Enter the Project Name.
   New Server Runtime Environment window appears.
7. Under Select the type of runtime that you want to define:, expand Apache and select the runtime.
8. Click Next.
   Tomcat Server window appears.
10. Under JRE, click Installed JREs.
    Installed JREs window appears.
11. Click Add, select the type of JRE to add to the workspace, click Next.
12. Enter JRE name, JRE home directory and click OK.
13. In the Installed JREs window, select the JRE and click OK.
14. Click Finish.
15. Click Finish.

Results

A dynamic web project is created. We recommend that you work in a JEE perspective.
5.1.2.2 Adding Rebean jars to the project

Procedure

1. In the Project Explorer, right-click on the project, select **Build Path** and click **Configure Build Path**. Properties for Project window appears.
2. Select Java Build Path and click on Libraries tab.
3. Click on Add External JARS.
4. Click on Add External JARS.
5. Navigate to <BOBJ_INST_DIR>\SAP BusinessObjects\SAP BusinessObjects Enterprise XI 4.0\java\lib directory, select all the jars and click Open.
   The Rebean jars get added in the library ...\java resources\libairies.
6. Click OK.
7. Expand project, under WebContent directory, expand the WEB-INF directory. Right-click on lib directory and click on Import.
   Import window appears.
8. Click General, select File System and click Next.
   Click Browse, navigate to <BOBJ_INST_DIR>\SAP BusinessObjects\SAP BusinessObjects Enterprise XI 4.0\java\lib directory, select all the jars and click Finish.
   The jars get imported to WEB-INF\lib directory.

5.1.2.3 Creating and running a JSP page

Procedure

1. In the Project Explorer, right-click on the WebContent directory, select New and then click JSP file.
   New JavaServer Page window appears.
2. Enter the File name and click Finish.
   JSP file gets created under WebContent directory.
3. Double-click on the JSP file and write the code in it.
4. In the Project Explorer, right-click on the JSP file, select Run As and click Run On Server.
   Run On Server window appears.
5. Move the project to the right to configure it on the server and click Finish.
   The JSP file will be executed and the result will be shown in a browser.
6 Developing a web application using ReportEngine SDK

This section describes procedures to develop a custom web application using ReportEngine SDK.

6.1 Overview

Before providing viewing functions, the ReportEngine SDK applications need to perform basic tasks, such as referencing the appropriate package and creating a BusinessObjects Business Intelligence session for the user. This section covers the essential concepts and tasks, which all the applications that customize BusinessObjects Business Intelligence must use.

See Business Intelligence platform Java SDK Developer Guide at the link referenced below.

Related Information

http://help.sap.com

6.2 Working with the ReportEngine SDK

All applications that view Web Intelligence documents must:

- reference ReportEngine SDK
- retrieve the ReportEngines service
- retrieve the ReportEngine instance
  This links the ReportEngine and platform packages.
- close the ReportEngines object

i Note

You can use the ReportEngine SDK to view, refresh, fill prompt, fill context, drill (partially), save Web Intelligence documents.
6.2.1 Referencing ReportEngine SDK


6.2.2 Closing a ReportEngines object

To close a BusinessObjects BI session, you must close a ReportEngines instance when you have finished with it. To close a ReportEngines instance use `ReportEngines.close()`. This method deallocates the memory assigned to the object and must be called before you call `IEnterpriseSession.logoff()`.

6.3 Document Management - Overview

Two key parts of a typical ReportEngine SDK application are:

1. Listing the documents that the user can view.
2. Depending on the user’s rights, allowing the user to browse and manage categories and folders used to organize the documents.

This section explains how to open, list, and work with documents.

6.3.1 Document types

With BusinessObjects Enterprise SDK, you can work with the following document types:

- Web Intelligence documents
- Crystal Reports documents
- third-party documents, for example Microsoft Excel (.xls), and Adobe Acrobat (.pdf) files

Working with the Web Intelligence documents using the ReportEngine SDK

Using ReportEngine SDK you can do everything that Web Intelligence users can do with Web Intelligence documents:

- view as DHTML
- view as PDF
- view as Excel
- view as XML
6.3.2 Document state: storage tokens

Storage tokens represent a state of a document at a particular stage, each time a document is modified. For example, the document state changes when the document is refreshed.

Storage tokens are used to restore a document state previously saved.

6.3.2.1 Storage token life cycle

ReportEngine SDK creates a temporary state that contains information about the document’s current state. A temporary state is created when (an only when) one calls `DocumentInstance.getStorageToken()`.

As the document changes state through the execution of JSP files, the storage token changes too, and more temporary states are created representing the document’s states.

You can retrieve the maximum number of storage tokens that can be created as a document changes state programmatically and validate that a storage token is valid by calling `ReportEngine.getStorageTokenStackSize()` and `ReportEngine.IsStorageTokenValid` respectively.

6.3.2.2 Advantages of storage tokens

The advantages of storage tokens are as follows:

**Performance**

Using storage token, you can quickly switch between the document states. It also helps to optimize the memory usage.

**Lifetime**

A storage token lasts as long as the `ReportEngines` object used to create `ReportEngine` instance exists. Business Objects recommends you to store a `ReportEngines` object in the JSP session object once it is created and retrieve it again as required, for document storage tokens to be valid throughout an application.
Browser navigation: back and forward

After progressing through several document states, users can go back and forward in their browser to a previous document state and continue working from that state.

Undo mechanism

You can build an undo mechanism by passing the storage token from the previous action performed on the document to the next web page.

6.3.3 Working with documents

Working with documents involves opening and listing the documents, and providing facilities for saving, sending, scheduling, and organizing documents.

Note

For more information on listing, sending, scheduling see Business Intelligence platform Java SDK Developer Guide at the link referenced below.

Related Information

http://help.sap.com

6.3.3.1 Opening documents

Context

After establishing a session for a user, you can open a document on behalf of the user.

Procedure

1. Establish a BusinessObjects BI session for the user.
2. Get an identifier for the document.
   - The identifier can be a storage token, or more simply the document's ID.
3. Retrieve the `ReportEngine` instance to open the document of desired type.

### 6.3.3.2 Saving documents

To save a document to Central Management Server, use `DocumentInstance.save` and `DocumentInstance.saveAs`.

**Example**

**Saving a document**

The following example shows how to open a document to be worked on, then open and save the original version of the document.

```java
<% 
DocumentInstance doc = myReportEngine.openDocument(docID); 
String docToken = doc.getStorageToken(); 
// User perform actions on document doc... 
// Open the first version of the document and save. 
DocumentInstance docToSave = theReportEngines.getDocumentFromStorageToken(docToken); 
docToSave.save(); 
%>
```

### 6.3.3.3 Using ReportPart

A report part can be a block, section, image or cell within a report. Each report part is identified by a reference. You can extract a report part reference of the particular report element by getting the report output in XML format.

The report part feature allows you to select:

- A specific part of a report
- Multiple parts of the same report
- Report parts located in different reports, but in the same document

Each report part is associated with an Unique Reference and a Temporary Reference. A brief explanation of references is given below:

- **Unique Reference**: It is a constant reference that identifies a single element in the report structure. Its occurrence in the report output is associated with the data in the cube.
- **Temporary Reference**: It is a temporary reference that identifies a single element in the report structure. Its occurrence in the report output is index-based.

For example, consider a report with sections containing "Year" object.

- 2001 - (temporary reference - t1, unique reference - u1)
- 2002 - (temporary reference - t2, unique reference - u2)
2003 - (temporary reference - t3, unique reference - u3)

When you request for report part using "t2" temporary reference, you will get "2002" section.

Again, consider that after data refresh, the "2002" section in the report gets deleted,

2001 - (temporary reference - t1, unique reference - u1)

2003 - (temporary reference - t2, unique reference - u3)

2004 - (temporary reference - t3, unique reference - u4)

When you request for report part using "t2" temporary reference, you will now get "2003" section. However, if you use "u3" unique reference in both cases, you will get "2003" section.

### Example

**Accessing the report part using the ReportPartReference in Java**

```java
DocumentInstance wiDoc = repEng.openDocument(Integer.parseInt(strDocId));
String[] reference = new String[1];
//Find the report part reference by getting XMLView of the document. In the
XMLView, you will find only
temporary reference (for example, <bag layout="bag" index = "Caroline"
ref = "2.3----y.1">)
reference[0] = "2.3----y.1";
ReportParts objReportParts = wiDoc.getReportParts(reference, OutputFormatType.PDF);
ReportPart objReportPart = objReportParts.getItem(0);
//To find unique reference, you should get the ReportPart using temporary
reference and find out
unique reference associated to it.
String uniqueRef = objReportPart.getUniqueReference();
BinaryView myBOView = (BinaryView) objReportParts.getView();
OutputStream outputStream = response.getOutputStream();
response.setContentType("application/pdf");
response.setHeader("Content-Type", "application/pdf");
response.setDateHeader("expires", 0);
myBOView.getContent(outputStream);
```

### 6.3.3.3.1 Report part output customization

A set of properties could be used for output customization.

The following properties are supported only in XML format:

- **OutputPropertiesType.XML_UNIT** (metric, pixel) requests the XML report output to be expressed using a specific measurement unit.
- **OutputPropertiesType.XML_FONT_MAPPING** (none, html, java, win, ttf) requests that the XML report output maps logical font names to platform specific names.

The following property is available for all output formats:

- **OutputPropertiesType.XML_DPI_FACTOR** (default 96 dots per inch) sets the DPI factor used by the report engine to performe measurement unit conversions.
6.3.4 Document properties

Some important characteristics of Web Intelligence documents are stored as part of the document. You can access these properties using the DocumentInstance object.

For ReportEngine SDK, following is the process for accessing the properties of a Web Intelligence document:

1. Get the collection of properties.
2. Set/Read the property.
3. Commit the change, if any, to the collection of properties.

6.3.4.1 Working with properties

Using ReportEngine SDK, you can access the properties of the document using DocumentInstance.getProperties and DocumentInstance.setProperties methods. These methods make use of the java.util.Properties class.

There are predefined properties defined in PropertiesType. You can also add your own properties to the document.

To avoid confusion in the Central Management System, PropertiesType.NAME is read only.

Example

Getting the name of a document

The following code fragment illustrates how to get the NAME property of a Web Intelligence document.

```java
DocumentInstance doc = myReportEngine.openDocument(docID);
java.util.Properties props = doc.getProperties();
String dName = props.getProperty(PropertiesType.NAME);
```

6.3.4.2 Adding your own properties to the document

Use Properties.setProperty to add your own properties to the document.
Example

Adding RefreshCount property to a document

The following code fragment illustrates how to add the RefreshCount property to a Web Intelligence document.

```java
DocumentInstance doc = myReportEngine.openDocument(docID);
java.util.Properties props = doc.getProperties ();
props.setProperty ("RefreshCount", "12");
doc.setProperties (props);
```

If you create a new Properties collection and add properties to that collection, when you call DocumentInstance.setProperties the values in the new object are added to the standard property collection. Permanent properties such as NAME and AUTHOR are not erased in the merge. Values for the properties that you have set in the new collection take precedence over those in the one attached to the DocumentInstance.

6.4 Viewing reports - Overview


This chapter discusses how to use ReportEngine SDK to view Web Intelligence reports.

6.4.1 Viewing reports

You can view complete documents, including all reports or individual reports or report pages in a variety of different formats. The following tables show the different formats available:

Web Intelligence Documents

Table 5:

<table>
<thead>
<tr>
<th></th>
<th>BINARY CONTENT</th>
<th>EXCEL Data-Centric</th>
<th>EXCEL</th>
<th>PDF</th>
<th>CSV</th>
<th>XML</th>
<th>DHTML</th>
<th>HTML</th>
<th>MHTML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Report Page</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DataProviders</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataProvider</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.4.1.1 Viewing all the reports in a document

Context

To view all the reports in a document:

Procedure

1. Open the document.
2. Get a binary view of the document.
   
   You can get a binary view of a document in PDF or Microsoft Excel format. When ReportEngine SDK converts the document into PDF, it places the reports on separate pages and creates a bookmark for each report. In Microsoft Excel format, the reports of a Web Intelligence document appear on separate sheets.
3. You must prepare the response object to receive an appropriate content type. For Adobe Acrobat format, set the content type to "application/pdf". For Microsoft Excel format, set the content type to "application/vnd.ms-excel".

Example

Viewing all the reports in a Web Intelligence document

The following code fragment displays all the reports of a document in Adobe Acrobat format. In this example, the name of the document is stored in docName and the ID is stored in docID. Typically these parameters are passed to the script in the query string.

```java
<%  // get the document
    String strDocID = request.getParameter("DocID");
    int intDocID = Integer.parseInt(strDocID);
    String strQuery = "Select SI_KIND from CI_INFOOBJECTS " + "where SI_ID=" + strDocID;
    IInfoObjects iDocuments = iStore.query(strQuery);
%>
```
6.4.1.2 Navigating the report

The PaginationMode object helps you to navigate through the displayed report.

The following are the supported pagination modes:

1. **Page mode**: In Page mode, a report is displayed in different pages as per the format settings. You can use all the methods in PageNavigation interface to navigate to different pages of the report.
2. **Listing mode**: In Listing mode, the entire report is displayed in a single page.

**Note**

Listing mode can bring in performance overhead on the large report data.

3. **QuickDisplay mode**: In QuickDisplay mode, you can control the amount of data displayed in the report. You can also use all the methods in PageNavigation interface to navigate to different pages of a report. The maximum number of records, height and width of page can be managed through Central Management Console Web Intelligence Properties.

When you save a document, the current PaginationMode is saved along with each report of the document. When the user reopens the document, the pagination mode information with which it was saved, is automatically considered. For performance reason, when viewing a document, the Listing mode is changed to QuickDisplay mode by default.

To navigate to the last page of the report in QuickDisplay mode:

1. Get the document instance.
2. Choose a report.
3. Set the pagination mode to QuickDisplay.
4. Navigate the report to the last page using PageNavigation.last().
5. Display the report.

The following code explains how to navigate to the last page of the report using ReportEngine SDK API.

```java
DocumentInstance wiDoc = repEng.openDocument(docId);
Reports reps = wiDoc.getReports();
```
6.4.1.3 Viewing an individual report in DHTML

Context

DHTML is the standard format for viewing reports.

To view an individual report in DHTML format:

Procedure

1. Get a document instance
2. Choose a report
3. Get a view of the report in DHTML format
4. Display the report

Example

Viewing a report in a Web Intelligence document

The following code fragment displays a selected report of a Web Intelligence document in DHTML format.

```java
Report rep = reps.getItem(0);
rep.setPaginationMode(PaginationMode.QuickDisplay);
pageNavigation.last();
HTMLView htmlView = (HTMLView)rep.getView(OutputFormatType.DHTML);
PrintWriter writer = response.getWriter();
htmlView.getContent(writer, "", "");
writer.close();
```
6.4.1.4  Viewing an individual report in another format

Context

You can view reports in Adobe Acrobat or Microsoft Excel format. The procedure is same as DHTML format except that, for viewing all the reports in a document, you must get the content of the report as a binary stream which you can output through the response object.

To view a report in a non-DHTML format:

Procedure

1. Get the binary view of the report.
2. The binary view of the document can be obtained through an implicit response object.

Example

Viewing a report in Microsoft Excel format

The following code fragment displays all the reports in a Web Intelligence document in Microsoft Excel format.

```java
//get the report to view (Report rep)
...
//get the report contents as a binary stream in Excel format
BinaryView docBinaryView = (BinaryView)rep.getView(OutputFormatType.XLS);
//output the binary stream via the response object
response.setContentType("application/vnd.ms-excel");
response.setDateHeader("expires", 0);
//output the binary stream via the response object
byte[] abyBinaryContent = docBinaryView.getContent();
//output the binary stream via the response object
outputStream.write(abyBinaryContent);
```
6.4.1.5 Track Data Changes

The following sections briefly describe the Track Data Changes feature and the workflow you need to perform to activate and deactivate Track Data Changes feature.

The Track Data Changes feature

The Track Data Changes feature allows you to visualize the differences between two states of the data: a reference state and the last refresh. It saves your time in the data analysis and reduces the time exploring the irrelevant data. This feature compares the current data with the data that is set as a reference and highlights any data that is added, removed, updated, increased, or decreased.

Activating and disabling the Track Data Changes feature

Before showing the data changes, the system has to track the data changes along refreshes. Since tracking of the data changes consumes significant amount of memory and process time, it has to be explicitly activated in the document to maintain the performance.

Use `TrackData.setTrackDataMode(ReferenceUpdateMode.USER_DEFINED | ReferenceUpdateMode.AUTO)` to activate the Track Data Changes and `TrackData.setTrackDataMode(ReferenceUpdateMode.DISABLED)` to disable the Track Data Changes feature in the document.

Setting the data to use as a reference

When you activate the Track Data Changes, you need to set the data as a reference for comparison. You can use the following modes to set the data reference:

- **ReferenceUpdateMode.AUTO**: automatically sets the current data as a reference when you refresh a document.
- **ReferenceUpdateMode.USER_DEFINED**: allows you to explicitly set the data as a reference using `TrackData.setCurrentAsReference()`. This way, you replace the previous data reference with the current data. The future refreshes will be compared with the current data.
- **ReferenceUpdateMode.DISABLED**: allows you to disable the Track Data Changes feature.

Example

**Activating the Track Data Changes**

```java
if(repEng.getCanTrackData())
{
    TrackData data = wiDoc.getTrackData();
    // activate trackdata
    data.setTrackDataMode(ReferenceUpdateMode.USER_DEFINED);
    data.setCurrentAsReference();
}
```
Setting the display options

After you activate the Track Data Changes, you must set the display options for Track Data Changes.

The following table summarizes the type and condition of the tracked data changes according to object qualification:

Table 6:

<table>
<thead>
<tr>
<th>Data changes event</th>
<th>Dimension</th>
<th>Details</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added data</td>
<td>Tracked</td>
<td>Tracked</td>
<td>Tracked</td>
</tr>
<tr>
<td>Removed data</td>
<td>Tracked</td>
<td>Tracked</td>
<td>Tracked</td>
</tr>
<tr>
<td>Changed data</td>
<td>N/A</td>
<td>Tracked</td>
<td>N/A</td>
</tr>
<tr>
<td>Increased data</td>
<td>N/A</td>
<td>N/A</td>
<td>Tracked</td>
</tr>
<tr>
<td>Decreased data</td>
<td>N/A</td>
<td>N/A</td>
<td>Tracked</td>
</tr>
</tbody>
</table>

The changed data are displayed with a predefined style based on a combination of the following cell settings:

- Font Style
- Text Underlined
- Text Strikethrough
- Text Color
- Cell Background

The default values for these parameters are defined at server side and managed through the Central Management Console Web Intelligence Properties. These changes will take effect only after restarting the Web Intelligence server.

Example

Setting the track data changes options

```java
// get style options
TrackDataOptions options = data.getTrackDataOptions();
TrackDataOption option = options.getInsertedOption();
Decoration decoration = option.getDecoration();
Attributes attributes = decoration.getAttributes();
attributes.setBackground(Color.WHITE);
attributes.setForeground(Color.GREEN);
// change the font style
Font font = decoration.getFont();
font.setStyle(StyleType.ITALIC);
data.update();
```
Viewing and hiding the data changes

The following table describes the methods you can use to view and hide the data changes:

<table>
<thead>
<tr>
<th></th>
<th>To view the data changes</th>
<th>To hide the data changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the current active report</td>
<td>TrackDataInfo.showChanges(true)</td>
<td>TrackDataInfo.showChanges(false)</td>
</tr>
<tr>
<td>In all reports</td>
<td>TrackData.showChanges(true)</td>
<td>TrackData.showChanges(false)</td>
</tr>
</tbody>
</table>

Example

Viewing the data changes in a specific report

```java
Reports reps = wiDoc.getReports();
Report rep = reps.getItem(0);
TrackDataInfo trackDataInfo = rep.getTrackDataInfo();
trackDataInfo.showChanges(true);
```

6.4.2 Handling prompts

Prompts are the way to get supplementary information into a query before executing it. In ReportEngine SDK, the way you handle prompts depends on the type of document you are working on. This section discusses how to handle prompts in Web Intelligence documents.

Prompts are raised when a document is refreshed using DocumentInstance.refresh. The refresh operation executes the query and therefore needs to collect the supplementary information that resolves filters and contexts.

i Note
Prompts that resolve universe contexts must be filled before the standard prompts.

6.4.2.1 Documents with a single simple prompt

One of the simplest and most common prompt is one that resolves a query filter. For example, "Enter the year: " to which a user responds "2003" and the data retrieved by the query is restricted to the year 2003.
Handling a single, simple prompt

The following code fragment illustrates how to handle a single, simple prompt. ("Simple prompt" means standard, text-entry prompts only)

The code in this example is from a JSP called refresh.jsp. When the user chooses to refresh a document, refresh.jsp is executed, then, when the prompt is filled, view.jsp is executed.

```java
DocumentInstance doc = reportEngines.getServiceFromStorageToken(token);
if (doRefresh) doc.refresh(); //refresh first time only
Prompts prompts = doc.getPrompts();
//try to enter values and set prompts
String[] values = request.getParameterValues("PromptInput");
if ((values != null) && !(values[0].equals("")))
{
    prompts.getItem(0).enterValues(values);
    doc.setPrompts(); //also sets getMustFillPrompts to false
}
//get user input
if (doc.getMustFillPrompts()) {
    //build a form to get user input for the prompt
    <form name="PromptsForm"
        action="refresh.jsp?Token=<%=(doc.getStorageToken())%>"
        method="post">
        <input name="PromptInput" type="text" />
        <input name="Submit" type="submit" value="OK" />
    </form>
}
Response.sendRedirect("view.jsp?
    Token="+doc.getStorageToken());
```

i Note

This example does not show how to set the Boolean variable doRefresh. One way to set this variable is to include an extra parameter in the query string for refresh.jsp. When DoRefresh is "F", doRefresh is false.

The workflow for refresh.jsp for a document with simple prompt is:

2. Get the Prompts collection for the document.
3. Enter the values and set the Prompt for the document.

i Note

The first time refresh.jsp is executed, this step will not work as the user has not yet entered a value for the Prompt.

4. Do one of the following:
   ○ if the Prompt is not filled, display a form to get a value from the user.
     The first time the file is executed.
     The form's action parameter is set to refresh.jsp so that this file is executed when the user clicks the
     OK (submit) button.
   ○ Otherwise, execute view.jsp to display the document's reports.
6.4.2.2 Using the storage token to retrieve a DocumentInstance

When handling any kind of prompt, if you use the storage token to retrieve the document, you must be careful to use the correct token.

For more information on storage tokens see the link referenced below.

Related Information

Document state: storage tokens [page 17]

6.4.2.3 Documents with many simple prompts

To handle many simple prompt objects, you can extend the workflow of a single simple prompt with a form into which the user can enter values for all the prompts, and a script that handles these values.

The problem this extension raises is that, you must dynamically set the names of the inputs in the form that carry the values of each prompt. To achieve this, you can construct a name using a string constant, for example “PV”, and the position of the prompt in the prompts collection.

Example

Dynamically creating input names

The following code fragment illustrates how to dynamically create names for the inputs of a form.

```html
<form name="PromptsForm" action="refresh.jsp" method="post">
  <table>
    <tr>
      <td>Enter a value for prompt number %i%:</td>
      <td><input name="%PV"+i%" type="text"/></td>
    </tr>
  </table>
  <tr><td><input name="Submit" type="submit" value="OK" /></td></tr>
</form>
```

Note

See the example "Handling many, simple prompts" for an illustration on how to retrieve the values in these inputs from the query string.
Handling many, simple prompts

The following code fragment illustrates how to handle many, simple prompts in a Web Intelligence document. ("Simple prompt" means standard, text-entry prompts.)

The code in this example is from a page called refresh.jsp. When the user chooses to refresh a document, refresh.jsp is executed, then, when the prompt is filled, view.jsp is executed.

```java
<% // get the document and its prompts DocumentInstance doc = reportEngine.getServiceFromStorageToken(token); if (doRefresh) doc.refresh(); // refresh first time only Prompts prompts = doc.getPrompts(); // try to get and enter values from the query string // valuesSelected is true when the user completes the form if (valuesSelected) {
    for (int j = 0; j < prompts.getCount(); j++) {
        // use (recreate) the parameter names created in the form
        String[] values = request.getParameterValues("PV" + j);
        if ((values != null) && (values.length != 0))
            prompts.getItem(j).enterValues(values);
    }
    doc.setPrompts(); // also sets getMustFillPrompts to false
} // get user input if (doc.getMustFillPrompts()) {
    // build a form to get user input for the prompt
    %>
    <form name="PromptsForm" action="refresh.jsp" method="post">
        <table>
            <% // add a row to the table for each prompt
            // the names of inputs are created dynamically
            for (int i = 0; i < prompts.getCount(); i++) {
                %>
                <tr>
                    <td><%= prompts.getItem(i).getName() %></td>
                    <td><input name="PV"+i type="text" /></td>
                </tr>
                <% }
            // and add a row to the table for the submit button
            // DoRefresh and ValuesSelected help keep track of
            // where we are in the workflow
            %>
            <tr><td><input name="Token" type="hidden" value="<%=doc.getStorageToken()%>" />
                <input name="DoRefresh" type="hidden" value="false" />
                <input name="ValuesSelected" type="hidden" value="true" />
                <input name="Submit" type="submit" value="OK">
            </td></tr>
        </table>
    </form>
    %>
} else // all prompts are filled and can now display the report
    response.sendRedirect("view.jsp?Token="+doc.getStorageToken());
%>
```

Developing a web application using ReportEngine SDK
A flag is now required to determine if there are prompt values in the query string. This example uses a hidden input in the form, `ValuesSelected`, which is converted to a boolean when it is retrieved. However, you can also use the first prompt parameter PV0 for this purpose.

### 6.4.2.4 Prompts with simple lists of values

A List Of Values (`Lov`) is a set of values associated with a universe object. These values are the corresponding values found for the object in the database. The Information Design tool can edit this set as a part of creating the object in the universe. For example, the Quarter object can have the following list of values `{Q1, Q2, Q3, Q4}`, and the Customer object can have a list of values like this: `{Adams, Arkwright, Baker, Bean, ..., Zane}`. These values are defined when the designer creates the object, but can be refreshed, to accommodate changes in the database, automatically or manually.

Handling prompts that contain `Lov` objects involves extending the workflow for many, simple prompts. To do this, check if the prompt object has an associated `Lov` by calling `Prompt.hasLOV`. If there is a list of values for the prompt, get the values in the list with `Lov.getAllValues`, then display them in the user input form, with `Lov.getValues`, as options of HTML `select` tag.

**Note**

`Prompt.getLOV` refreshes data. This can cause problems with nested prompts which must be handled separately.

### Example

**Displaying a simple list of values**

The following code fragment illustrates how to get the values of a `Lov` object and display them in a form.

```java
//get user input
if (doc.getMustFillPrompts()) {
    //build a form to get user input for the prompt
    <form name="PromptsForm" action="refresh.jsp" method="post">
        <table>
            <tr>
                <td>
                    <%= prompt.getName() %>
                </td>
                <td>
                    <select>
                        <%
                        Prompt prompt = prompts.getItem(i);
                        if (prompt.hasLOV()) {
                            Values lovValues = prompt.getLOV().getAllValues();
                            for (int i = 0; i < lovValues.size(); i++) {
                                out.print("<option value="
                                        + lovValues.get(i) + ">
                                        " + lovValues.get(i) + "</option>\n                                ");
                            }
                        }
                        %>
                    </select>
                </td>
            </tr>
        </table>
    </form>
}
```
<select name="PV"+i%> size=1>
  <%
  for (int k=0; k < lovValues.getCount(); k++){
    <option value="<%=lovValues.getValue(k)%>">
      <%=lovValues.getValue(k)%>
    </option>
  }
  %>
</select>

<i>Note</i>
The example "Handling multivalued prompts" shows how to modify the select tag to handle multivalued prompts.

### 6.4.2.5 Optional Prompts

When a prompt is set as optional and when you run the query, user input for that prompt is optional.

Use `Prompt.isOptional()` to check if the prompt is an optional prompt.

For example, if you have three prompts in a Web Intelligence report and one of them is defined as optional and when you run this query, you need not fill value for this optional prompt.

<i>Note</i>
Optional prompts are not applicable for nested and hierarchical prompts.

### 6.4.2.6 Constrained Prompts

A prompt is constrained if the answer must come from the prompt’s list of values (LOV). In Web Intelligence, to create constrained prompts select <i>Select Only From List</i> when creating a prompt filter for a query.

You can detect if a prompt is constrained with `Prompt.isConstrained`. 
6.4.2.7 Multivalued prompts

Until now we have considered only prompts that can have one value. However, some prompts can require several values before it is filled. For example, the prompt, “Store name In list” can have one or many responses before it is filled, whereas the prompt, “Year Equals” can have only one response.

**Note**

For prompts that use the Between operator, such as, Sales revenue is between X and Y; the user must specify both X and Y before the prompt is considered complete. However, Web Intelligence forms these kinds of prompts as two separate prompts, you need not treat them as a multivalued prompt.

Use `Prompt.getType()`, to detect how many values the prompt requires. A prompt can either be `PromptType.Mono` or `PromptType.Multi`.

**Example**

Handling multivalued prompts

The following code fragment illustrates how to modify the `<select>` tag of the previous example to handle multivalued prompts.

```java
<td>%
  if (prompt.getType()==PromptType.Mono){%>
    <select name="%="PV"+i%" size=1>
  %}
  //multivalued LOV needs a different select statement
  else {%>
    <select name="%="PV"+i%" multiple size=5>
  %}
  //add values in LOV to the select list
  for (int k = 0; k < lovValues.getCount(); k++) {%>
    <option value="%=lovValues.getValue(k)%">
    <%=lovValues.getValue(k)%>
  </option>%>
  %>
  </select>
%</td>
```

6.4.2.8 Refreshing a list of values

You can use `Lov.refresh` to provide a *Refresh List* button so that the user can update the list manually.

**Note**

This process is independent of the document refresh mechanism.

After calling `Lov.refresh`, the next time you call `Lov.getValues` or `Lov.getAllValues`, the list of values will be refreshed.
6.4.2.9 Prompts with multicolumn lists of values

A universe designer can create a list of values that has many columns.

You can check a list of values with more than one column using `Values.isMultiColumns`.

In ReportEngine SDK, a multicolumn list of values is represented as rows in a table with the `RowValue` interface. Use `Values.getRowValue` to get a row in the table. And then `RowValue.getHeader` and `RowValue.getItem` displays the values in the list.

**Note**

`Prompt.enterValues` uses only the first value in each row to identify the row. When the user selects a row, make sure you pass just the value in the first column (index = 0) of the row to `Prompt.enterValues`. The example below illustrates this.

**Example**

Handling multicolumn lists of values

The following code fragment illustrates how to display multicolumn lists of values in the select block.

```java
if (prompt.hasLOV()) {
    //display the list of values
    Values lovValues = prompt.getLOV().getAllValues();
    <tr>
        <td><%=prompt.getName()%></td>
        <td>
            if (lovValues.isMultiColumns()) {
                //MULTICOLUMN VALUES (1/2) display the column names
                String colNames = "";
                for (int m = 0; m < lovValues.getRowValue(0).getColumn(); m++)
                    colNames = colNames + " | " + lovValues.getRowValue(0).getColumn();
                %><%=colNames%><br><%
            }
            //assume multivalue LOV
            <select name="PromptValue" multiple size=5>
                <% //get value and text for the option tag
                    for (int k = 0; k < lovValues.getCount(); k++) {
                        String value = lovValues.getValue(k);
                        String valueText = value;
                        if (lovValues.isMultiColumns()) {
                            //MULTICOLUMN VALUES (2/2) option is a row
                            RowValue row = lovValues.getRowValue(k);
                            value = row.getItem(0);
                            valueText = "";
                            for (int n = 0; n < row.getCount(); n++) {
                                //build a string to display the <option> tag
                                if (n == 0) valueText = row.getItem(n);
                                else valueText = valueText + " | " + row.getItem(n);
                            }
                        } %>
                        <option value="" + value %>
                            <%=valueText%>
                        </option>
                    %>
                </select>
```
In multicolumn lists of values, value and valueText are different. The variable value contains just the first value in the row, which is passed to Prompt.enterValues to fill the prompt, whereas valueText displays all the values in a row.

6.4.2.10 Chunking long lists of values

Some objects, for example Customer name, can have a very long list of values. The Lov interface has methods that you can use to divide long lists of values into chunks that are easier to display, and easier for the user to browse.

Following are the methods used for Lov batching:

- Lov.getCurrentBatchIndex
- Lov.getValues
- Lov.getCurrentBatchName
- Lov.setBatchSize

6.4.2.11 Handling nested prompts

Sometimes a list of values contains its own prompts. A prompt in a list of values is called a nested prompt. Nested prompts can be raised when a document containing prompts is refreshed and the prompts have lists of values that contain their own prompts.

If a list of values contains prompts, those prompts must be filled before you can fill the prompts in the next level up.

Example

Order of filling a hierarchy of nested prompts

Consider the following set of prompts:

Select a town from the list: (prompt in country)
   Select a state from the list: (prompt on list of towns)
      Select a country from the list: (prompt on list of states)

Each prompt qualifies the one above it in the hierarchy; the country must be defined before a state can be defined and, similarly, a state must be defined before a town can be defined.

Use Lov.mustFillNestedPrompts to check if a list of values contains its own prompts, then use Prompt.enterValues and Lov.setNestedPrompts to enter values and set the nested prompts respectively.
Example

Handling nested prompts

The following code fragment illustrates the recommended method for handling nested prompts. In this example, prompts and nested prompts are handled by the recursive function `fillPrompts`.

```java
//get the document and its prompts
DocumentInstance doc = re.getDocumentFromStorageToken(token);
if (doRefresh) doc.refresh(); //refresh first time only
Prompts prompts = doc.getPrompts();
//if there are prompts, fill them
if (doc.getMustFillPrompts()) {
    this.fillPrompts (prompts, null);
    doc.setPrompts();
}
else //all prompts are filled and can now display the report
    response.sendRedirect("view.jsp?
        Token="+doc.getStorageToken());
//FUNCTION - fill prompts and handle nested prompts
public void fillPrompts (Prompts pmts, Lov parentLOV) {
    //pmts is the set of prompts or nested prompts to be filled
    //parentLOV is a LOV for which there are unfilled nested prompts
    for (int i = 0; i < pmts.getCount(); i++) {
        Lov lov = pmts.getItem(i).getLOV();
        if (lov.mustFillNestedPrompts()) {
            Prompts nestedPrompts = lov.getNestedPrompts();
            fillPrompts (nestedPrompts , lov);
        }
        //get user input for pmts
        //enter values for pmts
    }
    if (parentLOV != null) parentLOV.setNestedPrompts();
}
```

6.4.2.12 Refreshing a list of values containing nested prompts

If you provide a manual refresh option, you can handle any nested prompts contained in the list of values with the `fillPrompts` function.

Example

Refreshing a list of values containing nested prompts

The following code fragment containing how to use the `fillPrompts` function.

```java
//get and display the LOV
//if the user clicks "Refresh List"
lov.refresh();
Prompts nestedPmts = lov.getNestedPrompts();
this.fillPrompts (nestedPmts, lov);
```
6.4.2.13 Handling context prompts

Context

Universe designers define context prompts in universes to ensure that the users retrieve appropriate data when there is more than one way to get the results of a query. For more information about defining contexts, see Information Design Tool User Guide.

The process of filling a context prompt is similar to that of filling many, simple prompts. The contexts in a document are represented by the Contexts collection, and you can check if a document has contexts to fill with DocumentInstance.getMustFillContexts.

To fill a context prompt:

Procedure

1. Get contexts.
2. Display the possible values for each context and get the user's response.
3. Enter the values for the context provided by the user.
4. Set the contexts.

6.4.3 Displaying a report map

ReportMap is a representation of the reports and the report sections in a Web Intelligence document.

Content of a report map is retrieved incrementally, this means that only the requested information is calculated.

To get the report map of a document, call DocumentInstance.getReportMap.

Example

Traversing the report map of a Web Intelligence document

The following code fragment traverses the following report structure and gets a section as DHTML.

```java
// Get the ReportMap from the document instance
ReportMap map = doc.getReportMap();
// point to the root node of the structure
ReportMapNodes root = map.getStructure();
// Get the number of reports in the document
// There are three in this case: Report1, Report2, and Report3
int count = root.getChildCount();
// Get the first report
ReportMapNode report1 = root.getChildAt(0);
// Get the name of the report (Report1)
String reportName = report1.getName();
```
6.5 Drilling in Web Intelligence Reports - Overview

Drilling is one of the key features of Web Intelligence documents, which helps users to analyze the document data. This chapter discusses how to provide drilling facilities for Web Intelligence documents.

Note
Drill feature is partially supported starting with BusinessObjects BI 4.0 release.

6.5.1 Introduction to drilling

Drilling controls the amount of details in a report. Universe designers create hierarchies of dimensions when they create universes, for example Country, State, City, Zip Code, Street. When users view reports, they can adjust the amount of details in the reports by entering into the drill mode and also drill up or down according to the dimension hierarchies (often called drill hierarchies).

Note
Depending on the user rights, some users do not have access to the drilling functions.

Users set the scope of analysis to control how much data the Web Intelligence includes in the DataProvider (cube) it creates, when a query is executed.

When a user executes a query, Web Intelligence retrieves data not only for the dimensions in the query, but also for the dimensions that the user has included in the scope of analysis. This means that when a user drills through a dimension hierarchy, the information for the new report is in the DataProvider, and it is not necessary to execute a new query to display the drilled report.
6.5.1.1 The drilling process

Context

To provide users with a drill function, implement the following process:

Procedure

1. Request for a drill operation by clicking on a drill link in a report.
   The request contains information about the drill operation. For example, the name of the script that will handle the request, from which dimension the drill starts, and to which dimension it goes.

2. Define the drill operation by initializing the document’s drilling objects with the information passed in the drill request.

3. The script instructs Web Intelligence to generate the HTML for the drilled report.
   Web Intelligence uses the information in the document’s drilling classes to generate the drilled view.

4. Display the drilled view generated by Web Intelligence.

5. Repeat steps 2 to 4 for each request for a drill operation.

6.5.1.2 Defining the drill operation

You can only define drill operations for drillable documents. A document is considered as drillable, if the dimensions used to form the query are part of a dimension hierarchy.

Defining the drill operation involves:

- getting the parameters of the drill request
- defining the query string parameters
- entering and leaving drill mode
- setting the drill path

The details of how you define the drill operation depends on the type of report.

6.5.1.3 Generating the HTML and viewing a drilled report

This step involves generating the HTML for the report using the drill operation you have defined. Once you have done this, you can view the report as HTML using the normal workflows.
6.5.1.4 Drilling out of scope: the scope of analysis

An out of scope drill is one that goes up or down to a dimension that is not in the DataProvider. To fulfill this request the query must be reformed and rerun to retrieve the requested data.

You can handle out of scope drills manually or transparently. How you manually handle out of scope drills depends on the type of report, however, the mechanism for transparently handling out of scope drills does not depend on the report type.

6.5.1.5 Transparent Drill Outside of Cube

The Web Intelligence facility for automatically handling out of scope drill requests is called Transparent Drill Outside of Cube. When this is functioning, the scope of analysis of the Query is automatically reset to incorporate the data for the requested dimension.

Administrators can control a user’s access to this facility using Central Management Console.

6.5.1.6 Drill hierarchies

Hierarchies contain dimensions and are defined by universe designers. You can see drill hierarchies in the Report Panel.

A dimension contains a list of values. When a value is selected from the list, it acts as a dimension filter.

6.5.2 Drilling in reports

Drilling Information

You can access drilling information by examining the contents of the universe (the drill hierarchies and dimensions), or by examining the drill bar.

Drill hierarchies defined in a universe

Use `Report.getNamedInterface("DrillInfo")` to get a `DrillInfo` instance.

To get the `DrillHierarchies` used in the report, use `DrillInfo.getDrillHierarchies`. A hierarchy is used if one of its dimensions is included in the query for the report.

To get the dimensions and details that are not included in the hierarchy but are used in the report, use `DrillInfo.getFreeDimensions`.
Defining the drill operation

A drill operation is defined in terms of drill path. A drill path is represented in ReportEngine SDK by the `DrillPath` interface, and consists of a set of parameters such as the IDs of the objects (dimensions) “from” and “to” which the user is drilling. Defining the drill operation, therefore, involves setting the parameters of the drill path object.

To define the drill operation:
1. Enter drill mode.
2. Define the query string parameters, if required.
3. Get the parameters of the drill request.
4. Set the drill path.

Entering and leaving drill mode

You must switch a Web Intelligence report into a drill mode so that it can perform the drilling functions. There are two modes: `ReportMode.Viewing` and `ReportMode.Analysis`.

Calling `DrillInfo.beginDrill` puts a report in `ReportMode.Analysis` (drill) mode. Calling `DrillInfo.endDrill` stops the drill session and puts a report in `ReportMode.Viewing` mode.

Setting the drill path

The `DrillPath` defines the drill operation. To set the drill path:
1. Get the `DrillPath` object.
2. Set the drill action.
3. Set the block id.
4. Set the object id of the "drill to" element.
5. Set the object id of the "drill from" element.
6. Set the filter.

Setting the action: up, down, by, or slice

A user’s drill actions are classified by the way they move through the drill hierarchies.

Table 8:

<table>
<thead>
<tr>
<th>Drill action</th>
<th>Resulting report engine action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>The report engine replaces the current object with its parent in the drill hierarchy.</td>
</tr>
</tbody>
</table>
Drill action | Resulting report engine action
---|---
Down | The report engine replaces the current object with its child in the drill hierarchy.
By | The report engine replaces the current object with an object that is not adjacent to it in the drill hierarchy.
Slice | The report engine adds or removes the filtered values.

You set the drill action with `DrillPath.setAction`. The actions are enumerated by `DrillActionType`.

### Setting the "from" and "to" parameters

To set the from and to parameters of the drill, you must define the elements of the drill. The elements of the drill are the dimensions involved in the drill action and are represented by the `DrillElements` interface. There are `DrillElements` object for the "to" and the "from" dimensions. Individual "from" and "to" elements are represented by `DrillFromElement` and `DrillToElement` respectively.

For each drill operation:
1. Get the drill from and to elements with `DrillPath.getTo` and `DrillPath.getFrom` respectively.
2. Add drill elements to each collection.
3. Set the object IDs of each drill element using the values you retrieve from the query string.
4. For each from element, set the filter, if any.
   1. The filter remains for drill-by actions between dimensions in the same hierarchy. However for drill-up actions, the filter is removed.
   2. For example, in the Time Period hierarchy, if you drill down to Quarters from Year = 2003, all the quarters for 2003 are displayed, then if you drill by Year, only Year= 2003 is displayed. However if you drill up from Quarters to Year, all the values for Years are displayed.

#### Example

**Setting the drill path**

The following code fragment shows how to set the values of the drill path.

```java
DrillPath drill = info.getDrillPath();
//set the ACTION
if (action.equals("down"))
    drill.setAction(DrillActionType.DOWN);
else if (action.equals("up"))
    drill.setAction(DrillActionType.UP);
else if (action.equals("slice"))
    drill.setAction(DrillActionType.SLICE);
else if (action.equals("by"))
    drill.setAction(DrillActionType.BY);
//set the BLOCK
drill.setBlockID(block);
```
//set the TO drill elements
if (to.length > 0) {
    DrillElements toElements = drill.getTo();
    for (int j = 0; j < to.length; j++) {
        DrillToElement toElement = (DrillToElement) toElements.add();
        toElement.setObjectID(to[j]);
    }
}

//set the FROM drill elements
if (from.length > 0) {
    DrillElements fromElements = drill.getFrom();
    for (int k = 0; k < from.length; k++) {
        DrillFromElement fromElement = (DrillFromElement) fromElements.add();
        fromElement.setObjectID(from[k]);
        if ((filter != null) && (filter.length > 0))
            fromElement.setFilter(filter[0]);
    }
}

This example assumes that there is only one value in the array filter: fromElement.setFilter(filter[0]). This is true for simple drilling, however, to handle more sophisticated drilling you need to use all the values in the array of filters. Also, to set the filter, you can add the condition if !action.equals("up"), since there is no filtering in a drill-up operation. If you set a filter for a drill-up operation, the filter is ignored.

Generating the HTML and viewing a drilled report

To generate the HTML for the report, execute the drill with DrillInfo.executeDrill, then view the generated HTML using HTMLView.getContent.

6.5.2.1 Displaying the drilled report

To display the drilled report, call HTMLView.getContent.

Example

Displaying the drilled report

The following code fragment illustrates how to display a report after executing the drill:

```xml
<% htmlView = null;
htmlView = (HTMLView) rep.getView(OutputFormatType.DHTML);
String htmlHeader = htmlView.getStringPart("head", false);
%>
<html>
    <!-- DRAW HEADER -->
    <head>
        <link rel="stylesheet" type="text/css" href="style/bomenu.css">
        <%=htmlHeader%>
    </head>
    <!-- DRAW REPORT-->
    <%=htmlView.getStringPart("body", true)%>
```
6.5.2.2  Manually handling out of scope drill requests

Once you have set the `DrillTo` and `DrillFrom` elements, you can check if a drill dimension is not in scope with `DrillDimension.isInScope`.

If the defined drill operation contains out of scope dimensions with `DrillInfo.willGoOutOfScope`.

If the `DrillDimension` is outside the scope of analysis, you can manually extend the scope of analysis with `DrillInfo.extendScopeOfAnalysis`. This method returns a `DrillElements` collection to which you can add extra drill dimensions that the user wants to include in the analysis.

Use `DrillInfo.executeDrill` to commit the changes you make to the scope of analysis.

If you want to add a query filter to the extended scope of analysis, use `DrillInfo.addQueryConditions`. This returns a `DrillElements` collection to which you can add the ID of the filter dimensions and the associated filter values. You commit this change with `DrillInfo.executeDrill`.

6.5.2.3  Adding objects that have prompts

If you extend the scope with an object that contains a prompt in its universe definition, you must fill the prompts it raises before you execute the drill.

6.5.2.4  Taking a snapshot of a drill

You can use `DrillInfo.snapshot` to take a snapshot of a drilled view and continue drilling. This method adds a report containing the current view to the document’s report list and report map.

6.6  Working with Recordsets - Overview

The ReportEngine SDK often uses the recordset data structure for storing information such as the results in a data provider.

This section explains how to use the recordset classes of the ReportEngine SDK.
6.6.1 Anatomy of a recordset

In ReportEngine SDK, recordsets provide a generic way of representing the data contained in a group of objects.

Each recordset is divided into rows and columns. Each row can be considered as a record (or a set of fields) and you can access only one record at a time.

Recordsets

In ReportEngine SDK, the Recordset interface represents recordsets.

Field names

The Recordset.getColumnName(n) gives the name of the nth field in the current record.

The names of the fields in most recordsets are fixed. You can find the names listed under the method descriptions in the Web Intelligence Report Engine Java API Reference.

The exception to this is DataProvider.getResult which returns a recordset in which the names of the fields are the names of the results (columns) of the query.

Moving around the recordset

To navigate into a recordset, use the following move methods of Recordset:

- first, last, next, previous, and setRow

The Recordset class has a feature for setting the direction. You can set the direction to FORWARD (default) or REVERSE. Recordset.first and Recordset.last do not depend on the direction and always move to row 0 and row (Recordset.getColumnCount - 1) respectively.

It is good practice to set the direction and call Recordset.first or Recordset.last before you start processing a recordset so that you know which row the Recordset object contains.

You can use Recordset.first and Recordset isFirst to control loops that move through recordsets.

Accessing the value of a field

In ReportEngine SDK, you can access the values of a field directly.
To access the fields of a record, use `Recordset.getCellObject` and provide the index of the column in which you are interested.

**Example**

**Accessing the values in a recordset**

The following code fragment prints the types of columns and the contents of the results of a query contained in a data provider.

```java
Recordset rs = dp.getResult(0); // 0: assume query has one flow
rs.first(); // Print the column types. They can be Integer, String, // or Date.
for (int i = 0; i < rs.getColumnCount(); i++) {
    Class c = rs.getColumnType(i);
    StringBuffer sbt = new StringBuffer();
    if (c.equals(Integer.class))
        sbt.append("Integer");
    if (c.equals(String.class))
        sbt.append("String");
    if (c.equals(Date.class))
        sbt.append("Date");
    sbt.append(";");
    System.out.println(sbt.toString());
}
// Print the recordset contents: column names and data
while (!rs.isLast()) {
    // column names
    StringBuffer sbn = new StringBuffer();
    for (int j = 0; j < rs.getColumnCount(); j++) {
        sbn.append(rs.getColumnName(j).toString());
        sbn.append(";");
    }
    System.out.println(sbn.toString());
    // data
    for (int k = 0; k < rs.getColumnCount(); k++) {
        sbd.append(rs.getCellObject(k).toString());
        sbd.append(";");
    }
    System.out.println(sbd.toString());
    rs.next();
}
```

This code fragment assumes that `rs.getDirection` equals FORWARD, and the data provider (dp) is populated.
7 Best Practices to improve ReportEngine performance

The following sections help you to work with ReportEngine in a better way.

7.1 Best Practices

SAP BusinessObjects recommends the following best practices while working with ReportEngine SDK:

- The documents opened during the BusinessObjects Enterprise session are stored and available for the user as long as the session lasts. As a consequence, opening many documents will significantly consume heap memory of the server. To increase the performance, close the document explicitly when it is no longer required. Use `DocumentInstance.closeDocument()` method to close documents.

- It is recommended to store a `ReportEngines` object in the JSP session object once it is created for document storage tokens to be valid throughout an application. You can retrieve `ReportEngines` object when required through session object.

- Store the user's `IEnterpriseSession` object in the Application server’s session object, in this way the user session has the same timeout as the Application server’s session.

- It is recommended to use `getContent(java.io.OutputStream outputStream)` method to retrieve the view of the report content. This method improves the performance and scalability of the document retrieval by enabling chunked data transfer. It also improves the memory consumption on both the application and the BusinessObjects servers.

- It is recommended to retrieve the List Of Values (LOV) in chunks or batches when the LOV size is large. This will significantly reduce the amount of time taken to display the LOVs. For even better performance, you can enable the search mode and specify the search pattern which will retrieve only required LOVs.

- It is not recommended to use `ReportInfo.getNumberOfPages()` method for very large reports. As the report must be completely computed to retrieve the number of pages, it introduces performance overhead on the server.

- It is not recommended to use `PageNavigation.last()` method unless it is mandatory. This method will compute all the pages in a report and can cause performance overhead on the server.

- Each time `DocumentInstance` object is retrieved from the SAP BusinessObjects Web Intelligence server, it is serialized and its state is recreated. Calling `ReportEngine.getDocumentFromStorageToken(java.lang.String)` method uses many resources on the SAP BusinessObjects Web Intelligence server. So, it is recommended to use this method as less as possible for optimal performance.
7.2 Developer tips

- When working with report parts, use unique reference to retrieve the report part from the report. An unique reference is a constant reference that identifies a single element in the report structure, and is associated with the data in the cube.
- If a document contains many optional prompts, then calling Prompt.enterValues() method is optional, but DocumentInstance.setPrompts() method must be called in any case.
- The ViewModeType.ReportPart does not support OutputFormatType.HTML. Hence you cannot view a Web Intelligence document using ViewModeType.ReportPart and OutputFormatType.HTML in combination.
- The Lov setSearchMatchCase(boolean) call does not have any effect on the LOVs defined as "Delegated Search" in the universe, as they are searched directly from the database in a case-sensitive manner.
Important Disclaimers and Legal Information

Coding Samples

Any software coding and/or code lines / strings ("Code") included in this documentation are only examples and are not intended to be used in a productive system environment. The Code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the Code given herein, and SAP shall not be liable for errors or damages caused by the usage of the Code, unless damages were caused by SAP intentionally or by SAP’s gross negligence.

Accessibility

The information contained in the SAP documentation represents SAP’s current view of accessibility criteria as of the date of publication; it is in no way intended to be a binding guideline on how to ensure accessibility of software products. SAP in particular disclaims any liability in relation to this document. This disclaimer, however, does not apply in cases of willful misconduct or gross negligence of SAP. Furthermore, this document does not result in any direct or indirect contractual obligations of SAP.

Gender-Neutral Language

As far as possible, SAP documentation is gender neutral. Depending on the context, the reader is addressed directly with “you”, or a gender-neutral noun (such as “sales person” or “working days”) is used. If when referring to members of both sexes, however, the third-person singular cannot be avoided or a gender-neutral noun does not exist, SAP reserves the right to use the masculine form of the noun and pronoun. This is to ensure that the documentation remains comprehensible.

Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint about where to find related information. SAP does not warrant the availability and correctness of this related information or the ability of this information to serve a particular purpose. SAP shall not be liable for any damages caused by the use of related information unless damages have been caused by SAP’s gross negligence or willful misconduct. All links are categorized for transparency (see: http://help.sap.com/disclaimer).