PlanetJ WOW DataEngine

Programmer's Guide

Version 6.4

PlanetJ Corporation
9212 Mira Este Ct. Suite 203
San Diego, CA 92126

Table of Contents

TABLE OF CONTENTS....2

1. USING THE PLANETJ WOW DATAENGINE....6

1.1.Key Design Features....6
1.2.Architecture....7

2. INTRODUCTION TO JAVA AND JSP'S....7

2.1.Java Education Links:....7
2.2.Specialized Programming Online Resources....8
2.3.JSP Education Links:....9

3. CUSTOMIZING USER INTERFACES....9

3.1.How to Develop JSP's....13
3.2.WOW JSP Layout....14
3.2.1.Specifying Custom JSP's....15
3.2.2.Including Information From Signed-On User....18
3.2.3.Key Components of a Custom JSP....19
3.2.4.Detailed Example of Custom JSP....21
3.2.5.Adding a Button to a Custom JSP....27
3.3.Controlling Row Colors Programmatically....28

4. BUSINESS PROGRAMMING WITH WOW....29

4.1.Subclassing the WOW "Row" for Data Validation....29
4.2.Work Flow....29

5. SIGNON (AUTHENTICATION) AND USAGE LOGGING....31

6. CONNECTIONS....32

6.1.Connection Pooling....32
6.1.1.See If Connection Pool Exists....32
6.1.2.Creating a Connection Pool....32
6.1.3.Restarting a Connection Pool....33
6.1.4.Shutting Down a Connection Pool....33
6.2.Getting and Using a Connection....34
6.3.Freeing a Connection....34
6.4.Transaction Support....34

7. DATAENGINE....35

7.1.Executing Statements....35
7.1.1.Queries (Selecting Records)....35
7.1.2.Updates....36

7.1.3.Using Prepared Statements 36
7.2.Getting Database Objects 36
7.2.1.Retrieving All Libraries 36
7.2.2.Retrieving All Tables 36

8. SQLCONTEXT 36

8.1.Creating an SQLContext 37
8.2.SQLContext Attributes 37
8.2.1.Row Count 37
8.2.2.Set RowCollection Subclass (Type of RowCollection) 37
8.2.3.Set Row Subclass (Type of Row) 38
8.2.4.Set System Alias 38
8.2.5.Set Connection 38
8.2.6.Set SQL 38
8.2.7.Set AutoRefresh 39
8.3.SQLContext Operations 39
8.3.1.Adding/Removing Listeners 39
8.3.2.Cloning an SQLContext 39
8.3.3.Use Caching 40
8.3.4.Using Prepared Statements 40
8.4.Executing the SQLContext's SQL 41

9. ROWCOLLECTIONS 41

9.1.Creating a RowCollection 42
9.2.RowCollection Operations 42
9.2.1.Getting Next or Previous RowCollection 42
9.2.2.Sorting a RowCollection 43
9.2.3.Refreshing a RowCollection 43
9.3.Generating Files From a RowCollection 44
9.3.1.CSV File From a RowCollection 44
9.3.2.Microsoft Word File From a RowCollection 44
9.3.3.XML File From a RowCollection 44
9.3.4.PDF File From a RowCollection 45
9.3.5.FDF File From a RowCollection 45
9.4 Example of a Row Collection Operation 46
9.5 Operation Actions 51

10. ROWS 54

10.1.Row Subclasses 54
10.2.Creating a Row 55
10.3.Row Operations 56
10.3.1.Inserting a Row 56
10.3.2.Updating a Row 56
10.3.3.Deleting a Row 56
10.3.4.Cloning a Row 56
10.4.Retrieving a Row's Fields (FieldCollection) 57

10.5.Generating Files From a Row......57
10.5.1.CSV File From a Row......57
10.5.2.Microsoft Word File From a Row......58
10.5.3.FDF file from a Row......58
10.5.4.PDF file from a Row......58
10.5.5.XML File From a Row......59
10.6.Row Subclassing......59
10.6.1.Create Row SubClass......60
10.6.2.Set WOW to Recognize New Java Class......60
10.6.3.Overriding Methods......61
10.6.4.Another Example of Overriding Row and Field Methods......65
10.6.5.Multiple Methods and Parameters......66
10.6.6.Example of Overriding the Update Method in a Row Subclass......66
10.6.7.Example of Overriding the Insert Method in a Row Subclass......67
10.7.Row Actions......69
10.7.1. Example of Another RowAction......72
First Compulsory Method:......73
Second Compulsory Method:......74

11. FIELDS......76

11.1.Creating a Field......76
11.2.Field Classes......76
11.2.1.Default Class......76
11.2.2.Custom Class......77
11.2.3.Benefits of Using Custom Field Classes......80

12. WORKING WITH FIELDDESCRIPTORS (FD'S)......81

12.1.Creating......81
12.1.1.Auto Population of Field Data File......81
12.2.Retrieving an FD......82
12.2.1.FieldDescriptorManager......83
12.3.FD Operations......83
12.3.1.Cloning an FD......83

13. POSSIBLE VALUES (PV)......84

13.1.Creating Possible Values......84
13.1.1.Using an SQL Statement For Possible Values......84
13.1.2.Using a Key For Possible Values......85
13.1.3.Using a Value For a Possible Value......85
13.1.4.Using a Value and a Display Value For Possible Values......85
13.2.Creating a PV Class......86
13.2.1.Using Existing PV Classes......86
13.2.2.Create Custom PV Classes......86
13.3.Retrieving Possible Values......86
13.3.1.FieldDescriptor......87
13.3.2.PossibleValueManager......87

14. “MAGIC” REQUESTS....87
14.1.Requests....87
14.1.1.Page....87
14.1.2.Sort....88
14.1.3.Refresh....88
14.1.4.CSV/Excel....88
14.1.5.Microsoft Word....88
14.1.6.XML....88
14.1.7.PDF....88
15. REPORT BREAKS....88
15.1.Report Break Functions....89
15.2.Break Columns....89
15.3.Overall Report Breaks....89
15.4.Creating Report Breaks....90
16. HTML HELPERS....91
16.1.HTML Generator....91
16.2.HTML Extractor....91
16.3.HTMLComparisonInput....92
16.4.HTML Elements....93
16.4.1.HTMLTable....93
16.4.2.HTMLField....95
16.4.3.SimpleHTMLSelect....96
17. LOGGING WITH LOG4J....97
18. PLANETJ HELPERS....98
18.1.CSVHelper....99
18.2.DOCHelper 99
18.3.FDFHelper 99
18.4.PDFHelper....99
18.5.XMLHelper....99
18.6.QIFHelper....100
19. TRANSACTIONS....100
APPENDICES....100
1.Setting Up WOW With IBM’s RAD 6.0....100

1. Using the PlanetJ WOW DataEngine

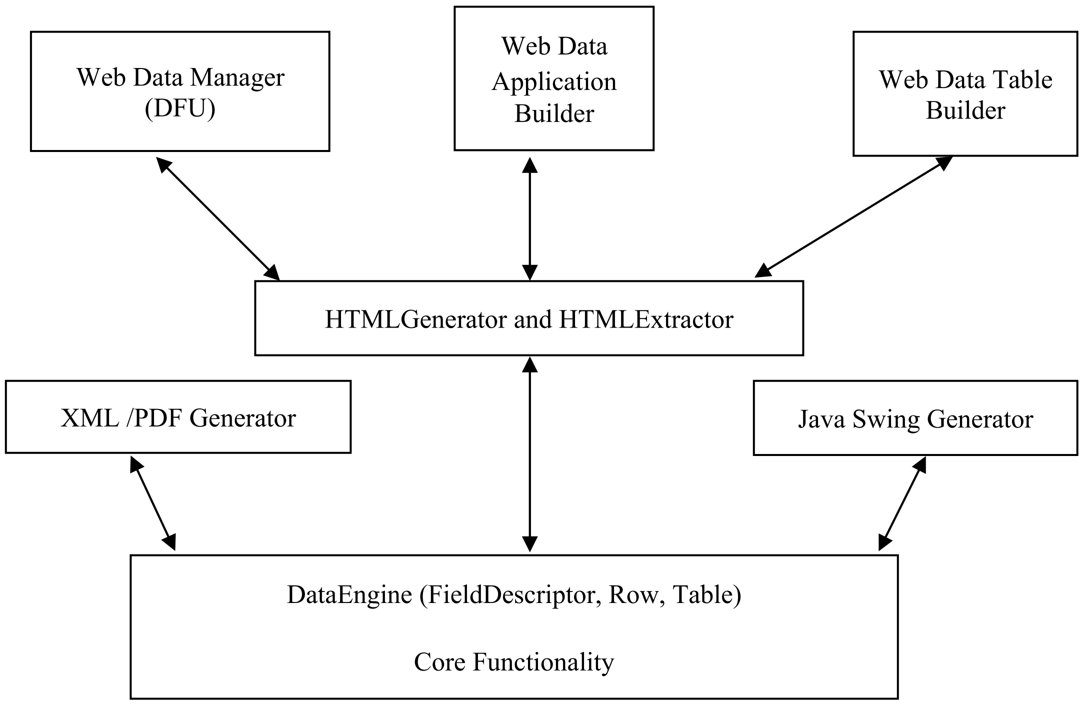
The WOW DataEngine is a set of Java based frameworks (APIs) designed to simplify the task of creating data centric applications. These APIs reside beneath the WOW product and allow WOW developers to accomplish any programming task possible with Java/JSP and HTML technologies. The WOW product is intended to allow businesses to easily develop powerful applications without the need to know and use complex technologies such as Java and JSP's. However, we do recognize that WOW will not provide all the features necessary to accomplish all tasks. Thus, WOW is architected in a very open manner and allows custom logic and custom views to be inserted at any point. This is accomplished by powerful code generation and framework technologies. This manual is intended for WOW developers that need to insert custom logic and custom views within their WOW application. Among other things, the DataEngine automatically manages the following features:

1. Extraction of relational database rows into business objects. These business objects can be used for Web based applications or desktop applications.
2. Update a Row in the database.
3. Delete a Row from the database.
4. Insert a Row into the database.
5. Connection pooling (operates with or without WebSphere)
6. Generation of HTML
7. Generates a List View of multiple Row objects.
8. Generates a Detail View of a single Row object.
9. Extraction of HTML data
10. Extract selected business objects from a List View.
11. Extract HTML detail data into a Row object.

1.1. Key Design Features

- Extremely easy to use (many features automatically generated)
- High performance
- NLS enabled
- Multi-user enabled
- ASP model enabled, Multi-company, multi-apps
- OO / MVC model
- UI independent (used for web and non-web apps)
- Customizable (config files to control internals operations)
- Highly productive for programmers and end users
- Usable against any JDBC database
- Designed for 2 or 3 tier operations

1.2. Architecture



2. Introduction to Java and JSP's

WOW is developed in 100% pure Java using standards-based technologies. Much of the user interface is developed in JSP (Java Server Pages). This document assumes the reader is familiar with Java and JSP technologies. If you are not, you should obtain education, which is available from a variety of sources.

2.1. Java Education Links:

The Sun Java Tutorials Site
http://java.sun.com/docs/books/tutorial/index.html

The Java Developer ConnectionSM
http://developer.java.sun.com

Code Conventions
http://java.sun.com/docs/codeconv/
Conventions that developers within Sun try to follow when writing programs in the Java programming language.

About.com Focus on Java
http://java.about.com
Has articles and links to many resources relating to Java

JavaWorld Magazine
http://www.javaworld.net
Has articles about current programming topics.

Java Boutique
http://javaboutique.internet.com
Has applets (some including source code), tutorials, book reviews, and more.

Java Developer's Journal
http://www.sys-con.com/java/
An online subset of a print magazine. Includes source code and selected articles.

2.2. Specialized Programming Online Resources

The J2EE™ Tutorial
http://java.sun.com/j2ee/tutorial/index.html
A beginner's guide to developing enterprise applications on the Java 2 Platform, Enterprise Edition SDK.

Java 3D™ API Tutorial
http://java.sun.com/products/java-media/3D/collateral/
Has descriptions and examples of the most commonly used features in the Java 3D API.

XML Technologies
http://java.sun.com/xml/tutorial_intro.html
Introduces XML and tells you how to use the Java XML APIs.

The JNDI Tutorial
http://java.sun.com/products/jndi/tutorial/
Tells you how to use the Java™ Naming and Directory Interface (JNDI) for associating names and attributes with objects.

JDC Tutorials
http://developer.java.sun.com/developer/onlineTraining/
Many tutorials, covering a wealth of topics. Subjects include JavaBeans, Enterprise JavaBeans, JDBC, and Java 2D.

The Swing Connection
http://java.sun.com/products/jfc/tsc/
The Swing engineering team's Web site. Has articles written both by Swing creators and by external programmers who use Swing.

The Swing/JFC FAQ
http://www.drye.com/java/faq.html
Unofficial answers to frequently asked questions about Swing.

Apple's Developer Site
http://developer.apple.com/java/

2.3. JSP Education Links:

JSP Resource Index
http://www.jspin.com/
Has tutorials, articles, everything to do with JSP's and web applications.

JSP Tags Getting Started
http://jsptags.com/gettingstarted/index.jsp
Beginning Steps to get on the right track in developing Java Server Pages.

JSP Insider
http://www.jspinsider.com/content.jsp
Free good resource for getting example JSP code and help.

Cetus Links
http://cetus-links.org/oo_javaserver_pages.html
A collection of links on objects & components relating to JSP's.

The Server Side
http://www.theserverside.com/
Articles, news, discussion, and other resources for JavaServer Pages.

JSP Tutorial
http://www.visualbuilder.com/jsp/tutorial/
The Visual Builder's JSP Tutorial

JSP and J2EE Design Tutorial
http://www.visualbuilder.com/jsp/design/
The Visual Builder's JSP Design Tutorial

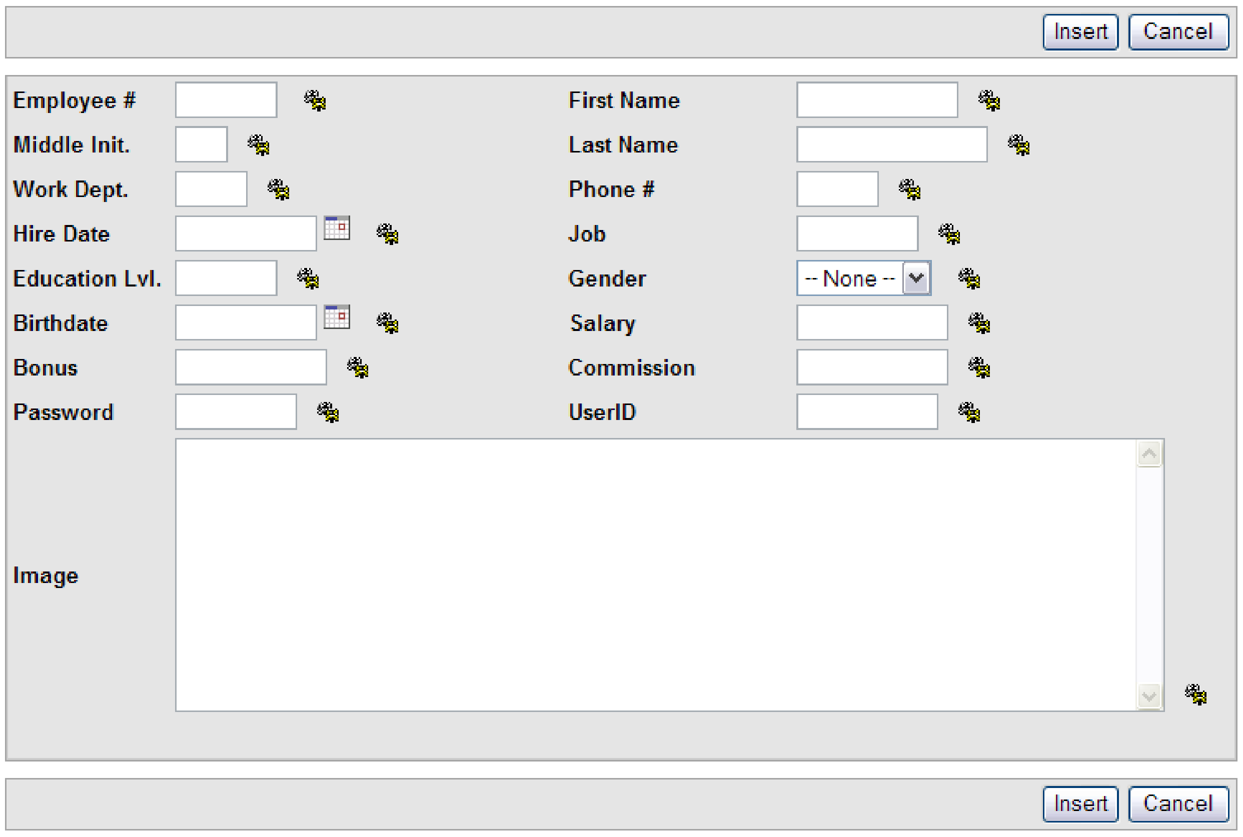
3. Customizing User Interfaces

All screens produced by WOW can be replaced with customized versions specific for each user. This can be done by creating custom JSP's and then instructing WOW to use the custom JSP for various operations.

You may need to produce a custom JSP when you need to provide UI behavior that is not possible with the base WOW Builder. NOTE: Color, font, images, etc can normally be customized through CSS. Refer to the CSS sections of this document as well as the WOW Builders Guide.

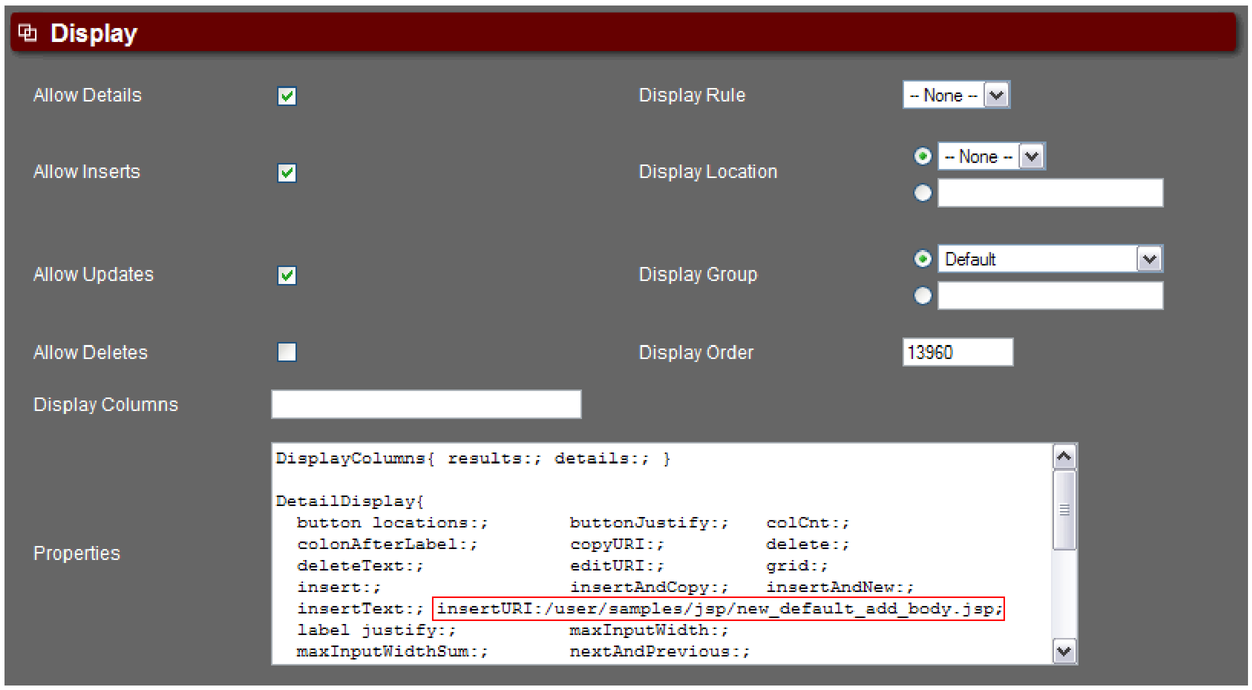
As a simple introduction to JSP customization, we will walk through the following example on creating a custom insert screen.

The screenshot below shows the default insert screen before any customization:



Employee #		First Name	
Middle Init.		Last Name	
Work Dept.		Phone #	
Hire Date		Job	
Education Lvl.		Gender	-- None --
Birthdate		Salary	
Bonus		Commission	
Password		UserID	
Image			

First, determine which operation is being used to display the insert button that goes to the insert screen you'd like to customize. In this example, that operation is *All Employees* (which selects from and inserts into the employee table). Edit the *All Employees* operation and specify the custom JSP (which we'll call *new_default_add_body.jsp*) in the Properties field as highlighted in the figure below.



Allow Details	✓	Display Rule	-- None --
Allow Inserts	✓	Display Location	● -- None -- ○
Allow Updates	✓	Display Group	● Default ○
Allow Deletes	☐	Display Order	13960
Display Columns			

Properties

```
DisplayColumns{ results:; details:; }

DetailDisplay{
  button locations:;          buttonJustify:;    colCnt:;
  colonAfterLabel:;           copyURI:;          delete:;
  deleteText:;                editURI:;          grid:;
  insert:;                    insertAndCopy:;    insertAndNew:;
  insertText:; insertURI:/user/samples/jsp/new_default_add_body.jsp;
  label justify:;             maxInputWidth:;
  maxInputWidthSum:;          nextAndPrevious:;
```

The original insert JSP can be found under
<Tomcat>/webapps/wow64/dataengine/jsp/default_add_body.jsp .
Make a copy of this file and rename it *new_default_add_body.jsp* . Move it to the following location (if this directory structure does not exist, create it):
<Tomcat>/webapps/wow64/user/samples/jsp .

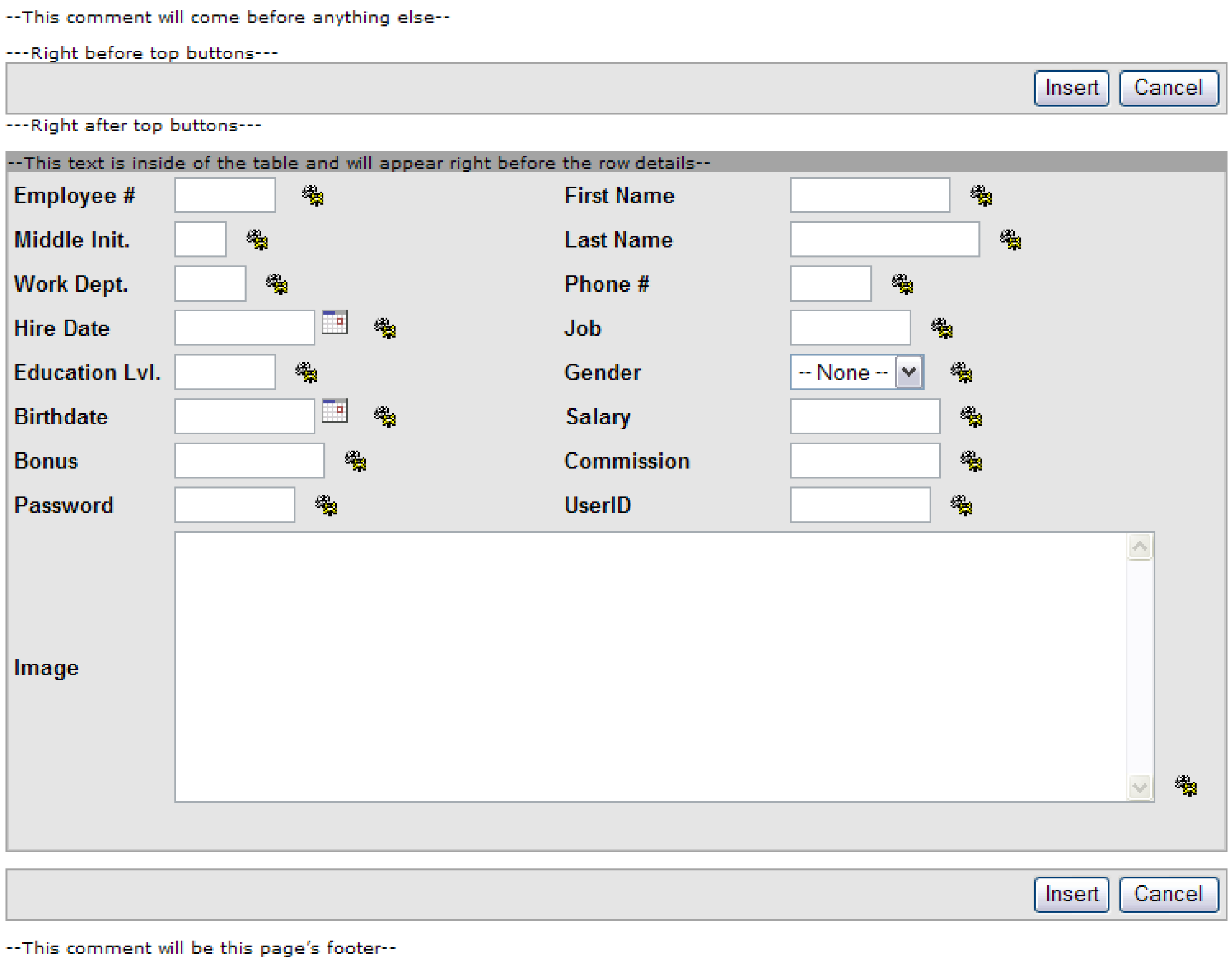
Now, open this new JSP, add the text highlighted in red below, and save. The full code from the new JSP (with customizations highlighted in red) is given below:

```
==================================================================================
new_default_add_body.jsp
==================================================================================

<%@ page import="planetj.database.*" %>
<%@ page import="planetj.dataengine.*" %>
<%@ page import="planetj.dataengine.display.*" %>

<%
Row row = (Row) DataEngineManager.getCurrentRow(request);
boolean includeTop = true;
boolean includeBottom = true;

if (row != null) {
    String addButtons = IJSPPages.DEFAULT_ADD_BUTTONS_JSP;
    // set flag to denote updateable
    request.setAttribute(IDataEngine.SHOW_UPDATEABLE_ROW, Boolean.TRUE);

    DetailDisplayPropertyGroup ddpg = (DetailDisplayPropertyGroup)
     row.getPropertyGroup(DetailDisplayPropertyGroup.DETAILS_DISPLAY);
    if (ddpg != null) {
        // check to see if should include buttons on top and bottom
        includeTop = ddpg.isShowButtons(DetailDisplayPropertyGroup.BUTTON_LOCATION_TOP);
        includeBottom =
            ddpg.isShowButtons(DetailDisplayPropertyGroup.BUTTON_LOCATION_BOTTOM);
```

```
      addButtons = ddpg.getAddButtons();

      if (IJSPPages.DEFAULT_ADD_BUTTONS_JSP.equals (addButtons)) {
         // check request
         String requestAddButtons = (String) request.getAttribute
          (DetailDisplayPropertyGroup.JSP_ADD_BUTTONS);
         if (null != requestAddButtons) {
            // use override from request
            addButtons = requestAddButtons;
         }
      }
   }
%>

<table cellpadding="0" cellspacing="10">
   <tbody>
      <tr>
         <td>--This comment will come before anything else--</td>
      </tr>

<% if (includeTop) { %>
       <tr>
         <td>
            ---Right before top buttons---
            <% request.setAttribute("GENERATETOPBUTTONS", new Object()); %>
            <jsp:include page="<%= addButtons %>" flush="true" />
            <% request.removeAttribute("GENERATETOPBUTTONS"); %>
            ---Right after top buttons---
         </td>
      </tr>
<% } %>
      <tr class="details-border">
         <td>
            <table cellpadding="0" cellspacing="1" border="0" width="100%">
               <tbody>
                  <tr class="details-body">
                     <td>
                        --This text is inside of the table and will appear right
before
                            the row details--
                        <% request.setAttribute(IDataEngine.ROW_KEY,
                            row.getNavigationKey()); %>
                        <jsp:include page="<%= IDataEngine.DEFAULT_ROW_DETAILS %>"
                            flush="true" />
                     </td>
                  </tr>
               </tbody>
            </table>
         </td>
      </tr>

   <% if (includeBottom) { %>
      <tr><td><jsp:include page="<%= addButtons %>" flush="true" /></td></tr>
   <% } %>

      <tr>
         <td>--This comment will be this page’s footer--</td>
      </tr>
   </tbody>
</table>
<% } %>
=================================================================================
```

Note that these customizations are all simple HTML. There was no Java coding needed. Study the JSP code and notice that it is simply a mixture of web standards (HTML, CSS, JavaScript, etc.) and Java code.

Next, run the application and relevant operation and click the Insert button to see the new insert screen with the custom text added.



--This comment will come before anything else--

---Right before top buttons---

---Right after top buttons---

--This text is inside of the table and will appear right before the row details--

Employee #		First Name	
Middle Init.		Last Name	
Work Dept.		Phone #	
Hire Date		Job	
Education Lvl.		Gender	-- None --
Birthdate		Salary	
Bonus		Commission	
Password		UserID	
Image			

--This comment will be this page's footer--

This is a very simple change of the insert JSP that merely adds text to show the basic process behind overriding a certain operation or process with a new custom JSP. Many more examples and advanced subjects will be provided in the following sections.

3.1. How to Develop JSP's

JSP development requires expertise in Java, JSP, and HTML technologies. JSP's can be developed with tools as simple as Notepad (or any text editor) or using more robust editors such as WebSphere Studio or Eclipse. If you are familiar with Eclipse or Websphere, we recommend either one (PlanetJ uses MyEclipse in-house). Both of these development environments are very powerful but have a large learning curve, especially Websphere. For simple customizations, Notepad will work just fine.

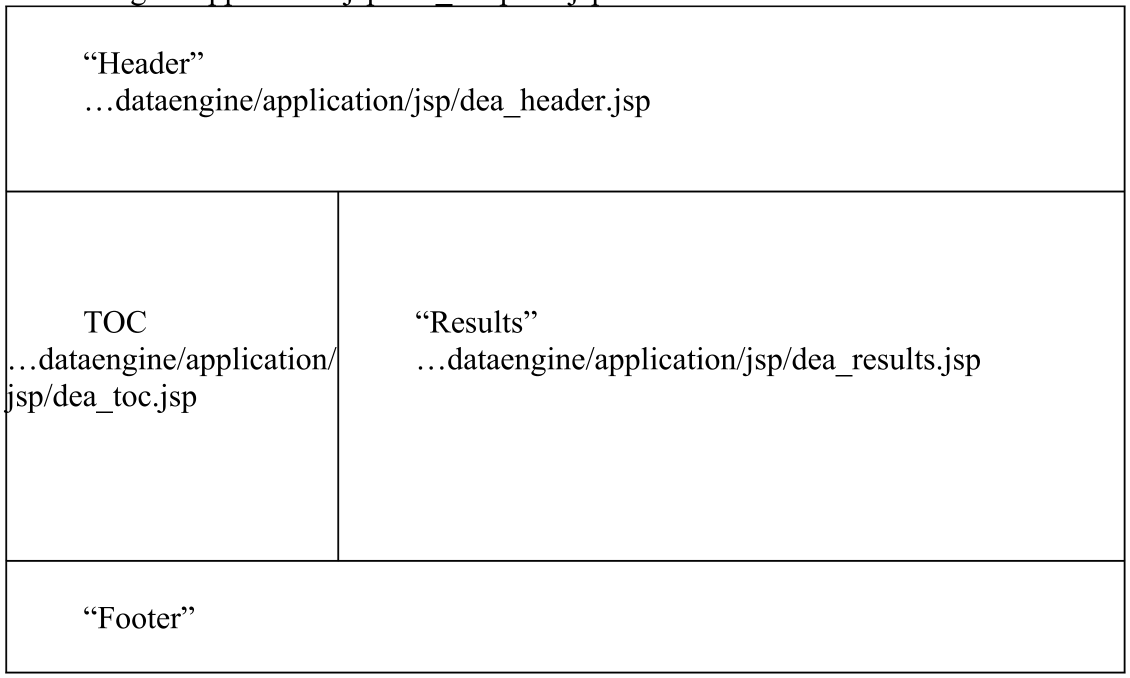
Developing with Notepad is as simple as copying an existing file and changing the text to include any customizations. JSP's must be compiled prior to use. Application Servers such as Tomcat and WebSphere can be configured to automatically compile JSP's when they are referenced. In Tomcat, the default is to compile. This means you can drop your new JSP into your webapp folder and Tomcat will compile it for you when you first start it. For more information, refer to the Tomcat optimization paper.

3.2. WOW JSP Layout

A "template" controls the general layout of WOW applications. The template brings together the various JSP components. Below are two of the more common templates:

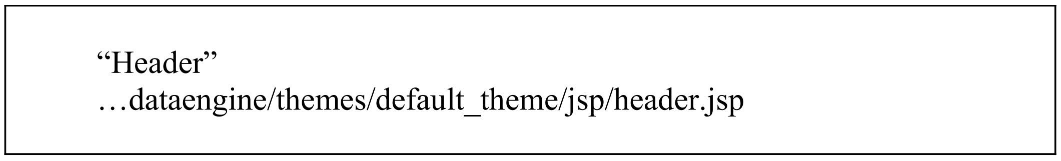
Left-hand TOC Layout:

…dataengine/application/jsp/dea_template.jsp


"Header"
…dataengine/application/jsp/dea_header.jsp
TOC
…dataengine/application/jsp/dea_toc.jsp
"Results"
…dataengine/application/jsp/dea_results.jsp
"Footer"

Top Navigation Layout:

…dataengine/themes/default_theme/jsp/tab_template.jsp



“Navbar” - … dataengine/themes/default_theme/jsp/dropDownNavBar.jsp
“Body” …dataengine/application/jsp/dea_results.jsp
“Footer”

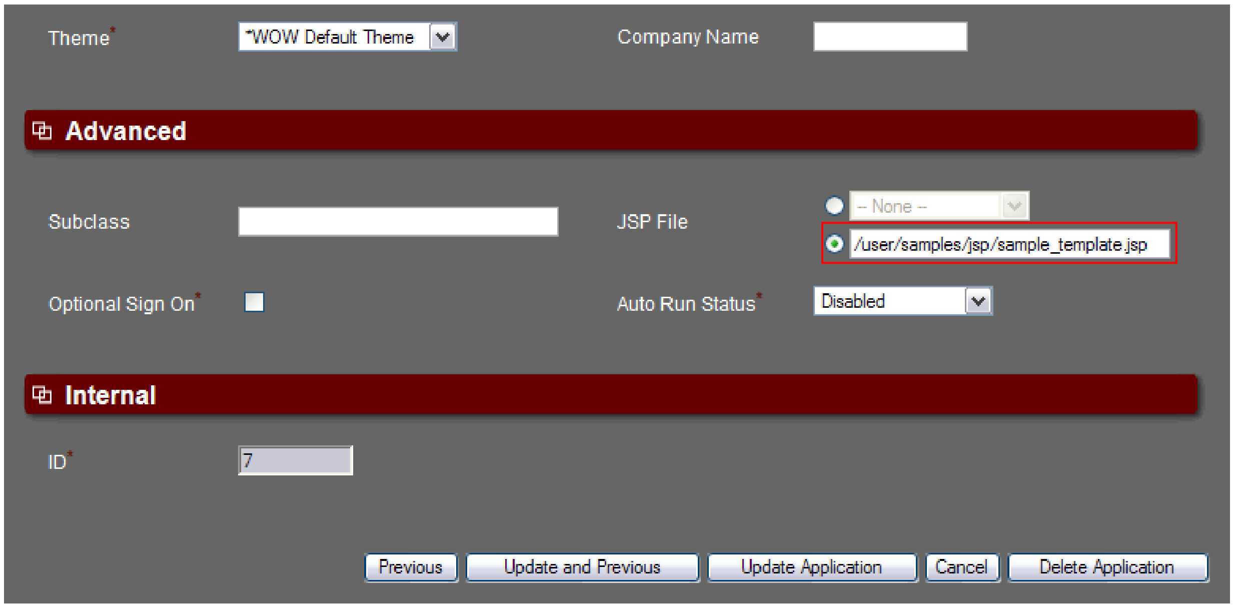
These files are found in the “context” of the web application server (*ex.* <Tomcat>/wow64/).

3.2.1. Specifying Custom JSP’s

When creating custom JSPs, copy the associated WOW file or use an existing “template” JSP provided with WOW. All sample JSP’s will be located in <Tomcat>/webapps/wow64/user/samples/jsp. The JSP can be added copied it into your file structure such as <Tomcat>/webapps/wow64/user/companyname/jsp. When a custom JSP is ready for use, it can be specified as shown below:

JSP’s Specified at the Application Level

At the application level you can specify a JSP that creates a template of how the application will look and react. For example, some of the JSP’s that are provided allow the user to use portals where different operations run all at the same time but in different screens. Grab these from a drop-down list shown below or input your own:


Theme* *WOW Default Theme
Company Name

Advanced

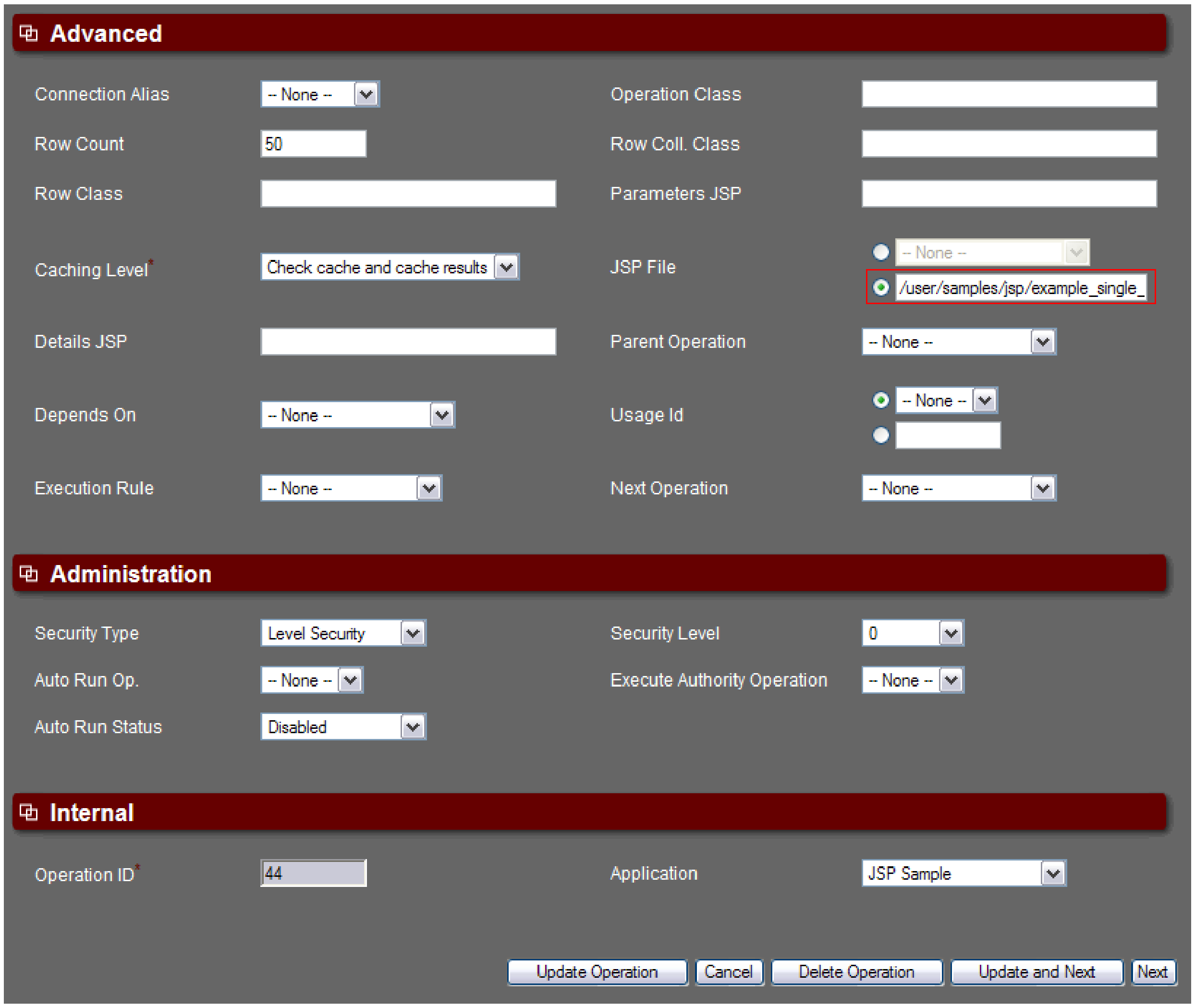
Subclass
JSP File -- None -- [✓ /user/samples/jsp/sample_template.jsp]
Optional Sign On* ☐
Auto Run Status* Disabled

Internal

ID* 7

JSP's Specified at the Operation Level

At the operation level there are a few ways to specify a JSP depending on what the JSP will be overriding or taking the place of. For example, if changing the look of the operation or its actions, set the custom JSP in the Advanced **::** JSP File section and set the URL "/user/samples/jsp/example_single_row.jsp" as shown in the following screenshot:


Advanced

Connection Alias	-- None --	Operation Class	
Row Count	50	Row Coll. Class	
Row Class		Parameters JSP	
Caching Level*	Check cache and cache results	JSP File	-- None -- / [✓ /user/samples/jsp/example_single_]
Details JSP		Parent Operation	-- None --
Depends On	-- None --	Usage Id	[✓ -- None --] /
Execution Rule	-- None --	Next Operation	-- None --

Administration

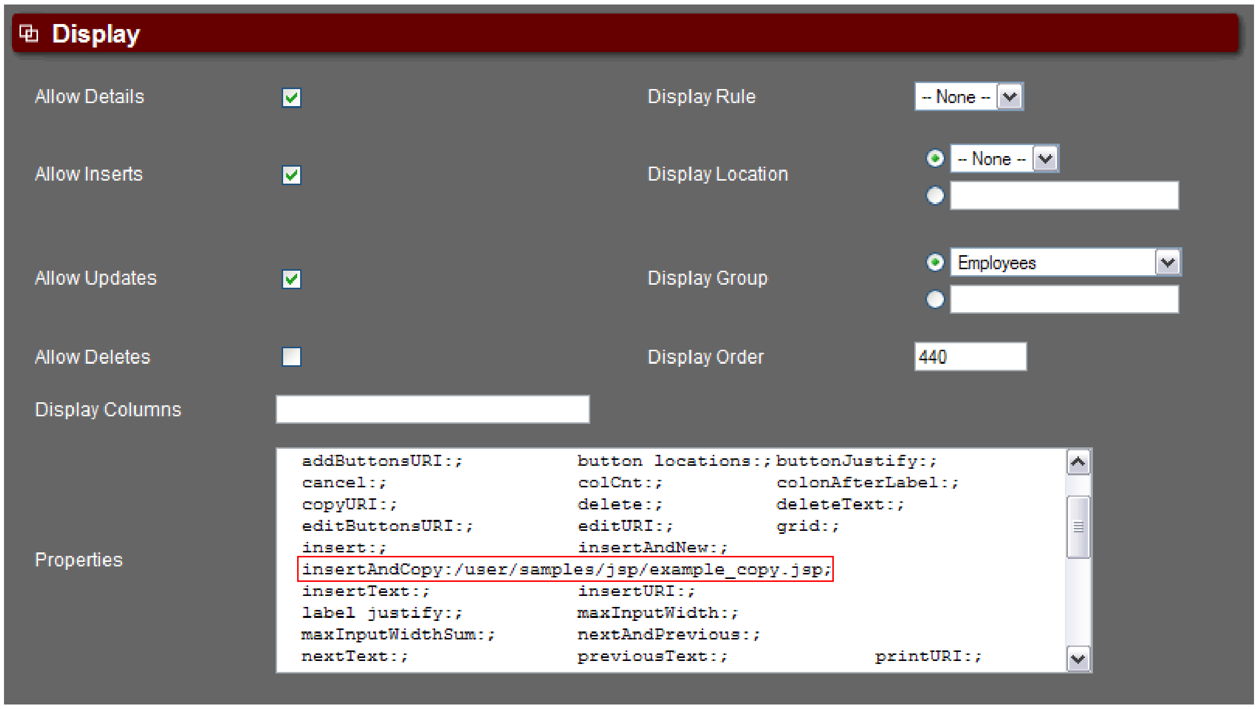
Security Type	Level Security	Security Level	0
Auto Run Op.	-- None --	Execute Authority Operation	-- None --
Auto Run Status	Disabled		

Internal

Operation ID*	44	Application	JSP Sample

Another way that custom JSP's are utilized at the operation level is to customize the screens associated with specific actions performed on an operation's data. For example, to customize the print, insert, copy, view and delete JSP's that are automatically controlled by WOW, find out which JSP is being called when each action is performed. Copy it and make any desired changes like tests, add-ons and/or comments. Also, look through the ready-made examples provided in the <Tomcat>/webapps/wow64/user/samples/jsp folder. Once you have created the custom JSP, look in the operation's Properties field and find that action's URL input area (ex. Insert:;) and input the path of the new JSP. The following screenshot demonstrates this.

In this example, there is an override of the copy action's JSP, which is called the InsertAndCopy action.



Allow Details	✓	Display Rule	-- None --
Allow Inserts	✓	Display Location	● -- None --
Allow Updates	✓	Display Group	● Employees
Allow Deletes		Display Order	440
Display Columns			

Properties

```
addButtonsURI:;           button locations:; buttonJustify:;
cancel:;                  colCnt:;           colonAfterLabel:;
copyURI:;                 delete:;           deleteText:;
editButtonsURI:;          editURI:;          grid:;
insert:;                  insertAndNew:;
insertAndCopy:/user/samples/jsp/example_copy.jsp;
insertText:;              insertURI:;
label justify:;           maxInputWidth:;
maxInputWidthSum:;        nextAndPrevious:;
nextText:;                previousText:;              printURI:;
```

3.2.2. Including Information From Signed-On User

If the user signed on with an Authentication Operation, they can access and include user information in any JSP.

Code Changes Needed for the JSP

In the custom jsp file that is displayed after the user signs-on, add something similar to the following line to create an instance of a Sign-On Receipt that holds the user's information.

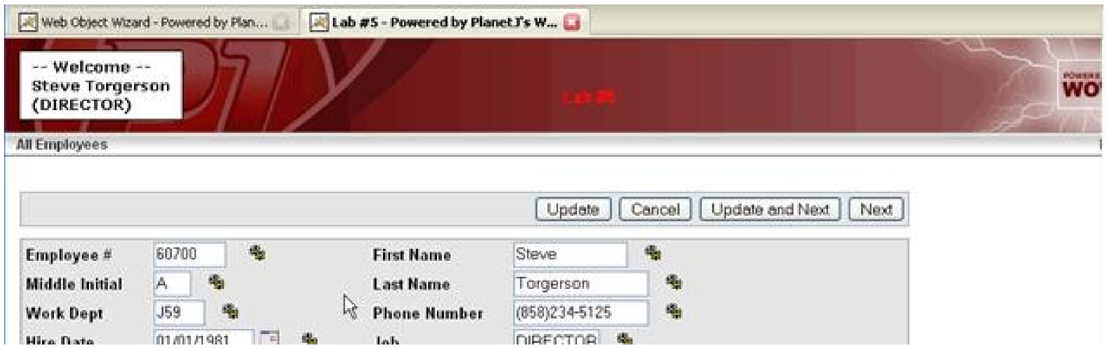
```
SignOnReciept testSignOnReceipt = new DataEngineManager.getUser(request);

String userid = testSignOnReceipt.getUserId;
```

The SignOnReceipt object (testSignOnReceipt) contains Java code that grabs data from the user row after they sign on. The getUserId method is then called to retrieve the User Id field from that user row.

Now, add HTML code similar to the following to display the User Id in your JSP.

```
<table><tr><td>Welcome, <%= userid %> </td></tr></table>
```

Below is an image of this custom JSP. It shows a small "Welcome" box in the upper left corner of the application, telling the user who they are currently logged onto the system as.



Employee #	60700	First Name	Steve
Middle Initial	A	Last Name	Torgerson
Work Dept	J59	Phone Number	(858)234-5125
Hire Date	01/01/1981	Job	DIRECTOR

3.2.3. Key Components of a Custom JSP

The following is a list of some of the key components that compose a typical custom jsp file:

Import statements

The first component of a JSP is to import any necessary code that is used within the JSP. For example, the planetj.magic project is needed for creating a magic request object:

```
<%@ page import="planetj.magic.*" %>
<%@ page import="planetj.dataengine.*" %>
<%@ page import="planetj.database.field.*" %>
<%@ page import="planetj.dataengine.operation.*" %>
<%@ page import="planetj.dataengine.display.*" %>
<%@ page import="planetj.dataengine.sqloperation.*" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
<%@ page import="java.math.*" %>
<%@ page import="planetj.database.*" %>
<%@ page import="planetj.html.*" %>
<%
```

Note: '`<%`' designates the code that follows is java. '`%>`' designates the code that follows is HTML.

Output a Single Row (Record)

To display a single row in your custom JSP, call an external method to read a single row. Methods for reading a row are mostly comprised of an SQL select statement. The generateDetails method will dynamically build HTML containing data from the row and the corresponding field labels.

```
HTMLRowDetails detailGenerater = new HTMLRowDetails();
detailGenerater.setUpdateable(false); // For display only
Row row = null;  // Holds the row (record)

/************************************************************************/
/* Display a row                                                        */
/************************************************************************/
row = MyJavaClass.getDataRow();   // Read the row
if (row != null) {       // A row was found?
   // Write the row
   out.println(detailGenerater.generateDetails(row, request, response));
}
```

Output a RowCollection (A Group of Rows)

To display a row collection in your custom JSP, call an external method to read a row collection. Methods for reading row collections are mostly comprised of an SQL select statement. The generateDetails method will dynamically build HTML containing data from each row and the corresponding field labels.

```
/*********************************************************************/
/* Display a Row Collection                                          */
/*********************************************************************/
HTMLRowDetails detailGenerater = new HTMLRowDetails();
Row row = null;  // Holds the row (record)
RowCollection myRC = MyJavaClass.getAllDataRows();
if (myRC != null && !myRC.isEmpty()) {
   // loop through rows to display
   int size = myRC.size();
   for (int i = 0; i < size; i++) {
      // get next location
      row = myRC.getRow(i);
      out.println(detailGenerater.generateDetails(row,
request,
response));
   }
}
```

Sample Method to Return a Single Row

```
/**
 * Get a Row for this file.
 */
public static Row getDataRow(String id) throws CMException {

   Row row = null;
   SQLContext context = new SQLContext("MYCONNECTION");
   context.setRowClass(Row.class);
   context.setSQL(
      "SELECT * FROM MYLIB.MYFILE WHERE FIELD1 = '" + id + "'");

   row = DataEngine.getRow(context);

   return row;
}
```

Sample Method to Return a Row Collection

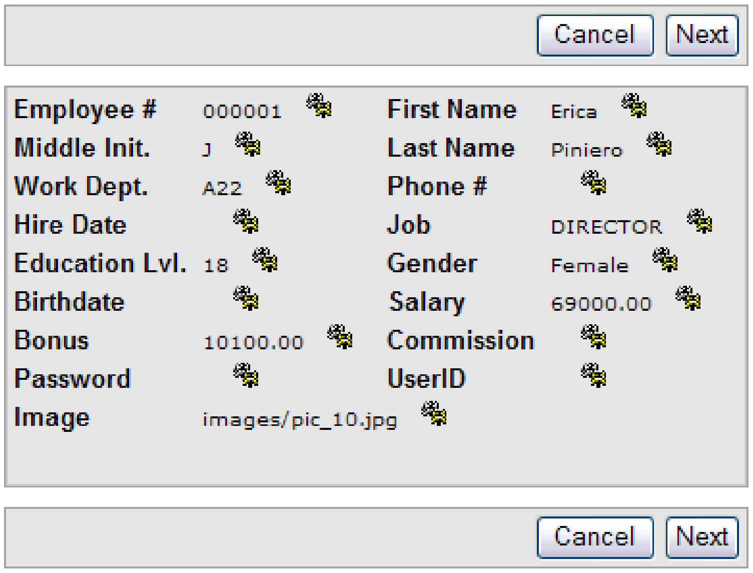
```
/**
 * Get a RowCollection populated with all records for this file.
 */
public static RowCollection getAllDataRows() throws CMException {

   RowCollection rc = null;
   SQLContext context = new SQLContext("MYCONNECTION");
   context.setRowClass(Row.class);
   context.setSQL("SELECT * FROM MYLIB.MYFILE");

   rc = DataEngine.getRows(context);

   return rc;
```

}

3.2.4. Detailed Example of Custom JSP

Custom Detail Display JSP

When looking at a results table in WOW, clicking on a 'View Record' icon (🔎) normally displays the data of that specific row:



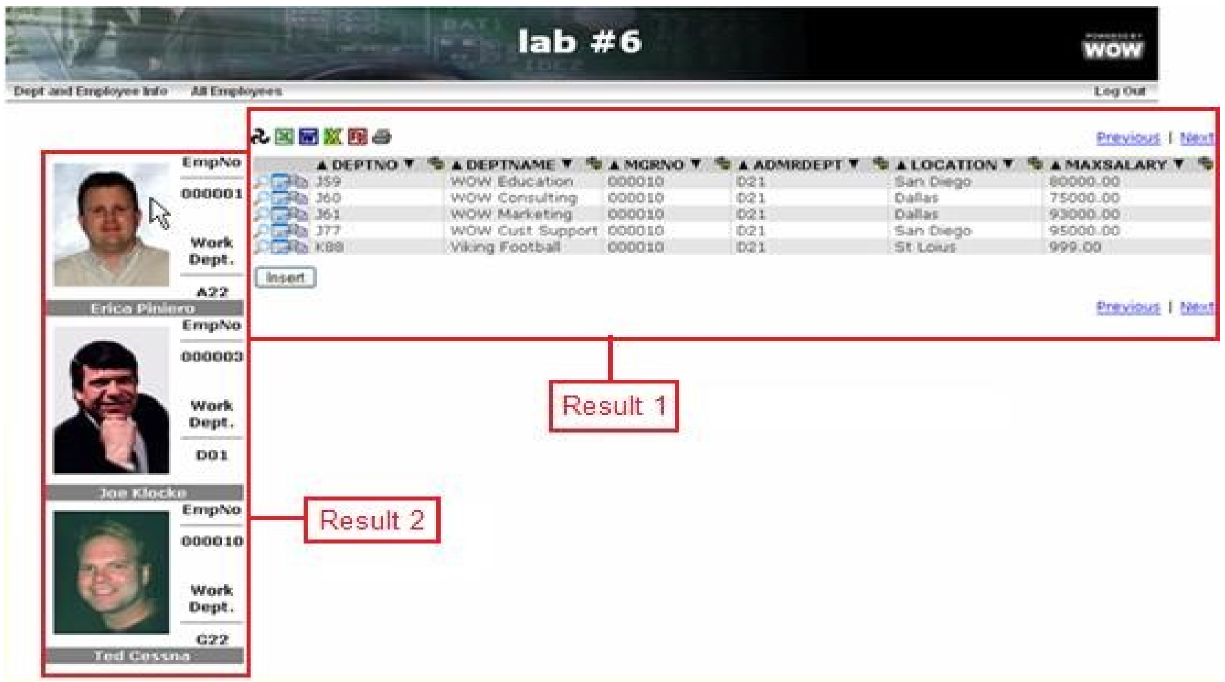
Employee #	000001	First Name	Erica
Middle Init.	J	Last Name	Piniero
Work Dept.	A22	Phone #	
Hire Date		Job	DIRECTOR
Education Lvl.	18	Gender	Female
Birthdate		Salary	69000.00
Bonus	10100.00	Commission	
Password		UserID	
Image	images/pic_10.jpg		

However, it is often necessary to customize this view and show different details of the row (maybe even pictures, links to more information, etc.). In this example, a new Details Display JSP will be created that shows an employee's information along with an image and information about that employee's department. The picture will also link to the employee's resume. To do this, create a new JSP called custom_detail_display1.jsp.

{!------------- JSP code needed for above example ---------------}

Customized Result Page JSP

Often times it is necessary to created a custom result page for your operations. This can be done rather easily by creating a JSP to control your results. In this example, a new Result Page JSP will be created that shows results from multiple tables (and multiple operations) on the same screen. On the left side of the screen we'll show the employee profiles, including images, while in the center body of the page we'll show departmental information. The results page will look something like this:

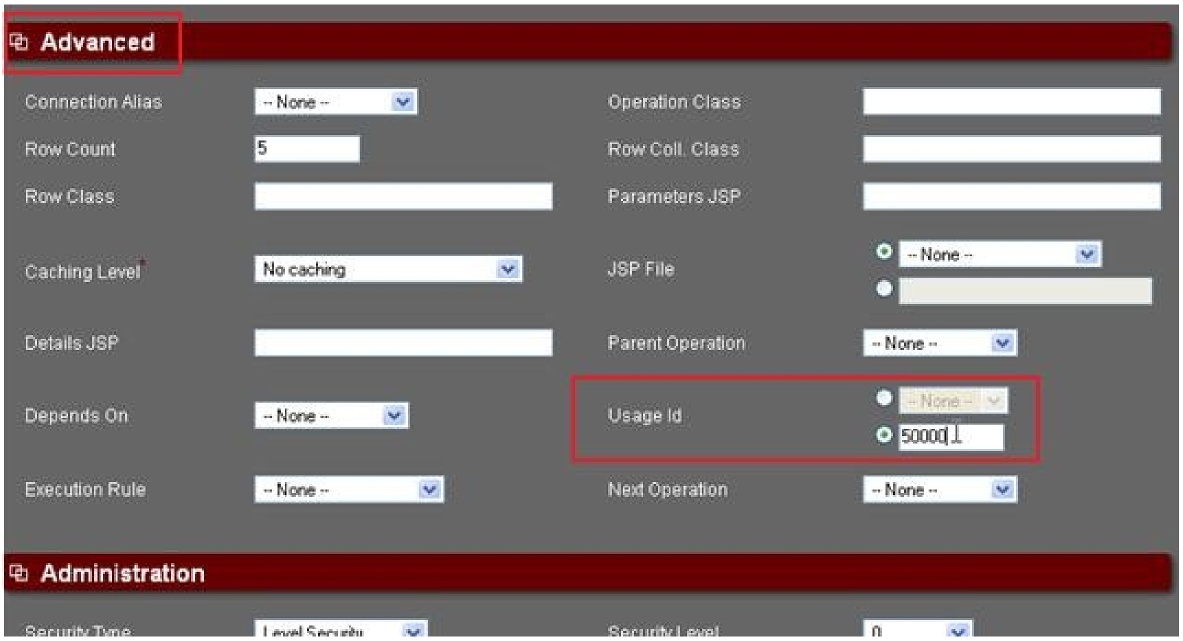


DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION	MAXSALARY
J59	WOW Education	000010	D21	San Diego	80000.00
J60	WOW Consulting	000010	D21	Dallas	75000.00
J61	WOW Marketing	000010	D21	Dallas	93000.00
J77	WOW Cust Support	000010	D21	San Diego	95000.00
K88	Viking Football	000010	D21	St Louis	999.00

To make your results screen turn out something like this, we have to create a new JSP file. The example JSP will sook something like this (notice that when we get an operation we are getting it by selecting the operation with a certain usage id, in this case 5000):

```
   /**********************************************************************/
   /* example_print1.jsp                                                 */
   /**********************************************************************/

   <%@ page import="planetj.dataengine.application.*" %>
   <%@ page import="planetj.dataengine.operation.*" %>
   <%@ page import="planetj.dataengine.display.*" %>
   <%@ page import="planetj.dataengine.*" %>
   <%@ page import="planetj.database.*" %>
   <%@ page import="planetj.magic.*" %>
   <%@ page import="java.util.*" %>

   // Get current executing Contesxt
ExecutingContext ec = DataEngineManager.getCurrentExecutingContext(request, response);

   // Get Current Application
Application app = DataEngineManager.getCurrentApplication(request);

   // Get Operation from usage id set on operation
Operation employeeOp = app.getOperationByUsageId(5000, ec);

   // Execute the department Operation and grab all rows
RowCollection employeeRC = (RowCollection) employeeRC.execute(ec);

boolean hasData = employeeRC != null && employeeRC.size() > 0;
if( hasData) {
// Create and Iterator to step through rows.
Iterator itr = employeeRC.iterator();
Row row = null;
------------------
------------------
}
else {
   <div> No Result Found </div>
}

<td>
<%=  HTMLField.appendDisplayValue(row.getField("empNo"), null, request, response); %>
</td>

<!- - include the normal WOW Results - ->
<jsp:include page="/dataengine/jsp/default_result.jsp" flush="true" />

<table>
  <tr>
    <td> Result of 1st Operation </td>  <td> Result of 2nd Operation </td>
  </tr>
</table>

<td width="30%"> Result of 1st Operation </td>  <td width="70%"> Result of 2nd
Operation </td>
```

The next step in creating an operation with a custom result page is to create the actual operations that are required to create the results. The first operation we will create is the operation we are grabbing by usage id number.

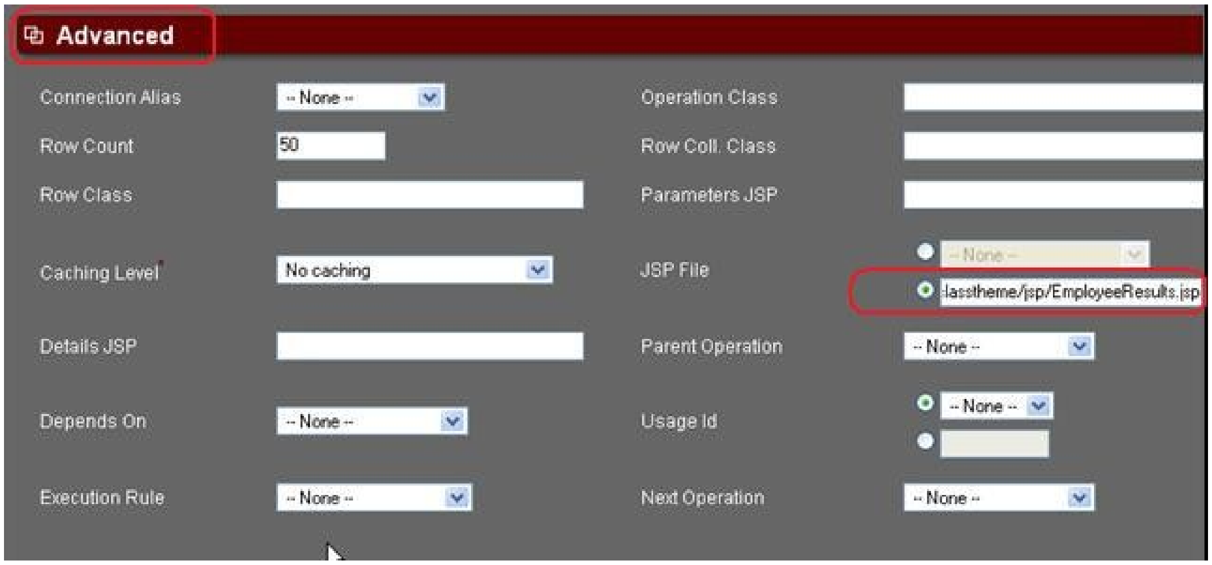


Connection Alias	-- None --	Operation Class	
Row Count	5	Row Coll. Class	
Row Class		Parameters JSP	
Caching Level*	No caching	JSP File	-- None --
Details JSP		Parent Operation	-- None --
Depends On	-- None --	Usage Id	50000
Execution Rule	-- None --	Next Operation	-- None --
Security Type	Level Security	Security Level	0

As you can see in the above screen shot (in the operations Advanced section) there is a field called Usage Id. You can set this operation's usage id here. Once you have set any operations Usage Id you'll be able to access this method by simply entering its usage id.

The next step to creating the custom result page is to create another operation that you will use to associate the JSP to the result page. To do this, create a new operation in your WOW Builder. For this example we made a SQL operation with the code:

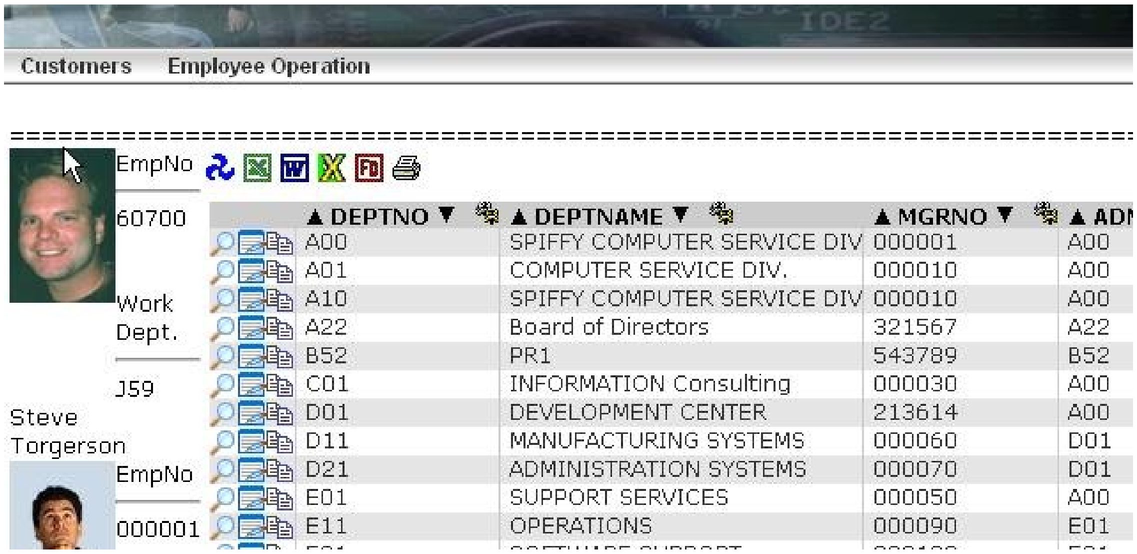
```
SELECT * FROM PJDATA.DEPARTMENT
```


Advanced

Connection Alias	-- None --	Operation Class	
Row Count	50	Row Coll. Class	
Row Class		Parameters JSP	
Caching Level*	No caching	JSP File	○ -- None -- / ● lasstheme/jsp/EmployeeResults.jsp
Details JSP		Parent Operation	-- None --
Depends On	-- None --	Usage Id	● -- None -- / ○
Execution Rule	-- None --	Next Operation	-- None --

Then, in the 'Advanced' section of the Builder, under 'JSP File' select the bottom radio button and enter the complete path of your results jsp and hit enter.

Now run the application and check the results. The result should be similar to the screenshot below, with the results of 2 different queries shown on the same screen. The left side is showing the results of the operation created with the usage id set (in this case set to 5000) while the right (or center body) of the page is showing the results of the operation that you associated the JSP with.



EmpNo
60700
Work Dept.
J59
Steve Torgerson
EmpNo
000001

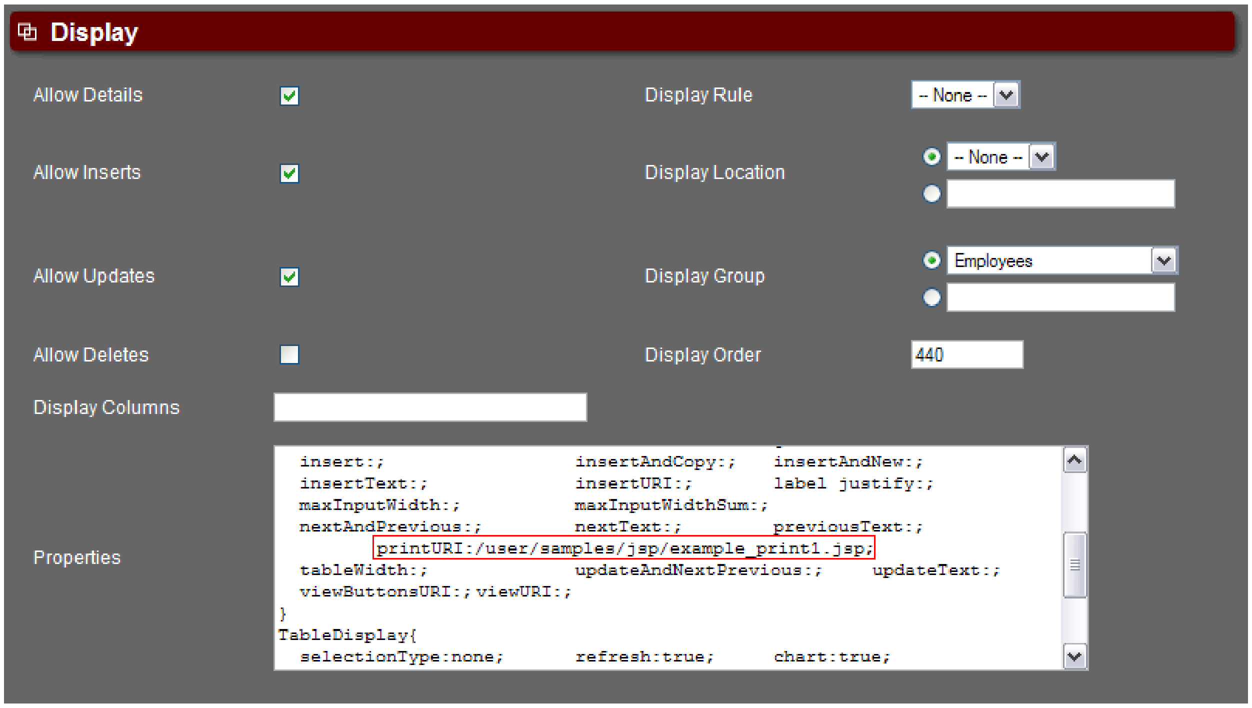
DEPTNO	DEPTNAME	MGRNO	AD
A00	SPIFFY COMPUTER SERVICE DIV	000001	A00
A01	COMPUTER SERVICE DIV.	000010	A00
A10	SPIFFY COMPUTER SERVICE DIV	000010	A00
A22	Board of Directors	321567	A22
B52	PR1	543789	B52
C01	INFORMATION Consulting	000030	A00
D01	DEVELOPMENT CENTER	213614	A00
D11	MANUFACTURING SYSTEMS	000060	D01
D21	ADMINISTRATION SYSTEMS	000070	D01
E01	SUPPORT SERVICES	000050	A00
E11	OPERATIONS	000090	E01

As you can see, the style of the page looks exactly the same as a regular WOW operation. This is fully customizable and can be edited by using CSS.

Customized Print Page JSP

When you view a result set in WOW and click on the print icon (🖨) to print those records, by default WOW will render a printable page that looks very similar to the result set page. This may not always be what you want though and may not give the user all the information that is needed on the print page. Here is an example JSP that overrides the original print JSP and puts in the company's logo, date, and company name on the print page.

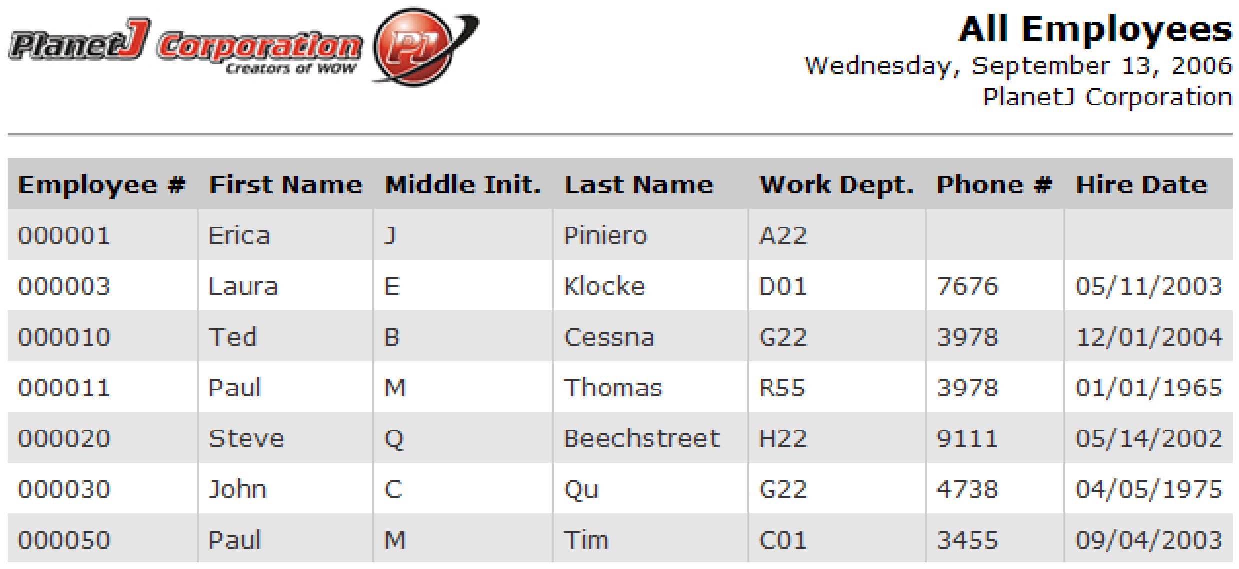
```
/*********************************************************************/
/* example_print1.jsp                                                */
/*********************************************************************/

<%@ page import="planetj.dataengine.*" %>
<%@ page import="planetj.database.*" %>
<%@ page import="planetj.html.*" %>
<%@ page import="java.util.*" %>
<%@ page import="java.text.*" %>


<%  RowCollection rc = (RowCollection)
    request.getAttribute(IDataEngine.ROW_COLLECTION);
    Date date = new Date(System.currentTimeMillis());
    SimpleDateFormat sdf = new SimpleDateFormat();

    HTMLTable results = null;

    if (rc != null) {
        results = new HTMLTable(IDataEngine.ROW_COLLECTION);
    }

    if (results != null) {
        results.setAllRowFunctions(false);
        results.setAllTableLinks(false);
        results.setDisplayGrid(false);
        results.setSelectionType(HTMLTable.NOSELECTION);
        results.setDisplayNextAndPrevious(false);
        results.setIgnoreRowCollectionProperties(true);
%>
<table>
    <tr>
        <td>
            <table cellpadding="0" cellspacing="0" border="0" width="100%">
                <tr>
                    <td rowspan="3">
                        <img src="user/samples/images/PJEmailLogo.jpg"
                            alt="PlanetJ Corporation">
                    </td>
                    <td align="right">
                        <b>
                            <font size=+1>
                                <%= rc.getTitle() == null ? "" : rc.getTitle()%>
                            </font>
                        </b>
                    </td>
                </tr>
                <tr>
                    <td align="right">
                        <% sdf.applyPattern("EEEE', 'MMMM' 'dd', 'yyyy"); %>
                        <%= sdf.format(date) %>
                    </td>
                </tr>
                <tr>
                    <td align="right">
                        PlanetJ Corporation
                    </td>
                </tr>
            </table>
         </td>
      </tr>
      <tr><td><hr /></td></tr>
      <tr>
         <td>
            <%= results.generateTable(rc, null, request, response) %>
         </td>
      </tr>
   </table>
   <% } else { %>
   <table>
      <tr>
         <td>
            No results to display.
         </td>
      </tr>
   </table>
   <% } %>

/*********************************************************************/
/* end example_print1.jsp                                            */
/*********************************************************************/
```
After creating the JSP, override the operation's default print page by adding the URL of the JSP to the printURL property in the Properties field (see following screenshot). This will direct WOW to use this JSP for this operation when the print button is clicked.



Allow Details ✓
Allow Inserts ✓
Allow Updates ✓
Allow Deletes ☐
Display Columns
Display Rule: -- None --
Display Location: -- None --
Display Group: Employees
Display Order: 440

Properties
```
  insert:;                  insertAndCopy:;   insertAndNew:;
  insertText:;              insertURI:;       label justify:;
  maxInputWidth:;           maxInputWidthSum:;
  nextAndPrevious:;         nextText:;        previousText:;
        printURI:/user/samples/jsp/example_print1.jsp;
  tableWidth:;              updateAndNextPrevious:;    updateText:;
  viewButtonsURI:; viewURI:;
}
TableDisplay{
  selectionType:none;       refresh:true;     chart:true;
```

Now run the application and then the operation. Click on the print icon and see the new print page.


All Employees

Wednesday, September 13, 2006

PlanetJ Corporation

Employee #	First Name	Middle Init.	Last Name	Work Dept.	Phone #	Hire Date
000001	Erica	J	Piniero	A22		
000003	Laura	E	Klocke	D01	7676	05/11/2003
000010	Ted	B	Cessna	G22	3978	12/01/2004
000011	Paul	M	Thomas	R55	3978	01/01/1965
000020	Steve	Q	Beechstreet	H22	9111	05/14/2002
000030	John	C	Qu	G22	4738	04/05/1975
000050	Paul	M	Tim	C01	3455	09/04/2003

All of the sample JSPs referenced in the Programmer's Guide can be found in the <Tomcat>/webapps/wow64/user/samples/jsp folder. Copy them and put them under your own user folder.

3.2.5. Adding a Button to a Custom JSP

To add a button to a custom JSP, a magic request must be implemented. In addition to changes to your custom JSP, it is necessary to create a magic request object (java file) which will be invoked by the new button/magic request. In this example, the name of the java source file will be called MyRequest.java and will be located in com.mypackage.

Code Changes Needed for the JSP

In the custom JSP file, add something similar to the following line to create an instance of the request object. The Request object (myRequest) will contain java code to be run when the button is pressed. The first parameter ("RequestName") in the request constructor call is simply a name given for the magic request. The new JSP file should be located in the JSP directory (usually under a user folder).

```
com.mypackage.MyRequest myRequest =
   new com.mypackage.MyRequest("RequestName",request,response);
```

Add HTML code similar to the following to add the new button. The button definition references the newly created request object as a parameter of the performMagic invocation.

```
<input type ="button" name="mybutton" value="Button Text"
   onClick='performMagic("<%= myRequest.getKey() %>")' />
```

Request File

The java class file (MyRequest) will contain the code that is run when the magic request button is clicked.

```
package com.mypackage;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import planetj.magic.*;

/*
 *==================================================================
 * Magic Request code
 *==================================================================
 */

public class MyRequest extends MagicRequest {

   // Constructor called when object is created.
   public StatementsSaveRequest(String pId,
       HttpServletRequest request,HttpServletResponse response) {

       super(pId, request, response);

    }

   // Executed by the magic request button defined in the JSP
   public IMagicRequest execute(HttpServletRequest request,
       HttpServletResponse response, DataEngineServlet servlet)
       throws CMException {

       // Add code to be run when the button is pressed.

       return this;
    }
}
```

3.3. Controlling Row Colors Programmatically

Controlling the color of rows presented in a table is often necessary. For example: making "overdue" to-do rows show up as red and making "due today" tasks show up as yellow. Using colors provides visual feedback to end users. Using standard WOW supplied CSS, it is possible to set the row color used for odd and even rows.

To programmatically control the background color of a Row, use the following steps:

1. Create a Row subclass of WOW's planetj.dataengine.Row
2. Implement the method "`getRowDisplayAttributes()`" in the new Row subclass.
3. In this method, implement whatever logic is necessary and return the CSS class name that will ultimately control the color of the row. The example below uses the class name "redRow".

4. Once the Row subclass is complete, assign this row subclass to the authentication operation.
5. Run the application to test the logic.

```
GetRowDisplayAttributes() {
   // Make priority 1 items red
   if   (getValueAsInt("priority")  == 1 ){
      return "redRow"; }
   else {return null;}
}
```

This example method will set append the returned CSS class to the HTML row's <tr> class attribute. Be sure to add the new CSS style class to one of your CSS files.

4. Business Programming With WOW

The following sessions provide examples and pointers for implementing common business tasks using WOW APIs along with the WOW Builder.

4.1. Subclassing the WOW "Row" for Data Validation

A Row or record represents a set of fields returned from an SQL query. To enable custom validation, the user can create a subclass of the WOW supplied "Row". Row methods are then overridden allowing the WOW programmer to customize data validation and behavior for insert, update and delete.

Here is a sample Row subclass containing one of the more commonly overridden methods:

```
public class MyRow extends Row {
/**
   * This method allows for altering the current row before an insert is performed.
   * The citystate field requires that the city value (chars 1-28) and the state
   * abbreviation (chars 29-30) be concatenated into a CHAR 30 value before insertion.
*/
   protected void preInsert(ExecutingContext exContext) throws CMException {
      super.preInsert(exContext);
      setCityStateValueAsCombinedField(exContext);    // Combines fields

   }

}
```

4.2. Work Flow

Work flow may require programmatic control in terms of what the next completed operation should be. For example, an international flight may require different sequence of steps then domestic. If an insurance quote is inserted and it is more than 10,000,000 then it requires additional info, etc. To accommodate these advanced work flow needs, the WOW Java based framework can be overridden allowing you to control the exact flow. The logical place to allow control is after Rows are inserted, updated, and deleted. This can be easily accomplished by overriding the appropriate Row methods. To programmatically change the next operation to be run, override Row.getNextOperationToRun(Operation, ExecutingContext) and return the number of the operation you want to run. NOTE: This requires the user to have Java skills and be familiar with the WOW Dataengine API.

In addition, to make WOW programming easier, new methods will be added to Row to hide the complexity of the Row, RowCollection, Operation interaction with navigation and the JSPs that will display them.

Low Level Design

New Methods

Method Name	Class	Comments
isWorkFlowHandler()	Row	Returns false by default. Subclasses return true if they will control the application flow. Requests will be able to ask Rows and then if the Row handles flow then the request will not forward to any JSPs.
openFor(MODE,LayoutProp)	Row	This method was created to simplify the user's job when opening a Row for display. This method will set keys and objects in the session, request, and navigation and then forward control to the appropriate JSP for UI handling. Mode represents the type of open for example (DISPLAY, EDIT) LayoutProperties allows the users to set titles, footers and control other UI elements for this particular UI display instance.
getJSPFor(MODE)	Row	Allows Rows and subclasses to specify what JSP should be used when opening this Row in the supplied Mode. A particular Row may always want to open with a special JSP. Default to the dataengine supplied default JSPs.
openForExecution(LayoutProp)	Operation	Allows programmers an easy way to say execute this operation. The operation is then smart enough to know what JSP to use, etc. This method sets up all needed objects in navigation, session, and request.

Example Work Flow Scenario:

1. The programmer will override Row.insert or Row.update
2. After a successful or failed insert or update, the programmer may want to direct flow to another Row or operation. For example:

```
Public int insert() {

  super.insert();

  Row nextWorkFlowRow = DataEngine.getRow("select * from x.y where this = that");

  LayoutProp.setTitle("Next you must edit your profile to add your VISA number");
  // This will cause the row to be opened in Update mode
  NextWorkFlowRow.openFor(UPDATE, layoutProp);

  }
```

When that Row is updated, it may direct flow to another operation or Row.

5. SignOn (Authentication) and Usage Logging

Controlling user access to applications often involves complex rules specific to each business. This may include integration with existing security schemes and databases. Usage of applications may also want to be tracked for security or usage fee reasons. For example, each user access to application x is recorded in a file and used for monthly invoices.

WOW provides the basic operations, but cannot generically handle all the custom behavior required. However, in all cases, WOW can be extended to provide this behavior. The major steps involved are listed below. The following example and directions apply when an application is secured using SQL Operation authentication. In this case, the application specifies "SQL Operation" for the security type and then specifies which Operation provides the authentication.

1. Create a Row subclass (ex. MySignonRow) of WOW's planetj.dataengine.security.http.SignonRow.
2. Implement the method "isOkToSignOn(…..) " in the new Row subclass.
3. In this method, implement ANY additional logic required to determine access rights as well as log user access. If the particular user should not have access rights to this application, an exception should be thrown. For example:

```
public boolean IsOkToSignOn(ExecutingContext ec) throws DataEngineException {
   if (MyCustomSecurityManager.isAuthorized(this) {    }
   else {
      throw new DataEngineException("You are not allowed in this application");
   }
   return true;  // User is authorized
}
```

4. Once the Row subclass is complete, create a SignOn operation with:

```
Operation Type = Authentication
```

```
Row Class = com.xxx.MySignonRow
```

5. Run the application to test the logic.

6. Connections

In order to perform any database operations, you need a java.sql.Connection. The DataEngine provides support for managing connections using connection pooling. Connection pooling is implemented in Java and therefore support for this feature is not tied to an application server.

The DataEngine encapsulates a java.sql.Connection into a DataEngineConnection. These connections are created and managed by the DatabaseManager in the form of connection pools.

6.1. Connection Pooling

DataEngineConnectionPools are managed by the DatabaseManager and can be referenced by their connection alias. DataEngineConnectionPools are based off of DBConnectionBroker, which is a freeware JDBCConnection pooling class available from www.javaexchange.com.

6.1.1. See If Connection Pool Exists

The DatabaseManager can be used to see if a connection pool already exists for a given connection alias. Calling doesConnectionPoolExist(String) and specifying the connection alias will return true if the connection pool already exists:

```
if (!DatabaseManager.doesConnectionPoolExist(“MYCONNECTION”)) {
    DatabaseManager.createConnectionPool(driver, system, user, password,
    1, 15, connRecreateSeconds, orphanTimeOutSeconds);
}
```

6.1.2. Creating a Connection Pool

Parameters Required to Create

The following parameters need to be supplied in order to create a DataEngineConnectionPool:

- JDBC driver
- URL of the server
- alias the connection pool will use
- user login name
- user password
- minimum number of connections to create upon pool creation
- maximum number of connections the pool allows
- number of seconds before the pool cleans up old connections
- number of seconds a connection is allow to be held for before it is reclaimed

Methods Called to Create

The DatabaseManager has five methods that can be used to assist in creating DataEngineConnectionPools. They are all called createConnectionPool(…) and each takes different parameters.

1. One takes in all 9 of the above mentioned parameters needed to create a connection pool.

2. Two others take almost all of the parameters with the exception of the connection's url and the alias. These are replaced with a DBSystem object (see Creating a DBSystem). The prior method taking all 9 parameters internally creates the DBSystem object .

3. Another method takes in the connection alias of the connection pool to create and the connection alias of the DataEngine Connection table which is used to retrieve all parameters necessary to create the DataEngineConnectionPool. **Note:** This only works if the Connection table exists and the table contains a record for the given connection alias.

4. The last method takes in a ConnectionRow containing all the information necessary to create the connection pool. A ConnectionRow is a record from the Connection table; however, it can be created itself (see Creating a ConnectionRow) without reading from the database by specifically setting its values and passing it to the DatabaseManager.

6.1.3. Restarting a Connection Pool

A DataEngineConnectionPool can be restarted by calling its reconnect() method. This gives any connections in use 10 seconds to be freed (see Freeing a Connection) before the connection pool is shut down and then restarted. If 10 seconds is not long enough, the method reconnect(int) can be called. This method set the number of milliseconds to wait before shutting down and restarting the connection pool.

```
DataEngineConnectionPool.reconnect(10);
```

Similarly, the DatabaseManager provides the same methods to restart connection pools except the DatabaseManager methods also require the DataEngineConnectionPool's connection alias to be specified.

```
DatabaseManager.reconnect(alias);
```

6.1.4. Shutting Down a Connection Pool

A DataEngineConnectionPool can be shut down by calling its destroy() method. This gives any connections in use 10 seconds to be freed (see Freeing a Connection) before the connection pool is shut down. If 10 seconds is not long enough, the method destroy(int) can be called. This method takes the number of milliseconds to wait before shutting down the connection pool.

```
DataEngineConnectionPool.destroy();
```

Similarly, the DatabaseManager provides the same methods to shut down connection pools except the DatabaseManager methods also require the DataEngineConnectionPool's connection alias to be specified.

```
DatabaseManager.destroy(alias);
```

6.2. Getting and Using a Connection

Connections can be retrieved from their connection pool by calling getConnection() or through the DatabaseManager by calling getConnection(…) and passing in the connection alias of the DataEngineConnectionPool. **Note:** When a connection is retrieved from a DataEngineConnectionPool it should be "freed" (see Freeing a Connection) when it is no longer used so that the connection can be used by someone else or for another operation.

```
Connection c = DatabaseManager.getConnection("MYCONNECTION");
```

In many cases, instead of directly obtaining a connection, only the connection alias (of the DataEngineConnectionPool you wish to connect to in order to carry out an operation) is needed. In these cases, the DataEngine will retrieve (and free) the connection itself.

6.3. Freeing a Connection

After connections are retrieved and used they should be returned or freed up for the next user's use. To free connections, you can use the DataEngineConnectionPool by calling freeConnection(…) and specifying the Connection you wish to free.

The DatabaseManager can also be used with the same method call and by also specifying the connection alias of the DataEngineConnectionPool you wish to free the connection from. The latter method call on the DatabaseManager saves time in that it does not need to get the connection alias from the connection.

```
DatabaseManager.freeConnection (connection);
```

6.4. Transaction Support

Connections can also be used for database transaction support using a single connection for multiple operations. First, obtain the connection, set its autocommit to false, perform operations on the connection, commit the connection, and then free the connection.

```
Connection conn = DatabaseManager.getConnection("MYSYSURL");
conn.setAutoCommit(false);
row1.insert (conn);
row2.update (conn);
row3.delete (conn);
conn.commit();
DatabaseManager.freeConnection(conn);
```

7. DataEngine

7.1. Executing Statements

The methods provided by DataEngine do the same thing as what could be done with regular SQL statement objects. The advantage to using the DataEngine, however, lies in what is done underneath the covers. If you make a call to a DataEngine.getRows() method or DataEngine.getRow() method, you don't need to go through all the hassle of getting a connection, creating a statement, executing the statement, error handling if something goes wrong while executing, parsing through ResultSets, etc. Instead, DataEngine handles all of this internally. For queries, a RowCollection object is returned than can be easily accessed, reused, manipulated, updated, and displayed in a variety of formats and/or technologies (Swing, HTML, PDF, etc).

7.1.1. Queries (Selecting Records)

A select statement can be executed by making a call to the DataEngine using the getRow() or one of the getRows() methods. These methods return a RowCollection object that contains one Row object for each record in the database that meets the criteria of the select statement.

```
// SQLContext
DataEngine.getRow(SQLContext)
DataEngine.getRows(SQLContext)

// Connection, SQL String, and optionally a Row subclass
DataEngine.getRows(Connection, String)
DataEngine.getRows(Connection, String, Class)

// System alias, SQL String, and optionally a Row subclass
DataEngine.getRows(String, String)
DataEngine.getRows(String, String, Class)
```

For example, the following method would return all the records in the table users in the library mylib from the database for the specified connection alias.

```
DataEngine.getRows (sysAlias, "select * from mylib.users");
```

If you would like to have the results be more intelligent and meaningful, then an SQLContext should be used. An SQLContext has a series of attributes and properties that can be set (see the SQLContexts section for more details on how to create and modify).

Also, if you wish to just receive a ResultSet rather than a RowCollection, the following method can be used. It requires that an SQLContext be passed in as a parameter.

```
DataEngine.executeQuery(SQLContext context)
```

7.1.2. Updates

For executing other SQL statements, the DataEngine provides several executeUpdate() methods. These methods work in a similar fashion as Statement.executeUpdate().

```
// SQLContext
DataEngine.executeUpdate (SQLContext)

// Connection, SQL String
DataEngine.getRows(Connection, String)

// System alias, SQL String
DataEngine.getRows(String, String)
```

For example, the following method would delete all records in the given table.

```
DataEngine.executeUpdate("myconnectionalias", "delete from mylib.users");
```

7.1.3. Using Prepared Statements

See Use Caching in the SQLContext section.

7.2. Getting Database Objects

7.2.1. Retrieving All Libraries

The static method getAllLibraries(DBSystem) on the DataEngine can be used to retrieve a Map of the Libraries for a DBSystem keyed by library name. If a Connection is not specified, then the Library objects are created for the given DBSystem by obtaining a connection for the DBSystem's connection alias.

7.2.2. Retrieving All Tables

The static method getAllTables(Library) on the DataEngine can be used to retrieve a Map of the Tables for the given Library keyed by table name. If a Connection is not specified, then the Table objects are created for the given Library by obtaining a connection for the Library's DBSystem's connection alias.

8. SQLContext

An SQLContext contains information about an SQL command, including parameters on how to run the command, the command itself, and how to organize the command's results. A context can be created and used to retrieve a RowCollection for displaying or manipulating data.

8.1. Creating an SQLContext

A context can be created by invoking the SQLContext's constructor with or without the system alias of the connection to use for execution. If the alias is not specified in the constructor, it will need to be set before executing the context's SQL statement.

```
SQLContext context = new SQLContext();
SQLContext context = new SQLContext(String);
```

8.2. SQLContext Attributes

An SQLContext contains many different attributes and properties that can be used to provide greater functionality when working with results. Most attributes and/or properties of an SQLContext are optional with the exception of providing its SQL String and connection alias or Connection to use when executing that SQL. If you don't specify a value for a property, then the SQLContext's default value for that property will be used.

8.2.1. Row Count

The context's row count is the number of Rows that are retrieved during an execution of an SQL query statement returned in the form of a RowCollection. This value defaults to SQLContext.ALL_ROWS, which means all Rows will be retrieved that meet the search criteria.

```
setRowCount(int);
```

Additionally, the following method can be used to retrieve the current Row count of the SQLContext.

```
getRowCount();
```

8.2.2. Set RowCollection Subclass (Type of RowCollection)

If you want the RowCollection (results) being returned to be of a certain type of RowCollection, specify the value on the SQLContext. This is used in a situation where you have a RowCollection subclass that contains methods that perform actions or tasks specific to a query. For example, a ReportRowCollection is a RowCollection that has a method for generating ReportRows when it is displayed. This value defaults to RowCollection.class.

```
context.setRowCollectionClass(RowCollectionSubClass.class);
```

Additionally, the following method can be used to retrieve the current RowCollection subclass of the SQLContext.

```
getRowCollectionClass();
```

8.2.3. Set Row Subclass (Type of Row)

If you want the RowCollection (results) being returned to contain a certain type of Row, specify the value on the SQLContext with the following method. This is used in a situation where a Row subclass contains methods that perform actions or tasks specific to data from the table or tables being read. This value defaults to Row.class.

```
context.setRowClass(RowSubClass.class);
```

Additionally, the following method can be used to retrieve the current Row subclass of the SQLContext.

```
getRowClass();
```

8.2.4. Set System Alias

The context's system alias is the alias of the connection pool from which to get a connection for executing any statements. This value is required if a Connection is not specified.

```
context.setSystemALias(String);
```

Additionally, the following method can be used to retrieve the current system alias of the SQLContext.

```
getSystemAlias();
```

8.2.5. Set Connection

If you currently already have a connection and wish to have the SQLContext execute using that connection, set it by using the following method. This value is required if the connection alias is not specified.

```
setConnection(Connection);
```

Additionally, the following methods can be used to retrieve the current Connection to use for executing the SQLContext's SQL String. The Boolean parameter specifies whether or not to create the Connection if it does not already exist.

```
getConnection();
getConnection(boolean);
```

8.2.6. Set SQL

The context's SQL is the String to be executed when requested. The given String should be in proper SQL format. This value is required.

```
context.setSQL ("select * from mylib.users");
```

Additionally, the following method can be used to retrieve the current SQL String set in the SQLContext.

```
getSQL();
```

8.2.7. Set AutoRefresh

The auto-refresh attribute specifies whether or not a RowCollection retrieved via an SQLContext should automatically refresh itself when its data becomes stale. When a RowCollection contains data from a table, and the DataEngine is used to update that table, then that RowCollection will refresh itself from the database automatically (if the auto-refresh property is true). If the table is updated outside of the DataEngine, the RowCollection will not refresh itself.

By default, this value is true.

8.3. SQLContext Operations

8.3.1. Adding/Removing Listeners

If a listener is added to an SQLContext it will be informed when the SQLContext is used to run an SQL statement. Usually, listeners are used when the statement is being run in the background (see Run in Background above). If the statement is not run in the background, the listeners are all informed of the result before the DataEngine method that caused the statement to run returns. Also the listener must be of type IDataEngineListener. Below are the two methods on an SQLContext used to add and remove listeners.

```
addListener(IDataEngineListener);
removeListener(IDataEngineListener);
```

To see if a context has listeners, the following method can be used.

```
hasListeners();
```

8.3.2. Cloning an SQLContext

Cloning might be used in a situation where you already have an SQLContext with several attributes already set. Rather than creating a new one exactly the same except with a different system alias or SQL String, the existing SQLContext can be cloned and the new properties set.

```
SQLContext clone = context.cloneContext();
```

The SQLContext's clone() method could be called as well. Internally it calls cloneContext() and returns the SQLContext as an Object.

8.3.3. Use Caching

By default, caching is not used. Accessing the database every time for a given SQL query could cause performance issues. This is where the caching of queries can be useful. You can specify on the SQLContext if you would like to check to see if results already exist for the given SQL or if you would like to cache the results retrieved from executing the SQL. There are several different caching states provided by the SQLContext that can be accessed as public static variables.

```
// Does not provide any caching support (SQLContext defaults to this)
SQLContext.CACHING_NONE

// If results have already been cached, use them
SQLContext.CACHING_CHECK

// Cache results after retrieval for future reference
SQLContext.CACHING_STORE

// Provides the caching functionality of both check and store
SQLContext.CACHING_CHECK_AND_STORE
```

The following methods can be used to set and retrieve the current caching state of the SQLContext. Setting the caching level sets the caching state directly, but check cache and store into cache attributes can also be set and retrieved individually.

```
// Setting or retrieving caching state
setCacheLevel(int)
getCacheLevel()

// Set or retrieve whether or not to check cache for existing results
setCheckCache(boolean)
isCheckCache()

// Set or retrieve whether or not to cache results after retrieved
setCacheResults(boolean)
isCacheResults()
```

8.3.4. Using Prepared Statements

The DataEngine supports the use of prepared statements. If you wish to use a prepared statement, pass the SQL to the setSQL() method of an SQLContext as usual, and set the UsePreparedStatement property to true. By default, no Statements are prepared. This value should only be set to true when an SQL String containing a prepared statement has also been set.

```
SQLContext context = new SQLContext().setSystemAlias ("mySystem");
context.setSQL ("select * from planetj.customers where id = ?");
context.setUsePreparedStatement (true);
```

You can use the setPreparedStatementValue() value method to provide values for the parameters in the prepared statement.

```
context.setPreparedStatementValue (0, new Integer (52));
```

When the SQLContext in the above example is passed to the DataEngine.getRows() method, the query will run in a prepared statement. First, the DataEngine will look for a connection in which the SQL statement has already been prepared; if such a connection exists, and is available, then that connection will be used to execute the statement. If no such connections are available, the statement will be prepared and executed in the connection that has the fewest prepared statements.

The maximum number of prepared statements that can exist in a single connection is specified by the MaxPreparedStatements property of the DataEngine. If a statement is prepared using a connection that already has the maximum number of prepared statements, the prepared statement which has been unused for the longest amount of time is closed, thus preserving the number of prepared statements in the connection.

8.4. Executing the SQLContext's SQL

Static methods in DataEngine have been provided for executing the context's SQL after it has been created, modified, and prepared.

The following methods can be used for executing a query.

```
// Retrieve a RowCollection
DataEngine.getRows (SQLContext)

// Retrieve a single Row
DataEngine.getRow(SQLContext)

// Retrieve a ResultSet
DataEngine.executeQuery(SQLContext)
```

The following method can be used for executing other statements.

```
// Executes the statement
DataEngine.executeUpdate(SQLContext)
```

9. RowCollections

A RowCollection is a group of Row objects, representing rows in the database. Typically, a RowCollection is obtained by invoking one of the getRows() methods of the DataEngine class.

Important methods of a RowCollection include:

- getRowCount() which returns the number of Rows in the RowCollection
- getRow(int) which returns the Row at the specified index.
- GetRows() which returns a list of Rows

9.1. Creating a RowCollection

In most cases, the DataEngine will create and populate RowCollections for you. However, you can create a RowCollection and add Rows to it from a ResultSet using the addRows(int, ResultSet) method. When invoking this method, you must specify the number of rows from the ResultSet to add:

```
RowCollection rc = new RowCollection();
rc.addRows (numberOfRows, myResultSet);
```

You can also specify which row of the ResultSet should be the first row added to the RowCollection with the addRows(int, int, ResultSet) method:

```
RowCollection rc = new RowCollection();
rc.addRows (firstRow, numberOfRows, myResultSet);
```

9.2. RowCollection Operations

9.2.1. Getting Next or Previous RowCollection

In many cases, a RowCollection contains a subset of the rows that were returned from a query. The number of Rows that you want in a RowCollection can be specified with the setRowCount() method of the SQLContext class. In this example, the returned RowCollection will have no more than 20 rows:

```
SQLContext context = new SQLContext();
context.setRowCount (20);
context.setSQL ("select * from mylib.users");
context.setSystemALias ("mySystem");
RowCollection myRowCollection = DataEngine.getRows (context);
```

If there are more than 20 rows in the users table, the row collection will only contain the first 20 rows. If there are fewer than 20 rows in the table, then the RowCollection will contain all of the rows.

To check and see if there were more rows returned by the query that were not included in the RowCollection, you can use the hasNextRowCollection() method. To actually get the next group of rows, invoke the getNextRowCollection() method:

```
RowCollection nextRowCollection = null;
if (myRowCollection.hasNextRowCollection())
   nextRowCollection = myRowCollection.nextRowCollection();
```

The "next" RowCollection always contains the same number of Rows that were retrieved into the original RowCollection, unless there aren't enough rows in the database.

The hasPreviousRowCollection() and getPreviousRowCollection() methods work in a similar fashion.

In many cases you will not have to directly invoke these methods – when HTML is generated for a RowCollection, next and previous links are automatically generated as well. When the user clicks on these links the DataEngine invokes the appropriate methods for you, and displays the next or previous RowCollection to the user

9.2.2. Sorting a RowCollection

The most common way to sort a RowCollection is from the GUI using the table header up {▲} and down {▼} arrows. The up arrow will sort that column in an ascending order; likewise the down arrow will sort that column in a descending order. Currently this is a magic function (See Magic Request Sort).

The logic under the covers is pretty basic. Currently the sort works in two different ways. If the RowCollection is not full, meaning it does not contain the same amount of Rows retrieved from the database (this is internally called `hasNextRowCollection()` and/or `hasPreviousRowCollection()`), then a sort must be done on the database side. This involves appending an "order by" statement to the SQL String and re-running the query by doing a RowCollection.refresh(). Performance is considerably affected due to the IO/database calls. The other method of sorting is in a memory sort which performs much faster than the database sort. The internal sort uses the Collection.sort(List, Comparator) method. To sort a RowCollection, call the following method passing it a String[] of column names to sort by and a sort order (SortRequest.ASC or SortRequest.DESC).

```
RowCollection.sort(String[], sortOrder);
```

Future Sorting Requests

Currently Rows of a RowCollection are sorted by a single column (single level sorting). To enhance this, we would like to pass in String[] of column names, and sort by the order which the column names are specified in the Array (i.e. sort by String[0], then sort by String[1] etc…). The API is currently set up to support this.

9.2.3. Refreshing a RowCollection

From time to time, you may want to refresh your RowCollection to ensure you're viewing the latest data. The most common way to do this is from the GUI by clicking on the refresh pinwheel{}. This will refresh the display with the values stored in the database. Currently, refreshing is a magic function (See Magic Request Refresh). Refresh is also done upon sorting of the RowCollection. Below is the method to invoke if you wish to refresh a RowCollection.

```
RowCollection.refresh();
```

9.3. Generating Files From a RowCollection

9.3.1. CSV File From a RowCollection

If you wish to write a table/file out to a file for analysis, it can be painful. How do you do it? The DataEngine conveniently accomplishes this by writing the data out to a CSV (Comma Separated Values) file. A CSV file can be read by Excel or many other application to do further analysis (graphs, charts, etc…). The most common way to write an HTML table to a CSV file is by clicking on the Excel icon {}. Currently this is a magic function (See Magic Request CSV/Excel). (See CSVHelper)

In order to write a RowCollection to a CSV file, create a CSVFileDescriptor that defines attributes about a CSV file.

```
CSVFileDescriptor descriptor = CSVHelper.singleton().newCSVFileDescriptor();
descriptor.setDelimiter('¤'); // Delimiter default to a comma {,}
descriptor.setFileName("C:\\Temp\\Test.csv");

// If you want to append to an existing CSV file
descriptor.setAppendToFile(true); // Defaults to false

// If you want all the output on a single line
descriptor.setOutputAsSingleLine(true); // Defaults to false

// If you want each new row to begin with a delimiter
descriptor.setLeadingDelimiter('×'); // Default to an empty space ""
```

After your CSVFileDescriptor is all set up, invoke the toCSV() method passing in a Boolean true if you want to include the column heading as the first row in the CSV file and the CSVFileDescriptor.

```
RowCollection.toCSV(includeColumnHeadings, descriptor);
```

9.3.2. Microsoft Word File From a RowCollection

To produce a Microsoft Word file (.doc) from a RowCollection, click on the Word icon {}. Currently this is a magic function (See Magic Request Microsoft Word). (See DOCHelper) (See CSVHelper)

In order to generate the Microsoft Word file (.doc), call the toDOC method of RowCollection. A CSV file is generated under the covers and passed into word. For more information on generating a CSV file from a RowCollection See Generating a CSV file from a RowCollection.

```
RowCollection.toDOC(includeColumnHeadings, descriptor);
```

9.3.3. XML File From a RowCollection

To view RowCollections data in XML format you can click on the XML icon {}. Currently this is a magic function (See Magic Request XML). (See XMLHelper)

In order to write a RowCollection to a XML file, create an XMLFileDescriptor that defines attributes about an XML file.

```
XMLFileDescriptor descriptor = XMLHelper.singleton().newXMLFileDescriptor();

// to write the XML data out to a file, set a file name
// property in the descriptor.  if this file name is set, then the
// RowCollection's XML data will be output to the file.  if no file
// name is set, then it will be output to the browser.
descriptor.setFileName("C:\\Temp\\Test.xml");
```

After your XMLFileDescriptor is all set up, invoke the `toXML()` method passing in the descriptor.

```
RowCollection.toXML(descriptor);
```

9.3.4. PDF File From a RowCollection

To view RowCollections data in PDF format you can click on the PDF icon {}. Currently this is a magic function (See Magic Request PDF). (See PDFHelper)

In order to write a RowCollection to a PDF file, create a PDFFileDescriptor that defines attributes about a PDF file.

```
PDFFileDescriptor descriptor = new PDFFileDescriptor();

// to write the PDF data out to a file, set a file name
// property in the descriptor.  If this file name is set, then the
// RowCollection's PDF data will be output to the file.  If no file name
// is set, then it will be output to the browser's output stream as shown
// below.
descriptor.setFileName("C:\\Temp\\Test.pdf");

// to display this PDF in a browser using the Adobe Acrobat browser
// plug-in, then set the OutputStream of the descriptor like this.
descriptor.setOutputStream(response.getOutputStream());
```

After your PDFFileDescriptor is all set up, invoke the `toPDF()` method passing in a Boolean true if you want to include the column heading as the first row in the PDF file and the PDFFileDescriptor.

```
RowCollection.toPDF(includeColumnHeadings, descriptor);
```

9.3.5. FDF File From a RowCollection

This is normally used along with a PDF template to plug the FDF data into. (See FDFHelper)

In order to write a RowCollection to an FDF file and display that data in a template, create an FDFFileDescriptor that defines attributes about an FDF file.

```
FDFFileDescriptor descriptor = FDFHelper.singleton().newFDFFileDescriptor();

// set the template so the FDF file knows which PDF to open
```

```
   fileDescriptor.setPDFFileName("http://" + request.getRemoteHost() + ":" +
      request.getServerPort() + "/Template.pdf");

   // to write the FDF data out to a file, set a file name
   // property in the descriptor.  If this file name is set, then the
   // RowCollection's FDF data will be output to the file.  If no file name
   // is set, then it will be output to the browser's output stream as shown
   // below.
   descriptor.setFileName("C:\\Temp\\Test.fdf");
```

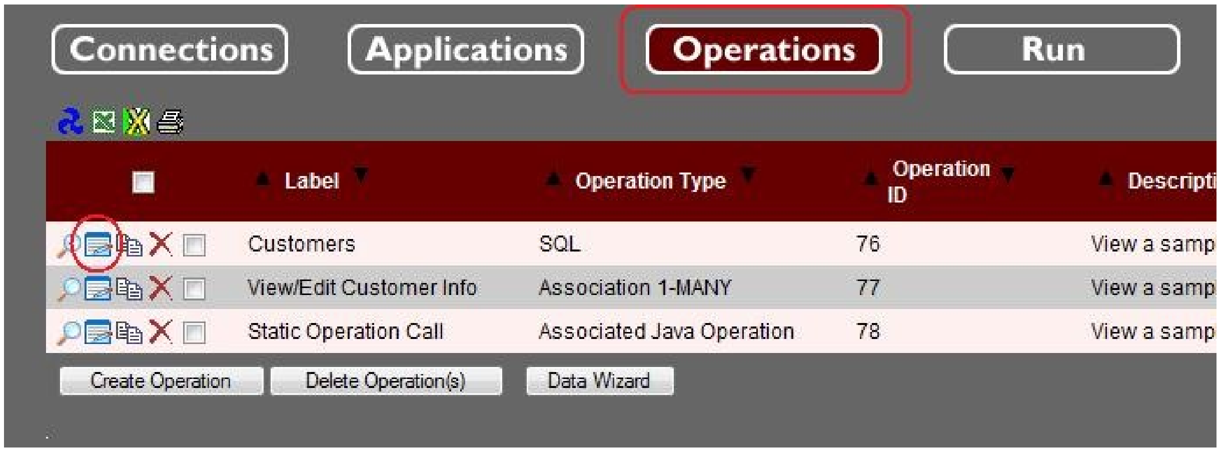
After your FDFFileDescriptor is all set up, invoke the `toFDF()` method passing in the descriptor.

```
RowCollection.toFDF(descriptor);
```

9.4 Example of a Row Collection Operation

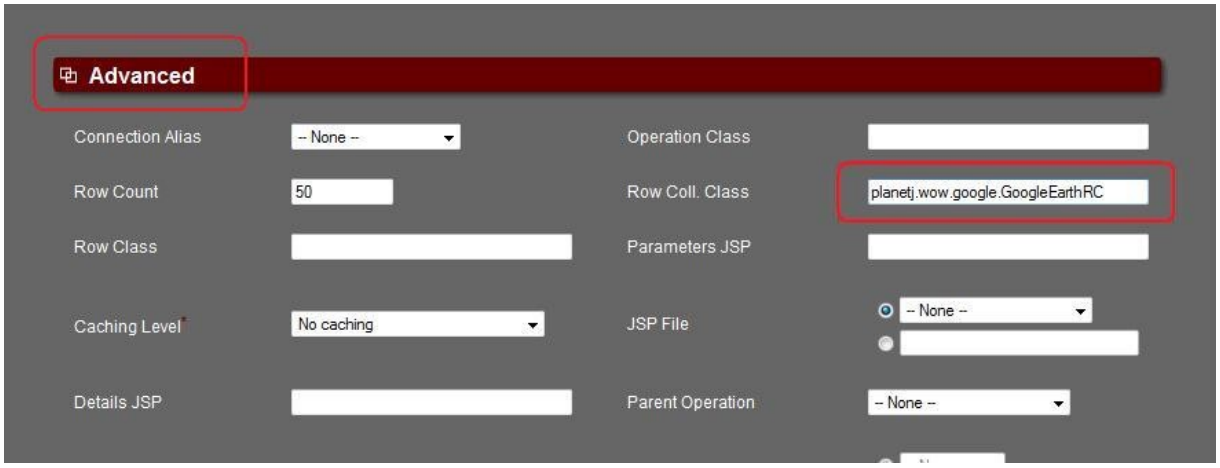
As you know, in WOW you need to create a custom java class for adding additional functionality to any operation. For this purpose, you always create a new java class and extend it with the required WOW class. So in a new custom class you can add your specific functionality. You can extend any class with the RowCollection class. As an example, in the following code we are creating a new class 'GoogleEarthRC' extending the 'RowCollection'.

```
public class GoogleEarthRC extends RowCollection
   //class body
}
```



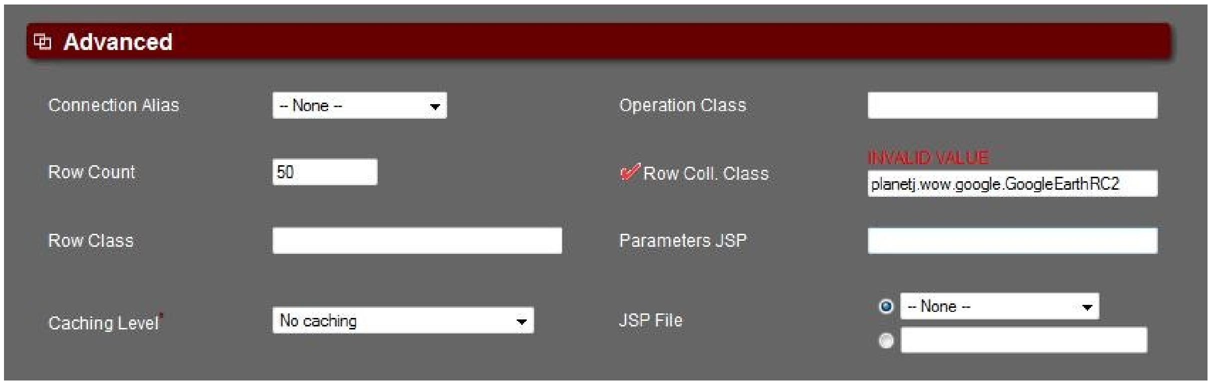
	Label	Operation Type	Operation ID	Descripti
	Customers	SQL	76	View a samp
	View/Edit Customer Info	Association 1-MANY	77	View a samp
	Static Operation Call	Associated Java Operation	78	View a samp

You can easily add a class within any operation by simply clicking the edit button. Then, in the Advanced Section under 'Row Coll. Class,' you will need to give the complete path of class. There is no need to type the .class extension, WOW already knows about that.



Connection Alias	-- None --	Operation Class	
Row Count	50	Row Coll. Class	planetj.wow.google.GoogleEarthRC
Row Class		Parameters JSP	
Caching Level*	No caching	JSP File	-- None --
Details JSP		Parent Operation	-- None --

In the case of a wrong package or class name, WOW will display a Message and indicat the Error and display the Roll Coll. Class Field like this.



Connection Alias	-- None --	Operation Class	
Row Count	50	Row Coll. Class	INVALID VALUE planetj.wow.google.GoogleEarthRC2
Row Class		Parameters JSP	
Caching Level*	No caching	JSP File	-- None --

To add the button which will work as a Row Collection action on the screen, you have to follow these steps:

On the edit screen of the operation go to the Display Section

In the “Properties” Text Area you can add functionality for different purposes

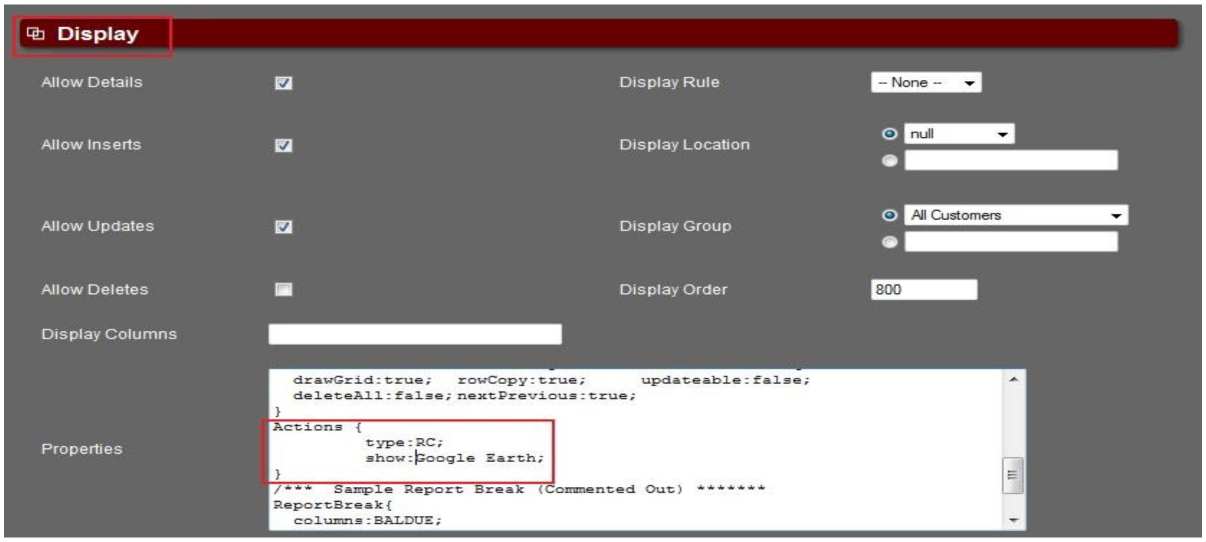
Here you’ll need to add “Actions{ }” tag behind the “Table Display{ }” tag

Add the following lines in Properties text area

```
Actions {
    type:RC;
    show:Google Earth;
}
```

type:RC; shows that you are adding an action for Row Collection.

show:Google Earth; 'Google Earth' will be the label of the button which you want to display as the Row Collection for table. 'Show' is a reserve word for WOW and you can enter any text which you want to display on the screen as a label of a button.



Allow Details	✓	Display Rule	-- None --
Allow Inserts	✓	Display Location	● null
Allow Updates	✓	Display Group	● All Customers
Allow Deletes		Display Order	800
Display Columns			

Properties

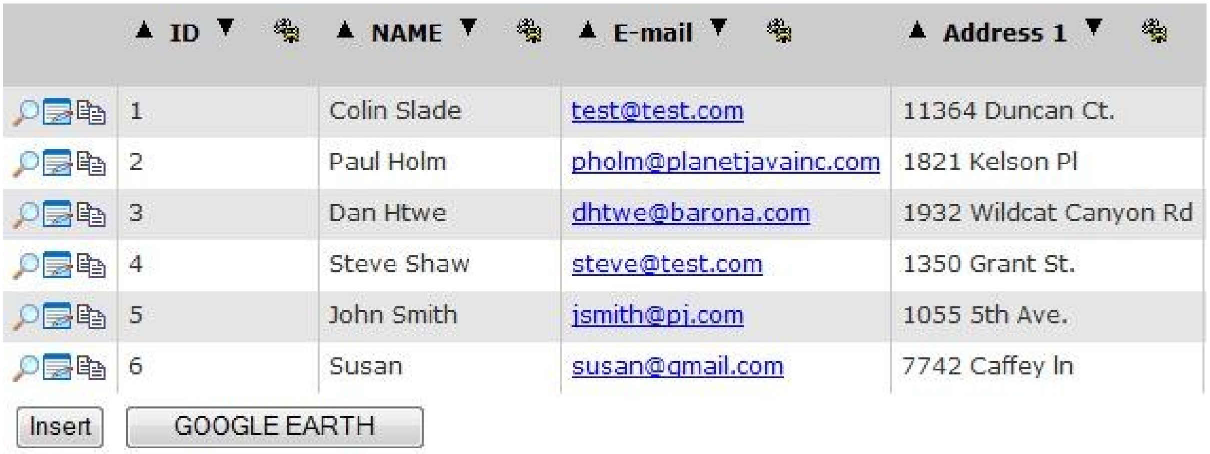
```
  drawGrid:true;   rowCopy:true;      updateable:false;
  deleteAll:false; nextPrevious:true;
}
Actions {
            type:RC;
            show:Google Earth;
}
/***  Sample Report Break (Commented Out) *******
ReportBreak{
  columns:BALDUE;
```

When you update the operation and run the application now your table will look like following screen shot.

The new button with the label which you entered in Properties,

```
show:Google Earth;
```

will display on the screen. With this button you can add your custom logic in the custom class which could be applied for all rows of the table.



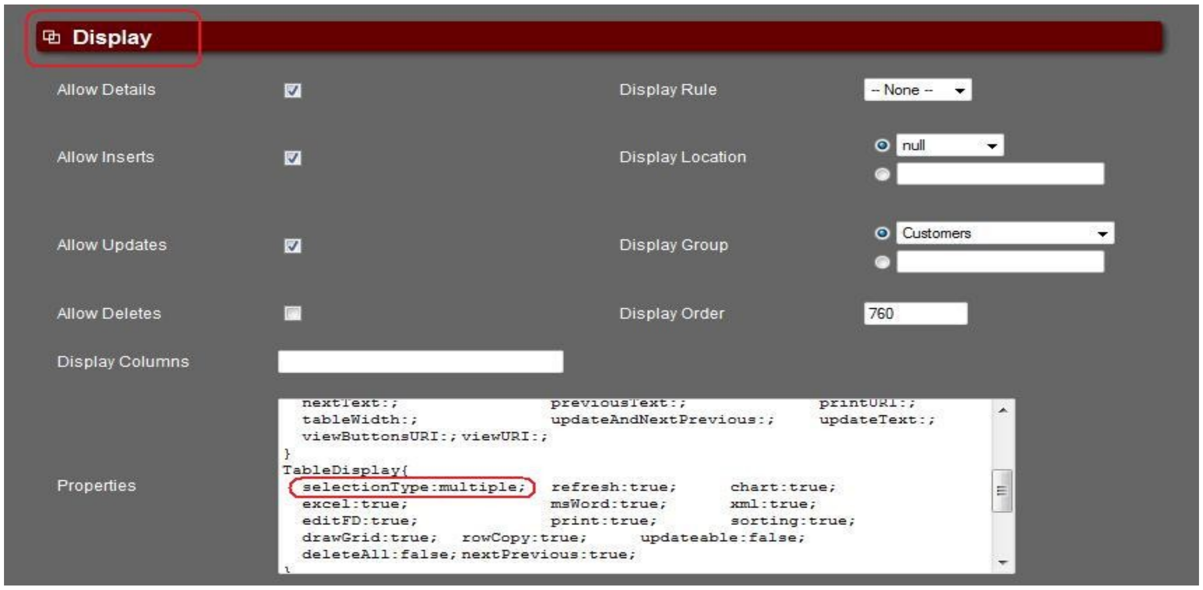
	ID	NAME	E-mail	Address 1
	1	Colin Slade	test@test.com	11364 Duncan Ct.
	2	Paul Holm	pholm@planetjavainc.com	1821 Kelson Pl
	3	Dan Htwe	dhtwe@barona.com	1932 Wildcat Canyon Rd
	4	Steve Shaw	steve@test.com	1350 Grant St.
	5	John Smith	jsmith@pj.com	1055 5th Ave.
	6	Susan	susan@gmail.com	7742 Caffey ln

Now in your newly created custom class you can add your custom logic. To extend the RowCollection class you'll have to override the 'handleAction( )' method.

This method takes a String and ExecutingContext as parameters and returns an Object.

```
public Object handleAction(String action, ExecutingContext ec) throws CMException {

    if ("GOOGLE EARTH".equalsIgnoreCase(action)) {
        RowCollection completeRC = super.getCompleteRowCollection();
        Iterator itr = selectedRC.iterator();
    }
    return super.handleAction(action, ec);
}
```

By editing the operation; in the Display section in Properties text area within the TableDisplay { } section you can set "selectionType:multiple".

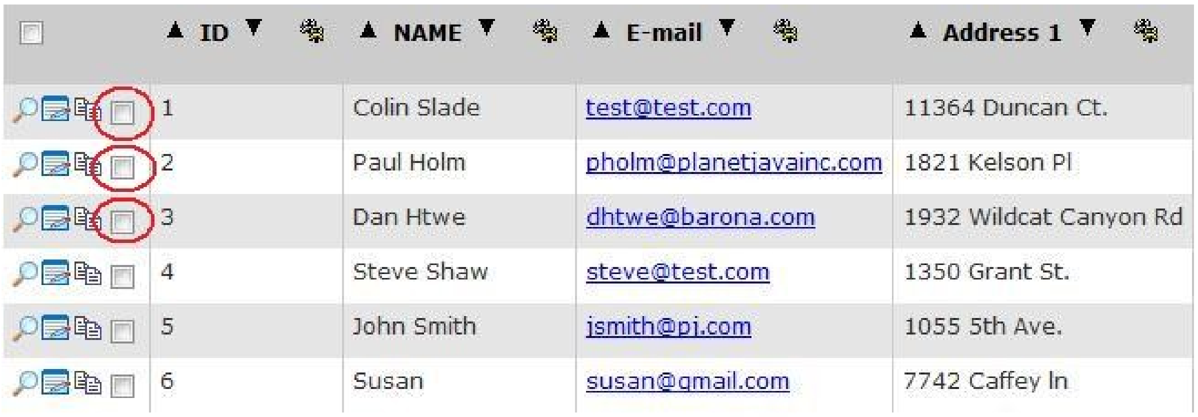


Allow Details	✓	Display Rule	-- None --
Allow Inserts	✓	Display Location	● null
Allow Updates	✓	Display Group	● Customers
Allow Deletes	☐	Display Order	760
Display Columns			

Properties

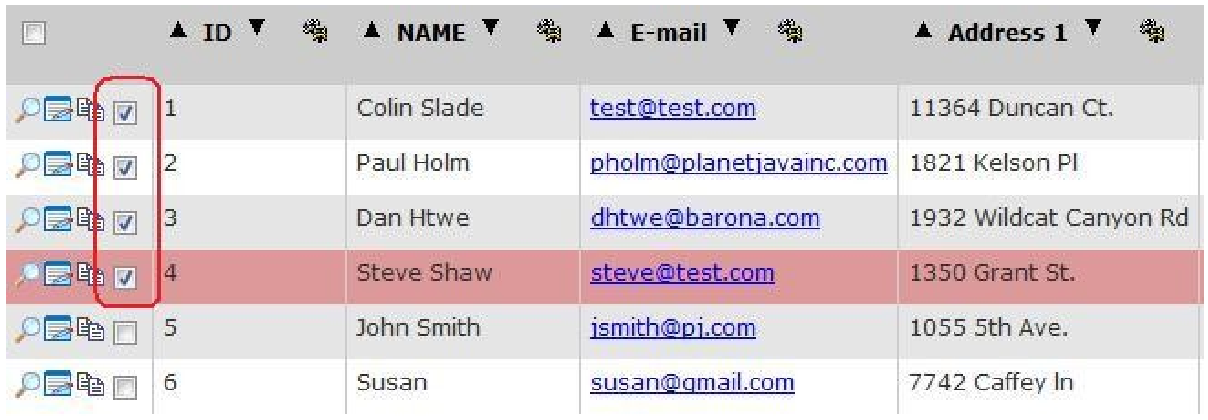
```
  nextText:;                previousText:;              printURL:;
  tableWidth:;              updateAndNextPrevious:;     updateText:;
  viewButtonsURI:; viewURI:;
}
TableDisplay{
  selectionType:multiple;   refresh:true;     chart:true;
  excel:true;               msWord:true;      xml:true;
  editFD:true;              print:true;       sorting:true;
  drawGrid:true;   rowCopy:true;     updateable:false;
  deleteAll:false; nextPrevious:true;
```

By setting the "selectionType:multiple" a check box will be displayed with each row and your output table will look like following screen shot



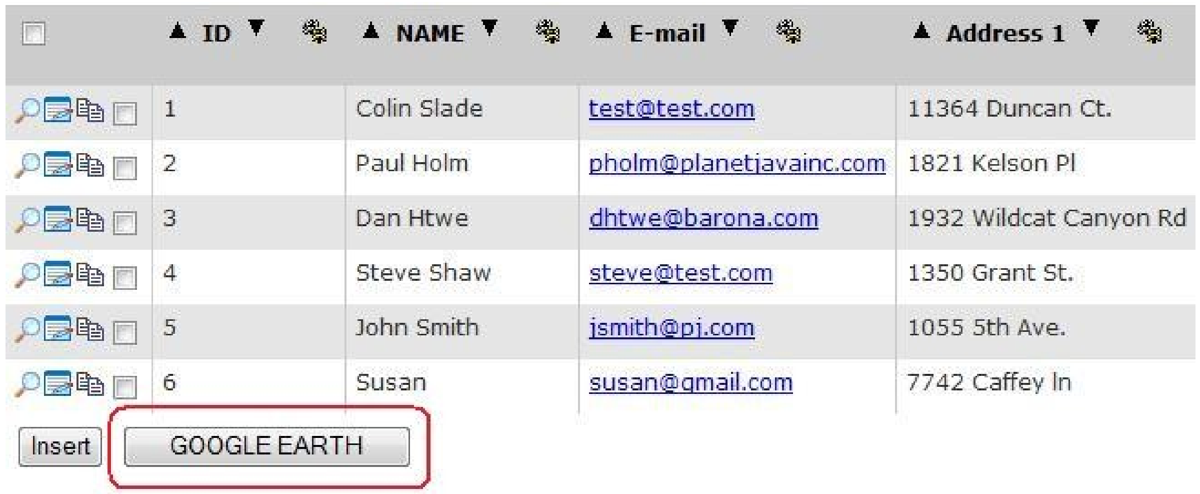
☐	▲ ID ▼	▲ NAME ▼	▲ E-mail ▼	▲ Address 1 ▼
☐	1	Colin Slade	test@test.com	11364 Duncan Ct.
☐	2	Paul Holm	pholm@planetjavainc.com	1821 Kelson Pl
☐	3	Dan Htwe	dhtwe@barona.com	1932 Wildcat Canyon Rd
☐	4	Steve Shaw	steve@test.com	1350 Grant St.
☐	5	John Smith	jsmith@pj.com	1055 5th Ave.
☐	6	Susan	susan@gmail.com	7742 Caffey ln

You can select multiple rows and click the Row Collection button. In the handleAction( ) method you can get the selected rows.



	▲ ID ▼	▲ NAME ▼	▲ E-mail ▼	▲ Address 1 ▼
✓	1	Colin Slade	test@test.com	11364 Duncan Ct.
✓	2	Paul Holm	pholm@planetjavainc.com	1821 Kelson Pl
✓	3	Dan Htwe	dhtwe@barona.com	1932 Wildcat Canyon Rd
✓	4	Steve Shaw	steve@test.com	1350 Grant St.
	5	John Smith	jsmith@pj.com	1055 5th Ave.
	6	Susan	susan@gmail.com	7742 Caffey ln

By clicking on the Row Collection button in “handleAction( )” method you can get selected rows with a “RowCollection” class method.

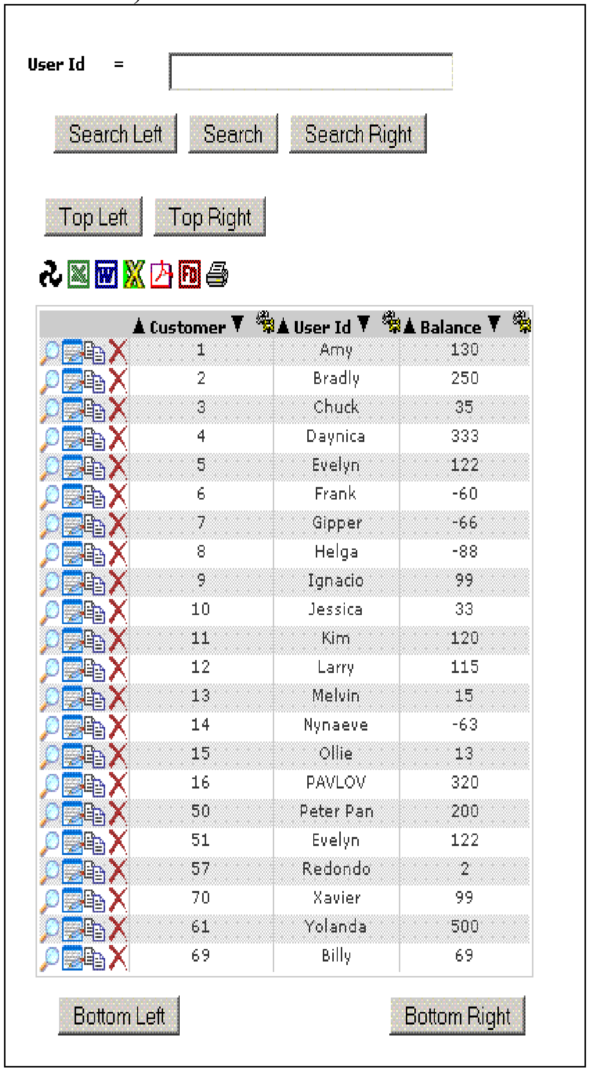


	▲ ID ▼	▲ NAME ▼	▲ E-mail ▼	▲ Address 1 ▼
	1	Colin Slade	test@test.com	11364 Duncan Ct.
	2	Paul Holm	pholm@planetjavainc.com	1821 Kelson Pl
	3	Dan Htwe	dhtwe@barona.com	1932 Wildcat Canyon Rd
	4	Steve Shaw	steve@test.com	1350 Grant St.
	5	John Smith	jsmith@pj.com	1055 5th Ave.
	6	Susan	susan@gmail.com	7742 Caffey ln

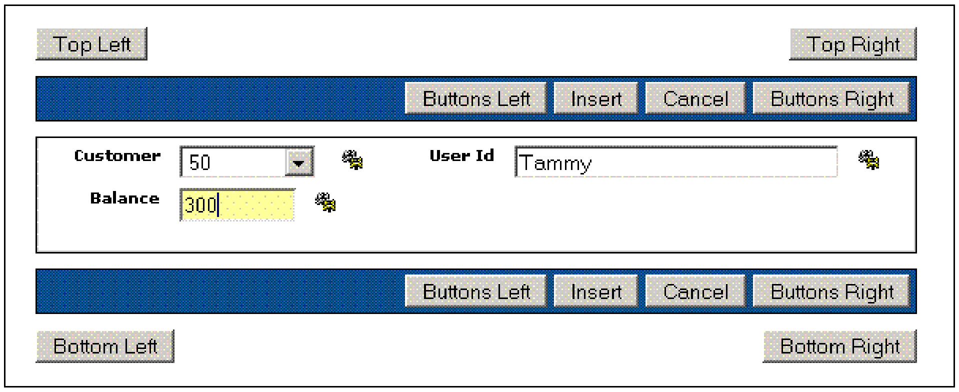
```
public Object handleAction(String action, ExecutingContext ec) throws CMException {

    if ("GOOGLE EARTH".equalsIgnoreCase(action)) {
        RowCollection selectedRC = super.getSelectedRowCollection();
    }
    return super.handleAction(action, ec);
}
```

9.5 Operation Actions

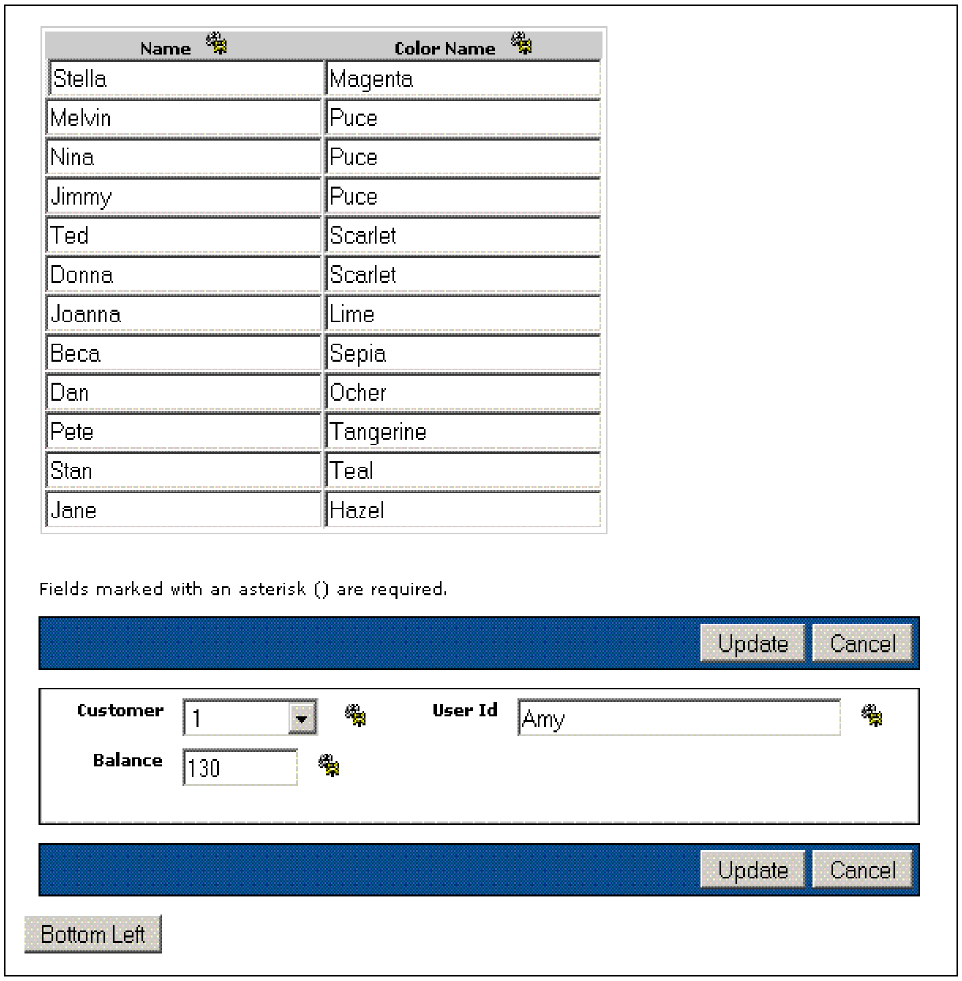
Similar in appearance to RowCollection actions, Operation actions invoke the Operation.handleAction() method (instead of RowCollection.handleAction()). Unlike RowCollection actions which can only appear when a RowCollection is being displayed to the user, Operation actions can be shown in many more places. Below are some screen shots showing Operation action buttons (the location name for each of the buttons is shown as text in the button):



	Customer	User Id	Balance
	1	Amy	130
	2	Bradly	250
	3	Chuck	35
	4	Daynica	333
	5	Evelyn	122
	6	Frank	-60
	7	Gipper	-66
	8	Helga	-88
	9	Ignacio	99
	10	Jessica	33
	11	Kim	120
	12	Larry	115
	13	Melvin	15
	14	Nynaeve	-63
	15	Ollie	13
	16	PAVLOV	320
	50	Peter Pan	200
	51	Evelyn	122
	57	Redondo	2
	70	Xavier	99
	61	Yolanda	500
	69	Billy	69



Customer	50
User Id	Tammy
Balance	300



Name	Color Name
Stella	Magenta
Melvin	Puce
Nina	Puce
Jimmy	Puce
Ted	Scarlet
Donna	Scarlet
Joanna	Lime
Beca	Sepia
Dan	Ocher
Pete	Tangerine
Stan	Teal
Jane	Hazel

Fields marked with an asterisk () are required.

Customer	1
User Id	Amy
Balance	130

The three screen shots above depict results, details, and execution group screens with action buttons.

Like a RowCollection action, a single Operation action can be shown in multiple locations. The following operation property groups create a "My Action" button that appears in 2 places on both the results and details screen (upper and lower left for the results screen, lower left and to the left of the buttons for the details screen):

```
Actions {
   type: op;
   show: action_name;
}

ActionDescriptor {
   name: action_name;
   actTyp: op;
   loc: bottom left, top left;
   detailsloc: bottom left, buttons left;
   label: My Action;
}
```

Both of the property groups above are necessary to create an action. The table below lists all the possible properties and values for the ActionDescriptor property group.

Property	Value	Description
actTyp	COLUMN | ROW | RC | OP	Designates the type of action: column, row, row collection, or operation.
ContextMenu	TRUE | FALSE	Whether or not the action should be shown in the context menu, in addition to its other locations. Defaults to true.
desc	*text*	Action description.
detailsLoc	BOTTOM LEFT | BOTTOM RIGHT | TOP LEFT | TOP RIGHT | SEARCH LEFT | SEARCH RIGHT	Specify the location(s) where the action button will be generated in the details view. If multiple locations are used, separate each with a comma.
detailsMode	DELETE | VIEW | EDIT | INSERT | COPY	Designates in which modes to generate the actions when showing the details screen. If multiple modes are used, separate each with a comma.
dspType	BUTTON | LINK | CHECKBOX	Specifies *one* form in which to display the action.
dspOrd	*integer*	The display order for this action. Actions with smaller display orders are displayed before actions with larger display orders
end group	TRUE | FALSE	Whether or not this action should end the current navigation group.
group	*text*	The name of the group to which the action belongs (optional). Separators are placed between different groups of actions in the action context menu.
imgsrc	*file path* | *URL*	Path to the image to use for the generated action's background.
label	*text*	The label is the text that appears on the action button or link.

loc	BOTTOM LEFT | BOTTOM RIGHT | TOP LEFT | TOP RIGHT | HEADER | INLINE | TOOLBAR | NONE | UNKNOWN | SEARCH LEFT | SEARCH RIGHT | RC	Specify the location(s) where the action button will be generated in the results view. If multiple locations are used, separate each with a comma.
name	*text*	Action name.
noSelectionError	*text*	The error message to display when no row is selected.
target	*text*	Target window to load action in.
start group	TRUE | FALSE	Whether or not this action should start a new navigation group.
window properties	property=value, property=value, etc..	Properties for the new window (only applicable if a target is specified). For example, toolbar=no or resizable =yes. See the 'specs' parameter at http://www.w3schools.com/htmldom/met_win_open.asp for a list of all possible window properties. If multiple properties are used, separate each with a comma.

10. Rows

10.1. Row Subclasses

In most cases you will want to create a subclass of Row for each database table in your application. This allows you to create specific get and set methods for each of the values in the table, and enables you to add business logic to a Row. When retrieving a RowCollection with the DataEngine, you can specify the type of Row subclass the RowCollection should contain by setting the RowClass property of the SQLContext that gets passed to the DataEngine:

```
mySQLContext.setRowClass (StudentRow.class);
RowCollection rc = DataEngine.getRows (mySQLContext);
```

A more flexible approach to choosing a subclass of Row is to override the createRow() method of your Row subclass:

```
public Row createRow (Row pDefaultRow) throws DataEngineException;
```

After the DataEngine reads the information from the database Row into a Row object, it invokes the createRow() method on that Row (passing in that same Row as an argument). Whatever Row is returned from the createRow method is the one which actually gets added to the RowCollection. By default, the Row.createRow() method just returns itself, but it can also examine the data contained in the default Row, and instantiate the correct subclass of Row based on that data. The Row.transferDataTo() method can then be used to transfer the data from the default Row into the new instance of Row and return. In this way, the DataEngine can retrieve a RowCollection that contains many different subclasses of the Row class.

You can also use an external object (which implements the IRowCreator interface) to create the appropriate subclass of Row. When an SQLContext has an IRowCreator (specified with the setRowCreator() method) and that SQLContext is passed to the DataEngine.getRows() method, then the DataEngine will use the createRow() method of that IRowCreator instead of the createRow() method of the Row to get the appropriate instance of a Row.

10.2. Creating a Row

New Rows can be created directly by the user or through a RowCollection. It is recommended that if you have a RowCollection and want to add a Row to it, you should use the RowCollections's new row method (this will do everything for you):

```
rowCollection.newRow()  // return a new Row of enclosed Row Class type
```

Otherwise, you can create a new Row yourself by doing one of the following:

```
(1) Row row = new Row()
(2) Row row = Row.create(Table)
(3) Row row = Row.create(Table, Class)
```

Note: If you want to create a subclass of Row, number (3) should be used. Also, if you use method number (1), keep in mind that in order for the Row to function properly (inserting, updatinge, etc…) you must set its table [row.setTable(Table)].

Once a row has been created, you can use the populate(ResultSet) method to fill a Row based on the current row of a ResultSet:

```
myRow.populate (myResultSet);
```

The populate method creates and adds fields to a row as necessary, filling in the values of the fields based on the data in the result set. Alternatively, you could create the field objects yourself and add them to the row via the add() method:

```
Field field = Field.create ("myField", java.sql.Types.CHAR)
myRow.add (field);
```

For more information about using/creating fields see (Creating Fields).

10.3. Row Operations

10.3.1. Inserting a Row

After a Row has been created or updated, the user may wish to insert a new record into the database. There are two methods for inserting a Row:

```
row.insert()             // insert row into database
row.insert(Connection conn)   // insert row into database
```

When called, the insert methods build up an SQL statement of the Row's Fields and their values.

Note: If a row has been updated and you don't want a new record, then row.update() should be used.

10.3.2. Updating a Row

When the update method of the Row is called, an SQL update statement is built using the Row's Library, Table, updated values, and keys. After the update takes place in the database, the Row's current values for its fields (if existing) are set as the new original values. Note: when any of the Row's field values are changed, the Row's current values are changed, while the original values stay the same. Then, when a successful update has been made to the database, the Row's original values are set to the current values.

```
row.update()             // update a row in database
row.update(connection)     // update a row in database
```

The Library and Table are retrieved from the Row. The SQLGenerator generates the set clause by using the row's current values (the values needing updating). The SQLGenerator also generates the where clause by using the row's key fields.

10.3.3. Deleting a Row

Deleting a Row works similar to updating a Row. When the delete method is called on a Row, an SQL statement is built using the Row's Library, Table, and key fields in a similar fashion as updating.

```
row.delete()             // delete a row in database
row.delete(connection)     // delete a row in database
```

When a row is deleted, it must also be removed from the object it is contained/stored in (e.g. if a row is contained in a RowCollection, it must also be removed from the RowCollection).

10.3.4. Cloning a Row

Cloning might be used in a situation where two different tables have the same Fields, but different table names. For example, an item for an order might be in a OrderPending table until it is processed and then moved to the Order table. To clone a Row all you need to do is call:

```
Row clone = row.cloneRow()    // returns a clone of the row
```

Both rows point to the same table after the clone. If the row clone is to contain a different table other than the previous, its set table method should be called:

```
clone.setTable(newTable)      // sets the clone's table
```

10.4. Retrieving a Row's Fields (FieldCollection)

Each Row has a FieldCollection object which contains all the Fields in a given Row. This allows for retrieval of a Field in two ways. (1) Name, and (2) Index.

```
getField(String fieldname) // return Field with given name
getField(int index)      // return Field at specified index

(e.g. row.getField("LASTNAME") - return LASTNAME Field)
(e.g. row.getField(1) - return the first Field in Row)
```

Note: Fields are indexed in the Row starting at 1 in correlation with SQL indexing.

10.5. Generating Files From a Row

10.5.1. CSV File From a Row

See Also: (Generating a CSV file from a RowCollection)

If you wish to write a Row out to a file for analysis, it can be a pain. How do you do it? The DataEngine conveniently accomplishes this by writing the data out to a CSV (Comma Separated Values) file. A CSV file can be read with Excel or some other application to do further analysis (graphs, charts, etc…). The most common way to write an HTML table to a CSV file is by clicking on the Excel icon {}. Currently this is a magic function (See Magic Request CSV/Excel).

In order to write a Row to a CSV file, create a CSVFileDescriptor that defines attributes about a CSV file.

```
CSVFileDescriptor descriptor = CSVHelper.singleton().newCSVFileDescriptor();
descriptor.setDelimiter('¤'); // Delimiter default to a comma {,}
descriptor.setFileName("C:\\Temp\\Test.csv");

// If you want to append to an existing CSV file
descriptor.setAppendToFile(true); // Defauts to false

// If you want all the output on a single line
descriptor.setOutputAsSingleLine(true); // Defaults to false

// If you want each new row to begin with a delimiter
descriptor.setLeadingDelimiter('×'); // Default to an empty space ""
```

After your CSVFileDescriptor is all set up, invoke the `toCSV()` method passing in a boolean true if you want to include the column heading as the first row in the CSV file and the CSVFileDescriptor.

```
Row.toCSV(includeColumnHeadings, csvFileDescriptor);
```

10.5.2. Microsoft Word File From a Row

To produce a Microsoft Word file (.doc) from a Row, click on the Word icon {W}. Currently this is a magic function (See Magic Request Microsoft Word). (See DOCHelper) (See CSVHelper)

In order to generate the Microsoft Word file (.doc) you must call the toDOC method of Row. A CSV file is generated under the covers and passed into word. For more information on generating a CSV file from a RowCollection (See Generating a CSV file from a Row).

```
Row.toDOC(includeColumnHeadings, descriptor);
```

10.5.3. FDF file from a Row

This is normally used along with a PDF template to plug the FDF data into. (See FDFHelper)

In order to write a Row to an FDF file and display that data in a template, create an FDFFileDescriptor that defines attributes about an FDF file.

```
  FDFFileDescriptor descriptor = FDFHelper.singleton().newFDFFileDescriptor();

  // You must set the template so the FDF file knows which PDF to open
  fileDescriptor.setPDFFileName("http://" + request.getRemoteHost() + ":" +
request.getServerPort() + "/Template.pdf");

  // to write the FDF data out to a file, set a file name
  // property in the descriptor.  If this file name is set, then the
  // Rows FDF data will be output to the file.  If no file name is set,
  // then it will be output to the browser's output stream as shown below.
  descriptor.setFileName("C:\\Temp\\Test.fdf");
```

After your FDFFileDescriptor is all set up, invoke the `toFDF()` method passing in the descriptor.

```
Row.toFDF(descriptor);
```

10.5.4. PDF file from a Row

To view Row data in PDF format, click on the PDF icon {}. Currently this is a magic function (See Magic Request PDF). (See PDFHelper)

In order to write a Row to a PDF file, create a PDFFileDescriptor that defines attributes about a PDF file.

```
PDFFileDescriptor descriptor = new PDFFileDescriptor();

// to write the PDF data out to a file, set a file name
// property in the descriptor.  If this file name is set, then the
// Rows PDF data will be output to the file.  If no file name is set,
// then it will be output to the browser's output stream as shown below.
descriptor.setFileName("C:\\Temp\\Test.pdf");

// to display this PDF in a browser using the Adobe Acrobat browser
// plug-in, then set the OutputStream of the descriptor like this.
descriptor.setOutputStream(response.getOutputStream());
```

After your PDFFileDescriptor is all set up, invoke the `toPDF()` method passing in a boolean true if you want to include the column heading as the first row in the PDF file and the PDFFileDescriptor.

```
Row.toPDF(includeColumnHeadings, descriptor);
```

10.5.5. XML File From a Row

To view Rows data in XML format you can click on the XML icon {}. Currently this is a magic function (See Magic Request XML). (See XMLHelper)

In order to write a Row to a XML file, create an XMLFileDescriptor that defines attributes about a XML file.

```
XMLFileDescriptor descriptor = XMLHelper.singleton().newXMLFileDescriptor();

// to write the XML data out to a file, set a file name
// property in the descriptor.  If this file name is set, then the
// Rows XML data will be output to the file.  If no file name is set,
// then it will be output to the browser.
descriptor.setFileName("C:\\Temp\\Test.xml");
```

After your XMLFileDescriptor is all set up, invoke the `toXML()` method passing in the descriptor.

```
Row.toXML(descriptor);
```

10.6. Row Subclassing

Row subclassing is a very important part of WOW customization that can be done without an extensive amount of coding and can be implemented rather easily. Why do we need Row subclassing? We need row subclassing because every project, database and set of data is different. To extend the power of WOW we can override a RowCollection, Row, Field and hundreds of methods that allow WOW to handle almost any type of need and customization. For example you may have an option where you do not allow someone to set the salary field greater than $70,000. That is where a new row subclass (i.e. a smart row) would be very convenient.

10.6.1. Create Row SubClass

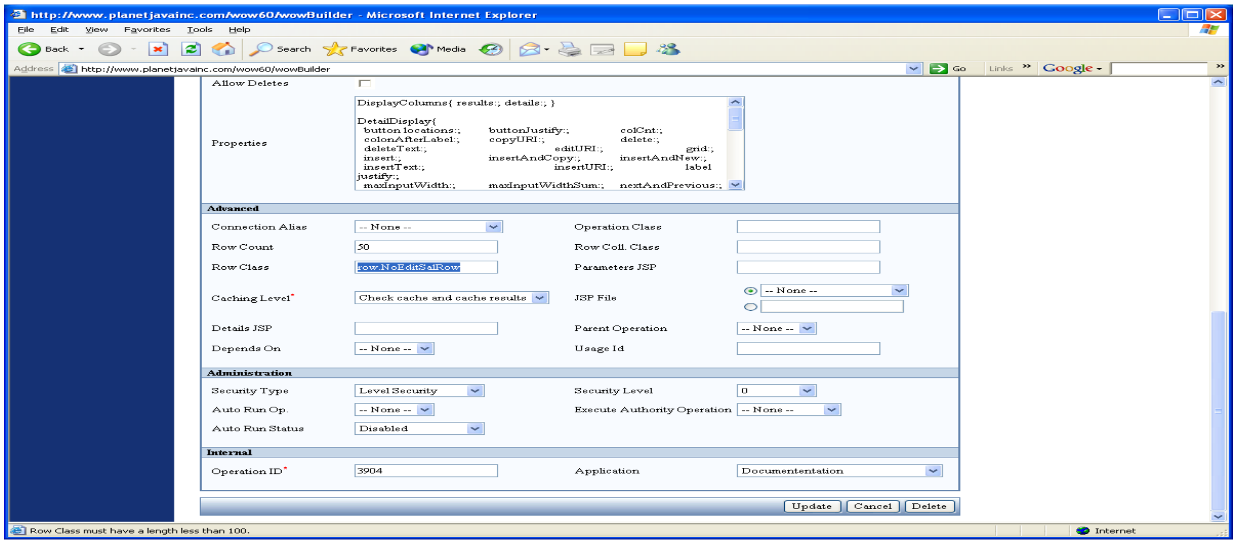
Create a file called noEditSalRow that extends and overrides the Row class like shown below. You can either use notepad or you can use some other tool such as IBM's WebSphere to edit and create the java file.

```
package planetj.samples.row

import planetj.database.*;
import planetj.dataengine.*;
import planetj.exception.*;

public class NoEditSalRow extends Row {

   public boolean validate(ExecutingContext context) throws CMException {

      if (context.getMode() == DataEngineManager.getIntMode(IDataEngine.MODE_EDIT)) {
         double salary = getValueAsDouble("Salary");
         if (salary > 70000) {
            throw new CMException("Salary exceeds cash. Salary cannot be above
               70,000.");
         }
      }
      return super.validate(context);
   }
}
```

After you have created the Java class, you'll need to compile it using either the command line with javac command or using WebSphere. This will pick up syntax errors which need to be corrected and then the following class file needs to be saved in the <Tomcat>/webapps/wow64/WEB-INF/classes directory. It needs to be saved in the same path as is set in the java file such as *package row,* so it needs to be saved in a folder called row inside of the classes folder. Now that you have created the class and saved it in the classes' folder in Tomcat Directories, we need to set WOW to pick up that new class.

10.6.2. Set WOW to Recognize New Java Class

Start the WOW Builder by bringing up the WOW application. Then bring up the operation that you are updating or overriding. In the operation's properties under the Advanced section, add the row subclass to the Row Class field so that it will override the current row. Make sure that you keep the path the same, starting in the classes folder, since NoEditSalRow is saved in the Row folder in the classes folder.


http://www.planetjavainc.com/wow60/wowBuilder - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://www.planetjavainc.com/wow60/wowBuilder
Go Links Google
Update Cancel Delete
Internet

Allow Deletes	☐
Properties	DisplayColumns{ results:; details:; } DetailDisplay{ button locations:; buttonJustify:; colCnt:; colonAfterLabel:; copyURI:; delete:; deleteText:; editURI:; grid:; insert:; insertAndCopy:; insertAndNew:; insertText:; insertURI:; label justify:; maxInputWidth:; maxInputWidthSum:; nextAndPrevious:;

Advanced

Connection Alias	-- None --	Operation Class	
Row Count	50	Row Coll. Class	
Row Class	row.NoEditSalRow	Parameters JSP	
Caching Level*	Check cache and cache results	JSP File	-- None --
Details JSP		Parent Operation	-- None --
Depends On	-- None --	Usage Id	

Administration

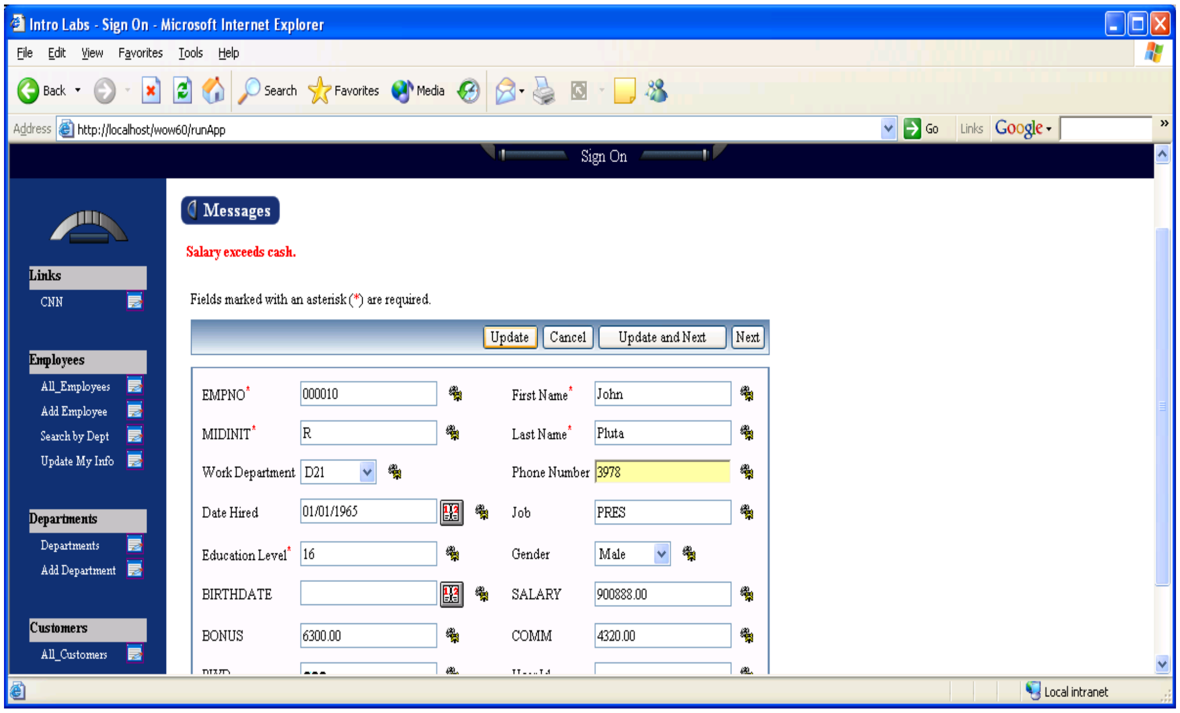
Security Type	Level Security	Security Level	0
Auto Run Op.	-- None --	Execute Authority Operation	-- None --
Auto Run Status	Disabled		

Internal

Operation ID*	3904	Application	Documententation

Row Class must have a length less than 100.

After you have input the path for the new class, click Update and then run the operation. Next, edit a row and change the salary to something over 70,000. It should throw an error message like the one shown below and not allow you to update.


Intro Labs - Sign On - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/wow60/runApp
Go Links Google
Sign On
Links
CNN
Employees
All_Employees
Add Employee
Search by Dept
Update My Info
Departments
Departments
Add Department
Customers
All_Customers
Messages
Update Cancel Update and Next Next
Local intranet

Salary exceeds cash.

Fields marked with an asterisk (*) are required.

EMPNO*	000010	First Name*	John
MIDINIT*	R	Last Name*	Pluta
Work Department	D21	Phone Number	3978
Date Hired	01/01/1965	Job	PRES
Education Level*	16	Gender	Male
BIRTHDATE		SALARY	900888.00
BONUS	6300.00	COMM	4320.00

10.6.3. Overriding Methods

The Method that is overridden in this case is validate (ExecutingContext), which checks to see whether or not data entered is correct. If not, it then throws an exception at that point, stopping the action (which in this case is an edit command). There are many different methods that you can override to change the behavior of RowCollection, Row and Field.

Field Methods

These are some of the methods of a Field which are frequently overridden in Field subclasses. Remember the general pattern is for WOW to invoke the method on a field, which by default invokes the equivalent method on its Row, which by default invokes the equivalent method on the Field Descriptor. So you have two chances to override, in the Field or the Row. Usually it is easier to have a single Row subclass rather than numerous field subclasses, but certain methods (like getValue() or getDisplayValue()) are only present in the Field class, and therefore can only be overridden by Field subclasses.

Association Operation - The operation which is used to retrieve the associated data for a field. This is null when a field does not have an association

```
Field.getAssociationOperation()
Row.getAssociationOperation(String)
FieldDescriptorRow.getAssociationOperation()
```

Displayable - Whether or not a field should be displayed to the user

```
Field.isDisplayable(ExecutingContext)
Row.isFieldDisplayable(Field,ExecutingContext)
FieldDescriptorRow.isDisplayable()
```

DisplayValue - The String value that is displayed on the screen to the user. This is usually dependent (but possibly different than) the actual value of the field, which may not be a String. This is the method to override when you want to specially format what is displayed to the user. Note that this method only determines the actual value of what is displayed, not its CSS style.

```
Field.getDisplayValue()
```

External Name - The field name displayed to the user

```
Field.getExternalName()
Row.getExternalName(String)
FieldDescriptorRow.getExternalName()
```

Possible Values Operation - The operation which is used to retrieve the possible values for the field. This is null when a field does not have possible values

```
Field.getPossibleValuesSQLOperation()
Row.getPossibleValuesSQLOperation(String)
FieldDescriptorRow.getPossibleValuesSQLOperation()
```

Read Only - Whether or not the field is read only

```
Field.isReadOnly(ExecutingContext)
Row.isFieldReadOnly(Field,ExecutingContext)
FieldDescriptorRow.isReadOnly()
```

Required - If a user must enter a value for the field when inserting or updating the row

```
Field.isRequired(ExecutingContext)
Row.isFieldRequired(Field,ExecutingContext)
FieldDescriptorRow.isRequired()
```

StatusChanged - Invoked when a field is given a new value by the user. By default all fields in the row are notified when a field in that row has its value changed, but only if that field has "Notify Status Change" set to "Yes" in its Field Descriptor

```
Field.statusChanged(Field,ExecutingContext)
Row.statusChanged(Field,ExecutingContext)
```

Style Class - This is the CSS style which is used to display the field's display value to the user.

```
Field.getStyleClass(String,String,ExecutingContext)
Row.getStyleClass(Field,String,String,ExecutingContext)
FieldDescriptorRow.getStyleClass()
```

Value - The current value contained in the field. This is different than the display value. For example, if a field's value is a date, its display value could be "03/10/2004" or "3-10-04" or "March 10, 2004" but in all cases the field's actual value is the same date.

```
Field.getValue()
Field.setValue(Object)
```

SQLOperation Methods

This section shows the typical flow of control through an SQLOperation as a user runs it. When overriding SQLOperations, it is important to remember that every single user accesses the exact same instance of an SQLOperation. So, generally, you should not create fields at the class level in your SQLOperation subclass, unless it is okay for all users to see the same value for those fields. Local variables within a method are OK.

1. User clicks on link to run the operation. The getNewContextInternal() method is invoked, which creates and returns an SQLContext object. This SQLContext contains the SQL for the operation, including the parameters & parameter values, and is used to generate the parameter prompts on the screen for the user to interact with. If you need to change the parameters displayed to the user, you should override the getNewContextInternal() method.

2. User enters in values for the parameters and clicks the Search button. WOW takes the values from the screen and uses them to populate the SQLContext (created in step 1). At this point, the preExecute(SQLContext) method is invoked. Normally this method does nothing, but you can override it to alter the SQLContext object, or even to create & return a different SQLContext.

3. Using the SQLContext returned by the preExecute(SQLContext) method in step 2, WOW runs the SQL query

4. The results of running the query, as well as the SQLContext that was used to run the query are passed to the postExecute(Object,SQLContext) method. This method normally does nothing, but you can override it to alter the operation results, or to return different operation results

5. The operation results (typically a RowCollection) returned by the postExecute(Object,SQLContext) method are displayed on the screen to the user.

RowCollection Methods

Methods which are commonly overridden in RowCollection subclasses include:

RowCollectionCreated() - Invoked after a RowCollection is created. The RowCollection will have its SQLContext set, but will not yet contain any rows

RowCollectionPopulated() - Invoked after a RowCollection has been fully populated with all of the rows read from the DB

Row Methods

Many of the most commonly overridden methods of a Row are listed in the Field section above, because they deal with a particular field within the row. Methods that deal with the entire Row which can be overridden include:

delete(Connection,ExecutingContext) - Invoked to delete a row. Situations where you might want to override include cases where you need to perform some checks and possibly disallow the deletion, or if you need to delete other rows in other tables whenever a particular row is deleted.

insert(SQLContext) - Invoked whenever a row is inserted into the DB. You can override this method to adjust the row's values prior to insertion, or after insertion.

update(Connection,ExecutingContext) - Invoked when a row in the DB is updated. Override this method to alter the default update behavior

Other Methods

Boolean **validate**(Executing Context) - Validates the field based on its configuration and value. Customize the accepted values of a field

Boolean **isRequired**(ExecutingContext) - Tests whether or not field is required. Subclasses may need this because change of default values when overriding a row.

Boolean **isAuthorizedForEdit**(ExectutingContext) - Tests if users can edit this field. Can specify what user can edit what fields or what fields are not editable by anyone.

Boolean **isFieldRequired**(Field, ExecutingContext) - Same as Field's isRequired but is called by Row.

Row **prepareForDetails**(ExecutingContext) - Gets the proper Row for viewing this Rows Details. Can get extra fields for the details by overriding. Can alter field values before row is displayed.

Boolean **isFieldReadOnly**(Field, ExecutingContext) - Tests if the Field is read-only. Override to change the behavior and rules of read-only.

Boolean **isDeletable**(ExecutingContext) - Checks to see if this Row may be deletable (default: True). Can make certain rows undeletable while deleting others.

Boolean **isValidateRequired**() - Tests if row must be validated before it is inserted. Can skip over validation under customized circumstances.

Row **prepareForResultsDisplay**(ExecutingContext) - Allows the current row to handle any necessary actions before being displayed in a results table.

10.6.4. Another Example of Overriding Row and Field Methods

The first example overrode the validate method to not allow anyone to change the salary field to a value over 70,000. Now we are going to override the isFieldReadOnly method so that a field such as the salary can only be edited by a user that is defined as a manager in the database . We first must create the class and Java file that overrides the Row. Shown below is the basic code to start any of the classes to override a Row.

```
package planetj.samples.row

import planetj.database.*;
import planetj.dataengine.*;
import planetj.exception.*;

public class ManagerEditRow extends Row {

   public boolean isFieldReadOnly(ExecutingContext context, Field pField) {
      Field jobField = getField("JOB");
      boolean read = false;
      if (pField.getName().equals("SALARY")) {
         if (!jobField.getValueAsString().equals("MANAGER")) {
            read = true;
         }
      }
      return read;
   }
}
```

After writing the code above, we would compile it into a class file and copy that file into the classes directory under Apache Tomcat and wow64 (as talked about in the last section). Then specify the row class in the operation as a class to override Row as shown above. Next run the application to see if it works. All of the methods used can be referenced and accessed from our online JavaDocs at http://www.planetjavainc.com.

10.6.5. Multiple Methods and Parameters

It is possible and probably encouraged to put more than one of the methods you are overriding in the same Row Subclass. For example, we could put the same NoEditSalRow method (which is validate) into the ManagerEditRow Java file with its code so that you have one class and two methods that will override the original Row. That way we can get two different behaviors from the same file. The user cannot change any salary above 70,000 and can only change the salary when the job is set to manager.

```
package planetj.samples.row

import planetj.database.*;
import planetj.dataengine.*;
import planetj.exception.*;

public class ManagerEditRow extends Row {

   public boolean isFieldReadOnly(ExecutingContext context, Field pField) {
      Field jobField = getField("JOB");
      boolean read = false;
      if (pField.getName().equals("SALARY")) {
         if (!jobField.getValueAsString().equals("MANAGER")) {
            read = true;
         }
      }
      return read;
   }

   public boolean validate(ExecutingContext context) throws CMException {

      if (context.getMode()== DataEngineManager.getIntMode(IDataEngine.MODE_EDIT))
      {
         double salary = getValueAsDouble("Salary");
         if (salary > 70000) {
            throw new CMException("Salary exceeds cash. Salary cannot be above
               70,000.");
         }
      }
      return super.validate(context);
   }
}
```

10.6.6. Example of Overriding the Update Method in a Row Subclass

Overriden Update Method

This first method shows the update method overridden to set a field value (POLICYNUM) not specified on the edit screen. In this particular instance, we want to set the insurance policy number to the next available number prior to the update (we want to control this value internally).

```
/**
*   Updates the database with the values in this Row.
*/
public synchronized int update(Connection connection, ExecutingContext ec)
   throws CMException, DistributedException {

   this.setFieldValueAsString("POLICYNUM", getNewPolicyNumber());
   return super.update(connection, ec);      // Now update the row
}
```

Method Containing Internal Query

This second method queries the policy master file to determine the next available policy number.

```
/**
* Returns the next available policy number.  Only allow 1 user to extract
* policy number at a time.
*/

private synchronized String getNewPolicyNumber() throws CMException {

   String policyNum = "5000000"; // Set default value in case this
   // is the first one.
   Row row = null;
   SQLContext context = new SQLContext("MYCONNECTION");
   context.setRowClass(Row.class);
   context.setSQL("SELECT MAX(POLICYNUM) AS POLICYNUM FROM "
      + "MYLIB.MASTERFILE");

   row = DataEngine.getRow(context);  // Read the max row

   if (row != null) {
      int policyNumInt = row.getValueAsInt("POLICYNUM");
      policyNumInt++;      // Set to next available number
      policyNum = Integer.toString(policyNumInt);
   }

   return policyNum;
}
```

10.6.7. Example of Overriding the Insert Method in a Row Subclass

Overriden Insert Method

This first method shows the insert method overridden to call an external program after the insert is complete. An account number specified on the insert screen is passed to the program call method.

```
/**
* Here we need to call an external program after the insert is complete.
*
*/

public int insert(SQLContext context) throws CMException {
   int returnValue = super.insert(context);
   String acctNum = getValueAsString("ACCTNUM");
   callExternalProgram(acctNum);

   return returnValue;
}
```

Method Containing External Program Call

This second method calls an external program using an SQL procedure. A procedure provides the simplest method for calling native external programs from Java. In this example, we are calling program mylib/handleacct with one input parameter and one output parameter.
NOTE: You'll need to create an external SQL procedure on your local system using a CREATE PROCEDURE statement similar to the one documented in the method comments below:

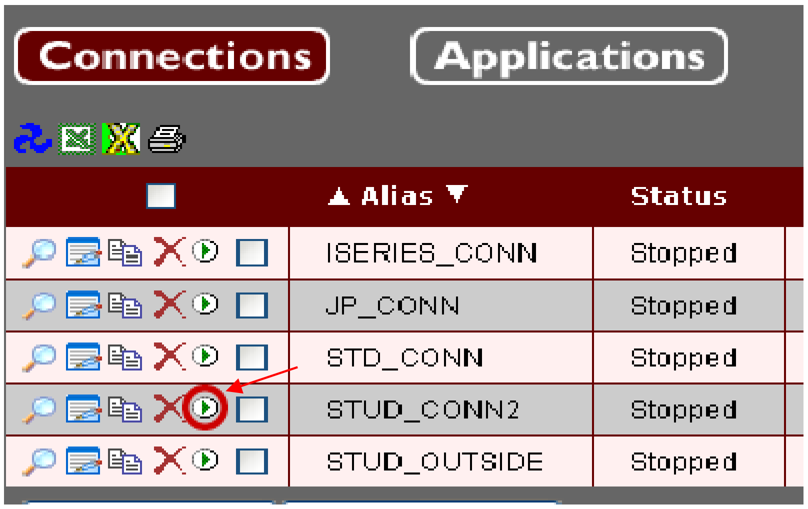
```
/**
*  The procedure below is used to call the external program with 2 parameters:
*   CREATE PROCEDURE MYLIB.HANDLEACCT (
*      IN ACCTNO CHAR(7) ,
*      OUT RC CHAR(2) )
*      LANGUAGE RPG
*      SPECIFIC MYLIB.HANDLEACCT
*      MODIFIES SQL DATA
*      EXTERNAL NAME 'MYLIB/HANDLEACCT'
*      PARAMETER STYLE GENERAL ;
*/

private void callExternalProgram(String acctNo) throws CMException {

    String errorMsg = "";
    Connection c = planetj.database.DatabaseManager.getConnection("MYCONNECTION");
    try {
       CallableStatement cs = c.prepareCall("CALL MYLIB.HANDLEACCT (?,?)");
       // Input Parm
       cs.setString(1, acctNo);   // Parm 1 = Acct #
       // Output Parm
       cs.registerOutParameter(2,Types.CHAR);  // Parm 2 = Return Code
       // Call the procedure (program)
       cs.execute();
       // Process return code
       String rCode = cs.getString(2);
       if (!rCode.equals("00")) {
           // Problem with program call?
           errorMsg = "Call to program failed with rc = " + rCode + ".";
       }
    } catch (Exception e) {
        throw new CMException(e);
    } finally {
       try {
        cs.close();  // close call statement
        } catch (SQLException sqle) {
        cat.error("Failed to close callable statement.", sqle);
        }
       // Return the connection
       if (c != null) {
        planetj.database.DatabaseManager.freeConnection(c);
        }
       if (!errorMessage.equals("")) { // Program encountered problems
        throw new CMException(errorMessage);
        }
    }
}
```

10.7. Row Actions

Row actions are actions that are performed on a row when the action button is pressed. A few common scenarios include restarting a connection, sending email, updating the row or an 'Add to Cart' button. Row actions are similar in appearance to RowCollection and Operation actions. They are also similar in that row actions invoke the

```
Row.handleAction() method
```

Row actions appear to the left of the Row when the when a Row is being displayed to the user. Below is a screen shot showing a "Start a Connection" Row action.



	Alias	Status
	ISERIES_CONN	Stopped
	JP_CONN	Stopped
	STD_CONN	Stopped
	STUD_CONN2	Stopped
	STUD_OUTSIDE	Stopped

```
      /**
      * Basic code for Start Connection Row Action shown above.
      */
public Object handleAction(String action, ExecutingContext ec) throws CMException
{

         if (ACTION_START.equalsIgnoreCase(action)) {
      DatabaseManager.performConnectionPoolAction(getAlias(),
       DatabaseManager.ACTION_START_CONNECTION_POOL);
      }
}
```

To create a basic row action you must first create a Row subclass and override the Row.getRowHandledActionNames() method. This method is called when WOW is generating the Row and checking whether or not any actions exist. For a Row Action to be displayed, you need to add it to the Action Array List which is created within getRowHandledActionNames(). Below, three actions are added for the connection row:

```
      /**
      * Get a list of action's this connection row supports.
```

```
*/
public List getRowHandledActionNames() {

  // see if super class has any actions to include
  List actions = super.getRowHandledActionNames();
  if (actions != null) {
        //Create new actions array list
        actions = new ArrayList(actions);
  }
  else {
        actions = new ArrayList();
  }

  // add connection row actions
  actions.add(ACTION_START);
  actions.add(ACTION_STOP);
  actions.add(ACTION_RESTART);

  return actions;
}
```

Now that we have added the action names to the actions list, these actions will show up when this Row is generated, but the actions when clicked will do nothing. To add Action functionality, the method Row.handleAction() is overridden. The method handleAction() takes in a String and current ExecutingContext. The String is the name of the action that was clicked on, if you have multiple action names with different actions then we need to test for the correct action within handleAction(). After testing, if it's the correct action, then all action code should be included within the handleAction() method.

```
/**
*Basic connection row handle action method which handles given action.
*/
public Object handleAction(String action, ExecutingContext ec) throws
CMException {


      if (ACTION_START.equalsIgnoreCase(action)) {
       DatabaseManager.performConnectionPoolAction(getAlias(),
       DatabaseManager.ACTION_START_CONNECTION_POOL);

  }
      else if (ACTION_STOP.equalsIgnoreCase(action)) {
       DatabaseManager.performConnectionPoolAction(getAlias(),
       DatabaseManager.ACTION_STOP_CONNECTION_POOL);
      }
}
```

Whenever the above row subclass is used, the row actions will display to the user. These are the basics of creating a Row Action. There are also many other configurations and extra properties that can be used to customize Row Actions.

To customize your row actions, you can override the getActionDescriptor() method and change settings such as action image and action display type. In the example below, the stop and start actions have the same display type but different images.

```
public ActionDescriptor getActionDescriptor(AbstractAction action) {

 // get type and name of action
 String type = action.getType();
 String name = action.getName();

 // return default descriptor for row actions
 if (IRowAction.TYPE.equalsIgnoreCase(type)) {
       if (ACTION_START.equalsIgnoreCase(name)) {
             ActionDescriptor ad = ActionDescriptor.createDescriptor(type,
 name);
             ad.setImageSource("dataengine/images/start.gif");
             ad.setDisplayType(ActionDescriptor.DISPLAY_TYPE_LINK);
             return ad;
       }
  else if (ACTION_STOP.equalsIgnoreCase(name)) {
   ActionDescriptor ad = ActionDescriptor.createDescriptor(type, name);
       ad.setImageSource("dataengine/images/stop.gif");
       ad.setDisplayType(ActionDescriptor.DISPLAY_TYPE_LINK);
       return ad;
   }
 }

      return super.getActionDescriptor(action);
}
```

In some cases you may want to hide actions depending on certain settings or behaviors. In that case, you can override the isActionApplicable() method. In the isActionApplicable() method, you would test for the particular action name, test if applicable and then return true if applicable. The isActionApplicable() is always called before displaying the actions.

```
    /**
    * Check to see if the action is applicable for this connection row
     */
    public boolean isActionApplicable(String action, ExecutingContext ec) throws
    CMException {

    if (ACTION_START.equalsIgnoreCase(action)) {
     if (doesConnectionPoolExist()) {
     return false;
      }
    else {
     return true;
        }
}
```

There may be cases were you add row actions to a row but want to control whether the action is displayed as well as its properties by using Property Groups. When you override the Row.getRowHandledActionNames() method, all the actions are displayed by default to the user. Another way to add actions is to add an Actions{} and ActionDescriptor{} property group with the Action specified. Define the type of action and action name (see example below) to include with the Row. In this case the Row.getRowHandledActionNames() does not have to be overridden. **NOTE: You still need to override the handleAction() to define action.**

The following row property groups create an “action_name” row action with the label “My Action” that appears to the left of the row when displayed to the user.

```
Actions {
type: row;
show: action_name;
}

ActionDescriptor {
   name: action_name;
   actTyp: row
   label: My Action;
}
```

Both of the property groups above are necessary to create an action. To check out all the possible properties and values for the ActionDescriptor property group, see the Operation Actions section.

10.7.1. Example of Another RowAction

In this example we will be creating a new “Compile” row action. To create a basic row action we must first create a Row subclass. In the WOW web project in the src folder you can create your own package to hold the new Row subclass. As an example, you can create a package in the src folder and name it com.test. This will extend the Row. To go back to our “compile” example, create a Row Subclass with the name of CompileActionClass as shown in example below.

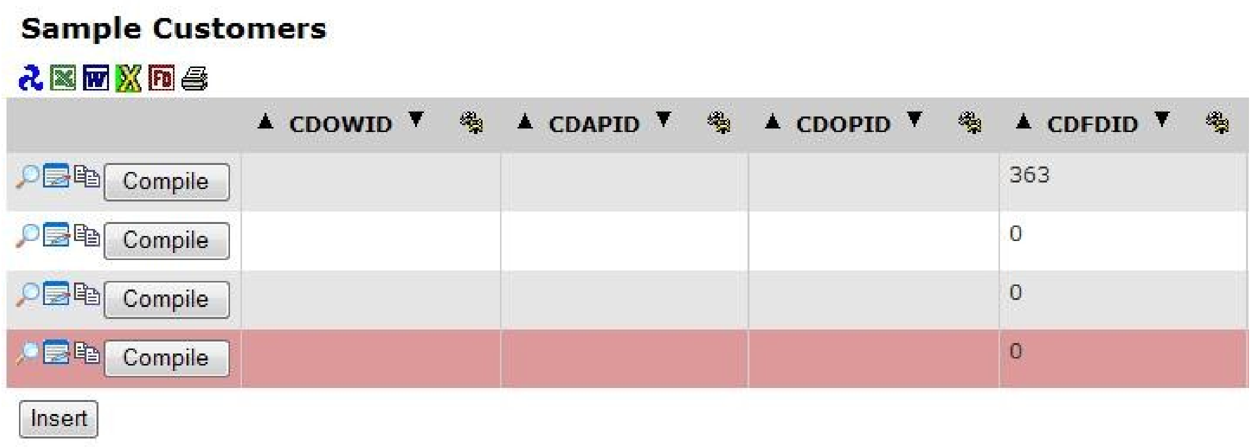
```
public class CompileActionClass extends Row {
 -------------------
 // class body //
 -------------------
}
```

Now, you must override the Row.getRowHandledActionNames() method. This method is called when WOW is generating the Row and checking whether or not any actions *exist*. For a Row Action to be displayed, you need to add it to the Action Array List which is created within getRowHandledActionNames(). Below you can see one action added to the connection row:

First Compulsory Method:

```
/**
* Get a list of actions this connection row supports.
*/
public List getRowHandledActionNames () {
 List actions = super.getRowHandledActionNames ();

 if ( actions == null ){
        actions = new ArrayList();
 }
 actions.add("Compile");
 return actions;
}
```

Running the application after adding the action, your screen will show the 'Compile' button shown below. Remember, you haven't added any action to the 'Compile' button yet, so it will not have functionality yet.



Sample Customers

	CDOWID	CDAPID	CDOPID	CDFDID
Compile				363
Compile				0
Compile				0
Compile				0

Insert

You can add multiple actions in the list like:

```
public List getRowHandledActionNames () {
 List actions = super.getRowHandledActionNames ();

 if ( actions == null ){
        actions = new ArrayList();
 }
 actions.add("Action1");
 actions.add("Action2");
 actions.add("Action3");
 actions.add("Action4");
 actions.add("Action5");
 return actions;
}
```

Second Compulsory Method:

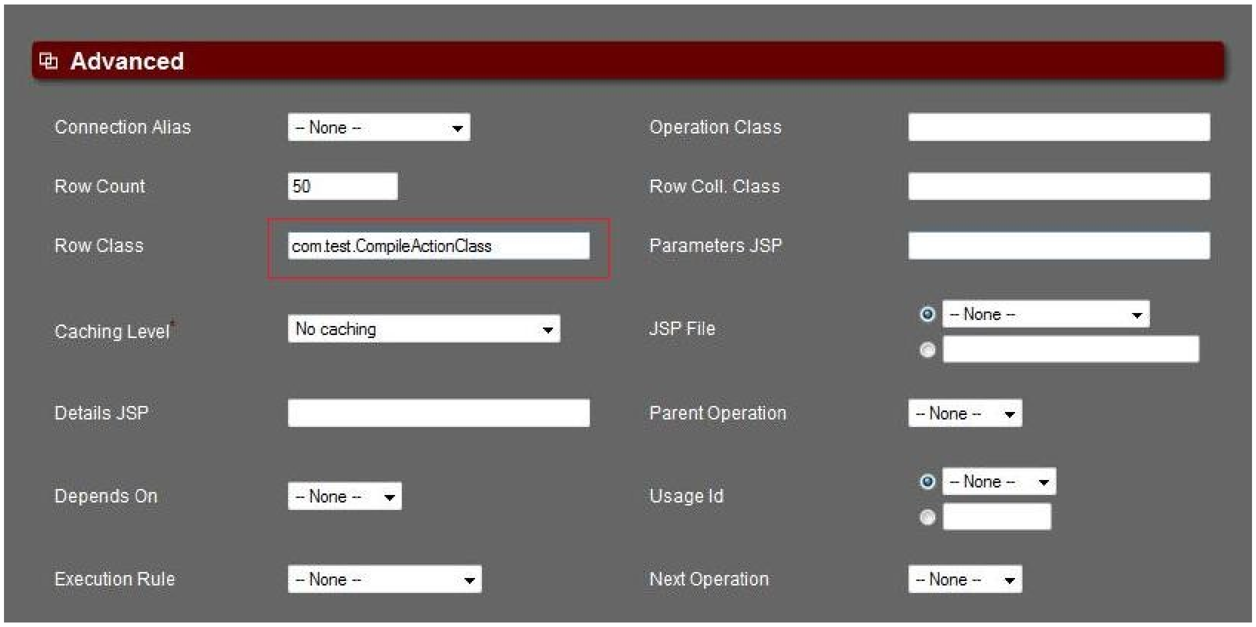
For a Single Action:

```
public Object handleAction (String action, ExecutingContext ec) throws
CMException {
 super.handleAction(action, ec);

 if( action.equals("Compile")){
       //Custom Coding of Action
       return "Compile";
 }
 return "Unknown Action Called";
}
```

For multiple actions:

```
public Object handleAction (String action, ExecutingContext ec) throws
CMException {
 super.handleAction(action, ec);

 if( action.equals("Actions1")){
       //Custom Coding of Action
       return " Actions1";
 }
 if( action.equals("Actions2")){
       //Custom Coding of Action
       return " Actions2";
 }
 if( action.equals("Actions3")){
       //Custom Coding of Action
       return " Actions3";
 }
 if( action.equals("Actions4")){
       //Custom Coding of Action
       return " Actions4";
 }
 return "Unknown Action Called";
}
```

After creating a new subclass, it is necessary to attach that class within an operation in WOW. When clicking on the edit button on any operation in the Advanced Section there is a field 'Row Class'. This is where the new class with its package name should be added. Suppose we created the class 'CompileActionClass' in com.test package. It is necessary now to insert the new class:'com.test.CompileActionClass' in the row class field.   It is not necessary to mention .class or .java with class name. WOW already knows about it.



Advanced

Connection Alias	-- None --	Operation Class	
Row Count	50	Row Coll. Class	
Row Class	com.test.CompileActionClass	Parameters JSP	
Caching Level*	No caching	JSP File	-- None --
Details JSP		Parent Operation	-- None --
Depends On	-- None --	Usage Id	-- None --
Execution Rule	-- None --	Next Operation	-- None --

After this, click the Update Operation button. If the class name is not correct, WOW will display an error. WOW will accept the operation only if the name is a valid class file.

Now that the action names have been added to the actions list, these actions will show up when this Row is generated, but they will do nothing when clicked. To add Action functionality, the method Row.handleAction() has to be overridden. The method handleAction() takes in a String and the current ExecutingContext. The String is the name of the action that was clicked on. If there are multiple action names with different actions, it is necessary to test for the correct action within handleAction(). After testing, if it's the correct action, then all action code should be included within the handleAction() method.

```
/**
*Basic connection row handle action method which handles given action.
*/
public Object handleAction (String action, ExecutingContext ec) throws
CMException {
 super.handleAction(action, ec);

 if( action.equals("Compile")){
       Field cdCode = this.getField("CDCODE");
       String cdCodeString = getValueAsString("CDCODE");

       // Custom Coding for Compiling Code

       return "Compile";

 }
 return "Unknown Action Called";
}
```

Whenever the above row subclass is used, the row actions will display to the user.

11. Fields

As previously mentioned above, Fields are contained within a Row's FieldCollection. Each Field has a FieldDescriptor (described below), as well as possibly having a Formatter and/or PossibleValues. All three are used for displaying or containing information about the Field.

11.1. Creating a Field

Fields are created internally within the DataEngine when RowCollections are retrieved. In some situations, however, they might need to be created by the user. For example, if a Field has an override class for its possible values, the user may wish to create their own RowCollection rather than using the DataEngine. To create a Field, call the following method:

```
Field.create(String fieldname, int sqlType).
```

This will return a subclass of Field (see next section) for the correct java.sql.Types with the given name and index. The name is the Field's column name within the database.

Note: SQL type arguments should be one of the java.sql.Types (e.g. java.sql.Types.INTEGER).

11.2. Field Classes

To allow for the correct formatting of different java.sql.Types, each type has its own subclass of Field. This allows for multiple benefits:

1. A Field subclass already knows its java.sql.Type, therefore it returns the proper format of its SQL value.
2. Proper value validation is handled by the Field's Class
3. If you know the Field subclass type, for example an IntegerField, you can directly retrieve its int value ((IntegerField) field).getInt(). [Similar for other fields as well]

11.2.1. Default Class

When a column value is read from the database, the proper Field class is created depending upon its type. The java.sql.Types are currently mapped to the following Field classes.

<u>BigIntegerField</u>

- SMALLINT
- INTEGER
- BIGINT

<u>BigDecimalField</u>

- FLOAT
- REAL

- DOUBLE
- NUMERIC
- DECIMAL

StringField

- CHAR
- VARCHAR
- LONGVARCHAR

DateField

- DATE

TimeField

- TIME

TimestampField

- TIMESTAMP

11.2.2. Custom Class

To enhance the meaning of data in the database even further, custom Field classes can be used. A custom class will ensure a Field's value has:

1. Correct display formatting
2. Proper validation

The following code snippet shows how to set the Field's Field Class on the FieldDescriptor (assumes the FD has already been created and retrieved).

```
fd.setFieldClass(<class>);
```

Note: When setting a custom class, the class's fully qualified name must be used. An example would be "planetj.database.field.UserIdField".

In the next few sections, some pre-built DataEngine custom Field classes are listed and described. All pre-existing Field classes provided by the DataEngine are in the package planetj.database.field.

DateField

In the database, a date field might not necessarily be a Date object. It could be a String or a number. To allow for proper reading and writing to the database, a Field's Field class could be set to the DateField, followed with a comma and any user-defined pattern for date formatting. The pattern should be the format the value should be written as when inserted or updated to the database. The DateField object uses a java.text.SimpleDateFormat to generate the correct value for inserting or updating. See below for an example pattern.

Note: See java.text.SimpleDateFormat JavaDoc to get a complete list of all reserved characters.

Fully qualified class name: planetj.database.field.DateField

Ex: Benefit of Setting DateField Field Class

In the database we have a field that is a CHAR with a length of 8. It takes the format 2 month, 2 day, and 4 year. When we read the value from the database, it has no meaning and has to be modified whenever we want to display the date. If we set the field class to DateField, then a java Date object will be generated, which gives the Field's value more meaning and flexibility. If we ever need to insert or update the value, it needs to be in the format listed above. Do this by using the reserved characters of java.text.SimpleDateFormat (MMddyyyy). Now all we need to do is set the field class to the following (fully qualified class name followed by a comma and then the pattern than the date needs to be in order to be written to the database).

```
planetj.database.field.DateField,MMddyyyy
```

EmailField

In the database, an EMAIL column's value might be support@planetjavainc.com. By setting the Field Class to an EmailField, the proper display value will be generated automatically. Plus, the proper validation will occur if the EMAIL value is changed. For example, in HTML, the display value would be generated as follows:

```
<a href=”mailto:support@planetjavainc.com”>support@planetjavainc.com</a>
```

Thus, a user could click on the link, then type and send an email to the address. This would be especially beneficial if you wanted to display a list of emails. Rather than manually coding each email link, setting the Field class to EmailField would automatically generate the links.

The validation of an email field ensures the value contains a '@' symbol as well as a '.' by using a helper class planetj.validation.Validator.

Fully qualified class name: planetj.database.field.EmailField

FirstNameField

The validation of a first name field to ensure that the value is valid for a first name.

Fully qualified class name: planetj.database.field.FirstNameField

LastNameField

The validation of a last name field to ensure that the value is valid for a last name.

Fully qualified class name: planetj.database.field.LastNameField

PasswordField

This Field ensures that when a PasswordField is displayed it will be replaced with asterisks. This can be very useful for sensitive information such as passwords.

Fully qualified class name: planetj.database.field.PasswordField

PhoneNumberAreaCodeField

The validation of an area code field ensures that the length of the area code is the correct length and only contains digits. If the value contains '(', ')', or '-' they are also accounted for. This class also uses a helper class planetj.validation.Validator for validation.

Fully qualified class name: planetj.database.field.PhoneNumberAreaCodeField

PhoneNumberField

The validation of a phone number field ensures that the length of the phone number is the correct length and only contains digits. If the value contains '(', ')', or '-' they are also accounted for. This class also uses a helper class planetj.validation.Validator for validation.

Fully qualified class name: planetj.database.field.PhoneNumberField

SocialSecurityField

The validation of a social security field ensures that the length of the social security is correct and its value only contains digits. If the value contains any '-' they are also accounted for. This class also uses a helper class planetj.validation.Validator for validation.

Fully qualified class name: planetj.database.field.SocialSecurityField

UpperCaseStringField

This Field ensures that its value is always upper case when displaying, inserting or updating to the database.

Fully qualified class name: planetj.database.field.UpperCaseStringField

UserIdField

This field has been enhanced to get the current user from its context.

Fully qualified class name: planetj.database.field.UserIdField

YBlankBooleanField

This Field can be used when you want the value in the database to be of type CHAR and length 1. Since the field is a boolean field, then it will be displayed in the form of a check box. Thus, the user cannot enter the wrong value. When setting the value programmatically, a boolean value 'true' or 'false' can be used, which will in turn set the value to 'Y' or ' '.

Fully qualified class name: planetj.database.field.YBlankBooleanField

YNBooleanField

Same as YBlankBooleanField except a ‘N’ instead of a ‘ ‘.

Fully qualified class name: planetj.database.field.YNBooleanField

ZipCodeField

The validation of a zip code field ensures the zip code is the correct length and only contains digits. If the value contains a ‘-‘ they are also accounted for. This class also uses a helper class planetj.validation.Validator for validation.

Fully qualified class name: planetj.database.field.ZipCodeField

ZipCodeSuffixField

The validation of a zip code suffix field ensures the zip code suffix is the correct length and only contains digits. If the value contains a ‘-‘ they are also accounted for. This class also uses a helper class planetj.validation.Validator for validation.

Fully qualified class name: planetj.database.field.ZipCodeSuffixField

Ex: Setting Field’s Custom Field Class

A value in the database is a CHAR with a length of 6. This value represents a Date. The problem is when it gets displayed in a gui, it doesn’t have any meaning. It will look like an ordinary number. The solution is simple, all that needs to be done is set the Field’s Field Class to a custom Field class that can handle the proper formatting of the value. For this example, lets say we want the date to be displayed like MM/dd/yyyy. All we have to do is set the proper Field Class in the Field’s FD. This code example assumes the FieldDescriptor has already been retrieved and is called ‘FD’ (see Retrieving a FieldDescriptor).

```
fd.setFieldClass(“planetj.database.field.DataField,MM/dd/yyyy”)
```

In the above code, the comma separates the fully qualified name of the Field class from the format of the Date we want to use.

11.2.3. Benefits of Using Custom Field Classes

At first glance, some of the above Field classes might not seem to be of much benefit. The real power behind them lies in what the DataEngine can do just by knowing a Field’s class.

1. Server-side validation
2. Display formatting
3. Client side validation
4. “Smart” code (see example below)

Example – Smart Sign on Using Field Classes

For some applications (#1) you need to have a sign on. You would then need to check the user id and password against some table in the database. This would require that you know the column name for the user id and the column name for the password, as well as the table to look in to check if the user exists. Then, there is another application (#2) which also needs a sign on. This would require you to know the same information. The problem is that this application might check a different table as well as have different column names for the user id and password.

This can be solved with "smart" code. Just set the user id Fields to use the custom Field class UserIdField. And then set the password Field's to use the custom Field class PasswordField. Then when you go to do the sign on, the same code can be used. This time all you need to do is specify the name of the table to look in. The column names of the user id and password can be retrieved from the table's FieldDescriptors. This allows the lookup to dynamically select the proper fields containing the user id and password. Plus the same code could be used for each sign on, and possibly other applications if there were more.

12. Working With FieldDescriptors (FD's)

A FieldDescriptor is an object that contains characteristics that describe any given field in a table. Some of the characteristics included in a FieldDescriptor are its field name, its label, max length, its type (class), and many others. FieldDescriptors allow for easier display and formatting of a field and its data. A good example would be in HTML; FieldDescriptors can be used to automatically format the proper HTML for any given field.

12.1. Creating

FieldDescriptors can be created by inserting FieldDescriptor information into the FLDDATA file. A FieldDescriptor is a Row and therefore can easily be inserted into the FLDDATA file after creation using the Row's insert() method. There is no need to enter it manually.

12.1.1. Auto Population of Field Data File

By using static methods on the FieldDescriptorManager, you can also auto populate the FLDDATA file using DatabaseMetaData to fill in the basic needed portion for a FieldDescriptor. Then all you need to do is modify parts of the record for the FieldDescriptor, rather than creating the entire entry. The following are the static methods on FieldDescriptorManager that can be used to create FD's for fields.

1. Library – generate FD's for all fields in all tables in the given Library

```
createFieldDescriptorsFromDBMetaData(Library)
createFieldDescriptorsFromDBMetaData(Connection, Library)
```

2. Table – generate FD's for all fields in the given Table

```
createFieldDescriptorsFromDBMetaData(Table)
createFieldDescriptorsFromDBMetaData(Connection, Table)
```

3. Field – generate an FD for the given Field or using the given field name for the given Table

```
createFieldDescriptorsFromDBMetaData(Field)
createFieldDescriptorsFromDBMetaData(Connection, Field)
createFieldDescriptorsFromDBMetaData(Table, String)
createFieldDescriptorsFromDBMetaData(Connection, Table, String)
```

Example – Auto Population of FD's

Let's say you had a table called 'Employees' in the library PlanetJ, that contained three fields; FN, LN, and SS#. These columns might not be very intuitive to a user who is viewing records. This is where FieldDescriptors come in handy. We can create an FD for each column of the table and set an external name for each.

First, the Table object needs to be created. The method below is just one of a couple ways. The boolean true means to create the Table object if it doesn't already exist.

```
Table table = Table.getTable(<system alias>, "PlanetJ", "Employees", true);
```

Next, we can use the Table object to create FieldDescriptors from database metadata. FD's could be created individually be creating the object manually and setting its values, but by auto populating using database metadata, most of its necessary properties get set (such as its size, name, library, table, default value, etc…).

```
FieldDescriptorManager.createFieldDescriptorsFromDBMetaData(table);
```

Instead of the above method, we could have made three separate calls to the FieldDescriptorManager's create FD method that takes a Table object and the field name. But that would take 3 database hits rather than 1. Now that the FD's are created, we can retrieve each and set its external name.

```
FieldDescriptorRow fd = table.getFieldDescriptor("FN");
fd.setExternalName("First Name");
fd.insert();

fd = table.getFieldDescriptor("LN");
fd.setExternalName("Last Name");
fd.insert();

fd = table.getFieldDescriptor("SS#");
fd.setExternalName("Social Security Number");
fd.insert();
```

Now that the FD's have been created, modified and inserted, whenever a RowCollection (results) gets displayed the column names will be that of the FD's' external names.

12.2. Retrieving an FD

A Field's FieldDescriptor can be retrieved in many ways. The following lists the objects and when they can be used to obtain a Field's FieldDescriptor as well as the methods to do it.

```
Field.getFieldDescriptor()
Row.getFieldDescriptor(String fieldName)
Table.getFieldDescriptor(String fieldName)

// get all FD's for all Fields in Row
Row.getFieldDescriptors()

// get all FD's for Table
Table.getFieldDescriptors()
```

Note: When calling getFieldDescriptor() from a Field or Row, the Field's FD is returned. FieldDescriptors are stored by the Table they belong to. A Field, however, may have its own unique instance of an FD, which will allow for special casing of a Field.

12.2.1. FieldDescriptorManager

FieldDescriptors can also be retrieved through static methods in a convenience class called FieldDescriptorManager.

```
getFieldDescriptor(String libName, String tblName, String fldName)

getFieldDescriptor(String sysURL, String libName, String tblName, String fldName)

getFieldDescriptors(String libName, String tblName)

getFieldDescriptors(String libName, String tblName, boolean checkFile)

getFieldDescriptors(String sysURL, String libName, String tblName)

getFieldDescriptors(String sysURL, String libName, String tblName, Boolean
   checkFile)
```

The parameters from the methods above are:

- sysURL: system url
- libName: library name
- tblName: table name
- fldName: field name
- checkFile: boolean indicating whether or not to see if the descriptor for the field is stored in the FLDDATA file (see FieldDescriptor Data File in FieldDescriptor section).

12.3. FD Operations

12.3.1. Cloning an FD

Cloning an FD can be very useful, especially when a Field needs to be handled differently in some case than it would normally. An FD is a subclass of Row and therefore is cloned the same way (see Cloning a Row).

Example – Why Cloning an FD Might Be Useful

This is a simple, but explanatory example. There are 5 different applications all using the same FieldDescriptors. The problem is that one of the applications wants the external name of one of the columns of a Table different than the other 4 applications want it. To solve this problem, the application that wants to have a different external name, can clone the FieldDescriptor, set the new external name, and then set the cloned FD on any Field's it wants to have use it. Then any call to get the Field's FD, will return the Field's own unique FD.

13. Possible Values (PV)

Fields often have possible values. For example, a gender field can be "Male" or "Female". Some fields can have possible values, as well as possible display values. The DataEngine allows for the configuration or population of possible values for a particular field. There are 3 ways to establish possible values for a field.

1. Specification of a Java class that will return possible values. The DataEngine supplies many common possible value classes for your use. You can also easily write your own class.
2. Population of possible values as described below.
3. Specification of an SQL statement as described below.

To allow possible value population and SQL specification, there is a file called PLANETJ.DEPVDTA (DataEngine Possible Values Data). Possible values for a field can be obtained directly from this file or by an SQL statement specified on this file.

13.1. Creating Possible Values

There are a couple of ways of creating possible values for fields.

13.1.1. Using an SQL Statement For Possible Values

An SQL statement can be used to specify to the DataEngine where and how a field's possible values can be retrieved. In order for it to work properly, the following apply:

1. The KEY for the possible value should be "*SQL*".

2. The VALUE contains the SQL statement that can be executed to retrieve the possible values. (see **Note** below for format of SQL statement)

3. Nothing needs to be put in the DSPVAL and DSPVALSQLT columns.

Note: SQL statements must be of the following format. When executed, they should return a result set containing either a column containing a value OR two columns, one with a value and the second with its display value.

```
SELECT <value column> [,<display value column>] FROM ……
```

*** Currently possible values do not work with prepared statements.**

13.1.2. Using a Key For Possible Values

A unique key can also be used for retrieving possible values from possible values file. In order to use a unique key, the following must apply.

1. The KEY field value should be the unique key (e.g. – "usstates").
2. The VALUE field would then contain the possible values identified for a field using that KEY (e.g. – CA, …, MN, … etcetera).
3. The DSPVAL and DSPVALSQLT can optionally have data (e.g.- California, … Minnesota, …, ecetra).

13.1.3. Using a Value For a Possible Value

The values set in the VALUE column need to be in the proper string format for the field's FieldDescriptor to be able to convert it into the correct object. The class SQLGenerator can be used to provide the proper string formatting for an object.

```
SQLGenerator.getValueAsString(Object obj, int sqlType)
```

Note: the SQL type would be the same java.sql.Types as the Field to which it is a possible value.

13.1.4. Using a Value and a Display Value For Possible Values

The <optional> values set in the DSPVAL column need to be in the proper string format as well. See "Using a value for a possible value" above for the reasons why. The same applies for display values. The only difference is if a value has a display value, then the display value's java.sql.Type needs to be supplied (DSPVALSQLT) in order for the display value to be converted to the correct object.

Example – Description of Possible Values' Display Values

A field's possible values might be "one", "two", and "three", but their respective display values might be 1, 2, and 3. The display values in turn need a java.sql.Types in order to convert them into their proper form. In this case it would be java.sql.Types.Integer.

Note: the value's SQL type can always be retrieved from the Field's FieldDescriptor.

13.2. Creating a PV Class

13.2.1. Using Existing PV Classes

Enter the package qualified name of the Java class into file PLANETJ.FLDDATA using the PVClassName field (char 50). When a field is asked for possible values, the supplied class will be instantiated and called to return the possible values. The DataEngine supplies the following classes:

Class Name	Key	Description
USStatesPV	*US_STATES*	Value: 2 digit State abbreviation Display Value: State description (Ex. IA-Iowa)
DayOfTheWeekFieldPV	*DAYS_OF_THE_WEEK*	Value: 1 digit integer Display Value: Day description (Ex. 1-Monday)
MonthFieldPV	*MONTHS_OF_THE_YEAR*	Value: 2 digit integer Display Value: Month description (Ex. 01-January)
SQLTypePV	*SQL_TYPES*	Value: SQL Type integer Display Value: String representation (Ex. 1-CHAR)

13.2.2. Create Custom PV Classes

To create your own possible value retriever class, follow the steps below:
Create a Java class that implements the planetj.dataengine.fielddescriptor.IPossibleValueGetter interface.

1. Implement the following methods:
2. RowCollection getPossibleValues() // Return the possible values
3. RowCollection getPossibleValues(Field) // Return the possible values considering the field or any other field in the field's row.
4. It is recommended that your possible value class use the DataEngine.getRows() methods to return a RowCollection. If the RowCollection returns 1 column, that column will be used for the value and display value. If the RowCollection returns 2 columns, it is assumed the 2nd column will be the display value.

13.3. Retrieving Possible Values

Possible values can be retrieved through a given Field, its FieldDescriptor, or the PossibleValueManager. If a possible values class is specified on the Field's FieldDescriptor, then it is used and invoked in order to retrieve the Field's possible values.

13.3.1. FieldDescriptor

```
getPossibleValues(Field field)
```

13.3.2. PossibleValueManager

The follow static methods can all be called to retrieve an array list of possible values from the PossibleValueManager.

The first four methods take the field itself as a parameter and optionally specify whether or not the possible values should be cached when retrieved and also optionally specifying a key (e.g. – “usstates”, or “*SQL*”)

```
getPossibleValues(Field)
getPossibleValues(Field, boolean)
getPossibleValues(Field, String)
getPossibleValues(Field, String, boolean)
```

The next four methods are similar to the above methods only differing in they take the field’s system url, library name, table name, and name.

```
getPossibleValues(String, String, String, String)
getPossibleValues(String, String, String, String, boolean)
getPossibleValues(String, String, String, String, String)
getPossibleValues(String, String, String, String, String, boolean)
```

Note: When no key is specified the PossibleValueManager will take the following actions.

1. First it checks to see if any possible values exist for the field that do not have a key.

2. Then, if there were not any, it checks to see if there are possible values specified by the key *SQL*.

3. If there still isn’t any, then null is returned.

14. “Magic” Requests

14.1. Requests

14.1.1. Page

When HTML is generated for a RowCollection, next and previous links are also generated as described in Next or Previous RowCollection. These links are handled via a Magic Request.

14.1.2. Sort

You can magically sort an HTML table by any column by clicking on the up {▲} and down {▼} arrows in the column header you wish to sort. To see more on sorting (See Sorting a RowCollection). The magic sort uses a SortRequest class which contains constants for the sort order (SortRequest.ASC and SortRequest.DESC).

14.1.3. Refresh

You can magically refresh an HTML table by clicking on the refresh pinwheel { }. To see more on refreshing (See Refreshing a RowCollection). This magic refresh uses a RefreshRequest class which is responsible for the magic.

14.1.4. CSV/Excel

You can magically display an HTML table as a CSV (Comma Separated Variable) file in your browser using an Excel plug-in. This can be done by clicking on the Excel icon { }. To see more on generating a CSV file (See Generating a CSV file from a RowCollection). This magic request uses an ExcelRequest class which is responsible for the magic.

14.1.5. Microsoft Word

You can magically generate a Microsoft Word file from any database table using the MSWord Magic Request by clicking on the Microsoft Word icon { }. Currently, the MSWord Magic Request calls the (DOCHelper) which uses (CSVHelper). To see more on more on generating a Microsoft Word file (See Generating a Microsoft Word file from a RowCollection). This magic request uses an MSWordRequest class which is responsible for performing the magic.

14.1.6. XML

You can magically display an HTML table or RowCollection in XML format in your browser. This can be done by clicking on the XML icon { }. To see more on generating a XML file (See Generating a XML file from a RowCollection). This magic request uses an XMLRequest class which is responsible for the magic.

14.1.7. PDF

You can magically display an HTML table or RowCollection in PDF format in your browser using an Adobe Acrobat plug-in. This can be done by clicking on the PDF icon { }. To see more on generating a PDF file (See Generating a PDF file from a RowCollection). This magic request uses an PDFRequest class which is responsible for the magic.

15. Report Breaks

A report break is one or more Rows containing metadata about a group of Rows in a RowCollection. For example, a report break might contain the average value of a column over all the Rows in a RowCollection. Typically a Row containing a report break is inserted into the RowCollection along with the Rows containing the actual data, but is displayed in a different color to distinguish it from the data Rows.

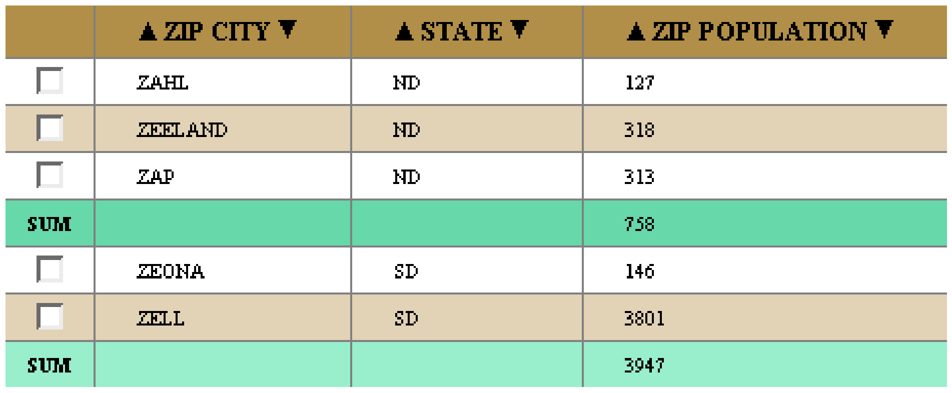
15.1. Report Break Functions

There are five functions for which Report Breaks can be generated:

SUM – The sum of a series of values
AVG – The average of a series of values
COUNT – The number of values in a series of values
MAX – The maximum value in a series of values
MIN – The minimnum value in a series of values

15.2. Break Columns

Sometimes you will want to generate a report break for a subset of the Rows in a RowCollection. For example, if your RowCollection has three columns CITY, STATE, and POPULATION (listing the population of major cities) you may wish to generate a report break containing the total population for each state. In this case the STATE column would be known as the break column, since every time the value in this column changes, a report break should be generated and inserted into the RowCollection (this assumes that the Rows are ordered by STATE). This is what the RowCollection might look like, after the report breaks have been inserted:

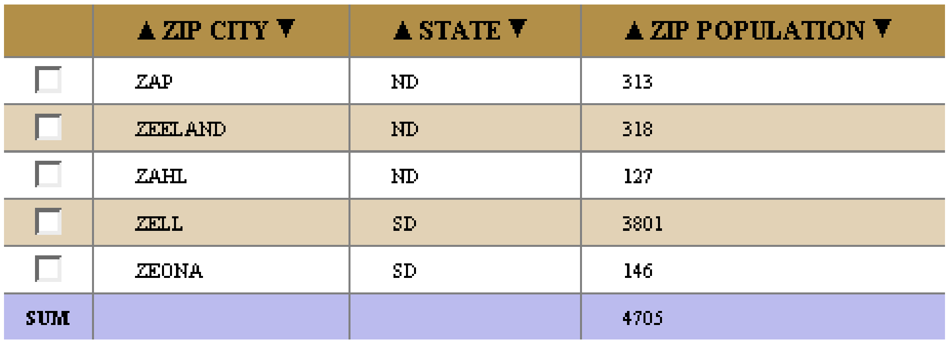


	▲ ZIP CITY ▼	▲ STATE ▼	▲ ZIP POPULATION ▼
☐	ZAHL	ND	127
☐	ZEELAND	ND	318
☐	ZAP	ND	313
SUM			758
☐	ZEONA	SD	146
☐	ZELL	SD	3801
SUM			3947

Report breaks with break columns are also referred to as “normal” report breaks, as opposed to overall report breaks, which are discussed in the next section.

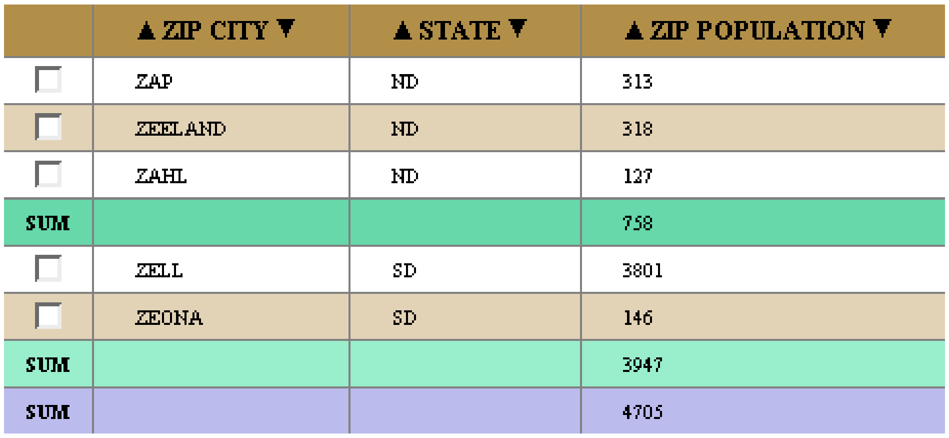
15.3. Overall Report Breaks

An overall report break is uses all the Rows in the RowCollection to generate its data. This means that overall report breaks do not have any break columns, since they are only generated at the very bottom of the RowCollection. If the above example used an overall report break instead of break columns, this is what it would look like:



	▲ ZIP CITY ▼	▲ STATE ▼	▲ ZIP POPULATION ▼
☐	ZAP	ND	313
☐	ZEELAND	ND	318
☐	ZAHL	ND	127
☐	ZELL	SD	3801
☐	ZEONA	SD	146
SUM			4705

It is also possible for both overall and normal report breaks can be included in the same RowCollection:



	▲ ZIP CITY ▼	▲ STATE ▼	▲ ZIP POPULATION ▼
☐	ZAP	ND	313
☐	ZEELAND	ND	318
☐	ZAHL	ND	127
SUM			758
☐	ZELL	SD	3801
☐	ZEONA	SD	146
SUM			3947
SUM			4705

15.4. Creating Report Breaks

In order to generate Report Breaks, you first need to create a ReportBreakCollection object:

```
ReportBreakCollection rbc = new ReportBreakCollection();
```

A ReportBreakCollection can contain multiple ReportBreak objects, each of which specifies a report break which should be generated. To add a new ReportBreak to a ReportBreakCollection, use the addNewReportBreak(String, List, String, boolean) method. The parameters to this method are:

<u>String **pColumnFunctionToken**</u> – A constant defined in the ColumnFunctionToken class, indicating what function (SUM, MAX, etc) the report break should use

<u>List **pFieldNames**</u> – A List containing the names of the columns for which report breaks should be generated.

String **pBreakColumnName** – The name of the break column. For overall report breaks, this should be null

boolean **pAddOverall** – If an overall report break should also be generated. Often when a normal report break is generated, you also want an overall report break to be generated. When this parameter is true an overall report break will be generated in addition to the normal report break (this parameter is ignored if the pBreakColumnName parameter is null, indicating that this is already a normal report break).

To create the report breaks (both normal and overall) in the above example, we would invoke the method like this:

```
List list = new ArrayList();
list.add ("ZPPOP");
rbc.addNewReportBreak (ColumnFunctionToken.SUM, list, "ZPSTA", true);
```

Note that ZPSTA and ZPPOP are the respective internal names of the State and Population columns.

Finally, our ReportBreakCollection needs to be added to an SQLContext:

```
SQLContext context = new SQLContext();
context.setReportBreakCollection (rbc);
```

When this context is used to retrieve rows from the database, the report breaks will be inserted into the context by the DataEngine.

16. HTML Helpers

The HTML helper classes allow programmers to generate JSP's without having to code a lot of Java or HTML. They are mostly designed to take in different DataEngine objects and generate HTML output for them.

16.1. HTML Generator

HTMLGenerator is a convenience class for generating HTML code.

16.2. HTML Extractor

HTMLExtractor is a convenience class for extracting data out of a HttpServletRequest.

16.3. HTMLComparisonInput

This helper class can be used to automatically generate HTML code for input fields to match search criteria for a given parameterized SQL String of an SQLContext. It also provides functionality to extract parameter values for the search criteria entered by a user into an SQLContext object.

This is the method used to generate input fields.

```
generateInput(SQLContext, HttpServletRequest, HttpServletResponse)
```

This is the method used to extract values from previously generated input fields.

```
loadSQLParameterValues(SQLContext, HttpServletRequest)
```

Example – Generate Search Criteria

The scenario is we want users to type in an email address and we will show them all the records of the PLANETJ.USERS file that have the same email address. The following steps illustrate how to go about doing this using the HTMLComparisonInput.

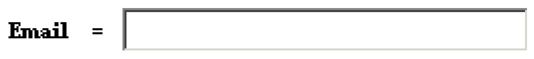
First we need to create an SQLContext containing the SQL for the record look up.

```
SQLContext context = new SQLContext(<system alias>);
context.setSQL("SELECT * FROM PLANETJ.USERS WHERE EMAIL = ?");
```

Next, in the JSP file we can use HTMLComparisonInput and context created to automatically generate HTML code for input fields of the search criteria.

```
HTMLComparisonInput.generateInput(context, request, response);
```

The above code will generate the following.



Email =

The JSP will also need some button or link that will call code to do the following. After the user types an email in, and clicks the link or button, the code being called will do the following.

```
HTMLComparisonInput.loadSQLParameterValues(context, request);
```

The context passed in should be the same context as created previously. This method call will extract the value entered by the user and set it as a parameter value on the SQLContext object. Then, to get all the records, do the following. It will return a RowCollection of all records in PLANETJ.USERS that have the same email address as the one entered by the user.

```
DataEngine.getRows(context);
```

16.4. HTML Elements

All HTML generation can be done using convenience methods in HTMLGenerator. For more functionality, however, individual HTML Elements should be created and used.

16.4.1. HTMLTable

This object can be used to auto generate HTML code to display results from a query. AN HTMLTable also contains MagicRequests that can also be used to carry out other tasks such as:

1. Convert results to CSV, Microsoft Word, XML, or PDF
2. Print results
3. Sort results
4. Copy, Delete, Insert, Update, and View record(s).
5. Modify FieldDescriptors associated with the results.

Creating an HTMLTable

The following three constructors can be used to create an HTMLTable object. When an HTMLTable generates HTML code for its results passed in, it needs a key to store the results (a RowCollection) in the session. This is to allow any MagicRequests to access the results to carry out their tasks. Also, when dealing with Row MagicRequests (insert, update, etc…), the HTML table needs a key for storing a Row in the session.

By default, the RowCollection session key is IDataEngine.ROW_COLLECTION, and the Row session key is IDataEngine.ROW. If multiple HTMLTables are on a page or if you want to store the RowCollection or a Row in the session using a different id, then you can specify that key in the constructor when creating the HTMLTable object.

```
HTMLTable htmlTable = new HTMLTable();
HTMLTable htmlTable = new HTMLTable(String);
HTMLTable htmlTable = new HTMLTable(String, String);
```

If another RowCollection needs to have HTML code generated for it, rather than creating a new HTMLTable object and setting all its properties again, you could use the same HTMLTable object except just set its new RowCollection session key and Row session key if needed. Then you could call the generate method passing in the second RowCollection.

```
htmlTable.setRowCollectionId(String)
htmlTable.setRowId(String)
```

Turning MagicRequests On or Off

An HTMLTable object contains a bunch of methods that can be used to turn MagicRequests on or off.

The following methods can be used to turn on or off the corresponding link (icon).

```
setMagicRefresh(boolean);
setMagicExcelLink(boolean);
```

	`setMagicMSWordLink(boolean);`
	`setMagicXMLLink(boolean);`
	`setMagicPDFLink(boolean);`
	`setAllowPrint(boolean);`
&	`setMagicEditFD’s(boolean);`
▲ & ▼	`setMagicSortingLinks(boolean);`

Also, all of these can be turned on or off at once with a call to this method.

```
setAllMagicLinks(boolean);
```

The following methods can be used to turn on or off the corresponding button with the given name.

Insert	`setMagicInsert(boolean);`
View	`setAllowViewDetails(boolean);`
Edit	`setMagicEdit(boolean);`
Copy	`setAllowRowCopy(boolean);`
Delete	`setMagicDelete(boolean);`
Delete All	`setAllowDeleteAllRows(boolean);`
Update All	`setUpdateable(boolean);`

Also, all of these methods can be turned on or off all at once with a call to the following method.

```
setAllRowFunctions(boolean);
```

Other HTMLTable Attributes

HTMLTable Selection Mode

The HTML code for the results (RowCollection) may be modifiable, meaning you might need to be able to select a Row or Rows to insert, edit, delete, or perform other functionality. There are three different selection types, which are all public static constants on HTMLTable.

1. **MULTIPLESELECTION**: Allows the user to select multiple records using check boxes.

2. **SINGLESELECTION**: Allows the user to only select one record using radio buttons.

3. **NOSELECTION**: Doesn’t allow the user to select any records.

To change the selection mode of the generated HTMLTable, use the following method passing in one of the HTMLTable constants.

```
setSelectionType(int);
```

HTMLTable style sheet

A style sheet can be added to the HTMLTable to allow result tables to have somewhat of a custom look and feel.

Generate HTML Code For RowCollection (Results)

To generate HTML code for a RowCollection, after the HTMLTable object is created and set up properly, call the following method:

```
generateTable(request, response, RowCollection, List);
```

This method will generate and return HTML code to display a table of results from the given RowCollection. The List passed in as a parameter contains the column names in the RowCollection to generate HTML code for. If List is 'null' or empty, then all of the RowCollection's columns are used for generation.

16.4.2. HTMLField

To allow for ease of displaying a Field's value in HTML, the HTMLField object has a bunch of generate methods to which can be used in different scenarios. When generating an HTMLTable, HTMLFields are automatically generated. If, however, you have a custom JSP that has Row details, you may want to display certain Fields in a certain way. This is where creating a separate HTMLField object to generate HTML code for a Field can come in handy.

Creating an HTMLField

There are two ways to create an HTMLField object. With or without specifying the Field to generate HTML code. If you don't specify the Field in the constructor, then it must be set before auto generating any HTML code. The latter method can be useful in that you only need to create one instance of an HTMLField, and then just change its Field object before generating HTML code. The following two constructors can be used to create an HTMLField.

```
HTMLField htmlField = new HTMLField();
HTMLField htmlField = new HTMLField(Field);
```

Changing the HTMLField's Field

To get or set the HTMLField's object use the following methods on the HTMLField object.

```
getField();
setField(Field);
```

Methods to Generate HTML Code

All of the following methods can be called to generate different output for the Field in HTML.

Generate Field's value as text so it is uneditable.

```
generateDisplayValue(request, response)
```

Generate Field's value as input field. Both methods allow for the HTML input to have additionally supplied attributes. The second method allows for the size of the input to be set.

```
generateInput(String, request, response)
generateInput(int, String, request, response)
```

Generate Field's label (Field's external name).

```
generateLabel(request, response)
```

There are also convenience methods that do combinations of the above methods. For example, generateLabelAndField(request, response), generates the Field's label and display value. See DataEngine JavaDocs for the rest of the convenience methods.

Note: Calling one of HTMLField's generateInput() methods will automatically generate the proper HTML element depending upon the Field and its attributes. For example, if a Field has possible values, then the HTMLField will generate a pick list. If a field is a IBoolean field, it will generate a checkbox.

16.4.3. SimpleHTMLSelect

This object can be used to generate HTML list or pick list. Rather than coding all the HTML manually, you can just pass in a List, String[], or RowCollection and all the HTML code for the pick list will be automatically generated.

Creating a SimpleHTMLSelect

There are two SimpleHTMLSelect constructor methods. One takes in a String id to be used for the name of the HTML pick list.

```
SimpleHTMLSelect pickList = new SimpleHTMLSelect();
SimpleHTMLSelect pickList = new SimpleHTMLSelect(String);
```

If a String id is not specified in the constructor, it can be set with the following method.

```
setId(String);
```

Note: if 'null' or empty string are passed as a parameter for the SimpleHTMLSelect's id, then the HTML list or pick list generated will not have a name and therefore its selected value cannot be extracted.

List or Pick List

If you want a list to be displayed rather than a pick list when generating HTML, the SimpleHTMLSelect's size should be set to the number of items you want shown at once in the list. The following method can be called on a SimpleHTMLSelect object to set the size of the list.

```
setSize(int)
```

Note: any value less than or equal to one will generate a pick list.

Changing Generated List to Multiple Selection

A generated list from the SimpleHTMLSelect can be either single or multiple selection. To set the selection mode of the list, call the following method on the SimpleHTMLSelect object passing in one of two constants of HTMLSelect (SINGLE_SELECTION or MULTIPLE_SELECTION).

```
setSelectionMode(int).
```

Note: pick lists cannot be multi selectable.

Methods to Generate HTML

To generate a list or pick list from a String[], use one of the following methods. The first String[] in each method are the values for each option in the list or pick list. The String and String[] attribute following the first String[] are to denote what values should be selected. In the last two methods, the first String is to allow setting attributes for the HTML select.

```
generate(String[], String)
generate(String[], String[])
generate(String, String[], String)
generate(String, String[], String[])
```

To generate a list or pick list from a List of objects, use one of the following methods. The first String is to allow the setting of additional attributes on the HTML select. The first List is the values for each option in the HTML select. And the last parameter is to denote the options to be selected.

```
generate(String, List, List)
generate(String, List, String)
```

To generate a list or pick list from a RowCollection, use the following method. The first String is for the list or pick lists' attributes. The second String is the name of the column in each Row of the RowCollection containing the value for the option. The third String is the name of the column in each Row of the RowCollection containing the display value for the option. And the fourth String is the value to be selected.

```
generate(String, String, String, String, RowCollection, request, response)
```

17. Logging with Log4J

Exception handling and logging is an important part of the DataEngine. If there is a problem, we need to have a place to look to see what went wrong. Currently most applications complain about the performance of logging exceptions and messages. Log4J presents a logging mechanism that claims it performs very fast, in fact as fast as System.out.print() in some cases.

Here is the data you will need in your class declaration.

```
import org.apache.log4j.*;
....
public final String className = XXXClass.class.getName();
static Category cat = Category.getInstance(className);
```

Please name these variables with the same name for consistency.

Log4J provides 5 different logging levels in which you will use in different situations throughout the code. Here they are.

- Debug – This is the most verbose and should be used to output messages such as stating a variable's value or other situations when debugging.
- Info – Use info if you want to print out some info about the current status of the application
- Warn – Use this if a problem has or might occur.
- Error – Used when an error or exception occurs, but the application can continue.
- Fatal – This means we have problems. Print out fatal statements in catch blocks when an exception is thrown.

Here are some examples of logging messages.

```
cat.debug("The value of {amount} should not be null.  Amount = " + amount);
cat.info("Successfully connected to System.");
cat.warn("Could not close the output stream.");
cat.error("Exception in finally clause, but we are going to try to continue.");
cat.fatal("Exception trying to run query.");
```

There are a few tips and techniques acquired from the Log4J website. The biggest point to make here is to ensure we are not creating new String Object through concatenation on log points. Look at the following line of code.

```
cat.info("Successfully connected to " + systemName + " with UserId = " + userId + "
   and password = " + password);
```

If logging is turned off or to a level that does not print out info log points, then by having this line of code in our application, we would be creating many String Objects through concatenation unnecessarily. Instead do this.

```
if (cat.isEnabledForPriority(Priority.INFO)){
   cat.info("Successfully connected to " + systemName + " with UserId = " + userId
      + "and password = " + password);
}
```

By doing this, we would only concat the strings if logging was turned on at the INFO level. Speaking of turning logging on and off, this is done from the DataEngine Admin Servlet. This can be accessed by clicking on the link to the left of the page when logged on.

18. PlanetJ Helpers

All Helper and Descriptor classes are generic to any application and can be used to convert any Object to any type of file.

18.1. CSVHelper

This class will assist in creating CSV files. Although it's called CSVHelper (Comma Separated Variable), the delimiter may be any 'char' value.

Currently this is a magic function (See Magic Request CSV/Excel).
RowCollection.toCSV uses CSVHelper (See Generating a CSV file from a RowCollection).
Row.toCSV uses CSVHelper (See Generating a CSV file from a Row).
(See Generating a Microsoft Word file from a RowCollection)
(See DOCHelper)

18.2. DOCHelper

This class will assist in creating DOC .doc Microsoft Word files.

Currently this is a magic function (See Magic Request Microsoft Word).
RowCollection.toDOC uses DOCHelper (See Generating a Microsoft Word file from a RowCollection).
Row.toDOC uses DOCHelper (See Generating a Microsoft Word file from a Row).
(See CSVHelper)

18.3. FDFHelper

This class will assist in creating FDF files. FDF files are opened with Adobe Acrobat or Acrobat Reader. The data in the FDF will be plugged into the corresponding form field in the PDF. This is normally used along with a PDF template to plug the FDF data into.

RowCollection.toFDF uses FDFHelper (See Generating an FDF file from a RowCollection).
Row.toFDF uses FDFHelper (See Generating an FDF file from a Row)

18.4. PDFHelper

This class will assist in creating PDF files.

Currently this is a magic function (See Magic Request PDF).
RowCollection.toPDF uses PDFHelper (See Generating a PDF file from a RowCollection).
Row.toPDF uses PDFHelper (See Generating a PDF file from a Row).

18.5. XMLHelper

This class will assist in creating XML files.

Currently this is a magic function (See Magic Request XML).
RowCollection.toXML uses XMLHelper (See Generating a XML file from a RowCollection).
Row.toXML uses XMLHelper (See Generating a CSV file from a Row).

18.6. QIFHelper

This class will assist in creating QIF files which can be imported into Quicken.

In order to generate a QIF file, create a QIFFileDescriptor that defines attributes about a QIF file.

```
QIFFileDescriptor fileDescriptor = QIFHelper.newQIFFileDescriptor();;

// If you want to write the QIF data out to a file, set a file name
// property in the descriptor.  If this file name is set, then the
// QIF data will be output to the file.
fileDescriptor.setFileName("C:\\Temp\\QifTest.qif");
```

After your QIFFileDescriptor is all set up, you must define a QIFTransaction. In this example I am going to build a QIFBankTransaction which is a subclass of QIFTransation.

```
QIFBankTransaction bankTrans = new QIFBankTransaction();

bankTrans.setPayee("A-1 Sanitation");
bankTrans.setDate(new java.util.Date());
bankTrans.setCategory("Utilities:Garbage");
bankTrans.setNumber("Print Check");
bankTrans.setTotal(-45.99);
bankTrans.setMemo("1/1/2002 to 3/31/2002, Acct#123456789, Bill#:987654321");
```

Generate a QIF file by passing in bankTrans, and fileDescriptor

```
QIFHelper.singleton().generateQIFFile(bankTrans, fileDescriptor);
```

19. Transactions

When a connection is created, its auto commit function is set to false. This is to enable multiple transactions to occur before committing to the database. For example, say there were a series of inserts, updates, and/or deletes that needed to take place. And if one of them should happen, all of them should. This would allow a rollback to occur if one of them failed. If they were all successful, then a commit could be applied.

Appendices

1. Setting Up WOW With IBM's RAD 6.0

Follow the steps below to setup a WOW dynamic web project:

- Switch to the Java Perspective.
- From the left navigation (package explorer), right click and select to create a new dynamic web project.
- Leave the settings (including JSP settings) as their defaults. The project should be assigned the Websphere 6.0 test server.
- You should not have to change the project's properties (no need to add external jars to the Java Build Path)
- Copy in the following (if you prefer, you can use drag and drop or copy and paste, instead of the import feature):
- Copy any existing custom Java source (if applicable) into the web project's source file (default is Src), unless you have an external Java project.
- Copy resource folders (*i.e.* dataengine, wow, and user.xxx (if applicable)) from Tomcat project webapp folder (*ex.* wow64).
- Copy the web.xml and WOW license file (*xxx*.lic) from Tomcat WEB-INF folder and place them in the same WEB-INF folder in your web project.
- Copy the jars from your Tomcat lib folder to the web project lib folder.
- Click on the server tab at the bottom. Right click and choose "Add/Remove Projects" (same as publish). Add the default EAR project (created automatically when your web project was created) by clicking to move it to the right side. Click OK or Apply. You should see files getting copied to the test server.
- Start the test server. You should see startup messages for the WOW application.