



WorkSpace

Developer Guide

Version 7.3

Jacada Work Space Developer Guide

Copyright © 2000-2016, Jacada Ltd. All rights reserved.

Notwithstanding the copyright, this documentation contains trade secrets of Jacada; use of this documentation is subject to the terms and conditions of the software license agreement between the authorized user and its licensor. No part of this publication may be reproduced, translated, stored in a retrieval system, or transmitted in any form by any means, including but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of Jacada. Jacada reserves the right to revise this documentation and to make changes in content from time to time without obligation on the part of Jacada to provide notification of such revision or change.

Jacada® is a registered trademark of Jacada Ltd. “Jacada WinFuse,” “Jacada HostFuse,” and “Jacada Work Space” are trademarks of Jacada Ltd. All other company and product names may be registered trademarks or trademarks of the respective companies with which they are associated.

Table of Contents

Chapter 1: Introduction	1
How This Guide is Organized.....	2
Chapter 2: What is Jacada WorkSpace	5
Overview.....	5
Major Product Components.....	6
Development Environment.....	7
Menu Bar.....	8
Chapter 3: First Steps	12
Setting Up the Environment.....	12
Jacada WorkSpace Settings.....	13
Creating a Project.....	16
Creating an Enterprise Application Project.....	20
Testing a Project.....	21
Setting Up the Session Cookie Name.....	22
Chapter 4: Developing Web Content for a Jacada WorkSpace Solution	24
Overview.....	24
Configuring the Menus Displayed in Jacada WorkSpace.....	26
Specifying Predefined Favorites.....	28

JavaScript Functions Availability to Use With Menus	30
Configuring the Toolbar in Jacada WorkSpace.....	30
Configuring Buttons for the Toolbar	31
Defining the Button Icons in the Toolbar	37
Toolbar-Related Development.....	38
Enabling/Disabling Features in Jacada WorkSpace.....	42
Overriding Core Functionality.....	43
Specifying the Layout of Jacada WorkSpace.....	44
Working in Context Mode.....	45
Configuring the Screen Pop Tab.....	47
Determining the Jacada WorkSpace Layout Programmatically.....	49
Customizing the CTI Bar.....	50
Customizing Date and Time Displays.....	57
Specifying a Theme for Jacada WorkSpace.....	57
Specifying the List of Themes Available to the Agent.....	58
Managing Tabs in the Jacada WorkSpace Work Area.....	60
Determining the Default Tab.....	60
Developing Multiple Level Tabs	60
Assigning Keyboard Shortcuts in Jacada WorkSpace.....	61
Using Jacada Specific JavaScript Functions	62
Using the \$() Function.....	62
Using the \$F() Function	62
Using the \$D() Function.....	62
Using the \$W() Function.....	63
Functions to Use For Client Network Configuration Information.....	63
Functions to Use With Menu Items.....	64

Functions to Use With Tabs.....	64
Functions to Use With Nesting.....	66
Modifying Jacada WorkSpace JavaScript Methods	68
Developing WorkSpace UI Components.....	69
Debugging a Project.....	71
Chapter 5: Main Portlets of Jacada WorkSpace	75
The Agent Disposition Portlet.....	75
Development Infrastructure.....	77
Example: Incorporating Listeners in the Project.....	83
Example: Disabling Automatic Disposition in the Project.....	85
The Agent Tasks Portlet.....	87
Development Infrastructure.....	89
The Document Search Portlet.....	99
Development Infrastructure.....	101
The Help-On-Hand Portlet.....	107
Development Infrastructure.....	108
The Smart Pad Portlet.....	108
Working with the Smart Pad	110
Setting Up the Smart Pad Portlet.....	113
Transferring Information When Transferring a Call.....	124
Chapter 6: Creating Dashboards Using the Dynamic Views Designer	128
Installation of Dynamic Views	129
Accessing the Dynamic Views Designer.....	129
Dynamic Views Developer Interface.....	130
Menu Bar.....	131

Dashboards Panel.....	135
Resources Panel.....	137
Creating a New Dashboard.....	138
Adding a Data Source Widget.....	140
Customization Options.....	141
Deploying a Dashboard in Jacada WorkSpace.....	160
Functional Widgets.....	161
Grouped Widgets	162
Jacada Widgets.....	163
Logger Widget.....	164
Push Widget.....	165
Auditor Widget.....	166
Update Properties Widget.....	167
Coordinating Dashboards Among Multiple Developers.....	175
Creating a New Widget and Making It Available.....	175
Programming a Widget Repository.....	181
Code Snippets	181
Creating a Custom Action.....	191
Creating Data Sources	192
Creating a Data Source Using the URL DataSource Wizard.....	194
Creating a WorkSpace-related Data Source.....	200
Working With Input Forms	206
Creating a WorkSpace-related Input Form.....	209
Creating a New Input Form Widget.....	216
Managing the Layout of an Input Form Widget.....	222
Managing Environment Variables for Dynamic Views.....	258

Customizing Dashboards Programatically	263
Chapter 7: Developing a Timeline in a Portlet or Tab	269
Timeline Components	269
Defining a Timeline.....	277
Methods Available for Timeline Development.....	279
Client Processing Methods	279
Server Processing Methods	283
Timeline Processing.....	286
Event Filtering.....	287
The Event Information Bubble.....	287
Chapter 8: Customizing Server-side Functionality	288
Overview.....	288
Defining Variables for Use During the Project Life Cycle.....	289
Setting Up the Project Variables Infrastructure.....	290
Defining and Using Project Variables	290
Generating Project Features.....	294
Server-side Components.....	297
Client-side Components.....	297
Updating a Feature.....	297
Manually Creating a New Custom Manager.....	299
Injecting the Custom Manager.....	300
Creating a System Manager Extension.....	302
Writing Manager Code.....	303
Chapter 9: Personalizing What the Agent Can See and Do	306

Controlling the Agent's Personalization Options.....	306
Chapter 10: Pushing Information to the Agent	308
Overview.....	308
Publishing Messages.....	310
Publishing When the HttpSession is Available	310
Publishing When an HttpSession is not Available.....	312
Subscribing to Receive Messages.....	313
Client Side Configuration.....	314
Push Example.....	314
Chapter 11: Integration with a BPM System	319
Overview.....	319
Integrating with BPM.....	321
Customizing the Add Task.....	322
Bean Definition.....	323
Chapter 12: Integration with Jacada Interaction Server	328
Designing Interactions for Workspace Integration.....	330
Chapter 13: Integration with JIA Services	331
Chapter 14: Jacada Interaction Manager Integration	334
Overview.....	334
Administration Needed for Script Integration.....	335
Jacada Management Console Pages Used with Script Integration.....	336
Project Development.....	338
Extending Jacada Workspace Handling of Call Operations.....	339
Messages.....	344

Chapter 15: Integrating a Jacada HostFuse Project in Jacada WorkSpace	346
Chapter 16: Exposing Web Service Operations	351
Predefined Web Service Operations.....	351
Adding Custom Web Service Operations	352
Chapter 17: Computer Telephony Integration	354
Developing CTI Functionality.....	355
Setting Up the Call Mode of an Application: Inbound or Outbound.....	356
Setting Up Transfer and Consultation.....	356
Handling Direct-Dial Calls	360
Logging Into CTI Via the Phone	361
Configuring Jacada WorkSpace to Answer Incoming Calls	361
Using CTI Functionality Within a Portlet or Call-Handling Script.....	366
Configuring Disposition Script Behavior.....	370
Enabling Call Disposition Before Ending the Call.....	371
Disabling CTI Buttons.....	372
Managing Jacada WorkSpace Indicators	372
Overriding the Default CTI Behavior.....	374
Obtaining Information About Calls	380
Attaching Data to a Call	384
Configuring the Dial List.....	385
CTI Integration with External Applications.....	391
Developing Avaya one-X Agent Specific CTI Functionality	393
Avaya one-X Integration: Project Files.....	394
Request/Response-related Methods.....	395

Notification-related Methods.....	397
Developing Cisco Specific CTI Functionality.....	399
Overriding Default WorkSpace CTI Behavior in Cisco Environment.....	400
Obtaining Call Information Using Attached Data.....	402
Setting Up a List of Reasons for the Busy State (Aux Codes).....	404
Configuring Auto-Answer Functionality Per Call.....	405
Configuring Logging for Cisco.....	406
Supporting Multiple Sites	407
Login Issues and Handling CTI Login Errors.....	408
Working When CTI Connectivity is Lost.....	409
Configuring Transfer and Consultation.....	410
Configuring Cold Transfer.....	410
Configuring Busy State Reasons and Display.....	411
Developing Genesys Specific CTI Functionality	413
Setting Up a List of Reasons for the Busy State.....	414
Retrieving the Agent Skill Set from Genesys.....	416
Support for Transfer Connect.....	416
Support for Custom Item.....	417
Attaching Data to a Call or Custom Item.....	427
CTI Integration with External Applications.....	428
Filtering Genesys CTI Events.....	428
Chapter 18: Working With the Supervisor View	432
The Dashboard Tab.....	432
Using the Agent Task Manager.....	434
Adding a Group Task.....	435

Adding an Agent Task.....	436
Reassigning a Task.....	437
Customizing the Supervisor View.....	438
Chapter 19: Messaging Capabilities	445
Message Center Methods	445
Instant Messaging Methods.....	448
Customizing Date and Time Display.....	449
Chat Session History.....	450
Chapter 20: Logging Capabilities	451
Producing Log Files.....	451
Jacada WorkSpace Log Configuration File.....	452
Logger Names and Appender Additivity.....	453
Log Files.....	454
Log Configuration File Settings.....	454
Logging Information in a Project.....	459
Adding Logging to a Page.....	459
Excluding Sensitive Information From the Log	460
Creating a Session Dump.....	460
Adding Logging to Java Code	460
Specifying the Error Message to Write to the Log.....	462
Supported Appenders	463
Session-Related Logging.....	464
The Session Dump.....	464
Nested Application Logging	467

Chapter 21: Authentication and Authorization	469
Overview.....	469
Predefined Authorization in Jacada WorkSpace.....	471
Setting Up Authorization in Jacada WorkSpace.....	471
Authorizing the Display of JSP Pages.....	472
Methods Available for Use with Authorization.....	473
Server Side Methods.....	473
Client-Side Methods.....	477
Chapter 22: Event Auditing	481
Auditing Data Model.....	481
Session Auditing Data Model.....	482
Application Auditing Data Model.....	483
Set of Events for Audit.....	486
Developing Auditing.....	490
Auditing of the Agent Session Context.....	492
Auditing of the Interaction Context.....	493
Auditing of the Process Context.....	495
Auditing of the Action Context.....	496
Audit Code Examples.....	497
Chapter 23: Failover Support	500
Enable Session Replication in Project Code.....	500
Data Recovery with Spring.....	501
Data Recovery with Beehive.....	502
Re-establishing a JSF Script After a Failure.....	503

Controlling the Impact of Infrastructure Failures	503
Chapter 24: Localization	505
Setting Up Localization.....	505
Localization Functions.....	506
Chapter 25: Preparing a Project for Production	509
Deploying to an IBM WebSphere/Oracle WebLogic Application Server.....	509
Deploying to an Apache Tomcat Server.....	510

Introduction

Jacada WorkSpace is a contact center solution that provides a single user interface to the mission-critical applications and tools required by the call center representative, or agent, for effective customer interactions. With a unified service desktop, agents can shorten calls, improve customer interactions, be trained more rapidly and provide a consistent experience to customers. Jacada WorkSpace enables the business to concentrate on the customer interaction and process rather than the navigation of desktop applications.

Jacada WorkSpace enables creating unique WorkSpaces, which are Web-based, thin-client unified desktops. A WorkSpace provides the single user interface to the mission-critical applications and tools required by the call center representative, or agent.

This guide describes how to create a Jacada WorkSpace solution. The guide is intended for developers creating Jacada WorkSpaces for a call center. Developers should have a knowledge of the following:

- Java programming
- HTML, JSP programming
- Apache Tomcat
- IBM WebSphere application server
- Oracle WebLogic application server

How This Guide is Organized

This guide is organized as follows:

	Chapter Title	Description
2	What is Jacada WorkSpace	A high level overview of Jacada WorkSpace.
3	First Steps	In the Jacada WorkSpace development environment environment you can develop and test Jacada WorkSpace projects. This chapter introduces you to key terms and elements in the Jacada WorkSpace development environment.
4	Developing Web Content for a Jacada WorkSpace Solution	The various procedures involved in developing the look and feel of Jacada WorkSpace.
5	Main Portlets of Jacada WorkSpace	How to use the main portlets supplied by Jacada WorkSpace.
6	Creating Dashboards Using the Dynamic Views Designer	How to use the Dynamic Views Designer in the development environment.
7	Developing a Timeline in a Portlet or Tab	How to develop a timeline portlet or tab.
8	Customizing Server-side Functionality	How to extend the Jacada WorkSpace server-side functionality.
9	Personalizing What the Agent Can See and Do	How to personalize a Jacada WorkSpace solution for agents.

	Chapter Title	Description
10	Pushing Information to the Agent	How to push information to Jacada WorkSpace so that the agent receives information in real-time, without having to refresh Jacada WorkSpace.
11	Integration with a BPM System	How to integrate an external Business Process Management (BPM) system to add tasks to users defined in the target BPM system.
12	Integration with Jacada Interaction Server	How to integrate self service Interactions in a Jacada WorkSpace project.
13	Integration with JIA Services	How to invoke JIA services from WorkSpace JavaScript code.
14	Jacada Interaction Manager Integration	Different ways of displaying a Jacada Interaction manager script in Jacada WorkSpace.
15	Integrating a Jacada HostFuse Project in Jacada WorkSpace	How to integrate a Jacada HostFuse service in a Jacada WorkSpace project.
16	Exposing Web Service Operations	How to expose Web service operations from Jacada WorkSpace.
17	Computer Telephony Integration (CTI)	How to configure Jacada WorkSpace to use CTI, and developing features or using methods specific to particular CTI providers.
18	Working With the Supervisor View	How to work with and customize the Supervisor View in Jacada WorkSpace.
19	Messaging Capabilities	The messaging center used by a supervisor and the instant messaging portlets available to both supervisors and agents.

	Chapter Title	Description
20	Logging Capabilities	The Jacada WorkSpace logging mechanism.
21	Authentication and Authorization	Incorporate the privilege attribute in project code to enable user authorization of both Jacada WorkSpace features and user interface components. Describes the available server-side and client-side methods for obtaining and processing authorization-related information. Authorization of the users' assigned privileges controls both agent and supervisor use of Jacada WorkSpace features and user interface components.
22	Event Auditing	How to set up auditing.
23	Failover Support	How to set up failover.
24	Localization	How to set up language support.
25	Preparing a Project for Production	How to prepare the project for production.

2

What is Jacada WorkSpace

The Jacada WorkSpace contact center solution provides a single user interface to the mission-critical applications and tools required by the call center representative, or agent, for effective customer interactions. With a unified service desktop, agents can shorten calls, improve customer interactions, be trained more rapidly and provide a consistent experience to customers. Jacada WorkSpace enables the business to concentrate on the customer interaction and process rather than the navigation of the desktop applications.

The following topics include:

- [Overview](#)
- [Major Product Components](#)
- [Development Environment](#)

Overview

Jacada WorkSpace is a unified service desktop presenting all business applications, contact center tools, tasks and channels of communication within one console.

Jacada WorkSpace offers the following:

- Automated logon to all applications and tools.
- Integrated softphone and computer telephony integration (CTI) support.
- Applications managed inside tabs.
- Task-based user interface with assisted navigation – step-by-step task navigation ensures rapid access to the right applications for a given task or process. Within a session, the agent has access to a number of scripts, which provide links to the relevant applications as well as the conversation required between the agent and the caller. Each script is customized to the specific reason for the call. The agent can easily navigate between various scripts as the conversation with the caller changes during a call, providing flexibility and enabling the agent to respond

quickly to the callers needs.

- Audit trails and compliance monitoring.
- Business analyst view of tasks and flows.
- Real-time alerts and messaging – enabling fully customizable alerts to communicate critical information about customers, up-sell/cross-sell offers, or corporate information.
- Copy-and-paste between screens and applications.

Jacada WorkSpace is completely modular, enabling your organization to configure the interface with the features you need. The resulting WorkSpace developed using Jacada WorkSpace is designed as a thin client application, delivered to the user using Microsoft Internet Explorer.

Major Product Components

Jacada WorkSpace includes the following:

Development Environment – A development platform tailored to enable the creation of Jacada WorkSpace solutions.

Note: The information in this guide focuses on the Jacada WorkSpace options. This guide does not aim to provide comprehensive documentation of the IDE. For fuller information about using the IDE, refer to the Eclipse documentation.

Runtime Environment – The runtime environment includes the following components:

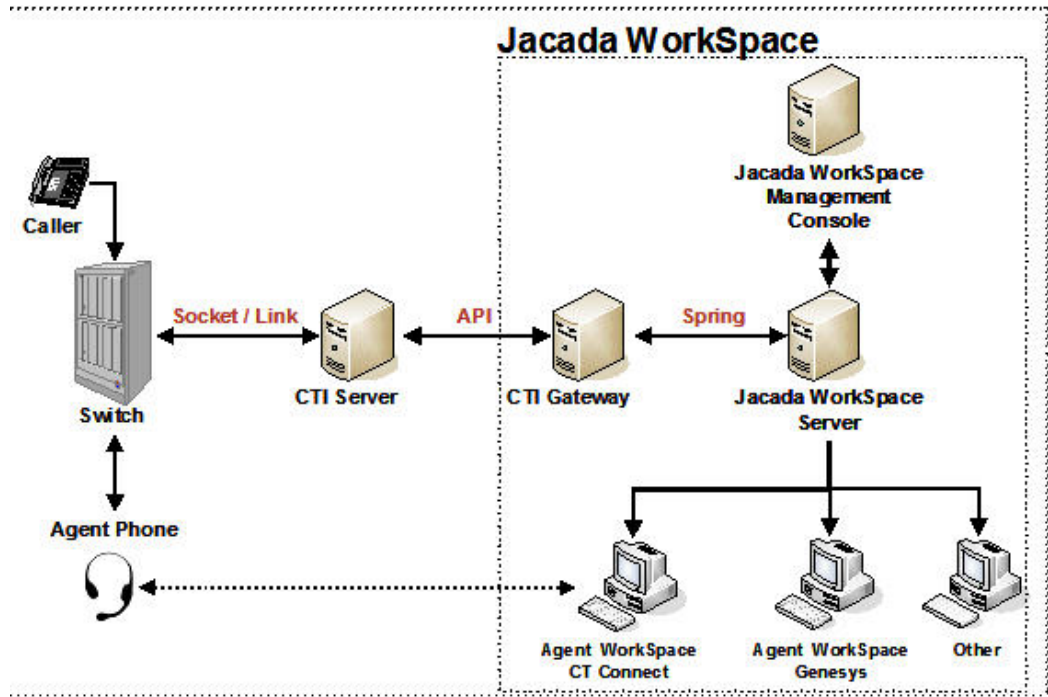
Jacada WorkSpace Server – A J2EE Enterprise Application Archive (EAR) file deployed on an application server.

Jacada WorkSpace – A browser-based unified desktop for agents, providing a single point of access to all the applications and tools required by the agent.

Jacada Management Console – A browser-based console, providing administration and monitoring functionality for Jacada WorkSpace during runtime, such as configuring the database and logs and monitoring the status of sessions.

CTI Gateway – A gateway that enables interfacing Jacada WorkSpace with call telephony (CTI) software.

The following figure shows a call center using Jacada WorkSpace in a runtime environment:

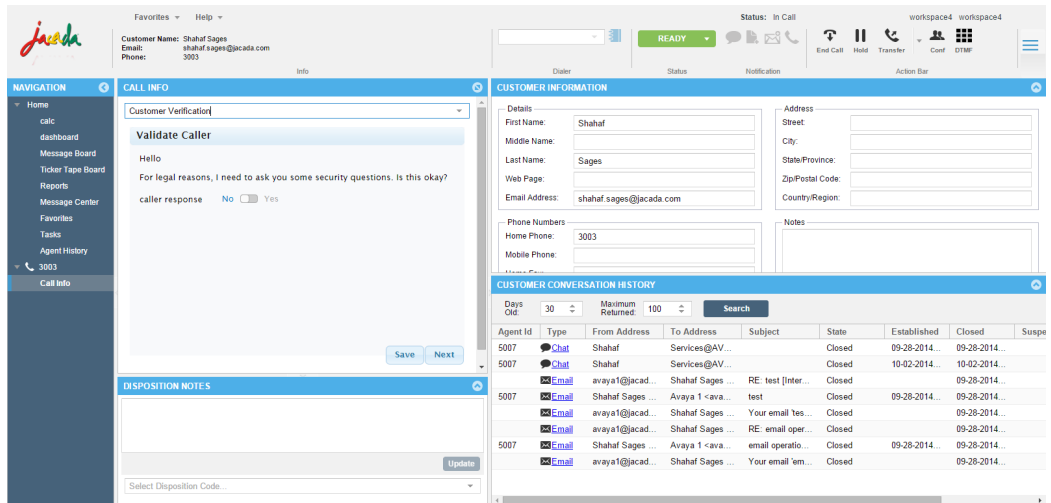


Development Environment

Jacada WorkSpace is a development environment for developing Jacada WorkSpace projects. Jacada WorkSpace is an extension of the application server development environment.

The development environment is based on a Jacada WorkSpace template, which includes a default Jacada WorkSpace solution and the tools to change or add to it.

A completed Jacada WorkSpace project is deployed as a WAR or EAR file. The resulting Jacada WorkSpace is a browser-based unified desktop for agents. Jacada WorkSpace is composed of a number of areas including areas for computer telephony, CTI, portlets and a central workarea. Portlets can be placed on all four sides of the central workarea: top, bottom, left and right.



Menu Bar

The menu bar items displayed depend on the access rights of the agent. The default items are:

Favorites – Web favorites that can be opened by the agent. You can define Web favorites under this menu as well as enable the agent to add to the list of Web favorites.

Help – Provides *About Jacada WorkSpace* help information.

The right side of the menu bar includes the agent name and the logout link.

Toolbar

The toolbar shares the center area of the menu bar and makes specific types of Jacada WorkSpace capabilities available as buttons. The types available as toolbar buttons are: opening a portlet and running JavaScript code. An opened portlet displays on top of Jacada WorkSpace and not in one of the designated portlet areas. Jacada WorkSpace includes a toolbar with the following buttons:

Button	Description
	Opens the <i>Chat</i> portlet.
	Opens the <i>Agent Tasks</i> portlet.
	Opens the <i>Document Search</i> portlet.
	Opens the <i>Help-On-Hand</i> portlet.
	Opens the <i>Agent Disposition</i> portlet.

The toolbar is located to the right of the menus in Jacada WorkSpace.

Call-handling Bar, Progress Bar and Ticker Tape

Call-handling Bar

The call-handling bar includes statistics about the calls handled by the agent and buttons to handle calls, such as putting the caller on-hold or transferring a call.

Progress Bar

When enabled, the progress bar allows the agent to view the progress of business calls and the call handling time, to monitor performance against the SLA (Service Level Agreement). This progress bar changes color from green to red as the call progresses, and at the same time the CTI timers also change color. In addition, when the SLA bar reaches its maximum time, the SLA bar begins to flash.

By default, the progress bar is disabled.

Ticker Tape

Messages created in the *Message Center* tab can be displayed on the Ticker Tape. The ticker tape properties are defined when adding or editing messages, as described in the *Jacada WorkSpace Administrator Guide*. These include defining the message priority, color, the display date and time

range and more. In addition, the *Ticker Tape Board* tab displays RSS feeds that are defined in the Jacada Management Console.

Portlet Areas

The portlet areas are used to display different information and functionality to an agent. The following table lists the portlets that Jacada WorkSpace provides as part of each new project:

Agent Disposition – A notes pane for the agent to enter ad-hoc text and a disposition code area for the agent to select disposition codes during the agent's call wrap-up status. This portlet is available from the toolbar.

Agent Tasks – Enables agents to create and manage their own personal list of tasks. This portlet is available from the toolbar.

Chat – Enables agents and supervisors, logged in to a Jacada WorkSpace session, to initiate chat sessions and send interactive messages to one another. This portlet is available from the toolbar.

Dial Pad – A telephone dial pad and text box that the agent uses to place outbound calls, dependent on the CTI configuration at the site. If CTI is not available or during a call, the portlet content is disabled. This portlet is available from the toolbar.

Document Search – Enables the agent to conduct a text search within documents in specified document sources. This portlet is available from the toolbar.

Guidance – Enables the agent to access self service Interactions from the Jacada Interaction Server.

Help On Hand – Conducts background, text searches, using the project's configured document sources, to proactively provide relevant search results to the agent. This portlet is available from the toolbar.

Favorites – Links to the Web site favorites that are also available in the *Favorites* menu and tab. Any favorites added in the *Favorites* menu item are automatically added to the tab and portlet content. This portlet can be added to the layout by editing the default layout.

Sample Portlet – A sample portlet, which you can use as the starting point when developing new portlets. This portlet is displayed by default in Jacada WorkSpace. This portlet can be added to the layout by editing the default layout.

Smart Pad – A clipboard to store and view information during a call. Shortcut keys enable cut and paste of information between the Smart Pad and other applications. This portlet can be added to the layout by editing the default layout.

Workarea

The workarea is composed of tabs. By default, the workarea displays left-justified in Jacada WorkSpace. You can add tabs to the workarea. For example, add tabs that include the applications the agent uses.

The following tabs are provided by default:

Message Board – Displays messages sent by the supervisor. The Message Board is refreshed after the agent logs in or after finishing a call. The messages displayed on the *Message Board* are the same as the messages displayed in the *Welcome* page after logging in.

TickerTape Board – Displays the full message of a message on the ticker tape. In addition, the *Ticker Tape Board* tab displays RSS feeds that are defined in the Jacada Management Console. The content of this message is replaced each time a ticker tape message or RSS feed is selected. For more information, see the *Jacada WorkSpace Administrator Guide*.

Customer Profile – Details of the customer calling in. Some of the information in this tab, such as the email address, can be changed directly in the tab.

Message Center– Enables a supervisor to send messages to agents. Regular agents do not see this tab.

Chat History – Enables supervisors to view the statistical data regarding agent chat sessions. The information in this tab includes the chat initiator and recipient, the start of the chat session and the length of the chat session.

Favorites – Links to the Web site favorites that are also available in the *Favorites* menu and portlet. Any favorites added in the *Favorites* menu item are automatically added to the tab and portlet content.

Multi-Tabs – A sample multiple level tab, providing an additional tab level containing two sub-tabs. Use this tab as the starting point to develop new, multiple level tabs.

Tasks – Displays a list of tasks and enables the agent to manage them. The task list is retrieved from Jacada Flows.

3

First Steps

The following topics describe the initial steps required to create a WorkSpace:

- [Setting up the Environment](#)
- [Creating a Project](#)
- [Testing a Project](#)

Setting Up the Environment

After installing Jacada WorkSpace you can start developing a project. You deploy the project to an application server configured for Jacada WorkSpace. This application server is defined when you run the *Server Setup Wizard*, either at the end of the installation, as described in the Jacada WorkSpace Installation Guide for the Developer, or at any time before testing the project.

After finishing the Jacada WorkSpace installation, you run the Jacada Server Setup wizard to create the application server and the database schema used by Jacada WorkSpace.

The following procedure explains how to create an Apache Tomcat server.

To create a server and a database schema:

1. Select *Start > Programs > Jacada > Jacada WorkSpace <version> > Server Setup Wizard*.
2. In the *Select Application Server* dialog, select *Apache Tomcat 7.0*.
3. In the *Tomcat Details Panel* dialog, provide both the Apache Tomcat server home folder and the server's HTTP port to use with Jacada WorkSpace.
4. In the *Database Details* dialog, select the required database from the dropdown list and dependent on the database, enter the other details.
5. In the *Schema Details* dialog, specify the name and password you want for the schema.

Note: After the Tomcat server has been created, you can manage the server using the Tomcat Management Console with a default user name of admin and a password of admin. You can change these defaults in the `<tomcat_server_install_dir>\conf\tomcat-users.xml` file, where `tomcat_server_install_dir` is the folder you specified when creating the Tomcat server, by default C:\Program Files\Apache Tomcat 7.0.

When using Windows 7, the **Program Compatibility Assistant** dialogue box may appear at the end of the installation. Select the **This Program installed correctly** option.

Jacada WorkSpace Settings

There are a number of settings that must be configured on the agent desktop in order to use Jacada WorkSpace.

Internet Explorer Settings

Jacada WorkSpace requires a medium font size: In Internet Explorer, select *View > Text Size > Medium* to view Jacada WorkSpace correctly.

For Jacada WorkSpace to function with Internet Explorer, the browser settings must be configured to work with Jacada. When using Internet Explorer version 8, the Jacada WorkSpace solution must open in the first tab.

Make sure the Internet Explorer settings for Jacada WorkSpace, in *Tools > Internet Options*, are as follows:

General Tab

1. In the *Temporary Internet files* section, when using Internet Explorer 6, or in the *Browsing history* section, when using either Internet Explorer 7 or Internet Explorer 8, click the *Settings* button.
2. For the setting *Check for newer versions of stored pages*, select the *Automatically* radio button.

Security Tab

- Select the relevant Web content zone, and click the *Custom Level* button.

The *Security Settings* dialog box opens. The following are the required settings:

Security Setting Category	Security Setting Name	Required Setting Status
ActiveX controls and plug-ins	<i>Binary and script behaviors</i>	Enable
	<i>Initialize and script ActiveX controls not marked as safe</i>	Enable
	<i>Run Activex controls and Plug-ins</i>	Enable
	<i>Script ActiveX controls marked safe for scripting</i>	Enable
Miscellaneous	<i>Access data sources across domains</i>	Enable Required for integration with Jacada Integration and Automation distributed DTA.
	<i>Allow script-initiated windows without size or position constraints</i>	Enable Required when using Internet Explorer 8 and any certified Microsoft operating system runs the agent desktop.

- Additional security settings are required when the operating system on which the Jacada WorkSpace solution runs is either Vista or the Windows 7. In Vista and Windows 7, Microsoft uses a User Account Control (UAC) technology and security infrastructure to improve security by limiting application software to standard user privileges, unless specified otherwise, using integrity levels.

There are three integrity levels: high, medium and low. The higher the integrity level, the greater the access privileges. A high integrity level allows administrative privileges as opposed to a low integrity level, which enables only generic execute access. A high integrity level can run a high,

medium and low executable, while a medium integrity level can run a medium or low executable. When applications need to talk with each other, they must have the same integrity level. For example, medium integrity level applications can talk to a low integrity level application. However, the low integrity level application cannot respond.

Note: Applications usually spawn other application with their integrity levels.

This security mechanism in Vista and Windows 7 is used when protected mode is On. However, legacy applications are often required to run with protected mode Off. If this is the case, automation under Vista and Windows 7 functions normally and there is no need to change any settings.

Add the URL to log on to the Jacada WorkSpace solution to the *Trusted sites* zone: select the *Trusted sites* zone, and click the *Sites* button. Make sure that the “Enable Protected Mode” option is set to the default, which is off.

Advanced Tab

Under the *HTTP 1.1 settings*, make sure that the *Use Http 1.1* option is checked.

Note: Jacada recommends not using add-ons in Internet Explorer. For example, using the Skype add-on causes a memory leak every time a web page is reloaded. This memory leak also occurs when the Web page is loaded in a tab in Jacada WorkSpace.

If you must use an add-on, check the impact on Jacada WorkSpace and if necessary nest the Internet Explorer application with the Web page in the tab and not just the frame.

When using Internet Explorer 8, make sure that the browser is set for 100%.

Note: When running Jacada WorkSpace with Internet Explorer 7 and 8, IE should be launched using the no-merge option (IE7) or the noframemerging option (IE8), so that any Jacada WorkSpace tab will run as a separate process. For more information, refer to

[http://msdn.microsoft.com/en-us/library/ee330728\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ee330728(v=vs.85).aspx).

Restart the browser.

Search Portlet Settings

To ensure that results from search portlets are displayed in the correct destination, for example, in a

new browser window, the PCs in which Jacada WorkSpace runs must have the *Windows Folder Option File Type* setting enabled with the *Browse in same window* option.

To enable the File Type setting option for each applicable document file type:

1. In the Jacada WorkSpace host machine, right-click the *Start* menu.
2. In the menu, select the *Explore* option.
3. In the displayed *Start Menu* window, select *Tools > Folder Options... > File Types* tab.
4. In the *File Types* tab, select from the *Registered file types* pane a file type for those source documents that are available to be searched by Jacada WorkSpace. For example, *Microsoft Office Word Document*.
5. In the *Details for <selected_file_type> extension* pane, click the *Advanced* button.
6. In the displayed *Edit File Type* dialog, check the *Browse in same window* option.

Note: This option is not available for every document file type. For example, the *Adobe Acrobat Document* file type. Such a file type does not require any option enabled.

7. Click *OK*.
8. Repeat Step 4 through Step 7, per applicable document file type of the documents available for search by both the *Document Search* and the *Help On Hand* portlets.
9. Click *Close*.

Creating a Project

You create a Jacada WorkSpace project in the IDE, which is deployed in an application server as Jacada WorkSpace.

To create a Jacada WorkSpace project:

1. Start the development environment via *Start > Programs > Jacada > Jacada Work Space <version> > Eclipse IDE*.
If this is the first time you start the development environment, the *Workspace Launcher* dialog box is displayed. In this dialog box, enter the workspace location and click *OK*.

Note: A workspace is the location where the Jacada WorkSpace project is stored. It is recommended to specify a new location the first time you start Eclipse after installing Jacada WorkSpace.

The development environment is displayed.

2. Close the *Welcome* page, if it is displayed.

3. Click *File > New > Project*.

The *New Project* dialog opens.

3. Select *Jacada WorkSpace Project* and click *Next*.

The *New Jacada WorkSpace Project* dialog opens.

4. In the *Project name* textbox, Enter the name of the project.

The name cannot include the following characters: #, *, ", :

5. Select a target runtime, using ONE of the following options:

- From the *Target runtime* dropdown list, select Apache Tomcat v7.0. Options in the EAR Membership frame are disabled.
Proceed to Step 7.
- From the *Target runtime* dropdown list, select Oracle WebLogic server. Then, proceed to Step 6.
- From the *Target runtime* dropdown list, select IBM WebSphere v8.5.5. Then, proceed to Step 6.

Note: The target runtime is the runtime where the project will be deployed for production. You don't need to have defined a WebSphere profile/WebLogic domain to specify a target runtime, you just need to specify the installation directory for the relevant runtime. If a runtime is not listed, click *New*, select the runtime environment, click *Next* and then specify where the runtime is installed.

6. (WebSphere/WebLogic only)

In the **EAR Membership** frame, configure the following parameters:

- Select the *Add project to an EAR* checkbox. (While this checkbox is selected, Apache Tomcat does not appear as an option in the *Target runtime* dropdown list.)

- If desired, change the name of the default EAR project name.

Note: You can include more than one project in an EAR, by replacing the default EAR with the name of an existing EAR defined for another project. The *EAR project name* combo box lists all existing Enterprise Application projects.

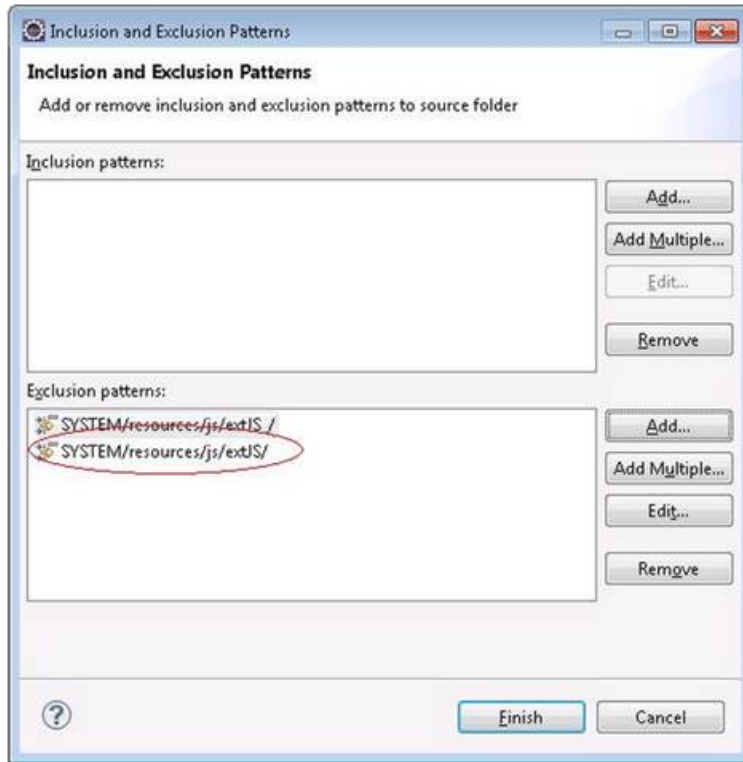
7. Click *Finish*.

The following are created:

<project_name> – Files and folders relating to the Jacada WorkSpace project.

<project_nameEAR> – The EAR for the project (WebSphere/WebLogic only).

8. After creating the project, perform the following steps to prevent the first build from failing. (This failure is due to JavaScript validation exception in Eclipse.)
- a. Right-click on the project, and select *Properties*.
 - b. Go to *Builders*, and uncheck the *JavaScript Validator* builder.
 - c. Go to *JavaScript > Include Path*. From the *Source* tab, select *Excluded*. Then, at the right side of the dialog, click *Edit*.
 - d. In the *Exclusion patterns* frame, add the following:
 - SYSTEM/resources/js/extJS_
 - SYSTEM/resources/js/extJS



e. Click OK.

The Jacada WorkSpace project includes the following:

- **Java Resources** – Contains Java resources to extend and customize the built-in functionality.
- **Web Content** – Contains the definitions for the Web content of Jacada WorkSpace.

The project is configured as a default Jacada WorkSpace project and can be tested to see the basic default features included in the project.

Creating an Enterprise Application Project

If a Jacada WorkSpace project was created without an Enterprise Application project, it may be added later on using the EAR Application Project wizard.

To create an Enterprise Application project:

1. From the *New Project* dialog, expand the *Java EE* directory and select *Enterprise Application Project*. Then, click *Next*.

The *EAR Application Project* dialog appears.

2. In the *Project name* textbox, enter a relevant name for the project.
3. From the *Target runtime* dropdown list, select *WebSphere Application Server version 8.5.5*.

Note: The target runtime is the runtime where the project will be deployed for production. You don't need to have defined a WebSphere profile to specify a target runtime, you just need to specify the installation directory for the relevant runtime. If a runtime is not listed, click *New*, select the runtime environment, click *Next* and then specify where the runtime is installed.

4. In the *Configuration* frame, click *Modify*.

The *Project Facets* dialog opens.

5. Select the following project facets:

- *EAR*
- *Jacada Application*

Set the EAR version to *1.4*.

Then, click *OK*.

6. At the bottom of the *EAR Application Project* dialog, click *Next*.

The *Enterprise Application* dialog appears.

7. In the *Java EE module dependencies* frame, select the Jacada WorkSpace project.
8. Click *Finish*.

Testing a Project

To test the Jacada WorkSpace solution for the project, you must start a server and then run the project on the server.

During development it is recommended to use the Apache Tomcat server to test the project.

Note: For details about deploying the project to a WebSphere application server, refer to [Preparing a Project for Production](#).

To define and start an Apache Tomcat application server:

1. Right-click within the *Servers* tab area, and select *New > Server*.
The *Define a New Server* dialog is displayed.
2. Drill-down to the server, *Apache Tomcat v7.0 Server*, select this server and click *Finish*.

Note: The Java version required for Tomcat and the Eclipse IDE are different. The correct versions should have been defined as part of the pre-installation of Jacada WorkSpace and Eclipse set up to recognize the correct versions to use. For details, refer to the Jacada WorkSpace Installation Guide for the Developer (Pre-Installation).

A *Servers* entry is added to the Project Explorer view.

3. Double-click the new server to open the server overview and make the following changes:
 - a. Under *Server Locations*, select the Use Tomcat installation (take control of Tomcat installation) option.
 - b. Under *Timeouts*, specify 120 for the start time and 150 for the stop time.
4. Add the `JACADA_OPTS` variable to the VM arguments:
 - a. At the lower right corner of the *VM arguments* frame, click *Variables*.
 - b. From the list of variables, select `JACADA_OPTS`, and then click OK.

The *VM arguments* frame is populated with the selected variable.

5. Apply the changes.
6. Start the server: Right-click the server and select *Start*.

To test a project:

1. In the IDE, in *Window > Web Browser* select *Internet Explorer*.
2. Right-click the project and select *Jacada Work Space > Configure Project for Tomcat Runtime*.

Note: If the project is already configured for Tomcat, the entry will be *Configure Project for WebSphere Runtime*. Before continuing, verify that an [Enterprise Application project](#) has been created.

- 3 Right-click the project and select *Run As > Run on Server*.

Note: Right-click the project and not the project EAR file.

The *Run on Server* dialog is displayed.

4. Leave the defaults and click *Finish*.
The project is published to the server and started in the browser. If prompted, to restart the server, restart it.
5. In the *Login* page that opens, type *workspace* and *workspace* as the user name and password, respectively, and click *Login*.
6. In the *Welcome* page, click *Done*. Jacada WorkSpace opens.

Setting Up the Session Cookie Name

The default name for the cookie used to track a Jacada WorkSpace session is `jsessionId`. If desired, you can change the cookie name at the project level, to avoid cookie sharing between Jacada WorkSpace and other nested web applications.

Note: Changing the cookie name is currently supported only for WebLogic users.

Changing the cookie name involves opening **weblogic.xml** inside an application, uncommenting the `session-descriptor` tag, and replacing `@NEW_COOKIE@` in the subtag `cookie-name` with the required name. If there are other parameters under `session-descriptor` that are not required in the project, leave them commented.

```
<session-descriptor>
  <persistent-store-type>replicated_if_clustered</persistent-store-type>
  <cookie-name>@NEW_COOKIE@</cookie-name>
</session-descriptor>
```

When a cookie name other than the default (`jsessionid`) is used, verify that the following additional required configurations are done:

- Defining the `sessionCookieName` setting in the *Global Settings* page of the Jacada Management Console.
- Modifying relevant file settings when configuring the Apache web server.

For more information about these configurations, refer to the *Jacada WorkSpace Administrator Guide*.

4

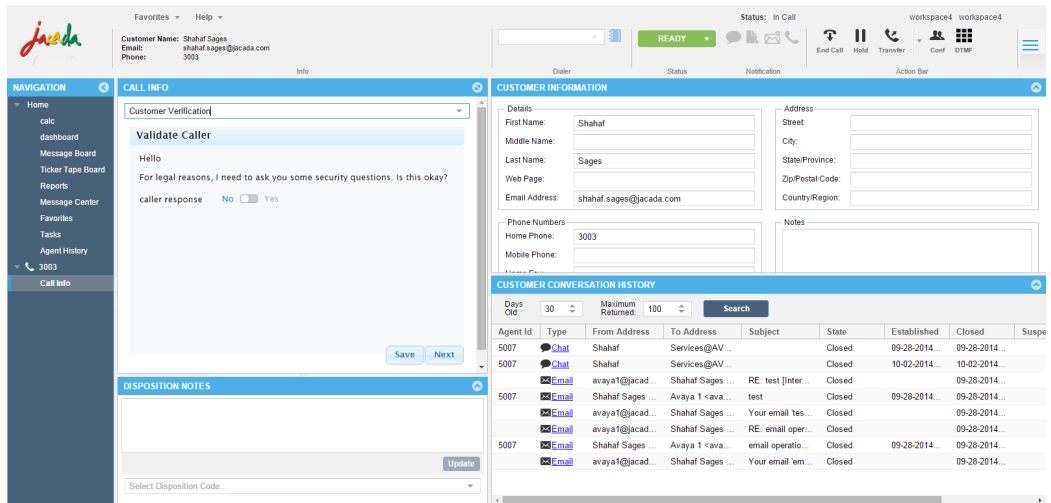
Developing Web Content for a Jacada WorkSpace Solution

The following topics describe how to set up the content of a Jacada WorkSpace solution:

- [Overview](#)
- [Configuring the Menus Displayed in Jacada WorkSpace](#)
- [Configuring the Toolbar in Jacada WorkSpace](#)
- [Enabling/Disabling Features in Jacada WorkSpace](#)
- [Overriding Core Functionality](#)
- [Specifying the Layout of Jacada WorkSpace](#)
- [Specifying a Theme for Jacada WorkSpace](#)
- [Managing Tabs in the Jacada WorkSpace Work Area](#)
- [Assigning Keyboard Shortcuts in Jacada WorkSpace](#)
- [Using Jacada Specific JavaScript Functions](#)
- [Debugging a Project](#)

Overview

Jacada WorkSpace is the deployed Jacada WorkSpace project solution. Jacada WorkSpace is composed of a number of areas including areas for computer telephony, CTI, portlets and a central work area. Portlets can be placed on all four sides of the work area: top, bottom, left and right.



The work area is composed of tabs. You can add tabs to the work area. For example, add tabs that include the applications that your agents use, one application per tab.

Four portlet areas surround the work area. Portlets are framed areas that contain and display specific content. Jacada WorkSpace provides certain portlets and you can also develop your own portlets. Place portlets in the four portlet areas positioned at the top, bottom, left and right of the work area. You can specify the initial, display size of each portlet in a portlet area as a percentage of that area.

Jacada WorkSpace is designed in the IDE using the following components:

Java Resources – Contains Java resources to extend and customize the built-in functionality.

Web Content – Contains the definitions for the Web content of Jacada WorkSpace. This folder includes the following main folders:

SYSTEM – Contains system files. These files are for internal use and must not be modified.

USER – Contains files that you use to modify and customize the Jacada WorkSpace project to suit the customer business requirements. Information in files in the USER folder is appended to information in corresponding files contained in the SYSTEM folder. When information in the USER folder files is in disagreement with SYSTEM folder information the USER folder file information overrides the SYSTEM folder information.

Folders in WebContent/USER Folder

The following folders are located in the WebContent/USER folder:

portlets – The folder where portlets supplied as part of any project are defined and where new portlets should be saved.

resources > css – Contains the override.css file used to override default style sheet definitions in the SYSTEM/resources folder.

resources > images – Contains image files used in a Jacada WorkSpace solution. You can replace these images and add images to this folder. Added images override images defined in the SYSTEM/resources folder.

resources > js – Contains the override.js file used to override JavaScript functions in the SYSTEM/resources folder.

resources > jsp – Contains the include file which is part of the call-flow script template.

resources > jspf – Contains the include files for jspf.

resources > layouts – Contains Jacada WorkSpace layouts.

resources > projectVariables – Variables for use in the project so that different values can be used with the project in different environments, without the need to recompile the project.

resources > properties – Provides the validation.properties file. Use this file to add new and modify exiting message classes for validation messages.

resources > resourceBundles – Contains localization files.

resources > themes – Contains the themes available by default for Jacada WorkSpace. Define new themes to use in Jacada WorkSpace in this folder.

resources/faq.xml – The xml file used for the default FAQ.

resources/toolbar.xml – The file used to configure the toolbar in Jacada WorkSpace.

error.jspf – Use this file to customize the error page presentation for standard errors.

errorPage.jspf – Use this file to customize the error page presentation for severe errors that cause Jacada WorkSpace to crash.

Configuring the Menus Displayed in Jacada WorkSpace

By default, Jacada WorkSpace provides infrastructure for a menu bar with the following items:

Favorites – Enables the agent to manage a list of Web sites that are frequently accessed.

Help – Displays information about Jacada WorkSpace.

The Jacada WorkSpace menu is configured in **WebContent/USER/resources/menus.xml**, which contains a section for each menu bar item, and defines the menu items within each menu. For example:

```
<menus>
  <item ...>
    <item name="application.menuBar.menu.edit"

class="SYSTEM.global.conditions.PersonalizationE
nabled"
      privilege="MenuBarItemEdit">
      <item
accessKey="application.menuBar.accessKey.layout"
      name="application.menuBar.item.layout"
      value="javascript:toggleAddContent();"
privilege="LayoutManager"/>
      ...
      <separator/>
      ...
    </item>
    <item ...>
  </menus>
```

where:

<item> – The menu bar item. An **<item>** element can be included under a parent **<item>** to provide items under the menu.

name – The attribute that references the relevant key in the **USER/resources/resourceBundles/UI_Texts_<language>.properties** file and defines the menu or menu item name. Menu and menu item names are localized.

accessKey – The attribute that references the relevant key in the **USER/resources/resourceBundles/UI_Texts_<language>.properties** file and defines the shortcut key. Menu item shortcut keys are localized. When defining shortcut keys, make sure the shortcut keys do not interfere with the functionality and the shortcuts of the external applications used by the agent. For a list of Windows shortcuts, go to <http://support.microsoft.com/kb/126449>.

Shortcut keys are activated using **ALT+<key>**, as is standard in Windows.

class – The full package name of the `conditions` class. In the above example, the `PersonalizationEnabled` class determines whether any personalization options are available to the agent.

role (deprecated) – The role required to authorize use of the menu or menu item. For details about

roles, see [Authentication and Authorization](#).

privilege – The privilege required to authorize use of the menu or menu item. For details about privileges, see [Authentication and Authorization](#).

value – The JavaScript to run when a menu item is selected.

separator – Creates a separating line between menu items.

To change the contents of the Jacada Workspace menu:

1. In the IDE, open **WebContent/USER/resources/menus.xml**.
2. Modify the **menus.xml** file, as necessary, using a pointer one of the following for the name attribute value:
For a new menu – `application.menuBar.menu.<string>`
For a new menu item – `application.menuBar.item.<string>`
3. Open the **USER/resources/resourceBundles/UI_Texts_<language>.properties** file and add the localization string defined in the menu with the value to be displayed:
For a new menu – `application.menuBar.menu.<string>=<menu_name>`
For a new menu item – `application.menuBar.item.<string>=<menu_item_name>`
For example, `application.menuBar.item.calculator=Calculator`

Specifying Predefined Favorites

The **menus.xml** file contains a section for the *Favorites* menu. By default, there are no predefined favorites in the menu, and the agent can add favorites as necessary while working. However, you can specify predefined favorites to appear in the menu, by adding items to the *Favorites* menu in the **menus.xml** file.

In the following example, the “MyCompany” Web site is added to the list of favorites. In this case, the global setting `dynamicallyChangeFavoritesTabTitle` (in the Jacada Management Console) is set to `false`.

```

<item
name="application.javascript.menuBar.menu.favorites"
    privilege="MenuBarItemFavorites">
    <item
accessKey="application.menuBar.accessKey.manageFavorites"
name="application.menuBar.item.manageFavorites"
    class="SYSTEM.global.conditions.PersonalizationEnabled"
value="javascript:openFavoritesDialog();"/>
    <item
name="application.menuBar.item.accessMyCompanyWebsite"
    value="javascript:openFavorite('http://www.MyCompany.com', true);"
    privilege="PortletTools"/>
</item>

```

The `true` parameter, which is supplied to the JavaScript `openFavorite` method, causes the supplied URL to be opened in a new browser window.

When the global setting `dynamicallyChangeFavoritesTabTitle` is set to `true`, and a project defines favorites in **menus.xml**, they must be in the following form:

```

<item
name="application.javascript.menu.favorite.Jacada" value="javascript:openFavorite('http://jacada.com', false
'application.javascript.javascript.menu.favorite.Jacada');"/>

```

where localization keys must have client form (`application.javascript...`).

Note: The JavaScript function `openFavorite` has a third argument, which must be the same as the name attribute of the `item` tag. The first two arguments are the same as the ones presented before.

JavaScript Functions Availability to Use With Menus

A number of JavaScript functions are used with menu definitions. For details of the functions that you can use, refer to [Functions to Use With Menu Items](#).

Configuring the Toolbar in Jacada WorkSpace

The toolbar, which is located on the right side of the Jacada WorkSpace screen, provides the agent access to necessary functionality when needed, without requiring a dedicated area in Jacada WorkSpace. The toolbar is delivered with the following options:

Button	Description
	Opens the <i>Chat</i> portlet.
	Opens the <i>Agent Tasks</i> portlet.
	Opens the <i>Document Search</i> portlet.
	Opens the <i>Help On Hand</i> portlet.
	Opens the <i>Agent Disposition</i> portlet.

Toolbar buttons provide the agent access to necessary functionality as and when needed, without requiring a dedicated area in Jacada WorkSpace. When a toolbar button opens a portlet, Jacada WorkSpace uses the popup window format to display the portlet.

When relevant to a particular toolbar button, Jacada WorkSpace informs the agent that the toolbar button has received one or more notifications from Jacada WorkSpace, in the following ways:

- By changing its background color and flashing several times.
- If the portlet, associated with the toolbar button, is currently open, the portlet flashes several times.

Pressing any toolbar button in the notified state clears its notifications.

An example of the toolbar button in the notified state is the *Chat* button. When the agent has been sent chat messages from other agents or supervisors and the Chat portlet is currently closed, the *Chat* toolbar button indicates that it was notified of chat messages awaiting the agent by changing its background color and flashing.

Opening a Portlet

When a toolbar button opens a portlet, Jacada WorkSpace uses the popup window format to display the portlet.

- The first time the portlet is opened during a Jacada WorkSpace session, it appears in the center of the workarea. On subsequent openings, the portlet appears in most recent position, it was located, prior to being closed.
- An opened portlet can be always visible and prevent agent access to any other areas within Jacada WorkSpace until the portlet is closed. or it can be *initially* visible and the agent can access other areas within Jacada WorkSpace and collapse the portlet.

For more information about configuring new toolbar buttons, refer to [Configuring Buttons for the Toolbar](#).

Configuring Buttons for the Toolbar

You can configure additional project-specific buttons for the toolbar. Customized toolbar buttons can supply the agent with either of the following actions:

- Open a portlet
- Run JavaScript code

Add project-specific buttons to the toolbar by configuring them in the file **WebContent/USER/resources/toolbar.xml**. Buttons that open a portlet use the `portlet` tag. Buttons that run JavaScript code use the `javascript` tag.

Attributes Used with portlet or javascript Tag:

Tag	Attribute	Description
Portlet	availableCondition	Optional. Id of a custom Spring bean that extends the class <code>AbstractCondition</code> and implements the <code>isValid()</code> method. The method's result determines whether or not to make the portlet available in the toolbar.
	enablePortletDragging	Optional. Enables dragging the opened portlet. Default value is <code>true</code> .
	enablePortletScrolling	Optional. Enables scrollbars (vertical, horizontal or both) with the portlet. If scrolling is disabled, any content that does not fit within the dimensions of the portlet's display, is not displayed. Default value is <code>false</code> .
	height	Optional. Height of the opened portlet in pixels. Default is 600.
	id	Required. Id of the toolbar button.

Tag	Attribute	Description
	loadOnLogin	Optional. The portlet is loaded when an agent logs in. The logon is longer but the first access to the portlet is quicker. If the attribute is disabled (<i>false</i>), the portlet is loaded with the first access to the portlet via the toolbar. Default value is <i>false</i> .
	modal	Optional. Makes the portlet modal. There is no access to other areas in Jacada WorkSpace until the portlet is closed. If the attribute is disabled (<i>false</i>), the user can access other areas in Jacada WorkSpace. Default value is <i>true</i> .
	portletTitle	Required. Title of the portlet; displayed in the open portlet. This attribute is localized. For information about localization, see Setting Up Localization .
	portletURL	Required. The relative project path to the portlet that opens when the toolbar button is pressed. For example, USER/portlets/myForexConverter/ForexConverter.jsp or SYSTEM/portlets/dialPad/index.jsp .
	privilege	Optional. The privilege that must be assigned to agents in order for them to access the portlet via the toolbar button.

Tag	Attribute	Description
	<code>reloadContentOnNextOpen</code>	Optional. The portlet is reloaded each time the portlet is re-opened. If the attribute is disabled, Jacada WorkSpace caches the portlet's existing content and in the same session displays this content when the portlet is re-opened. Default value is <code>false</code>. Note: This attribute does not apply to the <i>Chat</i> portlet.
	<code>role</code> (deprecated)	–
	<code>styleClass</code>	Required. The icon name of the button, as defined in the project's USER/resources/themes style.css files. For example, <code>foreignExchangeConverterIcon</code>.
	<code>tooltip</code>	Optional. Tooltip text displayed for the button. This attribute is localized. For information about localization, see Setting Up Localization. If not supplied, the <code>portletTitle</code> is used as the tooltip.
	<code>width</code>	Optional. Width of the opened portlet in pixels. Default is 800.

Tag	Attribute	Description
JavaScript	<code>availableCondition</code>	Optional. Id of a custom Spring bean that extends the class <code>AbstractCondition</code> and implements the <code>isValid()</code> method. The method's result determines whether or not to make the JavaScript available in the toolbar.
	<code>id</code>	Required. Id of the toolbar button.
	<code>javascriptCode</code>	Required. Assigned JavaScript code that is run when the toolbar button is pressed. JavaScript code runs at the main window level (root index.jsp) of Jacada WorkSpace.
	<code>privilege</code>	Optional. The privilege that must be assigned to agents in order for them to run the javascript via the toolbar button.
	<code>role (deprecated)</code>	–
	<code>styleClass</code>	Required. The icon name of the button, as defined in the project's USER/resources/themes style.css files. For example, <code>agentCallHandlingStatsFunctionIcon</code> .

Tag	Attribute	Description
	tooltip	Required. Tooltip text displayed for the button. This attribute is localized. For information about localization, see Setting Up Localization.

Remove a button from the toolbar, including any of the Jacada-provided ones, by deleting its configuration from the **toolbar.xml** file.

Example of the toolbar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<toolbar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="..\..\SYSTEM\resources\xmlValidators\
                                                                    toolbar.xsd">

  <portlet id="DialPadWindow"
    portletURL="SYSTEM/portlets/dialPad/index.jsp"
    portletTitle="application.javascript.toolbar.dialPad.portletTitle"
    styleClass="dialPadIcon" tooltip="application.toolbar.dialPad.tooltip"
    height="100" width="100" loadOnLogin="false"
    reloadContentOnNextOpen="false" modal="true"
    enablePortletDragging="true" enablePortletScrolling="false"
    privilege="PortletDialPad" />
  <portlet id="myForexConverterWindow"
    portletURL="USER/portlets/myForexConverter/ForexConverter.jsp"
    portletTitle="application.javascript.toolbar.forexConverter.portletTitle"
    styleClass="foreignExchangeConverterIcon"
    tooltip="application.toolbar.forexConverter.tooltip"
    height="200" width="300" loadOnLogin="true" reloadContentOnNextOpen="true"
    modal="false" enablePortletDragging="true" enablePortletScrolling="true"
    privilege="PortletForex"
    availableCondition="agentSkillSet" />
  <javascript id="MySummarizeCallHandlingStatsFunction"
    javascriptCode="mySummarizeCallHandlingStats();"
    styleClass="myCallHandlingStatsFunctionIcon"
    tooltip="application.toolbar.mySummarizeCallHandlingStats.tooltip"
    privilege="PortletJSCode" />
</toolbar>
```

Defining the Button Icons in the Toolbar

Each button must have an icon to represent it in the toolbar button.

If your project uses the Jacada WorkSpace-provided themes, define an entry for the button icon in the provided CSS style sheets:

- **WebContent/USER/resources/themes/Jacada/style.css**
- **WebContent/USER/resources/themes/Sample/style.css**

If your project uses a custom created theme, define an entry for the button icon in the theme's **style.css** file. For information about themes and creating themes, see [Specifying a Theme for Jacada WorkSpace](#).

In the relevant **style.css** file, define the button icon in the following manner:

```
.<styleClass>{background-image: url(images/  
toolbar/<icon_image_filename>); height: 28px;  
width: 28px;}
```

Example:

```
.dialPadIcon {background-image: url(images/  
toolbar/dialpad.gif); height: 28px; width:  
28px;}
```

The following button states are available:

Hover – The button can be pressed and the agent's cursor is hovering over the button.

Notified – The button can be pressed and has received one or more notifications from Jacada WorkSpace.

Notified with Hover – The button can be pressed, has received one or more notifications from Jacada WorkSpace, and the agent's cursor is hovering over the button.

Note: If a portlet, associated with a toolbar button, is currently open, the portlet flashes several times to indicate current receipt of one or more notifications from Jacada WorkSpace.

Pressed – The button was pressed by the agent, resulting in either a portlet being opened or JavaScript code being run.

Toolbar-Related Development

The following sections provide information for developing a project with toolbar buttons.

Methods for Working with The Toolbar

Jacada WorkSpace exposes the following methods for working with the toolbar:

Side	Method	Description
Server Processin g	<pre>void notifyToolbarButton(String buttonId) throws ToolbarException</pre>	The <code>ToolbarManager</code> is a Spring bean. <code>toolbarManager</code> provides this method. Use this method to send a notification to a toolbar button identified by <code>buttonId</code>. The server can send an unlimited number of notifications to a toolbar button. All notifications are cancelled immediately upon the agent pressing the toolbar button.
	<pre>boolean isButtonInToolbar(String buttonId)</pre>	The <code>ToolbarManager</code> is a Spring bean. <code>toolbarManager</code> provides this method. Use this method to determine if the button, identified by <code>buttonId</code>, is available in the toolbar for agent use. The method returns <code>true</code> if the specified button is available for agent use. The method returns <code>false</code>, when either: • The agent is not assigned the role required to use the button, making the button not available for the agent. • The button does not exist in the toolbar.

Side	Method	Description
Client Processing	<code>\$W().notifyToolbarButton(buttonId);</code>	Use this method to send a notification to a toolbar button identified by <code>buttonId</code> . Send an unlimited number of notifications to a toolbar button. All notifications are cancelled immediately upon the agent pressing the toolbar button.
	<code>function onPortletShow(reload)</code>	Use this method to reload a toolbar portlet, (when its <code>reloadContentOnNextOpen</code> attribute is <code>true</code>). When <code>reload</code> is <code>true</code> the method reloads the toolbar portlet.

Determining Button Availability

The optional `availableCondition` attribute which is available for use in both `portlet` and `javascript` tags, references a custom Spring bean that extends the class `AbstractCondition` and implements the `isValid()` method. The result returned by the method determines whether or not to make the relevant portlet or JavaScript available in the toolbar.

To use the `availableCondition` attribute:

1. Under `src/USER`, define a new class that extends the Jacada `AbstractCondition` class.
2. In the new class, implement the `isValid()` method, based upon your custom criteria.
3. In the file `WebContent/WEB-INF/applicationContext-user.xml`, define a new Spring bean that implements an instance of the defined class.
4. In the file `WebContent/USER/resources/toolbar.xml`, in the `availableCondition` attribute of the relevant `portlet` or `javascript` tag, reference the `id` of the new Spring bean.

Development Considerations

If the project requires Jacada WorkSpace to be made aware whenever the server sends a notification to a toolbar button, the client must register with the Push handler to receive these events. Make sure that the JavaScript code of the `<client_function_name>` referenced in the push registration receives `buttonId` as an argument. Use the following method to register Jacada

WorkSpace with the Push handler:

```
Push.registerEventHandler('toolbarButtonNotification', <client_js_name>);
```

When developing portlets to be opened by toolbar buttons, do the following:

- For all URLs that are used in the portlet's .jsp file, provide the explicit path and not the relative path of the file being referenced. For example, provide the URL `<My_project_name>/USER/resources/images/myPortletImage1.gif`.
- Do not include, in the portlet's .jsp file, both the `prototype.js` and the `ExtJS` JavaScript libraries. Jacada WorkSpace project's, by default, always include these libraries.
- If the portlet is one that requires its content reloaded each time that it is opened for display (in `toolbar.xml`, portlet tag `reloadContentOnNextOpen` attribute is `true`), implement the `onPortletShow ()` method for the portlet, in the following manner:
 - The code calls the `onPortletShow` at portlet display.
 - The method accepts the reload parameter, which directs it to perform reload of the toolbar portlet (`true`) or not (`false`).
 - The method must reset all its relevant components, for example, clearing stores, loading stores and clearing fields.

The following example shows how to implement this method in the portlet's .js file:

```
function onPortletShow(reload){
    if(reload){
        custom toolbar portlet reload processing
        here;
    }
}
```

When developing a toolbar button that opens a portlet and the portlet moves between pages, the following development is required:

- In **toolbar.xml**, the `portletURL` attribute of the toolbar button must be assigned a URL that directs Jacada WorkSpace to access a *wrapper* portlet page and not the portlet itself.

The following is an example of the required `portletURL` attribute assignment, shown in **bold**, for the toolbar button, whose id is `Sample2Wrapper`:

```

<portlet id="Sample2Wrapper"
portletURL="USER/portlets/sample/sampleWrapper.jsp"
styleClass="chatIcon"
portletTitle="application.portlet.label.samplePortletTitle"
modal="false"
height="450"
width="350"
reloadContentOnNextOpen="true"
tooltip="application.portlet.label.samplePortletTitle" />

```

- In the code of the required *wrapper* portlet page, the `iframe` tag must include the following:
 - Refer to the actual portlet to open.
 - Include both `height` and `width` attribute definitions that are identical to the `height` and `width` attribute definitions assigned for the toolbar button in **toolbar.xml**.

Code Example

The following is an example of the required `iframe` tag code, shown in **bold**, in the *wrapper* portlet page, named `sampleWrapper.jsp`, which was referenced in the preceding **toolbar.xml** example:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- jsf:pagecode language="java" location="/src/pagecode/USER/portlets/
      sample/SampleWrapper.java" -%><%- /jsf:pagecode --%>
<%@page language="java" contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>
<html>
<head>
<title>sampleWrapper</title>
</head>
<body>
<iframe height="450" width="350"
src="<%=request.getContextPath()%>/USER/portlets/sample/sample2.jsp">
</iframe>
</body>
</html>

```

Enabling/Disabling Features in Jacada WorkSpace

The following features of Jacada WorkSpace can be enabled/disabled, as required:

- Toolbar
- Timeline
- Tickertape
- Supervisor View

Generally, these features are enabled/disabled using the *Feature Settings* page of the Jacada Management Console. For details, refer to the Jacada WorkSpace Administrator Guide.

The behavior defined in the Jacada Management Console can be overridden in the project code, using the `FeaturesManager` API. The exposed methods are described in the following table.

Method	Description
<pre>public void enableFeature (FeatureName feature);</pre>	Enables a specified feature. As input, use the feature that you want to enable.
<pre>public void disableFeature (FeatureName feature);</pre>	Disables a specified feature. As input, use the feature that you want to disable.
<pre>public boolean isEnabled (FeatureName feature);</pre>	This method checks whether a specific feature is enabled or disabled. Use the relevant feature as input. The method returns <code>true</code> if the specified feature is enabled, and returns <code>false</code> if it is disabled.

Code Example

The following example shows how to enable the Toolbar feature from the `login` method of the `ExtensionManager` class:

```
@GlobalAction(actionName=GlobalActionNames.LOGIN
,order=GlobalActionOrder.BEFORE)
public void login(HttpServletRequest request)
{
    ServletContext servletContext =
request.getSession().getServletContext();
    WebApplicationContext context =
WebApplicationContextUtils
.getWebApplicationContext(servletContext);
    FeaturesManager featureManager =
(FeaturesManager)
context.getBean("featuresManager");

    featureManager.enableFeature(FeatureName.Toolbar
);
}
```

For a feature to be available to an agent, both of the following requirements must be fulfilled:

- The feature is completely enabled in the project code.
- The agent has the privilege to use the relevant feature. For more information about privileges, refer to the [Jacada WorkSpace Administrator Guide](#).

Note: By default, the Agent role includes all privileges that can be enabled/disabled.

Overriding Core Functionality

The core functionality of Jacada WorkSpace can be overridden by modifying the content of the **WorkSpaceProjectExtension.js** file.

In the following example, the word *Call* is added to precede the number displayed in the Navigation Pane when a call is being handled:

```
//change voice group node name example
Ext.define('Override.ContextNavigationPanel', {
    override : 'Jacada.core.layout.ContextNavigationPanel',
    createVoiceContextGroup: function(name, id){
        name = 'Call '+name;
        this.callOverridden(arguments);
    }
});
```

Specifying the Layout of Jacada WorkSpace

The definitions of all the portlets that are available in Jacada WorkSpace are specified in the database, and can be viewed and modified in the Jacada Management Console. However, the portlets that are actually displayed to an agent are determined by the layout in use.

A layout determines the following about portlets:

- The portlets to display in Jacada WorkSpace.
- The location and size of each portlet.
- The actions that can be performed in the portlets, such as adding, removing, moving or resizing.

Jacada WorkSpace provides the following predefined layouts:

- **default** – The default layout for the agent supplied by Jacada WorkSpace.
- **sample** – A sample layout that you can use as a template for your own layout.

In a layout configuration, you can set general attributes that define the personalization options available in Jacada WorkSpace when using this layout.

A layout configuration contains definitions for each portlet area, top, bottom, left and right. Each area definition can include the height and/or width settings for the area and can list the portlets to display in the area. When an area contains more than one portlet, you must specify a value for the portlet width in the top and bottom areas and a value for the portlet height in the left and right areas. For each portlet, you can also specify whether the portlet can be removed, and whether it can be collapsed and expanded.

For a full description of layout attributes, and an explanation of how to create a layout and set the default layout, refer to the Jacada WorkSpace Administrator Guide.

Working in Context Mode

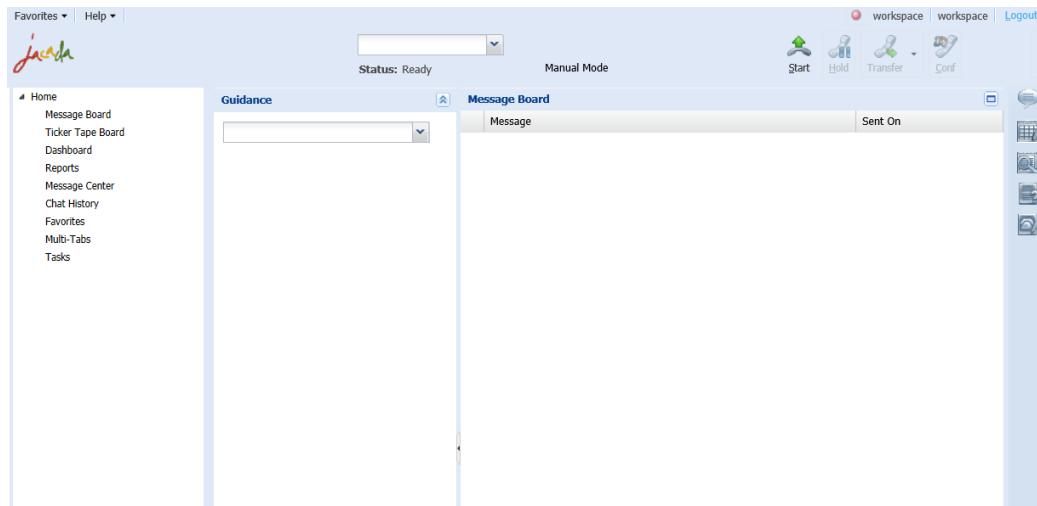
When Jacada WorkSpace is configured to run in Context mode, the layout changes automatically according to the activities in which the agent is currently engaged. The specific layouts and tabs that are displayed in a given context are defined in the Jacada Management Console. For more information, refer to the Layout Management section of the Jacada WorkSpace Administrator Guide.

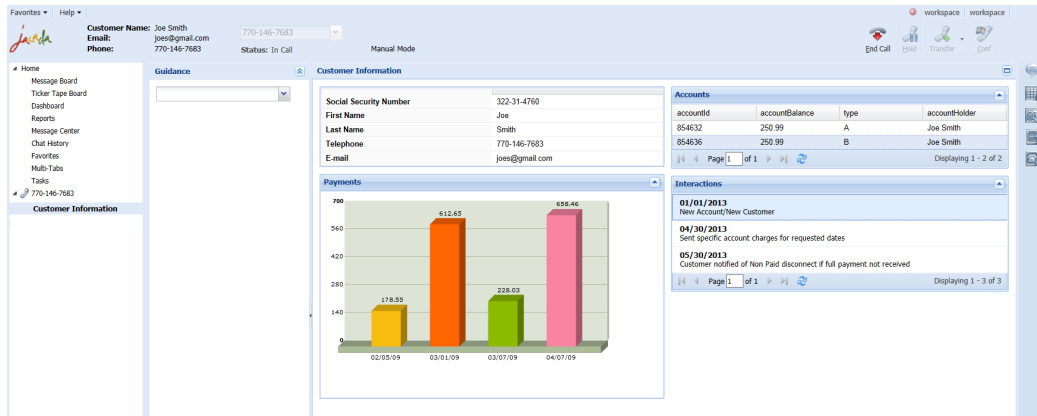
The contexts which are currently configurable are:

- **General:** A non-specific working environment. Layouts and tabs that are assigned to be displayed in the General context are always visible.
- **Voice:** The agent is working with one or more callers.
- **Work Item:** One or more work items (e.g., fax) has been generated.
- **Human task:** The agent is working on one or more tasks.

The items in the Navigation Pane (on the left side of the screen, under the Jacada logo) vary according to the context(s) in which the agent is working. The *Home* menu is always visible. Additional menus and relevant portlets appear when other contexts are activated. In addition, each context has its own Action Bar, which is a toolbar that enables agents to quickly access actions related to the context.

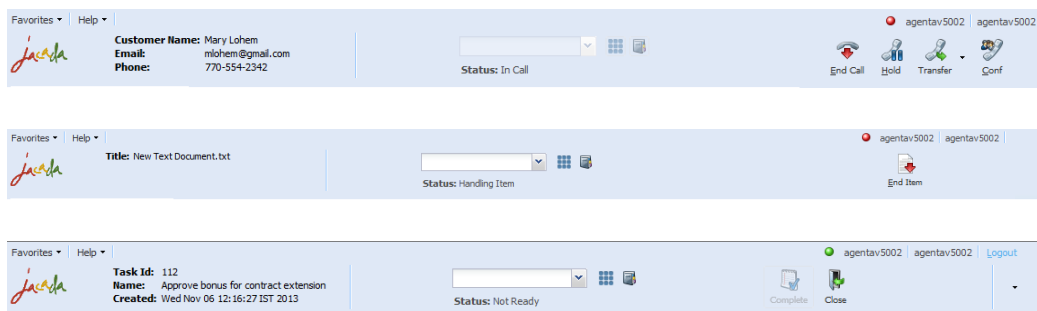
The following example shows the default General context layout, and the additional UI components that appear when the agent enters Voice context.





Action Bars

The following figures show examples of Action Bars for the Voice, Work Item and Task contexts.



Context Listeners

A Workspace portlet or tab can register to a context listener. For example, you can have an event fire upon a change of context, or upon selection of a new tab.

The following code snippets register a portlet/tab to a context listener:

```

initComponent: function(){
    var me = this;
    Ext.applyIf(me, {
        listeners : {

            contextStarted: function(type, data){
                //invoked when a new context is started
            },
            contextTabChanged: function(type, data){
                //invoked each time a new tab in the context
                navigation pane is selected
            },
            contextDataChanged: function(type, data){
                //invoked when the data of the context has changed.
                This is useful when the data may change after the context has already started.
                //Note: the developer must fire event
                with the new data, for example:
                var contextNavigation = Ext.getCmp('contextNavigation');
                contextNavigation.fireContextDataChangedEvent($W()
                .interactionType.Voice, <You New Data Object>,
                $W().activeContext.groupId);
            },

        }
    });
    me.callParent(arguments);
    Ext.getCmp('contextNavigation').registerContextListener(me);
},

```

Configuring the Screen Pop Tab

The Screen Pop is a tab that opens in Jacada WorkSpace when an agent enters Voice, Email or Chat context (according to the type of interactions selected in the Management Console). A Screen Pop tab generally is used to display customized information related to the opened context. For instance, when an agent accepts a call, the Screen Pop could show a customer profile that is retrieved based on the values passed in the tab's URL.

The URL for the tab's content is defined in the Jacada Management Console. (For more details, refer to the Layout Management section of the Jacada WorkSpace Administrator Guide.) The parameters to be returned are added to the URL as key-value pairs.

The following default parameters are included in the project code, and are replaced automatically when EMC is the provider:

- Caller name (CallerNameValue)
- Caller number (CallerNumberValue)
- Prompted Digits (PromptedDigitsValue)
- UUI (UUIValue)
- Email (EmailValue)
- DateTime (DateTimeValue)

You can override the default behavior by changing these values and/or defining additional values as required.

The development infrastructure makes available the `CustomScreenPopService` class, that exposes the following methods:

Method	Description
<code>replaceURLRequestParametersValue(ScreenPopDataDto screenPopInfo)</code>	Gets the dictionary of key-value pairs (by calling the <code>getURLRequestParametersReplacementValues</code> API), and replaces the placeholders in the URL with the actual values associated with the relevant customer.
<code>getURLRequestParametersReplacementValues(ScreenPopDataDto screenPopInfo)</code>	Gets the mapping of defined key-value pairs.

The `screenPopInfo` object contains the following values:

- `url`: The URL of the Screen Pop tab's content.
- `uniqueId`: The unique identifier of the customer interaction (the call, chat, etc.).
- `interactionType`: The type of customer interaction, according to the type of interactions selected in the Management Console (Voice, Chat or Email).

To configure the Screen Pop tab with the default implementation, use the following lines of code:

```
return super.replaceURLRequestParametersValue(screenPopInfo);  
return super.getURLRequestParametersReplacementValues(screenPopInfo);
```

Alternatively, you may build on the default implementation by adding your own pairs to the key-value dictionary, or overriding the default values. In the following example, one new parameter is added,

and the value of a default parameter is assigned a hard coded value:

```
/*Map<String, String> values = super.getURLRequestParametersReplacementValues
(screenPopInfo);
    values.put("myParam", "myValue");
    values.put(ScreenPopSupportedParameters.CALLER_NAME.value(), "Joe");
    return values;*/
```

Another option is to create your own implementation for the Screen Pop content. The following example provides a guideline for how to update the values in the key-value dictionary. The updated URL and a boolean value (indicating whether or not an update was done) are returned. If desired, you may also build your own mapping, instead of invoking the `getURLRequestParametersReplacementValues` API. However, to help ensure accuracy, it is recommended to use the method.

```
/*String url = screenPopInfo.getUrl();
    Map<String, String> values = getURLRequestParametersReplacementValues
(screenPopInfo);
    for(Entry<String, String> e: values.entrySet()){
        if(StringUtils.hasText(e.getKey())){
            url = url.replace(e.getKey(), e.getValue());
        }
    }
    return new ScreenPopUpdatedDataDto(url, true);*/
```

Determining the Jacada WorkSpace Layout Programatically

The layouts that can be displayed in Jacada WorkSpace are defined in the Jacada Management Console. (For details, refer to the [Jacada WorkSpace Administrator Guide](#).) The specific layout to be displayed to individual agents can be configured in your project code, by overriding the `prepareSession ()` method in class `Controller.java` under the `User` package.

When you are working in [Context mode](#), you need to specify the layout according to interaction type. For example:

```
getLayoutManager().setLayout(InteractionType.Voice, "sample");
```

When you are not using Context mode, the interaction type is not relevant:

```
getLayoutManager().setLayout("sample");
```

Customizing the CTI Bar

Administrators can remove the default CTI bar. In addition, you can create a customized CTI bar.

Removing the CTI Bar

Administrators can remove the CTI bar for agents that are not using a CTI provider.

To remove the CTI bar:

1. Using the Jacada Management Console, access the *Configuration > CTI > Settings > Default > MANUAL* navigation menu option.
2. In the *Profile Settings* table, define the value of the `CTIBarEnabled` parameter to `False`.

Defining a New CTI Bar

The `CTIControllerLogic.java` file contains a logic class and a controller class. When defining a new CTI bar, you can replace the following:

- Replace the logic class
- Replace the controller class
- Replace both the controller and logic classes

When replacing the controller class, you must change the `CTIBarURL` value to point to a new URL in the `CTI_SETTINGS` table. When replacing the logic class, you must register the new `CTIControllerLogic.java` file in the `applicationContext-user.xml` file, under `WebContent/WEB-INF`. Alternatively, for simpler modification cases, you can extend the existing controller and logic classes.

To define a new CTI bar:

1. In the `CTI_SETTINGS` table, define the value of the `CTIBarURL` to point to the new CTI Bar. The value can be one of the following:
 - A URL to a .jsp file, for example, `SYSTEM/portlets/CTI/index.jsp`
 - A fully qualified ExtJS class name, for example `Jacada.system.ui.cti.CTIBar`
2. Ensure the new CTI controller supports the controls used by `CTIClientAPI.js`. The controller view uses the methods from the `CTIClientAPI.js` and the `CTIClientAPI.js` sends a request to the `Controller.java` file for controlling the CTI bar.
3. Ensure the following actions are defined in the new controller:

doLogout – To perform the logout operations sent from the client side.

setTimers – To save the timers on the server side.

endDisposition – To send the disposition ended event to the state machines.

finishInit – To inform the CTI state machine that the CTI client side has finished loading and is prepared to receive information and notifications.

Note: Call this function when the client side had finished loading, so the server side can notify the client regarding the current agent status.

buttonClicked – To handle `clickButton` operations sent from the client side with the following input parameters:

- **Operation:** The operation to be performed. For example, `warmTransfer`, `consult`, or `hold`. For a complete list, refer to `CTIClientAPI.js`.
- **Dialled Number:** The dn for the operation, such as warm transfer, when relevant.

attachData – To handle the `attachData` and `deleteAttachData` operations arriving from the client side. The given key-value will be sent to be attached or deleted accordingly in the CTI switch. The input parameters of the `attachData` method include, `Operation(attachData or deleteAttachData)`, `Key` and `Value`.

4. In order to receive notification of a state change in the CTI state machine, register the `UpdateCTIClientevent` in your java script using the following function sample. In this example, `myHandler` is used to handle the object coming from the server. Replace it with your function, as required:

```
//registering myHandler method to deal with the object coming from the server
$W().Push.registerEventHandler( 'UpdateCTIClient',
this.myHandler.bind(this));
...
//the myHandler method can appear as follows:
myHandler: function (ctiClientInfo) {
    var clientStateName = ctiClientInfo.clientStateName;
    // handle busy/ready
    var busy = this.ctiClientInfo.bBusy;
    ...
}
```

Note: This is required only when replacing the CTI Controller.

The **CTIClientInfo.java** file describes the fields that are available for use, as shown below:

```
public class CTIClientInfo implements Serializable {
    public String clientStateName = ClientState.NOT_INITIALIZED;
    public String callId = "";
    public String customerId = "";
    public String superId = "";
    public String errorMessage = "";
    public String errorMessageParam = "";
    public int numberOfCalls = 0;
    public int numberOfCustomItems = 0;
    public String customerCallId = "";
    public String superCallId = "";
    public String customerState = "";
    public String superState = "";
    public String ANI = "";
    public String DNIS = "";
    public String agentName = "";
    public String extension = "";
    public boolean bBusy = true;
    public String errorType = "";

    // to know if Transfer, Consult options should be enable
    public boolean hasTransferInfo;
    public boolean hasConsultInfo;

    // set this to allow the 'recover connection' button
    public boolean allowCTIResync = false;

    public int stateVoiceIcon = ChannelState.NOT_LOGGED;
    public int stateEmailIcon = ChannelState.NOT_LOGGED;
    public int stateMultimediaIcon = ChannelState.NOT_LOGGED;
```

5. In order to control the `loadScript`, `unloadScript` and `NavigateToLink` operations, use the following methods from `CTIClientAPI.js`:

- `loadScript`
- `unloadScript`
- `navigateToLink`

Alternatively, use the code below, to register and listen to `loadScript` and `unloadScript` events arriving from the server.

```

//load call script
$W().Push.registerEventHandler( 'call_script_name',
                                this._onCallScriptName.bind(this));

//unload script
$W().Push.registerEventHandler( 'unload_script',
                                this._onUnloadScript.bind(this));

//load disposition script
$W().Push.registerEventHandler( 'loadDisposition',
                                this._onLoadDisposition.bind(this));

//handler for loading the call script
onCallScriptName: function(scriptUrl) {
    ...
    var msg = "in _onCallScriptName :"+scriptUrl;
    this.ctiAPI._debug(msg);
    var url = "USER/flows/Controller.jspf?";
    if (scriptUrl != null && scriptUrl.length > 0) {
        url = scriptUrl;
    }
    url += '&callerId=' + this.callerId;
    $W().showLogout(false);
    $W().loadScript(url);
},

//handler for unloading the script
_onUnloadScript : function() {
    ...
    $W().loadIntoIframe("MessageBoardFrame", "SYSTEM/portlets/messaging/
messageBoard.jsp");
    $W().ShowTabById('messagesTab');
    $W().unloadScript();
},

//handler for loading the disposition script
_onLoadDisposition: function(scriptUrl){
    ...
    //if navigation bar is not available, load into the script zone without the
    navigation bar
    if ($W().ScriptZone.NavigationBarFrame == null ||
        $W().ScriptZone.NavigationBarFrame.clickOnLink==null) {
        $W().loadScript(scriptUrl);} else {
        $W().navigateToLink("root.callDisposition");
    }
}
}

```

Also, see the javascript code in `CTIClient.js` representing the default CTI bar.

6. Replace the logic, the controller or both the logic and controller as required. Each of these options are described in the table below (from the simpler to the more complex option):

Option	Details
Replacing the logic Used for example, when replacing the logic class with another logic class or when extending the current logic class.	In the new CTI Controller Logic file, for example myCTIControllerLogic, perform the following: Implement the CTIControllerLogic interface or extend the default by adding the following code: <pre>public class MyControllerLogic extends SYSTEM.portlets.CTI. DefaultCTIControllerLogic { ... }</pre> Register your new CTIControllerLogic in the applicationContext-user.xml, by entering the following code: <pre><!-- Singleton bean for holding the CTIControllerLogic --> <bean id="ctiControllerLogic" class="USER.portlets.CTI.MyCTIControllerLogic"> <property name="globalAppHandler" ref="globalAppHandler"/> </bean></pre>

Option	Details
Replacing the CTI Controller Used for example, when a different technology is required, not dependent on NETUI or when extending the current controller class.	In MyCTIController class: Retrieve the <code>CTIControllerLogic</code> and delegate the methods to the default logic, by entering the following code: <pre>Public class MyCTIController { public void onEveryActionInThisClass(){ //Retrieve the ctiControllerLogic from the Spring context //by using the session WebApplicationContext context = WebApplicationContextUtils.getWebApplicationContext (getSession().getServletContext()); CTIControllerLogic ctiControllerLogic = (CTIControllerLogic) context.getBean("ctiControllerLogic"); //delegate the methods to the logic Method method = ctiControllerLogic.getClass().getMethod(action, new Class[] { HttpServletRequest.class }); method.invoke(ctiControllerLogic, new Object[] {getRequest() }); } }</pre> Or Extend the default <code>CTIController</code>, by entering the following code: <pre>public class MyController extends SYSTEM.portlets.CTI. CTIController {}</pre>
Replacing the logic and the CTI Controller	When defining a new controller class, you must enter the URL pointing to the controller in the database. When replacing the logic you must register the new controller logic in the XML. For further details on how to replace the logic and/or the controller refer to the options described above.

Customizing Date and Time Displays

Jacada WorkSpace formats all date displays and all time displays using the following properties:

- `DATE_FORMAT`
- `TIME_FORMAT`

These properties are defined in the file **WebContent/index.jsp**. Customize the display of all date information, all time information or both, by modifying the defined format for the relevant property in this file.

Jacada WorkSpace supports the following, standard JavaScript date and time formats:

Date formats—`m/d/Y`, `n/j/Y`, `n/j/y`, `m/j/y`, `n/d/y`, `m/j/Y`, `n/d/Y`, `m-d-y`, `m-d-Y`, `m/d`, `m-d`, `md`, `mdy`, `mdY`, `d`, `Y-m-d`

Time formats—`g:ia`, `g:iA`, `g:i a`, `g:i A`, `h:I`, `g:I`, `H:I`, `ga`, `ha`, `gA`, `h a`, `g a`, `g A`, `gi`, `hi`, `gia`, `hia`, `g`, `H`

For example, modifying the date and time format property definitions in `index.jsp` to be:

```
DATE_FORMAT = 'Y-m-d';  
TIME_FORMAT = 'ga';
```

results in Jacada WorkSpace displaying all date information in the format, 2009-03-05, and all time information in the format, 9pm.

To customize the date and time displays specifically associated with:

- The *Agent Tasks* portlet, see [Customizing Date and Time Display](#).
- Instant Messaging (the *Chat* portlet), see [Customizing Date and Time Display](#).
- The *Message Center*, see [Customizing Date and Time Display](#).

Specifying a Theme for Jacada WorkSpace

A theme determines the design of the Jacada WorkSpace user interface, in terms of color schemes and graphics. You can change the theme appearance by modifying any of the style sheets supplied with Jacada WorkSpace. You can also define additional themes for Jacada WorkSpace.

The themes available to an agent in Jacada WorkSpace are defined in the following file:

```
<project name>/WebContent/USER/resources/themes/themes.xml
```

You may change the default theme on the *Global Settings* page of the Jacada Management

Console (`theme` property).

To create a theme:

1. Create a new folder under `USER/resources/themes`.
2. Copy the `style.css` files from the `USER/resources/themes/NeptuneBlue` folder and paste them into the newly created folder.
3. In the new folder, change the `style.css` file to the style you want.
4. Add an `images` folder in the new folder.
5. Copy the entire contents of the `USER/resources/themes/NeptuneBlue/images` folder to the new `images` folder, including the `buttons` folder.
6. Replace any of the images that you want with a different image.

Note: Do **not** change the size of the images. If you change the name of an image, you must update the `style.css` file accordingly.

7. Modify the list of themes to include your new theme. For details, refer to [Specifying the List of Themes Available to the Agent](#).

Specifying the List of Themes Available to the Agent

The `themes.xml` file contains all the themes available in Jacada Workspace.

The `themes.xml` provided by Jacada Workspace contains both the `Jacada` and the `Sample` themes. This file also specifies the `Jacada` theme as the one to display when the agent initially logs in to a Jacada Workspace session. The content of `themes.xml` is as follows:

```
<themes xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="..\..\..\SYSTEM
\resources\xmlValidators\themes.xsd">
  <theme name="application.theme.label.default"
    directory="..\..\..\USER/resources/
themes/Jacada"
    ext_style="ext-all.css"
    default="true" />
  <theme
name="application.theme.label.sampleTheme"
    directory="..\..\..\USER/resources/
themes/Sample"
    ext_style="xtheme-gray.css"/>
</themes>
```

where:

name – The theme name.

directory – The name of the folder containing all the folders and files associated with this theme.

ext3_style – The .css resource file that applies to the portlet that uses the *Ext JS* library (version 3.x). The assigned file provides a specific theme.

ext_style – The .css resource file that applies to the portlet that uses the *Ext JS* library. The assigned file provides a specific theme.

default – The theme used as the default theme in Jacada WorkSpace.

To modify the list of available themes:

1. In the IDE, open `WebContent/USER/resources/themes/themes.xml`.
2. Add a theme definition for the new theme. For example:

```
<<theme name="neptune-blue" directory="USER/
resources/themes/NeptuneBlue"
ext3_style="xtheme-gray.css" ext_style="ext-all-
neptune.css" />
```

Managing Tabs in the Jacada WorkSpace Work Area

The work area consists of tabs in which information feeds, broadcasted messages and applications used by the agent are displayed.

In Jacada WorkSpace, tabs are managed using the *Layout Management > Tabs Configuration* page of the Jacada Management Console. This page lists available tabs, and enables you to configure the order of their appearance in Jacada WorkSpace. It also provides functionalities for editing the parameters of existing tabs, and creating new tabs.

For details on how to add and modify tabs, refer to the Jacada WorkSpace Administrator Guide.

Determining the Default Tab

By default, the tab that is displayed when agents log into Jacada WorkSpace is the *Message Board*. To change the default tab, go to the **override.js** file and replace the `tabid` in the `getDefaultActiveTab ()` method.

For example:

```
function getDefaultActiveTab() {  
    return 'messagesTab';  
}
```

Developing Multiple Level Tabs

You can define multi-level tabs: A parent tab that contains multiple sub-tabs. The sub-tabs function as standard tabs, for example, for nesting applications. An example multi-level tab is included in the `tabs.jspf` file, supporting files in `WebContent/USER/portlets/multiTab`.

In the `WebContent/USER/resources/js/override.js` file remove the comments surrounding the code for each of the following functions in order to override the default functions to enable showing and hiding nested applications within sub-tabs, when moving between tabs:

- `onApplicationInitialized()`
- `afterShowTabEvent(tabElement)`
- `afterShowTabById(tabId)`
- `afterShowCurrentVisibleTab()`
- `afterHideCurrentVisibleTab()`

Assigning Keyboard Shortcuts in Jacada WorkSpace

Many of the user interface components in Jacada WorkSpace can be accessed using a keyboard shortcut. A keyboard shortcut is specified using an `accessKey` attribute.

For example, in `WebContent/USER/resources/menus.xml` shortcut keys are specified for various menu items, such as for the *Layout* item under the *Edit* menu:

```
<menus>
  <item ...>
  <item name="application.menuBar.menu.edit"

class="SYSTEM.global.conditions.PersonalizationE
nabled"
    privilege="MenuBarItemEdit">
    <item
accessKey="application.menuBar.accessKey.layout"
    name="application.menuBar.item.layout"
    value="javascript:toggleAddContent(); "
privilege="LayoutManager"/>
    ...
    <separator/>
    ...
  </item>
  <item ...>
</menus>
```

You can change the value of the `accessKey` attribute as necessary.

Note: When defining shortcut keys, make sure the shortcut keys do not interfere with the functionality and the shortcuts of the external applications used by the agent. For a list of Windows shortcuts, see <http://support.microsoft.com/kb/126449>.

A shortcut key is activated using `ALT+<key>`, as is standard in Windows.

Using Jacada Specific JavaScript Functions

The following sections present Jacada-specific, JavaScript topics:

- [Using the \\$\(\) Function](#)
- [Using the \\$F\(\) Function](#)
- [Using the \\$D\(\) Function](#)
- [Using the \\$W\(\) Function](#)
- [Functions to Use For Client Network Configuration Information](#)
- [Functions to Use With Menu Items](#)
- [Functions to Use With Tabs](#)
- [Functions to Use With Nesting](#)
- [Modifying Jacada WorkSpace JavaScript Methods](#)

Using the \$() Function

The `$()` JavaScript function, a prototype function, is a shortcut to the `document.getElementById()` function of the Document Object Model (DOM). Use the `$()` function in your JavaScripts to return the element object to where the ID string, (`$('<id element>')`) exists. You are also able to pass more than one ID string and the `$()` function returns an array object with all the requested elements.

Using the \$F() Function

The `$F()` JavaScript function, a prototype function, is a shortcut that returns the value of any field input control, like text boxes or drop-down lists. Use the `$F()` function in your JavaScripts; it takes the ID string (`$F('<id element>')`) as an argument.

Using the \$D() Function

The `$D()` function is a Jacada-provided JavaScript library function that enables you locate to the root document once it has been initially defined in the code. For example, to modify an attribute in the main DOM, define as follows:

```
var mainDocument=$D();
```

Now you can modify an attribute from any inner frame within the code file, for example to change the height of the main document define the following:

```
mainDocument.body.style.height="1000px";
```

Using the \$W() Function

The `$W()` function is a Jacada-provided JavaScript library function that enables you locate to the root window. Below is an instruction to locate to the main window by accessing `myCommonFunction`:

```
$W().myCommonFunction();
```

You can also use the `$W()` function to define a global variable to access from another location:

```
$W().myNewVar="the value";
```

Functions to Use For Client Network Configuration Information

Use the following Jacada-provided functions to obtain various network configuration information about a client (Jacada WorkSpace) host machine:

Note: In order for these functions to be available for use in clients, the required, signed ActiveX controls that are supplied by the Jacada-provided **ApplicationHolderSetup.msi** file must be installed in every client host machine that runs Jacada WorkSpace.

getClientFQDNs() – Retrieves all the fully qualified domain names of the client host machine and returns them in an array.

getClientHostname() – Retrieves the hostname of the client host machine.

getClientIPAddresses() – Retrieves all the IP addresses of the client host machine and returns them in an array.

Functions to Use With Menu Items

You can use the following JavaScript functions:

runApplicationFromMenu(exe) – Runs the application specified in the menu. When the agent logs off, the application is closed.

where:

exe – The full path of the application. Use double backslashes (\\) for folders. For example, C:\\WINDOWS\\System32\\calc.exe.

openFavorite(url, window) – Opens the specified web site.

where:

url – The web site to run.

window – Set the this value to `false` to open the web site in a new browser window.

Functions to Use With Tabs

You can use the following JavaScript functions:

\$W().HideTabById(tabId) – Hides a tab. (This function is supported only when the `LayoutByContext` setting is `false`.)

where:

tabId – The ID defined for the tab to be hidden.

\$W().ShowTabById(tabId) – Shows a tab.

where:

tabId – The ID defined for the tab to be shown.

\$W().setTabTitle(tabId, title, localizedTitle) – Sets the tab title with the specified title or a localized title.

where:

tabId – The ID defined for the tab.

title – Either the string to display as the tab title or the localization key to obtain the tab title.

localizedTitle – When true, the title parameter contains a localization key and the method uses this key to set the localized version of the tab title.

\$W().RemoveTabById(tabId) – Removes a tab.

where:

tabId – The ID defined for the tab to be removed.

\$W().AddNewTab(tabConfig, [index]) – Adds a tab.

where:

index (optional) – The index at which the new tab will be inserted.

tabConfig – The new tab configuration object. The structure of the `tabConfig` object varies according to whether the new tab is a Web tab or an Application tab. The following examples show the structure for each.

Note: For a detailed explanation of Application tab settings, refer to the *Jacada WorkSpace Administrator Guide (Adding and Modifying Tabs)*.

tabConfig Object Structure: Web Tab

```
{
  isWeb:true, // must be set to true
  title:'New Tab', // the tab's title
  tabId:'newTabId', // the tab's id
  frameId:'newTabFrameId', //the tab's frame id
  closable:true, // if true, the agent can close the tab
  maximizable:true, // if true, the agent can maximize the tab
  url:'USER/portlets/sample/sample.jsp' // the url to load inside the tab
}
```

tabConfig Object Structure: Hosting Tab

```

{
    isWeb':false, // must be set to false
    title':'New Hosting Tab',
    tabId':'newNostingTabId',
    closable':true,
    maximizable':true,
    exeName':'calc.exe',
    workingDirectory':'c:/windows/system32',
    arguments':'',
    exeNameRegKey':'',
    workingDirectoryRegKey':'',
    argumentsRegKey':'',
    backgroundColor':'800080',
    linkCaption':'',
    detectBusy':false,
    busyMessage':'',
    showLoadMsg':false,
    loadingMsg':'',
    enableScrollbars':false,
    avoidPlaceInsideHost':'',
    recreateApp':false,
    alwaysHideWin':'',
    centerApps':'',
    showWindowInCorrectTab':''
}

```

Functions to Use With Nesting

You can use the following JavaScript functions:

isApplicationRunning(id) – Check whether the application with the specified *id* is running. Returns a boolean value. *id* is the *id* of the nested application, as specified in the hosting tag.

terminateApplication(id) – Terminate the application with the specified *id*. *id* is the *id* of the nested application, as specified in the hosting tag.

isProcessNested(id) – Returns *true* if the application for the process *id* is nested in a tab. *id* is the process *id* of the application.

terminateProcess(hostingId, process id) – Terminate the process with the specified process *id* that is being run by the ActiveX with the specified *hostingID*.

runApplication(id, name, path, args) – Runs the nested application.

where:

id – The `id` of the nested application, as specified in the `hosting` tag.
name – The full path to run the application.
path – The application working directory.
args – Application command arguments to run when the application starts.

Note: The application cannot be run from a batch file using this function.

Usage Example

The following example demonstrates how to use the `isApplicationRunning()` and `terminateApplication()` functions, by overriding the `showLogoutDialog` function in the main `index.jsp` file. In the example, before showing the logout dialog, if the application `Calculator` is running, an alert is displayed to the agent before the application is terminated:

```
function showLogoutDialog() {  
    if (isApplicationRunning("Calculator")) {  
        alert('Calculator application will be terminated.');        terminateApplication("Calculator");  
    }  
    showDialog('LogoutDialog', 820, 600);  
}
```

You can clip the nested application such that it does not cover portlets. To ensure a portlet is not covered by the nested application, you can use the following JavaScript functions:

registerIframeInNestedApplication(string id) – Registers the given portlet. This function must be called after all iframes are initialized.

unRegisterIframeInNestedApplication(string id) – Unregisters the given portlet.

setCurrentActiveNestedApplication(String id, Object ActiveX) – Registers the given object as the current activeX. It is assumed there is only one active ActiveX at a time.

clearCurrentActiveNestedApplication(String id) – Unregisters the given object as the current activeX. It is assumed there is only one active ActiveX at a time.

To ensure the portlet is visible you can use one of the following, which must be located after all the iframes are initialized, which is usually at the end of the file, before the `registerIframeInNestedApplication(string id)` function:

- In the *Portlet* page:

```
$W().displayManager.displayPortletOnTopOfNestedApplication(document.all('DialPadPortletFrameId'));
```

- Via JavaScript in the portlet onload function:

```
<iframe id="myIframe" width="100%" height="100%"
style="z-index:1590;"
src="USER/portlets/MyPortlet/myportlet.jsp"

onload="$W().displayManager.displayPortletOnTopOfNestedApplication(this);"
/>
```

- Via the iframe HTML tag:

```
<iframe id="myIframe" width="100%" height="100%"
style="z-index:1590;"
src="USER/portlets/MyPortlet/myportlet.jsp"

displayManager.displayPortletOnTopOfNestedApplication="true"/>
```

Note: In an `iframe` you must include a `z-index` attribute, as shown in the above examples. When using an `iframe` in a table, you must embed the `iframe` in a `<div>` tag.

Modifying Jacada WorkSpace JavaScript Methods

Jacada WorkSpace provides a set of JavaScript files (*.js) that contain methods. Specific methods can be modified (overridden). To modify a provided method, define a method with the same name in the file `WebContent/USER/resources/js/override.js`.

The following JavaScript methods can be overridden:

onLogin() – Called upon submitting the login form. The method returns a boolean value indicating whether the method succeeded. The source code is located in `WebContent/SYSTEM/resources/js/common.js`.

onKeyDownHandler() – The handler for the `keyDown` event for portlets and tabs. It is used for ignoring events such as clicking “backspace” etc. The source code is located in `WebContent/SYSTEM/resources/js/common.js`.

ctiClientErrorHandler: function(ctiClientInfo, data) – The client side error handler. The parameter `ctiClientInfo` is the last `ctiClient` object as received from the server. This object contains relevant data for the error. The parameter `data` is a string containing additional data, for example an error message. The source code is located in `WebContent/SYSTEM/Portlets/CTI/CTIClient.js`.

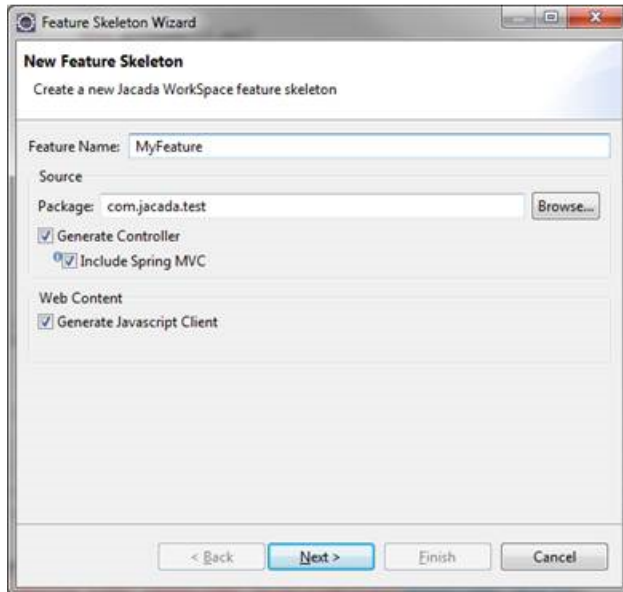
overrideCtiInit(ctiObject) – Used for overriding the `ctiClient init` function. The parameter `ctiObject` is the initialized `ctiClient` object as received from the server. This received object is initialized by the `Jacada init()` function. The source code is located in `WebContent/SYSTEM/Portlets/CTI/CTIClient.js`.

getCallCounterThresholdInSec() – The elapsed time in seconds to change the color of the counter to red. The default value is 120. The source code is located in `WebContent/SYSTEM/Portlets/CTI/CTIClient.js`.

handleCTIEvent (ctiClientEvent) – Activated when CTI is configured to send events to an external application. You add logic for handling the cti events. The parameter `ctiClientEvent` is the cti event object as received from the server. The source code is located in `WebContent/SYSTEM/Portlets/CTI/CTIClientEvent.js`.

Developing WorkSpace UI Components

WorkSpace UI elements are based on ExtJS components. To create a tab or a portlet in WorkSpace, use the Feature Skeleton wizard, and select the *Generate Javascript Client* checkbox. This creates a basic panel with a title only. For example:



In the above example, the following JavaScript class will be created:

```
Ext.define('Jacada.user.com.jacada.test.MyFeature', {
    extend: 'Ext.panel.Panel',

    title: 'MyFeature'

});
```

The package name, `Jacada.user.com.jacada.test.MyFeature`, indicates the location of the JS file containing the feature. The hierarchy is:

WebContent\USER\resources\js\com\jacada\test\MyFeature.js

If you need to add some content to an existing feature, you can modify the generated JS file, or manually create additional JavaScript files.

To configure a tab or portlet, use the following URL in the Management Console:

Jacada.user.com.jacada.test.MyFeature

Overriding WorkSpace Core UI Components

The WorkSpace user interface is developed with ExtJS. You can override almost any class to add or modify functionality.

For examples, go to **WebContent\USER\resources\js\WorkSpaceProjectExtensions.js**.

Debugging a Project

You use different tools to debug the Java code or JavaScripts in a Jacada WorkSpace project. The following sections describe how to debug the project.

Debugging the Java Code

Before being able to debug Java code, you need to enable debugging in the application server.

To set up Java debugging:

- Verify that the application server is started and displays the *Debugging* status. If not, start the server using the *Debug* option.

To debug a project:

1. In the development environment, open your project.
2. In *Project Explorer*, select the project you want to test.
3. Make sure that the project is built and that it is published on the server.
4. In the project Java files you want to test, add breakpoints to the code.
5. In an Internet Explorer web browser, run the project.

As each breakpoint is reached in the Java code, the development environment automatically switches to the *Debug* perspective, the affected Java file displays in the editor and the associated code is highlighted.

Debugging the JavaScripts

When you need to debug JavaScripts running in Jacada WorkSpace, use tools that enable debugging the JavaScript on the client. Jacada recommends using Microsoft Visual Studio 2008.

Setting Up JavaScript Debugging

Before being able to debug JavaScripts, you need to enable script debugging in Internet Explorer, and specify in the JavaScript where you want to start debugging.

To set up JavaScript debugging:

- In Internet Explorer, in the *Tools* menu, select *Internet Options* and in the *Advanced tab* uncheck the *Disable Script Debugging* checkboxes.

Just-In-Time Debugging

Just-in-time debugging enables debugging of JavaScripts that do not require sophisticated debugging.

To debug Jacada WorkSpace:

1. In the source JavaScript, add the line *debugger* wherever you want to check the code.
2. Run the application.
The *Just-In-Time Debugging* dialog opens when the execution comes to the *debugger* line with the possible tools you can use to debug the code.
3. From the dialog, choose the tool you want to use to debug the JavaScript from the list of possible debuggers.
It is recommended to choose Microsoft Visual Studio.

You enter the code from the point where you added the line *debugger* and can proceed through the code step by step from this point.

Full Debugging

If the *Just-in-time debugging* procedure does not resolve the problem, you can use the following procedure, which provides more sophisticated debugging options.

To debug Jacada WorkSpace:

1. Start running Jacada WorkSpace.
2. Open Microsoft Visual Studio.NET.
3. Select the debugging option. For example, in Microsoft Visual Studio.NET 2005, select *Tools > Attach to Process*.
A dialog opens displaying all the processes running on the PC.
4. Select the IEXPLORE.EXE process with the title of the Jacada WorkSpace application you need to debug.
5. Click the *Attach* button.
6. Select *Script* from the list of program types you want to debug.

7. Click *OK*.

The Internet Explorer process is displayed in the *Debugged Processes* area of the Processes dialog.

8. Click *Close*.

9. Select *Running Documents* from the *Debug > Windows* menu to open the *Running Documents* window.

The *Running Documents* window displays all the source files for the current page displayed in Internet Explorer.

10. In the *Running Documents* window, double-click the source file to debug.

The source of the selected file is displayed in Visual Studio.

11. Add breakpoints to the source to debug the JavaScript.

When execution gets to the breakpoint, you can start debugging the JavaScript.

12. After debugging the JavaScript, stop the debugging process from the *Debug* menu to finish.

Configuring Jacada WorkSpace Javadocs

The following procedure explains how to enable viewing of Javadocs documentation when a class with Javadocs available is selected in the project code. The workflow involves copying the Javadocs directory to your machine, and then configuring settings in Eclipse.

Before beginning the procedure, verify that you have created a project in Eclipse based on a JADTemplate.

To enable Javadocs documentation:

1. Copy the Javadocs directory from the installation CD to the hard drive of the machine where the Eclipse IDE is installed. You may archive the directory and copy the *.zip file.

2. In the Eclipse IDE, from the *Window* menu, select *Show View > Other*.

The *Show View* dialog opens.

3. In the textbox at the top of the dialog, enter *javadoc*. Then, select *javadoc* from the tree, and click *OK*.

The *Show View* dialog disappears, and the view is set.

4. From the *Project Explorer* tab, right-click on the relevant project, and select *Properties*.

The *Properties* dialog appears.

5. On the left side of the dialog, select *Java Build Path*.

The *Java Build Path* frame appears.

6. Select the *Libraries* tab, and expand the *Web App Libraries* node. Associated .jar files are listed.
7. Select a .jar file that you want to view (e.g., **JADSystem-7.3.jar**), and expand its node.
8. Select *Javadoc location*, and then, at the right side of the dialog, click *Edit*.
A pop-up window appears.
9. In the *Javadoc location path* textbox, enter the file path of the Javadocs directory. Alternatively, click *Browse*, and navigate to the relevant file path.
The file path can refer to either the archive or the Java directory of the Javadoc resources that you copied to your local disk.
10. To check the validity of the specified location, click *Validate*.
Validation results are displayed in a pop-up window.
11. After successful validation, from the pop-up windows, click *OK*.
Then, at the bottom of the *Properties* dialog, click *OK*. Configuration settings are saved.
To view the Javadocs documentation in the IDE, navigate (*Ctrl + Shift + T*) to a class that resides inside the Jacada WorkSpace jar and has Javadocs. (For example, in the **JADSystem-7.3.jar**, you can navigate to SearchBridge.) Then, in the editor pane, click the class name.

Main Portlets of Jacada WorkSpace

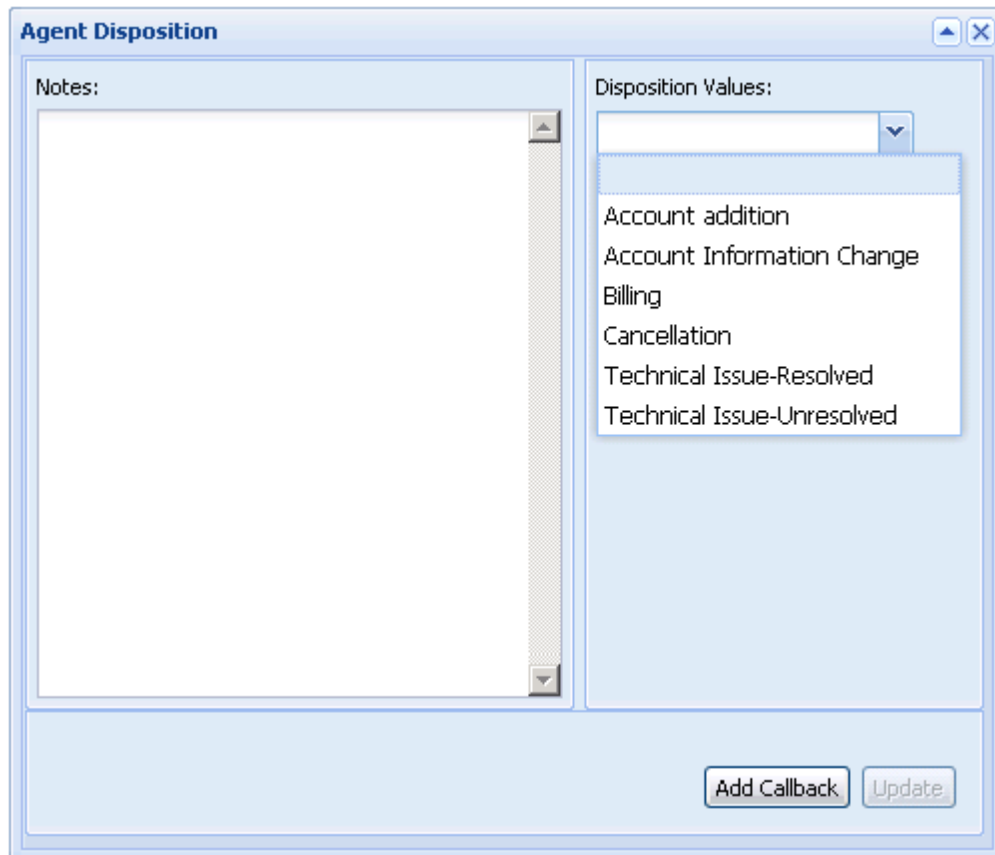
The following sections describe the main portlets supplied with Jacada WorkSpace:

- [The Agent Disposition Portlet](#): Enables agents to record information about a call.
- [The Agent Tasks Portlet](#): Enables agents to create, display, track, modify and delete their own set of tasks.
- [The Document Search Portlet](#): Enables agents to conduct a text search within documents in specified document sources.
- [The Help-On-Hand Portlet](#): Conducts text searches to proactively provide relevant search results to the agent.
- [The Smart Pad Portlet](#): Enables agents to store and view information required during a call.
- [The Guidance Portlet](#): Enables agents to access self service Interactions from the Jacada Interaction Server.

The Agent Disposition Portlet

The *Agent Disposition* portlet enables agents to record information about a call. When the agent clicks the *Update* button of the portlet, an event is fired, containing all the data in the portlet. A listener to the event can be registered at the project level, and the data can then be used as required.

The *Agent Disposition* portlet opens automatically when an agent clicks *End Call*, and enters the disposition state. (This flow can be configured via the Jacada Management Console.) It can also be accessed from the toolbar, by clicking .



The portlet contains the following frames:

- **Notes:** A free text field for the agent to record notes about a call.
- **Disposition Values:** Categories that provide summary information about the content of the call. Agents select disposition values from listboxes. By default, 3 listboxes are available, providing the following options:
 - Account addition
 - Account information change
 - Billing
 - Cancellation
 - Technical issue- resolved
 - Technical issue- unresolved

In the project code, you can change disposition values, or add more values to the default list. Disposition values can have a hierarchical relationship with one another or can be discrete, unrelated values.

Clicking the *Update* button in the portlet, either by an agent or in the project code, causes entries of recorded notes, disposition values or both to be archived in Jacada WorkSpace. The following timestamp information is provided for each disposition value in the archived entry:

- The time when the disposition value was selected.
- The time, relative to the start of the call, when the disposition value was selected.

In the Jacada WorkSpace runtime environment, clicking *Update* automatically ends the disposition state, freeing the agent to begin another call.

Development Infrastructure

Jacada WorkSpace provides a coding infrastructure that is used to develop *Agent Disposition* portlet-related processing. This infrastructure makes available the following interface:

```
public interface DispositionManager {  
    ..  
}
```

The `DispositionManager` interface exposes the following methods and listeners:

Methods and Listeners

Method	Description
<code>public void startDisposition();</code>	This method starts the call wrap-up state; In doing so, the method enables all UI components and initializes all existing information (notes, disposition values) in the <i>Agent Disposition</i> portlet.

Method	Description
<pre>public void setAutomaticDispositionEnabled(boolean enable);</pre>	This method enables/disables automatic disposition. When set to <code>true</code>, the <i>Agent Disposition</i> portlet automatically pops up when an agent enters a disposition state. If the method is not set, automatic disposition is set according to the value specified in the Jacada Management Console.
<pre>public boolean isAutomaticDispositionEnabled();</pre>	This method checks whether automatic disposition is enabled or disabled. If automatic disposition is enabled, the method returns <code>true</code>.
<pre>public void setSelection (String code, int index, boolean editable);</pre> or alternatively, <pre>public void setSelection (String code, boolean editable);</pre>	This method populates a call disposition value in a disposition value field in the Agent Disposition portlet. The method's alternate version only operates on the first disposition value field displayed. • <code>code</code> – specifies a disposition value. If the specified code is not available to the field, the method has no effect and no exception is thrown. • <code>index</code> – specifies a field to receive the disposition value. When a hierarchical relationship exists between disposition values, the method sets the <code>index</code> to the value 0, thereby always specifying the top tier field in the hierarchy. When disposition values are unrelated and the specified <code>index</code> does not identify an available field, the method has no effect and no exception is thrown. • <code>editable</code> – defines whether or not the specified field can be edited by the agent.

Method	Description
<code>public void sendDisposition();</code>	This method invokes a <i>disposition performed</i> event. In doing so, the generated event contains all existing information (notes, selected disposition values) from the Agent Disposition portlet and any associated timestamps.
<code>public void addListener(DispositionInitializedListener listener);</code>	When Java class code registers as a listener of this type, the <code>DispositionManager</code> notifies the code about the occurrence of the following event: Existing information (notes, disposition values) from the Agent Disposition portlet is initialized (the <i>disposition initialized</i> event), due to the start of the call wrap-up state.
<code>public void addListener(DispositionCodesLoadedListener listener);</code>	When Java class code registers as a listener of this type, the <code>DispositionManager</code> notifies the code about the occurrence of the following event: Server-side loads the list of relevant disposition values for client-side display (the <i>disposition codes loaded</i> event).
<code>public void addListener(DispositionEventListener listener);</code>	When Java class code registers as a listener of this type, the <code>DispositionManager</code> notifies the code about the occurrence of the following event: Agent clicks the <i>Update</i> button in the Agent Disposition portlet (the <i>disposition performed</i> event). The generated event contains all existing information (notes, selected disposition values) from the Agent Disposition portlet and any associated timestamps.

Method	Description
<pre>public void removeListener(eventListener listener);</pre>	In your Java class code, remove a listener registration. <i>eventListener</i> is any of the following: • <code>DispositionInitializedListener</code> • <code>DispositionCodesLoadedListener</code> • <code>DispositionEventListener</code>

Disposition Listener Interfaces

Each disposition-related event listener defines a listener interface that exposes its own methods. Use these methods to trigger processing following receipt of the related event. The code must always first register as a specific event listener.

The following methods are exposed:

Method	Description
<pre>public void afterDispositionInitialized();</pre>	The <code>DispositionInitializedListener</code> interface exposes this method. Invoke this method to trigger processing, upon being notified that a disposition initialized event occurred.

Method	Description
<pre>public void onDispositionCodesLoaded (DispositionCodesLoadedEvent eload);</pre>	The <code>DispositionCodesLoadedListener</code> interface exposes this method. Invoke this method to trigger processing, upon being notified that a disposition codes loaded event occurred. The parameter <i>eload</i> contains the list of relevant disposition values that was loaded by the server-side for client-side display.
<pre>public void dispositionPerformed (DispositionEvent eperform);</pre>	The <code>DispositionEventListener</code> interface exposes this method. Invoke this method to trigger processing, upon being notified that a disposition performed event occurred. The parameter <i>eperform</i> contains all existing information (notes, selected disposition values) from the Agent Disposition portlet and any associated timestamps.

DispositionCodesLoadedEvent Object

The `DispositionCodesLoadedEvent` object exposes methods for retrieving information from its associated parameter. The available information results from the occurrence of the disposition codes loaded event. The following methods are exposed:

Method	Description
<pre>public List<DispositionCode> getDispositionsCodes();</pre>	Retrieves from <i>eload</i> the list of relevant disposition values that was loaded by the server-side for client-side display. Method returns either a list of <i>DispositionCode</i> or null when there is no list of relevant disposition values.

DispositionEvent Object

The *DispositionEvent* object exposes methods for retrieving information from its associated parameter, for example, *eprform*. The available information results from the occurrence of the disposition performed event. The following methods are exposed:

Method	Description
<pre>public List<AgentDisposition> getAgentDispositionValues();</pre>	Retrieves from <i>eprform</i>. all the selected disposition values gathered from the Agent Disposition portlet. Method returns either a list of <i>AgentDisposition</i> or null when there are no disposition values.
<pre>public String getAgentNotes();</pre>	Retrieves from <i>eprform</i>. the notes gathered from the Agent Disposition portlet. Method returns either a string representing the notes or null when there are no notes.

Example: Incorporating Listeners in the Project

This example illustrates how to add the available disposition-related event listeners and enable the code to process disposition-related events.

1. Define a class in your Spring configuration file and set the `dispositionManager` property, as follows:

```
<bean id="myBeanId" class="myPackage.MyClass"
init-method="init">
  <property name="dispositionManager"
ref="dispositionManager"/>
</bean>
```

Note: If you do not have a Spring configuration file defined, create one named `applicationContext-operation.xml` and locate it under `WebContent/WEB-INF`.

Add the following `<beans>` element definition to this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/
schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://
www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-
beans-2.0.xsd">
</beans>
```

Add the `<bean>` element within the `<beans>` element.

In the file `WebContent/WEB-INF/web.xml`, find the `contextConfigLocation` `context-param`. This is where all Spring xml files are listed. Add the following entry to the list: `/WEB-INF/applicationContext-operation.xml`.

2. Create a new java class for the bean defined in the Spring configuration file, `myPackage.MyClass`, and create a setter method to the `DispositionManager` interface, as follows:

```
public class MyClass {  
    private DispositionManager agentDisposition;  
    ..  
    ..  
    public void  
    setDispositionManager(DispositionManager  
    agentDisposition) {  
        this.agentDisposition = agentDisposition;  
    }  
}
```

3. Add the code, in the new class, to register listeners and trigger project processing following receipt of the related event:

```

public void init() {
    agentDisposition.addListener(new
DispositionEventListener(){
    public void
dispositionPerformed(DispositionEvent eprform) {
        AgentDisposition disposition =
eprform.getAgentDisposition();
        // Now take the data and add your own flow.
    }});
    agentDisposition.addListener(new
DispositionInitializedListener(){
    public void afterDispositionInitialized() {
        // Set the first disposition value field
displayed to the disposition
        // value: billing code. Also, field cannot
be edited by the agent.
        agentDisposition.setSelection("BLG", 0,
false);
    }});
    agentDisposition.addListener(new
DispositionsCodesLoadedListener(){
    public void onDispositionsCodesLoaded
(DispositionsCodesLoadedEvent eload) {
        List<DispositionCode> codes =
eload.getDispositionsCodes();
        //Remove the first code in the list.
        codes.remove(0);
    }});
}

```

Example: Disabling Automatic Disposition in the Project

Generally, automatic disposition is enabled/disabled using the Enable Automatic Call Disposition setting, on the *Settings* page of the Jacada Management Console. (For details, refer to the Jacada WorkSpace Administrator Guide.) By default, the setting is set to `true` (automatic disposition is enabled).

The behavior defined in the Jacada Management Console can be overridden in the project code, using the following exposed methods in the `DispositionManager` class:

- `public void setAutomaticDispositionEnabled(boolean enable);`
- `public boolean isAutomaticDispositionEnabled();`

Note: For a description of the methods, refer to [Development Infrastructure](#).

The following example illustrates how to check whether automatic disposition is enabled, and disable it if it is enabled.

```
public class ExtensionManager implements
AbstractManager {
    private static final long serialVersionUID
= 1L;
    private transient SharedData globalApp;
    private DispositionManager disposition;

    public DispositionManager getDisposition()
{
        return disposition;
    }

    public void
setDisposition(DispositionManager disposition) {
        this.disposition = disposition;
    }

    @GlobalAction(actionName=GlobalActionNames.LOGIN
,order=GlobalActionOrder.AFTER
    public void login(HttpServletRequest request) {
        if
(disposition.isAutomaticDispositionEnabled()) {
disposition.setAutomaticDispositionEnabled(false
);
        }
    }
}
```

In addition, verify that the Extension Manager refers to the `dispositionManager` in the **WebContent/WEB-INF/applicationContext-user.xml** file, as shown in the following example:

```
<bean id="extensionManager"
class="USER.global.managers.ExtensionManager"
scope="session" lazy-init="true">
    <aop:scoped-proxy/>
    <property name="disposition"
ref="dispositionManager" />
</bean>
```

The Agent Tasks Portlet

The *Agent Tasks* portlet enables agents to create, display, track, modify and delete their own set of tasks. Created tasks include when the task is due and its status. Tasks persist across Jacada WorkSpace sessions and are stored in the database. By default, the *Agent Tasks* portlet is

available from the toolbar, by clicking .

The portlet is comprised of the two panes, one above the other:

Task pane – Displays basic information associated with each task entry: a description, due date, due time, reminder and status. A task can be added manually by the agent or by the developer via the project code. Tasks added by the developer in the code, by default, are always editable by the agent. The developer can make an added task not editable in the *Agent Tasks* portlet.

Additional information pane– Displays any additional information that exists in the database for a task currently selected in the task pane.

Additional information entries are created, when a task is added, using custom processing in the Jacada WorkSpace project code. Additional task information, when it exists, is read-only and cannot be modified by the agent.

Agent Tasks

View: Personal Status: All

Description	Due Date	Reminder	Status ▲
Review progress	12-07-2011 15:13	5 minutes	Not Completed
Call customer	12-07-2011 15:14	2 days	Not Completed

Refresh Add Delete Delete All

Review progress

Name	Value ▲

The portlet presents a task's *Due Time* in the local time zone associated with the agent.

Note: Tasks that are added by the developer via the project code can be added without an associated agent time zone. In this case, the *Due Time* of the added task is established with the time zone of the Jacada WorkSpace application server.

When the *Agent Tasks* portlet is open and a task *Reminder* comes due, the portlet automatically selects the relevant task entry and the display then flashes several times. When the portlet is open and a task comes due, based on its *Due Date* and *Due Time*, the portlet automatically selects the

relevant task entry and the display then flashes a single time. When the *Agent Tasks* portlet is not open and either a task *Reminder* or a task comes due, the portlet itself flashes several times and its color changes. Should the agent then subsequently open the *Agent Tasks* portlet, the portlet itself returns to its standard color for the open state.

Overdue tasks display in gray highlight, when not currently selected.

Available Actions

The agent can perform the following actions in the portlet:

Button	Action	Comments
	Double-click on a field to edit it.	The <i>Status</i> field of a non-editable task is always editable.
Refresh	Refreshes the information displayed in the portlet.	—
Add	Adds a new task.	Displays the <i>Add Task</i> dialog. The agent specifies the new task's description and assigns its due date, due time and reminder period.
Delete	Deletes the selected tasks.	This button is disabled when the selected task is not editable.
Delete All	Deletes all tasks.	Only editable tasks are deleted.

In the project code, tasks can be reassigned from one agent to another agent and from one agent to an external agent, who is defined in a BPM system that is integrated with Jacada WorkSpace. Also, tasks can be created such that they are automatically escalated from one agent to another agent, when they become overdue by a defined time period. Reassignment and escalation of tasks occur independently of agents being in logged in to a Jacada WorkSpace session.

Development Infrastructure

Jacada WorkSpace provides a coding infrastructure that is used to develop *Agent Task* related processing. This infrastructure makes available the following interface:

```
public interface taskManager {
    ...
}
```

taskManager Methods

The **taskManager** interface exposes the following methods:

Method	Description
<code>addTask(AgentTask task, TimeZone agentTimeZone)</code> throws <code>TaskLimitExceededException</code>; or alternatively, <code>addTask (AgentTask task)</code> throws <code>TaskLimitExceededException</code>;	Adds a task in the database. By default, an added task is editable by the agent. When an added task is made not editable, the agent can still modify the task's status. Supplying the agent's time zone in the <code>agentTimeZone</code> parameter, establishes the due time of the task with the provided time zone. Use the <code>agentTimeZone</code> parameter with the method, when the Jacada WorkSpace application server and the agents that it serves are not located in the same time zone. Not providing the <code>agentTimeZone</code> parameter in the method, establishes the due time of the added task with the time zone of the Jacada WorkSpace application server. If the number of tasks in the tab has reached its maximum, defined by the <code>maximumAgentTasks</code> setting, an exception is thrown.
<code>deleteTask(Long taskId);</code>	Deletes the task with the specified ID from the database.

Method	Description
<pre>reassignTaskInternalToRole(Long taskId, String role, Map<String, String>attachedData)throws TaskReassignmentException;</pre> Or, alternatively: <pre>reassignTaskInternalToRole(Long taskId, String role, Map<String, String>attachedData, boolean overrideExistingAttachedDataKey) throws TaskReassignmentException;</pre>	Reassigns the identified task, <code>taskId</code>, to any agent who is assigned the identified role. Optionally, add <code>attachedData</code> to the reassigned task (<code>attachedData</code> are key-value pair entries associated with the task that display in the portlet information pane). By default, an added key-value pair overrides an existing pair assigned the same key. Use the alternative method version to specify whether to override (<code>true</code>) or not (<code>false</code>). Method returns the new <code>taskId</code>. If the identified task or role does not exist or if no agent is assigned the identified role, an exception is thrown. Note: Users being reassigned tasks based on a specified role must also be assigned the role <code>SupervisorRole</code>.

Method	Description
<pre>reassignTaskInternalToUser(Long taskId, String username, Map<String, String>attachedData) throws TaskReassignmentException</pre> Or, alternatively: <pre>reassignTaskInternalToUser(Long taskId, String username, Map<String, String>attachedData, boolean overrideExistingAttachedDataKey) throws TaskReassignmentException</pre>	Reassigns the identified task, <code>taskId</code>, to the identified agent, <code>username</code>. Optionally, add <code>attachedData</code> to the reassigned task (<code>attachedData</code> are key-value pair entries associated with the task that display in the portlet information pane). By default, an added key-value pair overrides an existing pair assigned the same key. Use the alternative method version to specify whether to override (<code>true</code>) or not (<code>false</code>). Method returns the new <code>taskId</code>. If the identified task or agent does not exist, an exception is thrown.
<pre>reassignTaskExternal (taskId, String username)</pre>	Reassigns the identified task, <code>taskId</code>, to an external agent, <code>username</code>, in a BPM system.

In the portlet information pane of any reassigned task, Jacada WorkSpace, records the following entry for the associated task: *Name*: Task Reassigned and *Value*: Reassigned on <date> from agent <agent>.

Adding a task to the portlet, either by an agent or in the project code, or a task becoming due triggers processing in Jacada WorkSpace. The **taskManager** interface exposes the following listener methods:

- To listen for added tasks, use the method:

```
public void addListener(eventListener listener);
```

- To remove this listener, use the method:

```
public void removeListener(eventListener listener);
```

where *eventListener* is one of the following:

TaskAddedListener – Listens for added tasks.

TaskBecameDueListener – Listens for tasks that have become due.

Listener and Event Handling Processing Methods

Once a listener is defined, use the following methods, which identify the event to listen for, to trigger processing:

Event Method	Description
<code>public void onTaskAdded (TaskAddedEvent event);</code>	Event when a task is added.
<code>public void onTaskBecameDue (TaskDueEvent event);</code>	Event when a task becomes due.

The following table lists the methods that can be executed for an added task or for a task becoming due:

Method	Description
<code>public String getAgentName();</code>	Retrieves the name of the agent assigned to the task.
<code>public String getDescription();</code>	Retrieves the description of the task.

Method	Description
<pre>public void addAttachedData(String name, String value, Boolean override);</pre> or alternatively, <pre>public void addAttachedData(String name, String value);</pre>	This method is only exposed to use with the <code>TaskAdded</code> event. For the added task, this method writes task information in the database. When the <code>override</code> parameter is set to true, then for the given name, the method overwrites the existing value in the database with the supplied value. The default <code>override</code> value is false.
<pre>public Long getTaskId();</pre>	This method is only exposed to use with the <code>TaskDue</code> event. Retrieves the ID of the task.

Escalating Tasks

In project code, you can create tasks such that they are automatically escalated from one agent to another agent, when they become overdue by a defined time period. In support of task escalation, the agent task class includes the following properties:

Property Name	Description
Escalate	The default is <code>false</code>. Indicates whether the task can be escalated or not.

Property Name	Description
<code>escalateTime</code>	Integer value. Identifies the waiting period, prior to escalating the task that has reached its due date.
<code>escalateRecipientUser(agent name)</code>	String value. Identifies the agent to receive the task, in the event of escalation.
<code>escalateRecipientRole(role name)</code>	String value. Identifies the role that must be assigned to any agent receiving the task, in the event of escalation. Note: Users being reassigned tasks based on a specified role must also be assigned the role <code>SupervisorRole</code>.

The agent task class exposes the following methods that define the waiting period, prior to Jacada WorkSpace escalating the task that has reached its due date.

Method	Description
<code>setEscalateTime (int time)</code>	Integer value. Defines the waiting period in number of days.

Method	Description
<code>setEscalateTime(int time, EscalateUnits units)</code>	<code>time</code> is an integer value. <code>units</code> is any one of the following valid values: Days, Hours, Minutes. Defines the waiting period in number of supplied units.

Code Example

The following code example constructs a couple of escalatable tasks:

```
AgentTask task = new AgentTask();
task.setAgentName("CatalogSales");
task.setDescription("Compile targets for holiday
sales campaign");
task.setDueDate(new Date());
task.setEscalate(true);
task.setEditable(false);
task.setEscalateTime(3); // 3 day overdue
waiting period.
task.setEscalateRecipientRole("SupervisorRole");

AgentTask task = new AgentTask();
task.setAgentName("CustomerSupport");
task.setDescription("Schedule meeting to revise
call protocol");
task.setDueDate(new Date());
task.setEscalate(true);
task.setEditable(true);
task.setEscalateTime(54,
EscalateUnits.Hours); // 54 hour overdue

/

/ waiting period.
task.setEscalateRecipientUser("workspace");
```

In the portlet information pane of any escalated task, Jacada WorkSpace records the following entry for the associated task: **Name:** Task Escalated and **Value:** Escalated on <date> from agent

`<agent>`.

Customizing Date and Time Display

By default, Jacada WorkSpace formats all date and time displays using the `DATE_FORMAT` and the `TIME_FORMAT` properties that are defined in the `WebContent/index.jsp` file. Customize Jacada WorkSpace's display of date and time information for Agent Tasks related messages.

Accomplish this by overriding the default date and time display formats. In the `WebContent/USER/rresources/js/override.js` file, define any of the following properties:

- `TASK_DATE_FORMAT`
- `TASK_TIME_FORMAT`

Each property that is defined, customizes Jacada WorkSpace's display of the associated information. Define these properties using a set of standard JavaScript notations for the supported date and time formats. For a listing of the supported, format notations, see [Customizing Date and Time Displays](#).

For example, incorporating the following format property definitions in `override.js`:

```
TASK_DATE_FORMAT = 'm/d/y';
TASK_TIME_FORMAT = 'g:i A';
```

results in the *Agent Tasks* portlet displaying its task entries, where the *Due Date* field displays the custom formatted date and time in the following manner: 03/05/09 7:09 AM.

Example: Agent Task Processing

The following code examples illustrate how to enable the code to add a task, add the available listeners and handle events and reassign a task.

1. Define a class in your Spring configuration file and set the `taskManager` property, as follows:

```
<bean id="myBeanId" class="myPackage.MyClass" init-method="init">
  <property name="taskManager" ref="taskManager"/>
</bean>
```

Note: If you do not have a Spring configuration file defined, create one named `applicationContext-operation.xml` and locate it under `WebContent/WEB-INF`.

Add the following `<beans>` element definition to this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/
schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://
www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-
beans-2.0.xsd">
</beans>
```

Add the `<bean>` element within the `<beans>` element.

In the file `WebContent/WEB-INF/web.xml`, find the `contextConfigLocation context-param`. This is where all Spring xml files are listed. Add the following entry to the list: `/WEB-INF/applicationContext-operation.xml`.

2. Create a new java class for the bean defined in the Spring configuration file, `myPackage.MyClass`, and create a setter method to the `TaskManager` interface, as follows:

```
public class MyClass {
    private TaskManager taskManager;
    ..
    ..
    public void setTaskManager(TaskManager taskManager) {
        this.taskManager = taskManager;
    }
}
```

3. The following code, in the new class, illustrates how to add a task in the project:

```

public void addTaskExample() {
    AgentTask task = new AgentTask();
    task.setAgentName("Agent");
    task.setDescription("Task description");
    task.setDueDate(new Date());
    task.setReminder(TaskReminder.FIVE_MINUTES.getMinutes());
    task.setEditable(false);

    taskManager.addTask(task);
}

```

4. The following code, in the new class, illustrates how to handle `TaskAdded` and `TaskDue` events:

```

public void addListenersExample() {
    taskManager.addListener(new TaskAddedListener() {
        public void onTaskAdded(TaskAddedEvent event) {
            // Add your code here
        }
    });

    taskManager.addListener(new TaskBecameDueListener() {
        public void onTaskBecameDue(TaskDueEvent event) {
            // Add your code here
        }
    });
}

```

5. The following code, in the new class, illustrates how to reassign a task in the project:

```

public void myOwnLogic(long taskid) {
    Map<String, String> myData = new HashMap<String, String>();
    myData.put("reassign date", Calendar.getInstance().toString());
    taskManager.reassignTask(taskid, "workspace", null, myData)
}

```

The Document Search Portlet

The *Document Search* portlet provides the agent with ability to conduct a text search within documents in specified document sources. By default, the *Document Search* portlet is available

from the toolbar, using the following button: 

Text Searches

The *Document Search* portlet supports text searches of its document source as follows:

- A search of both a document's title and body. The search string must either appear in the title of a document, the body of a document or in both.
- In order to search for an exact phrase, enclose the phrase within double-quotes [""]. For example, when searching for the phrase *Four score and seven years ago our fathers brought forth on this continent a new nation*, you must enter the string in the search field as `"Four score and seven years ago our fathers brought forth on this continent a new nation"`.
- When a phrase is not enclosed within double-quotes, the search attempts to find a match of any of the words entered in the search field.
- Should the agent want to match all of the words entered in the search field, the AND operator is used between the words. For example, to locate a match for all three words, children, love and animals, the search string is `children AND love AND animals`.

Wildcard Characters

The *Document Search* portlet supports using the following wildcard characters within a search string:

Asterisk [*] – A multiple character wildcard.

Question mark [?] – A single character wildcard.

Wildcard characters cannot be used as the first character in a search string.

All special character, for example, the dollar sign or exclamation point[\$, !], and the wildcard characters, if the agent does not want them processed as wildcard characters, must be prefaced by the backslash character [\\]. For example, to search for the string *Bravo!*, the agent must enter the string as `Bravo\\!`.

Should the agent modify the Jacada WorkSpace layout, by selecting an available layout option or the *Reset to Defaults* option, the result panes for every document source are cleared of any displayed search results.

When a document source result pane has more results to display than fit on a page, the paging controls are made available for use. Relevant only when the document search setting *Results Mode* is defined, in Jacada Management Console, as *separate*.

Each search result is presented as a hyperlink in the applicable document source result pane. Click a result to display it. Search results display in either Jacada WorkSpace's *Document Display* tab or

a new Internet Explorer window, depending how the document search setting *Document Display Destination* is defined, in Jacada Management Console.

Document Sources and Search Results

A document source is any of the following:

Internal document source – The collection of directories that are defined, using the Jacada Management Console, as available for the project to search and located on a shared network drive that is accessible to all application servers hosting the Jacada WorkSpace project.

External document source – Any public or private document source that exists outside of the network resources of the Jacada WorkSpace project. The portlet can be developed to search multiple, external document sources.

The portlet presents its search results in the form of selectable hyperlinks, using one of the following, configurable, display options:

Separate – A separate pane presents the results obtained from each document source.

Combine – A single pane presents only a single page of combined results obtained from all document sources.

Note: When combining the obtained results to prepare a single page of results, search processing selects, in an iterative manner, the top result from each contributing document source until the defined results page size is reached.

The ability to search requires settings in Internet Explorer and the Jacada Management Console, as described in the Jacada WorkSpace Administrator Guide.

Development Infrastructure

Jacada WorkSpace provides a coding infrastructure that is used to develop *Document Search* related processing. This infrastructure makes available the SearchBridge interface. Implementing the SearchBridge interface enables the *Document Search* portlet to access and search a document source. For the portlet to access and search a specific, external document source, create a class that implements the SearchBridge interface.

SearchBridge Interface

The SearchBridge interface exposes the following methods:

Method	Description
<code>String getName();</code>	Retrieves the name of the specific document source for display in the relevant results pane of the portlet.
<code>boolean supportsPaging();</code>	Determines whether or not an external document source supports paging of its results. Method returns either <code>true</code> or <code>false</code>. If an external document source supports paging, use the <code>PagingResults<SearchResult> search</code> method in your <code>SearchBridge</code> implementation to search and perform paging of the obtained results.
<code>Set<SearchResult></code> <code>search(SearchQuery query)</code> throws <code>SearchReportedException;</code>	Performs a search based on the provided <code>query</code> parameter. If an external document source does not support paging, use this method in your <code>SearchBridge</code> implementation to search. <code>query</code> – parameter specifies a <code>SearchQuery</code> object. Method returns either a complete set of search results or null, when there are no search results obtained from the external document source. Method can throw an exception for displaying an error message to the user. See Error Message Processing.

Method	Description
<pre>PagingResults<SearchResult> search(SearchQuery query, int firstResult, int maxResults) throws SearchReportedException;</pre>	Performs a search based on the provided query parameter. If an external document source supports paging, use this method in your SearchBridge implementation to search and perform paging of the obtained results. <code>query</code> – parameter specifies a <code>SearchQuery</code> object. <code>firstResult</code> – parameter specifies the index of the first result for the current search round. <code>maxResults</code> – parameter specifies the maximum number of results to return for the current search round. Method returns a <code>PagingResult</code> object. This object contains the following items: • A set of search results based on the provided parameters <code>query</code>, <code>firstResult</code> and <code>maxResults</code>. • A count of the total number of search results for displaying the number of pages of results. Method can throw an exception for displaying an error message to the user. See Error Message Processing.

Add the Spring bean that defines each specific `SearchBridge` to the `WebContent/WEB-INF/USER/applicationContext-search.xml` file, in the following section:

```
<!-- The list of the SearchBridges that will be used by search portlet -->
<util:list id="bridgesList" list-class="java.util.ArrayList"
value-type="com.jacada.jad.search.SearchBridge">
  <ref bean="internalIndexingSearchBridge"/>
  <ref bean="sampleSearchBridge"/>
</util:list>
```

The Spring bean definition of the Jacada WorkSpace-provided `SearchBridge`, named **`internalIndexingSearchBridge`**, is used to access and search the project's internal document source appears in the section of the `applicationContext-user.xml` file shown

above. If the project does not need the portlet to search the internal document source, or any customized SearchBridge for that matter, either delete its Spring bean definition or place this definition within a comment.

The Jacada WorkSpace-provided SearchBridge, `internalIndexingSearchBridge`, supports using the following wildcard characters within a search string:

Asterisk [*] – A multiple character wildcard.

Question mark [?] – A single character wildcard.

Wildcard characters cannot be used as the first character in a search string.

All special characters, for example, the dollar sign or exclamation point[\$, !], and the wildcard characters, if the agent does not want them processed as wildcard characters, must be prefaced by the backslash character [\]. For example, to search for the string *Bravo!*, the agent must enter the string as `Bravo\!`.

By default, a project also supplies a sample SearchBridge, named `sampleSearchBridge`. This sample SearchBridge produces search results for the following strings, which are entered in the search field of the *Document Search* portlet:

- `billing`
- `insurance`
- `telco`

The relevant implementation classes are `SampleSearchBridge` and `SampleSearchResult` in `src/USER/search`. The relevant bean configurations are located in the `WebContent/WEB-INF/USER/applicationContext-search.xml` file, in the `sampleSearchBridge`, `sampleSearchResults`, `billingResults`, `insuranceResults` and the `telcoResults` beans.

SearchQuery Class

The development infrastructure makes available the `SearchQuery` class that represents the actual search query. The `SearchQuery` class exposes the following methods:

Method	Description
<code>public String getText();</code>	Retrieves the text from the SearchQuery object.

Method	Description
<code>Public void setText();</code>	Sets the text in the SearchQuery object.

SearchResult Interface

The development infrastructure makes available the **SearchResult** interface. The SearchResult interface exposes the following methods:

Method	Description
<code>String getTitle();</code>	Retrieves a search result title for display as a hyperlink in the relevant results pane of the portlet.
<code>String getUrl();</code>	Retrieves the URL of a search result for opening and displaying the associated document in the display destination, as defined in the Jacada Management Console.

Localization for Document Indexing

In order for document indexing to function for a specific language, the project must have a dedicated analyzer for the language. Jacada WorkSpace provides an English language analyzer for its provided SearchBridge, named **internalIndexingSearchBridge**.

If you require document indexing to function for any additional languages, other than English, you must define and register a dedicated analyzer per language. Add language analyzers in `WebContent/WEB-INF/USER/applicationContext-search.xml` file.

The following sample code provides an example of defining and registering a dedicated analyzer for French:

```

<!-- Define an analyzer for French
<bean id="frAnalyzer"
class="org.apache.lucene.analysis.snowball.SnowballAnalyzer">
  <constructor-arg value="French"/>
</bean>

<!-- Register the French analyzer in the existing analyzers map'
<util:map id="analyzersMap" map-class="java.util.HashMap">
  <entry key="en" value-ref="standardAnalyzer" />
  <entry key="fr" value-ref="frAnalyzer" />
</util:map>

```

The dedicated analyzer for French, **frAnalyzer**, defined in the above sample code, is of type `SnowballAnalyzer`. The `SnowballAnalyzer` type enables a `SearchBridge` to use **stemming** in its document searches. Generally, stemming involves extracting the root words, dropping such things as plurals and verbs ending with *ing*. A stemming-based search using the term “reformation” produces results that includes instances of the provided term and also variations of the provided term, for example, *reform*, *reforming*, *form* and *formation*.

The following sample code defines a dedicated analyzer for French, **frStdAnalyzer**, that is of type `StandardAnalyzer`. The `StandardAnalyzer` type does not enable a `SearchBridge`’s use of **stemming**:

```

<!-- Define an analyzer for French
<bean id="frStdAnalyzer"
class="org.apache.lucene.analysis.standard.StandardAnalyzer" />

<!-- Register the French analyzer in the existing analyzers map'
<util:map id="analyzersMap" map-class="java.util.HashMap">
  <entry key="en" value-ref="standardAnalyzer" />
  <entry key="fr" value-ref="frStdAnalyzer" />
</util:map>

```

Error Message Processing

In response to method processing encountering an exception condition, customize a method to display an associated error message in the portlet. To respond to an exception by displaying an error message, the following processing steps must be coded in the relevant method:

- Create the search exception.
- Set the message to be displayed using the `setReportedMessage` method.
- Throw the exception using the class `SearchReportedException`.

Note: An exception's displayed error message can be retrieved by the `getReportedMessage` method.

Error messages are displayed in Jacada WorkSpace.

For example, any entered search string that begins with the wildcard character asterisk [`*`], which the Jacada WorkSpace-provided SearchBridge does not support, results in an error message being displayed.

The Help-On-Hand Portlet

The *Help-On-Hand* portlet conducts background, text searches, using the project's configured document sources, to proactively provide relevant search results to the agent. No agent input is required by the portlet to conduct its searches. Portlet processing can be activated in the project to be triggered at various points during a Jacada WorkSpace session.

This portlet requires the same document sources that are available for use by the [Document Search portlet](#).

The portlet presents its search results in the form of selectable hyperlinks, using the **Combine** display option (an option that is also available to the *Document Search* portlet). A single pane presents only a single page of combined results obtained from all document sources.

Note: When combining the obtained results to prepare a single page of results, search processing selects, in an iterative manner, the top result from each contributing document source until the defined Help-On-Hand results display limit is reached.

For the *Help-On-Hand* to operate in Jacada WorkSpace, the portlet requires administrative activities to be performed using the Jacada Management Console. For the necessary portlet administration, see the *Jacada WorkSpace Administrator Guide*.

By default, the *Help-On-Hand* portlet is available from the toolbar, by clicking .

Development Infrastructure

Jacada WorkSpace provides a coding infrastructure that is used to develop *Help-On-Hand* related processing. This infrastructure makes available the Spring bean `helpOnHandService`. Use this bean, as needed, anywhere in your project code to activate a *Help-On-Hand* text search.

`helpOnHandService` exposes the following methods:

Method	Description
<pre>public void findHelpOn(String keyword);</pre>	Searches, using the project's available SearchBridges for internal and any external document sources, for the provided string and displays any found results in the portlet. This method is asynchronous and does not have any return value.
<pre>public void clearPreviousResults();</pre>	Clears all currently displayed results from the portlet.

The following code is an example of how to implement the `findHelpOn()` method:

```
getHelpOnHandServiceBean().findHelpOn("Handling  
Billing Problems");
```

The Smart Pad Portlet

The Smart Pad enables agents to store and view temporary information that is required during a call, such as information that is received from the caller or extracted from various applications. The application can be an application that is nested in Jacada WorkSpace or an application on the agent's desktop. Use Smart Pad to accomplish the following actions:

- Paste values from a Smart Pad entry into an application.
- Copy selected text from an application and store it in a Smart Pad entry.

Thus the Smart Pad is a central repository for temporary call information, replacing the need to write such information on a piece of paper or in other applications such as Notepad.

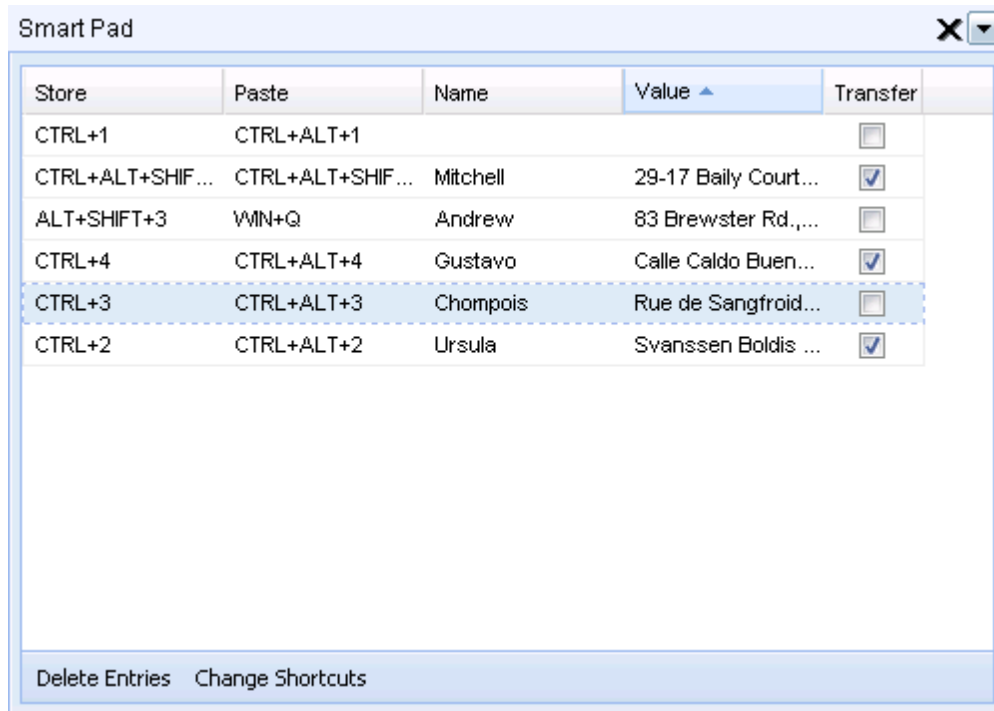
Smart Pad also enables designating entries for transfer and then transferring those entries to the Smart Pad of another agent or supervisor, as part of performing certain types of call transfers to those users.

The Smart Pad portlet includes a table. Each row in the table represents an information entry comprised of a name, a value and a transfer indicator. In addition, each entry has an associated pair of keyboard shortcuts that accomplish the following actions; pasting the entry's value into another application and storing information from another application into the entry's value. Both a Jacada WorkSpace user (agent, supervisor) and the `project` code can populate Smart Pad entries - *Name* and *Value* fields - with information.

Defined Smart Pad shortcuts are made available to the agent for assignment to a Smart Pad entry for performing either a paste or a store action. The agent can change any entry's assigned shortcuts, even when the entry cannot be edited, that is, view-only *Name* and *Value* fields. These shortcuts are enabled when the Smart Pad portlet is included in the layout. If the portlet is included in the toolbar, then the shortcuts are available even when the portlet is not open.

Note: If the Smart Pad portlet is not loaded during logon (`loadOnLogin` is false), even though it is available from the toolbar, the shortcuts only work after the portlet is opened for the first time.

In the following example, the Smart Pad contains completed entries and an additional, empty row into which the agent can type the information of a new, Smart Pad entry.



The following sections describe:

- [Working with the Smart Pad](#)
- [Setting Up the Smart Pad Portlet](#)
- [Transferring Information When Transferring a Call](#)

Working with the Smart Pad

The Smart Pad portlet makes available column checkbox options, column sorting and column resizing to enable agent control of the table display. Each cell in the table provides a tooltip that displays the entire contents of the cell. Columns can be resized without losing the ability to view cell information.

Whenever the *Value* field of an entry is view-only, its *Store* field does not have an assigned shortcut. For the entry, the portlet's *Change Shortcuts* button remains enabled, since the user can always change the *Paste* field's shortcut.

To enable saving space in Jacada WorkSpace, each cell in the Smart Pad has a tooltip that

displays the entire contents of the cell.

The agent can configure the columns displayed in the table, using the check box options displayed below the table.

The Smart Pad portlet does not display the *Transfer* column, should either of the following conditions be true:

- When CTI is not configured in the Jacada WorkSpace database.
- When the Jacada WorkSpace session operates in CTI manual mode.

The Smart Pad portlet does not display both the *Select All* and *Select None* buttons, when the Jacada WorkSpace session operates in CTI manual mode. The *Select All* and *Select None* buttons are used to select or clear, respectively, all entries for transfer. When the Jacada WorkSpace session operates in CTI mode, the Smart Pad portlet always displays the *Select All* and *Select None* buttons and displays the *Transfer* column according to user preferences (*Show/Hide* column display available with the portlet's columns).

The Smart Pad portlet is not displayed in Jacada WorkSpace by default. To add the Smart Pad portlet to the workspace, use *Edit > Layout* in Jacada WorkSpace.

Note: In order for the Smart Pad to function in Jacada WorkSpace, install in your desktop machine the required, signed ActiveX controls that are supplied by the Jacada-provided `ApplicationHolderSetup.msi` file. The Smart Pad should be used with IE 32-bit version only.

Information Transfers with Other Applications

While handling a call, an agent may need to transfer information from Jacada WorkSpace to other applications, or store information from other applications in Jacada WorkSpace. The application may be another application on the desktop, or an application nested in a Jacada WorkSpace tab.

Jacada WorkSpace enables automated transfer of information between Smart Pad and other applications using shortcut keys. This capability is implemented using a Jacada ActiveX control.

Note: 1) To use Smart Pad, make sure that your Internet Explorer browser's security setting *Run ActiveX controls and plug-ins* is set to *Enable*.

2) In order for nested applications and the Smart Pad to function in Jacada WorkSpace, install in your desktop machine the required, signed ActiveX controls that are supplied by the Jacada-provided ApplicationHolderSetup.msi file.

Smart Pad shortcuts containing the *ALT* key require that the agent must always press a keystroke series beginning with the *ALT* key. For example, the defined shortcut of *ALT+SHIFT+7* must be pressed in that order, in order to produce its assigned action (either a paste or a store). This is in contrast to shortcuts that do not contain the *ALT* key. For example, the defined shortcut of *CTRL+SHIFT+W*; pressing its keystroke series in either one of the following orders, produces its assigned action (either a paste or a store):

- *CTRL+SHIFT+W*
- *SHIFT+CTRL+W*

To paste the value of a particular entry into another application:

1. Place your cursor in the desired location in the application.
2. Press the *Paste* field shortcut assigned to the Smart Pad entry whose *Value* field content you want to paste.

The content of the Smart Pad entry's *Value* field is pasted in the cursor location.

To store copied text into a Smart Pad entry:

1. Select the desired text in the application.
2. Press the *Store* field shortcut assigned to the Smart Pad entry whose *Value* field you want to contain the selected text.

The selected text is pasted into the Smart Pad entry's *Value* field.

Note: A Smart Pad entry's *Name* and *Value* fields can each contain a maximum of 440 characters.

If the Smart Pad shortcuts are not available (because the required ActiveX control is not installed), use the following options to transfer information from Smart Pad to other applications:

To select the contents of a Smart Pad entry's Name or Value field:

- Click within the *Name* or *Value* field.

To copy the contents of a Smart Pad entry's Name or Value field to the clipboard:

- Double-click within the *Name* or *Value* cell.

Setting Up the Smart Pad Portlet

Set up the Smart Pad as described in the following sections:

- [Defining a Smart Pad Object](#)
- [Using the Smart Pad](#)
- [Adding Entries to the Smart Pad](#)
- [Presenting an Empty Entry for Agent Input](#)
- [Customizing the Smart Pad Shortcuts](#)

Defining a Smart Pad Object

In your project's server-side code, define a Smart Pad entry object using any of the following constructors:

Constructor	Description
<pre>public SmartPadEntry(String name, String value, String description, boolean disableNameEdit, boolean disableValueEdit, boolean userDefined, boolean clearValueField, boolean transfer, boolean removable)</pre>	Returns a new SmartPadEntry object containing the provided property values.

Constructor	Description
<code>public SmartPadEntry(String name, String value)</code>	Returns a new <code>SmartPadEntry</code> object containing the provided <code>name</code> and <code>value</code> . All other properties contain their default values.
<code>public SmartPadEntry(SmartPadEntry orig)</code>	Returns a copy of the original <code>SmartPadEntry</code> object, excluding its shortcuts and id.

A Smart Pad entry object has the following properties; establish their values using the available, relevant setter method:

name – The content assigned to the entry’s *Name* field. The property’s default value is the empty string.

value – The content assigned to the entry’s *Value* field. The property’s default value is the empty string.

description – A description of the entry, which is not displayed in the portlet.

disableNameEdit – Specifies whether to disable the editing of the entry’s *Name* field. The default value is `true`.

disableValueEdit – Specifies whether to disable the editing of the entry’s *Value* field. The default value is `false`.

userDefined – Specifies whether the entry was automatically added in the Smart Pad as part of default Smart Pad processing. Entries added by the Jacada WorkSpace user have this property value set to `true`. The default value is `false`. See the associated Smart Pad method `public void deleteUserDefinedEntries();` in [Using the Smart Pad](#).

clearValueField – Specifies whether the `clearValueField()` method clears the entry’s *Value* field. For more information, see [Using the Smart Pad](#). The default value is `true`.

transfer – Marks the entry’s *Name* and *Value* fields for transfer in the event of certain types of CTI call transfers. The property’s default value is `false`. For details about transferring Smart Pad entries, see [Transferring Information When Transferring a Call](#).

removable – Marks the entry as one that the Jacada WorkSpace user can delete from the Smart Pad. This argument’s default value is `true`.

id – A unique, integer identifier per Smart Pad entry assigned by Jacada WorkSpace. During runtime, the project must not modify the id.

Using the Smart Pad

Jacada WorkSpace provides the server-side SmartPadManager interface that exposes methods used to develop the project's Smart Pad-related processing. SmartPadManager is a session scoped bean that is accessed using the Spring framework.

Accessing the Smart Pad Object

In your project, access the Smart Pad object using the following code:

1. Define a class in your Spring configuration file and set the `smartPadManager` property, as follows:

```
<bean id="myBeanId" class="myPackage.MyClass" init-method="init">
  <property name="smartPadManager" ref="smartPadManager"/>
</bean>
```

Note: If you do not have a Spring configuration file defined, create one named `applicationContext-operation.xml` and locate it under `WebContent/WEB-INF`.

Add the following `<beans>` element definition to this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/
schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://
www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-
beans-2.0.xsd">
</beans>
```

Add the `<bean>` element within the `<beans>` element.

In the file `WebContent/WEB-INF/web.xml`, find the `contextConfigLocation` `context-param`. This is where all Spring xml files are listed. Add the following entry to the list: `/WEB-INF/applicationContext-operation.xml`.

2. Create a new Java class for the bean defined in the Spring configuration file, `myPackage.MyClass`, and create a setter method to the `SmartPadManager` interface, as follows:

```
public class MyClass {

    private SmartPadManager smartPadManager;
    ..
    ..
    public void setSmartPadManager(SmartPadManager smartPadManager) {
        this.smartPadManager = smartPadManager;
    }
}
```

Smart Pad Methods

The SmartPadManager interface exposes the following methods for working with the Smart Pad:

Method	Description
<pre>public boolean hasEmptyEntry();</pre>	Checks whether the Smart Pad contains an empty entry, that is, an entry with empty <i>Name</i> and <i>Value</i> fields. The method returns <i>True</i> when Smart Pad contains an empty entry.
<pre>public boolean isHideColumns();</pre>	Returns the value of the <i>Hide Columns</i> setting defined in the Jacada Management Console.
<pre>public boolean isSortColumns();</pre>	Returns the value of the <i>Sort Columns</i> setting defined in the Jacada Management Console.
<pre>public Collection<SmartPadEntry> getAllEntries();</pre>	Returns a collection of all Smart Pad entries.
<pre>public int getMaximumEntries();</pre>	Returns the value of the <i>Maximum Entries</i> setting defined in the Jacada Management Console.
<pre>public List<SmartPadShortcut> getDefinedShorcuts();</pre>	Returns the list of the Smart Pad shortcut keys that are defined in the <code>WebContent/USER/portlets/smartPad/smartPad.xml</code> file.
<pre>public SmartPadEmptyEntryPosition getEmptyEntryPosition();</pre>	Returns the value of the <i>Empty Entry Position</i> setting defined in the Jacada Management Console.

Method	Description
<code>public SmartPadEntry addEmptyEntry();</code>	Adds a new, empty Smart Pad entry with its <code>userDefined</code> property set to <code>true</code> . The method does not check whether the added entry causes Smart Pad to exceed the <i>Maximum Entries</i> setting.
<code>public SmartPadEntry getEntry(Long id);</code>	Returns the SmartPad entry identified by provided <code>id</code> . If such an entry does not exist, the method returns null.
<code>public SmartPadEntry getNewEntry();</code>	Returns a new, empty SmartPad entry with its properties set to their default values. An empty entry is an entry with empty <i>Name</i> and <i>Value</i> properties.
<code>public String getValue(Long id);</code>	Returns the information contained in the <i>Value</i> field of the SmartPad entry identified by provided <code>id</code> . If such an entry does not exist, the method returns null.
<code>public void clearValueFields();</code>	Clears the <i>Value</i> field of all Smart Pad entries that have their <code>clearValueField</code> property set to <code>true</code>. Use case: Use this method between calls, either at the end or at the beginning of a call script.
<code>public void deleteAllEntries (Collection<Long> ids);</code>	Deletes all the Smart Pad entries that have their <code>id</code> provided in the collection of <code>ids</code> .
<code>public void deleteEntry(Long id);</code>	Deletes the SmartPad entry identified by the provided <code>id</code> .

Method	Description
<code>public void deleteUserDefinedEntries();</code>	Deletes all Smart Pad entries that have their <code>userDefined</code> property set to <code>true</code>. Use case: use method between calls, either at the end or at the beginning of a call script.
<code>public void refreshSmartPadUI();</code>	Refreshes the Smart Pad in Jacada WorkSpace. Use this method either after each change or after a series of changes to display these changes in the <i>Smart Pad</i> portlet.
<code>public void setTransfer(Long id, boolean transfer);</code>	Sets the transfer property of the SmartPad entry, identified by provided <code>id</code> , to the provided transfer.
<code>public void setValue(Long id, String value);</code>	Sets the <i>Value</i> field of the SmartPad entry, identified by provided <code>id</code> , to the provided value.
<code>public void updateEntries(List<SmartPadEntry> newEntries);</code>	Updates all the Smart Pad entries in the list, specified by their provided <code>newEntries.id</code> .
<code>public void updateEntry(SmartPadEntry newEntry);</code>	Updates the Smart Pad entry, specified by the provided <code>newEntry.id</code> .
<code>public void addEntry(SmartPadEntry entry);</code>	Adds a new Smart Pad entry. If an entry with <code>entry.id</code> already exists, it is replaced it with the new entry.

Adding Entries to the Smart Pad

By default, the Smart Pad presents a single entry in which its *Store* and *Paste* fields are each populated with a shortcut and its *Name* and *Value* fields are empty. The following section describes how to add a Smart Pad entry and then update its values.

Updating the Smart Pad

Use the Smart Pad methods to write code that manages Smart Pad entries.

Each project is responsible for handling Smart Pad clean-up, that is the deletion of added entries (rows) from the Smart Pad. It is especially important to incorporate Smart Pad entry clean-up processing in project code given the fact that Smart Pad entries can be transferred from one agent's Smart Pad to another agent's Smart Pad as part of certain types of call transfers.

Note: The Smart Pad maximum number of entries limitation is not imposed upon entries added as part of a call transfer or due to project code. The setting limits the number of entries that can be added by the agent.

The following code illustrates how to add in the project a Smart Pad entry containing the customer's name:

```
Long customerNameEntryId;
protected void addCustomerNameFromSessionVars() {

    SmartPadEntry entry = smartPadManager.getNewEntry();
    customerNameEntryId = entry.getId();
    entry.setName("CustName");
    entry.setValue(globalApp.sessionVars.getCustomerName());

    //required to update the database
    smartPadManager.updateEntry(entry);

    //required to update the client (the Smart Pad in Jacada WorkSpace)
    smartPadManager.refreshSmartPadUI();
}
```

You are required to save the Smart Pad entry id to be able to refer to it during any future, project processing.

The following code illustrates how to update the an existing Smart Pad entry with a customer name stored in a session variable. This code refers to the Smart Pad entry id that had to be previously saved when the entry was added:

```
protected void setCustomerNameFromSessionVars() {

    //no need to run updateEntry because smartPadManager method is used
    //to update entry's value

    smartPadManager.setValue(customerNameEntryId,globalApp.sessionVars.getCustomerName());

    //required to update the client (the Smart Pad in Jacada WorkSpace)
    smartPadManager.refreshSmartPadUI();
}
```

In both of the previous code examples, the `refreshSmartPadUI` method is used at the end to update the Smart Pad in Jacada WorkSpace with the changes made.

Using the Smart Pad in a Call-Handling Script

The contents of the Smart Pad can be defined in the call-handling script to automatically display information, as soon as the information is available during script execution. For example, you can write code to display a customer number or credit card details, which can then be pasted from the Smart Pad into other applications, as necessary. In addition, the agent can type information in the Smart Pad during a call.

Note: To use the Smart Pad, it is necessary to have write access for the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT
\CurrentVersion\

Windows\AppInit_DLLs
```

Presenting an Empty Entry for Agent Input

The Smart Pad can present an empty entry for agent input. Once the agent populates this entry with information, a new empty entry is automatically added to the Smart Pad. By default, the Smart Pad always presents an empty entry. Administer the Smart Pad's presentation of an empty entry using the Jacada Management Console. For configuration specifics, see the *Jacada WorkSpace Administrator Guide*.

Customizing the Smart Pad Shortcuts

A Smart Pad shortcut is a series of keystrokes that is used to perform either one of the following actions:

- Paste values from a Smart Pad entry into an application.
- Copy selected text from an application and store it in a Smart Pad entry.

Each Smart Pad shortcut is comprised of a shortcut key and one or more than one shortcut modifier key. Defined Smart Pad shortcuts are made available to the agent in the *Smart Pad* portlet for assignment to a Smart Pad entry for performing either a paste or a store action.

The Jacada WorkSpace-provided set of Smart Pad shortcut keys are defined in the file `<program_name>/WebContent/USER/portlets/smartPad/smartPad.xml`. In this file, define custom Smart Pad shortcut keys too.

To define a shortcut:

- In the **smartPad.xml** file, define each shortcut using its own `shortcut` tag. In the following example, two custom shortcuts are defined:

```
<smartPad xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="..\..\..\SYSTEM\resources\xmlValidators
\smartPad.xsd">
  <shortcuts>
    ..
    <shortcut key="Z" alt="true" ctrl="false" shift="false"
      win="true" display="Alt+WIN+Z"/>

    <shortcut key="p" alt="false" ctrl="true" shift="true"
      win="false" display="Shift+Ctrl+p"/>
  </shortcuts>
  ..
</smartPad>
```

where:

key – required attribute that defines the shortcut key.

alt / ctrl / shift / win – required attributes that define each of the shortcut modifier keys.

Specify `true` for the shortcut modifier keys to be used in the shortcut and `false` for all the rest of the modifiers.

Shortcuts that define the **alt** modifier key to be true, their keystroke series must always begin with the *ALT* key being pressed. For example, the defined shortcut of `key="7" alt="true"`

`ctrl="false" shift="true" win="false"` must be pressed in the order *ALT+SHIFT+7* in order to produce its assigned action (either a paste or a store).

display – optional attribute that defines the text that is displayed in the Smart Pad for the defined shortcut. If display is not defined, the default text display is used for the shortcut. For example, the defined shortcut of `key="g" alt="false" ctrl="false" shift="true" win="true"` has the default text display of *SHIFT+WIN+g*.

Note: When defining a Smart Pad shortcut, make sure that the shortcut is not used by another application, since the application shortcut takes precedence over the Smart Pad shortcut.

Supported Keys for Defining Shortcuts

The following keys can be used to define a Smart Pad shortcut:

Note: When defining shortcut keys, make sure the shortcut keys do not interfere with the functionality and the shortcuts of the external applications used by the agent. For a list of Windows shortcuts, see <http://support.microsoft.com/kb/126449>.

- *[Backspace]*, *[Tab]*, *[Enter]*, *[Esc]* and *[Space]*
- *[Page Up]*, *[Page Down]*, *[Home]*, *[End]*, *[Insert]* and *[Delete]*
- *[LeftArrow]*, *[UpArrow]*, *[RightArrow]* and *[DownArrow]*
- *[0]* through *[9]* and *[a]* through *[z]* (case insensitive)
- *[NumPad *]*, *[NumPad +]*, *[NumPad -]*, *[NumPad .]* and *[NumPad /]*
- *[F1]* through *[F12]*
- *[+]*, *[-]*, *[,]* and *[.]*

Transferring Information When Transferring a Call

Jacada WorkSpace provides the ability to transfer information collected during a call when transferring a call from one agent to another agent or supervisor. As part of an agent transferring a call, all designated portlet entries are transferred by Jacada WorkSpace to the Jacada WorkSpace user receiving the transferred call. Transferring data is supported for the following CTI providers and call transfer types:

CTI Provider	Call Transfer Type
Avaya Inc. – Application Enablement Service (AES)	• Handshake transfer • Warm transfer • Conference + transfer • Conference + join
Genesys Telecommunications Laboratories Inc. – Genesys	Note: Even when the call transfer is a two step process – Warm, Conference+transfer and Conference+join – Jacada WorkSpace still only transfers information marked for transfer as of the start of the call transfer process.

The ability to transfer data when transferring a call is incorporated in the Smart Pad portlet. Smart Pad portlet entries can be designated as transferable by the Jacada WorkSpace user and within the project code. Additionally, other Jacada WorkSpace portlets or tabs can register for the call transfer event and then, upon the occurrence of this event, retrieve all Smart Pad portlet entries marked for transfer and display them.

When a Jacada WorkSpace user adds a new, Smart Pad portlet entry, the entry, by default, is not designated for transfer.

Note: The Smart Pad maximum number of entries limitation is not imposed upon entries added as part of a call transfer.

Users, working in *CTI* mode in JacadaWorkSpace, mark Smart Pad entries for transfer, using the available *Transfer* column checkbox for the applicable entry in the portlet.

To designate a Smart Pad portlet entry for transfer:

- Check the *Transfer* column box of the applicable row entry.

The Smart Pad portlet, receiving the entry transfer, functions as follows:

- For the entire duration of the transferred call, the portlet highlights all transferred entries in a different color.
- The portlet displays each transferred entry with its *Transfer* cell checked.
- Even if the portlet is closed at call transfer time, all transferred entries display in the Smart Pad, upon opening of the portlet.

Creating an Entry for Transfer

In your code, when creating a new Smart Pad entry, the entry object constructor incorporates the following property:

```
boolean transfer
```

Use this property to designate the entry being added as one to be transferred to the Smart Pad portlet of a receiving Jacada WorkSpace user, in the event of Jacada WorkSpace processing of either a handshake, a warm or a conference transfer. In the following code example, a new Smart Pad entry is added with its `name` property set to `Store This Info`, its `value` property set to be empty and its `transfer` property set to `true`:

```
SmartPadEntry entry = new SmartPadEntry("Store  
This Info", "");  
entry.setTransfer(true);  
smartPadManager.addEntry(entry);
```

Users, working in *CTI* mode in JacadaWorkSpace, mark Smart Pad entries for transfer, using the available *Transfer* column checkbox for the applicable entry in the portlet.

Manipulating Existing Entries

In your code, manipulate the `transfer` property of existing Smart Pad entries using the following Jacada WorkSpace method:

```
void setTransfer(Long id, boolean transfer)
```

This method sets the `transfer` property value of an existing entry, identified by the supplied `id`. The second argument is either `true` to designate the entry for transfer or `false` to designate the entry as not for transfer.

In the following code example, an existing Smart Pad entry is designated as transferable:

```
smartPadManager.setTransfer(id, true);
```

Entry Retrieval by Other Tabs or Portlets

In addition to the Smart Pad portlet, other Jacada WorkSpace tabs and portlets can receive transferred Smart Pad entries. To accomplish this, each tab or portlet must be coded to handle the following:

- Register to be notified by the CTI notification manager about the occurrence of the event `PROJECT_NOTIFICATIONS.TRANSFERABLE_DATA_RECEIVED`.

This event contains the event object `TransferredDataReceivedEvent` that provides all relevant transferred information.

The following is a code example of event registration:

```
Observer observer = new Observer() {
    public void update(Observable o, Object arg) {
        TransferredDataReceivedEvent event =
            (TransferredDataReceivedEvent) arg;
        //Provide your event handling code here, for
        example:

        (MyExample.this).gotTransferredDataEvent(event);
    }
};
this.getCTIManager().getCtiNotificationManager()
    .addObserver(observer,
        PROJECT_NOTIFICATIONS.TRANSFERABLE_DATA_RECEIVE
        D);
```

- Receive the event.
- Retrieve the transferred data associated with the event, those being the Smart Pad entries. Data retrieval is accomplished using the following method:

```
public Map<String, SmartPadEntry>
getDataReceived()
```

Map returns a mapping of `String` to `SmartPadEntry` entries. Each `String` contains the database key to its `SmartPadEntry`. Use map-specific processing methods to then obtain, from the `SmartPadEntry`, the name and value information that was transferred.

Creating Dashboards Using the Dynamic Views Designer

The Dynamic Views Designer enables you to easily design and build any combination of visual data representations in any workspace, tab or portlet. The feature allows you to access WorkSpace data sources, widgets and environment variables, and customize their display. Using Dynamic Views, you can design your own Inline (JavaScript) and HTML wizards, to match the exact and unique requirements of your call center.

Creation and modification of Dynamic Views is done in a real-time iterative development mode. Any changes you make can be viewed immediately. Rapid and precise customization is accomplished using drag-and-drop of both pre-designed and custom widgets.

The following sections describe:

- [Installation of Dynamic Views](#)
- [Accessing the Dynamic Views Designer](#)
- [Dynamic Views Developer Interface](#)
- [Creating a New Dashboard](#)
- [Deploying a Dashboard in Jacada WorkSpace](#)
- [Functional Widgets](#)
- [Grouped Widgets](#)
- [Jacada Widgets](#)
- [Coordinating Dashboards Among Multiple Developers](#)
- [Creating a New Widget and Making It Available](#)
- [Programming a Widget Repository](#)
- [Creating Data Sources](#)
- [Working With Input Forms](#)

- [Managing Environment Variables for Dynamic Views](#)
- [Customizing Dashboards Programatically](#)

Installation of Dynamic Views

Dynamic Views development capability is automatically installed in the development environment when the product key supplied with the installation CD supports Dynamic Views. After a Dashboard is developed, it is deployed to Jacada WorkSpace via the Jacada Management Console as a tab or portlet. (For details, refer to [Deploying a Dashboard in Jacada WorkSpace](#).)

Accessing the Dynamic Views Designer

The Dynamic Views Designer and Jacada WorkSpace run in the same application servers. Access the Dynamic Views Designer using one of the following methods:

- **Via the WorkSpace IDE:** This method allows you to access the Dynamic Views Designer directly from the development environment, when using an Apache Tomcat server.
- **Via Internet Explorer:** Use a Microsoft Internet Explorer browser and supply the applicable URL. This access method is available for both cluster and single server configurations. This method provides both local and remote access of the Dynamic Views Designer.

Note: The recommended access method is via the WorkSpace IDE.

Opening the Dynamic Views Designer automatically creates a Jacada WorkSpace session. If you log out of the Jacada WorkSpace session, some of the widgets in the Dynamic Views Designer will no longer be available.

To access the Dynamic Views Designer from the development environment:

1. From the Eclipse IDE, right-click on the relevant project, and select *Jacada WorkSpace> Jacada Dynamic Views*.
The Jacada WorkSpace *Login* page appears in the Internet Explorer browser.
2. Log in to WorkSpace.
The Dynamic Views Designer opens.

To access the Dynamic Views Designer via Internet Explorer:

1. In the browser's URL field, enter the following:

`http://<host>:<Port>/<Jacada Workspace project name>/dynamicviews`

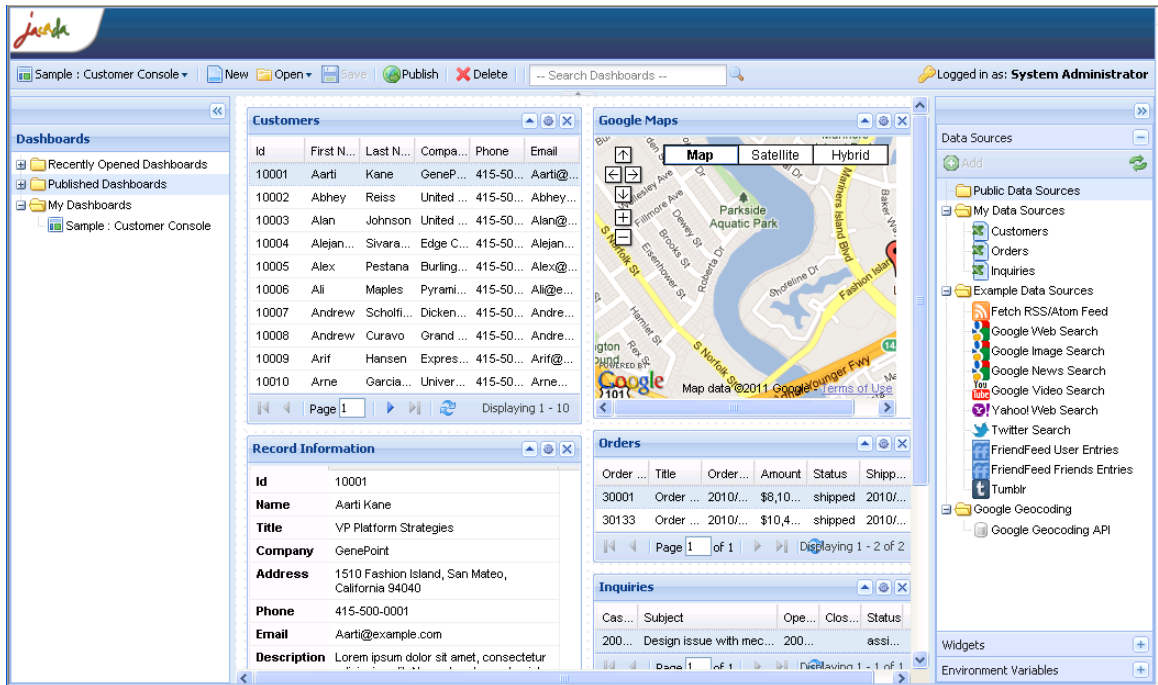
where:

- *host* is either the IP or machine name of the machine hosting the Dynamic Views Designer.
 - In a cluster configuration, *Port* is the web container default host port, `WC_defaulthost`, of any application server in the cluster.
In a single server configuration, *Port* is the HTTP transport port of the application server.
2. In the Dynamic Views *Login* dialog, enter the *user name* and *password* established for the Jacada Dynamic Views administrator.

Dynamic Views Developer Interface

The Dynamic Views interface is divided into the following main sections:

- Work Area: The main area, in the center of the page, where Dashboards are displayed
- [Menu Bar](#)
- [Dashboards Panel](#)
- [Resources Panel](#)



Menu Bar

The menu bar, at the top of the page, includes the following options:

Item	Description
Dashboard Name	Your development mode name for the Dashboard appears at the left of the menu bar.
Dashboard Name menu	Click the Dashboard name and select one of the available options. For details, refer to Dashboard Name Menu .
New	Opens a new Dynamic Views Dashboard.

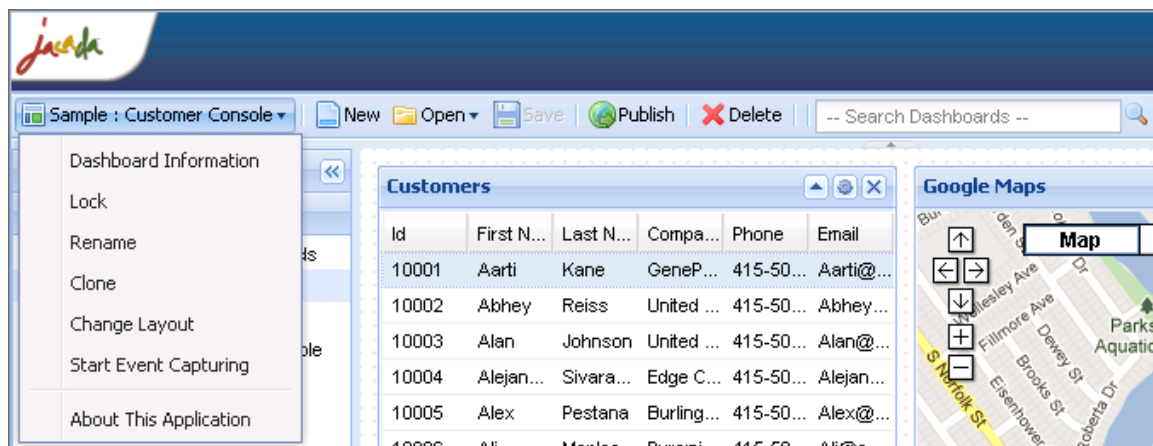
Item	Description
Open	Opens an existing Dynamic Views Dashboard.
Save	Saves the configuration of the currently visible Dashboard design.
Publish	Not relevant to Jacada WorkSpace Developer environment. To deploy a Dashboard for the first time, configure it to a tab or portlet using the Jacada Management Console.
Delete	Deletes the currently visible Dashboard.

Item	Description
NowCapturingEvent icon	TheNowCapturingEvent pop-up - is displayed whenever an event is fired. It can be dragged to another field in the view or another view, enabling you to drop an object attached to the event in the required destination. In order to activate the NowCapturingEvent pop-up, one of the following conditions must exist: *Some widget or data source is in edit configuration mode OR * Event capturing has been turned on (Start Event Capturing) via the drop-down context list attached to the Dashboard name.

Item	Description
Dashboards Search Box	Enables location of project Dashboards whose names include the search string. Clicking a Dashboard name in the resulting drop-down opens it in the Jacada Dynamic Views Designer.

Dashboard Name Menu

The Dashboard Name menu is located at the upper left corner of the page.



Featured options are listed in the following table:

Item	Description
Dashboard Information	Selecting Dashboard Information displays the Dashboard ID, Dashboard Name and Owner of the Dashboard. The Dashboard ID, a long hexadecimal, can be used to view the Dashboard as a web page in runtime and development mode. To view the Dashboard in runtime mode, add the Dashboard ID to the

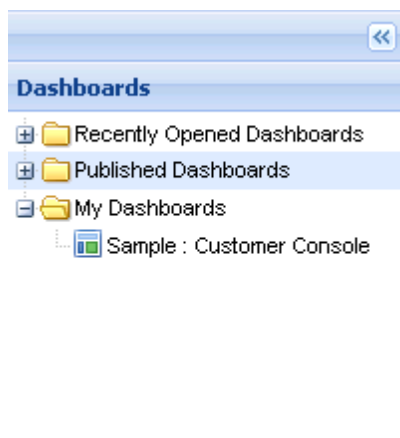
Item	Description
	string "Http://localhost:8080/dynamicviews/view/#". To view the Dashboard in development mode, add the Dashboard ID to the string "http://localhost:8080/dynamicviews/#".
Lock	Puts the Dashboard in a non-edit mode. Note: This parameter is relevant to development mode only. (Agents always receive WorkSpace in non-edit mode.)
Rename	Renames the Dashboard.
Clone	Clones the current Dashboard.
Change Layout	Enables you to change your Dashboard Layout selection.

Dashboards Panel

The collapsible Dashboards Panel, located on the left side of the screen, allows you to open an existing Dashboard for editing.

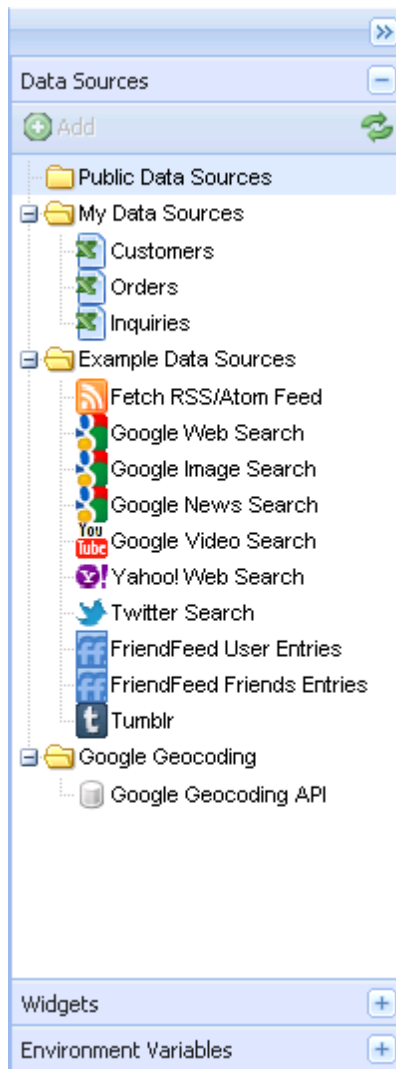
Note: You can only edit one Dashboard at a time. You will be asked to save your work or abandon your changes for any Dashboard you may be currently editing when you attempt to open a different Dashboard.

Dashboards can also be opened for editing using the Dynamic Views menu bar. Click *Open > Open My Dashboard*. Select a Dashboard from the list, and click *Open*.



Resources Panel

The collapsible Resources Panel, located on the right side of the screen, displays available data sources and widgets you can drag into your work area to configure your Dashboard.



Available resources include:

- **Data Sources:** Resources that are linked to specified Web sources or databases. Data sources can be out-of-the-box, or customized.
- **Widgets:** Can be unconfigured or customized. Unconfigured widgets that are available include HTML, Rich Text, Simple Text Input, Calendar, Inline Frame, Grid Table, List, Image Tile, Record Information, Select Box, Chart, and Input Form.
- **Environment Variables:** Fields that contain information associated with a user's account, such as user name, user email, organization name, etc.

Handling Empty Data Sources

If the linked data source is empty (e.g., currently has no data, or gets its data from another source), the Data Source on the Dashboard cannot be opened, and the Refresh icon (at the bottom of the Data Resource) is inaccessible.

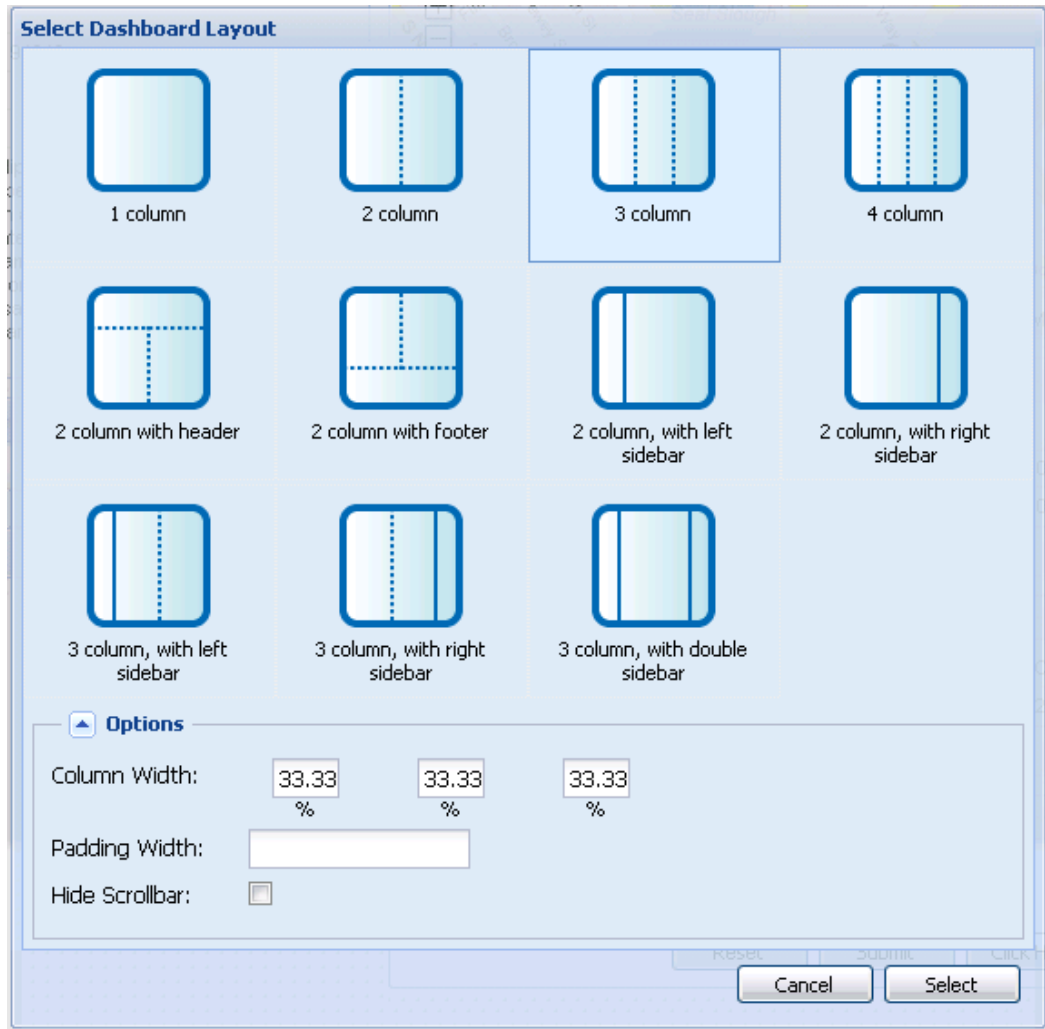
To prevent this issue, when configuring the Data Source, from the

Creating a New Dashboard

The following procedure explains how to build a new Dashboard.

To create a new Dashboard:

1. In the Eclipse IDE, right-click on the relevant project_name, and select *Jacada Workspace > Jacada Dynamic Views > Designer*.
The Jacada WorkSpace *Login* screen appears.
2. Log in to Jacada WorkSpace, using the relevant credentials.
The Dynamic Views Designer appears.
3. From the menu bar, click *New*.
The *Select Dashboard Layout* window appears.



4. Select one of the displayed layouts. The *Select* button (in the lower right corner of the window) is enabled.
5. At the bottom of the window, expand the *Options* frame, and configure the following options, as required:
 - **Column Width:** By default, multiple columns are evenly divided across the width of the Dashboard. You can change the default values to customize your Dashboard layout. Column widths for layouts without sidebars are expressed in percentage (of total

Dashboard width). Column widths for layouts with sidebars are expressed in pixels.

- **Padding Width:** By default, there is no extra space between the widgets in a Dashboard. This option enables you to specify a value (in pixels) for padding width between the widgets on the Dashboard.
- **Hide Scrollbar:** When the checkbox is selected, a scrollbar is not rendered in the Dashboard, even when the Dashboard overflows the screen content area.

6. Click *Select*.

The Dashboard is created and displayed. Widgets can now be added to the Dashboard.

For an example, refer to [Adding a Data Source Widget](#).

7. To save your Dashboard, from the menu bar, click *Save*.

Adding a Data Source Widget

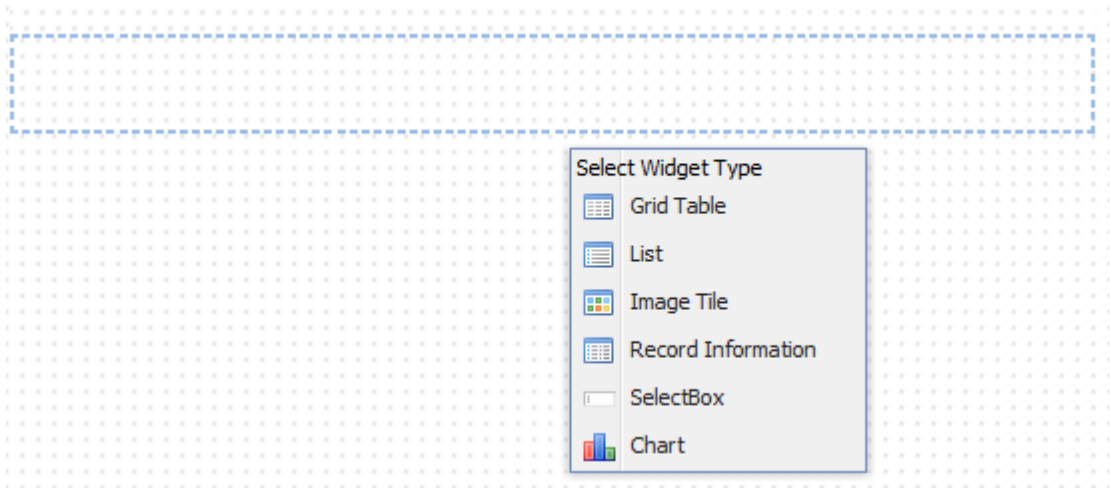
The following procedure explains how to add a Data Source widget to a new Dashboard.

To add a Data Source widget to a new Dashboard:

1. From the Resources Panel (on the right side of the screen), select the *Data Sources* frame, and drag one of the available data source widgets to the required position.

Note: When the resource can be displayed in more than one manner, an empty outline is displayed conforming to the shape of the layout section, and the *Select Widget Type* menu is displayed below it. To proceed, continue to Step 2.

2. If the *Select Widget Type* menu is displayed, click on one of the listed widget types.



The display type is set, and the widget is deployed in edit mode.

3. Using the available tabs, edit the configuration properties of the widget, and then click **OK**.
The configuration properties of the widget are saved in the system.

Note: To re-edit the configuration, in the upper right area of the widget, click .

4. If desired, use the settings tabs of the widget to customize display customization options. (For details, refer to [Customization Options](#).)
5. To save the Dashboard changes, at the top of the Dynamic Views Designer, click **Save**.

Customization Options

The following sections describe display customization options for a Dashboard, according to widget type:

- [Customization for All Widget Types](#)
- [Customization for Grid Tables](#)
- [Customization for Image Tiles](#)
- [Customization for Lists](#)
- [Customization for Charts](#)

- [Customization of Widget Display Status](#)
- [Customization of Error Messages](#)

Customization for All Widget Types

The following customization options are relevant to all widget types:

Tab	Item	Description
Properties	Title	Title for the grid table widget.
	<u>Options:</u>	
	Widget Height	Height (in pixels) of the entire widget. Note: Adjusting the height in pixels does not automatically change the number of rows. To change the number of rows, select the <i>Detail</i> tab.
	Show Header	If unchecked, the widget header is hidden. Note: Relevant for locked Dashboards only.
	Show Collapse Button	The <i>Collapse</i> button allows the agent to hide the body of the widget (showing the header only), and to restore the widget as required.
	Auto-Reload Interval (min)	Number of minutes from one data refresh to the next.
Detail	Show Reload Button	Allows agent to initiate data refresh.
	Num of Rows	Note: The number of rows must be manually coordinated with the widget height. If the widget height cannot accommodate the number of rows specified, all or some of the data may not be visible.

Tab	Item	Description
	Check Box Options	Use the check boxes to determine: • Presence of horizontal scroll • Table header display • Paging toolbar • Default selection of the first row of the grid
Filter	Various	Filter designer using available fields of the data resource.
Summary	Property Source	Drag required summary fields from this pane to the <i>Summary Fields</i> pane.
	Summary Fields	Drag fields from the property source to this pane to edit their appearance as summary fields (design field label). To edit/delete an item, use the <i>Edit</i> and <i>Delete</i> icons at the bottom of the pane.
Sort	Column List	Select a column from the drop-down list.
	Direction List	Select <i>Ascending</i> or <i>Descending</i> .
	Add button	Adds another column/direction combination to sort by.
	Delete button	Deletes selected sort definition segments.
Image (Not available with Grid Table Display)		

Customization for Grid Tables

The following customization options apply to grid table widgets only:

Tab	Item	Description
Display Data	Property Source	Pane containing all available items from the data source.
	Columns to Display	Displays all selected items plus designer defined fix-value items.
	Plus Button	Displays <i>Field Header</i> and <i>Displaying Value</i> fields for static data that is permanently displayed.
	Minus Button	Removes selected items from the <i>Columns to Display</i> pane.

Setting Styles to Grid Cells

Dynamic Views supports setting customized styling (e.g., font color, font style, image URLs, etc.) to table grid cells. The style settings are selectively applied in runtime, according to specified conditions.

Examples

In the following example, style settings are applied to cells in the *Balance* column, according to the following conditions:

- **Value greater than 1 million:** Bold font
- **Negative value:** Red font color

TestTable		
Year ▼	Balance	
2012	1,412,980	
2011	219,906	
2010	-218,080	
2009	-198,010	
2008	830,020	
2007	4,114,480	
2006	3,864,230	
2005	3,860,940	
2004	1,810,424	
2003	980,672	
Page 1 of 2		Displaying 1 - 10 of 20

In the following example, different icons are added to rows, according to the value in the *Status* column:

- **Completed:** Check icon
- **Expired:** Stop icon
- **In Progress:** No icon is displayed

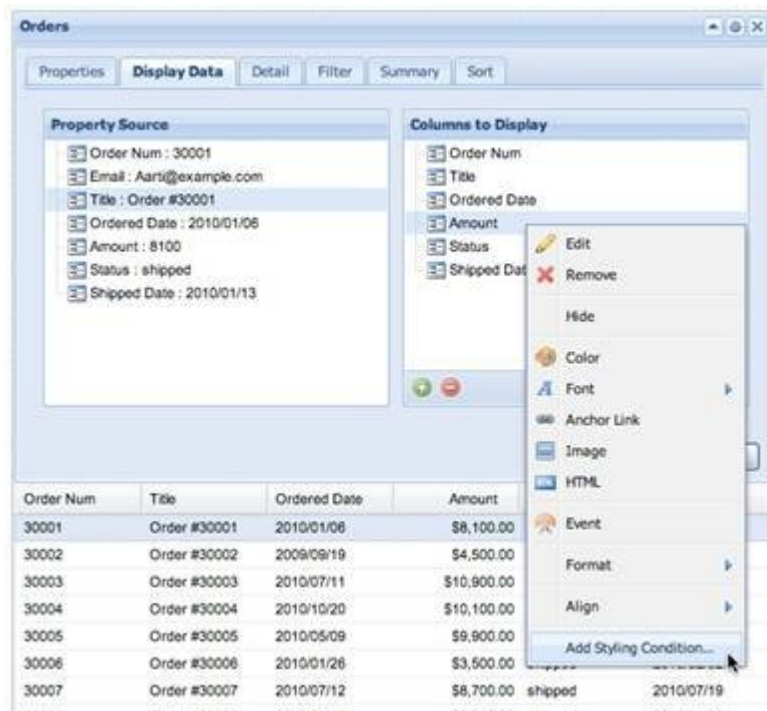
Tasks			
Subject	Due Date ▲	Status	Icon
Visit to customer's site	2012/11/09	Completed	✓
Respond Email	2012/11/09	Completed	✓
Order office supply	2012/11/09	In Progress	
Contribute to local society	2012/11/09	In Progress	
Contract reseller agreement	2012/11/09	Expired	●
Analyze web traffic	2012/11/09	In Progress	
Write technical document	2012/11/09	In Progress	
Confirm the letter	2012/11/09	Completed	✓
Page 1 of 1			Displaying 1 - 8 of 8

Implementation

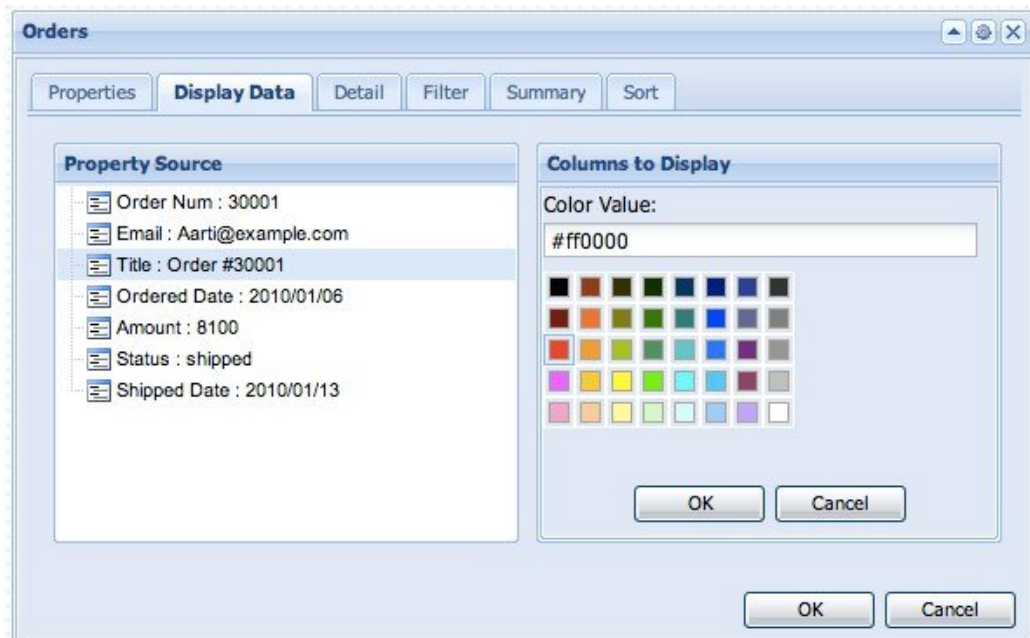
Implementing customized grid styling involves specifying the style, and then defining the styling condition.

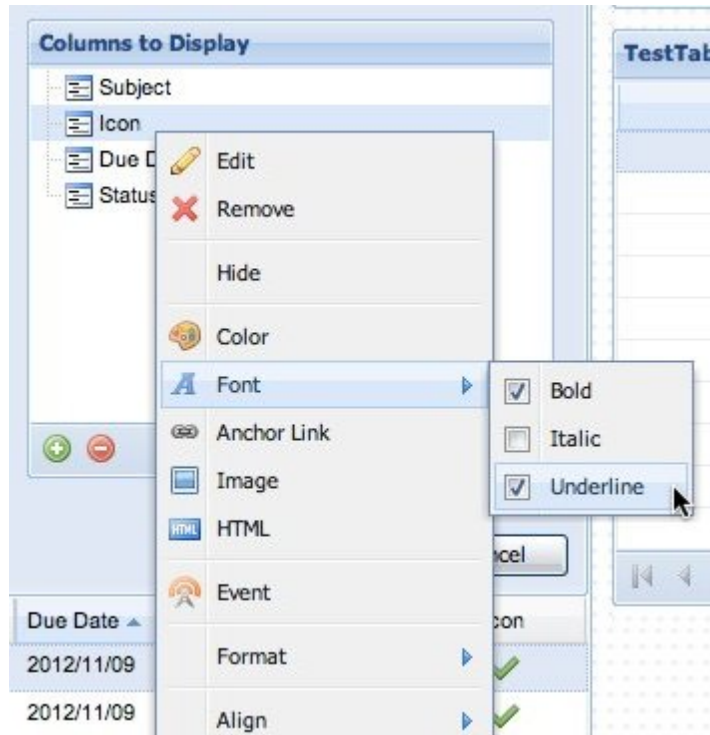
To implement customized styling:

1. From the *Display Data* tab of the grid, right-click on the relevant column, and define the styling as required.

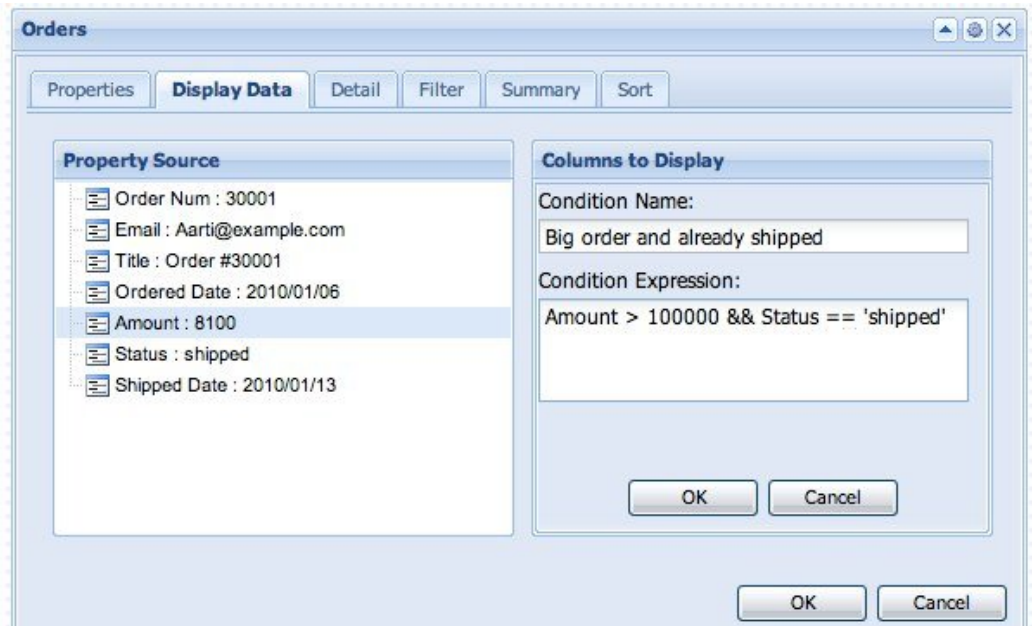


For example, selecting *Color* enables you to input or select a color. Selecting *Font* enables you to specify bold, italic, or underline.





2. Select *Add Styling Condition*, and enter a condition name and JavaScript expression in the appropriate fields. Variables in the condition expression can be selected from the *Property Source* frame, and inserted into the condition expression by drag-and-drop.



3. Click OK. The styling condition is saved, and its name is listed in the *Styling Conditions* cascaded menu.



Customization for Image Tiles

The following customization options apply to image tile widgets only:

Tab	Item	Description
Display Data	Image URL	Formula for image URL which may be built using one or more of the Property Source field expressions, e.g., <code>\${this["Last Name"]}</code>. TIP: To view property source expressions, drag them into an available text box. Use Notepad (or similar application) to build the URL expression, and then copy the completed expression into the <i>Image URL</i> text box.
	Caption	Enter structure of caption. You can build the caption from text segments and <i>Property Source</i> pane field expressions. TIP: To view property source expressions, drag them into an available text box. Use Notepad (or similar application) to build the expression, and then copy the completed expression into the <i>Caption</i> text box.

Customization for Lists

The following customization options apply to list widget types only:

Tab	Item	Description
Display Data	Title	Link Text.
	URL	URL destination associated with Title (link text).

Tab	Item	Description
	Summary	Summary Information to display. Example: \${this["Last Name"]} -- \${Street} -- \${Phone}

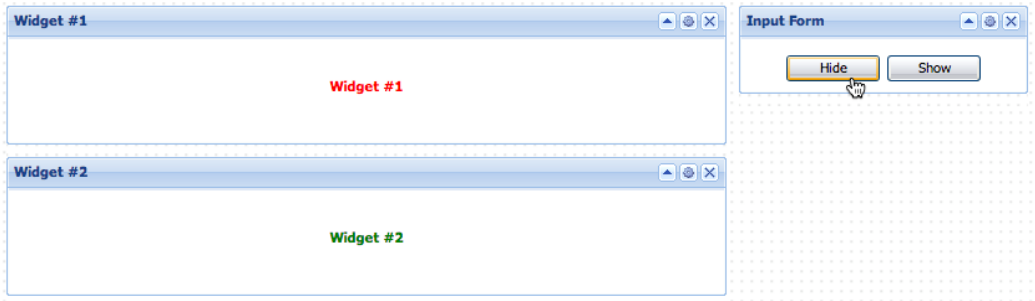
Customization for Charts

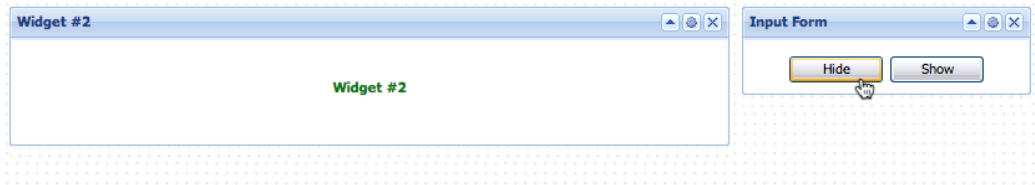
The following customization options apply to chart widget types only:

Tab	Item	Description
Display Data	Label to Display	TIP: To view property source expressions, drag them into an available text box. Use Notepad (or similar application) to build the expression, and then copy the completed expression into the <i>Caption</i> text box.
	2nd-level Group Label	
	Value to Display	

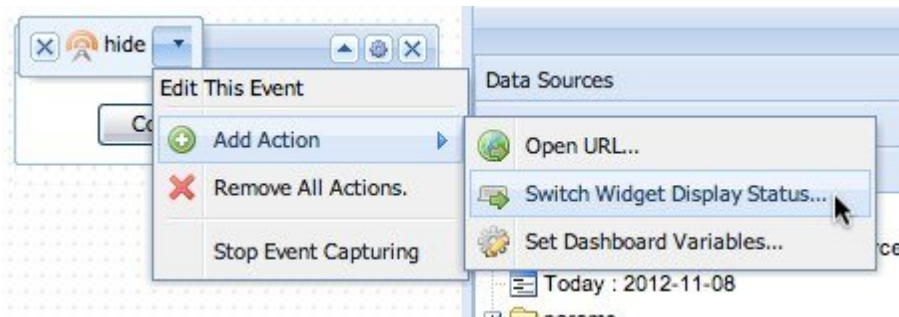
Customization of Widget Display Status

Dynamic Views supports changing the display status (expand/collapse/show/hide/minimize/maximize) of widgets upon the firing of any event in the Dashboard. The following figures show an example of hiding Widget #1 when the *Hide* button is clicked.

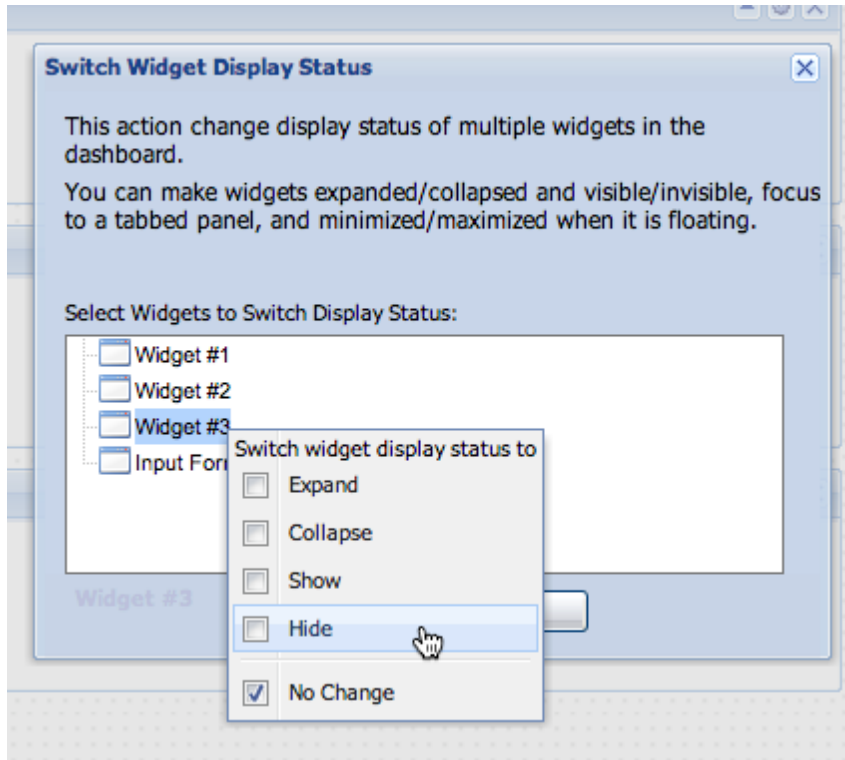




To bind widget display status to an event, select *Switch Widget Display Status* from the Action list.



The *Switch Widget Display Status* dialog enables you to set the display status on an event for each widget in the Dashboard. Display status options include *Expand/Collapse*, *Show/Hide*, *Focus* (for tab grouped widgets), and *Minimize/Maximize* (for floating widgets). The *No Change* option maintains a widget's current display status.



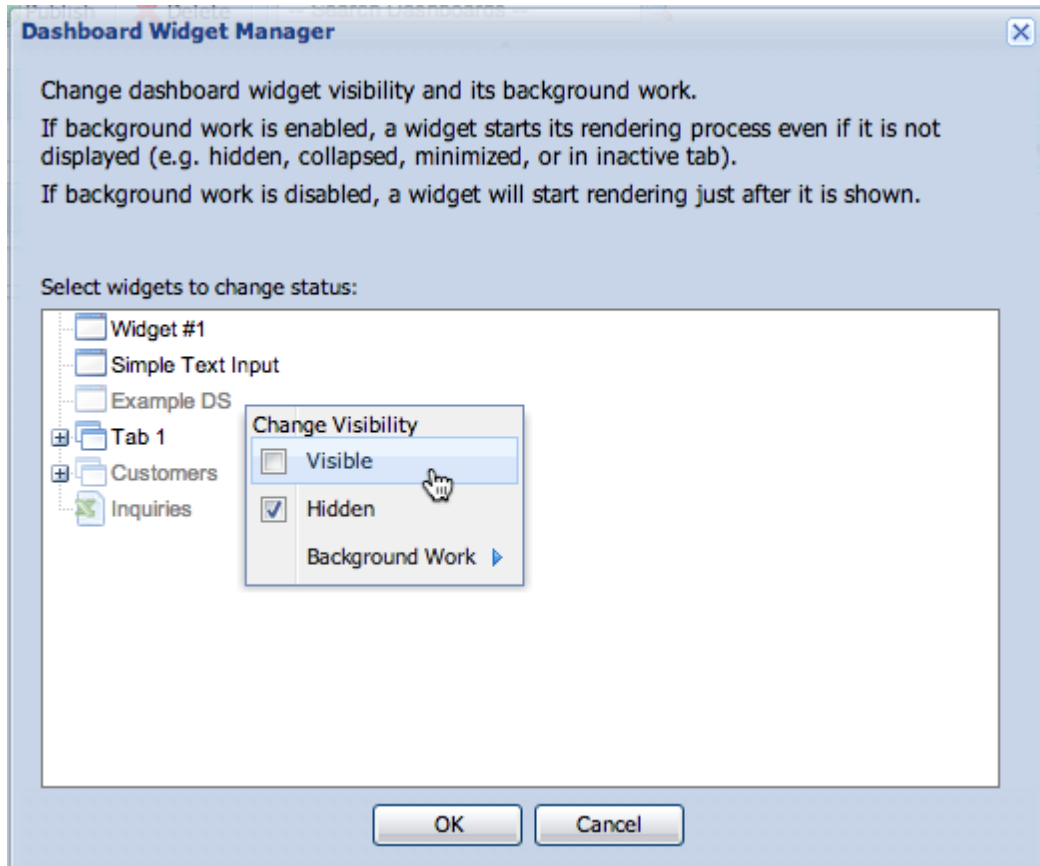
Note: The first time you configure a grouped widget in a tab layout, if you hide a tab, the tab content appears in the next active tab until you reopen the dashboard.

Managing Show/Hide Status from the Dashboard Editor

A widget can be hidden directly from the Dashboard, by right-clicking the title bar and selecting *Hide Widget*.

Customers			
Id	Last Name		Title
10001	Kane		VP Platform Strategies
10002	Reiss		Vice President, Oper...
10003	Johnson		
10004	Sivaraman	Alejandra	Systems Integration ...
10005	Pestana	Alex	Systems Administrator
10006	Maples	Ali	Systems Administrator

As they are no longer accessible from the Dashboard, hidden widgets are managed from the *Dashboard Widget Manager*. The Manager is accessed from the *Manage Dashboard Widget* menu.



The *Dashboard Widget Manager* enables you to:

- Make hidden widgets visible in the Dashboard (restore visibility).
- Configure hidden widgets to work in the background. (By default, hidden widgets do not start the rendering process.)

Customization of Error Messages

Dynamic Views supports customization of form submit related error messages that are returned from the server. For example, you can hide the error code, disable the link that displays details of the error, and so on.

Customization of error messages is configured by modifying the following lines in the project code:

```
public void testException(String param1) throws DynamicViewsException {  
  
    DynamicViewsException e = new DynamicViewsException("#The Message#");  
    e.setErrorCode("#The ErrorCode#");  
    e.setStatus("#The Status#");  
    e.setError(true);  
    throw e;  
}
```

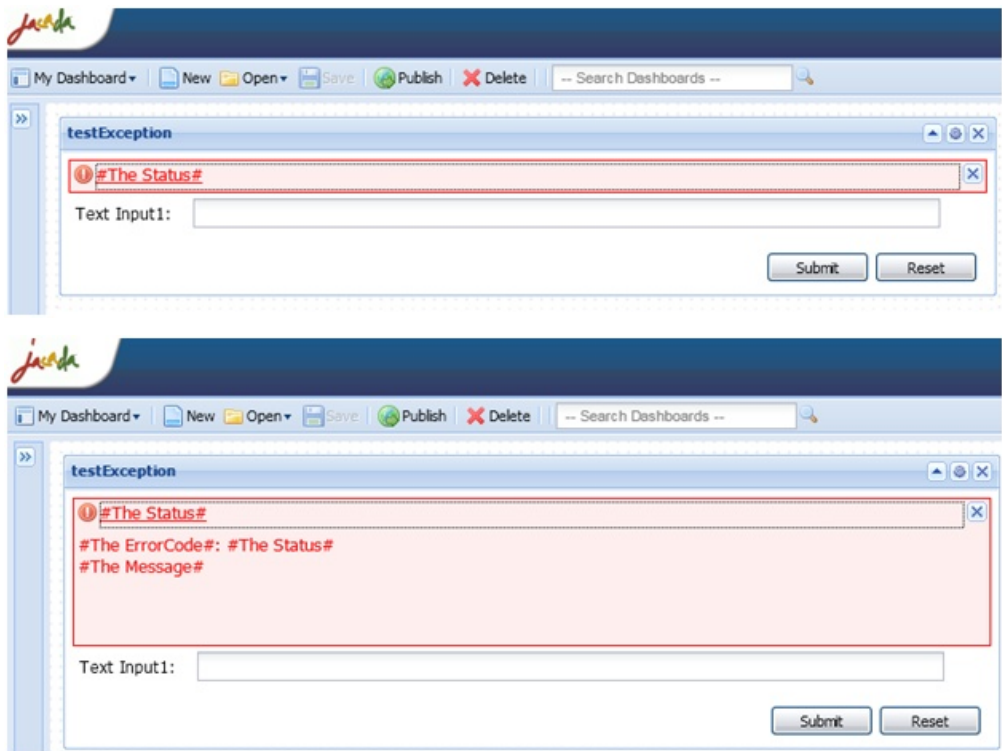
When error code information is not required, `errorCode` should be empty. In order to detect an error response without `errorCode`, you can check whether the `error` property in the JSON response is true or not.

If the server response includes a non-empty `status` property in the JSON, it will show the value as the label of the alert box. If the `status` property is not available or empty, it uses a concatenation of `errorCode` and head part of `message` as a label.

Additionally, if the `message` property in the JSON is not available or empty, it will not enable the expand link in the label that shows a detail message.

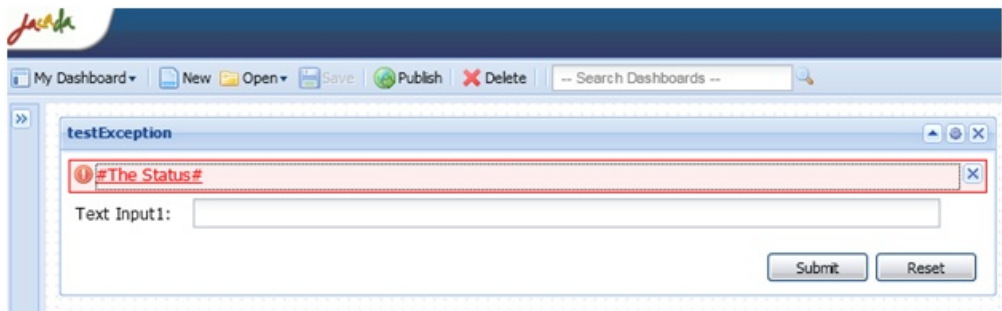
Examples

```
public void testException(String param1) throws DynamicViewsException {  
  
    DynamicViewsException e = new DynamicViewsException("#The Message#");  
    e.setErrorCode("#The ErrorCode#");  
    e.setStatus("#The Status#");  
    e.setError(true);  
    throw e;  
}
```

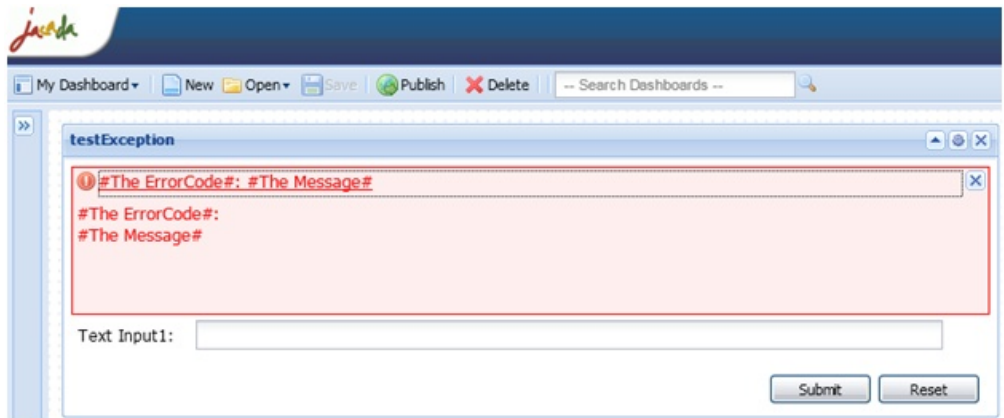
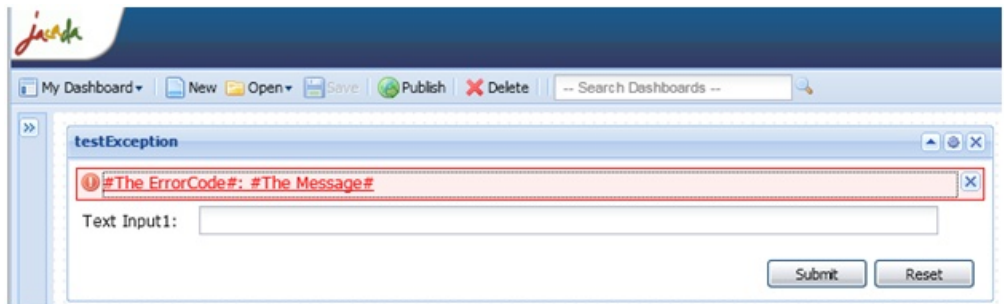


```
public void testException(String param1) throws DynamicViewsException {

    DynamicViewsException e = new DynamicViewsException();
    e.setErrorCode("#The ErrorCode#");
    e.setStatus("#The Status#");
    e.setError(true);
    throw e;
}
```

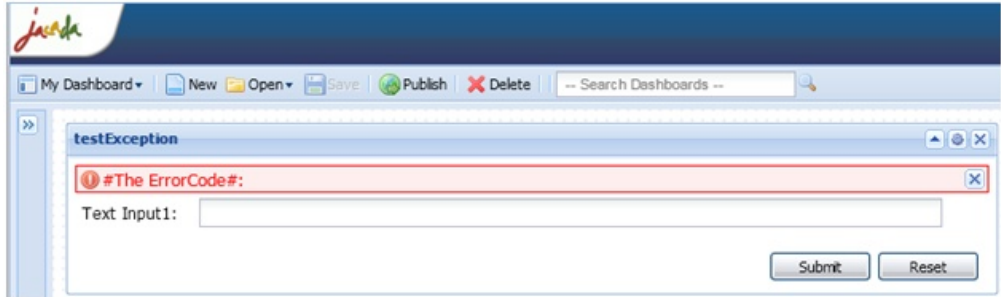


```
public void testException(String param1) throws DynamicViewsException {  
    DynamicViewsException e = new DynamicViewsException("#The Message#");  
    e.setErrorCode("#The ErrorCode#");  
    e.setError(true);  
    throw e;  
}
```



```
public void testException(String param1) throws DynamicViewsException {

    DynamicViewsException e = new DynamicViewsException();
    e.setErrorCode("#The ErrorCode#");
    e.setError(true);
    throw e;
}
```

Deploying a Dashboard in Jacada WorkSpace

Once a new Dashboard has been created and developed, it can be added as a tab or portlet to Jacada WorkSpace. Tabs are managed using the *Tabs Configuration* page of the Jacada Management Console.

To add a new Dashboard as a tab:

1. Verify that the Management Console is open.
If necessary, open the Management Console as follows:
 - a. In the Eclipse IDE, right click on <project name>, and select Jacada WorkSpace Agent Desktop > Management Console.
The Management Console opens.
 - b. If required, enter Username and Password.
2. From the tree on the left, select *Layout Management > Tabs Configuration*.
The *Tabs Configuration* page appears.
3. At the bottom of the page, click *Add*.
The *Edit Tab Settings* page appears.
4. Specify parameters for the tab in the *General Settings* frame and the *Web Application Settings* frame.
For the *URL* parameter, select the name of the relevant Dashboard from the *Dynamic View Dashboard* dropdown list. The URL of the relevant Dashboard appears automatically in the *URL* field.

Note: For full descriptions of all parameters, refer to the Jacada WorkSpace Administrator Guide, (Using the Jacada Management Console, Tabs Configuration).

5. Click **Save**.

A tab containing the Dashboard is added to Jacada WorkSpace.

To add a new Dashboard as a portlet:

1. Verify that the Management Console is open.

If necessary, open the Management Console as follows:

- a. In the Eclipse IDE, right click on <project name>, and select Jacada WorkSpace Agent Desktop > Management Console.

The Management Console opens.

- b. If required, enter Username and Password.

2. From the tree on the left, select *Layout Management > Portlets*.

The *Portlets* page appears.

3. At the bottom of the page, click *Add*.

Portlet parameters appear.

4. Specify parameters for the portlet (verify that the portlet is large enough to contain the dashboard).

For the *URL* parameter, select the name of the relevant Dashboard from the *Dynamic View Dashboard* dropdown list. The URL of the relevant Dashboard appears automatically in the *URL* field.

Note: For full descriptions of all parameters, refer to the Jacada WorkSpace Administrator Guide.

5. Click **Save**.

A portlet containing the Dashboard is added to Jacada WorkSpace.

Functional Widgets

The widgets described in the following table are general use widgets that are included in the Jacada WorkSpace Development Package.

Widget Name	Description and Developer Instructions
HTML	Simple HTML-rendering widget, in which HTML coding can be entered or pasted. The widget's title and the height can be configured.
Rich Text	HTML rendering widget with rich text editor. Rich text entered or copied is rendered as HTML. The widget's title and the height can be configured. The first time rich text is used, a question mark is added to the text when saving. This is resolved by reediting the text.
Simple Text Input	Displays text.
Calendar	Enables you to choose dates that can later be submitted, e.g., birthdates of customers.
Inline Frame	Displays a custom-designed URL, or any available URL.
Grid Table	Displays a table (in grid format).

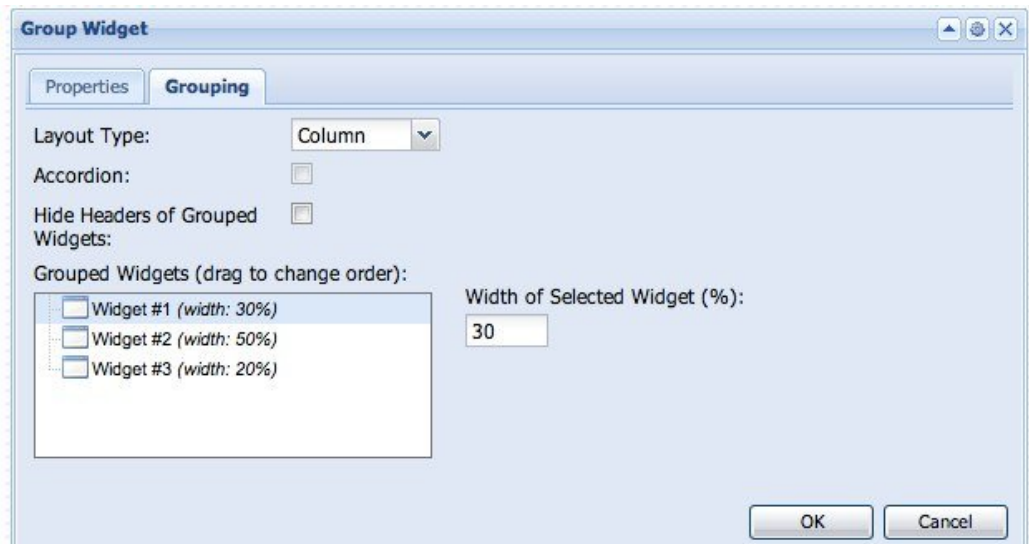
Grouped Widgets

A Grouped Widget is a single widget that contains a set of widgets within it. The widget set can be displayed horizontally or vertically.

When the widget set is displayed horizontally, the width of each widget can be configured as a percentage of the entire widget. The following figure shows an example of a Grouped Widget that has 3 inner widgets, whose width preference is 30%, 50%, and 20%.



The inner widget width is configured from the *Grouping* tab of the *Group Widget* dialog. To update the width of an inner widget, select the relevant widget from the list in the *Grouped Widgets* frame, and then enter the desired width (in percent) in the field to the right.



If the sum of the widgets' width exceeds 100%, a horizontal scroll bar is displayed.

Jacada Widgets

The widgets described in the following sections were especially designed for Dashboard development by Jacada WorkSpace developers:

- [Logger Widget](#)

- [Push Widget](#)
- [Auditor Widget](#)
- [Update Properties Widget](#)

Logger Widget

This widget allows you to graphically program logging and bind data.

To graphically program logging without adding lines of code:

1. In development mode, drag the *Logger Widget* into a Dynamic View Dashboard.
2. In the parameter tag of the widget (Edit mode), enter the log message, and choose a log level.
3. Before saving the Dashboard, collapse the widget. If it is not collapsed, it is displayed at runtime.

Note: Testing in design mode causes a message to appear on the widget itself. However, in runtime, the widget is not visible, and a well-formatted message is written to the log file.

The following examples show how to bind data to the *Logger Widget*.

To bind grid data to a Logger Widget:

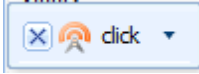
1. In development mode, drag the *Customers* data source widget into a Dynamic View Dashboard, and select *Grid Table* as the widget type.
2. Select the *Display Data* tab, and drag all relevant fields from the *Property Source* to the *Columns to Display* field.
3. Click *OK*.

The *Customers* data source widget is displayed in the grid.

To bind the Customers grid to the Logger Widget:

1. Drag and drop the *Logger* widget from the Resource Panel into the Dynamic View Dashboard.
2. On the *Logger* widget's *Parameter* tab, select the log level.
3. Click on a row in the *Customers* grid.

A *NowCapturingEvent* pop-up is displayed:



4. Drag the *NowCapturingEvent* pop-up to the *Log Message* field of the *Logger* widget's *Parameter* tab, until the pop-up is directly over the *Log Message* field, and the field turns pink. When the field is pink, release the mouse pointer.
A pop-up box appears, displaying available properties.
5. From the popup box, double-click on the relevant property. The property is displayed in the *Logger* widget.
6. To bind additional properties, repeat Steps 4-5. Press *Alt* while performing Step 4.

Note: In order not to overwrite properties that are already bound, make sure to press *Alt* while doing additional drag-and-drop processes.

7. To display the *Logger* widget, click *OK*.
8. Test the functionality added by clicking on various lines in the grid and verifying that the expected log message is printed to the *Logger* Widget.

Push Widget

This widget enables you to graphically program a push event to be fired whenever the server sends a push message with the specified input keyword.

To graphically program a push event:

1. Drag the *Push* widget into a Dynamic View Dashboard (development mode).
2. In the parameters tag of the widget (edit mode), enter the *Push Keyword* and set *Print Debug Messages* to True.
3. Click *OK*.
The *Fire Push Event* button is displayed.
4. Click the *Fire Push Event* button.
The *NowCapturingEvent* pop-up is displayed.
5. Mouse over the *Event Capture* widget until a crossbow cursor is displayed, and drag the *Event Capture* widget into a relevant display widget, e.g., an HTML widget.
When the target widget's background is pink, the event's attributes are displayed.
6. Click on the attribute/object to be displayed in the target widget.

Results Example:

If you dropped the push event data object into the HTML widget, the HTML Content tab of the widget will contain:
`${w1_push.data}`

Note: Generally, the server sends a JSON object on push messages. To display one of the object's fields, concatenate the field's name to the push event parameter. For example, if the field name is "title", and the push event data was `${w1_push.data}`, change that to `${w1_push.data.title}`.

7. Collapse the *Push* widget, so that it will not be visible at runtime.
8. Save the Dashboard.

In design mode, a *Fire Event* button is displayed on the widget. A push event will be fired whenever this button is pressed. The data attached to this event is the current time.

In runtime, whenever the server sends a push message with the input keyword, a push event is fired with the server's data.

Auditor Widget

This widget enables you to graphically create auditors.

Note: The client auditor must be enabled for the *Auditor* widget to be functional.

To graphically create an auditor:

1. Drag the *Auditor* widget into a Dynamic View Dashboard (development mode).
2. In the parameters tag of the widget (edit mode), specify the following values:
 - Enter the action name, parameter name, and parameter value in the appropriate textboxes.

Note: A parameter value is not required.

- From the *Scope* drop-down list, select a scope for the action.
3. Test the widget in design mode. It will display ("print") all audited parameter values that should be sent to the server.

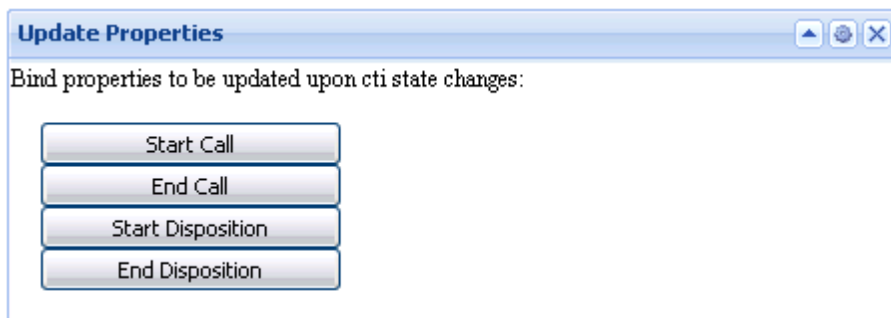
4. Collapse the *Auditor* widget, so that it will not be visible at runtime.
5. Save the Dashboard.

Update Properties Widget

This widget enables you to bind predefined properties to specific Dynamic View Dashboards. The properties, in turn, are bound to predefined CTI events. Upon invocation of a specific CTI state (e.g., when an agent starts a call), the set of properties defined for that state are automatically updated in the relevant Dashboards, eliminating the need to reload the entire Dashboard.

The CTI states that can be linked to Dashboard properties are:

- Start call
- End call
- Start disposition
- End disposition



Associating properties with CTI states and binding them to a Dynamic Views Dashboard involves the following processes:

1. **Matching CTI states to desired properties:** This is done from the Eclipse IDE, with the Dashboard Update Configuration wizard. For details, refer to [Using the Dashboard Update Configuration Wizard](#).
2. **Binding the properties to relevant widgets:** This is done from the Dynamic Views Designer, using the *Update Properties* widget. For details, refer to [Using the Update Properties Widget](#).

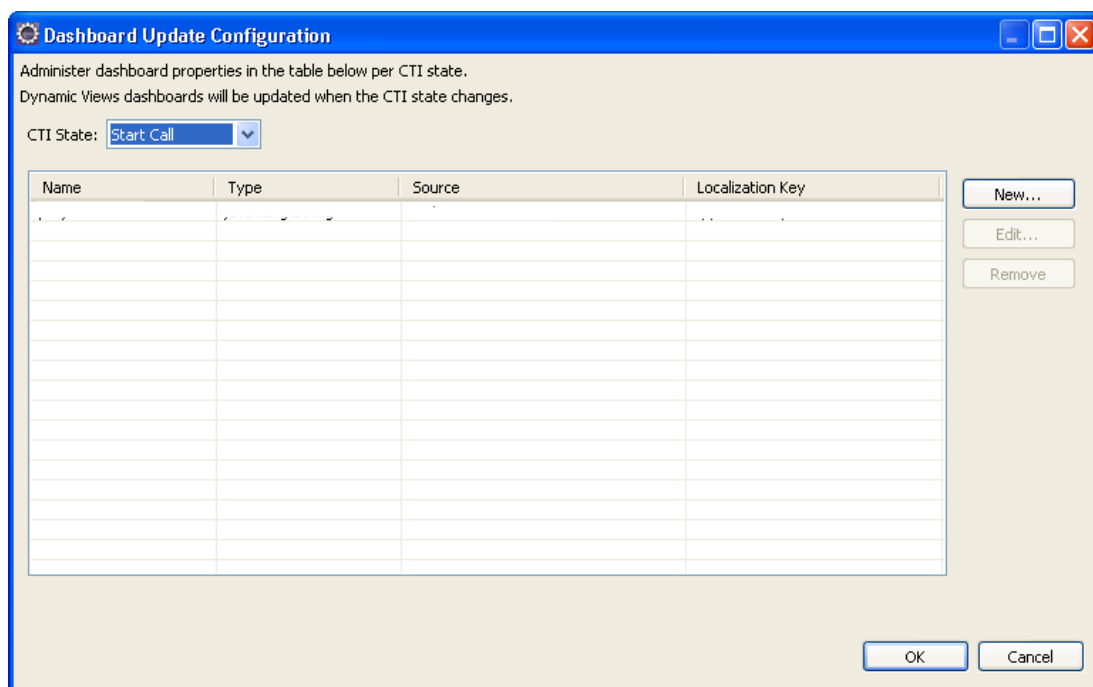
Using the Dashboard Update Configuration Wizard

This wizard, which is accessed from the IDE, enables you to match specific properties with specific CTI events. After the project is republished, the properties can be viewed when using the Dynamic Views *Update Properties* widget, and can be bound to other widgets.

To use the Dashboard Update Configuration wizard:

1. From the Eclipse IDE, right-click on the relevant project, and select *Jacada Work Space > Jacada Dynamic Views > Dashboard Update Configuration*.

The Dashboard Update Configuration wizard appears.



2. From the *CTI State* drop-down list, select the CTI state to which you want to bind properties. Then, at the right side of the wizard, click *New*.

The *New Dashboard Property* dialog appears.

New Dashboard Property

Property Information

Property Name:

Property Type:

Display Name:

Property Source

Bean Name:

Bean Interface or Class:

Methods to Call:

Select from the list below.

Use a period as a separator between successive methods, if required.

3. In the *Property Name* textbox, enter a name for the property, as you want it to appear in the Update Properties widget.
4. From the *Property Type* drop-down list, select the relevant property type.
5. In the *Display Name* field, enter a name for the property that will be displayed during a binding action. Clicking *Browse* enables you to navigate to a desired Display Name.

Note: Display Names need to be previously configured in the project code. Display Names are defined in the **UI_Texts_<language>** properties file (e.g., **UI_Texts_en.properties**).

This file is located in the

<project name>/WebContent/USER/resources/resourceBundles directory.

The Display Name must begin with

application.dynamicviews, for example **application.dynamicviews.headerName=MyHeader**.

6. In the *Bean Name* field, enter the name of the Spring Bean to be returned. Alternatively, click *Browse*, and then navigate to the desired bean.

The remaining fields of the *Property Source* frame are enabled.

7. From the *Bean Interface or Class* drop-down list, select the relevant interface or class.

Associated methods appear in the list box at the bottom of the dialog.

8. From the list box, select a method. The selected method appears in the *Methods to Call* field.

To specify additional methods, enter a period after the method in the field. The methods lists reappears, allowing you to choose another method.

Note: Make sure that each method is separated by a period.

9. Click *OK*.

The Dashboard Update Configuration wizard appears, with the defined property parameters displayed in the grid.

10. If necessary, make modifications to the grid:

Add a property: Click *New*, and refer to Steps 3-9 above.

Modify property parameters: From the grid, select the relevant property, and then click *Edit*. Alternatively, double-click the relevant row in the grid.

In the *Edit Dashboard Property* dialog that appears, update relevant parameters, and then click *OK*. (Refer to Steps 3-9 above).

Delete a property: From the grid, select the relevant property, and then click *Remove*.

11. If desired, match properties with additional CTI events:

From the *CTI State* drop-down list, select the relevant CTI state, and click *New*.

Then, follow Steps 3-10 above.

12. When all desired CTI events are matched with properties, at the bottom of the Dashboard Update Configuration wizard, click *OK*.

The wizard disappears, and the project is republished, and several files are created. One of these is able to be read by the Dynamic Views Designer, allowing you to bind selected properties to Dashboards. For more details, refer to [Using the Update Properties Widget](#).

Using the Update Properties Widget

This widget enables you to bind properties linked with CTI events to a Dynamic Views Dashboard. When a CTI state begins (start call, end call, etc.), selected properties defined for that state are automatically updated in all widgets to which the properties are bound.

Note: Dashboard properties are defined using the Dashboard Update Configuration wizard. For details, refer to [Using the Dashboard Update Configuration Wizard](#).

The following procedure explains how to bind predefined Dashboard properties to a target Dynamic Views widget. The procedure needs to be repeated for each widget to which you want to bind properties.

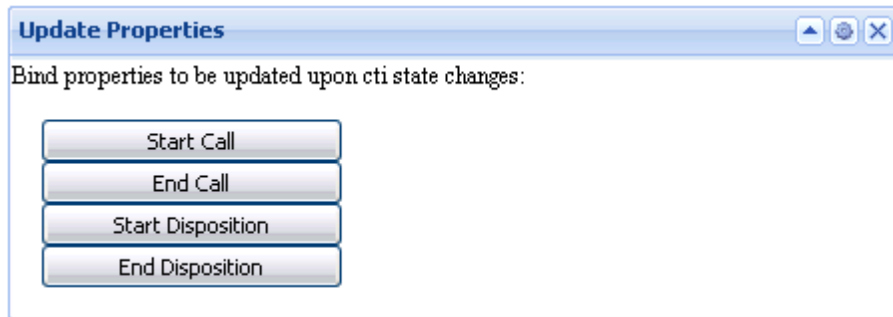
To bind Dashboard properties to a Dynamic Views widget:

1. From the Resources Panel on the right side of the Dynamic Views Designer, select the *Widgets* frame.

The *Widgets* frame expands.

2. From the *Jacada Widgets* directory, drag the *Update Properties* widget to the desired location in the work area.

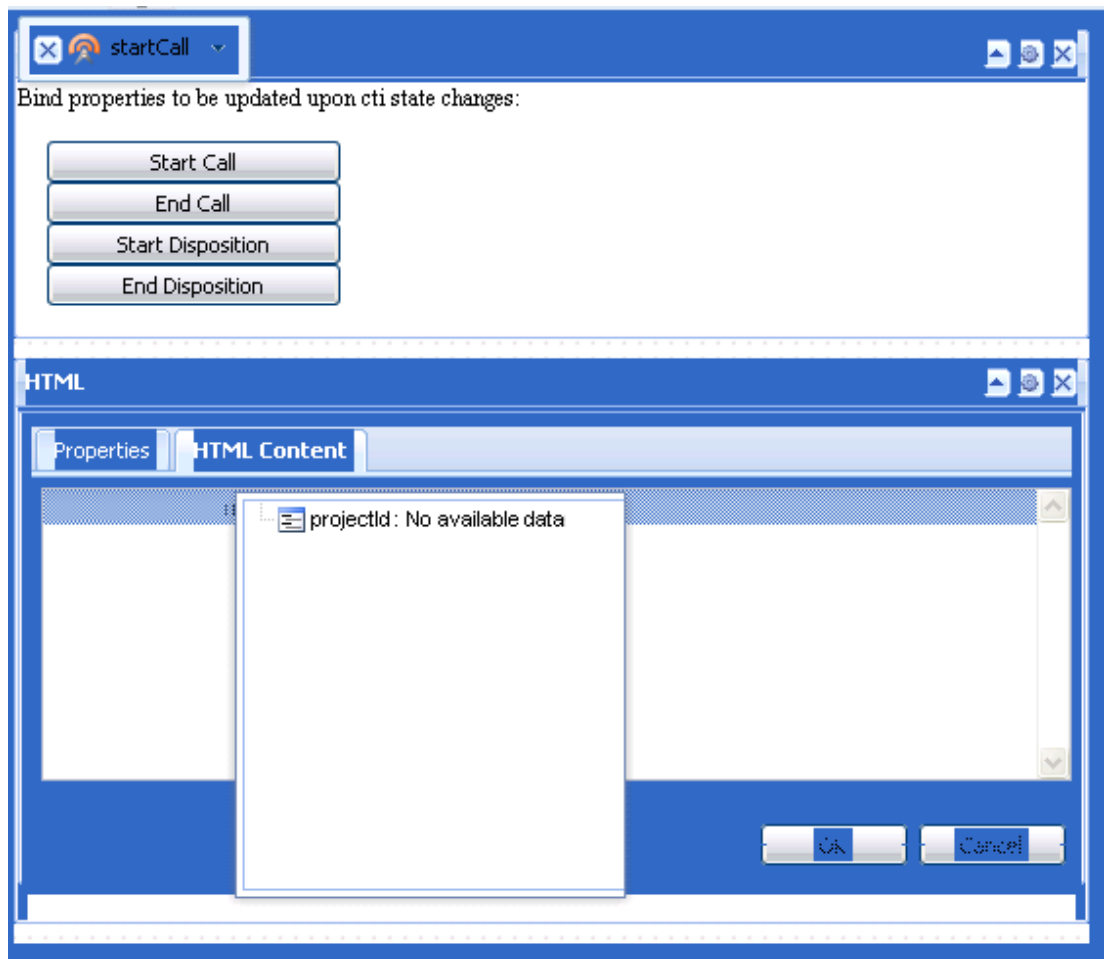
The widget expands, displaying the possible CTI states whose properties can be bound to other Dashboards.



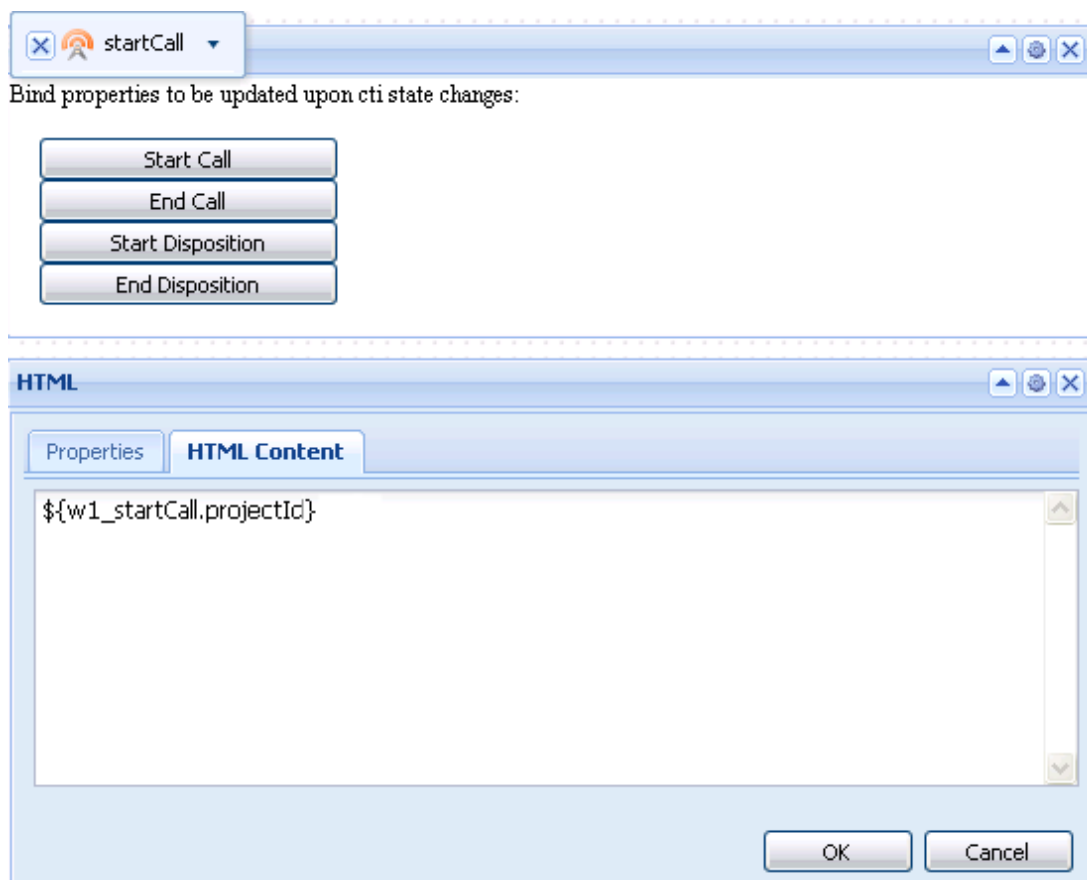
3. Select a CTI state. A *NowCapturingEvent* pop-up appears at the upper left corner of the *Update Properties* widget.

Note: The *NowCapturingEvent* pop-up appears only when the target widget is opened for editing.

4. Drag the *NowCapturingEvent* pop-up to the relevant area in the widget to which you want to bind properties. When the background of the target widget turns pink, release the mouse. A pop-up window appears, listing all the predefined properties for the selected CTI event.

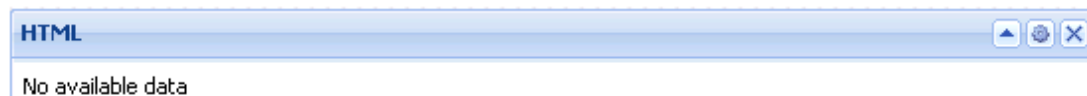


5. From the pop-up window, click the property to be bound to the target widget.
The value of the selected property appears in the widget.



6. To bind additional properties for the CTI state, press *Alt* while dragging *NowCapturingEvent* pop-up to the target widget. Then, repeat Steps 4-5 above.
7. If desired, bind properties for other CTI states to the target widget, by repeating Steps 3-6 above.
8. When all desired properties are displayed in the target widget, at the bottom of the target Dashboard, click *OK*.

The *NowCapturingEvent* pop-up disappears, and the target widget is displayed as it will appear in runtime.



Coordinating Dashboards Among Multiple Developers

During the development phase, each developer may create and work on a separate Dashboard. Techniques of synching the Dashboards vary according to development conditions.

- **Developers are working with the same database and application server:** In this case, since all Dashboards are stored in the database, the developers have the same Dashboards. However, depending on which user created the Dashboards, not all developers may be able to view all the Dashboards.

To ensure that multiple developers are able to see all Dashboards, the following techniques are recommended:

- Have all developers use one user when working in the Dynamic Views Designer.
- If developers use separate users, upon completing development they should make their Dashboards **public**.
- **Developers are working with different databases and application servers:** In this case, the developers have different Dashboards. To sync Dashboards, it is recommended that developers use the Transport feature to export and import Dashboards between environments. For details about the Transport feature, refer to the [Jacada WorkSpace Administrator Guide](#), (Using the Jacada Management Console, Layout Management).

Creating a New Widget and Making It Available

A widget is a small visual component of the Dynamic Views design environment that can be dragged to a Dashboard to perform a task. A widget can also be used in the design of another widget's functionality.

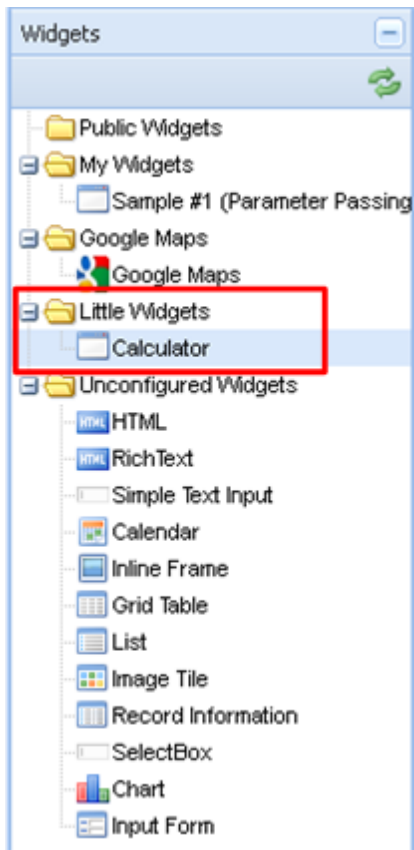
You can create inline widgets (written in JavaScript, etc.) or IFrame widgets (displaying an URL). The advantages/disadvantages of each widget type are summarized in the following table.

	IFrame	Inline
Advantages	Ease of writing Independence from other variables used on the page	Provision of framework for more complicated scripts Ability to take advantage of existing system programming Speed of execution
Disadvantages	Slow loading	Inadvertent overriding of system variables

Generally, inline widgets are preferable to IFrame widgets in terms of performance. However, when you are creating a widget that will be repeatedly dragged into the same Dashboard, an IFrame widget is recommended, since inline widgets containing multiple variables do not work in that situation.

New widgets are generally created in the development environment using the New Widget Skeleton wizard. The files generated by the wizard are concatenated in a file called **plugin.js**. This file is exposed to the Dynamic Views Designer as a plug-in file that defines the new widgets at load time.

In the Dynamic Views Designer, widgets are stored in folders according to group. The group name is the name of the folder in which a category of widgets is stored. In the following illustration, the *Calculator* widget is part of the *Little Widgets* group.



When creating a widget, you can specify a new group name, or use an existing one. In addition, you can associate any parameter (string, integer, float, boolean, or object) with a widget.

To create a widget

1. From the Eclipse IDE, right click on the relevant project name, and select *Jacada Work Space > Jacada Dynamic Views > New Widget Skeleton*.

The New Widget Skeleton wizard opens.

Widget Skeleton Wizard

New Widget Skeleton

Create a new Jacada Dynamic Views widget skeleton

Widget Name:

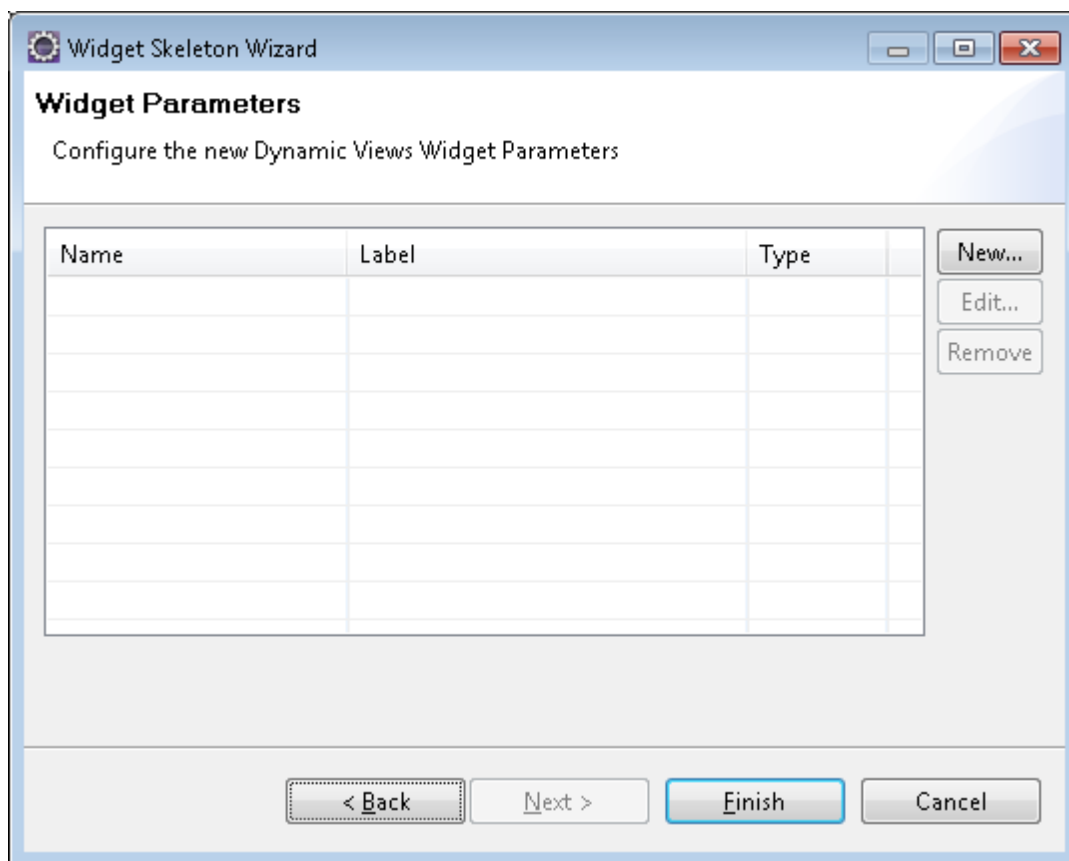
Widget Title:

Group Name:

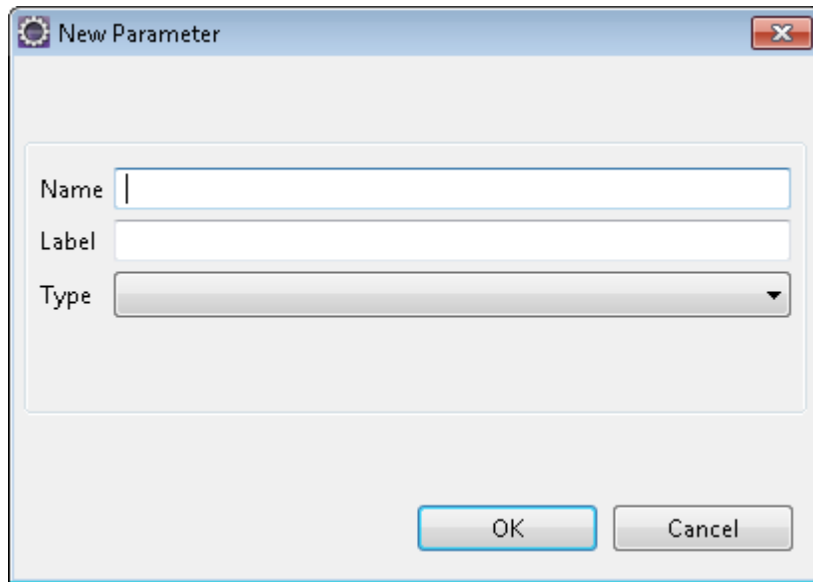
☐ Load widget in a separate IFrame

< Back Next > Finish Cancel

2. Enter the following required parameters in the appropriate textboxes:
 - **Widget Name:** A unique identifier for the widget. This is also the folder name under which the widget will be built. The widget name cannot contain spaces or special characters.
 - **Widget Title:** The display name that appears in the *Widgets* frame of the Dynamic Views Resources Panel.
 - **Group Name:** Name of the folder of the group to which the new widget will belong (as it appears in the Resources Panel of the Dynamic Views Designer). The group name cannot contain spaces or special characters.
3. To create an IFrame widget, check the *Load widget in a separate IFrame* checkbox.
(By default, the checkbox is cleared, and an inline widget is created.)
4. If desired, define widget parameters, by clicking *Next*.
The *Widget Parameters* dialog appears.



5. Click **New**.
The *New Parameter* window appears.



6. Specify the following parameter definitions:

In the *Name* textbox, enter the internal name of the parameter.

In the *Label* textbox, enter the display name of the parameter (as it will appear in the widget).

From the *Type* drop-down list, select the parameter type.

7. Click *OK*.

The *Widget Parameters* dialog appears, with the defined parameter displayed.

8. If necessary, make modifications to the *Widget Parameters* list:

- **Add a parameter:** Click *New*, and refer to Steps 6-7 above.
- **Modify parameters:** From the list, select the relevant parameter, and then click *Edit*. In the *Edit Parameter* window that appears, update relevant parameters, and then click *OK*. (Refer to Step 6 above).
- **Delete a parameter:** From the list, select the relevant method, and then click *Remove*.

9. When you are finished editing the *Widget Parameters* list, click *Finish*.

The New Widget Skeleton wizard disappears, and a .json file (inline widget) or .html file (IFrame widget), whose name is based on the widget name specified, is displayed in the IDE Console for your modification.

10. Edit the code of the file to fit the intended functionality.

Note: To edit this in the future, use your IDE project explorer and navigate to *ProjectName>WebContent>USER>Widgets><GROUPNAME>WidgetName*.

11. Republish the project. (You will be prompted to save the .json or .html file.)

Note: When you save the .json or .html file, any modifications you made to it are saved.

The new widget is saved, and is available the next time you access the Dynamic Views Designer.

Programming a Widget Repository

Generally, programming is done using the New Widget Skeleton wizard. Alternatively, you can program using the `afrous.widget.repository.register` method.

By calling the `afrous.widget.repository.register` method, you can create a directory which contains multiple widget items. This directory is rendered as a tree in the widget section of the Dynamic Views Dashboard panel.

Code Snippets

Method Interface	Description
<code>afrous.widget.repository.register(directory)</code>	Using this method you can create a directory to contain any number of widget items.

Method Parameters	Description
<code>directory</code>	Directory configuration object

Required Fields for Directory Configuration

Parameter	Description
name (required)	Name of the directory entry. Name must be unique in the repository.
title (required)	Display name of the directory, rendered as a node label in tree.
items (required)	Array of widget configurations.

Common Widget Configurations

Parameter	Description
title (required)	Display name of the widget, rendered as a node label in tree.
description	Detail description of what the widget is. Description information pops up as a quick tip of the node.
icon	16x16 size Icon URL, for displaying the widget in recognizable way.
wtype (required)	Widget type. Currently only “url” is supported.
definition	Widget definition configuration object

URL Widget Definition Configurations

Parameter	Description
url (required)	URL of the resource you want to display in the widget.

Parameter	Description
params	Array of parameter configuration to be passed as query parameter when showing the URL content.
firesEvent	If this configuration is set to true, the URL widget can fire events and report them to its outer environment, i.e. Dashboard.
paramsAsMessage	If this configuration is set to true, the URL widget receives its parameter not through the URL query parameter but by inter-frame message, which can be listened by widget-inner JavaScript. This is useful when it needs dynamic update inside of the widget without server refresh.

Parameter Configurations

Parameter	Description
name (required)	Parameter name, equivalent to query parameter name in content request.
label	Display name of the parameter. This is used, for example, in widget parameter prompt.
type	Parameter data type. Following types are available: 'string', 'integer', 'float', 'data' and 'boolean'.
required	Boolean value which tells this parameter is required or not.
options	Array of candidate values for the parameter. It can be also regarded as a pick list.

Example of Server Configuration

Dynamic Views supports configuration of the server from the Java code. The following example shows how you can dynamically register plug-in script information (URLs) to the metadata server.

Note: This API overrides existing data. If you want to delete or update existing records, you must first retrieve all existing records, make the necessary changes, and then invoke the API with the updated data.

At the moment, the system supports up to 8 records.

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;

import org.json.JSONObject;

public class ChangeSettingApp {

    static String BASE_URL = "http://localhost:8080/dynamicviews/";

    List<String> cookies = new ArrayList<String>();

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        String login = "admin";
        String password = "admin";
        Boolean isResetting = args.length > 0 &&
args[0].equals("--reset");

        ChangeSettingApp app = new ChangeSettingApp();

        System.out.println("--- Login ---");
        app.login(login, password);

        if (isResetting) {
            System.out.println("---- Resetting
default setting ---");
            app.updateDefaultSetting(app.crea
teResetRequest());
        } else {
            System.out.println("---- Update
current default setting ---");
            app.updateDefaultSetting(app.crea
teUpdateRequest());
        }

        System.out.println("--- Getting current default
setting ---");

        System.out.println(app.getDefaultSetting().toStri
ng());

        System.out.println("--- Done. ---");
    }
}

```

Example of a Programmed Widget Plug-in Definition

When you create new widget resources using the `afrous.widget.repository.register` method, a specific format must be used. The following example shows the format required for creating one widget resource. (Generally, multiple items are listed in the file.).

```
afrous.widget.repository.register({
  name : 'my-custom-widgets',
  title : 'My Custom Widgets',
  items : [{
    title : 'My Custom Widget',
    wtype : 'url',
    definition : {
      url : 'http://mywidget.example.com/hello.html',
      params : [{
        name : 'name',
        label : 'Your Name',
        type : 'string'
      }]
    }
  }]
});
```

The main sections are:

- Name:** Internal name for the group to which the new widget resource will belong.
- Title:** Name of the folder of the group to which the new widget resource will belong (as it appears in the Resources Panel of the Dynamic Views Designer).
- Items:** A list of the widgets to be added to the group, and definitions of the attributes of each new widget. Widget attributes are listed and described in the following table.

Attribute	Sub-attribute	Description
title		Name of the widget, as it is to appear in the Resources Panel of the Dynamic Views Designer.

Attribute	Sub-attribute	Description
wtype		Type of widget. Values can be: • url: Iframe widget • custom: Inline widget
header		If set to true, the widget header will be displayed.
allowBackground Work		If set to true, the widget will be loaded, even if it is hidden.
definition		
	url	The location of the code defining the widget.
	params	A list including: • name: Internal name of the widget parameter. • label: The display name of the parameter (as it will appear in the widget). • type: String (default), integer, float, boolean, or object.

Example of a Programmed Custom Action Definition

The following example provides a template to use when defining a custom action. The code should be placed in **jacadaActions.js** (provided out-of-the-box with Dynamic Views).

```
afrous.dashboard.actions.register({
  name : "my-custom-actions", // should be unique in custom action groups
  title : "My Custom Actions", // shown as separator header in action menu.
  items: [{
    name: "console-log-debug", // should be unique in this custom action group
    title: "Log event in console.log", // shown as custom action menu label
    params: [{
      label: "Param #1", // displayed in event action setting dialog
      name: "param1" // key name of params passed to action function
    }, {
      // ...
    }],
    action : function(event, params, data) { // action function
      // event is name of the event (e.g. "click" or "submit"). Type:String.
      // params is a JavaScript key-value pair of parameters which is set in
      event action binding.
      // data is a JavaScript object which is assigned to the event. Type:Any.
      console.log(event, params, data);
    }
  }, {
    // .. other custom actions definition in the menu.
  }]
});
```

Attribute	Sub-attribute	Description
name		A unique name for the actions group.
title		Title of the actions group.
items		An array of actions.

Attribute	Sub-attribute	Description
	name	The unique action name in this group.
	title	The action title that appears in the menu.
	action	The Javascript action.
	params	A list including: • name: The internal name of the action parameter. • label: The display name of the parameter (as it will appear in the action).

Example of a Programmed Inline Widget

Dynamic Views supports changing the styling of a custom (inline) widget by applying CSS styles to it. The following sections present an example of a custom widget and sample customized styling.

Custom Widget Example

The following is a sample custom widget, defined in **custom-inline-widget-plugin.js**.

```

afrous.widget.defineCustomWidgetRenderer({
  // "name" is required and must be unique in whole dashboard system.
  name : 'my-styled-inline-widget-renderer',

  // JavaScript Method for rendering content.
  renderContent : function(context) {
    var params = context.getParams();
    var panel = context.getPanel();
    panel.removeAll(true);
    panel.add({
      xtype : 'form',
      itemId : 'form',
      cls: 'my-styled-inline-widget-content',
      html:
        '<div class="title">Foods</div>' +
        '<ul>' +
        '<li class="fruit">apple</li>' +
        '<li class="vegi">potato</li>' +
        '<li class="fruit">orange</li>' +
        '<li class="vegi">radish</li>' +
        '</ul>',
      buttons : [{
        text : 'OK'
      }]
    });
  }
});

```

Styling Example

The following example shows how styling of the above widget is customized in **custom-inline-widget-plugin.css**.

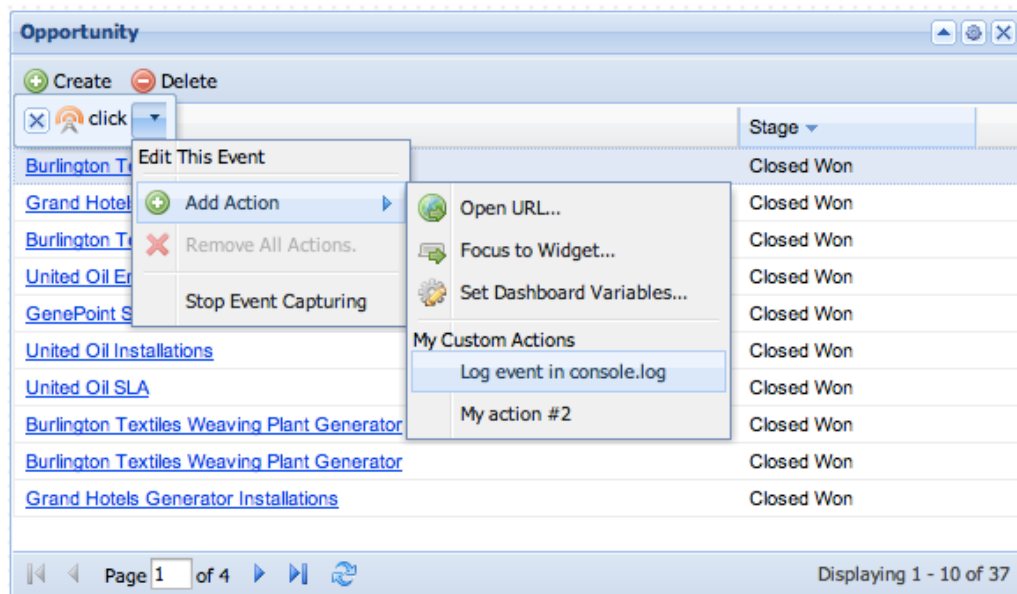
```
.my-styled-inline-widget-content div.title {
  margin: 5px 0px;
  text-align: center;
  font-size: 24px;
  font-weight: bold;
}
.my-styled-inline-widget-content ul {
  list-style: circle;
  font-size: 14px;
}
.my-styled-inline-widget-content ul li {
  margin-left: 20px;
  color: red;
}
.my-styled-inline-widget-content ul li.vegi {
  color: green;
}
.my-styled-inline-widget-content button.x-btn-text {
  color: blue;
  font-weight: bold;
}
```

Creating a Custom Action

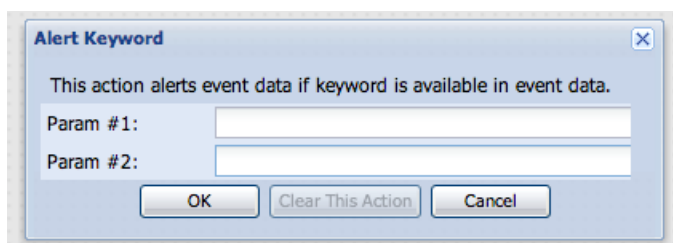
Dynamic Views Dashboard developers can extend standard behavior by defining custom actions in a plug-in script ([Programmed Custom Action Definition](#)). When an action is bound to an event, parameter values, defined in the custom event action definition, can be set.

Binding a Custom Action to an Event

In the Dashboard, added custom actions are listed in the *Action* menu.

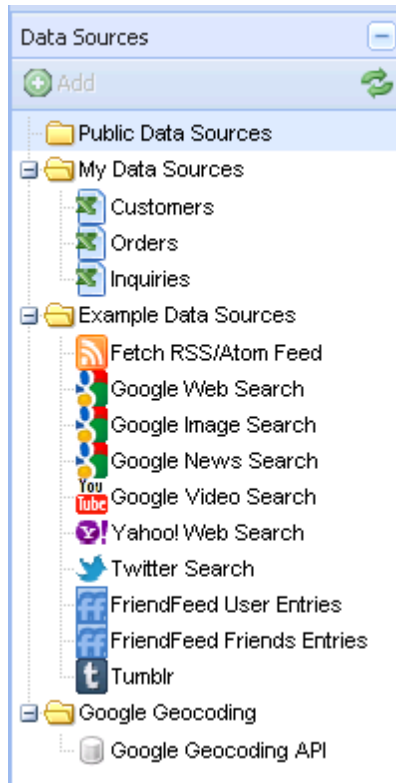


When the action is selected, binding of the action to the event takes place, and a confirmation dialog is displayed. If parameter are defined inside the custom action definition, a Parameter list is displayed, in which you can enter parameter values.



Creating Data Sources

The *Data Sources* frame of the Dynamic Views Designer enables you to create widgets that are linked to specified Web sources or databases. The Designer is delivered with several sample data sources (listed in the *My Data Sources* and *Example Data Sources* directories). In addition, you can create custom data sources that are specific to the customer's environment.



A new data source can be created using any of the following tools:

- **URL DataSource Wizard:** This wizard is accessed from the Dynamic Views Designer, and is used to create a generic URL-based data source. For details, refer to [Creating a Data Source Using the URL DataSource Wizard](#).
- **Publish Data Source to Dynamic Views dialog:** This dialog is accessed from the Eclipse IDE, and is used to create a data source linked to a Jacada WorkSpace project. For details, refer to [Creating a Workspace-related Data Source](#).
- **API:** This technique involves calling the `afrous.datasource.repository.register` API method to create a directory containing multiple data source items. For details, refer to the mashmatrix document: **mashmatrix Dashboard Plug-in Script Developer Guide**.

Creating a Data Source Using the URL DataSource Wizard

Using the URL DataSource wizard involves entering the URL of the data source, specifying how the request is to be processed, and then sending the request to the target resource. After the request is submitted and processed, parameters specific to the data source (such as name and description) are defined.

Initial request parameters are listed in the following table.

Parameter	Description/Notes
Base URL to Request	The URL of the data source to be returned. If the data source is defined in WorkSpace, use the {projectName} environment variable to define the URL. For example: <code>/ {projectName} /myService</code>
Output Format	Format of the content to be returned. Supported formats include .csv, .xml, and .json.
URL Parameters	If the data source accepts query parameters, any number of parameters can be defined.
Transport Option	Method of sending the request. The following options are supported: • Server Proxy: Request is sent via the server side. This is the default option. • Client Proxy: Request is sent via the client side. This option is used when firewall issues prevent the server from connecting to the resource. For more details, refer to Using Client Proxy Transport. • JSONP: Request is sent via JSONP transport. This option is relevant only to .json output formats.

To create a data source using the URL DataSource wizard:

1. From the Resource Panel on the right side of the Dynamic Views Designer, expand the *Data Sources* frame.
2. Right click on the *My Data Sources* folder. Then, from the pop-up window that appears, click *Add*.

The URL DataSource wizard is displayed.

The screenshot shows the 'URL DataSource Wizard' dialog box. It has a title bar with the text 'URL DataSource Wizard' and a close button. The main area contains the following elements:

- Base URL to Request:** A text input field.
- Output Format:** A dropdown menu.
- URL Parameters:** A large text area for entering parameters. Above this area are two buttons: 'Add Param' (with a green plus icon) and 'Remove Param' (with a red minus icon).
- Transport (Option):** A dropdown menu.
- Next >** A button at the bottom right.

3. Complete the following request parameters:
 - a. In the *Base URL to Request* textbox, enter the URL of the relevant data source.
 - b. From the *Output Format* drop-down list, select the format of returned content.
 - c. If relevant, add URL parameters to the data source query:

In the *URL Parameters* frame, click *Add Param*, and then complete the Reference Name, Prompt Label, Data Type, and Default Value in the relevant fields.

To specify more than one value, click the *Picklist* icon, and enter each value on a separate line in the textbox.

URL Parameters:

Add Param

Remove Param

param1 (Param #1)

Reference Name:

param1

Prompt Label:

Param #1

Data Type:

string

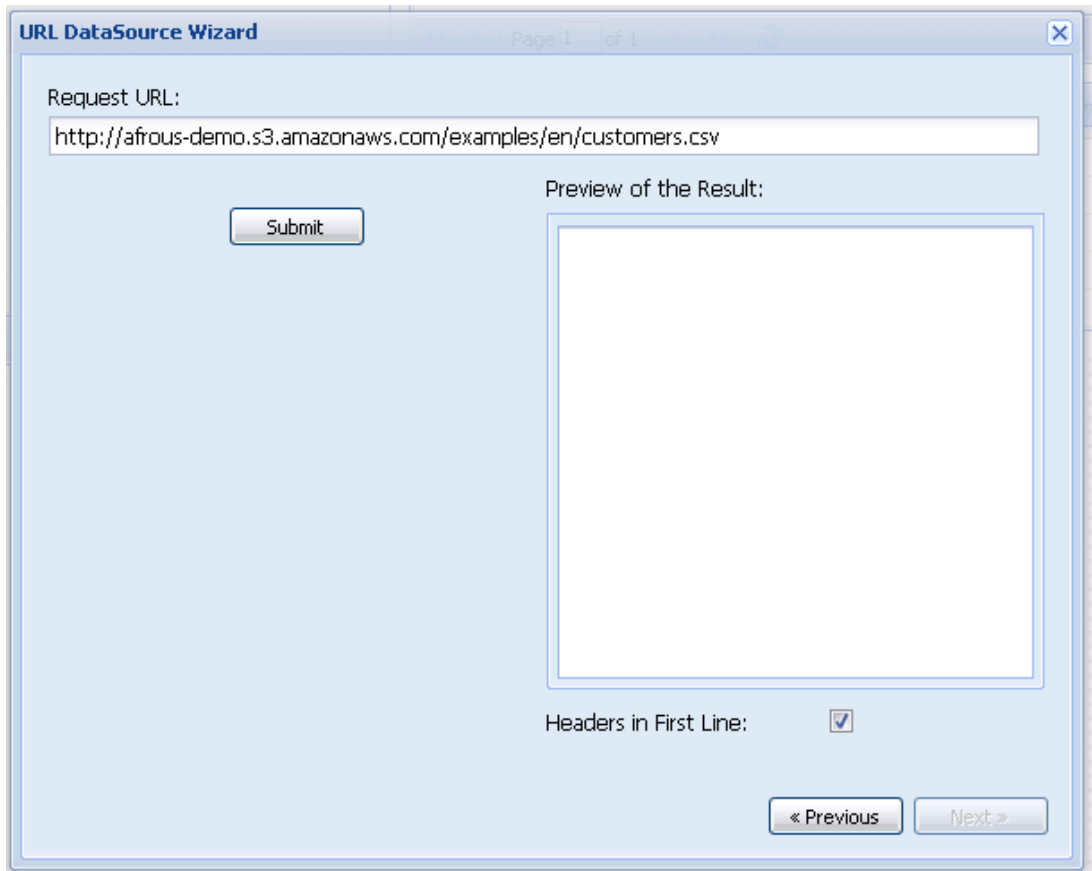
Default Value:

Picklist

- d. From the *Transport (Option)* drop-down list, select the method of sending the request. If you select *Client Proxy*, you will be prompted to specify the client proxy file location. For details, refer to [Using Client Proxy Transport](#).

4. Click *Next*.

The *Request URL* page of the wizard appears.



5. At the top of the page, in the *Request URL* textbox, verify that the correct URL is displayed. Then, click *Submit*.

One of the following preview types appears in the *Preview of the Result* frame:

- A preview of the .csv grid (.csv output format)
- A preview of the tree of the returned data (.xml and .json output formats)

URL DataSource Wizard

Request URL:

`http://afrous-demo.s3.amazonaws.com/examples/en/customers.csv`

Submit

Preview of the Result:

Id	Last Name	First Name
10001	Kane	Aarti
10002	Reiss	Abhey
10003	Johnson	Alan
10004	Sivaraman	Alejandr
10005	Pestana	Alex
10006	Maples	Ali
10007	Scholfield	Andrew
10008	Curavo	Andrew
10009	Hansen	Arif
10010	Garcia Lopez	Arne

Headers in First Line: ☒

« Previous Next »

6. Perform one of the following actions, according to the preview type:

- **.csv Output Format:** If desired, clear the *Headers in First Line* checkbox. When the checkbox is cleared, column descriptions (e.g., ID, First Name, etc.) are incorporated into the first row of the grid, and grid headers list only the column number.
- **.xml and .json output formats:** Navigate to the relevant output root (the root of the records array). The selected root path is displayed in the *Output Root Path* field.

Then, click *Next*.

The *Title* page of the wizard appears.

URL DataSource Wizard

Title:
Customers1

Description:
extra DS sample

Icon URL (Optional):

<< Previous Finish

7. Specify the following parameters:

- In the *Title* textbox, enter the name of the data source, as it is to appear in the list of data sources in the Resources Panel of the Dynamic Views Designer.
- In the *Description* textbox, enter a description text to appear in the tooltip of the data source.
- If desired, in the *Icon URL* textbox, enter the URL of an icon to appear to the left of the data source name (in the Resources Panel).

If no URL is specified, a default icon is used.

Then, click *Finish*.

The new data source is listed in the *Data Resources* frame of the Resources Panel, in the *My Data Sources* directory.

Note: The data source can now be used to create a new widget. For details, refer to [Creating a New Dashboard and Adding a Data Source Widget](#).

Using Client Proxy Transport

When firewall issues prevent the server from connecting to the required data source, you can configure the request to be sent via client proxy transport. To use client proxy transport, you need to

indicate the URL of the Crossdomain Proxy HTML file, and place this file in the same domain as the target resource.

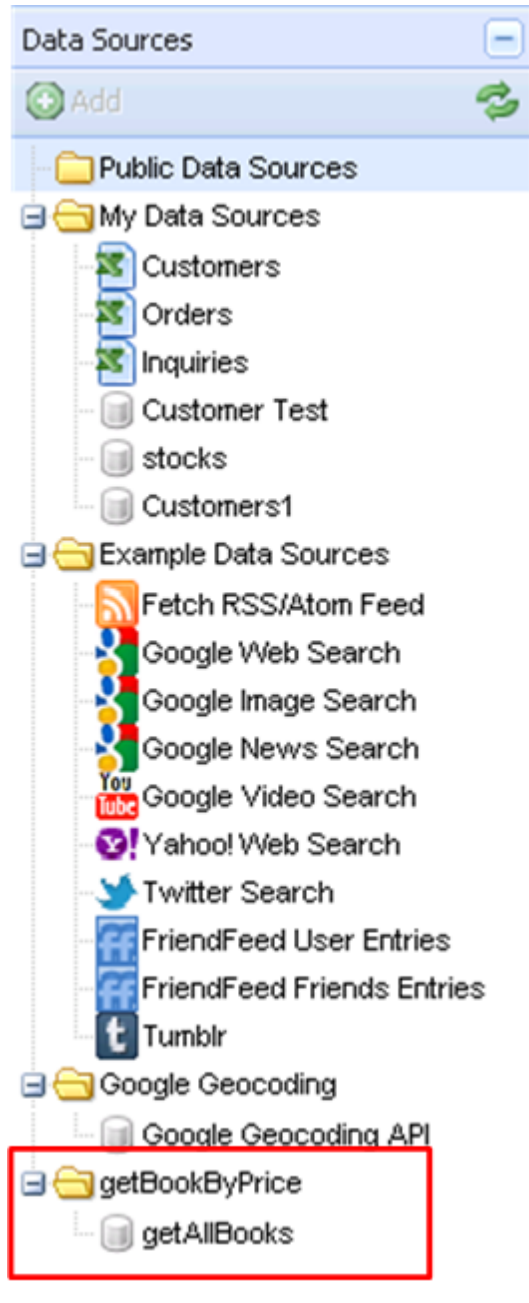
When you select *Client Proxy* as the transport option (on the first page of the URL DataSource wizard), an additional page of the wizard is displayed. This page provides a link from which you can download the Crossdomain Proxy HTML file, and a textbox in which you can enter its URL. Before continuing with the creation of the data source, verify that the target resource and the Crossdomain Proxy HTML file are located in the same domain.

Creating a Workspace-related Data Source

Workspace-related data sources return information from a specified Jacada Workspace project. This type of data source is based on a specific Feature Manager class name that is defined in a Workspace project.

When a stub DAO is used in the New Feature Skeleton, all return values for the data source automatically use the predefined stub values. For more information about the New Feature Skeleton and the stub option, refer to [Generating Project Features](#).

A new Workspace-related data source is created using the Jacada Workspace Feature Update Manager, which is accessed from the Eclipse IDE. Once the data source is created, it appears in the Dynamic Views Designer, at the bottom of the *Data Sources* frame of the Resources Panel.

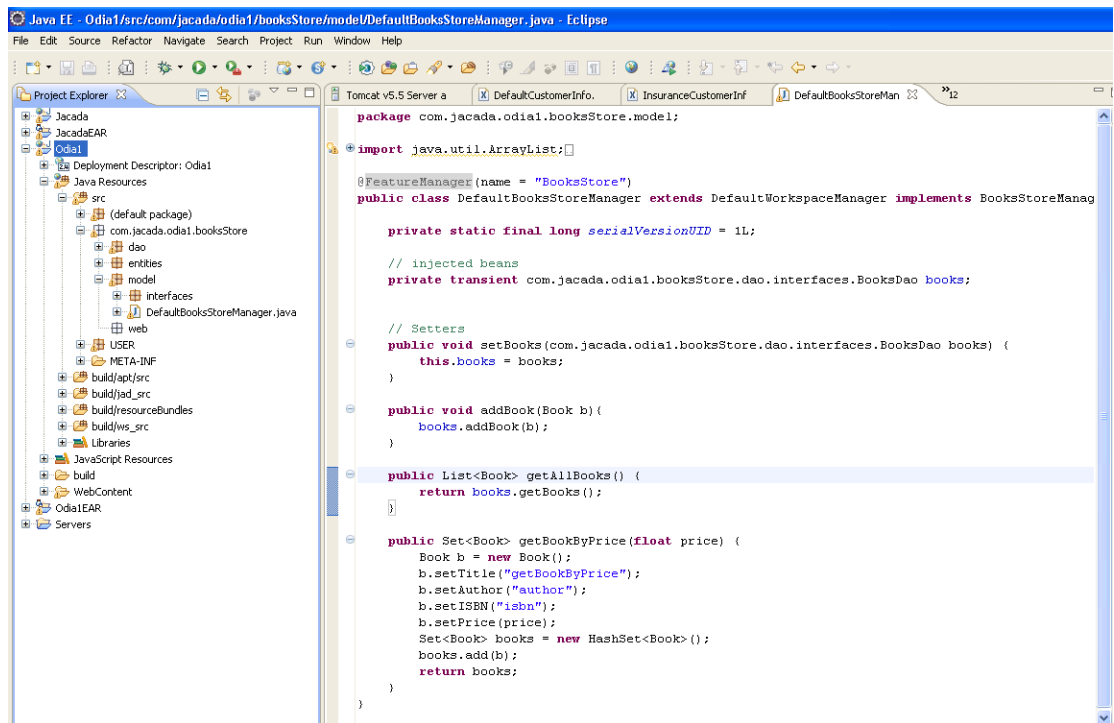


To create a WorkSpace-related data source:

1. From the *Project Explorer* tab in Eclipse, expand the *Java Resources* directory of the relevant project, and select the Feature Manager that you want to link to the new data source.

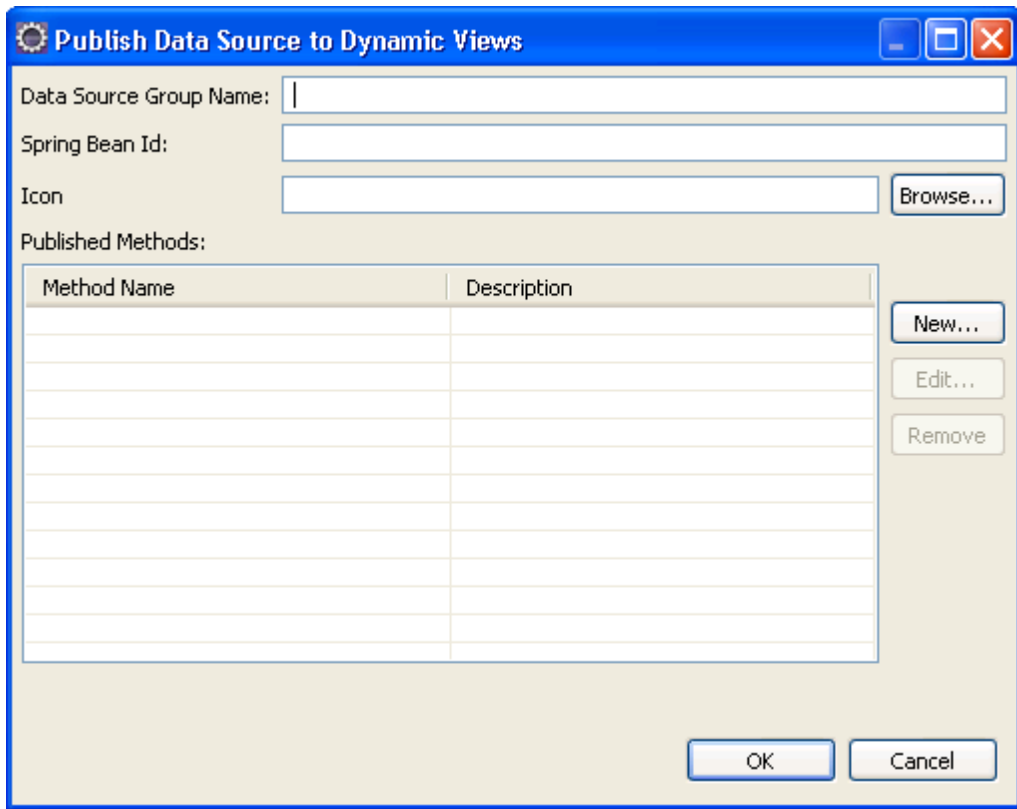
The relevant code appears in the work area.

Note: If necessary, you can create a new feature, using the New Feature Skeleton Wizard. For details, refer to [Generating Project Features](#).



2. In the work area, right-click on the relevant Feature Manager class name, and select *Jacada Work Space Feature Update > Manager > Publish Data Source*.

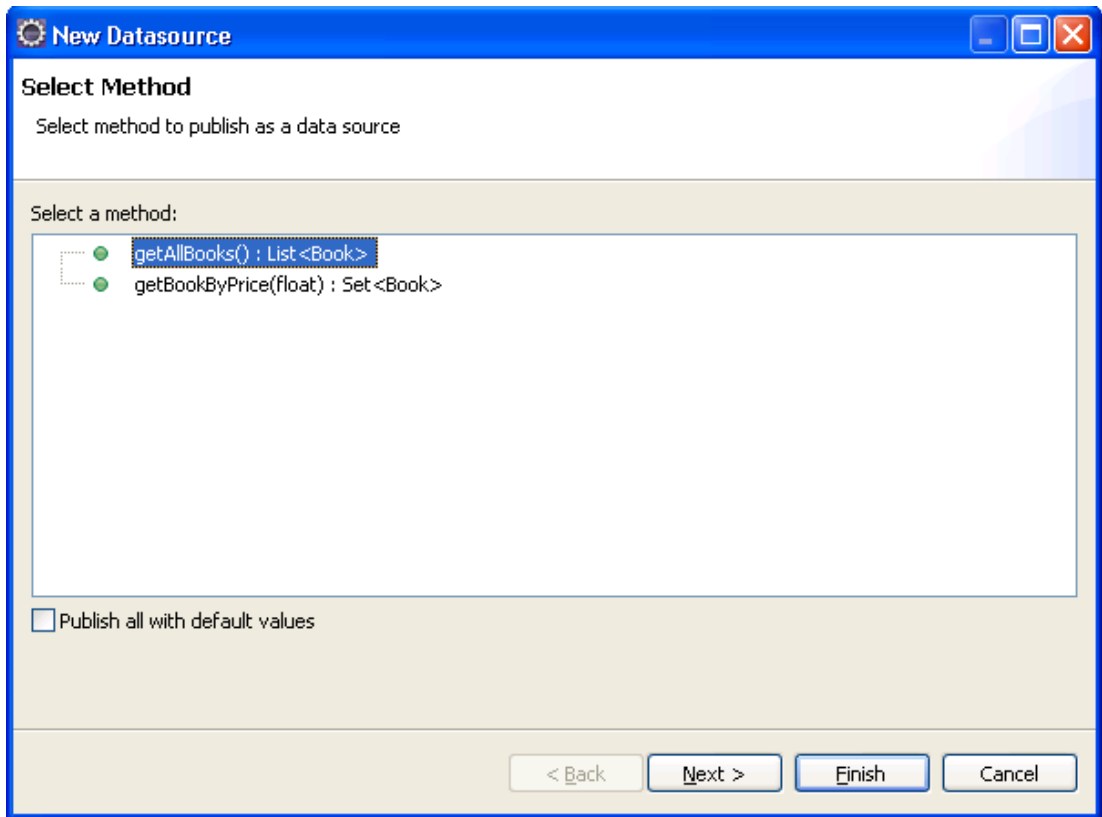
The *Publish Data Source to Dynamic Views* dialog appears.



3. In the *Data Source Group Name* textbox, enter the name of the folder (in the Dynamic Views Designer Resources Panel) to which you want to add the new data source.

Note: The *Spring Bean Id* field is filled automatically, after methods are defined.

4. If desired, in the *Icon* textbox, enter the URL of an icon to appear to the left of the data source name (in the Resources Panel).
If no URL is specified, a default icon is used.
5. On the right side of the dialog box, click *New*.
The *New DataSource* window appears, displaying all the public methods defined in the relevant Feature Manager.



6. Specify the methods to be linked with the new data source, using one of the following techniques:

- **Use default methods:** Check the *Publish all with default values* checkbox, and then click *Finish*.
The *New Datasource* window disappears, and all public methods of the feature are listed in the *Published Methods* frame of the *Publish Data Source to Dynamic Views* dialog.
- **Use customized methods:** Verify that the *Publish all with default values* checkbox is cleared. Then, select the row of the desired method, and click *Next*.
The *Define Properties for Datasource* dialog appears.

New Datasource

Define properties for datasource

Define properties for datasource

Datasource title:

Description:

Parameters

Name	Label	Default Value	Type
price	price		float

< Back Next > Finish Cancel

- a. In the *Datasource title* textbox, enter the name of the data source, as it is to appear in the Dynamic Views Designer Resources Panel.
- b. In the *Description* textbox, enter the text to appear in the tooltip of the data source.
- c. If desired, in the *Parameters* grid, change the *Default Value* to the text that you want displayed in the widget of the data source.

For example, if the default value in the project is **firstname**, you can change it to **First Name**.

Note: The other columns of the *Parameters* grid are not modifiable.

- d. Click *Finish*. The *Define Properties for Datasource* dialog disappears, and selected methods of the feature are listed in the *Published Methods* frame of the *Publish Data Source to Dynamic Views* dialog.
7. If necessary, make modifications to the *Published Methods* list:
- **Add a method:** Click *New*, and refer to Steps 5-6 above.

- **Modify method parameters:** From the list, select the relevant method, and then click *Edit*. In the *Edit Datasource* dialog that appears, update relevant parameters, and then click *Finish*. (Refer to Steps 6a-d above).
 - **Delete a method:** From the list, select the relevant method, and then click *Remove*.
8. When you are finished editing the *Published Methods* list, click *OK*.
The *Publish Data Source to Dynamic Views* dialog disappears.
 9. Republish the project, and then open the Dynamic Views Designer. (Refer to [Accessing the Dynamic Views Designer](#).)
The new data source is listed at the bottom of the *Data Resources* frame of the Resources Panel, under the selected directory.

Note: The data source can now be used to create a new widget. For details, refer to [Creating a New Dashboard and Adding a Data Source Widget](#).

Working With Input Forms

Input Form widgets allow you to create complex customized forms containing multiple field components (such as text fields, number fields, select boxes, radio buttons, checkboxes, and so on), using a drag-and-drop technique. Input Form values can be submitted to a specified URL, e.g., to update a database. You can also configure an Input Form to fire an event (upon submitting the form) that updates other widgets in the Dashboard.

Input Form

First Name: Last Name:

Gender: ☒ Male ☐ Female Category: (dropdown menu showing Customer, Partner)

Address:

Hobbies: ☐ Tennis ☒ Bike ☒ Snowboard
☐ Reading ☐ Movie

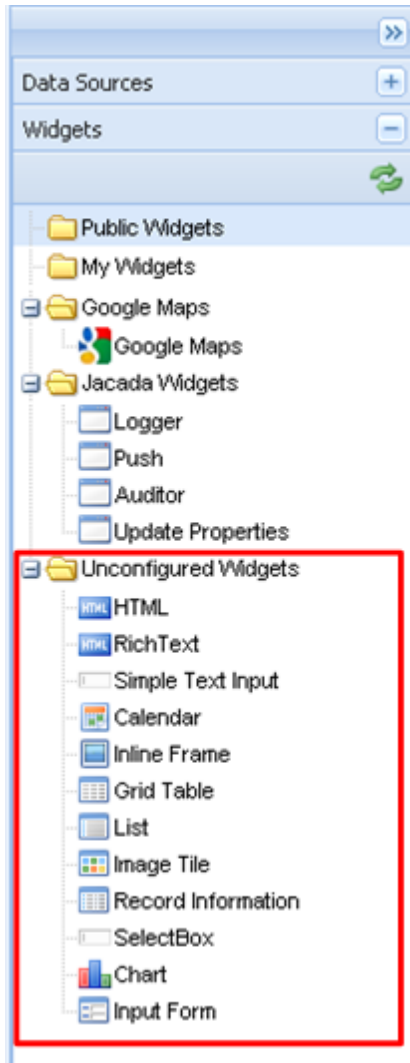
Birthday: (calendar icon)

The following types of Input Forms can be created:

- **WorkSpace-related:** Data is linked to a specific Jacada WorkSpace project. This Input Form type is created in the project IDE. (For details, refer to [Creating a WorkSpace-related Input Form.](#))

WorkSpace-related Input Forms are listed in the *Widgets* frame of the Dynamic Views Designer, under a separate directory whose name is related to the Feature Manager class name that the Input Form is based on.

- **Non-WorkSpace-related:** Data is not linked to a WorkSpace project. This Input Form type is provided with the Dynamic Views Designer (in the *Unconfigured Widgets* directory).



The process for creating widgets for both Input Form types is similar.

The following sections describe:

- [Creating a Workspace-related Input Form](#): Explains how to create an Input Form associated with a Feature Manager class in a Workspace project.
- [Creating a New Input Form Widget](#): Describes how to create and configure basic parameters

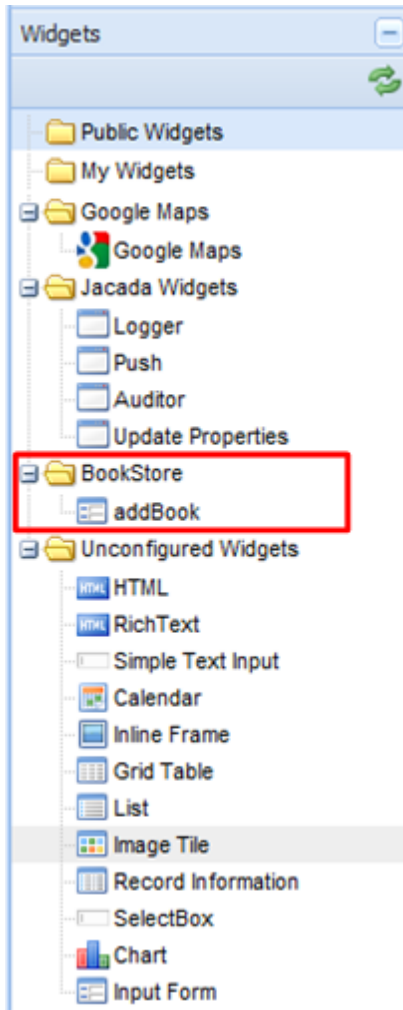
for a new Input Form widget.

- [Managing the Layout of an Input Form Widget](#): Provides details about the various field components that can comprise an Input Form widget.

Creating a Workspace-related Input Form

Workspace-related Input Forms are linked to a specific Jacada Workspace project. The information in this type of Input Form is based on a specific Feature Manager class name that is defined in the project.

A new Workspace-related Input Form is created using the Jacada Workspace Feature Update Manager, which is accessed from the Eclipse IDE. Once the Input Form is created, it appears in the Dynamic Views Designer, in the *Widgets* frame of the Resources Panel.

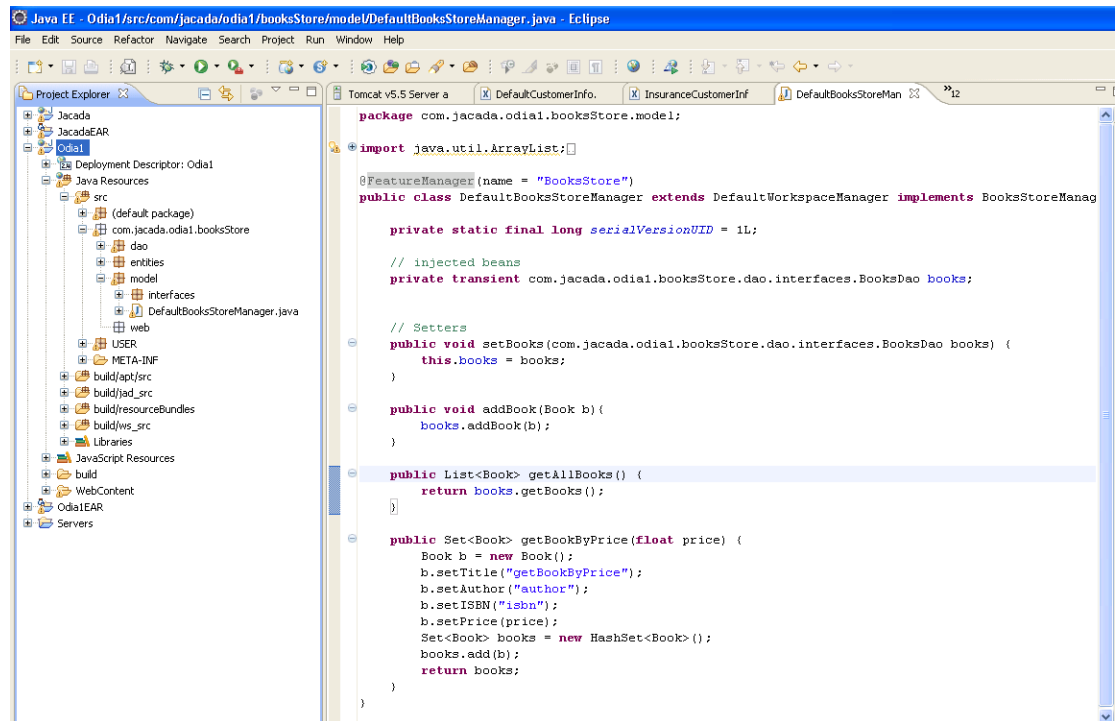


To create a Workspace-related Input Form:

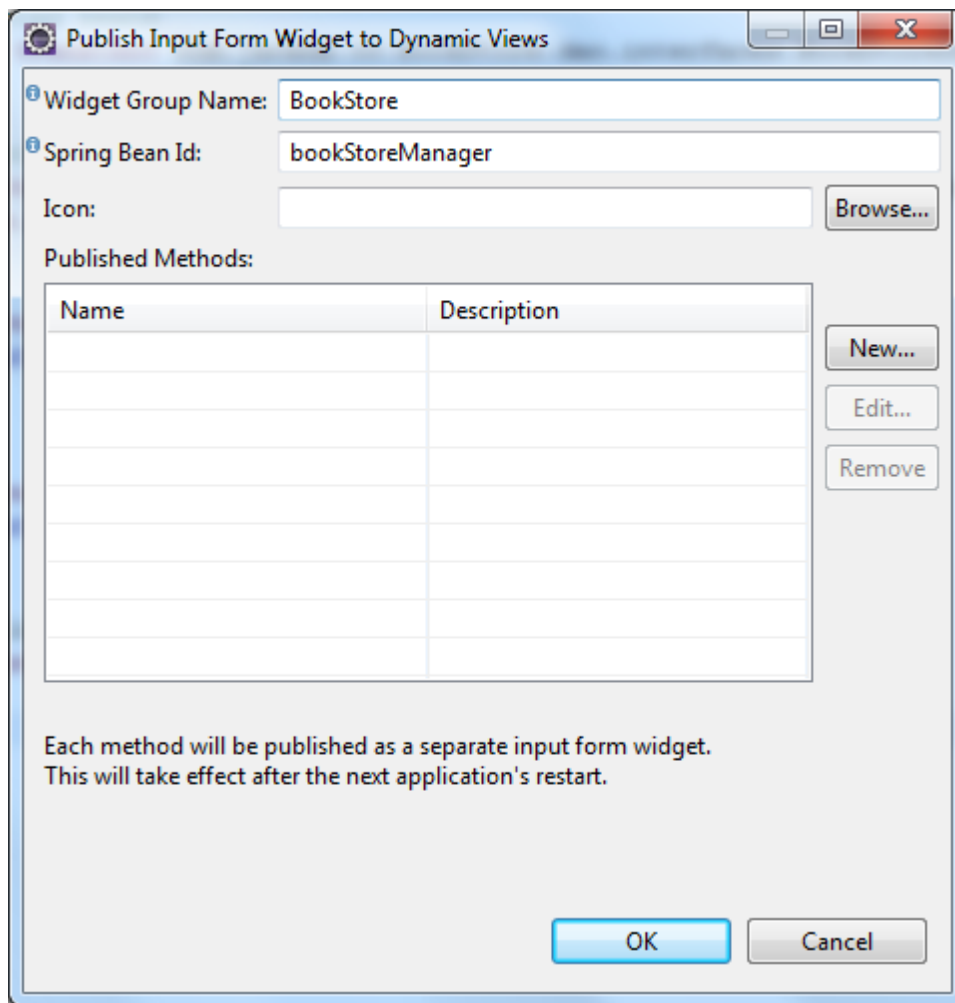
1. From the *Project Explorer* tab in Eclipse, expand the *Java Resources* directory of the relevant project, and select the Feature Manager that you want to link to the new Input Form.

The relevant code appears in the work area.

Note: If necessary, you can create a new feature, using the New Feature Skeleton Wizard. For details, refer to [Generating Project Features](#).



2. In the work area, right-click on the relevant Feature Manager class name, and select *Jacada Work Space Feature Update > Manager > Publish Input Form*. The *Publish Input Form Widget to Dynamic Views* dialog appears.



Publish Input Form Widget to Dynamic Views

Widget Group Name:

Spring Bean Id:

Icon:

Published Methods:

Name	Description

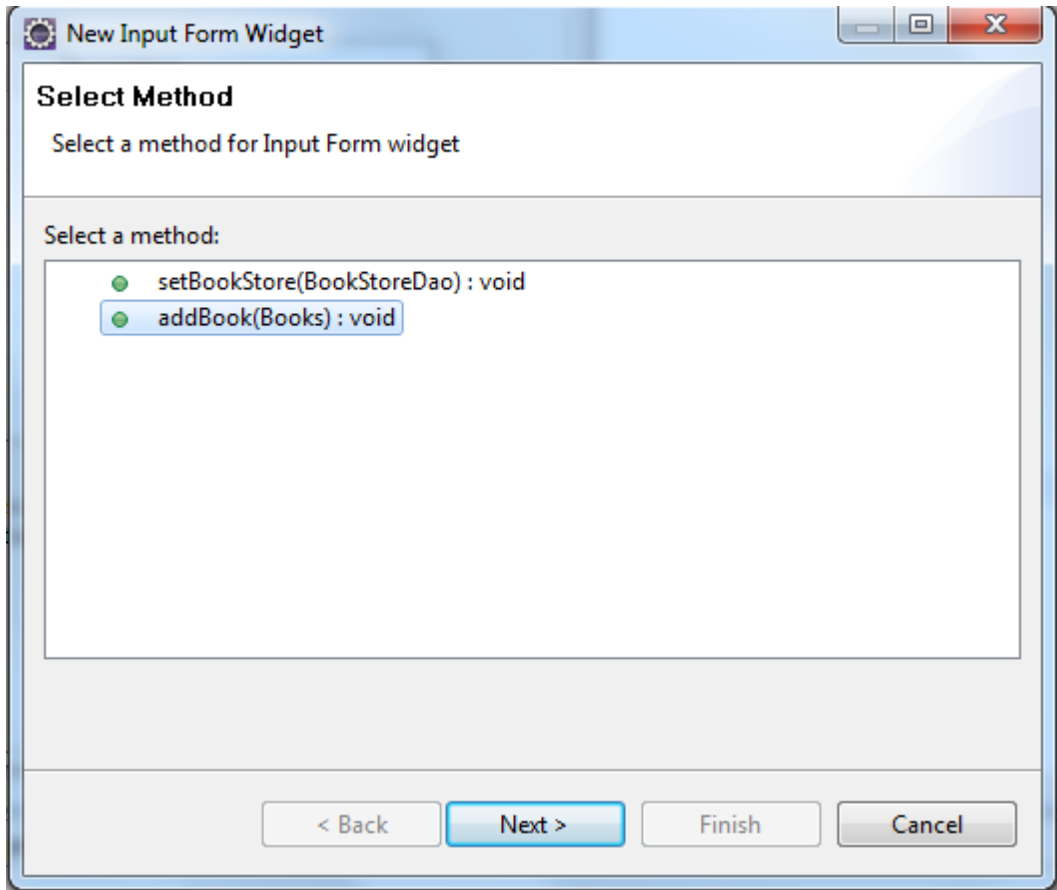
Each method will be published as a separate input form widget.
This will take effect after the next application's restart.

3. In the *Widget Group Name* textbox, enter the name of the folder (in the Dynamic Views Designer Resources Panel) to which you want to add the new Input Form.

Note: The *Spring Bean Id* field is filled automatically, after methods are defined.

4. If desired, in the *Icon* textbox, enter the URL of an icon to appear to the left of the Input Form name (in the Resources Panel).
If no URL is specified, a default icon is used.
5. On the right side of the dialog box, click *New*.

The *New Input Form Widget* window appears, displaying all the public methods defined in the relevant Feature Manager.



6. Specify a method to be linked with the new Input Form, using one of the following techniques:
 - **Use default methods:** From the methods list, select the desired method, and then click *Finish*.
The *New Input Form* window disappears, and the selected method is listed in the *Published Methods* frame of the *Publish Input Form Widget to Dynamic Views* dialog.
 - **Use customized methods:** From the methods list, select the desired method, and then click *Next*.
The *Define Properties* dialog appears, with the selected method displayed by default.

New Input Form Widget

Define properties
Configure display options for the Input Form widget

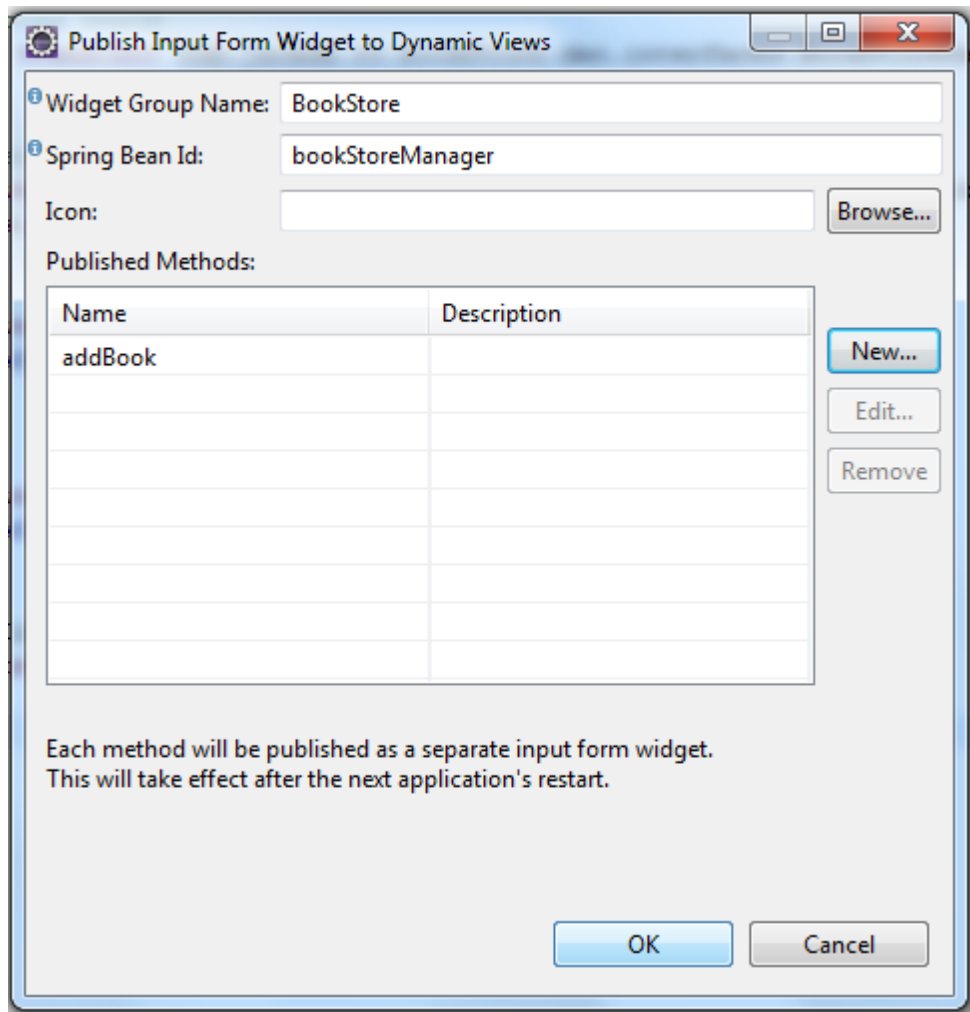
Method:

Display Name:

Description:

Icon:

- a. In the *Display Name* textbox, enter the name of the Input Form, as it is to appear in the Resources Panel of the Dynamic Views Designer.
- b. In the *Description* textbox, enter the text to appear in the tooltip of the Input Form.
- c. If desired, in the *Icon* textbox, enter the URL of an icon to appear to the left of the Input Form. If no URL is entered, a default icon is used.
- d. Click *Finish*.
The *Define Properties* dialog disappears, and the selected method is listed in the *Published Methods* frame of the *Publish InputForm Widget to Dynamic Views* dialog.



7. If necessary, make modifications to the *Published Methods* list:

- **Add a method:** Click *New*, and refer to Steps 5-6 above.

Note: A separate Input Form is created for each selected method.

- **Modify method parameters:** From the list, select the relevant method, and then click *Edit*. In the dialog that appears, update relevant parameters, and then click *Finish*. (Refer to Steps 6a-d above).
- **Delete a method:** From the list, select the relevant method, and then click *Remove*.

8. When you are finished editing the *Published Methods* list, click *OK*.
The *Publish Input Form Widget to Dynamic Views* dialog disappears, and a JavaScript plug-in is created.
9. Republish the project, and then open the Dynamic Views Designer. (Refer to [Accessing the Dynamic Views Designer](#).)
The new Input Form is listed in the *Widgets* frame of the Resources Panel, under the directory of the relevant Feature Manager.

Note: The Input Form can now be used to create a new widget. For details, refer to [Creating a New Input Form Widget](#).

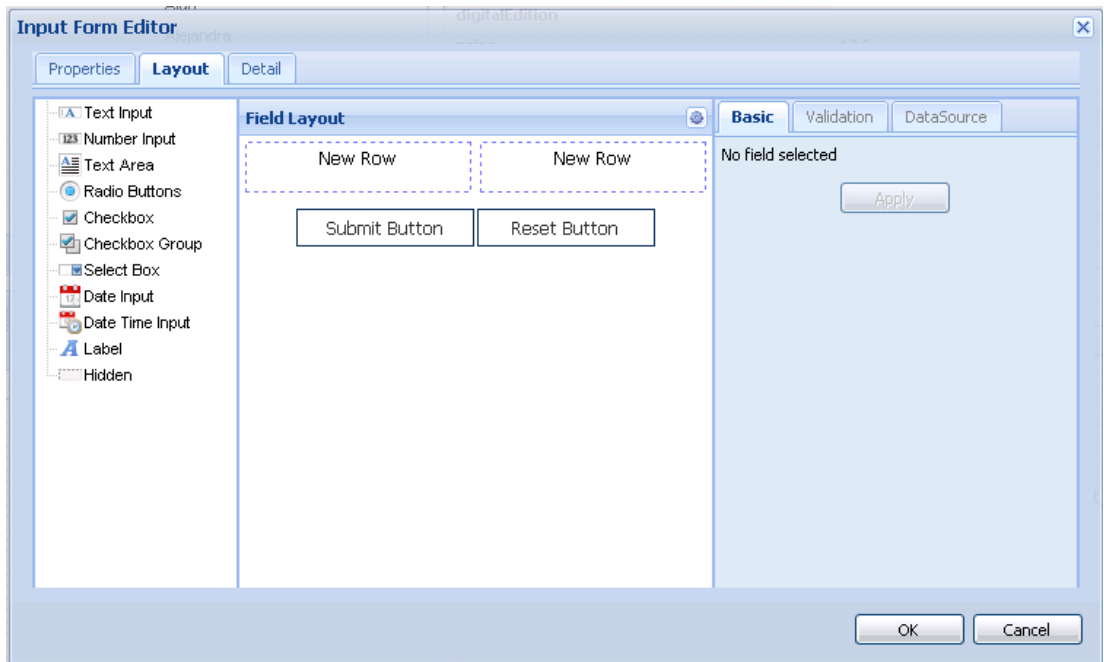
Creating a New Input Form Widget

Creating a Input Form widget involves dragging the relevant Input Form to the Dynamic Views Dashboard, and then using the Input Form Editor to configure the widget.

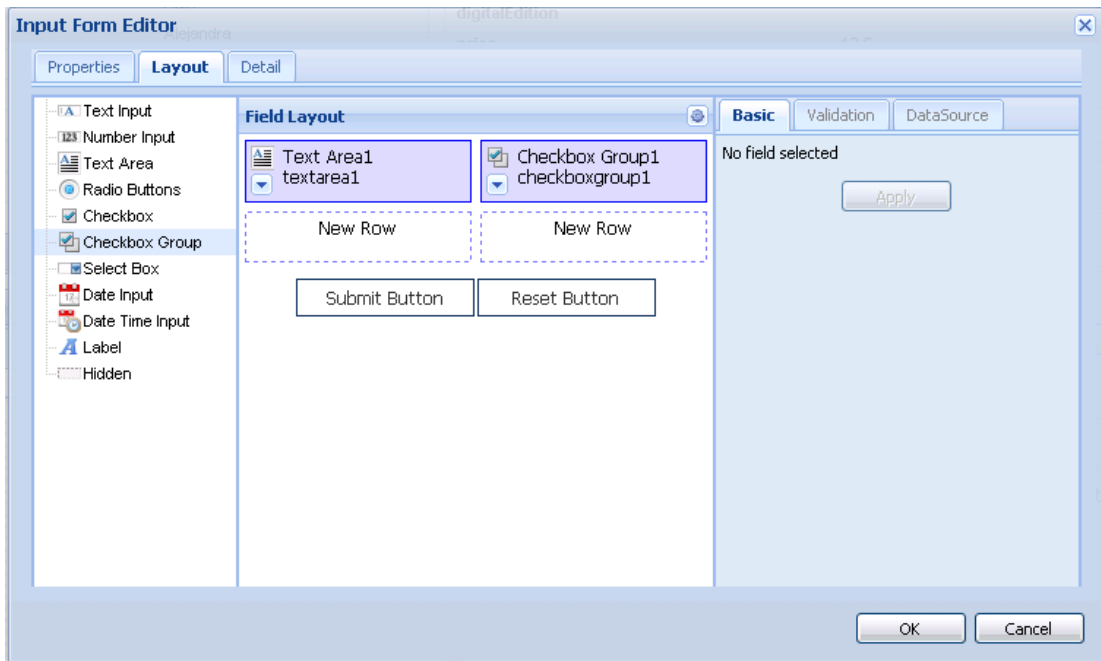
The following procedure explains the basic steps required to create a new Input Form widget. For more details about all the features of the *Layout* tab of the Input Form Editor, refer to [Managing the Layout of an Input Form Widget](#).

To create a new Input Form widget:

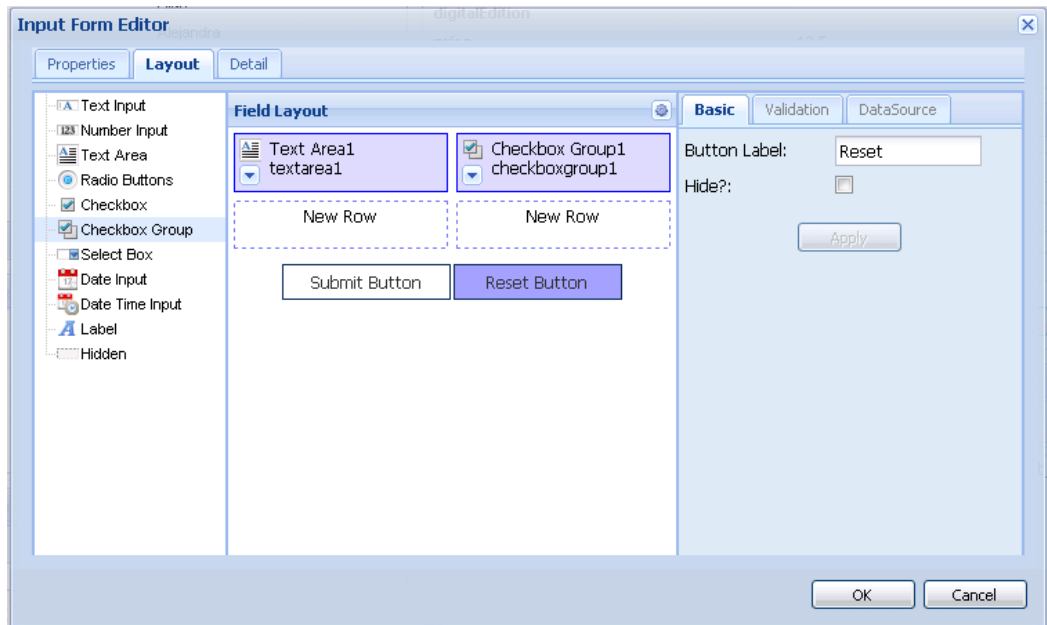
1. From the *Widgets* frame of the Resources Panel on the right side of the Dynamic Views Designer, drag the relevant Input Form (WorkSpace or non-WorkSpace-related) to the desired area of the Dashboard.
The Input Form Editor appears, with the *Layout* tab displayed by default.



2. From the list on the left side of the *Layout* tab, drag the relevant field component types to the desired areas of the *Field Layout* area.



3. Select a field component type, and configure the field's parameters:
 - a. Select the *Basic* tab, and define general parameters for the field. Parameters vary according to the field component type. For more information, refer to [Defining Basic Field Component Parameters](#).
 After defining basic parameters, click *Apply*.
 - b. If desired, select the *Validation* tab, and specify validation parameters for the field. Parameters vary according to the field component type. For more information, refer to [Defining Validation Field Component Parameters](#).
 After defining validation parameters, click *Apply*.
 - c. If desired, select the *DataSource* tab, and configure data source inputs for the field. The *DataSource* tab is relevant to only some field component types. For more information, refer to [Defining Validation Field Component Parameters](#).
 After configuring data source parameters, click *Apply*.
4. Repeat Step 3 for each field component of the Input Form.
5. If desired, update default configuration parameters for the *Submit* and/or *Reset* buttons:
 - a. Click the *Reset* or the *Submit* button. Button parameters appear in the frame on the right.



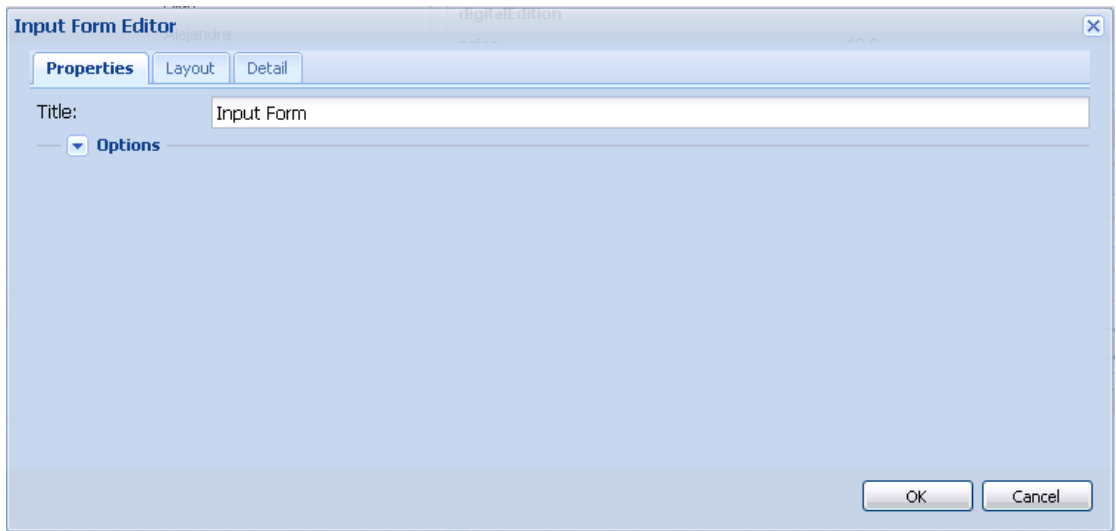
b. Modify the following parameters, as required:

- In the *Button Label* textbox, enter the desired text to appear on the button.
- To hide the button, click the *Hide* checkbox.

Then, click *Apply*.

6. At the top of the Input Form Editor, click *Properties*.

The *Properties* tab appears, with a default header displayed in the *Title* field.



7. If desired, replace the default title with the desired header.
8. At the top of the Input Form Editor, click *Detail*.
The *Detail* tab appears. In WorkSpace-related Input Forms, default values appear in the *Data Destination* frame.

Input Form Editor

Properties Layout **Detail**

Data Destination

URL to Post: /Odia1/datasource?method=getAllBooks&bean=booksStoreManager

Message Type: ☐ JSON ☒ Form Data

Form-wide Validation

Validation Formula:

Alert Message When the Validation Doesn't Pass:

Submit Reset OK Cancel

9. In the *URL to Post* textbox, enter the URL of the server to read and incorporate the data in the Input Form.
Then, to the right of *Message Type*, select the appropriate radio button: *JSON* or *Form Data*.
10. If desired, in the *Form-wide Validation* frame, specify parameters for validating the Input Form:
 - a. In the *Validation Formula* textbox, enter a validation formula that checks the validity of the entire Input Form.
The syntax of the formula is the same as JavaScript syntax, and field names can be used as variables in the scope.
Multiple formulas can be combined by using braces and logical operators.
 - b. In the *Alert Message...* field, enter the text of the message to appear when form-wide validation fails.

Form-wide Validation

Validation Formula: `(field1 && field2) || field3 == 'field3'`

Alert Message: Form-wide Validation Failed.
Input field1 and field2, or set field3 to 'field3'.

11. When all relevant parameters are specified, at the lower right corner of the Input Form Editor, click **OK**.

The Input Form is displayed in the Dashboard.

Managing the Layout of an Input Form Widget

The *Layout* tab of the Input Form Editor enables you to design the field layout of an Input Form, and configure the parameters of the field components. The *Layout* tab is used to:

- **Design a new Input Form:** The tab appears by default when an Input Form resource is dragged onto the Dashboard.
- **Update the layout and/or field components of an existing Input Form:** The tab is accessed by clicking the *Edit Configuration* icon at the upper right corner of the relevant Input Form widget.

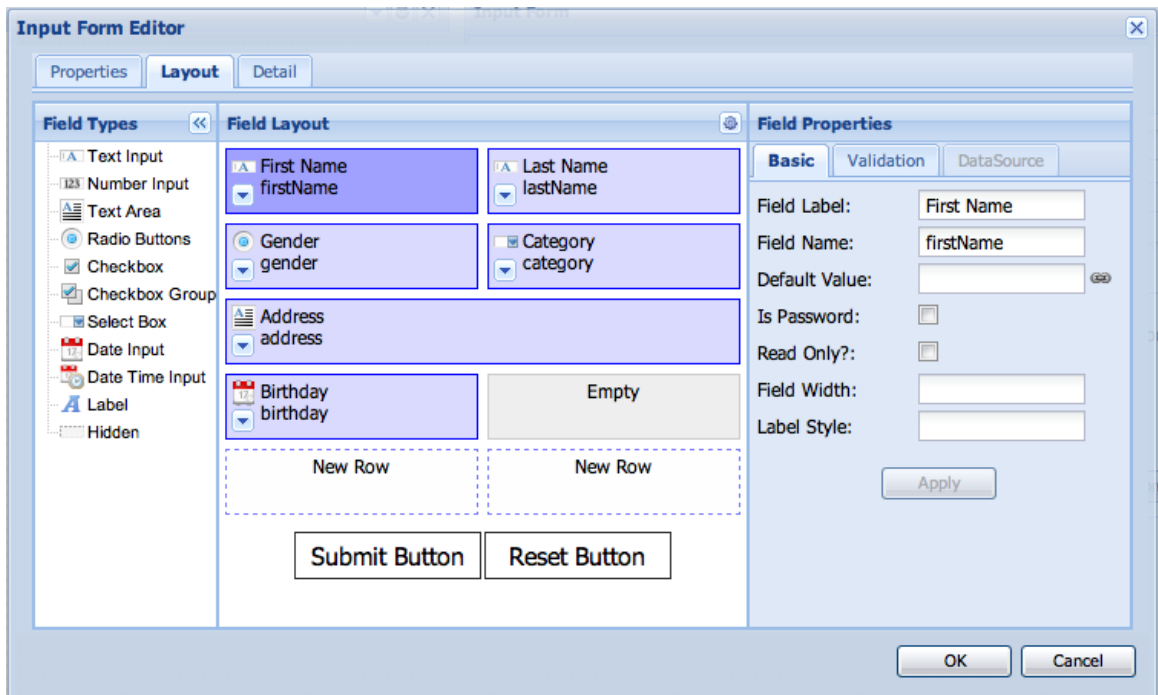
The following sections describe:

- [Editing the Field Layout of an Input Form](#)
- [Defining Basic Field Component Parameters](#)
- [Defining Validation Field Component Parameters](#)
- [Text Input Field Components](#)
- [Number Input Field Components](#)
- [Text Area Field Components](#)
- [Radio Group Field Components](#)
- [Checkbox Field Components](#)
- [Checkbox Group Field Components](#)
- [Select Box Field Components](#)
- [Date Input Field Components](#)

- [Date Input Field Components](#)
- [Label Field Components](#)
- [Hidden Field Components](#)
- [Defining a Data Source for Input Form Field Components](#)
- [Integrating an Input Form Event With Other Dashboard Widgets](#)

Editing the Field Layout of an Input Form

By default, the field components of an Input Form are arranged in two columns, with each component occupying the width of one column. You can use the Edit Layout options to modify the default layout.



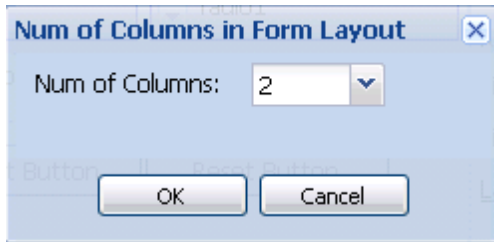
Changing Number of Columns

By default, the field components of an Input Form are arranged in two columns. The *Edit Layout* icon allows you to arrange the layout in up to 10 columns.

To change the number of columns of an Input Form:

1. At the upper right corner of the *Field Layout* frame of the relevant Input Form, click . Then, from the pop-up that appears, select *Change Column Num*.

The *Num of Columns in Form Layout* pop-up window appears.



2. From the drop-down list, select the desired number of columns, and then click *OK*.
The pop-up window disappears, and the selected number of columns are displayed in the Input Form.

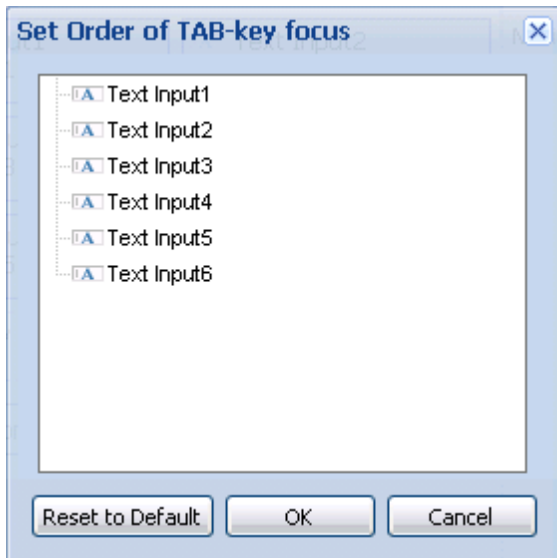
Changing Tab-key Order

By default, in Input Forms with multiple text fields, the tab-key order proceeds from left to right, and then top to bottom. The Set Tab-Key Order option allows you to customize the tab-key order.

To change the default tab-key order:

1. At the upper right corner of the *Field Layout* frame, click . A pop-up list appears.
2. From the pop-up list, select *Set Tab-Key Order*.

The *Set Order of TAB-key focus* window appears, with the field component names listed.



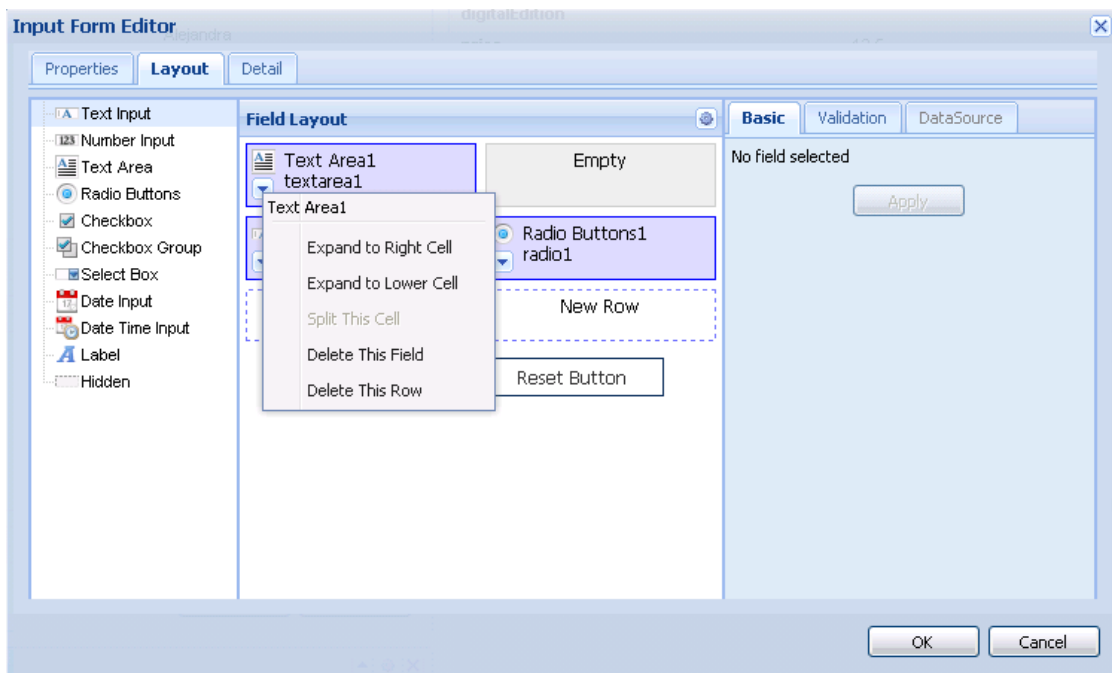
3. Drag-and-drop the field component names into the desired tab-key order, and then click *OK*.
The specified tab-key order is saved.

Editing Cells and Rows

By default, each field component of an Input Form occupies one row and one column. When required, you can expand field components, and delete cells or rows.

To edit the cells and rows of an Input Form:

1. From the *Field Layout* frame of the Input Form, right-click on the relevant field component. A pop-up list box appears.



- From the list box, select the required option. Available options are described in the following table.

Option	Description
Expand to Right/Left Cell	Extends the field component to span an additional column.
Expand to Lower Cell	Extends the field component to span an additional row.
Split This Cell	Reverses the Expand Cell process (relevant to expanded field components only).
Delete This Field	Removes the field component from the layout.
Delete This Row	Removes the row of the selected field component from the layout.

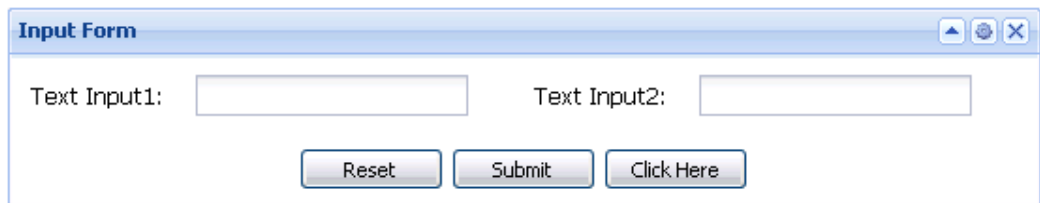
Customizing Input Form Buttons

By default, Input Forms have the following preconfigured buttons that appear in the lower right corner of the widget:

- **Reset:** Clears all data that has been entered in the form.
- **Submit:** Submits data to the server.

Input Form buttons can be customized to suit the purpose of the form. The default buttons can be removed, and custom buttons can be added as necessary. In addition, the position of the button group is adjustable (right, center, or left).

The following figure shows an example of an Input Form that has the button group in the center position. One custom button (*Click Here*) has been added to the default buttons.

A screenshot of a software window titled "Input Form". Inside the window, there are two text input fields. The first is labeled "Text Input1:" and the second is labeled "Text Input2:". Below these fields, centered horizontally, are three buttons: "Reset", "Submit", and "Click Here". The window has a standard title bar with a maximize button, a settings gear icon, and a close button.

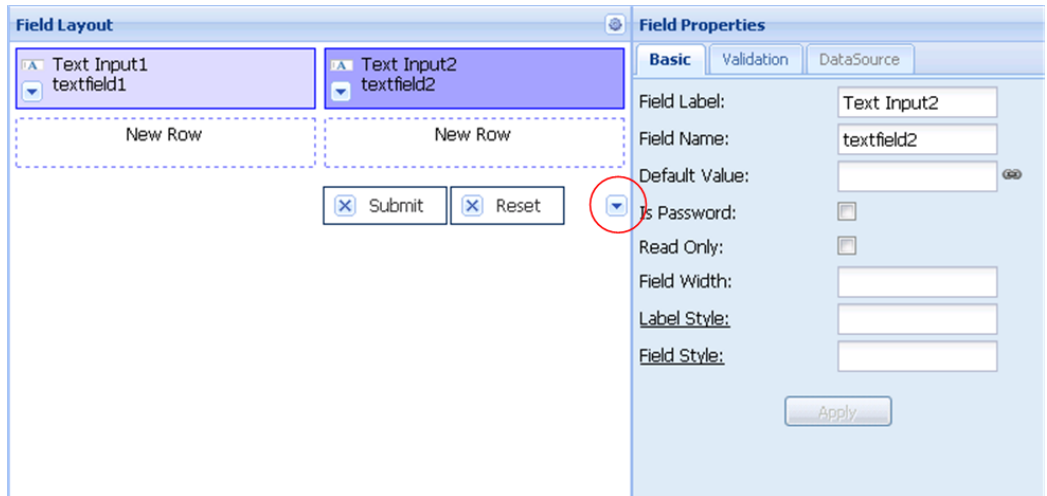
Working with Input Form Buttons

You can create custom buttons to perform any required function, for example, *Save*, *Edit*, *Delete*, etc. Custom buttons can also be used to create click events, by linking the buttons with other widgets in the Dashboard.

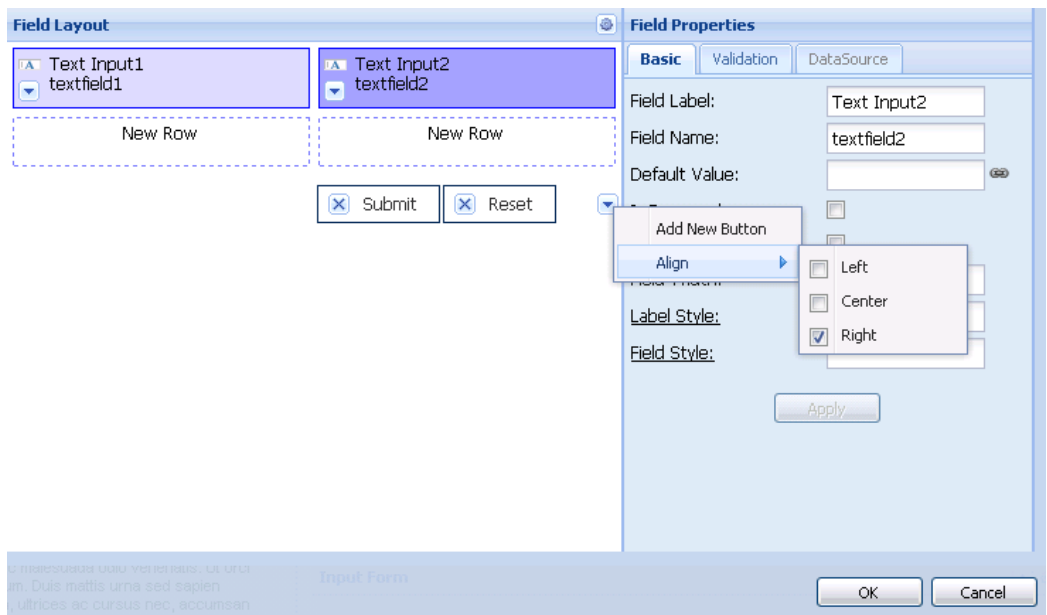
The following procedures explain how to manage Input Form buttons.

To change the position of the Input Form button group:

1. From the *Layout* tab of the Input Form Editor, click .



- From the popup menu, select *Align*. Then, select the checkbox of the preferred alignment (*Left*, *Center*, or *Right*).

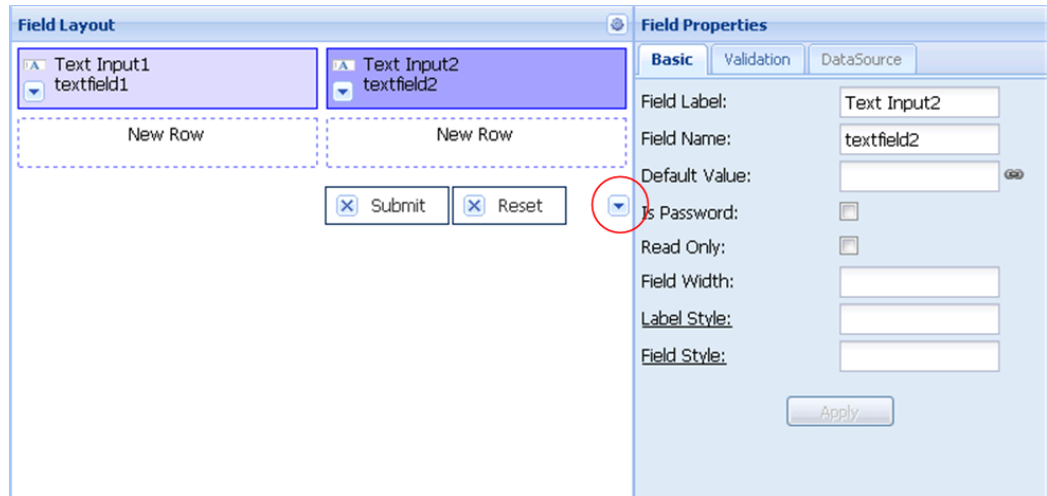


The group of buttons moves to the position of the selected alignment.

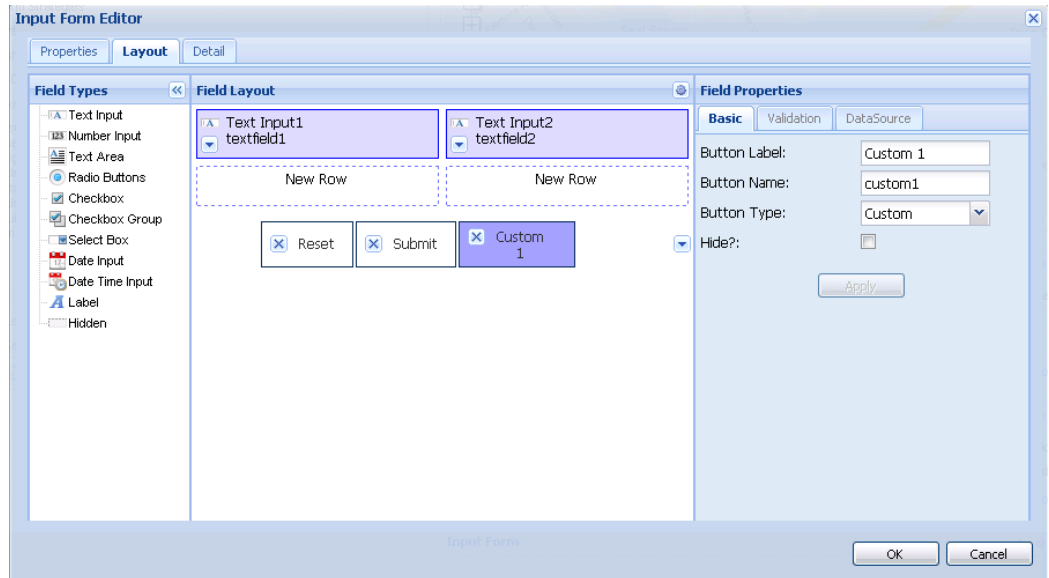
3. To save changes, click *OK*.

To add a custom button:

1. From the *Layout* tab of the Input Form Editor, click .



2. From the popup menu, select *Add New Button*.
A new button appears in the button group, with the default label of *Custom*.
3. To define button properties, select the new button.
In the *Field Properties* frame, the *Basic* tab is enabled, with default properties listed.



4. Define the following properties:

- **Button Label:** The name that is displayed in the Input Form.
- **Button Name:** The name that is used for coding purposes.
- **Button Type:** Select *Custom*. (The *Reset* and *Submit* options are used only to re-create default buttons, in the event that they have been deleted.)
- **Hide:** Select this checkbox to prevent the button from being displayed in the Input Form.

Then, click *Apply*.

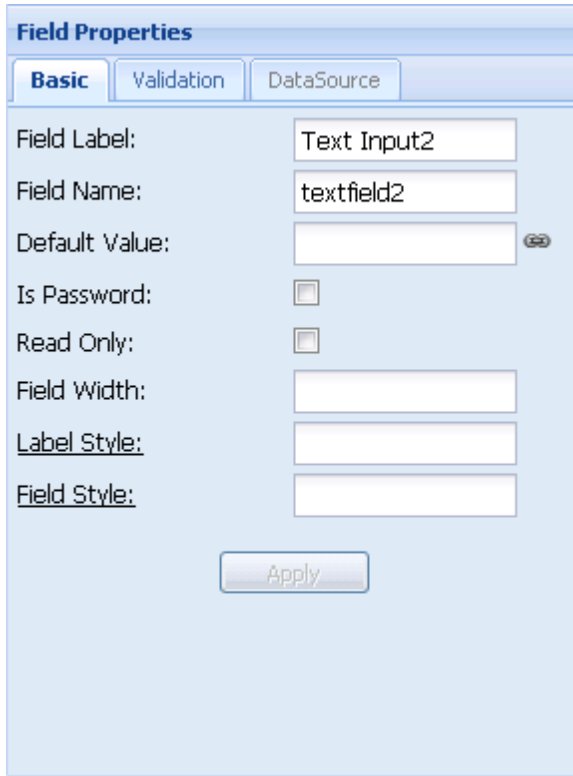
5. To save changes, click *OK*.

To delete a button:

- From the *Layout* tab of the Input Form Editor, in the button that you want to delete, click . The relevant button is removed from the Input Form button group.

Defining Basic Field Component Parameters

The *Basic* tab of the Input Form Editor enables you to specify general parameters of a field component, such as its title, width, and number of rows.



The screenshot shows the 'Field Properties' dialog box with the 'Basic' tab selected. The dialog has three tabs: 'Basic', 'Validation', and 'DataSource'. The 'Basic' tab contains the following fields and controls:

- Field Label:** Text Input2
- Field Name:** textfield2
- Default Value:** (empty text box) with a small icon to the right.
- Is Password:** ☐
- Read Only:** ☐
- Field Width:** (empty text box)
- Label Style:** (empty text box)
- Field Style:** (empty text box)
- Apply** button

In a WorkSpace-related Input Form, the *Field Name* property is a drop-down list (instead of a textbox), displaying the specified objects of the relevant method in the Feature Manager class on which the Input Form is based. Acceptable prefixes are:

- **text** (for text fields)
- **num** (for any kind of numeric field)
- **date** (for date fields)
- **boolean** (for booleans)

Parameter	Value
Field Label:	Title
Field Name:	text_title
Default Value:	text_ISBN
Is Password:	text_author
Read Only:	num_price
Field Width:	text_title

The parameters that appear in the *Basic* tab vary according to the field component type. For a full description of basic parameters, refer to the sections detailing the specific field components.

Note: After modifying parameters in the *Basic* tab, click *Apply* before continuing, to ensure that changes are saved.

Defining Validation Field Component Parameters

The *Validation* tab of the Input Form Editor enables you to specify validation parameters for a field component. Parameters can include minimum/maximum number of characters, entry type (email, URL, etc.), date/time requirements, and so on.

The tab also allows you to specify a customized validation error message. If no customized message is specified, a default error message is used.

Parameter	Value
Required:	☐
Alert Message for Required Error:	
Apply	Apply

The parameters that appear in the *Validation* tab vary according to the field component type. For a

full description of basic parameters, refer to the sections detailing the specific field components.

Parameters specified in the *Validation* tab apply to specific field components only. Form-wide validation parameters are defined using the *Detail* tab of the Input Form Editor.

Note: After modifying parameters in the *Validation* tab, click *Apply* before continuing, to ensure that changes are saved.

Text Input Field Components

This type of field component accepts one line of text as input.



Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.

Property	Type	Description
Field Width	Number	Width of the field (in pixels).
Password	Checkbox	Indicates whether or not the field is used to input secret information. When the checkbox is checked, the input text is masked.
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.
Alert Message	Text area	Text of the message that appears when the validation check is invalid.
Field Type	Drop-down list	The required format of the text input.
Min Length	Number	Minimum number of characters of the input text.
Max Length	Number	Maximum number of characters of the input text.
Regular Expression	Text	A custom regular expression to validate the input text.

Parameter	Type	Description
Alert Message	Text area	Text of the message that appears when the regular expression check is invalid.

Number Input Field Components

This type of field component accepts numeric values as input. Decimal input and decimal precision can be configured.



Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.
Field Width	Number	Width of the field (in pixels).

Property	Type	Description
Allow Decimals	Checkbox	Indicates whether or not decimals can be entered in the field. The default value is false.
Decimal Precision	Number	The number of digits allowed to be entered following the decimal point.
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.
Alert Message	Text area	Text of the message that appears when the validation check is invalid.
Min Value	Number	Minimum value of the input number.
Max Value	Number	Maximum length of the input number.

Text Area Field Components

This type of field component accepts multiple lines of text as input.



Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking allows you to bind a value from another widget.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.
Field Width	Number	Width of the field (in pixels).
Field Height	Number	Height of the field (in pixels).

Property	Type	Description
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

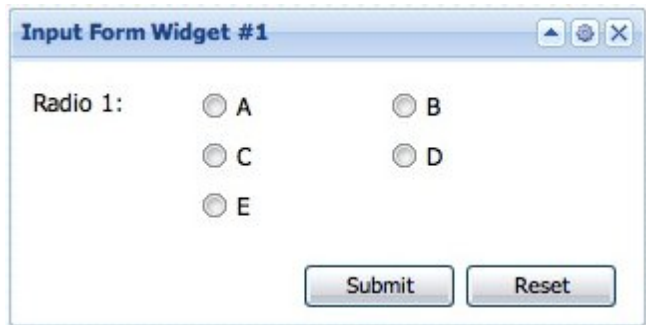
Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.
Alert Message	Text area	Text of the message that appears when the validation check is invalid.
Min Length	Number	Minimum number of characters of the input text.
Max Length	Number	Maximum number of characters of the input text.
Regular Expression	Text	A custom regular expression to validate the input text.
Alert Message	Text area	Text of the message that appears when the regular expression check is invalid.

Radio Group Field Components

This type of field component accepts a single choice from listed options.



Generally, a data source is used for listing the control options. For details, refer to [Defining a Data Source for Input Form Field Components](#).

Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.

Property	Type	Description
Column num of Controls	Drop-down list	Indicates number of columns in which control buttons appear. The default value is 1.
Control Alignment	Drop-down list	Alignment of the radio buttons in the layout. Radio buttons can be aligned top to bottom, or left to right.
Num of Controls	Number	The maximum number of radio buttons to appear in the display. This is relevant when the field component is bound to a data source (Defining a Data Source for Input Form Field Components).
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.
Alert Message	Text area	Text of the message that appears when the validation check is invalid.

Checkbox Field Components

This type of field component accepts a true/false value by checking/clearing the field.



Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking allows you to bind a value from another widget.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation parameters are not relevant for this type of field component.

Checkbox Group Field Components

This type of field component accepts multiple choices from listed options.



Generally, a data source is used for listing the control options. For details, refer to [Defining a Data Source for Input Form Field Components](#).

Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.

Property	Type	Description
Column num of Controls	Drop-down list	Indicates number of columns in which control buttons appear. The default value is 1.
Control Alignment	Drop-down list	Alignment of the checkboxes in the layout. Checkboxes can be aligned top to bottom, or left to right.
Num of Controls	Number	The maximum number of checkboxes to appear in the display. This is relevant when the field component is bound to a data source (Defining a Data Source for Input Form Field Components).
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

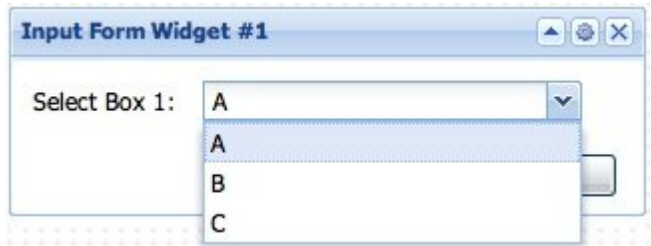
Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.
Alert Message	Text area	Text of the message that appears when the validation check is invalid.

Select Box Field Components

This type of field component accepts a single choice from options in a drop-down list.



Generally, a data source is used for listing the control options. For details, refer to [Defining a Data Source for Input Form Field Components](#).

Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.
Field Width	Number	Width of the field (in pixels).
Num of Rows	Number	The maximum number of options to appear in one page of the drop-down list.

Property	Type	Description
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.
Alert Message	Text area	Text of the message that appears when the validation check is invalid.

Date Input Field Components

This type of field component accepts a date value as input.



The screenshot shows a standard Java Swing window with a title bar. Inside, there's a text label 'Date1:' followed by a date input field. The field contains the text '20/07/1977' and has a small calendar icon to its right. Below the input field, there are two rectangular buttons with rounded corners, labeled 'Submit' and 'Reset'.

Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.
Date Format for Default Value	Drop-down list	The default format for inputting and displaying the date value.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.
Date Format	Drop-down list	The format for inputting and displaying the date value.
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.

Parameter	Type	Description
Alert Message	Text area	Text of the message that appears when the validation check is invalid.
Min Date	Number	The date that defines the lower threshold of the date range. Dates that are further in the past than this date are invalid.
Max Date	Number	The date that defines the upper threshold of the date range. Dates that are further in the future than this date are invalid.

Date Time Input Field Components

This type of field component accepts both date and time values as input.



The screenshot shows a window titled "Input Form Widget #1". Inside, there is a label "Date Time 1:" followed by two input fields. The first field contains the date "16/07/1977" and has a calendar icon to its right. The second field contains the time "07:30" and has a dropdown arrow to its right. Below these fields are two buttons: "Submit" and "Reset".

Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.

Property	Type	Description
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.
Date Format for Default Value	Drop-down list	The default format for inputting and displaying the date value.
Read Only	Checkbox	Indicates whether or not the field is read-only. The default value is false.
Date Format	Drop-down list	The format for inputting and displaying the date value.
Time Format	Drop-down list	The format for inputting and displaying the time value.
Label Style	Text	CSS style notation that is applied to the field label.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation Parameters

Validation parameters (defined in the *Validation* tab) include:

Parameter	Type	Description
Required	Checkbox	Indicates whether the field requires input, or whether it can be left blank.

Parameter	Type	Description
Alert Message	Text area	Text of the message that appears when the validation check is invalid.
Min Date	Number	The date that defines the lower threshold of the date range. Dates that are further in the past than this date are invalid.
Max Date	Number	The date that defines the upper threshold of the date range. Dates that are farther in the future than this date are invalid.
Min Time	Drop-down list	The time that defines the lower threshold of the time range. Times that are further in the past than this time are invalid.
Max Time	Drop-down list	The time that defines the upper threshold of the time range. Times that are farther in the future than this time are invalid.

Label Field Components

This type of field component displays static label and value pairs.



The screenshot shows a standard Java Swing window with a title bar that says "Input Form Widget #1". Inside the window, there is a text label "Label1:" followed by the text "Hello, World". At the bottom of the window, there are two buttons: "Submit" and "Reset".

Basic Properties

Basic properties (defined in the *Basic* tab) include:

Property	Type	Description
Field Label	Text	Label displayed to the left of the field.
Field Text	Text	The text value displayed. HTML is permitted. Clicking  allows you to bind a value from another widget.
Label Style	Text	CSS style notation that is applied to the field label.
Text Style	Text	CSS style notation that is applied to the field text.
Field Style	Text	CSS commands to customize the appearance of the field (padding, border, background color, etc.).

Validation parameters are not relevant for this type of field component.

Hidden Field Components

This field component type does not display any result in a widget. Instead, it holds a field name and value pair for submitting form values to the server side, or for firing as event data.

Basic properties (defined in the *Basic* tab) include:

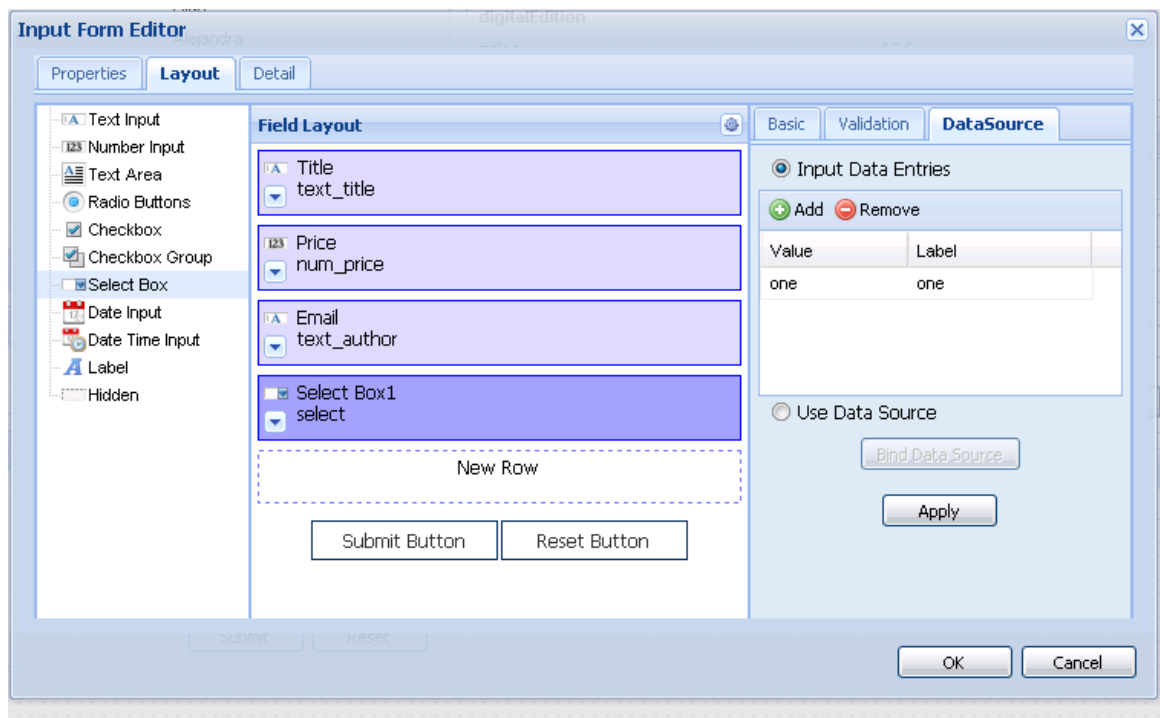
Property	Type	Description
Field Name	Text	Internal name of the field. Only characters that are valid variable names are allowed.

Property	Type	Description
Default Value	Text	Default value of the field. Clicking  allows you to bind a value from another widget.

Validation parameters are not relevant for this type of field component.

Defining a Data Source for Input Form Field Components

The *DataSource* tab of the Input Form Editor enables you to define a data source from which a field component obtains its control options. For example, you can configure a specific Select Box to get the options for its drop-down box from a predefined data source.



The following field component types can accept data sources for listing control options:

- Checkbox Group
- Radio Button Group
- Select Box

The *DataSource* tab provides the following methods for defining the data source to be used by a field component:

- **Input Data Entries:** Label and value pairs are entered manually in the Input Form Editor. For details, refer to [Creating Data Entries](#).
- **Use Data Source:** A data source is bound to the relevant field component. For details, refer to [Binding a Data Source to a Field Component](#).

Creating Data Entries

This method involves manually entering label and value pairs in an inline table. Generally, the data entry method is used to define the options for the drop-down list in a Select Box field component.

To create a data entry grid:

1. From the *Field Layout* frame of the Input Form Editor, select the relevant field component. Then, on the right side of the Input Form Editor, select the *DataSource* tab.
The *DataSource* tab is displayed, with the *Input Data Entries* radio button selected by default.

Basic Validation **DataSource**

☒ Input Data Entries

Value	Label
one	check

☐ Use Data Source

- In each row of the inline grid, enter relevant data for the following parameters:
 - Value:** The value that is sent to the server.
 - Label:** The text that is displayed in the field control(s) of the field component.

Use the *Add* and *Remove* icons to edit the grid by adding rows to the grid, and/or removing selected rows.
- When the grid is completed as required, click *Apply*.
Changes are saved.

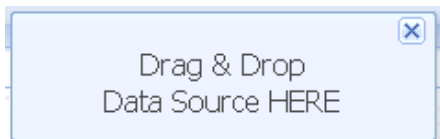
Binding a Data Source to a Field Component

This method involves binding a predefined data source to the field component, enabling the field component controls to be retrieved from the server side.

To bind a data source to a field component:

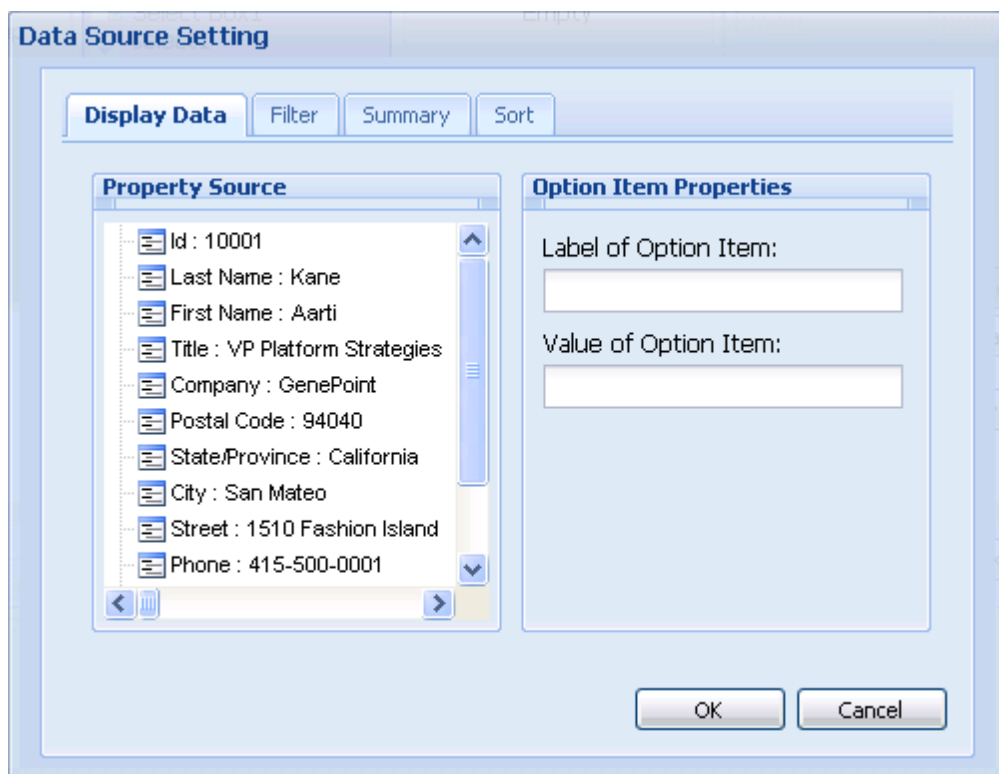
- From the *Field Layout* frame of the Input Form Editor, select the relevant field component. Then, on the right side of the Input Form Editor, select the *DataSource* tab.
The *DataSource* tab is displayed, with the *Input Data Entries* radio button selected by default.
- Select the *Use Data Source* radio button. The buttons on the lower portion of the *DataSource* tab are enabled.
- Click *Bind Data Source*.

The Drag & Drop Data Source HERE pop-up appears.



4. From the Resources Panel of the Dynamic Views Designer, click Data Sources. Then, drag the relevant data source to the pop-up. When the background of the pop-up turns pink, release the mouse pointer.

The *Data Source Setting* dialog appears, with the *Display Data* tab displayed by default.



5. To display the data source result in a Select Box field, in the Option Item Properties frame, enter the relevant label and value pairs.
6. If desired, define relevant values in the Parameters, Filter, Summary, and Sort tabs. Available tabs vary, according to the data source definition.

7. When all values are specified, at the bottom of the *Data Source Setting* dialog, click *OK*. The *Data Source Setting* dialog disappears. The *Edit Data Source Config* and *Remove Data Source* buttons appear at the bottom of the *DataSource* tab.

The screenshot shows the 'DataSource' tab of a dialog box. At the top are three tabs: 'Basic', 'Validation', and 'DataSource'. The 'DataSource' tab is active. Below the tabs is a section titled 'Input Data Entries' with a radio button. Below this is a table with two columns: 'Value' and 'Label'. The first row contains the text 'one' in both columns. Above the table are 'Add' and 'Remove' buttons. Below the table is a radio button labeled 'Use Data Source'. At the bottom are three buttons: 'Edit Data Source Config', 'Remove Data Source', and 'Apply'.

Value	Label
one	one

8. Click *Apply*.
Changes are saved.

Integrating an Input Form Event With Other Dashboard Widgets

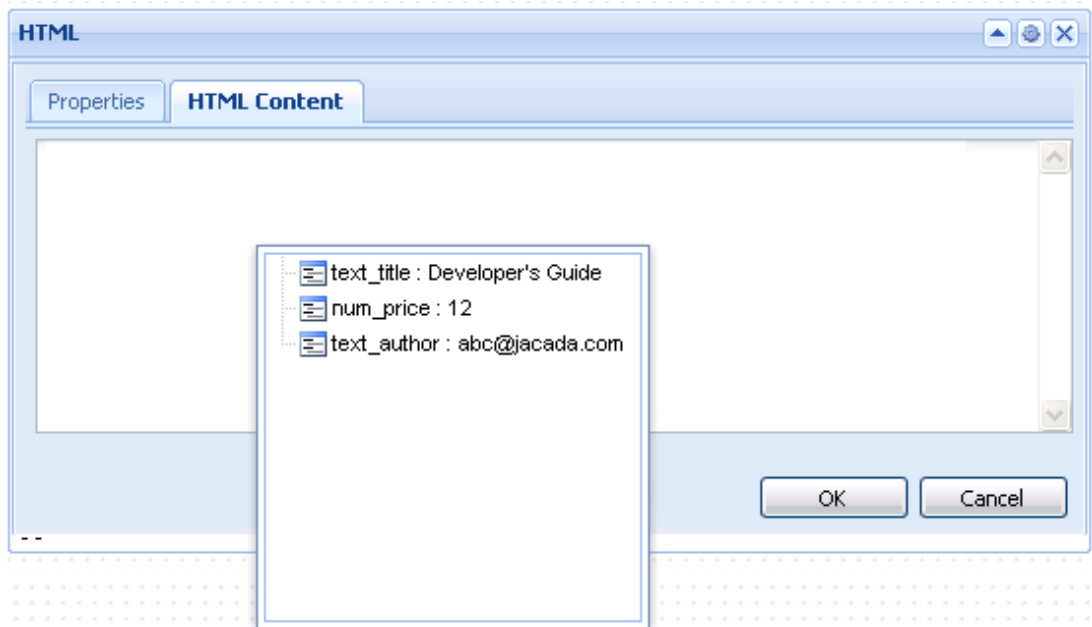
The Event feature enables you to capture specific Input Form events. You can then bind the events with other widgets in the Dashboard, enabling these widgets to be updated when the relevant Input Form event fires. Input Form widgets support the capture of the following event types:

- **Change Event:** Fires when a field component changes its value.
- **Submit Event:** Fires when the *Submit* button is clicked.
- **Success/Failure Event:** Fires when the post request response has returned from the data destination URL.

The following procedure provides an example of how to bind a Success event with another widget on the Dashboard.

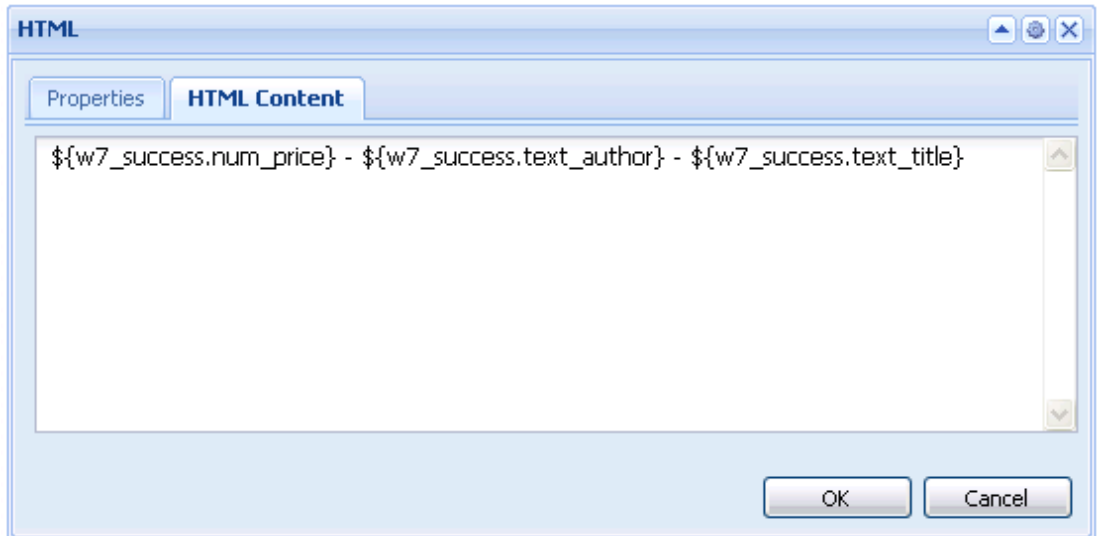
To integrate an Input Form event with another widget:

1. At the upper right corner of the target widget, click . When the widget expands, select the tab to which you want to bind event data (e.g., *Display Data*, *HTML Content*, etc.).
2. At the upper left corner of the Dynamic Views Designer, expand the *My Dashboard* menu, and select *Start Event Capturing*.
Event capturing is enabled.
3. In the relevant Input Form widget, click *Submit*.
A *Now Capturing Event* pop-up appears over the widget.
4. Drag the *Now Capturing Event* pop-up to the desired area of the target widget. When the background of the widget turns pink, release the mouse pointer.
A pop-up window appears, displaying all defined methods and values.



5. From the pop-up window, double-click the desired method to be bound with the target widget.
The selected method is displayed in the target widget.
6. To bind additional methods to the target widget, while pressing *Alt*, drag the *Now Capturing Event* pop-up to the desired area of the target widget. When the background of the widget turns pink, release the mouse pointer.

Then, from the pop-up window, double-click the desired method. The method is added to the target widget.

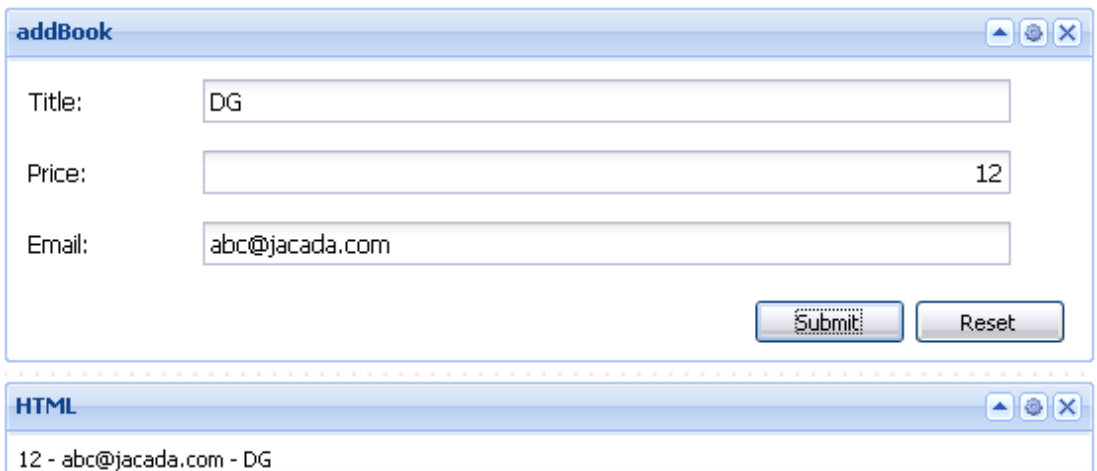


7. When all desired methods are displayed in the target widget, from the *My Dashboard* menu, select *Stop Event Capturing*.

Event capturing is disabled.

8. At the bottom of the target widget, click *OK*.

The widget displays selected content, as it appears in runtime.

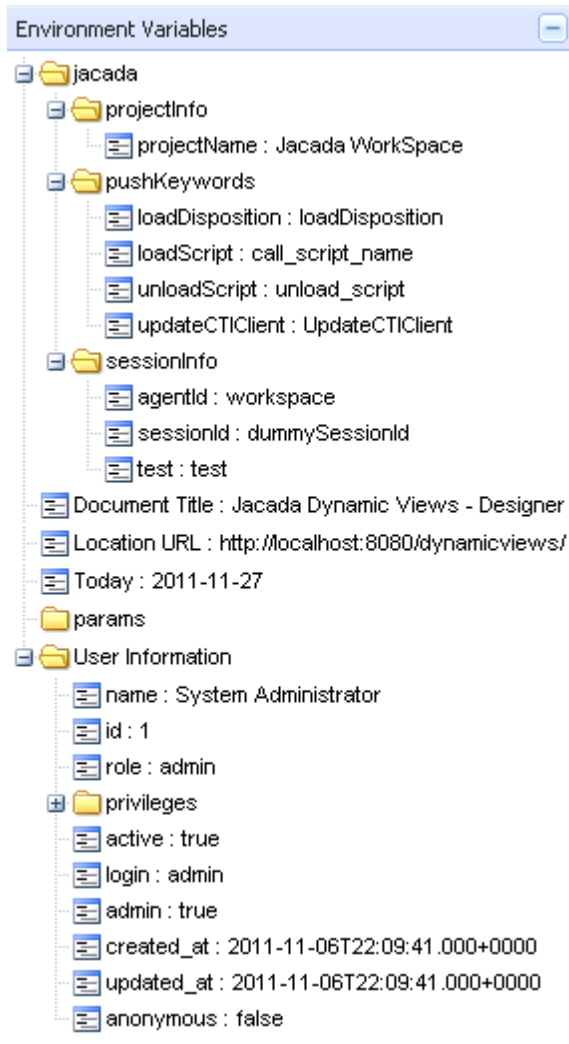


Managing Environment Variables for Dynamic Views

Environment variables are fields that contain information associated with a user's account, such as user name, user email, organization name, etc. These variables are loaded when Dynamic Views Dashboards are loaded, and their values remain constant throughout a Jacada WorkSpace session.

The Dynamic Views Designer supports the ability to use environment variables when designing Dashboard widgets. For example, if you want agents to view only records that they own, you can add a filter rule to the Email environmental variable, where Email equals `${env.userInfo.email}`, and then bind this variable to the appropriate filter column in the relevant widget.

Environment variables are listed in the *Environment Variables* frame of the Resources Panel. Several types of environment variables are provided with the Dynamic Views Designer. In addition, you can create environment variables that are related to a specific Jacada WorkSpace project. These WorkSpace-related variables are listed in the *Jacada* directory.



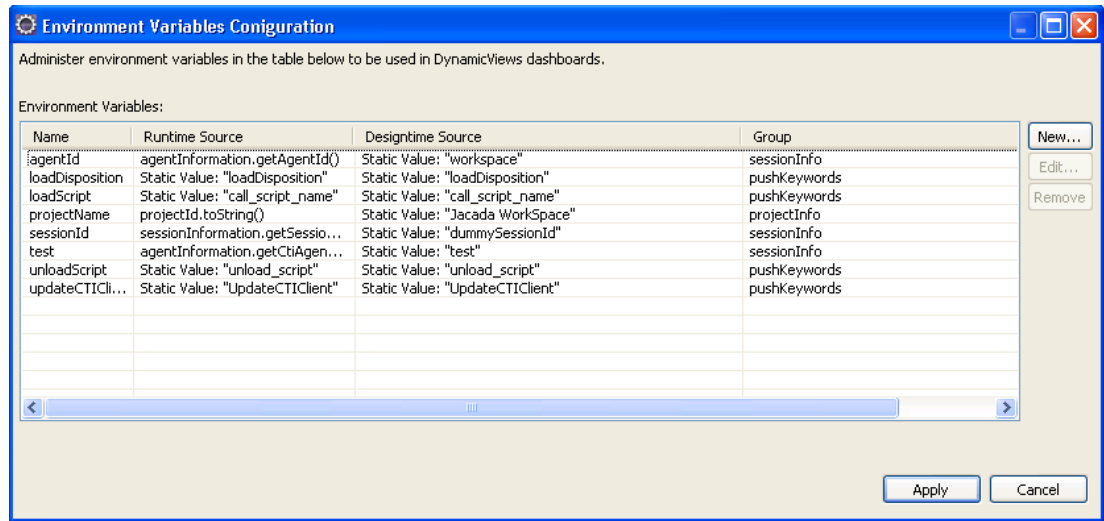
WorkSpace-related variables are created and modified using the Environment Variables Configuration wizard, which is accessed from the Jacada WorkSpace project.

To create WorkSpace-related environment variables:

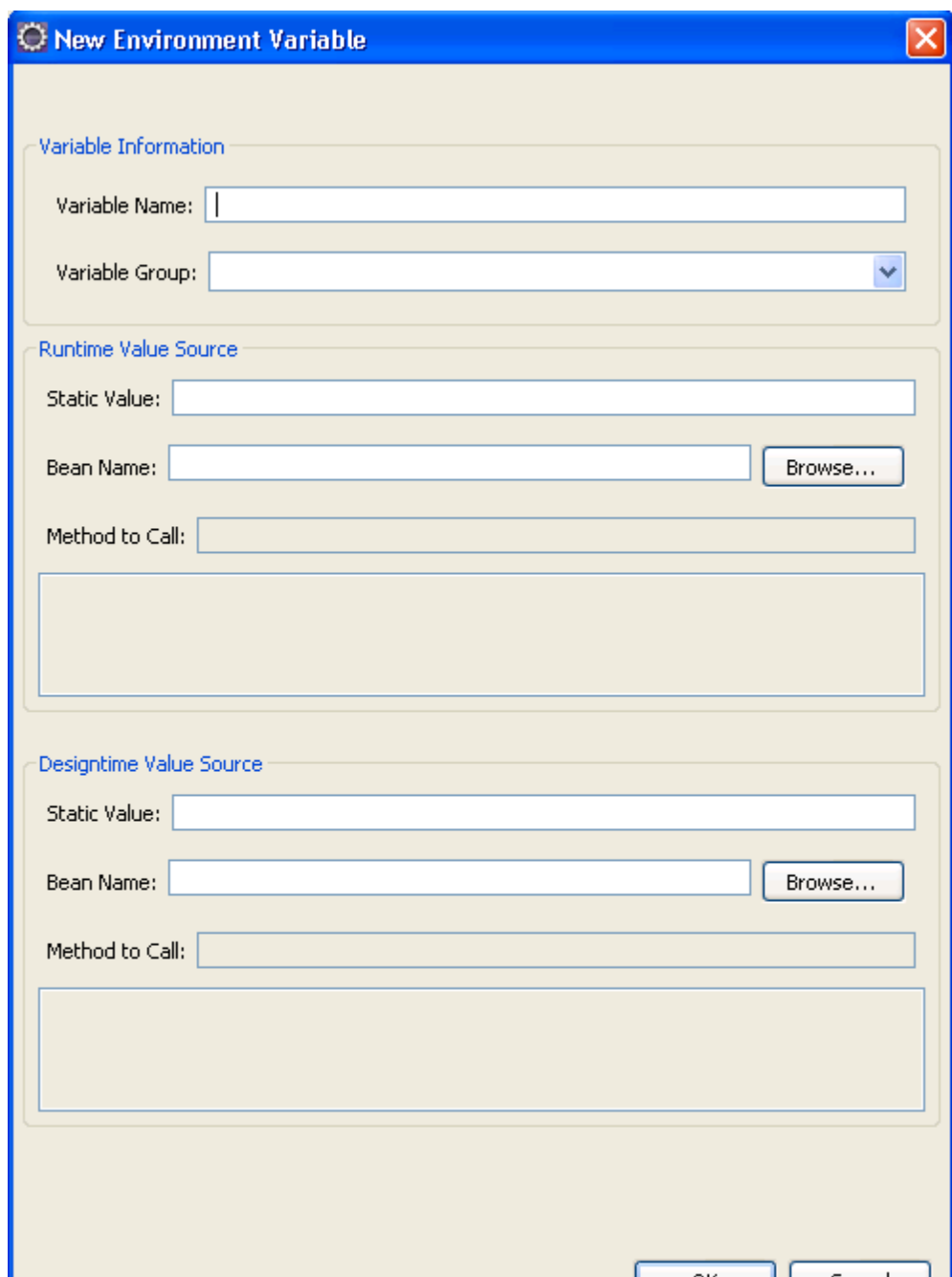
1. From the Eclipse IDE, right-click on the relevant project name, and select *Jacada WorkSpace > Jacada DynamicViews > Environment Variables Configuration*.

The Environment Variables Configuration wizard appears, displaying all previously-defined

variables.



2. At the right side of the Environment Variables Configuration wizard, click *New*.
The *New Environment Variable* dialog appears.

The image shows a 'New Environment Variable' dialog box with a blue title bar and a close button. It contains three sections: 'Variable Information', 'Runtime Value Source', and 'Designtime Value Source'. Each section has input fields for 'Static Value', 'Bean Name', and 'Method to Call', along with a 'Browse...' button. The 'Variable Information' section also includes a 'Variable Group' dropdown menu. The 'Runtime Value Source' and 'Designtime Value Source' sections each have a large text area below the input fields. At the bottom right, there are 'OK' and 'Cancel' buttons.

New Environment Variable

Variable Information

Variable Name:

Variable Group:

Runtime Value Source

Static Value:

Bean Name:

Method to Call:

Designtime Value Source

Static Value:

Bean Name:

Method to Call:

3. In the *Variable Name* textbox, enter a name for the new variable, as it will appear in the Resources Panel of the Dynamic Views Designer.
4. In the *Variable Group* field, enter a name for the folder that will contain the new variable. Alternatively, select a predefined folder from the drop-down list.
5. In the *Runtime Value Source* frame, specify a runtime value, using ONE of the following techniques:
 - In the *Static Value* textbox, enter the value to be returned.
 - In the *Bean Name* field, enter the name of the Spring Bean to be returned. Clicking *Browse* allows you to navigate to the desired bean. A list of associated methods is displayed in the list box below the *Method to Call* field.

Runtime Value Source

Static Value:

Bean Name:

Method to Call:

- getAccountNumber() : String
- getAccounts() : HashMap
- getAgentInformation() : AgentInformation
- getANI() : String
- getAuthorizedContact() : String
- getBillingCity() : String
- getBillingCounty() : String
- getBillingOther() : String

- From the list box, select a method. The selected method appears in the *Method to Call* field.
6. In the *Design-time Value Source* frame, specify a design-time value, using ONE of the following techniques:
 - In the *Static Value* textbox, enter the value to be returned.
 - In the *Bean Name* field, enter the name of the Spring Bean to be returned. Clicking *Browse* allows you to navigate to the desired bean. A list of associated methods is displayed in the list box below the *Method to Call* field.
 - From the list box, select a method. The selected method appears in the *Method to Call* field.

Note: The object types of the design-time method and the runtime method must match.

7. Click *OK*.

The environment variable is listed in a new row in the Environment Variables Configuration wizard.

8. If necessary, make modifications to the grid:

- **Add a variable:** Click *New*, and refer to Steps 2-7 above.
- **Modify variable parameters:** From the grid, select the relevant variable, and then click *Edit*. Alternatively, double-click the relevant row in the grid. In the *Edit Environment Variable* dialog that appears, update relevant parameters, and then click *OK*. (Refer to Steps 3-7 above).
- **Delete a variable:** From the grid, select the relevant variable, and then click *Remove*.

9. When all variables are defined as desired, at the bottom of the Environment Variables Configuration wizard, click *Apply*.

10. The Environment Variables Configuration wizard disappears, and the project is republished. When the Dynamic Views Designer is accessed, the new environment variables are listed in the Resources Panel, in the specified directories.

Customizing Dashboards Programatically

Jacada WorkSpace exposes several JavaScript APIs and a special .css file that you can use to customize the behavior of a dashboard and its widgets.

setActiveTab API

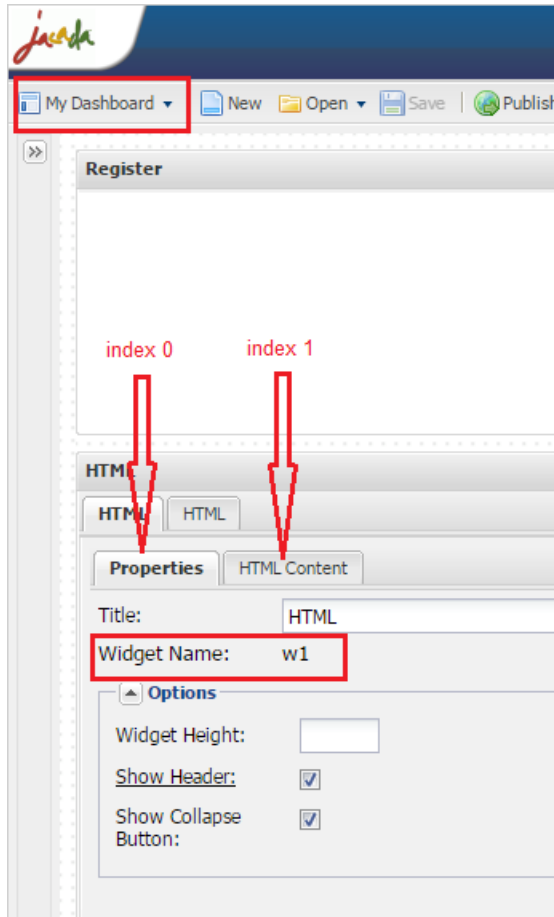
The following function sets the tab that will be active in a widget that has multiple tabs:

```
setActiveTab(String widgetName, int index)
```

Parameter	Description
String widgetName	Name of the relevant widget.
int index	Index number of the tab to be made active.

In the following example, the second tab (*HTML Content*) in the widget named *w1* is made the active tab:

```
setActiveTab('w1', 1);
```



getFormInputField API

The following function returns a reference to a specific input form field in a widget. You can then use this reference to put a value in the field, disable the field, etc.

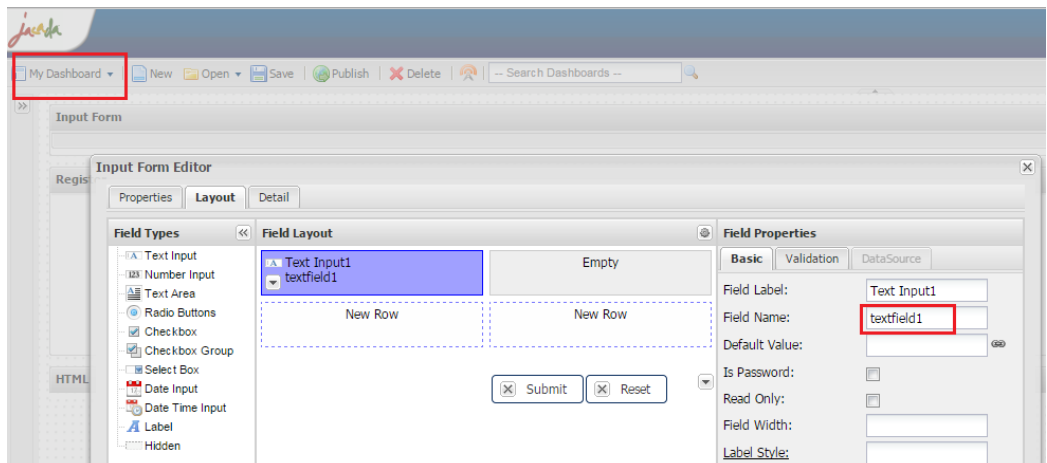
Note: The reference returned is to the ExtJS field (not to the DV field that wraps the ExtJS field).

```
getFormInputField(String widgetName, String fieldName)
```

Parameter	Description
String widgetName	Name of the relevant widget.
String fieldName	Name of the relevant input form field.

The following example shows how to get a reference for the *textfield1* input field in the widget named *w4*:

```
getFormInputField( 'w4', 'textfield1')
```



setVisible API

The following function shows or hides a specific widget. A common use case for the function is showing/hiding a widget according to the user's role and privileges.

The behavior configured in the project code overrides the settings of Dashboards containing predefined hidden widgets.

```
setVisible(String widgetName, boolean value)
```

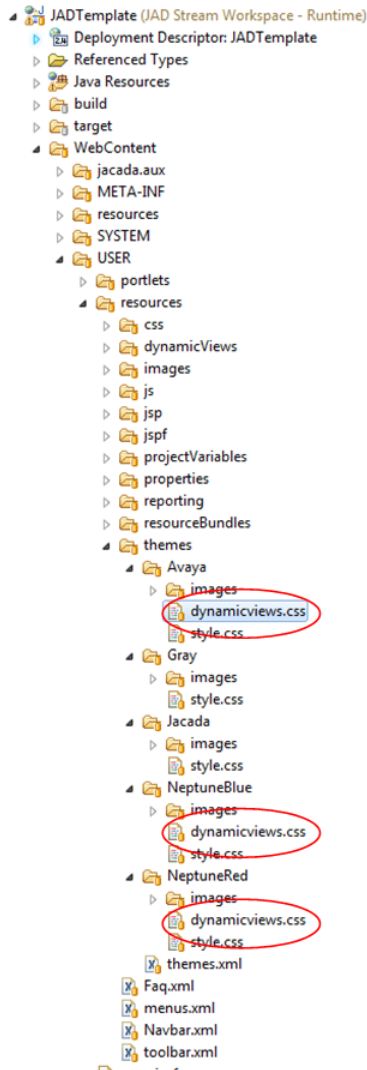
Parameter	Description
String widgetName	Name of the relevant widget.
boolean value	true or false.

In the following example, the widget named *w4* is hidden:

```
setVisible('w4', false)
```

Dynamic Views CSS

Jacada WorkSpace provides a customized .css file that changes the look and feel of a Dashboard (widget titles, buttons, and so on) to match the style of the WorkSpace interface. The file, which is called **dynamicviews.css**, is located in the **WebContent** directory inside the **Avaya**, **NeptuneBlue** and **NeptuneRed** themes folders.



As the .css file is located within the WorkSpace project and not inside the **dynamicviews.war** file, Dynamic Views automatically calculates the path to the styles and adds it to the DOM.

The following figure shows an example of a Dynamic Views Dashboard that is styled to match the look of WorkSpace:

Favorites ▾Help ▾

Customer Name: Joe Smith
Email: joe@gmail.com
Phone: 7701467683

7701467683

Manual Mode

Status: In Call

End CallHoldTransfer

InfoDealerStatusNotificationAction

CUSTOMERINFORMATIONTAB

Customer

Social Security Number

First Name

Last Name

Telephone

E-mail

Payments

NaN

NaN

NaN

NaN

NaN

0

Accounts

Account Id

Account Holder

Account Balance

Page 1 of 1

Interactions

Date

Type

Comments

Page 1 of 1

Creating Dashboards Using the Dynamic Views Designer

268

7

Developing a Timeline in a Portlet or Tab

This section describes how to develop a timeline in a Jacada WorkSpace solution. Timelines provide the ability to present a chronology of events, for example, company or customer related, to agents and enable them to view the relevant event detail.

Timelines are contained in either Jacada WorkSpace portlets or tabs. Jacada WorkSpace uses the Simile open source technology to support developing and displaying timelines in Jacada WorkSpace.

Note: For information about Simile timelines, go to <http://www.simile-widgets.org/timeline/>.

The following sections describe:

- [Timeline Components](#)
- [Defining a Timeline](#)
- [Methods Available for Timeline Development](#)
- [Timeline Processing](#)

Timeline Components

A timeline in Jacada WorkSpace displays chronological data horizontally. Each timeline incorporates the following components:

- **Event** – Descriptive and display information assigned to a specific event entry. One or more events are assigned to a timeline band. Jacada WorkSpace supports the JavaScript Object Notation (JSON) format to define event information.
- **Band** – A set of chronologically ordered events. Each timeline graphically displays one or more

bands.

Per displayed page in either a portlet or a tab, Jacada WorkSpace supports the presentation of a single timeline containing a maximum of two bands.

Event Attributes

The following are the attributes of an event:

Event Attribute	Description
caption	Optional. The tooltip for the event, that is the text displayed when the cursor hovers over the event in the timeline.
classname	Optional. The name of the class in the project's CSS style sheets that defines the style of the event's <code>title div</code> and <code>duration div</code>. The following files each contain a project CSS style sheet: • <code>WebContent/USER/resources/themes/Jacada/style.css</code> • <code>WebContent/USER/resources/themes/Sample/style.css</code>
color	Optional. The color used for displaying the event's tape (see <code>durationEvent</code> above) in the timeline. This attribute is also used for displaying the event's <code>title</code> in the timeline, if the <code>textColor</code> attribute is not defined.
description	Optional. Descriptive text about the event. Displays in the event's information bubble.

Event Attribute	Description
end	Optional. The end date and time of the event. See <code>dateFormat</code> below in band attributes.
icon	Optional. A URL to the icon image used to mark the event, when displayed in its band in the timeline. Displayed adjacent to the event's title.
image	Optional. A URL to the image selected for the event. Displays in the event's information bubble.
isDuration	Optional. Valid values: <code>true</code> or <code>false</code>. If true, an accompanying line marks the duration of the event in the timeline display. This accompanying line is referred to as a <i>tape</i>.
link	Optional. A URL that is linked to the event's title. When provided, the title displays as a hyperlink in the event's information bubble.

Event Attribute	Description
start	Required. The start date and time of the event. See <code>dateFormat</code> below in band attributes.
tapeImage	Optional. A URL to the image used for displaying the event's tape (see <code>durationEvent</code> above) in the timeline.
tapeRepeat	Optional. Defines the repetition of the <code>tapeImage</code>, as it applies to the tape's horizontal and vertical dimensions. Valid values: • <code>repeat</code> (apply repetition both horizontally and vertically), • <code>repeat-x</code> (only apply repetition horizontally), • <code>repeat-y</code> (only apply repetition vertically).
textColor	Optional. The color used for displaying the event's title in the timeline. If this attribute is not defined, then the <code>color</code> attribute is used in its place.
title	Optional. The text used for title of the event. Displays in the timeline and in the event's information bubble.

Band Attributes

The following are the attributes of a band:

Band Attribute	Description
<code>id</code>	Required. The id of the band.
<code>intervalUnit</code>	Required. The interval used in the band. Supported values: <code>yearly</code>, <code>monthly</code>, <code>weekly</code> and <code>daily</code>.
<code>intervalWidth</code>	Optional. The width of each interval in the band, which is specified in pixels. Default width is 200.
<code>height</code>	Optional. The height of the band, which is specified in a percentage, for example 25%. If not supplied, Jacada WorkSpace establishes the size a band's height; multiple bands are equally sized in width in the timeline.
<code>dateInFocus</code>	Required. The date that is displayed in central focus for the band, when the timeline is initially displayed.

Band Attribute	Description
startupJsonEventFunction	Optional. A custom developed function returning a JSON event structure. For example, <code>getMyEvents()</code>. This structure contains a defined set of events that is loaded into the band in the timeline.
startDate	Optional. The starting date and time of the band. Time zone is not supported per band. If not supplied, the user can endlessly drag or scroll left in the band.
endDate	Optional. The ending date and time of the band. Time zone is not supported per band. If not supplied, the user can endlessly drag or scroll right in the band.

Band Attribute	Description
filterHandler	Optional. The name of the developed method, for example, <code>filterHandler="myFilter"</code>, that is used to determine whether or not an event is displayed in the band or not (filtered). Filter methods return <code>true</code> or <code>false</code>. If <code>true</code>, the specified event can display in the band of a timeline. A <code>false</code> return value, removes the specified event from display in the band. By default, no event filter is defined for this attribute.
linkedBand	Optional. The id of another band with which this band is linked, that is, synchronized band-displays in the timeline.
zoomInOnMouseScroll	Optional. Valid values: <code>true</code> or <code>false</code>. If <code>true</code>, Jacada WorkSpace, by default, causes the band to zoom-in, when the user scrolls the mouse.

Band Attribute	Description
<code>dateFormat</code>	Optional. The format used to provide date and time information for events. Jacada WorkSpace supports the following formats: The default format of <i>May 10 1961 00:00:00 GMT-0600</i>. Any ISO 8601 format except for the following: <pre>2008-03-13T19:49:00-05:00 2008-03-13T19:49:00Z 2008-03-13T19:49Z</pre> To use any other ISO 8601 format, make the following attribute assignment: <pre>dateFormat = 'iso8601'.</pre> This attribute is not applicable to bands. For bands, the default format – <i>May 10 1961 00:00:00</i> – is always used to provide date and time information. Time zone information is not supported for bands.

Timeline Attributes

The following are the attributes of a timeline:

Timeline Attribute	Description
<code>id</code>	Required. The id of the timeline.

Timeline Attribute	Description
onClickHandler	Optional. The name of the developed function, for example, <code>onClickHandler="myClickEventHandler"</code>, that Jacada WorkSpace initially invokes when the viewer clicks on an event displayed in a band in the timeline. This click handling function optionally returns the variable <code>PROCEED_WITHOUT_DEFAULT_CLICK_HANDLING</code>.
styleClass	Optional. The name of the class in the project's CSS style sheets that defines the style of the timeline <code>div</code>. The following files each contain a project CSS style sheet: • <code>WebContent/USER/resources/themes/Jacada/style.css</code> • <code>WebContent/USER/resources/themes/Sample/style.css</code> If not supplied, Jacada WorkSpace uses the default class <code>timeline</code> that is defined in the file <code>WebContent/SYSTEM/resources/css/timeline.css</code>.

Defining a Timeline

Per displayed page in either a portlet or a tab, Jacada WorkSpace supports the presentation of a single timeline containing a maximum of two bands and provides the following tags to define a timeline:

- `timeline`
- `timelineBand`

Use these tags in the `.jsp` file containing the timeline code. The following code example defines a

timeline hosting two bands:

```
<jacada:timeline id="myTimeLine"
styleClass="timelineClass"

onClickHandler="userClickHandler">
  <jacada:timelineBand id="band1"
intervalUnit="yearly" intervalWidth="200"
                      height="80%" dateInFocus="Jun
16 2006 00:00:00 GMT"
                      startDate="Jun 16 2000
00:00:00 GMT"
                      endDate="Jun 16 2010
00:00:00 GMT"

filterHandler="userFilterByStartDate"/>
  <jacada:timelineBand id="band2"
intervalUnit="yearly" intervalWidth="200"
                      height="80%"
dateInFocus="Jun 16 2006 00:00:00 GMT"

startupJsonEventFunction="getEvents()"
                      startDate="Jun 16 2000
00:00:00 GMT"
                      endDate="Jun 16 2010
00:00:00 GMT"
                      linkedBand="band1"
zoomInOnMouseScroll="true"
                      dateFormat="iso8601"

filterHandler="userFilterByCaption"/>
</jacada:timeline>
```

In the **timeline** tag above, the attribute assignment `onClickHandler="userClickHandler"`, registers `userClickHandler` as the initial, click handling function that Jacada WorkSpace invokes. In the **timelineBand** tags above, the attribute assignments `filterHandler="userFilterByStartDate"` and `filterHandler="userFilterByCaption"`, register `userFilterByStartDate` and `userFilterByCaption` as the filter functions to be invoked, prior to drawing or re-drawing the timeline, for `band1` and `band2` respectively.

All functions developed for use by a timeline, that is those functions provided by the `onClickHandler`, the `startupJsonEventFunction` and the `filterHandler` attributes,

must be accessible. Accomplish this by incorporating these function into either one of the following files:

- The jsp page used to display the timeline.
- Any JavaScript file that is included in the jsp page used to display the timeline.

To synchronize the display of multiple bands in a timeline, you use the `linkedBand` band attribute within the `timelineBand` tag. When establishing this band linkage in your timeline code, always link the latter band (defined second in code) to the preceding band (defined first in code). In the preceding code example, note the use of the `linkedBand` attribute.

Methods Available for Timeline Development

Jacada WorkSpace exposes the many methods for developing a timeline. Each method is relevant for performing either server-side processing or client-side (Jacada WorkSpace) processing. In production, these methods modify a timeline's display. Each of the sections that follow describe these methods.

Jacada WorkSpace provides a dedicated include file that provides all the necessary imports. Reference it in your code, as follows:

```
<%@ include file = "/SYSTEM/resources/jspf/
includes/timeline.jspf" %>
```

The `timeline.jspf` does not include the handling of special characters. Therefore, if your project requires UTF-8 character support, edit the `.jsp` file containing the timeline code and update the file to include the following line:

```
<%@ page language="java" contentType="text/
html; charset=UTF-8" %>
```

Client Processing Methods

This section presents all the methods that support timeline processing in Jacada WorkSpace.

Refreshing a Timeline

To refresh, that is re-draw, a timeline, use this method. Provide the method with the id of the timeline. Invoke this method whenever any timeline attributes or any of its bands' attributes are modified.

The method is invoked as follows:


```
$W().timelineManager.refreshTimeline(<timeline_id>)
```

Use of this method is required when updating your filtering method. The most common use of this method is when filtering events. After modifying the filter method, invoke

`$W().timelineManager.refreshTimeLine()` to view the timeline with its events filtered.

Loading Events

Load events in the given object into a band with the given id. This method replaces all the existing events in a band with those provided by the events variable. Following the execution of this method, Jacada Workspace automatically refreshes the timeline, resulting in the display of the loaded events in the band/timeline.

The method is invoked as follows:

```
$W().timelineManager.loadEvents(<timeline_id>,  
<band_id>, <events>)
```

The events to be loaded must be in JSON format. The following is an example of JSON formatted events:

```
function getEvents() {
    var timeline_data1 = { // save as a global
variable

        'wikiURL': "http://simile.mit.edu/shelf/",
        'wikiSection': "Simile Cubism Timeline",

        'events' : [

            {'start': 'Jun 16 2010 09:00:00 GMT', 'end':
'Jul 15 2010 09:00:00
                GMT', 'title': 'event1',
'description': 'description1',

                'color':'red' },

            {'start': 'Aug 28 2010 09:00:00 GMT', 'end':
'Oct 15 2010 09:00:00
                GMT', 'title': 'event2',
'description': 'description2',

                'color':'green' }
        ]
    }
    return timeline_data1;
}
```

Removing All Events

Remove all the events from a band with the given id. Following the execution of this method, Jacada WorkSpace automatically refreshes the timeline, resulting in the event removal being evident in the band/timeline.

The method is as invoked as follows:

```
$W().timelineManager.removeAllEvents(<timeline_i
d>, <band_id>)
```

Jacada WorkSpace does not provide a method for the removal of either a single event or multiple events. Accomplish these actions by developing your own filtering methods, see [Event Filtering](#).

Adding an Event

Add an event in the given object into a band with the given id. Following the execution of this method, Jacada WorkSpace automatically refreshes the timeline, resulting in the display of the

added event in the band/timeline.

The method is as invoked as follows:

```
$W().timelineManager.addEvent(<timeline_id,  
<band_id>, <event_json>)
```

This method receives the event to be added in JSON format. The following is an example of an event to be added in JSON format:

```
var event_json ={'start': 'May 28 2006 09:00:00  
GMT',  
  'end': 'Jun 15 2006 09:00:00 GMT',  
  'title': 'Added Event',  
  'description': 'Added a Single Event in JSON  
Format',  
  'image': 'http://image2.allposters.com/images/  
BRGPOD/102770_b.jpg',  
  'link': 'http://www.allposters.com/-sp/  
Brotsky-Posters_ok58_.htm'  
}
```

Adding Many Events

Add multiple events in the given object into a band with the given id. Following the execution of this method, Jacada WorkSpace automatically refreshes the timeline, resulting in the display of all the added events in the band. The method is invoked as follows:

```
$W().timelineManager.addManyEvents(<timeline_id>  
, <band_id>, <events_list_json>)
```

This method receives a list events to be added in JSON format. The following is an example of a list events in JSON format:

```

var events_json = [ {'start': 'May 28 2006
09:00:00 GMT',
    'end': 'Jun 15 2006 09:00:00 GMT',
    'title': 'Added Event',
    'description': 'Added a Single Event in JSON
Format',
    'image': 'http://image2.allposters.com/images/
BRGPOD/102770_b.jpg',
    'link': 'http://www.allposters.com/-sp/
Brodsky-Posters_ok58_.htm'
    },
    {
        //additional event definitions
    },
    ..
    {
        //final event definition
    }
]

```

Server Processing Methods

This sections presents all the methods that support timeline processing in Jacada WorkSpace on the server-side.

For each [client-side method](#) there is an equivalent method to support the same processing from the server-side. Hence, for Jacada WorkSpace server processing, the following methods are available:

```

public void refreshTimeline(String
<timeline_id>) throws TimelineException;
public void loadEvents(String <timeline_id>,
String <band_id>,
                        List<TimelineEvent>
<event_list>) throws TimelineException;
public void removeAllEvents(String
<timeline_id>, String <band_id>)

                        throws TimelineException;
public void addEvent(String <timeline_id>,
String <band_id>,
                        TimelineEvent
event) throws TimelineException;
public void addManyEvents(String <timeline_id>,
String <band_id>,
                        List<TimelineEvent>
<event_list>) throws TimelineException;

```

Note: Due to the push mechanism supporting server-side methods, providing a very large number of events to either the `loadEvents` or the `addManyEvents` method to process, may result in high, server-side, processing overhead.

The class `TimelineEvent` is an object that represents a single event. Its attributes are described in the table of event attributes in [Timeline Components](#). For example:

```

TimelineEvent event1 = new TimelineEventImpl();
event1.setDescription("provide your event
description here");
event1.setTitle("provide your event title
here");
event1.setStart("May 27 2009 09:00:00 GMT");
event1.setEnd("June 8 2009 09:00:00 GMT");

```

Using Server-side Methods

The following code provides an example of how to use the available server-side methods:

1. Retrieve the `timelineService` singleton from the application; this singleton is responsible for the timeline Java methods:

```
ApplicationContext applicationContext =  
WebApplicationContextUtils.getWebApplicationContext(this.  
  
getRequest().getSession().getServletContext());  
TimelineService ts = (TimelineService)  
applicationContext.getBean("timelineService");  
  
// Initialize a timeline event list  
List<TimelineEvent> events = new ArrayList();
```

2. Create several timeline events:

```
for( int i=0; i<3; i++){  
    TimelineEvent event1 = new  
TimelineEventImpl();  
  
event1.setDescription("desc"+Integer.toString(i  
+1));  
    event1.setTitle("title"+Integer.toString(i  
+1));  
    event1.setStart("May "+Integer.toString(i+1)+"  
2006 09:00:00  
  
        GMT");  
    event1.setEnd("May 29 2006 09:00:00 GMT");  
    events.add(event1);  
}
```

3. Invoke various server-side methods to manipulate events in the timeline on the client-side:

```
Example: Add a single event to the timeline
ts.addEvent("t1", "b3", events.get(0));
Example: Add multiple events to the timeline
ts.addManyEvents("t1", "b3", events);
Example: Refresh the timeline
ts.refreshTimeline("t1");
Example: Remove all events from the timeline
ts.removeAllEvents("t1", "b3");
```

Timeline Processing

When an agent clicks on an event in the band of a timeline, Jacada WorkSpace invokes the registered, click handling function that is defined in the timeline attribute `onClickHandler`. This click handling function optionally returns the variable `PROCEED_WITHOUT_DEFAULT_CLICK_HANDLING`.

In the project, the developer must make the click handling function available by including its JavaScript code in this page. Jacada WorkSpace provides the registered, click handling function with the `x` and `y` coordinates of the clicked event and the event object. The following code example invokes the registered click handling function with the required parameters:

```
function myClickEventHandler(x, y, evt){
    alert ("Clicked_Event is
"+evt.getDescription());
}
```

Following the processing of the registered click handling function, Jacada WorkSpace automatically invokes the default, click handling function. The default, click handling function opens the event's associated information bubble.

Whenever the registered, click handling function returns the variable `PROCEED_WITHOUT_DEFAULT_CLICK_HANDLING`, Jacada WorkSpace does not invoke the default, click handling function. The following code example invokes the registered click handling function and, upon return, would result in Jacada WorkSpace not invoking the default, click handling function:

```
function myClickEventHandler(x, y, evt){
    alert ("Clicked_Event is
"+evt.getDescription());
    return PROCEED_WITHOUT_DEFAULT_CLICK_HANDLING;
}
```

Event Filtering

Filter methods are used to identify, based on specified comparison criteria, the events to remove from a specific band in a timeline. Filters are coded in JavaScript (client-side functions). At application start-up, before the timeline is ever drawn, Jacada WorkSpace invokes all registered filter methods. In your application, invoke any of the `refreshTimeline` methods to filter and re-draw timelines during production.

You are required to develop your own filter methods, to either remove a single event or remove multiple events from a band in a timeline.

The following code provides an example of a filter method:

```
function myFilter1(evt) {
    // evt is the event parameter passed to the
    method.
    var text = "myFilterCriterionText";
    if (text != null && text.length>0){
        var regex = new RegExp(text, "i");
        return regex.test(evt.getText());
    }
    return true;
}
```

The Event Information Bubble

An event information bubble displays on top of the timeline, right above its associated event. Many of the attributes defined for the event are included in the event's information bubble; these include, but are not limited to, title, description, start, end and image.

An agent clicking on the title in an event information bubble, when the title appears as a hyperlink, causes the *TimelineBoard* tab to open in Jacada WorkSpace and display the link's content. In order for the *TimelineBoard* tab to open in Jacada WorkSpace, the agent must be assigned the privilege `TabTimelineBoard`.

By default, the *TimelineBoard* tab does not automatically appear in Jacada WorkSpace; for the tab to appear and open, the agent must click on the hyperlinked title in an event information bubble.

Customizing Server-side Functionality

The following sections describe how to extend and customize the built-in server-side functionality provided by Jacada WorkSpace:

- [Overview](#)
- [Defining Variables for Use During the Project Life Cycle](#)
- [Generating Project Features](#)
- [Manually Creating a New Custom Manager](#)
- [Creating a System Manager Extension](#)
- [Writing Manager Code](#)

Overview

Jacada WorkSpace provides a modular infrastructure for Jacada WorkSpace functionality, to facilitate the extension and customization of the built-in functionality. This infrastructure is implemented within a class named `SharedData`, that contains a set of classes for managing different aspects of Jacada WorkSpace functionality. These classes are referred to as system managers. When you want to extend the built-in functionality, it is necessary to create additional managers, as needed. Jacada WorkSpace provides an empty class, `ExtensionManager`, as an example.

The following table lists the system managers provided with Jacada WorkSpace:

Manager	Description
<code>CTIManager</code>	Contains CTI logic and variables.

Manager	Description
ExtensionManager	An empty class for extensions.
LocaleManager	Contains localization logic.
IntegrationManager	Contains a session identifier for use when integrating Jacada WorkSpace with Jacada Fusion.
PresentationManager	Contains UI logic and variables.
ScriptManager	Contains call-flow script logic and variables.
SecurityManager	Contains LDAP and licensing logic and variables.

In addition, a Session manager is available, which you can use to monitor and track usage that is session-related.

Defining Variables for Use During the Project Life Cycle

During the life cycle of a Jacada WorkSpace project, there can be values used in the project code that are different during different phases in the project life cycle. For example, when connecting to a web service to retrieve information, the URL for the web service used during development and the URL used during testing or production can be different. You can define variables whose values are stored in the database whereby the value use is dependent on the project phase.

This section describes how to set up the project variables. The following project phases are defined in Jacada WorkSpace:

- Development
- Preproduction
- Production
- QA
- System Tests
- Training

- UAT (User acceptance testing)

Project variables can be defined for these phases, with a different value assigned to each phase. The current phase is set in the Jacada Management Console, in the `Environment` field under *Global Settings*.

The following sections describe:

- [Setting Up the Project Variables Infrastructure](#)
- [Defining and Using Project Variables](#)

Setting Up the Project Variables Infrastructure

Project variables are defined in the IDE and managed using the Jacada Management Console. The variables are stored in the Jacada WorkSpace database. When project variables are defined in the IDE, in order to be able to manage them in the Jacada Management Console, the Jacada WorkSpace database must be updated. If you have permission to update the database, you can set up the connection in the IDE to automatically reflect changes and additions to the project variables in the database. If you do not have permission to update the database, you can export any changes and additions to the project variables to an SQL file, formatted for the database, which can then be used by the database administrator to update the database.

For details about setting up the database connection, refer to the *Jacada WorkSpace Installation Guide for the Developer*.

You define project variables, which will either be updated directly to the database or an SQL script generated which can be passed to the database administrator.

Defining and Using Project Variables

You define project variables per project, in the IDE. Once defined, the variables can be used throughout the project. The defined project variables are written to the database, and can be managed using the Jacada Management Console.

To define project variables:

1. In the IDE, right-click the project and select *Jacada WorkSpace > Project Variables > Manage Variables*.
The *Project Variables* window is displayed.
2. Click *New*.
3. Specify a name, value and display name for the variable, and optionally a description.

Note: The display name and not the name is used in the Jacada Management Console to identify the variable. If a display name is not specified, the name is used.

4. Click *OK* to add the new variable to the list of variables. The variable is added to all the environments with the value specified.
5. Select the tabs to set the value for that environment. If the value is not set for an environment, it inherits the value set in the Common tab.
6. If the database can be updated, make sure the Update database on save checkbox is checked.

Note: If the database cannot be updated, a link is displayed to enable specifying the database details for the database connection displayed in the Data Source Explorer view. In the *Database Connection Profile* field select the database connection.

If you don't want to update the database, or do not have permission to update it, you can generate an SQL file to use at a later date to update the database.

7. Click *OK*.

To edit project variables:

1. In the IDE, right-click the project and select *Jacada WorkSpace > Project Variables > Manage Variables*.
The *Project Variables* window is displayed.
2. Select the tab for the environment for which you want to edit the variable.
3. Click *Edit* or double-click the variable to edit.
4. You can edit the value and the description fields only. Changes to these two fields are saved for the selected environment only.
5. Click *OK* to commit the change.
6. If the database can be updated, make sure the *Update database on save* checkbox is checked.

Note: If the database cannot be updated, a link is displayed to enable specifying the database details for the database connection displayed in the Data Source Explorer view. In the `Database Connection Profile` field select the database connection.

7. Click *OK*.

To delete project variables:

1. In the IDE, right-click the project and select *Jacada Work Space > Project Variables > Manage Variables*.

The *Project Variables* window is displayed.

2. Select the variable to remove.

3. Click *Remove*.

You are prompted to confirm the removal of the variable. The variable is deleted from all the environments.

4. Click *Yes* to delete the environment.

5. If the database can be updated, make sure the `Update database on save` checkbox is checked.

Note: If the database cannot be updated, a link is displayed to enable specifying the database details for the database connection displayed in the *Data Source Explorer* view. In the `Database Connection Profile` field select the database connection.

6. Click *OK*.

Updating the Database

If you have permission to update the database, and the connection is set up, as described in [Setting Up the Project Variables Infrastructure](#) you can update it directly from the IDE. If, however, you cannot update the database, for example, you do not have permission, you can generate an SQL file to enable the database administrator to update the database.

To update the database when a database connection is established:

- In the IDE, right-click the project and select *Jacada WorkSpace > Project Variables > Update Database with Variables*.

To generate an SQL file when there is not permission to access the database:

Note: You must have a database connection specified, as described in *Setting Up a Database Connection* in both the pre- and post-installation sections of the Jacada WorkSpace Installation Guide for the Developer to enable the correct SQL to be generated for the database type.

1. In the IDE, right-click the project and select *Jacada WorkSpace > Project Variables > Export Variables to SQL File*.

The *Save As* window is displayed.

2. Save the SQL file.

Using Project Variables

To retrieve a variable, use the following static method:

```
ProjectVariables.getValue("variable_name")
```

If the variable is a password, the value is retrieved encrypted.

Note: A variable is identified as being a password if the name includes the word password. in any form: at the beginning, middle or end of the name and with any letter case.

To retrieve the current working environment, use the following static method:

```
ProjectVariables.getCurrentEnvironment ()
```

Generating Project Features

The development of a Jacada WorkSpace project involves creating many server-side components, such as managers, controllers, and DAOs in addition to the client-side JSP pages, etc.

In order to simplify the process and ensure standards across the project when different components are created, Jacada WorkSpace includes a wizard which enables you to create the skeleton structure for any new feature which includes a server-side.

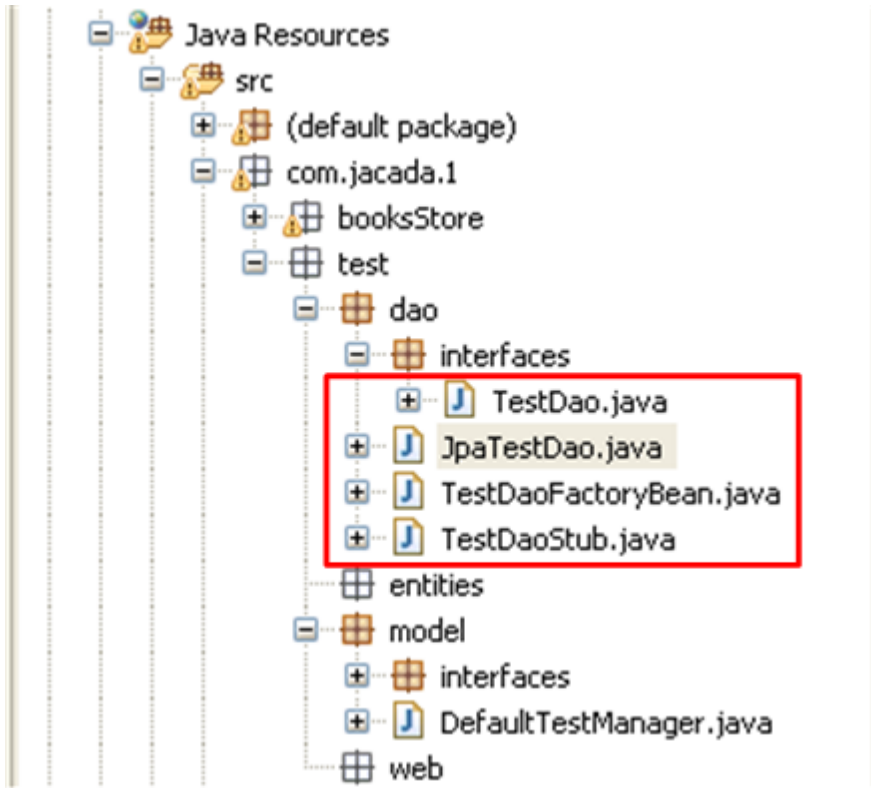
Note: For details of creating or extending managers directly, refer to [Manually Creating a New Custom Manager](#) and [Creating a System Manager Extension](#).

Classes and files generated by the feature skeleton wizard enforce the MVC pattern and use recommended technologies.

Using the wizard you can create the following:

- Valid Java package
- Controller class, with or without Spring MVC support
- Web content (JSP, JavaScript and CSS files), including ExtJS methods
- Manager beans
- DAOs

When using the wizard to create or update a DAO, Jacada WorkSpace offers an option of creating a stub class. A stub class is a parallel class (to the service class) that returns predefined dummy values in design-mode.



The stub option is useful when creating WorkSpace-related data sources for a Dynamic Views Dashboard ([Creating a WorkSpace-related Data Source](#)). When a stub DAO is used in the New Feature Skeleton, all return values for the relevant data source automatically use the stub values.

To generate a feature skeleton:

1. In the IDE, right-click the project and select *Jacada WorkSpace > New Feature Skeleton*. The *New Feature Skeleton* wizard is displayed.
2. Specify the basic content for the new feature, as follows:

Feature name – A unique name for the feature in the project. The name must be a valid Java type name.

Package – The location to save the generated package. The default package location is `<project>/src/com/jacada/<project>/<feature_name>`.

Generate Controller – Whether to create a controller for the feature or not.

Include Spring MVC – Whether to add Spring MVC support to the controller for the feature or not.

Generate JavaScript Client – Whether to create client-side content for the feature: a sample ExtJS panel.

3. Click *Next*.

4. Specify the scope for the manager:

Singleton – There is only one instance of the manager for the Jacada WorkSpace solution.

Session – The manager is used throughout the session.

Call – The manager is reset after every call.

5. Specify any beans to be included from the list of existing beans and/or create new beans. When creating a new bean, specify the following:

ID – A name for the bean. The name must be a valid Java type name.

Class – Whether to create a new empty class for the bean or use an existing class. If the class is an existing class, specify its location.

Scope – The scope for the bean in this manager:

Singleton – There is only one instance of the manager for the Jacada WorkSpace solution.

Session – The manager is used throughout the session.

Call – The manager is reset after every call.

6. Click *Next*.

7. Define any DAOs required for the feature. When adding a new DAO, specify the following:

Name – A name for the DAO. The name must be a valid Java type name.

Type – The type of DAO:

JPA for DB access – Java Persistence API used to manage relational data.

Web Service – If the DAO is a web service, specify the web service WSDL as a URL or a project file. A web service client to the specified URL is generated.

Empty class

Stub Option – To create an extra class that returns dummy values in development mode, verify that the Generate Stub checkbox is checked.

8. Click *Finish*.

Depending on what was defined in the wizard, the following are generated:

[Server-side Components](#)

[Client-side Components](#)

Server-side Components

The following components are created on the server side, dependent on the options specified in the wizard, under `<project>/src/com/jacada/<project>/<feature_name>`:

- A controller
- A manager
- A JPA DAO implementation
- A web service client
- Spring definitions in a new file, `WebContent\WEB-INF\USER\applicationContext-<feature_name>.xml` is created.
- The controller is defined in the same file as the manager unless Spring MVC is defined for the controller, in which case the file, `WebContent\WEB-INF\userJson-servlet.xml`, is updated.

Client-side Components

The following components are created on the client side, dependent on the options specified in the wizard, under `<project>\WebContent\com\jacada\<project>\<feature_name>`:

- A CSS file
- A JSP file
- A JS file, which includes the basic methods to handle Jacada WorkSpace UI events. If ExtJS is specified to be used, the JS file also includes the main method and a store example.
- A JSPF file under `USER\resources\jspf\includes\moduleIncludes.jspf`. This file contains the recommended set of technologies for project level UI development, and is included in every feature's JSP file. You can add project specific technologies, such as tag libraries, through this file.

Updating a Feature

After generating a feature skeleton, you can update the manager, as follows:

- Add a new or existing bean
- Add a new DAO

You can also update any web service client created for a DAO.

To add a new bean:

1. Right-click on a manager class and select *Jacada WorkSpace Feature Update > Manager > Add New Bean*.

The New Bean dialog opens.

2. Specify the following:

ID – A name for the bean. The name must be a valid Java type name.

Class – Whether to create a new empty class for the bean or use an existing class. If the class is an existing class, specify its location.

Scope – The scope for the bean in this manager:

Singleton – There is only one instance of the manager for the Jacada WorkSpace solution.

Session – The manager is used throughout the session.

Call – The manager is reset after every call.

3. Click *Finish*.

A property for this bean is added to the manager class and the spring files are updated.

To add an existing bean:

1. Right-click on a manager class and select *Jacada WorkSpace Feature Update > Manager > Add Existing Bean*.

The Bean Selection dialog opens.

2. Select a bean from the list of existing beans.
3. Where necessary, select the bean interface or class to inject into the manager.
4. Click *OK*.

A property for this bean is added to the manager class and the spring files are updated.

To add a new DAO:

1. Right-click on a manager class and select *Jacada WorkSpace Feature Update > Manager > Add New DAO*.

2. Specify the following:

Name – A name for the DAO. The name must be a valid Java type name.

Type – The type of DAO:

JPA for DB access – Java Persistence API used to manage relational data.

Web Service – If the DAO is a web service, specify the web service WSDL as a URL or a

project file. A web service client to the specified URL is generated.

Empty class

Stub Option – To create an extra class that returns dummy values in development mode, verify that the Generate Stub checkbox is checked.

3. Click *Finish*.

The generated DAO is injected into the manager class.

When a web service API changes, you can update the web service client.

To update a web service client:

- Right-click on a web service DAO implementation and select *Jacada WorkSpace Feature Update > DAO > Update Web Service Client*.

The WSDL location is taken from an annotation inside the DAO class and the client stub is regenerated.

Manually Creating a New Custom Manager

Extend the Jacada WorkSpace project by adding code to `src/USER/global/managers/ExtensionManager.Java`. Create new custom managers either to better organize your extensions or to override the managers provided with Jacada WorkSpace.

To create a custom manager:

1. Create a class that implements the `SYSTEM.global.managers.AbstractManager` interface. For example:

```
public class MyManager implements
AbstractManager {
    private transient SharedData globalApp;
    private int callCount;
    public SharedData getGlobalApp() {
        return globalApp;
    }
    public void setGlobalApp(SharedData globalApp)
    {
        this.globalApp = globalApp;
    }
    ...
}
```

2. In the `WebContent/WEB-INF/ applicationContext-user.xml` file, add the bean for your custom manager and add the custom manager mapping in the `userManagers` bean. For example, continuing with `MyManager`:

```
<bean id="userManagers"
      class="com.jacada.spring.types.MapWrapper"
      scope="session">
  <aop:scoped-proxy />
  <property name="content">
    <map>
      <entry
key="USER.global.managers.ExtensionManager"
value-ref="extensionManager"/>
      <entry
key="USER.global.managers.MyManager"
value-ref="MyManager"/>
    </map>
  </property>
</bean>
<bean id="MyManager"
      class="USER.global.managers.MyManager"
      scope="session">
</bean>
```

Injecting the Custom Manager

Inject the new custom manager, that is access the methods and properties associated with this new custom manager, into your project code. Accomplishing the code injection differs depending on whether your code is Spring-based or not (Apache Beehive-based code, for example).

To inject a custom manager in a Spring-based development environment:

1. Continuing with the `MyManager` custom manager, created above, define your class, `USER.global.managers.MyClass`, as follows:

```
package USER.global.managers;
public class MyClass {
    private MyManager myManager;
    public void setMyManager(MyManager myManager)
    {
        this.myManager = myManager;
    }
    public myMethod() {
        myManager.doWork();
    }
}
```

2. Define the required bean, as follows:

```
<bean id="myClass" class=" USER.global.managers.MyClass">
    <property name="myManager" ref="myManager" />
</bean>
```

To inject a custom manager in a non-Spring-based development environment:

1. Obtain the reference to the requested object.
For example, in an Apache Beehive controller, call the `getRequest()` method.
2. Continuing with the `MyManager` custom manager, created above, use the following code:

```
HttpSession session =
getRequest().getSession();
org.springframework.web.context.support.WebApp
licationContext context =

    WebApplicationContextUtils.

getWebApplicationContext(session.getServletCon
text());
USER.global.managers.MyManager myManager =

(USER.global.managers.MyManager)
context.getBean("myManager");
myManager.doWork();
```

Creating a System Manager Extension

In a Jacada WorkSpace project, extend and customize the processing capabilities of an existing system manager.

To create a system manager extension:

1. Create a class that extends an existing system manager. For example, extend the `CTI Manager`:

```
public class MyCTIManager extends CTIManager {
    @Override
    public void login(HttpServletRequest arg0)
    throws Exception {
        super.login(arg0);
        String agentId =
        getAgentInformation().getAgentId();
    }
    ...
}
```

2. In the file `</project_name>/WebContent/WEB-INF/applicationContext-user.xml`, add the following code:
 - a. Add the bean for your system manager extension.
 - b. Add the system manager extension mapping in the `extendedSystemManagers` bean.For example, with `MyCTIManager`:

```

<bean id="extendedSystemManagers"
class="com.jacada.spring.types.MapWrapper"
    scope="session">
    <aop:scoped-proxy />
    <property name="content">
        <map>
            <entry
key="SYSTEM.global.managers.CTManager"
                value-ref="MyCTManager"/>
            </map>
        </property>
    </bean>
<bean id="MyCTManager"
class="USER.global.managers.MyCTManager"
    scope="session">
</bean>

```

Writing Manager Code

In an extension manager class, in addition to writing new methods, you can also extend the built-in global functionality by adding variables or by adding event-handling methods. Within an extension manager, there is a reference to the current `SharedData` instance, which enables accessing all the managers.

Jacada WorkSpace enables you to customize the handling of system events, such as login, by adding code to be executed in relation to the event, and specifying when this code should be executed, either before or after the relevant system methods event-handling.

Event-handling code is written by adding an annotation to the method that should register to execute when a specific system event is fired. The annotation has the following attributes:

actionName – The name of the event.

order – When the event-handling code should be executed, in relation to the firing of the event. Values are either `Before` or `After`. This attribute is optional, and its default value is `After`.

Available Actions

The following table lists the actions that are available:

Action	Parameters	Description
END_CALL	—	Ending a call.
FAILOVER_RECOVERY	HttpServletRequest request, HttpServletResponse response,	For future use.
LOAD_INBOUND_SCRIPT	HttpServletRequest request, String dnis	Loading a call-flow script for an inbound call.
LOAD_OUTBOUND_SCRIPT	HttpServletRequest request, String number	Loading a call-flow script for an outbound call.
LOGIN	HttpServletRequest request	Logging in to Jacada WorkSpace.
LOGOUT	HttpServletRequest request, HttpServletResponse response, String reason, boolean isNewDepartment	Logging out of Jacada WorkSpace.
ON_CREATE	—	Creating a Jacada WorkSpace session.
ON_DESTROY	HttpSession	Destroying a Jacada WorkSpace session.
START_BUSINESS_CALL	HttpServletRequest request, String dnis	Starting a business call.
START_INTERNAL_CALL	—	Starting an internal call.
UNLOAD_SCRIPT	—	Unloading a call-flow script.

The following example shows how to add an annotation to a method so that it is performed after the `START_BUSINESS_CALL` action. Note that the signature of the method, `afterStartBSNS` in this

example, must use the parameters of the action. In this example, after starting a business call, the `dnis` of the call is written to the log:

```
@GlobalAction(actionName=GlobalActionNames.START
_BUSINESS_CALL,

order=GlobalActionOrder.AFTER)

public void afterStartBSNS(HttpServletRequest request, String dnis){
    LogWrapper.debug("##### AFTER
START_BUSINESS_CALL dnis is: " + dnis);
}
```

Personalizing What the Agent Can See and Do

Jacada WorkSpace provides several options for personalizing the agent's user interface. These include applying a different layout or theme, or manipulating portlets, areas, and tabs.

The user interface personalization options in Jacada WorkSpace can be controlled at the management level, as described in the following table.

Who	How	What
System Administrator	Privilege based access	Controls access to Jacada WorkSpace features per the privileges that are assigned to a role, the roles that are assigned to groups and the users that are assigned to groups. For more information, see Authentication and Authorization.

Controlling the Agent's Personalization Options

Jacada WorkSpace provides general and specific attributes for controlling user interface personalization options. The general attributes define personalization options in Jacada WorkSpace regarding portlets, areas and tabs. The specific attributes are used to define personalization options for a specific portlet or tab.

All layout-related attributes are defined using the *Layout Management* node of the Jacada Management Console. The following sections describe the general and specific attributes that are available for use.

General Attributes for Personalization

The layout of Jacada WorkSpace determines which available portlets appear, where on the page they are displayed, whether their location and/or size is adjustable, etc. Layouts are defined using the *Layouts* page of the Jacada Management Console. When several layouts are available, one is designated as the default layout.

For detailed information about layout attributes, refer to the Jacada WorkSpace Administrator Guide.

Portlet-specific Attributes

Portlet-specific attributes are defined using the *Portlets* page of the Jacada Management Console. This page enables you to add portlets and modify portlet properties, e.g., the minimum height and width of the portlet.

Note: Portlet-specific attributes defined on the *Portlets* page are overridden, if necessary, by the general layout definitions specified on the *Layouts* page.

For detailed information about portlet attributes, refer to the Jacada WorkSpace Administrator Guide.

Pushing Information to the Agent

The following sections describe the push mechanism provided by Jacada WorkSpace that enables pushing information to the agent, without a specific request from the agent:

- [Overview](#)
- [Publishing Messages](#)
- [Subscribing to Receive Messages](#)
- [Client Side Configuration](#)
- [Push Example](#)

Overview

Portlets within Jacada WorkSpace can display information that is pushed to the portlet, without a specific request from the agent. For example, the varying prices of stock market shares can be pushed to a portlet as they change. This capability is implemented using a push mechanism, that enables sending information updates from the server to the client asynchronously, without a need for the agent to request the information.

Push mechanism components

The push mechanism relies on a number of components:

Component	Location	Description
Publisher	Depends on implementation.	Sends messages to a messaging server. The entity referred to as the <i>Publisher</i> is actually the entity requesting the action from the application server. (The application server is not considered the publisher even though its server-side code performs the publish processing.)
Messaging Server	Server	Distributes messages, received from the publisher, to the relevant Jacada WorkSpace sessions on the server. Jacada WorkSpace uses a messaging server based on Java Message Service (JMS).
Push Handler	Client (browser)	Distributes messages received from server sessions to the relevant portlets or pages in the client browser. The messages are transferred from the server to the push handler using Reverse Asynchronous JavaScript and XML (AJAX).

Jacada WorkSpace provides methods for implementing the push mechanism. This includes a method for sending messages from the publisher to the messaging server. On the client side, a method is provided to enable a portlet or a page to register to receive notification about specific events in the form of messages. The connection between the messages sent by the publisher, and the client that subscribes to a specific message, is made using a logical keyword, that distinguishes between different types of messages.

To implement the push mechanism:

- Create a publisher that publishes messages
- Subscribe to messages on the client

Publishing Messages

Publishing messages is done by writing a publisher which uses the methods provided by Jacada WorkSpace to send the messages. The publisher determines to which Jacada WorkSpace sessions to publish the message: a specific session, all sessions of a specific project or all sessions in the application server.

Jacada WorkSpace enables publishing messages from different sources:

- From a server component in the application server. Messages can be published from the same Jacada WorkSpace project or in a separate project.
- From a server component external to the application server:
 - Using a Java Message Service (JMS) or a Web service

Separate methods are provided for publishing messages from within the application server and for publishing messages from an external component.

In addition, for performance reasons, separate classes are used to publish multiple events and a single event. To publish events it is necessary to create and start a publisher before publishing the events, and then stop the publisher when there are no more events to publish. These tasks may impact on performance. Therefore, when publishing a single event, these tasks are performed automatically by the event publishing method, whereas when publishing multiple messages, methods are provided to enable the developer to create and start the publisher once at the beginning, publish all the necessary messages, and stop the publisher at the end.

Jacada WorkSpace provides the `com.jacada.jad.push.PushHelper` class for publishing messages. The following sections describe the methods provided for publishing messages:

- [Publishing When The HttpSession Is Available](#)
- [Publishing When an HttpSession Is Not Available](#)

Publishing When the HttpSession is Available

Jacada WorkSpace supplies methods that enable server-side publishing of messages to recipients, when the publisher's context, the HTTP session object, is available. These methods are provided by the `com.jacada.jad.push.PushHelper` class.

The available methods are as follows:

Method	Parameters	Description
<code>PushHelper.pushMessageToGroup(session,String groupName,String keyword, Serializable data);</code>	<code>session</code> – The HTTP session of the publish requestor. <code>groupName</code> – A valid Jacada WorkSpace group. <code>keyword</code> – Identifies the event for which to send a message. <code>data</code> – An object that represents the data.	Publishes a message to the group of Jacada WorkSpace users identified by the provided <code>groupName</code> parameter. The publisher of this message is a Web application session.
<code>PushHelper.pushMessageToAgent(session,String agentName,String keyword, Serializable data);</code>	<code>session</code> – The HTTP session of the publish requestor. <code>agentName</code> – A valid Jacada WorkSpace user. <code>keyword</code> – Identifies the event for which to send a message. <code>data</code> – An object that represents the data.	Publishes a message to the valid Jacada WorkSpace user identified by the provided <code>agentName</code> parameter. The publisher of this message is a Web application session.
<code>PushHelper.pushMessageToAll(session,String keyword, Serializable data);</code>	<code>session</code> – The HTTP session of the publish requestor. <code>keyword</code> – Identifies the event for which to send a message. <code>data</code> – An object that represents the data.	Publishes a message to all Jacada WorkSpace users. The publisher of this message is a Web application session.

Publishing When an HttpSession is not Available

Jacada WorkSpace supplies methods that enable server-side publishing of messages to recipients, when the publish requestor is context-less, that is the requestor does not have an associated HTTP session object. These methods are provided by the `com.jacada.jad.push.PushHelper` class.

The available methods are as follows:

Method	Parameters	Description
<code>PushHelper.publishMessageToGroup(String groupName, String keyword, Serializable data);</code>	<code>groupName</code> – A valid Jacada WorkSpace group. <code>keyword</code> – Identifies the event for which to send a message. <code>data</code> – An object that represents the data.	Publishes a message to the group of Jacada WorkSpace users identified by the provided <code>groupName</code> parameter. The publisher of this message is a server-side Java class.
<code>PushHelper.publishMessageToAgent(String agentName, String keyword, Serializable data);</code>	<code>agentName</code> – A valid Jacada WorkSpace user. <code>keyword</code> – Identifies the event for which to send a message. <code>data</code> – An object that represents the data.	Publishes a message to the valid Jacada WorkSpace user identified by the provided <code>agentName</code> parameter. The publisher of this message is a server-side Java class.
<code>PushHelper.publishMessageToAll(String keyword, Serializable data);</code>	<code>keyword</code> – Identifies the event for which to send a message. <code>data</code> – An object that represents the data.	Publishes a message to all Jacada WorkSpace users. The publisher of this message is a server-side Java class.

Subscribing to Receive Messages

The client side of the push mechanism consists of a push handler that is instantiated in the main `index.jsp`, located under the project folder.

The push handler can be accessed from any JavaScript code, using:

```
$W().Push
```

where:

`$W()` – Returns a reference to the topmost window.

To display “pushed” information in a portlet or a page, it is necessary to subscribe to the corresponding event. Subscribing is done by registering, in the push handler, an event handler to a specific event. The message is identified by a specific keyword. When the push handler receives a new message, the message is sent to all the event handlers that are registered for the keyword.

Subscribing is done using the following method:

`registerEventHandler(keyword, handler)` – This method registers the specified event handler to a specific event, as identified by the **`keyword`**.

For example:

```
$W().Push.registerEventHandler("MyEventKeyWord",  
MyHandler)";
```

More than one portlet can subscribe to the same event. For example, if two portlets need to display the same information, but in a different format.

In your subscription code, precede the use of the `$W().Push.registerEventHandler()` method with code that imports the required JavaScript files `jad.js`, `push.js` and `prototype.js`. Doing so enables the code's use of both the push register event and the push unregister event methods.

The following are the required import statements:

```

..
<script language="JavaScript" type="text/
javascript"
    src="<%=request.getContextPath()%>/SYSTEM/
resources/js/jad.js"> </script>

<script language="JavaScript" type="text/
javascript"
    src="<%=request.getContextPath()%>/SYSTEM/
resources/js/push.js"> </script>

<script language="JavaScript" type="text/
javascript"
    src="<%=request.getContextPath()%>/SYSTEM/
resources/js/prototype.js">
</script>
..

```

When it is no longer necessary to receive messages, for example when closing a portlet or when moving away from a page, you need to end the subscription by using the following method:

unRegisterEventHandler(keyword, handler) – This method unregisters the specified event handler from a specific event as identified by the **keyword**. Typically, this method is used in the **onunload** browser event in the **HTML** page that contains the handler.

Client Side Configuration

You can change the Push Timeout by invoking `$W.Push.setTimeout(<timeout in milliseconds>);`

Push Example

In the following example, the push mechanism is used to update a stock quote in a portlet. The quote information is obtained from a Web service.

This example refers to a Web service named `getStockQuote.wsdl`, that provides an operation named `getQuote`.

Creating a Publisher

The following tasks are necessary to create the publisher:

- Create an AXIS client that receives the stock quote information from the Web service.
- In this example, the name of the control, created in a folder called `external`, is `StockQuoteService`
- Write the publisher code, that uses a thread to publish events asynchronously.

In this example, you create a stock quote publisher within the application server, within the same Jacada WorkSpace application. Therefore, the example uses the method for multiple messages from a Web application.

To write the publisher code:

1. Create a class that contains the publisher logic in the `src/USER/global/Managers` folder.
In this example, the stock price of a specified stock is sent, every 30 seconds, to all users assigned to the `CTIUsers` group that are logged in to a Jacada WorkSpace session running on the server:

```

package USER.global;

import com.jacada.jad.push.PushHelper;
import javax.servlet.http.HttpSession;

public class StockQuotePublisher implements
Runnable {
    external.stockQuoteService stockQuote;
    HttpSession session;
    boolean exit = false;
    public
StockQuotePublisher(USER.external.stockQuoteServ
ice stockQuote,

        HttpSession session) {
        this.stockQuote = stockQuote;
        this.session = session;
    }
    public void run() {
        while (!exit) {
            float price = stockQuote.getQuote("jcda");
            String priceQuote = price+"";
            try {

PushHelper.publishMessageToGroup(session,
                                "CTIUsers","quote",
priceQuote);
                Thread.sleep(30000);
            } catch (Exception ex) {
            }
        }
    }
    public void stop() {
        exit = true;
    }
}

```

2. To start the publisher when the agent logs into Jacada Workspace, add to the `login()` method in the `src/USER/global/Managers/ExtensionManager.java` file the following code that creates a thread to publish the information asynchronously:

```

@GlobalAction(actionName=GlobalActionNames.LOGIN
,order=GlobalActionOrder.AFTER)
public void login(HttpServletRequest request){
    HttpSession session = request.getSession();
    StockQuotePublisher publisher =
        new
    StockQuotePublisher(stock,
request.getSession());

    request.getSession().setAttribute("quotesPublish
er", publisher);
    Thread t = new Thread(publisher);
    t.start();
}

```

Client Side: Subscribing to an Event

On the client side, to subscribe to an event, you need to write an event handler and register the event handler to receive messages for the keyword that corresponds to the event.

You need to add the subscription code and the event handler code either to an existing portlet, or a new portlet.

In the `index.jsp` file of the portlet, add the subscription code, which registers and unregisters the method, and the event handler code, as shown in the following example, where:

- Prior to the use of the registration method, `$W().Push.registerEventHandler()`, the code includes the required import statements for the `prototype.js`, `push.js` and `jad.js` files, which enable use of both the push register event method and the push unregister event method. See item **1** in the subscription code that follows.
- The event handler, `onQuoteChange`, is registered to the Push handler to receive messages for the keyword `quote`. See item **2** in the subscription code that follows.
- The definition of the event handler, `onQuoteChange`. In this example, this method updates the price of the stock quote in the HTML. See item **3** in the subscription code that follows.
- The function that unregisters the event handler. See item **4** in the subscription code that follows.
- Instructing the HTML to perform the unregister when the page is unloaded. See item **5** in the subscription code that follows.

The subscription code to the `quote` event:

```

<%@...
...>
// Item 1
<script language="JavaScript" type="text/
javascript"
src="<%=request.getContextPath()%>/SYSTEM/
resources/js/prototype.js">
</script>
<script language="JavaScript" type="text/
javascript"
src="<%=request.getContextPath()%>/SYSTEM/
resources/js/push.js"></script>
<script language="JavaScript" type="text/
javascript"
src="<%=request.getContextPath()%>/SYSTEM/
resources/js/common.js"></script>
<script language="JavaScript" type="text/
javascript"
src="<%=request.getContextPath()%>/SYSTEM/
resources/js/jad.js">
</script>

<html>
  <head>
    <title>Stock Quotes Updates</title>
  </head>
  <body
onkeydown="onKeyDownHandler();onunload=unregist
er();">
    Current JCDA stock price
    <br>
    Price: $<label value="0.00" id="price"/>
  </body>
  <script language="JavaScript">

$W().Push.registerEventHandler("quote",onQuoteCh
ange);          (Item 2)

```

```

function onQuoteChange(price) {
    (Item 3)
    $("price").innerText = price;
}
function unregister() {
    (Item 4)

```

```

Pushi $W().Push.unregisterEventHandler("quote",onQuote
Change)          (Item 5)

```

```

</script>
</html>

```

Integration with a BPM System

Jacada WorkSpace supports integration with an external Business Process Management (BPM) system to add tasks to users defined in the target BPM system.

Jacada WorkSpace is certified to integrate with the *Intalio BPP* system.

The following sections present:

- [Overview](#)
- [Integrating with BPM](#)
- [Customizing the Add Task](#)

Overview

Jacada WorkSpace can receive requests to add tasks to BPM users, from any of the following sources:

The Agent Tasks manager – The Agent Tasks manager requests the project to reassign an existing agent task from the *Agent Tasks* portlet to a BPM user.

Agent input – An agent, working anywhere within Jacada WorkSpace, provides information that the project then uses to add a task to a BPM user.

A Jacada Interaction Manager script – The script issues an add BPM task request to the project by means of its custom item operation *Add task to BPM system*.

Default Task Parameters

The default parameters of a task added to a user in a BPM system are as follows:

Task Parameter	Description
title	BPM description of the task.
creationDate	Creation date and time of the task in the format: DD-MMM-YYYY hh:mm:ss
creator	The name of the source – system, application or user – that added the task to BPM.
assigneeRoleOrGroup	A valid BPM role or group to which the task is assigned.
assigneeUser	A valid BPM user to whom the task is assigned.
status	Status of the task. Note: The <i>Intalio BPP</i> system forces an added task's status to the value <code>READY</code> , regardless of the value set by Jacada WorkSpace.
expirationDate	Expiration date and time of the task in the format: DD-MMM-YYYY hh:mm:ss
duration	Duration, in days, that the task is in effect. Provide in the format <code>PnDT</code> , where <code>n</code> is the number of days. For example, a duration of 75 days is provided as <code>P75DT</code> . When both an expiration date and a duration are provided for the task, only the expiration date is considered.
priority	A valid BPM priority of the task. Provided as a number.

Jacada WorkSpace supports customizing this set of parameters provided with an added task. See [Customizing the Add Task](#).

Jacada WorkSpace audits the successful addition of a task to a BPM user. With every such

audited event, Jacada WorkSpace records the following default parameters: `title`, `assigneeRoleOrGroup` and `assigneeUser`. Customize the audited parameters according to your project requirements.

Integrating with BPM

To integrate Jacada WorkSpace with a BPM system for the purpose of adding tasks, there are several accommodations that must be incorporated into your Jacada WorkSpace project.

Note: The required code accommodations presented in this section are specific to integrating with an *Intalio BPP* system. Use them as a reference model to be adapted, when integrating your project with another BPM system.

The vehicle by which Jacada WorkSpace interacts with the *Intalio BPP* system is a Web service, which is defined in the BPM system. In order for Jacada WorkSpace to access this Web service, you must define in your project a Web service client that generates client-side code based on the BPM system WSDL.

To define a Web service client based on the BPM system WSDL:

1. Copy the relevant WSDL and xsd files, supplied by the BPM system, to any directory that is located under the `<project_name>` directory.
For example, the *Intalio BPP* supplied files `PA.xform.all.wsdl` and `PA.xform.xsd` are copied to `C:\...\<workspace_name>\<project_name>\WebContent`.
2. In the IDE, right-click on the `<project_name>` and select the **New > Other** option.
3. In the *Select a Wizard* dialog, select the **Web Services > Web Service Client** option and click **Next**.
4. In the *Web Service Client* dialog, do the following:
 - a. In the *Service definition* field, provide the path to the WSDL file that you stored under the `<project_name>` directory in step 1.
 - b. In the *Client type* field, select *Java Proxy* from the dropdown menu.
 - c. In the *Configuration* list, make sure that the Web service runtime is *Apache Axis*.
 - d. Using the slider control, set the level of client generation to be *Assemble client*.

5. Click *Finish*.

Generating the client-side code may take several minutes.

In the `src/USER/services/bpm/BPMConnectorImpl.java` file do the following:

- Remove the comment surrounding the code section that initializes the connection between Jacada WorkSpace and the *Intalio BPP* system.
- Remove the comment surrounding the code in the `addTask()` method that pertains to the application server hosting Jacada WorkSpace.
- In the code, replace both the `<webServiceLocator>` and the `<webServiceSoap>` variables with their respective, generated values, which were generated by the *Web Service Client* wizard during execution of the procedure [To define a Web service client based on the BPM system WSDL](#).
- In the `addTask()` method code that pertains to the application server hosting Jacada WorkSpace, provide the `<BPM_server_host_name>` in both the `setUserProcessEndpoint` and the `setPAPortEndpointAddress` methods.

Customizing the Add Task

The manager `DefaultBPMServiceManager` handles the processing for adding a task to a user in a BPM system.

Use the `DefaultBPMServiceManager` to activate the add task methods within your Jacada WorkSpace project. The `BPMConnectorImpl` class implements the `BPMConnector` interface. Override the methods provided by the `BPMConnectorImpl` class, which are contained in the `src/USER/services/bpm/BPMConnectorImpl.java` file.

BPMTask Methods

The `BPMTask` interface exposes the following methods:

Method	Description
<pre>public void convertBPMTask(HashMap<String,String> task);</pre>	Updates the <code>BPMTask</code> object, based on the parameters provided by <code>task</code>. <code>task</code> is a hash map of key-value strings. The Jacada Interaction Manager script provides this hash map when it issues its add BPM task request to the project.
<pre>public void convertBPMTask(AgentTask task);</pre>	Updates the <code>BPMTask</code> object, based on the parameters provided by <code>task</code>. <code>task</code> is an <code>AgentTask</code> object. The Agent Task manager provides it.
<pre>public Map<String, String> getBPMAuditParams();</pre>	Returns the key-value strings to be recorded as the parameters for the audited event.

To customize either the parameters provided with an added task or the audited, add task event parameters, override the relevant `BPMTaskImpl` method.

Note: The *Intalio BPP* system forces an added task's status to the value `READY`, regardless of the value set by Jacada Workspace.

Bean Definition

In order to activate the add task methods, add a reference in your bean to the bean `com.jacada.jad.services.DefaultBPMServiceManager`, located in the `WebContent/WEB-INF/applicationContext-services.xml` file.

The following beans can be changed:

bpmConnector – bpm bean containing the bpm logic, where the actual web service is called. Modify this bean to contain your `BPMConnector` implementation. This bean is in the singleton scope, therefore, when adding processing logic, make sure the bean is aware of the singleton scope. This bean contains the default implementation enclosed in comments.

BPMTaskBean – bpm bean containing the bpm task data. Modify this bean to contain your `BPMTask` implementation. This bean is in the prototype scope to allow for dynamic creation. Do not change the id of this bean.

Example: Add Task to BPM System

The following code examples illustrate how to add a task to a BPM user.

1. Define a class in your Spring configuration file and set the `BPMManager` property, as follows:

```
<bean id="myBeanId" class="myPackage.MyClass"
init-method="init">
  <property name="bpmManager" ref="BPMService"/>
</bean>
```

Note: If you do not have a Spring configuration file defined, create one named `applicationContext-operation.xml` and locate it under `WebContent/WEB-INF`.

Add the following `<beans>` element definition to this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/
schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://
www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-
beans-2.0.xsd">
</beans>
```

Add the `<bean>` element within the `<beans>` element.

In the file `WebContent/WEB-INF/web.xml`, find the `contextConfigLocation` context-param. This is where all Spring xml files are listed. Add the following entry to the list: `/WEB-INF/applicationContext-operation.xml`.

2. Create a new java class for the bean defined in the Spring configuration file, `myPackage.MyClass`, create a setter method to the `BpmManager` interface and implement `ApplicationContextAware`, in order that the class can obtain `BPMTask` objects:

```

public class MyClass implements
ApplicationContextAware {

    private BPMManager bpmManager;
    private ApplicationContext
applicationContext;

    ..
    public void setBpmManager(BPMManager
bpmManager) {
        this.bpmManager = bpmManager;
    }

    public void
setApplicationContext(ApplicationContext
applicationContext)
        throws BeansException {
        this.applicationContext =
applicationContext;
    }
}

```

3. The following code, in the new class, obtains a BPMTask, sets its parameters as needed and calls the appropriate addTask method:

```

..
public void addBPMTaskExample() {
    BPMTaskImpl bTask = (BPMTaskImpl)
applicationContext.getBean("BPMTaskBean");

    bTask.setTitle("sample addBPMTask test");
    bTask.setAssigneeUser("EdwardWilliams");
    bTask.setAssigneeRole("");
    bTask.setCreationDate(new Date());
    bTask.setCreator("JohnSmith");
    bTask.setDuration(P30DT);
    bTask.setExpirationDate(null);
    bTask.setPriority(13);
    bTask.setStatus("READY");

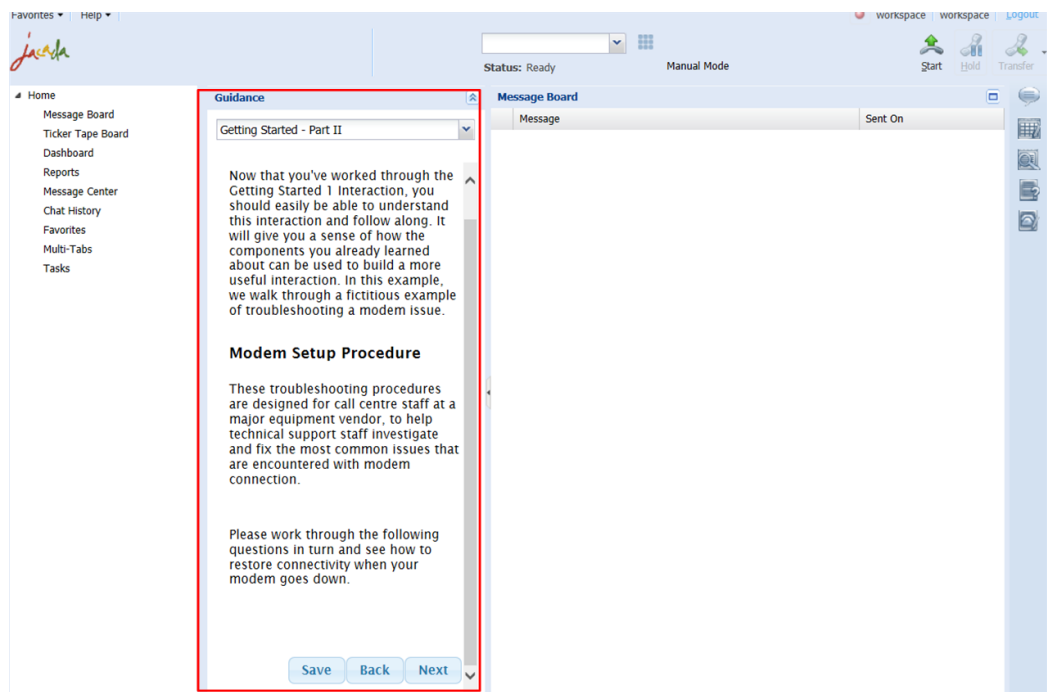
    bpmManager.addTask(bTask);
}
..

```


12

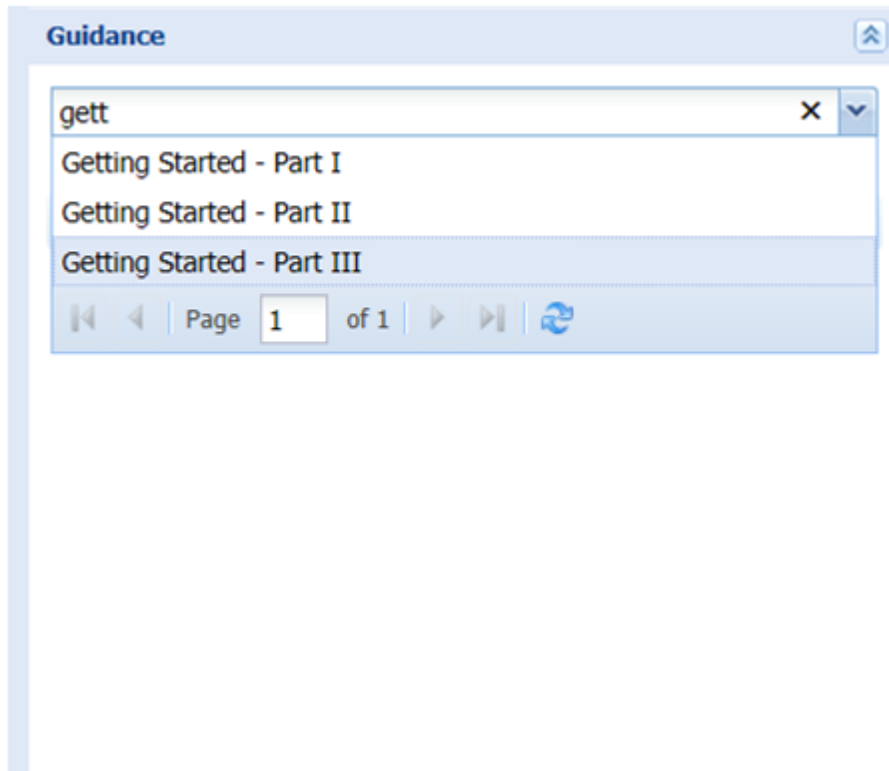
Integration with Jacada Interaction Server

The *Guidance* portlet enables agents to run Interactions located in the Jacada Interaction Server. Once an Interaction is retrieved from the server, its content is displayed in the *Guidance* portlet, and the agent is able to navigate through the Interaction using the *Next* and *Back* buttons.



To access an Interaction, agents can enter letters contained in the Interaction's name in the Search textbox. Interactions whose names start with the letters entered are displayed automatically, and the agent can then select the relevant Interaction. Alternatively, the agent can open the entire list of

available Interactions, and then navigate to the required one using the icons of the Navigation toolbar. Clicking  refreshes the Interactions list with the latest data from the Jacada Interaction Server.



Note: For successful integration with Jacada Interaction Server, some required settings need to be configured in the Jacada Management Console. For details, refer to the Jacada WorkSpace Administrator Guide (Interact Settings topic).

For details about configuring Jacada Interaction Server to support Interactions that integrate with WorkSpace, refer to [Designing Interactions for WorkSpace Integration](#).

Designing Interactions for WorkSpace Integration

The following procedure, intended for users familiar with Jacada Interaction Server and the Jacada Interaction Designer, explains how to define the Interaction elements required in Interactions that influence the WorkSpace environment. The example presented in the procedure involves an Interaction that shows a tab in WorkSpace.

The method that is exposed in the example (`showTab`) is provided in WorkSpace in the **USER.services.interact.InteractIntegrationController** java file. To expose additional methods from WorkSpace to Jacada Interaction Server, add the methods to this file, using the `showTab` method as a template.

To define elements for WorkSpace integration:

1. When designing the Interaction in the Interaction Designer, manually create the following variables:

- `[integrationService]`
- `[wsSessionCookieName]`
- `[wsSessionId]`

When an Interaction using these variables is called from WorkSpace, the variables are automatically populated by WorkSpace.

2. From the Jacada Interaction Server Admin Console, define an integration point whose URL includes the WorkSpace variables and the method that is to be exposed by WorkSpace. The following example shows the URL structure:

`http://[integrationService]/showTab;[wsSessionCookieName]=[wsSessionId]?tabId=[tabIdValue]`

When this integration point is called, Jacada Interaction Server will invoke the method `showTab` with `tabId` as an input variable.

Integration with JIA Services

Jacada WorkSpace supports an infrastructure that allows developers to invoke JIA services from WorkSpace JavaScript code. A JIA service can be executed using either the GET or POST protocol.

GET Request Structure

The following structure is used to invoke a GET request:

```
JIAClient.get(urlParams, requestParams)
```

where:

urlParams – The parameters for the service.

requestParams – The configuration object for the service.

Object properties are listed in the following table.

Property	Description
url	The URL of the JIA service.
async	Asynchronous indicator. The default value is <code>true</code> .
timeout	The timeout for the request. The default value is <code>0</code> (no timeout).
onsuccess	Success handler function.
onerror	Error handler function.
ontimeout	Timeout handler function.

For example:

```

var requestParams = new Object();
requestParams.url = 'http://localhost:8080/myService';
requestParams.onsuccess = function(response){
    alert('success');
};

requestParams.onerror = function(response){
    if (response != undefined) {
        if (response.target != undefined) {
            alert('error ' + response.target.responseText);
        } else {
            alert('error');
        }
    }
};

requestParams.ontimeout = function(response){
    alert('error');('timeout ' + response.target.responseText);
};

JIAClient.get('/customerSearch/123', requestParams);

```

POST Request Structure

The following structure is used to invoke a POST request:

```
JIAClient.post(postBody, requestParams)
```

where:

postBody – A JavaScript object that will be serialized to JSON format and sent in the post body.

requestParams – The configuration object for the service.

For a list of object properties, refer to the description of the GET request structure.

For example:

```

var requestParams = new Object();
requestParams.url = 'http://localhost:8080/myService';
requestParams.onsuccess = function(response){
    alert('success');
};

requestParams.onerror = function(response){
    if (response != undefined) {
        if (response.target != undefined) {
            alert('error ' + response.target.responseText);
        } else {
            alert('error');
        }
    }
};

requestParams.ontimeout = function(response){
    alert('error'); ('timeout ' + response.target.responseText);
};

var bodyContent = new Object();
JIAClient.post(bodyContent, requestParams);

```

Jacada Interaction Manager Integration

The following sections describe the integration of Jacada Interaction Manager scripts into Jacada WorkSpace:

- [Overview](#)
- [Administration Needed for Script Integration](#)
- [Project Development](#)
- [Messages](#)

Overview

The Jacada Interaction Manager Designer development environment provides the business analyst with the ability to develop scripts that provide processing for business-related interactions. In addition to general processing nodes and interaction elements, the Jacada Interaction Manager Designer includes the following Jacada WorkSpace-specific custom items, or processing nodes, for use in Jacada Interaction Manager scripts:

Computer Telephony Integration – Enables call handling control. This custom item provides a Jacada Interaction Manager script with the *Initiate Outbound Call*, *Cold Transfer*, *End Current Call*, *End Call Wrap-up* and the *Change Agent Status* operations.

Session Data Integration – Enables access to, and manipulation of, an agent's Jacada WorkSpace session information. This custom item provides a Jacada Interaction Manager script with the *Get Session Data* and the *Update Session Data* operations.

Business Logic Integration – Enables adding business interaction logic. This custom item provides a Jacada Interaction Manager script with the *Display Page* operation.

Jacada WorkSpace Integration – Enables the following from the script:

- Adding an agent note as a text string
- Sending a message to an agent as a text string
- Refreshing a portlet based on the portlet ID
- Set Jacada WorkSpace session data
- Get Jacada WorkSpace session data
- Send a custom message to Jacada WorkSpace

For information about developing Jacada Interaction Manager scripts, see the *Jacada Interaction Manager Designer* documentation.

Jacada WorkSpace supports running a Jacada Interaction Manager script as part of call handling, in one of the following modes:

Solo – where the Jacada Interaction Manager script is the only call handling script in the Jacada WorkSpace *Script* tab.

Side-by-side Interaction – where the Jacada Interaction Manager script is the initial script run during call handling in the Jacada WorkSpace *Script* tab. It is the script that controls the subsequent display of Jacada WorkSpace-developed Web pages. With side-by-side interaction mode, the available script presentation layouts are as follows:

Vertical – The Jacada WorkSpace *Script* tab is vertically divided between a Jacada Interaction Manager script display area and a Web page display area. The Jacada Interaction Manager script displays on the left side and the Web pages display on the right side of the *Script* tab.

Horizontal – The Jacada Interaction Manager script display area is positioned horizontally above the Jacada WorkSpace *Script* tab.

Administration Needed for Script Integration

In order for Jacada WorkSpace to integrate a Jacada Interaction Manager script into its call handling, the administrator must perform the following actions:

- Configure a Jacada Interaction Manager web application (a *.war* file).
- Deploy the configured Jacada Interaction Manager web application to an application server that is certified to support Jacada WorkSpace.
- Publish a Jacada Interaction Manager script to the Jacada Interaction Manager web application.

Note: For a description of the above actions, see the *Jacada Interaction Manager Installation and Administration Guide*.

- Configure the *External Script Settings* page of the Jacada Management Console to reference the Jacada Interaction Manager script that must be run for a given project and telephone number. For information about the *External Script Settings* page, see the *Jacada WorkSpace Administrator Guide*.

Jacada Management Console Pages Used with Script Integration

In order for Jacada WorkSpace to integrate the running of Jacada Interaction Manager scripts into its call handling, Jacada WorkSpace uses the information defined in the following Jacada Management Console pages:

RTN Mappings – Contains the mapping of Jacada WorkSpace project entries to their associated telephone number and script. Jacada WorkSpace requires this page to be configured to support running a Jacada Interaction Manager script, either in solo mode or in side-by-side interaction mode.

Script Mappings – Contains the mapping of script entries to their associated URL. The Jacada Interaction Manager script name and the URL to its location are provided in this page. Jacada WorkSpace requires this page to be configured to support running a Jacada Interaction Manager script, either in solo mode or in side-by-side interaction mode.

External Script Settings – Contains the settings that are specific to the side-by-side interaction mode, including if this mode is enabled and the associated layout and display in Jacada WorkSpace of a Jacada Interaction Manager script operating in this mode.

Page Mappings – Contains the mapping of project Web pages to their relative project path. For example, `/USER/resources/jsp/myPage.jsp`. Based on processing states, one or more of the page's Web page entries are displayed. In support of the side-by-side interaction mode of a Jacada Interaction Manager script, this page contains any Web pages that Jacada WorkSpace displays in the *Script* tab as instructed by the Jacada Interaction Manager script. The default .jsp pages are listed in the following table.

Default .jsp Pages

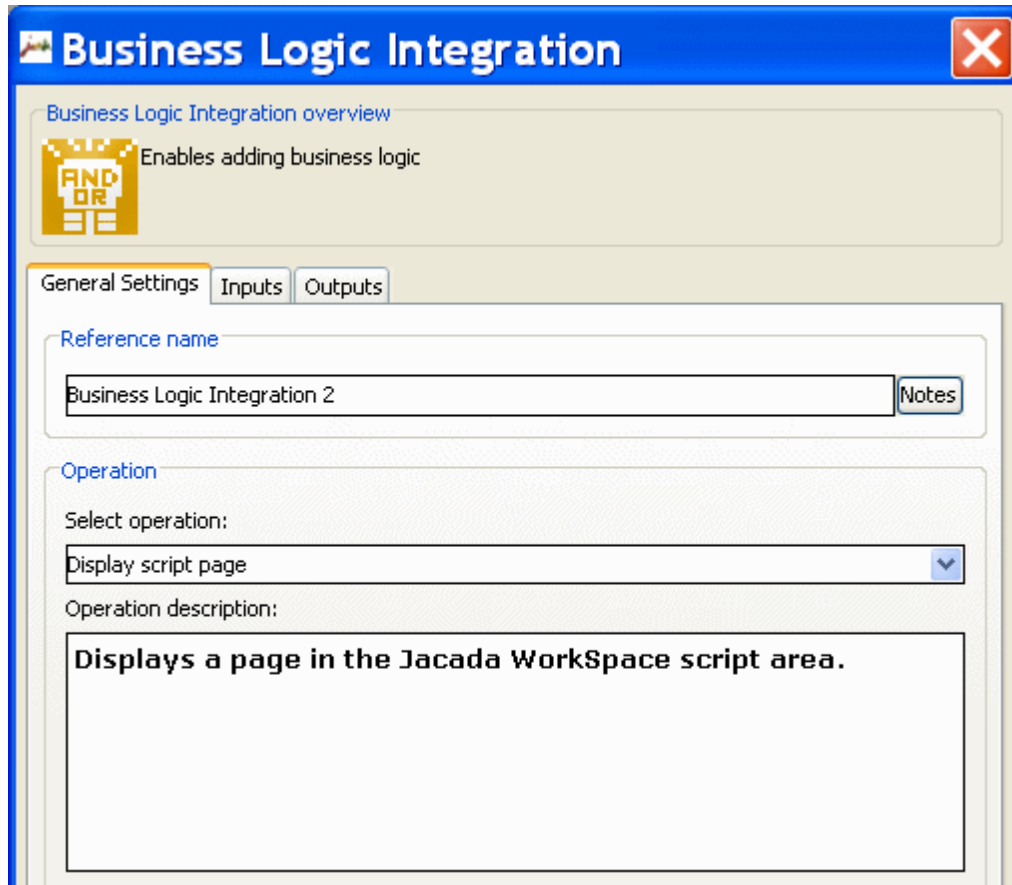
Page ID	Description
scriptStartPage	The page initially displayed in the Web page display area of the script tab when the Jacada Interaction Manager script starts. The default page can be replaced with a custom developed page.
missingMappingPage	The page displayed in the script tab whenever Jacada WorkSpace cannot locate the Web page that the Jacada Interaction Manager script instructed it to display. The default page can be replaced with a custom developed page.
horizontalExternalScriptDisplayPage	The page displayed in the Jacada Interaction Manager script display area, prior to a Jacada Interaction Manager script starting, when the horizontal layout is used for the display area. The default page can be replaced with a custom developed page.
collapsedExternalScriptDisplayPage	The page displayed by Jacada WorkSpace over the collapsed Jacada Interaction Manager script display area in the script tab. The default page can be replaced with a custom developed page.

Configure the relevant RTN Mappings, Script Mappings, External Script Settings and Page Mappings information using the Jacada Management Console. See the Jacada WorkSpace Administrator Guide for the necessary procedures.

Project Development

Integrate, in the side-by-side interaction mode, the call handling processing of a Jacada WorkSpace project together with the operation of the Jacada Interaction Manager script. This supported integration operates as follows: During call handling, the Jacada Interaction Manager script is the initial script to be invoked and displayed; at one or more specific processing points in the flow of the Jacada Interaction Manager script, the script instructs Jacada WorkSpace as to which project Web page to display in the *Script* tab.

Use the Jacada Interaction Manager Designer-provided custom item *Business Logic Integration* coupled with its *Display Page* operation, to incorporate these “instruction” points into your Jacada Interaction Manager script for side-by-side interaction mode processing.



Extending Jacada WorkSpace Handling of Call Operations

You can extend the project's handling of call operations, which it was instructed to perform by a Jacada Interaction Manager script. In the Jacada Interaction Manager script, use of the custom item *Computer Telephony Integration* coupled with either of its operations, *End Current Call* or *End Call Wrap-up*, instructs the Jacada WorkSpace project to proceed with handling either of these call operation situations.

In addition to the Jacada WorkSpace-provided handling of each of these call operation situations, extend the provided handling by adding your own custom code in the `src/USER/services/callHandling/CallHandlingService.java` file. Add the custom code to the

`handleCallOperation()` method, which is located in the `CallHandlingService` class definition. Add code that evaluates the operation parameter value that is provided to the project by the custom item operation specified in the Jacada Interaction Manager script. This value is a custom value other than default ones of `end_call` or `end_wrapup`.

Example: Where to Add Custom Code to Extend the Handling of Call Operations

This example illustrates where to add your custom code in the `CallHandlingService.java` file that will extend the call operation handling:

```

public class CallHandlingService extends
DefaultCallHandlingService {
    private static final long serialVersionUID =
1L;

    public void
handleCallOperation(HashMap<String, String>
input) {
    super.handleCallOperation(input);
    String operation =
input.get(CallHandlingKeys.OPERATION);
    if (operation == null){
        return;
    }
    try {
        if
(operation.equalsIgnoreCase(CallOperationValues.
END_CALL_OPERATION)) {

CTIHelper.getHelper(getSessionInformation().getS
ession()).

                disconnect();
        }
        else if
(operation.equalsIgnoreCase(CallOperationValues.

END_WRAPUP_OPERATION)){

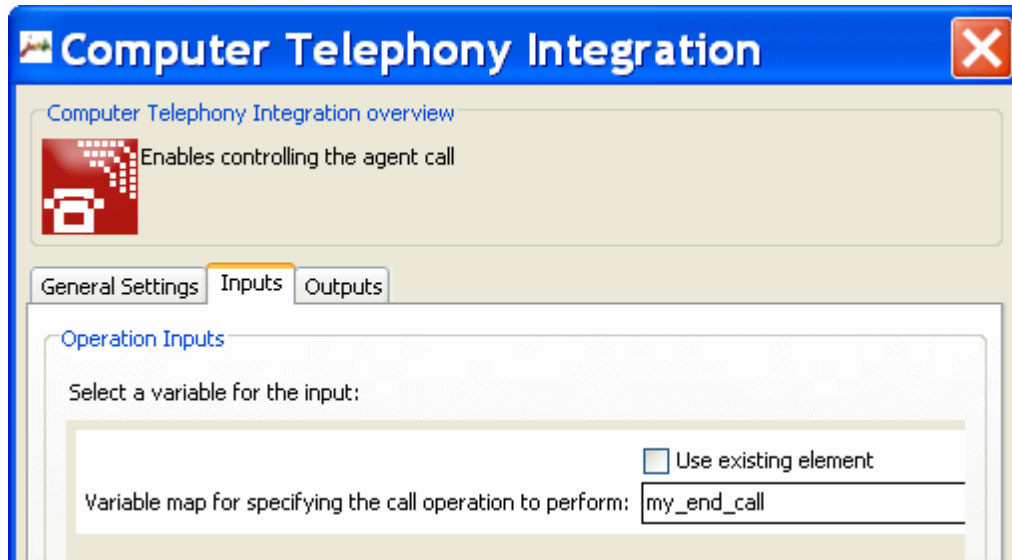
PushHelper.publishMessage(getSessionInformation(
).getSession()).

getId(), "endJimScript","no info");
        }
        else
if(operation.equalsIgnoreCase("my_end_call")) {
            // ADD CUSTOM HANDLING HERE When the
received
            // operation parameter value is
my_end_call
        }
    }
    ...
}

```

The operation parameter value `my_end_call` is provided to Jacada WorkSpace by the Jacada Interaction Manager script that interacts, in side-by-side mode, with this project. In the script's Jacada Interaction Manager development environment, you must incorporate a *Computer Telephony Integration* custom item that uses the *End Current Call* operation:

This operation specifies the `my_end_call` operation parameter value as its input.



Example: Extending the Handling of Call Operations

This example illustrates adding custom code to the `handleCallOperation()` method that extends the handling of the operation `end_call`. The code checks for the custom operation parameter value `my_end_call`, received from the Jacada Interaction Manager script, and then proceeds to both end the call and add a note to the *Agent Notes* portlet:


```

public class CallHandlingService extends
DefaultCallHandlingService {
    private static final long serialVersionUID =
1L;
    public void
handleCallOperation(HashMap<String, String>
input) {
    ...
    if(operation.equalsIgnoreCase("my_end_call"))
{

        //First, end the call

CTIHelper.getHelper(getSessionInformation().getS
ession()).disconnect();

        //Second, add a note to the Agent Notes
portlet

PushHelper.publishMessage(getSessionInformation(
).getSessionId(),

"onJimAgentNote","MY NOTE TEXT");
    }
    ...
}
}

```

Messages

When side-by-side interaction is the Jacada Interaction Manager script mode of operation, Jacada WorkSpace produces the following project related messages.

Messages Presented to the Agent

The following messages are presented to the agent in the Web page display area of the *Script* tab:

Condition	Message
The page mapping is missing.	Requested page <i><page_id></i> cannot be displayed.

The following messages are presented to the agent in the Jacada Interaction Manager script display area:

Condition	Message
The Script ID mapping is missing for the Jacada Interaction Manager script.	Jacada Workspace cannot display the script for the RTN <i><rtn></i> .

Log File Messages

The following are messages generated to the relevant Jacada Workspace log file:

Condition	Message
The page mapping is missing.	Requested page <i><page_id></i> cannot be displayed. The supplied page ID does not exist in the database.
The Script ID mapping is missing for the Jacada Interaction Manager script.	Jacada Workspace cannot display the script for the RTN <i><rtn></i> . There is no URL associated with script ID <i><script_id></i> in the database.

Integrating a Jacada HostFuse Project in Jacada WorkSpace

The following section describes how to integrate Jacada WorkSpace with Jacada HostFuse.

Integrating Jacada WorkSpace with a Jacada HostFuse Web Service

Jacada HostFuse services are exposed as web services. Web services are used for calls between Jacada HostFuse and Jacada WorkSpace. Thus you need to define a web service client to access the HostFuse web service.

To call a Jacada HostFuse service from Jacada WorkSpace:

1. In the development environment, define a Web Service Client to generate client side code based on the Jacada HostFuse service WSDL.
2. Select *File > New > Web Service Client*, either from the displayed list or by selecting *Other* and then from the *New* dialog using the filter if necessary.
The *New Java Web Service Client* dialog is displayed.
3. In the Web Service Client Dialog do the following:
 - a. Paste the WSDL URL of the Jacada HostFuse Web service.
 - b. Using the slider control, lower the level of client generation from `deploy` to `assemble`.
 - c. Click the link for selecting the web service runtime and select the Apache Axis Web Service runtime and select the server that the project is running on.
Then, click *OK*.
 - d. Click *Next*.
 - e. Select the source directory of your project as the output folder.
 - f. Click *Finish*.

Note: It may take several minutes for the code to be generated. Note that if the generated code contains errors (due to Axis bugs), delete the generated package and regenerate it.

4. Use the following code as a template to define a Spring bean:

```
package USER.com.jacada;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;

import org.apache.axis.message.SOAPHeaderElement;

import com.jacada.dev.HF.wsdlgen.svcHF.DtSignInInput_XSD;
import com.jacada.dev.HF.wsdlgen.svcHF.DtSignInInput_XSDWrapper;
import com.jacada.dev.HF.wsdlgen.svcHF.MthColors;
import com.jacada.dev.HF.wsdlgen.svcHF.MthColorsResponse;
import com.jacada.dev.HF.wsdlgen.svcHF.MthGetSystemDataResponse;
import com.jacada.dev.HF.wsdlgen.svcHF.SvcHFLocator;
import com.jacada.dev.HF.wsdlgen.svcHF.SvcHFStub;
import com.jacada.dev.HF.wsdlgen.svcHF.TtColors_XSD;
import com.jacada.jad.logging.LogWrapper;
```

```

public class HFUsage {
    public static final String NAMESPACE = "http://
svCHF.wsdlgen.HF.dev.jacada.com";
    private SvcHFStub svcHFStub;
    private SvcHFLocator locator;
    private String luName;
    private String hostName;
    private int hostPort;
    private String sessionName;
    private boolean stopSession;

    public SvcHFStub getSvcHFStub(){
        if(locator == null){
            locator = new SvcHFLocator();
        }
        if(svcHFStub != null){
            try{
                svcHFStub = (SvcHFStub)locator.getsvCHFPort();
            } catch(Exception e){
                LogWrapper.error(e);
            }
        }
        return svcHFStub;
    }
    public void checkDisplayValue(){
        SvcHFStub service = getSvcHFStub();
        SOAPHeaderElement header = getSOAPHeaders();
        if(header != null){
            service.setHeader(header);
        }
        try{
            MthGetSystemDataResponse mthGetSystemDataResponse =
                service.mthGetSystemData();
            System.out.println("Display: " + mthGetSystemDataResponse.
                getSystemData().getDtSystemData_XSD().getDisplay().getValue());
            System.out.println("System: " + mthGetSystemDataResponse.
                getSystemData().getDtSystemData_XSD().getSystem().getValue());
        } catch(Exception e){
            LogWrapper.error(e);
        }
    }
}

```

```

public TtColors_XSD[] getColors(){
    SvcHFStub service = getSvcHFStub();
    SOAPHeaderElement header = getSOAPHeaders();
    if(header != null){
        service.setHeader(header);
    }
    DtSignOnInput_XSD dtSignOnInput_XSD = new DtSignOnInput_XSD("milfirov",
"milfirov");
    DtSignOnInput_XSDWrapper signOnInput = new
DtSignOnInput_XSDWrapper(dtSignOnInput_XSD);
    MthColors mthColors = new MthColors(signOnInput);
    try{
        MthColorsResponse mthColorsResponse = service.mthColors(mthColors);
        return mthColorsResponse.getColors().getColors_XSD();
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
    return null;
}

private SOAPHeaderElement getSOAPHeaders() {
    SOAPHeaderElement jiHeader = new SOAPHeaderElement(new QName(NAMESPACE,
                                                                    "JIHeader"));
    try{
        SOAPElement header = jiHeader.addChildElement("sessionName");
        header.setValue(sessionName);
        header = jiHeader.addChildElement("stopSession");
        header.setValue(Boolean.toString(stopSession));
        header = jiHeader.addChildElement("hostName");
        header.setValue(hostName);
        header = jiHeader.addChildElement("hostPort");
        header.setValue(Integer.toString(hostPort));
        header = jiHeader.addChildElement("LUName");
        header.setValue(luName);

        return jiHeader;
    } catch(SOAPException se){
        System.err.println("Exception creating jiHeader: " + se.getMessage());
        se.printStackTrace(System.err);
    }
    return null;
}

```

```

public String getHostName() {
    return hostName;
}

public void setHostName(String hostName) {
    this.hostName = hostName;
}

public int getHostPort() {
    return hostPort;
}

public void setHostPort(int hostPort) {
    this.hostPort = hostPort;
}

public String getLuName() {
    return luName;
}

public void setLuName(String luName) {
    this.luName = luName;
}

public String getSessionName() {
    return sessionName;
}

public void setSessionName(String sessionName) {
    this.sessionName = sessionName;
}

public boolean isStopSession() {
    return stopSession;
}

public void setStopSession(boolean stopSession) {
    this.stopSession = stopSession;
}
}

```

5. Add the bean to the list of Spring beans.

Exposing Web Service Operations

Jacada WorkSpace exposes a Web service that contains a predefined set of operations, which can be used by other applications. The Web service is stateful, which means that after you connect to a specific Jacada WorkSpace session, all operations will be done in the context of that session.

The Jacada WorkSpace predefined Web service operations are defined by the following URL:

`http://<host>:<port>/<project>/services/JacadaService?wsdl`

In addition to the predefined exposed operations, it is also possible to extend the Jacada Web service by adding custom operations in `src/USER/external/JacadaCustomService.java`.

The following sections describe:

- [Predefined Web Service Operations](#)
- [Adding Custom Web Service Operations](#)

Predefined Web Service Operations

The following table lists the predefined Web service operations in `JacadaService`:

Topic	Operations	Description
Nesting Applications	<code>public void ShowTab(String tabId)</code>	Places the specified tab in focus. Used to switch between Jacada WorkSpace tabs.

Topic	Operations	Description
–	<code>public void releaseSession()</code>	Used to clean the session identifier from the SOAP header.
Push Mechanism	<code>public void pushMessage(String keyword, String json)</code>	Server-side pushes to the client-side (the browser) a JSON formatted string, which represents an object. The client-side uses the provided keyword to identify registered Java Script functions.
	<code>public void pushString(String keyword, String str)</code>	Server-side pushes to the client-side (the browser) a simple string (non-JSON). The client-side uses the provided keyword to identify registered Java Script functions.

Adding Custom Web Service Operations

Jacada Workspace provides a mechanism for adding custom Web service operations to be exposed. Custom operations are defined in the `IJacadaCustomService` interface, and are implemented in `src/USER/external/JacadaCustomService.java`

The `JacadaCustomService.java` file contains an infrastructure for adding custom operations to the Web service. This infrastructure includes the `connectToSession` and `releaseSession` operations, to enable using the custom operations within the scope of a specific session.

To add a custom Web service operation, you write the logic of the operation in `JacadaCustomService.java`, and then add the operation to the list of exposed operations in `IJacadaCustomService`. This interface is marked as a Jacada exposed service, using the `@ExposedService` annotation. For every custom operation you want to expose, you need the `@ExposedOperation` annotation, which receives the name of the method to expose as a parameter.

Example

The following example demonstrates how to add a custom Web service operation, `getNumber`, that

is given a limit parameter, and returns a random number between zero and the limit.

You begin by adding to `JacadaCustomService.java` the method, `getNumber`, to be exposed by the Web service:

```
public int getNumber(int limit) {  
    Random rnd = new Random();  
    return rnd.nextInt(limit);  
}
```

Then, in the `IJacadaCustomService` interface, add the following:

```
@ExposedService(name = "JacadaCustomService",  
                targetNamespace = "http://  
www.jacada.com")  
public interface IJacadaCustomService extends  
    IInvocableService {  
    @ExposedOperation(name = "getNumber")  
    int getNumber(int y);  
}
```

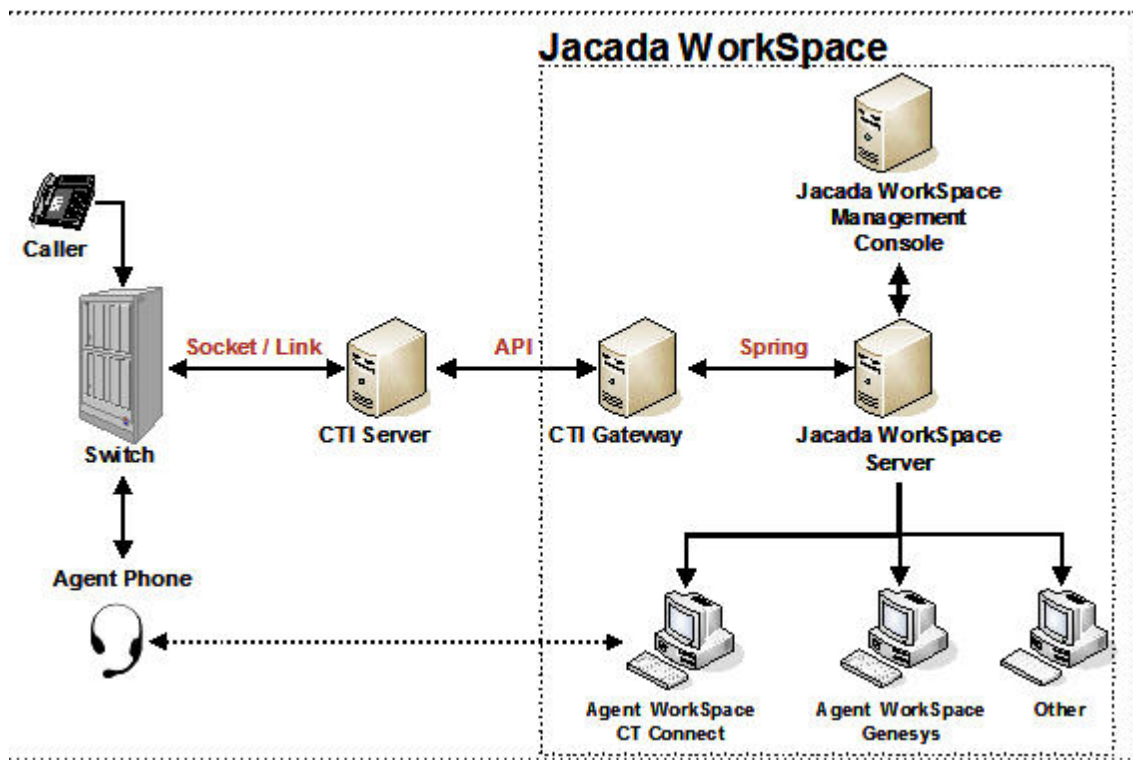
Jacada Custom Service WSDL

After you build and deploy your project, you can access the Jacada custom service WSDL using the following URL:

`http://<host>:<port>/<project>/services/JacadaCustomService?wsdl`

Computer Telephony Integration

Jacada WorkSpace provides a CTI Gateway to support computer telephony integration in Jacada WorkSpace. The CTI Gateway is packaged as an EAR file, that is deployed during installation to the application server. The CTI Gateway uses APIs and Spring remoting services to pass information between the Jacada WorkSpace application and the CTI server. The following diagram demonstrates the Jacada WorkSpace architecture for using CTI.



Jacada WorkSpace provides gateways to support the following CTI providers:

- Avaya Inc. – Application Enablement Service (AES)
- Avaya Elite Multichannel
- Cisco Systems, Inc.
- Genesys Telecommunications Laboratories Inc.

The functionality supported in Jacada WorkSpace depends on the CTI provider, and may vary between providers.

The following sections present:

- [Developing CTI Functionality](#): Describes general CTI functionality available to the developer (relevant to all supported CTI providers).
- [CTI Functionality for Avaya one-X Agent](#)
- [CTI Functionality for Cisco](#)
- [CTI Functionality for Genesys](#)

Developing CTI Functionality

The following sections describe the CTI functionality available to the developer that applies to all supported CTI providers:

- [Setting Up the Call Mode of an Application: Inbound or Outbound](#)
- [Setting Up Transfer and Consultation](#)
- [Handling Direct-Dial Calls](#)
- [Logging Into CTI Via the Phone](#)
- [Configuring Jacada WorkSpace to Automatically Answer Incoming Calls](#)
- [Using CTI Functionality Within a Portlet or Call-Handling Script](#)
- [Configuring Disposition Script Behavior](#)
- [Enabling Call Disposition Before Ending the Call](#)
- [Disabling CTI Buttons](#)
- [Managing Jacada WorkSpace Indicators](#)
- [Overriding the Default CTI Behavior](#)

- [Obtaining Information About Calls](#)
- [Attaching Data to a Call](#)
- [Configuring the Dial List](#)
- [CTI Integration with External Applications](#)

Setting Up the Call Mode of an Application: Inbound or Outbound

A Jacada WorkSpace application that uses CTI can handle either inbound or outbound calls. This is defined by the `CTICallMode` setting in the `CTI_SETTINGS` table.

You can also specify an access code for outbound calls, that appends a number to the dialed number. This is necessary, for example, when you need to dial a specific number, such as 9, to obtain an external phone line. To specify an access code for outbound calls, use the `CTIOutboundAccessKey` setting in the `CTI_SETTINGS` table.

Note: Agents can make outbound calls from inbound mode.

Overriding the Call Mode of a CTI Profile at the Project Level

It is possible to override the call mode of a CTI profile at the project level, by adding the following function in `src/USER/global/managers/CTIBehaviorHandler.java`:

```
/*
public boolean isOutboundAllowed()
{
    // return false; // inbound
    // return true;  // outbound
}
*/
```

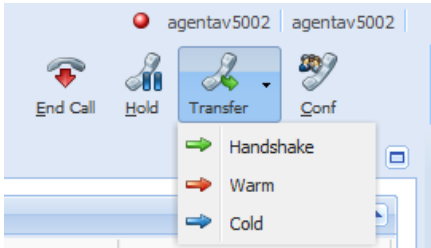
Setting the call mode in the code overrides the setting in the database. In this manner, you can use the same profile for different projects using different call modes.

Setting Up Transfer and Consultation

You configure the transfer types supported by the CTI provider and the switch in use in the database, as well as the default transfer type when there is more than one supported type.

Note: The transfer button functionality is configured in the CTI_SETTINGS table, using the following settings: CTITransferTypesList and CTIDefaultTransferType and CTISupportTransferFromConsult. For more information, see the Jacada WorkSpace Administrator Guide.

When more than one transfer type is configured, the *Transfer* button is displayed with a down arrow. Clicking the *Transfer* button activates the default transfer type, whereas clicking the arrow opens a drop-down list of the supported types. For example:



Note: The default transfer type is displayed in balloon text when hovering with the mouse over the button.

Setting the Numbers an Agent Can Use

You can set up the CTI system so that when an agent wants to dial a number, transfer a caller or consult with someone, one or more of the following options are available:

Dialing Option	Define in:	How
Connect directly to a predefined number.	Script code.	Using CTI functions, such as: - dial() - transfer() - consultation()

Dialing Option	Define in:	How
		For more information, see Using CTI Functionality Within a Portlet or Call-Handling Script .
Provide a dropdown list from which to select a number.	<code>src/USER/global/managers/ExtensionManager.java</code>, to initialize Jacada WorkSpace values. - You can also use the functions throughout a script, as necessary.	For more information, see Setting a List of Numbers .
Use the <i>Dial Pad</i> window, to specify the number to dial.	Provided with Jacada WorkSpace.	Select the <i>Dial number</i> option from the drop-down list.

Setting a List of Numbers

You can set up the CTI system so that when an agent clicks the *Dial*, *Transfer* or *Conf* buttons, a drop-down list opens with a list of possible numbers, from which the agent can select a number. If a button has no predefined numbers, the Dial Pad window opens automatically and enables the agent to specify the number to use for the operation. In addition, you can add numbers to the predefined list of numbers using the *Dial List* page of the Jacada Management Console. For information about this page, see the Jacada WorkSpace Administrator Guide.

The list of numbers is defined within the login method in the `src/USER/global/managers/ExtensionManager.java` file.

To set to a list of numbers:

1. Specify the list of numbers in the login method in `src/USER/global/managers/`

ExtensionManager.java.

2. To set a list of numbers to dial, add a line such as the following for each number:

```
globalApp.getCTIManager().getAgentInformation().  
                                addOutboundDialInfo(info);
```

To set a list of numbers to use to transfer a call, add a line such as the following for each number:

```
globalApp.getCTIManager().getAgentInformation().  
                                addOutboundTransferInfo(info);
```

To set a list of numbers to use for consulting, add a line such as the following for each number:

```
globalApp.getCTIManager().getAgentInformation().  
                                addOutboundConsultInfo(info);
```

where `info` is the following comma separated list:

number – The number to use, such as an internal extension number, an external telephone number, the number of a queue defined in the switch, or any other number that can be dialled.

name – A name that identifies the entry.

description – A description of the entry.

use access code – Indicates whether to use the access code. Specify true for an external number, or false for an internal number. For more information, see [Setting Up the Call Mode of an Application: Inbound or Outbound](#).

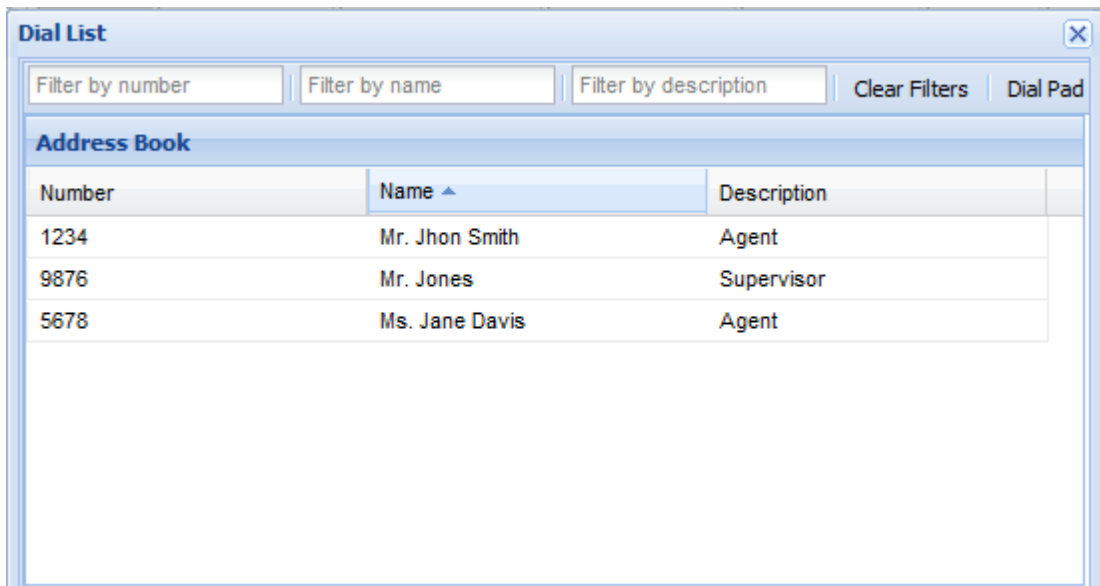
Corresponding methods are available to clear the list of values: `clearOutboundDialInfo`, `clearOutboundTransferInfo` and `clearOutboundConsultInfo`.

Example – Setting Up a List of Numbers for the Dial Button

The following code in the `login` method within the `src/USER/global/managers/ExtensionManager.java` file, sets up a list of numbers to display for the *Dial* button:

```
globalApp.getCTIManager().getAgentInformation().addOutboundDialInfo(new  
    CTIOutboundInfo("111","Mr. John Smith","Agent", false));  
globalApp.getCTIManager().getAgentInformation().addOutboundDialInfo(new  
    CTIOutboundInfo("123","Ms. Jane Davis","Agent", false));  
globalApp.getCTIManager().getAgentInformation().addOutboundDialInfo(new  
    CTIOutboundInfo("130","Mr. Jones", "Supervisor", false));
```

As a result, the following list is displayed in Jacada WorkSpace when the agent clicks *Dial*:



The ability to use the Dial Pad window is configured by the `CTIAllowDialPadWindow` setting in the `CTI_SETTINGS` table.

Handling Direct-Dial Calls

By default, calls received in the following cases are defined by Jacada WorkSpace as direct-dial calls:

- When Jacada WorkSpace status is *Not Ready*
- During call disposition
- When working in outbound mode

For calls that are received in other cases, the following method is activated:

```
public boolean isInternalCall(String ani, String dnis, String callType,
String callMode)
```

defined in `src/USER/global/managers/CTIBehaviorHandler.java`

where:

ani – The ANI of the call. The value must be lowercase.

dnis – The DNIS of the call. The value must be lowercase.

callType – The call type as supplied by the switch.

callMode – The call mode as supplied by the switch.

This method enables customizing the decision whether the received call is a non-business call, and should therefore be treated as a direct-call. By default, this method defines the call as a direct-dial call if the call type, supplied by the switch for this call, is 'internal' or 'consultation'. However, in some cases, the call type provided by the switch may not be accurate, and may require using the `ani` and `dnis` parameters to determine whether a call is a direct-dial call.

Logging Into CTI Via the Phone

By default, logging into or out of the CTI is done automatically as part of logging into or out of Jacada WorkSpace.

However, when using the **voice channel only**, in some cases, you may need the agent to perform the login to CTI via the phone, not automatically. For example, to take advantage of switch-specific login options or due to switch requirements. To log in to the CTI voice channel using the phone, you use the `CTILoginByPhone` setting in the `CTI_SETTINGS` table.

When logging in via the phone, the agent must first log in to Jacada WorkSpace and only then log in to CTI via the phone.

Note: When the agents logs into CTI via the phone, the agent **must also log out of CTI via the phone**.

Configuring Jacada WorkSpace to Answer Incoming Calls

Answering calls automatically is done either by the switch, or by Jacada WorkSpace. This is configured in the `CTI_SETTINGS` table.

To configure calls to be answered automatically:

1. Set the `CTICallAutoAnswer` setting to `true`.
2. Specify whether the switch is configured to automatically answer ACD calls, using the `CTISwitchAutoAnswer` setting.

When you do not want calls to be answered automatically, set the `CTICallAutoAnswer` setting to `false`, and then verify with your switch administrator that the switch is also configured accordingly. In this case, the value of the `CTISwitchAutoAnswer` setting is not used.

Configuring Auto-Answer Functionality Per Call

In addition to the database settings that configure the call answer mode in general, it is also possible to determine the auto-answer functionality on a per-call basis. This is done by adding the following API:

```
Public boolean autoAnswerIncomingCall();  
to src/USER/global/managers/CTIBehaviorHandler.java.
```

This method is called when Jacada WorkSpace receives an incoming call. The method is implemented at the SYSTEM level to check the database settings. You can override the database settings, as necessary. For example, the developer can use the attached data in the session, or the `PartyManager` object, to determine whether to auto-answer a specific call.

For more information on attached data, see [Attaching Data to a Call](#).

The `PartyManager` object contains information regarding the current call, such as ANI, DNIS, call ID. This information can be retrieved using getter methods on the `PartyManager` object:

```
IPartyManagerInfo voiceManager =  
getVoiceOperationProxy().getPartyManagerInfo();
```

Note: When Jacada WorkSpace answers calls automatically, both ACD calls and direct incoming calls are answered.

For more details, refer to [Obtaining Information About Calls](#).

Modifying Call Answering Behavior Based on CTI Data

By default, Jacada WorkSpace automatically reacts to all incoming calls. However, there is an option to configure WorkSpace to ignore certain incoming calls, based on parameters that are defined at the project level. For example, some calls can be configured to be ignored by WorkSpace, and handled by the agent exclusively within the CRM application.

The decision to ignore a call is made at the project level, based on call data. In order to make the decision, the following incoming call information is passed to the project level:

- Call id

- DN
- DNIS
- Call type (as defined by the CTI API)
- Attached data

Any of this information can be used to decide whether or not the call should be ignored.

Client Side Conditions

The agent must be in a *Ready* state in order to initiate the ignore operation. (This is validated by the product.) The ignore operation will not be initiated in any other agent state.

When a call is ignored, the agent workspace UI is updated as follows:

- The CTI soft phone buttons are disabled.
- The status text is set to reflect the new state (e.g., *Disabled*).
- Values about the ignored call, or any other CTI-related data, are not updated.

Server Side Configuration

By default, WorkSpace does not ignore any type of call. In order to initiate the ignore operation, a method is overridden at the project level. This method determines whether or not to ignore the call by returning the `command` object.

The `command` object implements the command interface and holds the following members:

- `Call id`
- `Operation`

In order to ignore the incoming call, it is required to set the command object members as follows:

- `Call id`: Call id that the operation is performed on.
- `Operation`: The value `ignore`, selected from the operations enumeration object.

The following example describes which method should be overridden in **CTIBehaviorHandler.java**. Upon an incoming call, this method will be called to acquire the operation to be performed.

In the example, `call type` (as taken from the attached data) is used to decide whether the call should be ignored. If the call type is `ACDIn`, the incoming call is ignored, by sending the `IGNORE_INCOMING_CALL` command. Otherwise, no operation is performed.

```

import net.sf.json.JSON;
import net.sf.json.JSONObject;
import net.sf.json.JSONSerializer;

...

@Override
    public IModifyCommand modifyCTIBehavior(ICTIData data)
    {
        LogWrapper.debug("in method modifyCTIBehavior");
        LogWrapper.debug("data call id: " + data.getCallId());

        JSONObject attachDataObject = null;

        Object attachData = data.getAttachedData();
        if (attachData != null) {
            JSON attachDataJson = JSONSerializer.toJSON(attachData);
            attachDataObject = JSONObject.fromObject(attachDataJson);
            attachDataJson = JSONSerializer.toJSON(attachDataObject);

            System.out.println(attachDataJson.toString());
        }

        String value = null;
        if(attachDataObject !=null){
            try{
                value = attachDataObject.getString("CALLTYPE");
            }catch(Exception e){
                //handle exception
            }
        }

        ModificationOperations operation = null;

        if(value !=null && value.equalsIgnoreCase("ACDIn")){
            operation = ModificationOperations.IGNORE_INCOMING_CALL;
        }else{
            operation = ModificationOperations.NONE;
        }

        LogWrapper.debug("modification operation: " + operation);
        return ModifyCommand.getInstance(operation, data.getCallId());
    }

```

Using CTI Functionality Within a Portlet or Call-Handling Script

You can include specific CTI functionality as part of a call-handling script, enabling the agent to use this functionality within the script, without the necessity of using the soft phone. Including CTI functionality in a call-handling script is done using the `CTIHelper` class.

To use a CTI function in your script, use the following code:

```
CTIHelper.getHelper(this.getSession()).<CTIMethod>;
```

For example, to dial to extension number 123, use the following code:

```
CTIHelper.getHelper(this.getSession()).dial("123");
```

CTI APIs

The following table describes the CTI APIs that can be used in a script.

Function	Description
<code>answer()</code>	Answer an incoming call. By default, this function answers calls automatically.
<code>coldTransfer (String number)</code>	Perform a cold transfer to the specified number.
<code>completeWarmTransfer()</code>	After talking to the supervisor, complete the transfer by transferring the caller to the supervisor.
<code>consultation(String number)</code>	Consult another agent or supervisor. The caller is put on hold.

Function	Description
<code>dial (String number)</code>	Dial to a specific number. If you use this function, disable the <i>Dial</i> button, otherwise, the agent can dial a number at the same time the script is attempting to dial, resulting in an error. For more information, see Disabling CTI Buttons.
<code>disconnect ()</code>	End a call.
<code>handshakeTransfer (String number)</code>	Perform a handshake transfer to the specified number.
<code>handshakeTransferConnect (String number)</code>	Perform a handshake transfer using transfer connect, which enables a transfer to be implemented by sending Dual Tone Multiple Frequencies (DTMF) tones to the switch, instead of using the transfer API.
<code>hold ()</code>	Put the caller on hold.
<code>holdOff ()</code>	Return to the caller.
<code>joinConference ()</code>	After a consultation, include the caller in the conversation, initiating a conference call between all three parties.
<code>logout ()</code>	Log out of the CTI server.
<code>returnFromConsultation ()</code>	After a consultation, return to the caller.
<code>returnFromWarmTransfer ()</code>	After talking to the supervisor, return to the caller, instead of transferring the call.

Function	Description
<code>sendDTMF(String customerId, String number)</code>	Calls a customer ID using Dual Tone Multiple Frequencies (DTMF) tones to the switch.
<code>setAgentStatus(String status)</code>	Specify whether the agent is ready to receive calls, by setting the <code>status</code> to one of the following: <code>ready</code> <code>notready</code>
<code>setAgentStatusWithReason(String status, String reason)</code>	Use this method to specify that the agent is not ready to receive calls with a reason. Set the <code>status</code> to: <code>notready</code> The <code>reason</code> parameter is a string with a reason code that the switch recognizes. For example, when using an Alcatel switch, the reason code is a string between “1” and “9”.
<code>transfer(String number)</code>	Transfer the call to the specified number, using the default transfer type.
<code>transferFromConsultation()</code>	After a consultation, transfer the caller to the supervisor with whom the agent consulted.
<code>warmTransfer(String number)</code>	Perform a warm transfer to the specified number.
<code>warmTransferConnect(String number)</code>	Perform a warm transfer to the specified number using transfer connect, which enables a transfer to be implemented by sending Dual Tone Multiple Frequencies (DTMF) tones to the switch, instead of using the transfer API.

Note: The use of some functions, such as `handshakeTransfer` or `warmTransfer`, depends on the functionality being supported by the CTI provider and switch in use.

Retrieving Reason for Ending Call Within Call Disposition Script

When a call ends and the call disposition script is started, you can check the reason for starting the call disposition script and take appropriate action depending on the reason. For example, if the agent ended the call normally, the call disposition script may not include any special processing. However, if the call ended for other reasons, such as a wrong number was dialed or the number dialed was busy, the call disposition script can handle the specific reason.

Jacada WorkSpace provides the following function for retrieving, within the call disposition script, the reason for ending the call:

```
public String getCallTerminationReason();
```

The `getCallTerminationReason()` function returns a string that describes the reason for ending the call. The following table lists the reasons that are recorded for ending a call, and the corresponding string returned by the function:

Reason for Ending Call	String Returned by Function
Call ended normally	Call ended
Dialing a wrong number	Invalid dial destination
Dialing failed	Dialing failed
Dialing a busy number	Busy number
Missed call	Missed call: <number>

To retrieve the reason a call ended in a call disposition script, use the following code:

```
globalApp.getCTIManager().getCtiBehaviorHandler(  
).getCallTerminationReason(arg0);
```

where `arg0` is for future use.

Note: The function is not supported when the CTI provider is CT Connect.

Configuring Disposition Script Behavior

When an agent finishes talking to a customer on the phone, the agent may need to wrap up the call by logging various call details, such as the reason for the call and the call resolution. The wrapping up of a call is also referred to as call disposition. During call disposition, Jacada WorkSpace is in a *After-Call-Work* status, and the agent is not available for handling new calls.

Note: During call disposition, Jacada WorkSpace sends to the CTI switch After Call Work (ACW) messages.

By default, when a agent ends a call and the *After-Call-Work* status appears, the call disposition script is displayed. However, the behavior of the disposition script can be configured, by overriding the following methods in the `src/USER/global/managers/CallScriptBehaviorHandler.java` class, as required:

- By default, the `loadDispositionScript` method loads the disposition using the specified URL. However, you can override it by configuring this method to perform other actions for example, to display no disposition script.
- By default, the `getDispositionScriptURL` method retrieves the URL. However, you can override this method to return to your custom script URL, for example.
- In JavaScript, the `getDispositionScriptURL` method exists in the `CTIClient.js`. By default, this method retrieves the URL. However, you can override this method in the `WebContent/USER/resources/js/override.js` file, to return to your custom script URL, for example.

Defining Call Disposition

Jacada WorkSpace provides a call disposition flow that serves as a basis for developing the call disposition necessary for your script. The call disposition flow is located in `WebContent/USER/`

flows/**callDisposition**.

The `callDisposition` flow ends with a predefined action, **endDispositionAction**, that is responsible for setting the values of the Jacada WorkSpace timers in call variables defined in `src/USER/flows/callDisposition/CallDispositionController.java`.

The `callDisposition` flow does not contain any predefined user interface. If it is necessary to obtain additional information from the agent at the end of a call, the developer needs to add to the flow the necessary user interface.

Navigating to the Call Disposition Flow

In addition to defining the call disposition for a script, it is necessary to navigate to the call disposition flow at the end of the script. This is done by adding a predefined JavaScript to the last JSP in the flow of the script. An example is provided in the `WebContent/USER/flows/sampleScript/End.jsp` file.

To navigate to the call disposition flow from your script, copy the following JavaScript from `End.jsp` and paste it into the last JSP in your flow:

```
<script language="JavaScript" type="text/javascript">
<!--
    function localDoNext() {
        var form = document.forms[0];
        if(form != null) {
            if (form.action == "goBack.do") {
                return;
            }
        }

        if (onSubmitValidation()) {
            $W().cti.startCallDisposition();
        }
    }
-->
</script>
```

When the page is submitted, the `startCallDisposition()` method is activated.

Enabling Call Disposition Before Ending the Call

To start the call disposition flow from a call-handling script, invoke the `$W().cti.startCallDisposition()` method in the flow of the script. The default `$W().cti.startCallDisposition(true)` ends the call before accessing the call disposition

script. To access the call disposition script while the caller is still online, set the method to `false`:
`$W().cti.startCallDisposition(false)`.

If access to the call disposition script is available from the navigation pane, the agent can advance to the disposition script at any time by clicking on the link. Since this operation does not change the status of the agent's CTI extension, the current call also does not end until the call disposition script ends.

Note: You can override the `getDispositionScriptURL` JavaScript method to return to the disposition script URL as described in [Configuring Disposition Script Behavior](#).

Disabling CTI Buttons

It is possible to disable one or more CTI buttons, if necessary. This is done by defining a JavaScript function named `overrideCtiInit` in `WebContent/USER/resources/js/override.js`.

In the `overrideCtiInit` function, list the CTI buttons to disable. For example, the following code disables the *Transfer* and *Hold* buttons:

```
function overrideCtiInit(ctiobject)
{
    // override CTI.js logic to disable buttons
    ctiobject.transferBtn.allowed = false;
    ctiobject.holdBtn.allowed = false;
}
```

Managing Jacada WorkSpace Indicators

Jacada WorkSpace includes indicators that enable the agent to visually view the call progress. The indicators described in this section contain predefined settings which can be changed as required.

Changing Jacada WorkSpace Counters to Red

If the progress bar is disabled, you can change the color of the Jacada WorkSpace counters, displayed in the call-handling bar, to red. This is performed using the following function: `getCallCounterThresholdInSec()`. This function returns the number of seconds to wait, before the color of the call counters turns to red. The default is 120 seconds (2 minutes).

To override this function to enable changing the time, create a function with the same name, in `/USER/resources/js/override.js`.

Note: This option is relevant only when the progress bar is disabled.

Managing the Progress Bar

The progress bar enables the agent to view the progress of business calls as well as the call handling time, for example, to monitor performance against an SLA (Service Level Agreement). You can configure the maximum time for the progress bar to complete as well as the action to take when at the maximum time reached. You can also configure whether or not to include the call disposition time as part of the progress bar. In addition you can restart the progress using a time parameter.

The progress bar exists by default. To change the default, perform the following procedure.

To configure the progress bar:

1. In the CustomSLABehaviour class:
 - Set the `IsDisplaySLAbar` to return false to remove the progress bar.
 - To include the call disposition time in the progress bar, set the `isDispositionTimeIncludedInSLA` to return true.
2. In `<project_name>\WebContent\USER\resources\js\override.js` customize your progress bar according to the project requirements:
 - In the `getSLAMaxTimeInSeconds` method, configure the maximum time for the progress bar to complete.
 - In the `onSLATimeExceeded` method, define the actions to take when the maximum SLA time is reached. The default action is for the bar to flash repeatedly.

To restart the progress bar with a specific time:

- In the `startSLAbar(seconds)` API, Java function, define a value in seconds for the time parameter to use when the progress bar restarts. This value overrides the value set in the `getSLAMaxTimeInSeconds`. However, if this time parameter value is empty, the value set in the `getSLAMaxTimeInSeconds` method will be used.

Overriding the Default CTI Behavior

The Default CTI behavior that can be overridden and the APIs can differ per CTI provider. For Cisco CTI, refer to [Overriding Default Workspace CTI Behavior in the Cisco Environment](#).

Genesys and Other Providers: Overriding Default CTI Behavior

You can customize CTI behavior in Jacada WorkSpace, by overriding the default built-in functionality. For this purpose, the `src/USER/global/managers/CTIBehaviorHandler.java` class is provided.

The `CTIBehaviorHandler` class is empty, and within this class you can override various methods that control the CTI behavior in Jacada WorkSpace.

CTI Methods that can be Overridden

The following table lists the CTI methods that can be overridden, including the default behavior, and examples of possible additional implementations:

Method	Default Behavior	Customization Example
<code>autoAnswerIncomingCall()</code>	Specifies whether to answer an incoming call automatically. The default, which is to automatically answer an incoming call, is derived from the following database settings in the <code>CTI_SETTINGS</code> table: <pre>CTICallAutoAnswer = True</pre> <pre>CTISwitchAutoAnswer = False</pre>	Use the call information, from the <code>PartyManagerInfo</code> object, to determine the auto-answer functionality per call.

Method	Default Behavior	Customization Example
<code>customItemAutoAnswer()</code> (Genesys only)	Specifies whether to answer an incoming custom item automatically. The default is derived from the following database setting in the CTI_SETTINGS table: CTICustomItemAutoAnswer = True	Use the custom item information, from the CustomItemParty-ManagerInfo object, to determine the auto-answer functionality per custom item.
<code>getCallTerminationReason(String callId)</code>	See <code>setCallTerminationReason</code> below.	
<code>isInternalCall(String ani, String dnis, String callType, String callMode)</code>	Determines the type of the call, either internal or business call. The default is determined according to the call type as received from the switch.	Use the call information, such as ani or dnis, to determine the type of the call. (See note below the table.)
<code>isOutboundAllowed()</code>	Determines whether outbound mode is allowed, for an entire agent session. The default behavior is to return the value of the following database setting in the CTI_SETTINGS table: CTICallMode	Determine whether outbound mode is allowed, according to the agent name for example.

Method	Default Behavior	Customization Example
<code>modifyCTIBehavior (ICTIData data)</code>	The default behavior is to not modify CTI behavior, and to not ignore any calls.	Overriding this method allows an option to ignore an incoming call, based on incoming call information. For more details, refer to Modifying Call Answering Behavior Based on CTI Data .
<code>onCTIOperationFailed Handler(String operation, String message)</code>	This method is called when a CTI exception is caught and handled. The default behavior is to print a log message.	Use this method to display a popup message to the agent with the failure details.
<code>returnDataToAttach(String operation, AbstractCTIState currentState, PartyManager partyManager)</code> (Genesys and CT Connect only)	Returns the string object to be sent as attached data in the following call operations: consult, handshake transfer and warm transfer. In CT Connect, any string can be used. In Genesys, the string must contain key-value pairs, separated by a . For example: name workspace password 1234 The default behavior of this method return a NULL string object (no data is attached to the operation).	Determine the data to attach according to the method parameters and the PartyManagerInfo. For example, on every transfer operation, attach the agent extension. The string must end with separator.

Method	Default Behavior	Customization Example
<code>setCallTerminationReason(String reason, String callId)</code> <code>getCallTerminationReason(String callId)</code>	The call termination reason is either “Call ended”, or an error message when the call ended due to an error. The <code>set...</code> method is used internally to set the call termination reason upon the end of a call. The default <code>set...</code> behavior saves the reason string internally. The default <code>get...</code> behavior retrieves this reason string. Note: The <code>callId</code> parameter is for future use. It can remain empty.	Override these methods when you need to perform actions with the reason string. For example, customize the <code>set....</code> method to write the reason to a database, and then customize the <code>get....</code> method to obtain the reason from the database.

Method	Default Behavior	Customization Example
<code>updateAgentOpenMediaTypes (I OpenMediaInformation openmediaInfo) (Genesys only)</code>	Sets the array list of the Open Media types. By default, this method is empty, which means leaving the list of Open Media types as is. For more information, see Personalizing What the Agent Can See and Do.	The following example adds the Open Media type "WorkItem" to the agent Open Media types list: <pre>ArrayList<String> list = new ArrayList<String> (); list.add ("WorkItem"); openmediaInfo.updateAgentOpenmediaTypes (list);</pre>

Note: In a Genesys-Avaya environment, when the agent works in either Outbound mode or Inbound mode with manual dial and the agent places an outgoing call, the DNIS (dialed number) and the ANI (calling party number) values are not supplied to Jacada WorkSpace when the `isInternalCall()` method is invoked to determine whether the call is an internal call or not. The project code should retrieve these values from another place, for example, by capturing the DNIS when the agent dials the number.

CTI Methods that can be Extended

The following table lists the CTI methods that can be extended:

Method	Default Behavior
<code>endDisposition()</code>	
<code>endInboundBusinessFlow()</code>	
<code>endManualCall()</code>	The actions to be performed when a manual call ends. It is called in Jacada WorkSpace when a manual call ends.
<code>endOutboundBusinessFlow()</code>	The actions to be performed when an outbound flow ends. It is called in Jacada WorkSpace when ending a call in outbound mode.
<code>incomingBusinessCall()</code>	The actions to be performed when a business call is incoming (ringing). It is called in Jacada WorkSpace when receiving a CTI event indicating an incoming business call.
<code>incomingInternalCall()</code>	The actions to be performed when an internal call is incoming (ringing). It is called in Jacada WorkSpace when receiving a CTI event indicating an incoming internal call.
<code>internalCallBecomesBusiness()</code>	The actions to be performed when a transfer internal call becomes a business call. It is called in Jacada WorkSpace every time an internal call becomes a business call.
<code>outgoingBusinessCall()</code>	The actions to be performed when a business call is outgoing (dialing). It is called in Jacada WorkSpace when receiving a CTI event indicating an outgoing business call.

Method	Default Behavior
<code>outgoingInternalCall()</code>	The actions to be performed when an internal call is outgoing (dialing). It is called in Jacada WorkSpace when receiving a CTI event indicating an outgoing internal call.
<code>startBusinessCall()</code>	The actions to be performed when a business call starts. It is called in Jacada WorkSpace every time a business call starts.
<code>startDisposition()</code>	The actions to be performed when disposition stage starts. It is called in Jacada WorkSpace when receiving a CTI event indicating the Wrap-up status.
<code>startManualCall()</code>	The actions to be performed when a manual call is established. It is called in Jacada WorkSpace when a manual call was established.
<code>startOutboundFlow()</code>	The actions to be performed when an outbound flow starts. It is called in Jacada WorkSpace when clicking <i>Start</i> in outbound mode.

Note: The CTI methods `startInternalCall()` and `unloadScript()` cannot be overridden however, these methods can be extended by annotation. For details, see [Writing Manager Code](#).

Obtaining Information About Calls

Jacada WorkSpace provides an API for obtaining information related to the calls an agent engages in, and to a custom items the agent handles.

Call-related CTI information is maintained in an object named `PartyManager`. The `PartyManager` object contains information regarding the calls an agent engages in, including the ANI, DNIS and the number of calls. In addition, a `Party` object contains information regarding the parties that participate in calls: agent, supervisor, customer and calling-agent. This information

includes the call ID, the `dn` (extension), and attached data.

Custom-item-related CTI information is maintained in an object named `CustomItemPartyManager`. The `CustomItemPartyManager` object contains information regarding the custom items an agent handles, including the Genesys interaction ID, the number of calls, and attached data. This chapter: describes the `PartyManager` Interface.

Obtaining Call Information Using the PartyManager Interface

Jacada WorkSpace provides the `IPartyManagerInfo` interface for obtaining call information. The following table lists the functions provided by the `IPartyManagerInfo` interface:

Function	Description
<code>public IPartyInfo getAgentPartyInfo();</code>	Returns a <code>PartyInfo</code> object containing agent information.
<code>public String getANI();</code>	Returns the <code>ANI</code> (Automatic Number Identification) of the call.
<code>public boolean getBusy();</code>	Returns a flag that indicates the status of the <i>Ready/Busy</i> button.
<code>public IPartyInfo getCallingAgentInfo();</code>	Returns a <code>PartyInfo</code> object containing calling-agent information.
<code>public IPartyInfo getCustomerInfo();</code>	Returns a <code>PartyInfo</code> object containing customer information.
<code>public String getDNIS();</code>	Returns the <code>DNIS</code> (Dialed Number Identification Service) of the call.
<code>public int getNumberOfCalls();</code>	Returns the number of calls handled by the agent in the current session, including the current call if engaged in one.
<code>public IPartyInfo getSupervisorInfo();</code>	Returns a <code>PartyInfo</code> object containing supervisor information.

Function	Description
<pre>public String getManualModeCallID();</pre>	This function is relevant only in manual mode. Returns the number entered by the agent in the Jacada WorkSpace <i>ID</i> combo box.

Obtaining Party Information Using the PartyInfo Interface

Jacada WorkSpace provides the `IPartyInfo` interface for obtaining call information regarding a party in a call. Information can be obtained for the following parties: agent, calling-agent, customer and supervisor.

IPartyInfo Functions

The following table lists the functions provided by the `IPartyInfo` interface:

Function	Description
<pre>public Object getAttachedData();</pre>	Returns the attached data of the call for the specified party.
<pre>public String getCallId();</pre>	Returns the CTI call ID for the specified party.
<pre>public String getDn();</pre>	Returns the extension of the specified party.
<pre>public String getHuntGroup();</pre>	Returns the Hunt Group information for the specified party.

Function	Description
<code>public String getRole();</code>	Returns the role of the party, which is one of the following strings: ROLE_AGENT ROLE_CALLING_AGENT ROLE_CUSTOMER ROLE_SUPER
<code>public String getStatus();</code>	Returns the status of the party, which is one of the following strings: STATUS_INCALL STATUS_ROCKING - The agent state STATUS_HELD STATUS_RINGING STATUS_CALLING STATUS_CONNECTING STATUS_DISCONNECTED STATUS_TRANSFERRED STATUS_INCONFERENCE STATUS_ABOUT_TO_BE_CALLED STATUS_DEST_BUSY

Examples for Obtaining Call Information

Jacada Workspace provides a `VoiceOperationProxy` object, which returns an `IPartyManagerInfo` object. To access call information, you use the `getPartyManagerInfo` method.

The following example shows how to obtain the ANI of a call:

```
String ani = globalApp.getCTIManager().getVoiceOperationProxy().  
                getPartyManagerInfo().getANI();
```

The following example shows how to obtain call ID of the customer:

```
IPartyInfo customer = globalApp.getCTIManager().getVoiceOperationProxy().  
                getPartyManagerInfo().getCustomerInfo();  
String callId = customer.getCallId();
```

Attaching Data to a Call

For some CTI providers, it is possible to attach data to a call via CTI routing. For example, when calls are received from customers, data that is attached to them by CTI routing can be used to determine which script is displayed to the agent. Attached data is composed of a set of key value pairs.

Note: Attached data be attached to or retrieved from a call during the lifetime of that call only.

Jacada Workspace provides the functions for getting and setting attached data. To get attached data directly from the customer party:

```
IPartyInfo customer =  
globalApp.getCTIManager().getVoiceOperationProxy  
().getPartyManagerInfo().getCustomerInfo();  
Object data = customer.getAttachedData()
```

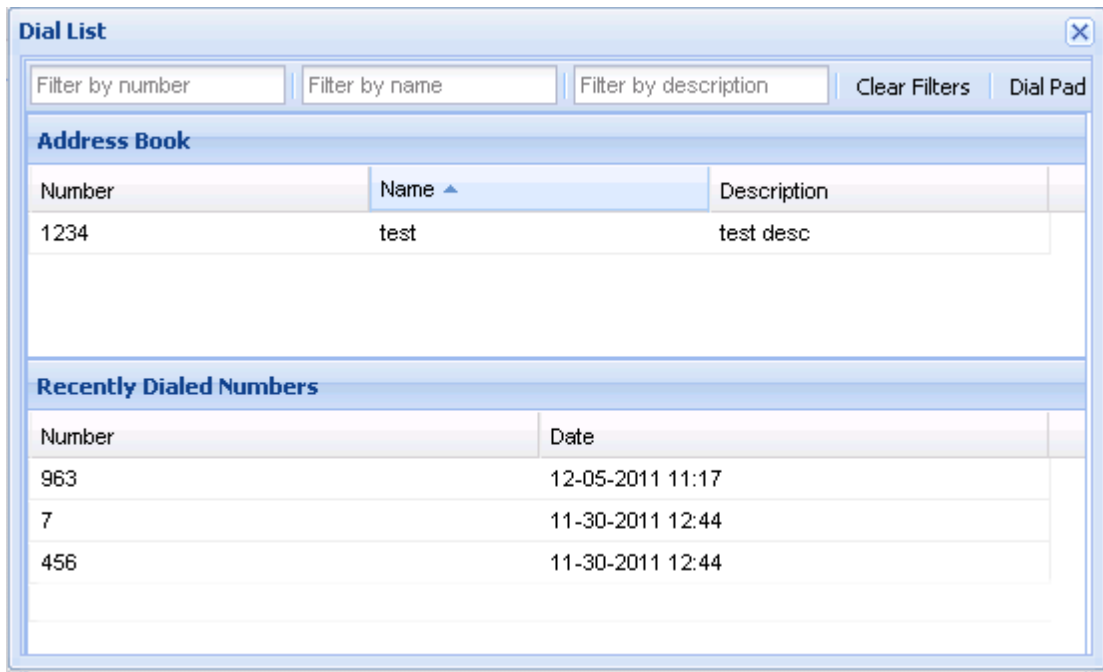
To get attached data based on the call id:

```
Object data = globalApp.getCTIManager  
().getVoiceOperationProxy().getAttachedData(Stri  
ng callId);
```

For CT Connect, EMC and AES the returned object is a string, while for Genesys the returned object is a map. For details of attaching data with Genesys, where attached data can also be set, see [Attaching Data to a Call or Custom Item](#).

Configuring the Dial List

To help agents easily locate and dial frequently used numbers, Jacada WorkSpace provides a *Dial List* pop-up. When is enabled, this pop-up appears each time an agent clicks *Dial*, *Transfer*, or *Conf* in the CTI toolbar.



The *Dial List* pop-up contains the following frames:

- **Address Book:** Displays a list of predefined numbers, together with the following optional parameters:
 - **Name:** An identifier for the number, e.g., a department name
 - **Description:** Additional notes/information about the number
- **Recently Dialed Numbers:** Displays a list of the numbers most recently called, together with the date and time that the call was made.

Double-clicking a row in either frame automatically dials the relevant number.

The bar at the top of the *Dial List* pop-up provides the following functionalities:

- **Filtering fields:** Agents can filter the Address Book according to number, name, and description, by entering the desired filter data in the appropriate fields. Jacada WorkSpace provides autocomplete functionality.

When filter values are entered in more than one field, filtering is performed with an **and** relationship between the filters.

Clicking *Clear Filters* removes filtering from the Address Book.

- **Dial Pad access:** Clicking the *Dial Pad* button opens the Dial Pad, which enables agents to call numbers that are not listed in the Address Book.

The Address Book is built from the *Dial List* page of the Jacada Management Console. For full details about this page, refer to the Jacada WorkSpace Administrator Guide, Using the Jacada Management Console (CTI section).



The screenshot displays the Jacada Management Console interface. The top header includes the Jacada logo and the title "Management Console". Below this, a breadcrumb trail shows "WorkSpace > Configuration > CTI > Dial List". The left sidebar contains a tree view of navigation options, with "Dial List" highlighted under the "CTI" section. The main content area features a table titled "Show table properties >>". The table has the following columns: Number (with a dropdown arrow), Transfer Type, Privilege, Name, Description, Use access code, Filter1, Filter2, and Action. A single row is displayed with the values: 1234, Standard, CTIBar, test, test desc, false, *, *, and Edit. Below the table are two buttons: "Add" and "Remove".

When the Address Book is updated via the Management Console, agents see the updated number list upon their next login to Jacada WorkSpace.

Note: In order for agents to be able to view the Address Book, they must be assigned the **CTIDialList** privilege.

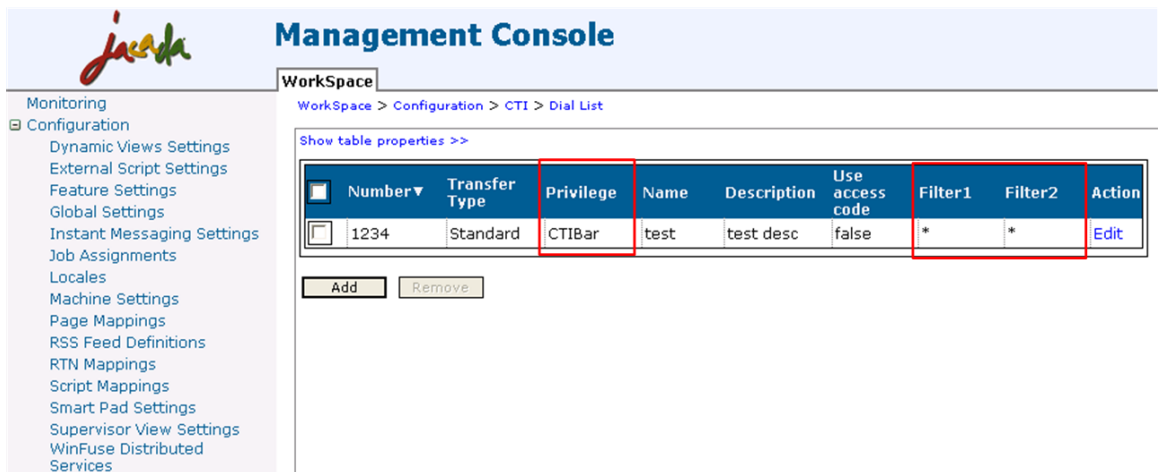
Managing Dial List Privileges and Filters

The number list that a given agent sees in the Address Book is determined by the following factors:

- The privilege and filters assigned to the number in the Jacada Management Console
- The filter inputs assigned to the relevant agent in the project code

Jacada Management Console Privileges and Filters

Each number listed on the *Dial List* page of the Jacada Management Console must have a privilege associated with it. In order to view that number in Jacada WorkSpace, agents must possess the relevant privilege.



The screenshot shows the Jacada Management Console interface. The sidebar on the left contains a tree view with categories like Monitoring, Configuration, and Dynamic Views. The main content area is titled 'Management Console' and 'WorkSpace'. It shows a breadcrumb trail: 'WorkSpace > Configuration > CTI > Dial List'. Below this, there's a table of dial list entries. The table has columns: Number, Transfer Type, Privilege, Name, Description, Use access code, Filter1, Filter2, and Action. One row is highlighted with a red box, showing the details for a number '1234'. The 'Privilege' column for this row is 'CTIBar', and the 'Filter1' and 'Filter2' columns are both set to '*'. The 'Action' column for this row has an 'Edit' link. Below the table, there are 'Add' and 'Remove' buttons.

Number	Transfer Type	Privilege	Name	Description	Use access code	Filter1	Filter2	Action
1234	Standard	CTIBar	test	test desc	false	*	*	Edit

In addition, a number can be assigned up to 2 filter types (*Filter 1* and *Filter 2*). For instance, Filter 1 can be a department name, and Filter 2 can be a location. Agents who have filter input(s) defined in their Agent Filter Input Lists are able to view only the numbers that have the corresponding filter(s) defined. (For details, refer to "Agent Filter Input List", below.)

Note: A number that has a value of * defined as both Filter 1 and Filter 2 can be viewed by all agents who have the relevant privilege.

Agent Filter Input List

In the project code, the list of filters to apply when loading the Dial List for the current agent is returned by the following method:

```
public List<String> getDialListFilterInputs(DialListFilterType filter)
```

The default implementation returns empty lists (i.e., agents are able to see all numbers for which they have the privilege to view). However, you can override this default behavior by assigning one or more filters to the Agent Filter Input Lists (Filter1 Input List and Filter2 Input List). When agents have input filters defined in their Filter Input Lists, the numbers that they can view in the Address Book are limited to those that have the corresponding filters.

- Preventing an agent from viewing numbers in the Dial List can be done using one of the following techniques:
- Denying the agent the privilege required to view the number

Inserting a single nonexistent value in the agent’s Filter Input List. For details, refer to [Overriding Default Agent Filter Inputs](#).

Dial List/Agent Filter Input List Example

In this scenario, 4 numbers with the following filters are defined in the database:

Number	Filter1	Filter2
111	US	Sales
222	US	*
333	*	Sales
444	*	*

The numbers that are displayed to the agents, based on each agent’s Input Filter List, are shown in the following table. It is assumed that all agents have the privilege to see all the numbers.

Agent	Filter1 Input List	Filter2 Input List	Displayed Numbers
1	{}	{}	{111, 222, 333, 444}
2	{US}	{}	{111, 222, 333, 444}

Agent	Filter1 Input List	Filter2 Input List	Displayed Numbers
3	{US, EMEA}	{}	{111, 222, 333, 444}
4	{US, EMEA}	{Sales}	{111, 222, 333, 444}
5	{Australia}	{}	{333, 444}
6	{Australia}	{Sales}	{333, 444}
7	{Australia}	{Marketing}	{444}

Overriding Default Agent Filter Inputs

The public `List<String> getDialListFilterInputs(DialListFilterType filter)` method, in the `CTIBehaviorHandler` class, returns the list of filters to apply when loading the Dial List for the current agent. The default implementation returns empty lists, meaning that the agent is able to see all numbers (assuming the relevant privilege is assigned).

You can override the default implementation by implementing the logic that returns a specific list of strings to a specific agent, according to the filter argument.

To override the method, add the following lines to the class

`USER.global.managers.CTIBehaviorHandler.`

```
@Override
public List<String> getDialListFilterInputs(DialListFilterType filter){
    //Implement here
}
```

If you want to filter the Dial List according to 2 filtering categories, be sure to implement both Filter 1 and Filter 2.

The following example shows how to implement filters for a department (Sales) and a location (US).

```

@Override
public List<String> getDialListFilterInputs(DialListFilterType filter){

    List<String> inputList = new ArrayList<String>();

    switch(filter){
    case FILTER_1:
        /* Implement logic for filter 1 */
        //Example:
        //inputList.add("Sales");
    case FILTER_2:
        /* Implement logic for filter 2 */
        //Example:
        //inputList.add("US");
    }
    return inputList;
}

```

Implementing Additional Filtering

If desired, you can further filter the Dial List after it has been retrieved from the database. This is done by implementing the following method in the `USER.global.managers.CTIBehaviorHandler` class:

```
public void filterDialListNumbers(List<DialListNumber> numbers)
```

Using this method, you can filter the final Dial List according to any logic.

The following example shows how to remove all numbers equal to 3002 from the Dial List.

```

@Override
public void filterDialListNumbers(List<DialListNumber> numbers) {

    Iterator<DialListNumber> iterator = numbers.iterator();
    while(iterator.hasNext()) {
        DialListNumber number = iterator.next();
        if(number.getTransferNumber().equalsIgnoreCase("3002")) {
            iterator.remove();
        }
    }
}

```

CTI Integration with External Applications

The Jacada CTI Gateway is used to handle CTI events related to voice interactions. Typically, all CTI events are passed by the CTI Gateway to Jacada WorkSpace. You can define that all events be handled by an external application, in addition to Jacada WorkSpace. An API is also provided, to enable the external application react to the CTI events; for example by ending a call according to attached data.

The following sections describe first how to pass events to an external system, and then the API provided for the external application.

Note: For details about filtering CTI events at the CTI Gateway level, so that specific events are not passed to Jacada WorkSpace, see [Filtering Genesys CTI Events](#).

Passing CTI Events to an External Application

To pass CTI events to an external application, in addition to passing the events to Jacada WorkSpace, in the `CTI_SETTINGS` table, you set the `CTIReportAllEventsToClient` to `true`. As a result, all information described in the class `CTIClientEventInfo.java` is pushed to the client side using `CTIClientEvent.js`.

The following table describes the information in the `CTIClientEventInfo.java` class. The `eventName` parameter always has a value, whereas the other parameters values are optional.

Parameter Name	Description
agentDn	The DN of the agent extension.
agentMode (Genesys only)	When the agent receives a <i>Ready</i> or <i>Not Ready</i> status event, the API handling the event includes a workmode parameter.
ani	The ANI (Automatic Number Identification) of the call.

Parameter Name	Description
attachedDataJsonString	A json representation of key-value pairs, currently used to describe attached data of the event. For more information, see Attaching Data to a Call .
callId	The ID of the primary call, as supplied by the switch.
callType	The type of call, as supplied by the switch.
channelType	The name of the channel type (<i>voice</i> , <i>customItem</i>).
data	For future use.
dn	The primary DN in the event.
dnis	The DNIS (Dial Number Identification Service) of the call.
errorMessage	The error message details, as supplied either by the CTI Gateway or the CTI switch. The format of the message is as follows: <pre><operation_name_in_workspace> operation failed .Error code=<error_code_information>:Error message= <error_message_information>:Error info=<additional_error_information>:</pre> For example, the following messages shows that the hold operation has failed, and provides additional information on the error: <pre>hold operation failed .Error code=1161:Error message=Incorrect object state:Error info=Unsupported operation:</pre>
eventName	The name of the CTI event.
eventReason	The reason for the event, as supplied by the switch.

Parameter Name	Description
eventStatus	The status of the event, as supplied by the switch.
externalId (Genesys only)	The ID, in the event, of the voice or custom item, as supplied by the switch.
paramsJsonString	A json representation of all the parameters from the original switch event.
secondCallId	The ID of the secondary call, as supplied by the switch. For example, a consult call.
secondDn	The secondary DN in the event.

CTI API for External Applications

Jacada WorkSpace provides an API, within the `CTIClient.js` file, for enabling an external application to communicate with the CTI provider via Jacada WorkSpace. This API includes methods such as `answerCallAPI`, `transferCallAPI`, `setAgentStatusAPI`. The operations can be executed by calling them in the following way:

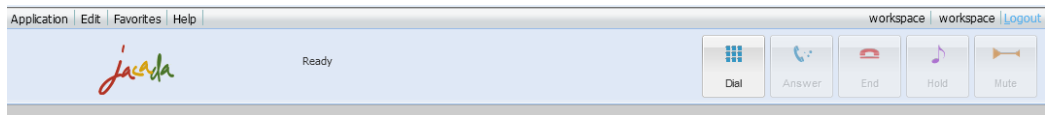
```
$W().cti.doSomething();
```

For example: `$W().cti.answerCallAPI();`

The list of methods and their parameters are in the `CTIClient.js` file between the comment referring to external applications containing the phrase "API for using the `CTIClient.js` from the outside".

Developing Avaya one-X Agent Specific CTI Functionality

When the one-X Agent program is integrated with Jacada WorkSpace, the Avaya one-X Agent is installed on the agent's desktop. CTI operations can be performed either from the WorkSpace CTI bar, or from the one-X agent client. The default WorkSpace CTI bar developed for use with Avaya integration is shown in the following figure.



The one-X Agent 2.5 API (also called the Event Notification API) enables WorkSpace to poll for one-X Agent notifications. This API is a XML/HTTP-based API in which clients make HTTP requests to one-X Agent and receive XML/HTTP responses. The events are not delivered by one-X Agent to the client, but are polled for by the client. Only a registered client can poll for events.

Note: The Agent WorkSpace URL must be added to the trusted site.

The `OneXAgentClientAPI` class, which is located in the **OneXAgentClientAPI.js** file, defines an interface between Avaya one-X Agent and Jacada WorkSpace. The `OneXAgentClient` class (located in the **OneXAgentAPI.js** file) contains the implementation. For more details about the project files, refer to [Avaya one-X Integration: Project Files](#).

The following sections describe the CTI functionality available to the developer when the CTI provider is Avaya one-X Agent:

- [Avaya one-X Integration: Project Files](#)
- [Request/Response-related Methods](#)
- [Notification-related Methods](#)

Avaya one-X Integration: Project Files

When using Avaya one-X Agent as the CTI provider, the following files are added to the Jacada WorkSpace project:

File	Description
onex.jsp	Specifies the relative project path of the file containing the CTI bar. For example: SYSTEM/portlets/CTI/providers/onexagent/onex.jsp
onexagentClientAPI.js	The interface that exposes methods for a jar script integration with Avaya one-X Agent.

File	Description
onexagentClient.js	Contains the default Jacada WorkSpace implementation of the APIs defined. The <code>onexagentClient</code> class is a wrapper for the Avaya one-X Agent REST services.
onexagentCTIBar.js	Defines the default Jacada WorkSpace CTI bar for the Avaya integration.

Request/Response-related Methods

The following sections describe APIs involving HTTP requests to/responses from Avaya one-X Agent.

Initiation and Registration APIs

`init`

This API specifies the port used by Avaya one-X Agent.

```
init : function (port)
```

`registerclient`

This API registers the client as a client of one-X Agent. Multiple clients are allowed to register with one-X Agent.

```
registerclient : function ()
```

`unregisterclient`

This API allows clients to signal that they will not be using the `registerclient` API anymore. Clients need to re-register in order to use the `registerclient` API again.

```
unregisterclient : function ()
```

Call-related APIs

makecall

This API allows a client to make a call using Avaya one-X Agent.

```
makecall : function (number)
```

This API can be used to enable click to dial functionality in a 3rd party application. The API works by making the call the way one would make a call using the one-X Agent UI. Any Agent Preferences settings related to prompting a dialog to confirm (or correct) the phone number before dialing or auto-holding an existing call are preserved.

The `makecall` API is asynchronous in that a successful response will be received by the client before the call is actually made. Any errors in making the call are reported via the one-X Agent UI.

mute

This API allows a client to mute the phone using one-X Agent. The API is asynchronous and works in a similar fashion to the `makecall` API.

```
mute : function ()
```

unmute

This API allows a client to unmute the phone using one-X Agent. The API is asynchronous and works in a similar fashion to the `makecall` API.

```
unmute : function ()
```

accept

This API allows a client to accept a voice interaction/call using one-X Agent. The API is asynchronous and works in a similar fashion to the `makecall` API.

```
accept : function (interactionid)
```

release

This API allows a client to release a voice interaction/call using one-X Agent. The API is asynchronous and works in a similar fashion to the `makecall` API.

```
release : function (interactionid)
```

hold

This API allows a client to hold a voice interaction/call using one-X Agent. The API is asynchronous and works in a similar fashion to the `makecall` API.

```
hold : function (interactionid)
```

unhold

This API allows a client to unhold a voice interaction/call using one-X Agent. The API is asynchronous and works in a similar fashion to the `makecall` API.

```
unhold : function (interactionid)
```

Notification-related Methods

The following sections describe APIs involving notifications from Avaya one-X Agent. The APIs that are implemented in Jacada WorkSpace are presented first.

WorkSpace-implemented APIs

onQueueEmpty

The `onQueueEmpty` notification is returned if the notification queue is empty.

```
onQueueEmpty : function (objectId,  
notificationId, timeStamp)
```

onVoiceInteractionCreated

The `onVoiceInteractionCreated` notification is added to the client's notification queue when a new `VoiceInteraction` is created by one-X Agent. This occurs both when a call is made from one-X Agent to another party, and when a call is received by one-X Agent.

```
onVoiceInteractionCreated : function  
(NotificationId, ObjectId, WorkItemId,  
RemoteUser, Topic, RemoteAddress,  
PromptedDigits, UI )
```

onVoiceInteractionMissed

The `onVoiceInteractionMissed` notification is added to the client's notification queue when a `VoiceInteraction(call)` is terminated by the remote party before the one-X Agent user could answer it.

```
onVoiceInteractionMissed :  
function(NotificationId, ObjectId, TimeStamp,  
WorkItemId, RemoteUser, Topic)
```

onVoiceInteractionTerminated

The `onVoiceInteractionTerminated` notification is added to the client's notification queue when a `VoiceInteraction` is properly terminated by one-X Agent. This happens when a call is hung up after it was answered.

```
onVoiceInteractionTerminated :  
function(NotificationId, ObjectId, WorkItemId,  
RemoteUser, Topic, RemoteAddress,  
PromptedDigits, UUI)
```

APIs Not Implemented in Workspace

onWorkItemAdded

The `onWorkItemAdded` notification is added to the client's notification queue when a new `WorkItem` is created by one-X Agent.

```
onWorkItemAdded : function (NotificationId,  
ObjectId, TimeStamp, Topic)
```

onWorkItemRemoved

The `onWorkItemRemoved` notification is added to the client's notification queue when a `WorkItem` is removed by one-X Agent.

```
onWorkItemRemoved : function (NotificationId,  
ObjectId, TimeStamp, Topic)
```

onIMInteractionCreated

The `onIMInteractionCreated` notification is added to the client's notification queue when a new `IMInteraction` is created by one-X Agent. This happens when an IM is either sent or received by the one-X Agent user.

```
onIMInteractionCreated :  
function(NotificationId, ObjectId, TimeStamp,  
WorkItemId, Topic)
```

onIMInteractionTerminated

The `onIMInteractionTerminated` notification is added to the client's notification queue when a `IMInteraction` is terminated. This happens when either the one-X Agent user or the remote party ends the IM session.

```
onIMInteractionTerminated :  
function(NotificationId, ObjectId, TimeStamp,  
WorkItemId, Topic)
```

Developing Cisco Specific CTI Functionality

The following sections describe the Cisco CTI functionality available to the developer:

- [Overriding Default WorkSpace CTI Behavior in the Cisco Environment](#)
- [Obtaining Call Information Using Attached Data](#)
- [Setting Up a List of Reasons for the Busy State \(Aux Codes\)](#)
- [Configuring Auto-Answer Functionality Per Call](#)
- [Configuring Logging for Cisco](#)
- [Supporting Multiple Sites](#)
- [Login Issues and Handling CTI Login Errors](#)
- [Working When CTI Connectivity is Lost](#)
- [Configuring Transfer and Consultation](#)
- [Configuring Cold Transfer](#)
- [Configuring Busy State Reasons and Display](#)

Overriding Default WorkSpace CTI Behavior in Cisco Environment

You can customize CTI behavior in Jacada WorkSpace, by overriding the default built-in functionality. For this purpose, the <project_name>\Java Resources\src \USER.global.managers.CTIBehaviorHandler.java class is provided.

The CTIBehaviorHandler class is empty, and within this class you can override various methods that control the CTI behavior in Jacada WorkSpace.

CTI methods that can be Overridden

The following table lists the CTI methods that can be overridden, and provides the default behavior and examples of possible additional implementations:

Method	Default Behavior	Customization Example
autoAnswerIncomingCall ()	Specifies whether to answer an incoming call automatically. The default, which is to automatically answer an incoming call, is derived from the following database settings in the CTI_SETTINGS table: <pre>CTICallAutoAnswer = True CTISwitchAutoAnswer = False</pre>	Use the call information, from the PartyManagerInfo object, to determine the auto-answer functionality per call.
getCallTerminationReason (String callId)	See setCallTerminationReason below.	
onCTIOperationFailed Handler(String operation, String message)	This method is called when a CTI exception is caught and handled. The default behavior is to print a log message.	Use this method to display a popup message to the agent with the failure details.

Method	Default Behavior	Customization Example
<code>setCallTerminationReason(String reason, String callId)</code> <code>getCallTerminationReason(String callId)</code>	The call termination reason is either “Call ended”, or an error message (when the call ended due to an error). The <code>set...</code> method is used internally to set the call termination reason upon the end of a call. The default <code>set...</code> behavior saves the reason string internally. The default <code>get...</code> behavior retrieves this reason string. Note: The <code>callId</code> parameter is for future use. It can remain empty.	Override these methods when you need to perform actions with the reason string. For example, customize the <code>set...</code> method to write the reason to a database, and then customize the <code>get...</code> method to obtain the reason from the database.

CTI methods that can be extended

The following table lists the CTI methods that can be extended:

Method	Default Behavior
<code>endManualCall()</code>	The actions to be performed when a manual call ends. It is called in Jacada WorkSpace when a manual call ends.
<code>incomingBusinessCall()</code>	The actions to be performed when a business call is incoming (ringing). It is called in Jacada WorkSpace when receiving a CTI event indicating an incoming business call.

Method	Default Behavior
<code>startBusinessCall()</code>	The actions to be performed when a business call starts. It is called in Jacada WorkSpace every time a business call starts.
<code>startDisposition()</code>	The actions to be performed when the disposition stage starts. It is called in Jacada WorkSpace when receiving a CTI event indicating the Wrap-up status.
<code>startManualCall()</code>	The actions to be performed when a manual call is established. It is called in Jacada WorkSpace when a manual call was established.

Obtaining Call Information Using Attached Data

This feature supports acquisition of call information by exposing Cisco's Expanded Call Context (ECC) variables and additional call context information. The information is stored in the attached data map, with pre-defined keys ([Additional Call Context Information](#)).

Attached data is stored internally in a map.

The following example shows how to obtain the attached data map from the Customer party:

```
IPartyInfo customer = globalApp.getCTIManager().getVoiceOperationProxy().
getPartyManagerInfo().getCustomerInfo();
Map attachedDataMap = (Map)customer.getAttachedData();
```

The entries in the map correspond to the call variables in the attached data. The keys for the data map are defined in the `com.jacada.jad.cti.constants.CTIConstants.CISCO_CONSTANTS` interface.

Standard Call Variables

Cisco exposes 10 standard call variables. These are stored in the attached data with the keys `CISCO_CONSTANTS.CTIOS_CALLVARIABLE1` .. `CISCO_CONSTANTS.CTIOS_CALLVARIABLE10`

To access Cisco standard call variables, use the following code:

```
String callVar1 = attachedDataMap.get(CISCO_CONSTANTS.CTIOS_CALLVARIABLE1);
String callVar2 = attachedDataMap.get(CISCO_CONSTANTS.CTIOS_CALLVARIABLE2);
```

Expanded Call Context (ECC) Variables

Cisco provides a customer-defined data layout for sending call context data with a call. This mechanism is called Expanded Call Context, or ECC.

ECC variables are stored in the attached data map, available through the **IPartyInfo** interface. ECC variables can be either scalar values (e.g., String, int, etc.), or arrays of scalar values.

The keys for the ECC variables are the same keys defined in the UCCE configuration. For example:

- Accessing a scalar ECC variable named "user.VariableName" is done using the name of the key in the attached data map:

```
String userVar = attachedDataMap.get("user.VariableName");
```

- Accessing a specific scalar variable, within an array called "user.ArrayName", is done using the string keyword "user.ArrayName[n]" where n is a zero based integer that notes the offset in the array:

```
String userArrVar0 = attachedDataMap.get("user.ArrayName[0]");  
String userArrVar1 = attachedDataMap.get("user.ArrayName[1]");
```

Additional Call Context Information

Additional call context information available through the attached data map is accessible using the keys listed in the following table.

Data	Description	Key
Call type		CISCO_CONSTANTS.CALLTYPE
CallWrapupData	Call-related wrap-up data	CISCO_CONSTANTS.WRAPUP_DATA
Caller Entered Digits in response to IVR prompting		CISCO_CONSTANTS.CED
DialedNumber	The number dialed	CISCO_CONSTANTS.DIALED_NUMBER
LineType	The line type	CISCO_CONSTANTS.LINETYPE

Data	Description	Key
UserToUserInfo	ISDN user-to-user information	CISCO_CONSTANTS.USER_TO_USER_IN FO

For example:

```
String lineType = attachedDataMap.get(CISCO_CONSTANTS.LINE_TYPE);
```

Setting Up a List of Reasons for the Busy State (Aux Codes)

The CTI bar includes a label to the left of the call-handling buttons that provides a user-friendly interpretation of the busy reason or Aux code. The label is displayed when the agent is in *Not Ready* mode, after the agent has clicked the *Busy* button. To see the label displayed for the busy reason, the agent must have the *BusyReasonCode* privilege.

You can set up the CTI system so that clicking the *Busy* button displays a list from which the agent needs to select the reason for not being ready to receive additional calls. If, when pressing *Busy*, the agent is in the process of handling a call, the *Not Ready* status takes effect only after the current call is completed.

Cisco supports specifying a set of reasons, each one associated with a reason code. By default, no reasons are defined, and clicking the *Busy* button uses the reason code '1', defined in the *CTIDefaultNotReadyReasonCode* in the database. When more than one reason is defined, clicking the *Busy* button displays the list of reasons for the agent to select from. If only one reason is defined, clicking *Busy* does not display the list.

The list of busy reasons is defined in the Jacada Management Console, or in the *login* method, as described in [Setting Up the List of Busy Reasons](#). When the list of reasons is defined both in the *login* method and in the Jacada Management Console, the *login* method reasons take precedence over the Jacada Management Console settings.

To set up the busy reason in the Jacada Management Console:

1. In the Jacada Management Console navigation menu, select the *Configuration > CTI > Busy Reason Codes* option.
The *Busy Reason Codes* page displays.
2. Click *Add*.
The *Edit CTI Busy Reason* page displays.
3. Enter the busy reason code, the busy reason description, and the busy reason display name.

Then, click **Save**.

4. Set the `CTIDefaultNotReadyReasonCode` in the `CTI_SETTINGS` table to the value you want displayed when the agent does not select a value from the list, or when no busy reason codes are defined.
5. In the Jacada Management Console navigation menu, select the *Authentication-Authorization > Groups* option.
6. The *Groups* page displays. Next to the group to which the agents are assigned, click *Edit*.

Note: The groups are mapped from the groups in the LDAP Active Directory.

The *Edit Groups* page displays, showing the details of the group, including the role assigned for the group.

7. In the Jacada Management Console navigation menu, select the *Authentication-Authorization > Roles* option.

The *Roles* page displays.

8. Next to the role to which the agent group is assigned, click *Edit*.
The *Edit Roles* page displays, showing the list of privileges assigned for the role.
9. Verify that the role is assigned the `BusyReasonCode` privilege, and then click **Save**.

Busy Reason Functionality in Jacada WorkSpace

When the agent logs on the default status is `Not Ready`. In this case, the default label defined in the `defaultBusyReason` field in the `CTI_SETTINGS` table is displayed.

When the agent is in `Ready` status and clicks the *Busy* button, a drop-down list is displayed with the list of busy reason codes defined in the database. The agent selects the relevant busy reason from the list. The busy reason is then displayed as a label to the left of the buttons.

Clicking the label displays the busy reasons drop-down list. Agents can change the displayed busy reason by selecting a different reason from the list.

Configuring Auto-Answer Functionality Per Call

In addition to the Jacada Management Console settings that configure the call answer mode in general, it is also possible to determine auto-answer functionality on a per-call basis. This is done by adding the following API:

```
Public boolean autoAnswerIncomingCall();  
to <project_name>\Java Resources\src\USER.global.managers.CTIBehaviorHandler.java.
```

This method is called when Jacada WorkSpace receives an incoming call. You can override the database settings, as necessary. For example, you can use the attached data in the session, or the `PartyManager` object, to determine whether to auto-answer a specific call.

The `PartyManager` object contains information regarding the current call, such as ANI, DNIS, call ID. This information can be retrieved using getter methods on the `PartyManager` object:

```
IPartyManagerInfo voiceManager =  
getVoiceOperationProxy().getPartyManagerInfo();
```

Note: When Jacada WorkSpace answers calls automatically, both ACD calls and direct incoming calls are answered.

For more details, refer to [Obtaining Information About Calls](#).

Configuring Logging for Cisco

An appender and logger are configured in `JacadaLogConfirmation.xml` for Cisco and all Cisco log messages are written to `Cisco.log`.

Note: In addition to the log4j configuration there is additional level of filtering - Cisco Log Mask. This mask is set by default to 7.

The CTI Gateway reads the `traceMask` property every 60 seconds, and updates the trace mask with any changes. A log message indicates any change in the mask, for example:

```
[INFO] Changing Cisco LogManager Trace mask to 9
```

To configure Cisco logging for Jacada WorkSpace:

1. In the application server directory, access the `JacadaConfiguration.properties` file. (The properties file is in the same location as the `JacadaLogConfirmation.xml` file.)
2. Set the trace mask level by adding a property to the properties file, as follows:

```
cti.cisco.traceMask=<int value>  
example:cti.cisco.traceMask=9
```

Note: The `cti.cisco.traceMask` property is not part of the default `JacadaConfiguration.properties` file.

Supporting Multiple Sites

When Jacada WorkSpace runs in several sites in a Cisco multi-site environment (where each site runs its own CTIOS process with one central UCCE controller managing all the sites), configure the Jacada WorkSpace CTI Gateway to connect to each CTIOS process on a per-agent connection basis.

To enable working in a multi-site environment:

1. In the `CTI_SETTINGS` table, set the `CtiMultiSiteSupport` property to `true`.
2. Define the connection parameters, `CTIServer`, `CTIPort`, `CTIBackupServer` and `CTIBackupPort` per agent in the `login` function.

Note: For a single-site environment, the `CtiMultiSiteSupport` field in the `CTI_SETTINGS` table is set to `false`, and the connection parameters, `sCTIServer`, `CTIPort`, `CTIBackupServer` and `CTIBackupPort`, are defined in the database. Multi-site environments ignore these database connection parameters.

In both single-site and multi-site environments, the `CTICiscoConnectionHeartBeat` value is taken from the database per CTI Gateway.

Use the following APIs in the `login` method of `src\USER\global\managers\ExtensionManager.java`:


```

@GlobalAction(actionName=GlobalActionNames.LOGIN,order=GlobalActionOrder.BEFORE)
public void login(HttpServletRequest request){
//Connection parameters (all Strings)
this.getGlobalApp().getCTIManager().getAgentInformation().setCtiServerHost("SiteA_IP");
this.getGlobalApp().getCTIManager().getAgentInformation().setCtiServerPort("SiteA_Port");
this.getGlobalApp().getCTIManager().getAgentInformation().setCtiBackupServerHost("SiteB_IP");
this.getGlobalApp().getCTIManager().getAgentInformation().setCtiBackupServerPort("SiteB_Port");

//setting Peripheral Gateway Id
this.getGlobalApp().getCTIManager().getAgentInformation().setCtiAgentPeripheralId(<PG_Id>);
}

```

where:

setCtiServerHost(SiteA_IP) – The IP address of the CTI server for the agent logging on.

setCtiServerPort(SiteA_Port) – The port of the CTI server for the agent logging on.

setCtiBackupServerHost(SiteB_IP) – The IP address of the CTI backup server for the agent logging on.

setCtiBackupServerPort(SiteB_Port) – The port of the CTI backup server for the agent logging on.

Thus, each agent is directed to the correct Cisco CTIOS in the multi-site environment.

Login Issues and Handling CTI Login Errors

By default, when an agent login to CTI fails, the agent is moved to manual mode and an error message is displayed, which remains on the agent's screen throughout the session.

If the agent is logged into CTI and attempts to re-login, Jacada WorkSpace is synchronized with the current CTI status:

- Not Ready
- Ready
- After Call Work
- In Call

However, if the agent attempts to log into CTI from a different extension, Jacada WorkSpace is synchronized with the current CTI status of the original extension, and a warning message is displayed so that the agent is aware that the login from the previous extension is still being used.

Note: If the agent is in a conference or any other type of multiple call, changing of the agent's status is not supported.

Customization for Cases of CTI Login Error

To customize the application flow when a CTI login error is encountered, use the following method in `CTIBehaviorHandler.java`:

```
@Override
public void on CTIOperationFailedHandler(String operation, String
message) throws CTIException {
    if (operation.equalsIgnoreCase(OPERATIONS.LOGIN)) {
        //handle login error
    }
}
```

Working When CTI Connectivity is Lost

When any Cisco component fails (PG, CTI server, CTIOS server, etc.) Cisco automatically moves the user to a temporary mode and then to the backup system.

If a Cisco component fails while an agent is logged in and is in one of the following states: Ready, Not Ready, In Call, In Hold, or In Wrap Up, the agent state changes to "Initializing....", and the CTI bar is disabled.

When the failed component is restarted, or when Cisco backup components become active, the agent is synchronized with the previous CTI state.

Note: Failover is not supported during consult/conference/warm transfer.

Configuring Transfer and Consultation

You can configure Cisco CTI to provide agents with one or more of the following transfer options:

Dialing Option	Define in:	How
Connect directly to a predefined number.	Script code	Using CTI functions, such as: • <code>transfer()</code> • <code>consultation()</code> For more information, see Disabling CTI Buttons .
Provide a dropdown list from which to select a number.	• <code><project_name>\Java Resources\src\USER.global.managers.ExtensionManager.java</code>, to initialize Jacada WorkSpace values. • You can also use the functions throughout a script, as necessary.	
Use the <i>Dial Pad</i> window, to specify the number to dial.	Provided with Jacada WorkSpace	Select the <i>Dial number</i> option from the drop-down list.

Configuring Cold Transfer

The following procedure explains how to configure a Cold Transfer Connect.

To configure Cold Transfer Connect:

- Implement the following method under `CTIBehaviorHandler.java`:

```

public String returnDataToAttach(String operation,
                                AbstractCTIState currentState, PartyManager
partyManager) {
    If(...Transfer Connect...){
        Return PARAMETERS.POST_ROUTE +
CTI_CONSTANT_CHARS.ATTACHED_DATA_DELIMETER + "true" +
CTI_CONSTANT_CHARS.ATTACHED_DATA_DELIMETER;
    }
}

```

The `returnDataToAttach` method is called every time a transfer occurs. It accepts:

- the operation type (`OPERATIONS.SINGLE_STEP_TRANSFER`)
- the current state machine state
- the current party manager

The method decides on Cold Transfer Connect or otherwise based according to the project level definition of the method. When it determines transfer connect, the method returns a string that indicates to the Gateway to use transfer connect.

Configuring Busy State Reasons and Display

You can set up the CTI system so that clicking the *Busy* button displays a list from which the agent needs to select the reason for not being ready to receive additional calls. If, when pressing *Busy*, the agent is in the process of handling a call, the *Not Ready* status takes effect only after the current call is completed.

Cisco supports specifying a set of reasons, each one associated with a reason code. By default, no reasons are defined, and clicking the *Busy* button uses the reason code '1', defined in the `CTIDefaultNotReadyReasonCode` in the database. When more than one reason is defined, clicking the *Busy* button displays the list of reasons for the agent to select from. If only one reason is defined, as a result of clicking *Busy* the list is not displayed.

Setting Up the List of Busy Reasons

The list of reasons is defined in Jacada Management Console, as described in [Setting Up a List of Reasons for the Busy State \(Aux Codes\)](#), or in the `login` method in the `<project_name>\Java Resources\src\USER.global.managers.ExtensionManager.java` file.

When the list of reasons is defined both in the `login` method and in the Jacada Management Console, the `login` method reasons take precedence over the Jacada Management Console settings.

To set up a list of ‘not ready’ reasons for the Busy button (login method):

- For each reason, add a line such as the following in the login method in `<project_name>\Java Resources\src\USER.global.managers.ExtensionManager.java`:
`globalApp.getCTIManager().getAgentInformation().addNotReadyCodesInfo(new CTIStateCodeInfo (info));`

where info is the following comma separated list:

Code – The associated reason code

Name – The reason for not being ready to receive calls.

Description – A description of the reason.

For example:

```
globalApp.getCTIManager().getAgentInformation().addNotReadyCodesInfo(new
CTIStateCodeInfo("Break","1","10 min"));
globalApp.getCTIManager().getAgentInformation().addNotReadyCodesInfo(new
CTIStateCodeInfo("Lunch","2","30 min"));
```

When more than one reason is defined, clicking the *Busy* button opens a list that displays the predefined information.

Configuring the Label Displayed for Busy Reason

The CTI bar includes a label to the left of the call-handling buttons that provides a user-friendly interpretation of the busy reason or Aux code. The label is displayed when the agent is in Not Ready mode, after the agent has clicked the *Busy* button. To see the label displayed for the busy reason, the agent must have the `BusyReasonCode` privilege.

To set up the busy reason label for display in Jacada Workspace:

1. In the Jacada Management Console navigation menu, select the *Configuration > CTI > Busy Reason Codes* option.

The *Busy Reason Codes* page displays.

2. Click *Add*.

The *Edit CTI Busy Reason* page displays.

3. Enter the busy reason code, the busy reason description, and the busy reason display name.

For example:

Show table properties >>

☐	Busy Reason Code	Busy Reason Description	Busy Reason Display Name	Action
☐	07	Official cigarette break	Break	Edit
☐	11	Weekly team meeting	Meeting	Edit

- Click **Save**.
- In the `CTI_SETTINGS` table, set the `CTIDefaultNotReadyReasonCode` to the value you want displayed when the agent does not select a value from the list or when no busy reason codes are defined.

Developing Genesys Specific CTI Functionality

Genesys supports a multi-site configuration, in which it is possible to work with multiple switches, located at multiple sites, that connect to multiple T-Servers. In order for a multi-site configuration to operate, the switches must share the Genesys Configuration Server.

Jacada WorkSpace provides support for a Genesys multi-site configuration. The Jacada CTI Gateway accesses the Genesys Configuration Server in order to determine to which switch (site) to transfer a call.

If there is a loss of connectivity to CTI, agents continue working in Jacada WorkSpace in manual mode until the connection to CTI is recovered and the T-Server is running and accessible. Jacada WorkSpace displays a *Reconnect Phone* button in the call-handling bar. To continue working with CTI again, the agent needs to click this button. However, it is not recommended to click the *Reconnect Phone* button while in the process of handling a call, as information may be lost as a result of the reconnection. After pressing the *Reconnect Phone* button, a disabled *Reconnecting...* button is displayed, until the CTI connection is recovered, and then the button disappears.

To enable the reconnect mechanism, the setting `CTIEnableReconnect` in the `CTI_SETTINGS` table must be set to true.

The following sections describe the CTI functionality available to the developer when the CTI provider is Genesys:

- [Setting Up a List of Reasons for the Busy State](#)
- [Retrieving the Agent Skill Set from Genesys](#)

- [Support for Transfer Connect](#)
- [Support for Custom Item](#)
- [Attaching Data to a Call or Custom Item](#)
- [CTI Integration with External Applications](#)
- [Filtering Genesys CTI Events](#)

Setting Up a List of Reasons for the Busy State

You can set up the CTI system so that clicking the *Busy* button displays a list from which the agent needs to select the reason for not being ready to receive additional calls. If, when pressing *Busy*, the agent is in the process of handling a call, the *Not Ready* status takes effect only after the current call is completed.

Genesys supports specifying a set of reasons, each one associated with a reason code. For example, in Alcatel, the reason code values are between '1' to '9'. By default, no reasons are defined, and clicking the *Busy* button uses the reason code '1', defined in the `CTIDefaultNotReadyReasonCode` in the database. When more than one reason is defined, clicking the *Busy* button displays the list of reasons for the agent to select from. If only one reason is defined, as a result of clicking *Busy* the list is not displayed.

The list of reasons is defined in database or in the login method in the `src/USER/global/managers/ExtensionManager.java` file. When the list of reasons is defined both in the login method and in the database, the login method reasons take precedence over the database settings.

To set up a list of 'not ready' reasons for the Busy button:

- For each reason, add a line such as the following in the login method in `src/USER/global/managers/ExtensionManager.java`:

```
globalApp.getCTIManager().getAgentInformation().addNotReadyCodesInfo(new CTIStateCodeInfo(info));
```

where info is the following comma separated list:

Code – The associated reason code

Name – The reason for not being ready to receive calls.

Description – A description of the reason.

For example:

```
globalApp.getCTIManager().getAgentInformation().addNotReadyCodesInfo(new
CTIStateCodeInfo("Break","1","10 min"));
globalApp.getCTIManager().getAgentInformation().addNotReadyCodesInfo(new
CTIStateCodeInfo("Lunch","2","30 min"));
```

As a result of defining more than one reason, clicking the *Busy* button opens a list that displays the predefined information.

Configuring the Label Display for Busy Reason

The CTI bar includes a label to the left of the call-handling buttons that provides a user-friendly interpretation of the busy reason or Aux code. The label is displayed when the agent is in Not Ready mode, after the agent has clicked the **Busy** button. To see the label displayed for the busy reason the agent must have the `BusyReasonCode` privilege.

To set up the busy reason label for display in Jacada WorkSpace:

1. In the Jacada Management Console navigation menu, select the *Configuration > CTI > Busy Reason Codes* option.

The *Busy Reason Codes* page displays.

2. In the page, click *Add*.

The *Edit CTI Busy Reason* page displays.

3. In the page, enter the busy reason code, the busy reason description and the busy reason display name and click *Save*.

For example:

[WorkSpace](#) > [Configuration](#) > [CTI](#) > [Busy Reason Codes](#)

[Show table properties >>](#)

☐	Busy Reason Code▼	Busy Reason Description	Busy Reason Display Name	Action
☐	07	Official cigarette break	Break	Edit
☐	11	Weekly team meeting	Meeting	Edit

Agents must have the `BusyReasonCode` privilege, in order to use this feature.

4. Set the `CTIDefaultNotReadyReasonCode` in the `CTI_SETTINGS` table to the value you want displayed when the agent does not select a value from the list or when no busy reason code are defined.

Using the Busy Reason in Jacada WorkSpace

When the agent logs on the default status is `Not Ready`. In this case the default label defined in

the `defaultBusyReason` field in the `CTI_SETTINGS` table is displayed.

When the agent is in `Ready` status and clicks the *Busy* button a drop-down list is displayed with the list of busy reason codes defined in the database. The agent selects the relevant busy reason from the list which is then displayed as a label to the left of the buttons.

The agent can change the label by clicking on it. The same drop-down list displayed when the *Busy* button is clicked is displayed and the agent can select a different reason to display.

Retrieving the Agent Skill Set from Genesys

An agent skill is a pair of name and level, that is defined in the Genesys Configuration Manager. In the Genesys Configuration Manager, an agent can be assigned a set of skills. The skill set information is retrieved by the Jacada CTI Gateway from the Genesys Configuration Manager, at agent login, and saved by Jacada WorkSpace.

Jacada WorkSpace provides the following method, in the `AgentInformation` class, to enable retrieving an agent's skill set:

```
public Map getSkillSet()
```

This method returns a map of key-value pairs that map a Genesys *SkillName* to a Genesys *SkillLevel*.

The following example shows how to retrieve the skill set of an agent:

```
Map skillset =  
    getGlobalApp().getCTIManager().getAgentInformation().getSkillSet();
```

Identifying the Agent Skill Set from the Log

Upon retrieval of an agent's skill set at login, the pairs of *SkillName-SkillLevel* are also written to the `CTI.log` (at `Info` level), in the following format:

```
Agent <agentName> has <numberOfSkills> skills:  
Skill: <skillName>, Level: <skillLevel>  
Skill: <skillName>, Level: <skillLevel>  
Skill: <skillName>, Level: <skillLevel>
```

Support for Transfer Connect

Transfer connect enables a transfer to be implemented by sending Dual Tone Multiple Frequencies (DTMF) tones to the switch, instead of using the transfer API.

For further details on transferring calls, refer to [Setting Up Transfer and Consultation](#).

Setting Up Transfer Connect

Configuring transfer connect involves enabling the transfer connect option. Then, if desired, you can set up transfer connect destination numbers in the Dial List.

To enable the transfer connect option:

1. Set `CTIHandshakeTransferConnectDTMFCode` and `CTIWarmTransferConnectDTMFCode` in `CTI_SETTINGS` table to the relevant DTMF code for the transfer type.
2. Assign the agent the `CTITransferConnect` privilege.

To define transfer connect destinations using a Dial List:

1. In the Jacada Management Console navigation menu, select *Configuration > CTI > Dial List*.
The *Dial List* page appears.
2. Click *Add*.
The *Edit Dial List Number* page appears.
3. Define the required parameters, and then click *Save*.
For a description of parameters, refer to the *Jacada WorkSpace Administrator Guide*.

Using Transfer Connect in Jacada WorkSpace

When the agent clicks the *Transfer* button in the CTI bar, the *Standard* and *Connect* buttons are displayed. The agent can perform a standard transfer by clicking the *Standard* button followed by the transfer type: Cold, Handshake, Warm. The agent can perform a Transfer Connect call by clicking the *Connect* button and then clicking the transfer type: Handshake, Warm. If a Dial List is defined, a window opens showing the possible destinations from the database. If a Dial List is not defined, the dial pad is displayed, and the agent enters the number to connect to, without the DTMF code.

Support for Custom Item

The Jacada WorkSpace CTI provides support for a multi-channel environment, in which an agent handles, in addition to voice calls, items such as faxes or scanned documents. These items are referred to as custom items.

Note: Custom items are referred to in Genesys as Open Media items.

The following sections describe:

- [Handling Custom Items in Jacada WorkSpace](#)
- [Configuring Jacada WorkSpace to Handle Custom Items](#)
- [Configuring the Answer Mode for Custom Items](#)
- [Custom Item Handling API](#)
- [Assigning Custom Items to Agent by Skillset](#)
- [Obtaining Custom Item Information Using the CustomItemPartyManager Interface](#)

Handling Custom Items in Jacada WorkSpace

In Jacada WorkSpace, the header bar displays icons that represent the channels the agent is logged in to: only voice, or voice and custom item. The state of each icon, enabled or disabled, indicates the state of the channel, ready or busy (respectively). In the following example, both the voice (phone icon) and custom item (fax icon) channels are ready:



The available item-handling buttons are *Answer*, *Dial*, *Ready/Busy* and *End*. The *Answer* button enables the agent to start working with a custom item, when Jacada WorkSpace is not configured to automatically answer custom items. The *Dial* button enables the agent to initiate an outbound call, as described in [Custom Item Handling API](#), while handling a custom item. The *Ready/Busy* buttons enable the agent to indicate whether additional call or custom items can be accepted (*Ready*) or not (*Busy*). The *End* button enables the agent to end the handling of the custom item in Jacada WorkSpace.

Note: The agent cannot accept inbound calls while handling a custom item.

Handling custom items in Jacada WorkSpace involves receiving the item from Genesys, handling the item, and informing Genesys that the handling is done. An external application is required to view the custom item. For example, when the custom item is a fax, the external application can be a fax viewer that displays the fax information and enables handling it. Configuring Jacada WorkSpace to use an external application is done using a custom item handling API, described in

Configuring Jacada WorkSpace to Handle Custom Items

When the CTI provider is Genesys, Jacada WorkSpace provides support for handling both voice calls and Genesys Open Media items, which are referred to as custom items. Genesys Open Media provides support for various types of interactions, such as faxes, scanned documents, and more.

When Jacada WorkSpace is configured to handle both calls and custom items, the agent can handle only one of these at a time. The only exception is the ability to initiate an outbound call while handling a custom item.

Note: Jacada WorkSpace cannot be configured to handle custom items only. Also, a script is not displayed when handling a custom item.

Before configuring Jacada WorkSpace to handle custom items, verify that the Genesys environment is configured to support Open Media items:

- You have a running Open Media server, that supplies Genesys with Open Media items (such as faxes).
- The Open Media Type is defined in *CME > Business Attribute > Media Type*
- There is a skill that represents the ability to handle Open Media items. This skill must be defined in *CME > Skills*, and must be assigned to the agents (Persons) that need to handle Open Media items.
- There is routing that directs incoming work to the agents (Persons).
- All the agents (defined in Open Media as *Persons*) that have the skill to handle Open Media items have their *Capacity Rule* defined as 1, meaning that they can only receive one call or one custom item at a time.

Configuring the Database to Handle Custom Items

You configure Jacada WorkSpace to handle custom items in the `CTI_SETTINGS` table in the database. To work with custom items, the following settings are required:

DB Setting	Value
CTIEnableCustomItems	True

DB Setting	Value
CTICustomItemNameList	A comma-separated list of the names of the Genesys Open Media types, as defined in Genesys CME. For example: <code>WorkItem, Fax</code>
CTICustomItemSkillName	The name of the Genesys skill that the agent must have in order to handle the Open Media types defined in <code>CTICustomItemNameList</code>.

Configuring the Answer Mode for Custom Items

A custom item can either be auto-answered by Jacada Workspace without a prompt, or else the agent receives an indication when a new call or custom item arrives and needs to accept the item by clicking the *Answer* button.

Configuring the answer mode for custom items can be done either in the database, or at the code level. When the configuration for all agents is the same, use the database. In projects that require more complex configuration, use the code level.

Note: The code level overrides the database setting. When the answer mode is not configured, neither in the database, nor in the code, the default is true, meaning that custom items are automatically answered.

To configure the answer mode for custom items in the database:

- In the `CTI_SETTINGS` table, use the following setting: `CTICustomItemAutoAnswer`.
Set the value of this setting to `true` or `false` in the relevant profile (such as `CTI_GENESYS`).

To configure the answer mode for custom items at the code level:

- Add the following method to `CTIBehaviorHandler.java`:

```
@Override
public boolean customItemAutoAnswer() {
    // TODO Auto-generated method stub
    return true;
}
```

You add this method in `src/USER/global/managers/CTIBehaviorHandler.java`.

Custom Item Handling API

Jacada WorkSpace provides an API for handling custom items, that enables sending notifications to and receiving notifications from an external application.

Sending Notifications to the External Application

Jacada WorkSpace provides a set of methods for sending notifications regarding a custom item from Jacada WorkSpace to an external application. The CTI events that arrive from Genesys, activate methods that correspond to each event.

These methods are defined in the `CustomItemHandler` class.

The following methods are provided:

Method	Description
<code>public void init()</code>	This method is called at login to Jacada WorkSpace, when the Agent object is initialized. Use this method to perform any initialization tasks. For example, setting the external application for viewing custom items, such as a fax viewer.
<code>public void incomingCustomItem()</code>	This method is called when a new custom item has arrived and is waiting to be answered. Use this method to perform tasks before the handling of the custom item begins.

Method	Description
<pre>public void startCustomItem()</pre>	This method is called when the handling of a custom item begins. Use this method to handle custom items. For example, open a fax in a fax viewer.
<pre>public void customItemDone()</pre>	This method is called when the custom item is marked as done in Genesys. Use this method to perform tasks for ending the handling of an item. For example, close the fax in the fax viewer.

Accessing Custom Item Data

The interaction ID and the open media item ID (custom item ID) is accessible via a custom item party manager object.

For example:

```
public void incomingCustomItem(){
    ICustomItemPartyManagerInfo dataMgr =
getOpenMediaOperationProxy().getPartyManagerInfo();
    System.out.println("Interaction Id = " + dataMgr.getInteractionId());
                                                                    //print Genesys interaction id
    System.out.println("Custom Item Id = " + dataMgr.getCustomItemId());
                                                                    //print Genesys External id
}
```

Receiving Notifications From an External Application

Jacada Workspace needs to receive an indication when the agent has started handling the custom item, and when the agent has finished handling the custom item, in order to update Genesys, as well as the Jacada Workspace user interface (status and buttons).

To indicate the start or end of the handling of a custom item, the Jacada Workspace *Answer* or *End* buttons can be pressed (respectively). However, instead of pressing a button in Jacada Workspace, Jacada Workspace also provides an API, that enables the external application to inform Jacada Workspace that the handling of a custom item has begun or has ended.

The Jacada Workspace API is provided in the following formats:

- As Java methods – used in server-side code, using the `CTIHelper` class

- As JavaScript functions – used in client-side code

Note: Only a single indication that the handling of a custom item has begun or ended is supported. That is, if Java methods or JavaScripts are used, the *Answer* and *End* buttons must be disabled. Also, if some of the code is on the server-side, using the Java methods, and some on the client-side, using JavaScripts, they cannot handle the same event.

The following table lists the API for an external application to send information to Jacada WorkSpace regarding a custom item:

Java Method	JavaScript Function	Description
<pre>public void acceptCustomItem (String string)</pre> where string is any string. Example: <pre>CTIHelper.getHelper (this.getSession()). acceptCustomItem ("abc");</pre>	<pre>acceptCustomItemAPI:function ()</pre> Example: <pre>\$W().cti.acceptCustomItemAPI();</pre>	Called by the external application, to indicate to Jacada WorkSpace that the agent has accepted the waiting item. Use this method when the agent does not use the WorkSpace <i>Answer</i> button to accept the custom item.

Java Method	JavaScript Function	Description
<pre>public void releaseCustomItem (String string)</pre> where string is any string. Example: <pre>CTIHelper.getHelper (this.getSession()). releaseCustomItem ("abc");</pre>	<pre>releaseCustomItemAPI:function()</pre> Example: <pre>\$W().cti.releaseCustomItemAPI();</pre>	Called by the external application, to indicate that the agent has finished handling the custom item. Use this method when the agent does not use the Workspace <i>End</i> button to end the handling of the custom item.

Initiating a Call While Handling a Custom Item

The agent can make an outbound call while working on a custom item, such as a fax. When in a call while handling a custom item, the agent has the option to either end the processing of the custom item before ending the call, or end the call before ending the custom item, by clicking the appropriate CTI button.

To initiate the outbound call, the agent uses the *Dial* button. After dialing, the agent can use the *Hold/Off Hold* button. It is not possible to use the *Ready/Busy* button.

When engaged in an outbound call while handling a custom item, the following applies:

- Clicking the *End Call* button ends the call and clicking the *End Item* button ends the custom item.
- No script is loaded.

Note: Scripts cannot be displayed when handling a custom item.

- The information in the Jacada Workspace item-handling bar reflects the handling of the custom item, including the counters and the ID.

In the *Handling Item Wrap Up* status, for business calls:

- If the agent ends the call, call disposition starts and continues while the agent processes the item.
- If the agent ends the processing of the custom item, the call flow begins and call timing counters start.

Note: When handling a custom item, and the agent receives a call, it is considered as a direct-dial call, but the call information is not reflected in Jacada WorkSpace.

Setting Jacada WorkSpace to Ready/Busy for Custom Items Only

By default, the *Ready/Busy* buttons set both the voice and custom item channels to ready or busy.

Jacada WorkSpace provides the following API for setting the status of Jacada WorkSpace to ready or busy for custom items:

Method	Description
<pre>public String readyMultimedia (ArrayList mediaNames) throws CTIException</pre>	Sets to ready the custom item (Open Media) channel. The <code>mediaNames</code> parameter is an <code>ArrayList</code> of <code>Strings</code> of custom item names.
<pre>public String notReadyMultimedia (ArrayList mediaNames) throws CTIException</pre>	Sets to busy (not ready) the custom item (Open Media) channel. The <code>mediaNames</code> parameter is an <code>ArrayList</code> of <code>Strings</code> of custom item names.

Assigning Custom Items to Agent by Skillset

By default, in order for an agent to be able to work with custom items, the custom item names that can be worked with, and the skill definition for the agent, must be defined in the `CTI_SETTINGS` table, in the `CTICustomItemNameList` and `CTICustomItemSkill` fields respectively. Also, the value of `CTICustomItemSkill` must match the agent skill set defined in Genesys.

You can use a different source of information, other than the agent skill defined in Genesys, in order to determine the custom items an agent handles. For example, this information may be retrieved

from an external database, or an external application.

To enable overriding the skill set defined in Genesys, Jacada WorkSpace add the following method, that enables building a new or updating an existing list of custom items to handle to `src/USER/global/managers/CTIBehaviorHandler.java`:

```
public void updateAgentOpenMediaTypes(IOpenMediaInformation openmediaInfo) {
    ArrayList<String> arr = new ArrayList<String>();
    arr.add("myMedia");
    this.getOpenMediaInformation().setOpenMediaTypesArray(arr);
}
```

This method is invoked after the list was updated according to the database, so that you can either update the list or create a new one.

Jacada WorkSpace uses a `MediaMap` object to map between `CHANNEL_TYPES` and an `ArrayList` of Strings of channel type names. In this case, the `CHANNEL_TYPE` is `CUSTOM_ITEM`. The functions `setMediaMap` and `getMediaMap` are provided to enable setting and getting the `MediaMap` object.

Obtaining Custom Item Info Using CustomItemPartyManager Interface

Jacada WorkSpace provides the `ICustomItemPartyManagerInfo` interface for obtaining custom item information. The following table lists the functions provided by the `ICustomItemPartyManagerInfo` interface:

Function	Description
<code>public Object getAttachedData();</code>	Returns an object containing the attached data.
<code>public boolean getBusy();</code>	Returns a flag that indicates the status of the <i>Ready/Busy</i> button.
<code>public String getCustomItemId();</code>	For future use.
<code>public String getInteractionId();</code>	Returns the Genesys interaction ID.
<code>public int getNumberOfCustomItems();</code>	Returns the number of custom items handled by the agent in the current session, including the current item.

Function	Description
<code>public boolean hasInteractionId();</code>	Checks if the Genesys interaction ID is null. Returns true if not null, and false if null.

Jacada WorkSpace provides an `OpenMediaOperationProxy` object, which returns an `ICustomItemPartyManagerInfo` object. To access custom item information, you use the `getPartyManagerInfo` method.

Attaching Data to a Call or Custom Item

In addition to using attached data retrieved from the Genesys routing, as described in [Attaching Data to a Call](#), it is possible to attach data in a Jacada WorkSpace call-handling script. For example, you can attach information within a Jacada WorkSpace call-handling script, so that upon transferring the call, the call information can be retrieved by the party receiving the call.

When working with a custom item, attached data can be used in the handler methods, for example, to determine which external application should handle the custom item. For more information on custom items, see [Personalizing What the Agent Can See and Do](#).

Note: Only during the lifetime of a call or a custom item, can attached data be either attached to or retrieved from the call or the custom item.

Attached data is composed of a set of key value pairs. Jacada WorkSpace provides functions for attaching data and retrieving attached data for voice and for custom items. For voice `globalApp.getCTIManager().getVoiceOperationProxy()` is used. For custom items `globalApp.getCTIManager().getOpenMediaOperationProxy()` is used. The following code examples show the use of the voice functions.

To get attached data directly from the customer party:

```
IPartyInfo customer = globalApp.getCTIManager().
    getVoiceOperationProxy().getPartyManagerInfo().getCustomerInfo();
Object data = (Map) customer.getAttachedData()
```

Note that the returned object is a map.

To get attached data based on the call id:

```
Object data = globalApp.getCTIManager ().  
    getVoiceOperationProxy().getAttachedData(String callId);
```

To set attached data:

```
globalApp.getCTIManager ().getVoiceOperationProxy().attachData(String key, String  
value);
```

where:

key – The key of the attached data. When the parameter is not supplied, with Genesys, when getting attached data, the attached data for all keys for the callId is returned.

value – The value of the attached data.

CallId – The call id.

CTI Integration with External Applications

The Jacada CTI Gateway is used to handle CTI events related to voice and custom item interactions.

Handling events related to custom item interactions is the same as for voice interactions. For details, see [CTI Integration with External Applications](#).

For details about filtering CTI events at the CTI Gateway level, so that specific events are not passed to Jacada WorkSpace, see [Filtering Genesys CTI Events](#).

Filtering Genesys CTI Events

The Jacada CTI Gateway is used to handle CTI events related to voice and custom item interactions. Typically, all CTI events are passed by the CTI Gateway to Jacada WorkSpace. However, it is possible to define a CTI event filter that prevents the CTI Gateway from sending a specific event to Jacada WorkSpace. This can be useful, for example, when you want specific events to be handled by an external system, instead of Jacada WorkSpace. The filtering mechanism can rely on information such as attached data. For more information, see [Attaching Data to a Call or Custom Item](#).

To implement a filter, you need to create a class that implements the following interface:

```
com.jacada.ctigateway.filter.CTIEventFilter
```

This interface includes the method:

```
public boolean isEventPassedToSession(Map params);
```

where the `Map` parameter contains the Genesys `InteractionEvent` object.

You use the `isEventPassedToSession` method to return a value indicating whether the CTI event should be passed to Jacada WorkSpace.

The following example shows the code required in order to receive the reference to the `InteractionEvent` object to use:

```
public boolean isEventPassedToSession(Map params) {
    InteractionEvent event =
        params.get(CTIConstants.PARAMETERS.EXTERNAL_FILTER_EVENT_PARAM_KEY);
    ...
}
```

In addition to creating the class, you also need to specify the name of the class in the `CTI_SETTINGS` table using one of the following keys:

CTIVoiceFilterClass – For filtering voice items

CTICustomItemFilterClass – For filtering custom items

The value of this setting must be the fully qualified name of the class. For example:
`com.jacada.external.CustomItemFilter`

To implement a CTI event filter:

1. In the IDE, create a new *Java Project*.
2. Unzip the following file:
`<InstallDir>/Management/CTIGateway.ear`
and extract its contents to a folder, such as `c:\CTIGateway`
3. In the IDE, add the following JAR files to the `CLASSPATH` of the project:
 - `AIL.jar`
 - `CTICommon.jar`
 - `JADCTI.jar`
4. In the project, create a class that implements the `CTIEventFilter` interface.

Note: The method `isEventPassedToSession` may be called multiple times during a single interaction. Therefore, when implementing this method, to avoid impact on performance, efficient code should be used.

5. Build the project. At the end of a successful build, a JAR file, `<project>.jar`, is created for the project. Note the path to the JAR file, listed towards the end of the *Build* messages.
6. Copy the `<project>.jar` file to the folder that contains the extracted JAR files, for example: `c:\CTIGateway\APP-INF\lib`

7. Select all the folders under the extraction folder (for example: `c:\CTIGateway`) and add them to a zip file named `CTIGateway.ear`.
8. Copy the newly generated `CTIGateway.ear` back under `<InstallDir>\Management`, overwriting the existing file.
9. Specify the fully qualified name of the class in the `CTI_SETTINGS` table, using the `CTIVoiceFilterClass` or the `CTICustomItemFilterClass` setting, for a voice or custom item filter (respectively).
10. Redeploy the `CTIGateway.ear`.

Using Attached Data to Filter Voice Items

Jacada WorkSpace provides an implementation for using attached data to filter voice items in the following class: `com.jacada.cti.Genesys.filter.GenesysVoiceFilter`. To filter voice items using this class, you need to specify the names and the values of the attached data keys to be used for filtering. A voice item whose attached values match the specified values, is filtered, meaning that it is **not** passed by the CTI Gateway to Jacada WorkSpace.

Specify the names and values of the attached data keys to use for filtering in the `CTI_SETTINGS` table as follows:

CTINumberOfAttachedDataKeys – The number of attached data keys for filtering voice items.

CTIAttachedDataKey<#> – The names of the attached data keys to use for filtering voice items. For each attached data key, you need an entry in the database, where the `<#>` is a number that begins at 1, and increments by 1 up to the total number of attached data keys used for the filtering. For example, if the total number of attached data keys to use for filtering voice items is 3, then the database must contain values for the following settings: `CTIAttachedDataKey1`, `CTIAttachedDataKey2`, `CTIAttachedDataKey3`.

CTIAttachedDataValue<#> – The values of the attached data keys for filtering voice items, which must correspond to the names specified in the `CTIAttachedDataKey<#>` settings.

The following example shows the database setting to filter voice items according to attached data:

ACTIVE	PROJECT	PROFILE	KEY	VALUE
true	*	CTI_GENESYS	CTIVoiceFilterClass	com.jacada.cti.Genesys.filter.GenesysVoiceFilter

ACTIVE	PROJECT	PROFILE	KEY	VALUE
true	*	CTI_GENESYS	CTINumberOfAttachedDataKeys	2
true	*	CTI_GENESYS	CTIAttachedDataKey1	Shipping
true	*	CTI_GENESYS	CTIAttachedDataValue1	Abroad
true	*	CTI_GENESYS	CTIAttachedDataKey2	ProductName
true	*	CTI_GENESYS	CTIAttachedDataValue2	SpecialOffer

For more information on attaching data, see [Attaching Data to a Call or Custom Item](#).

Working With the Supervisor View

The following sections describe how to work with the Supervisor View in Jacada WorkSpace:

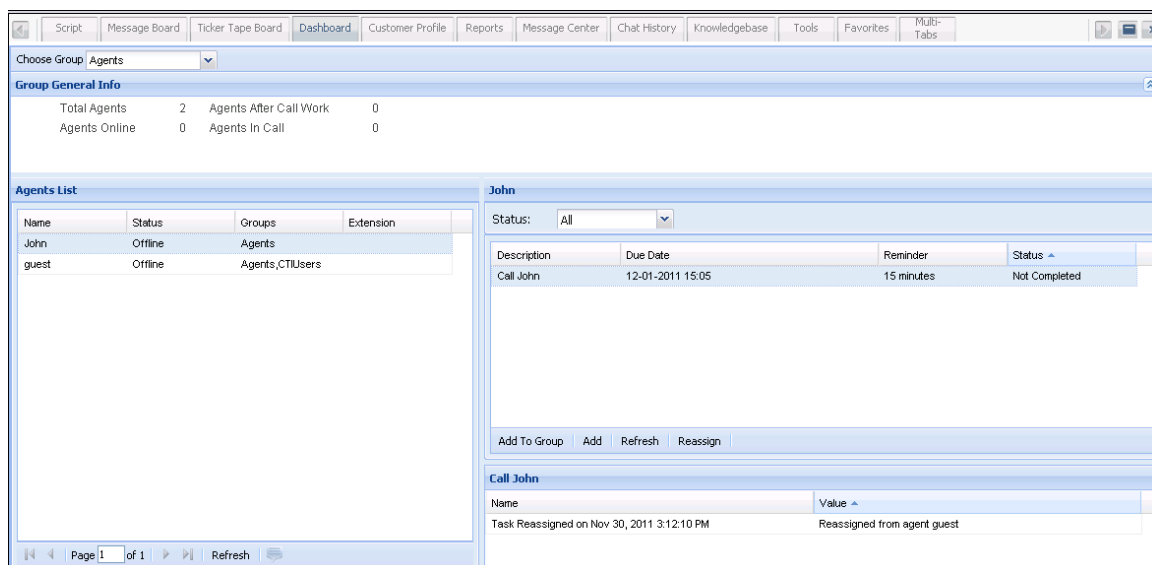
- [The Dashboard Tab](#)
- [Using the Agent Task Manager](#)
- [Customizing the Supervisor View](#)

The Dashboard Tab

The *Dashboard* tab enables call center supervisors to view the current work status of their agents, chat with agents, and manage the tasks of groups of agents under their supervision.

The *Dashboard* tab is accessible to all users who have the privilege to view it. The tab displays information about agents according to defined Jacada WorkSpace groups. The *Choose Group* drop-down list, at the upper left corner of the tab, allows you to select the group of agents to display. The list displays only the groups for which you are the defined supervisor.

Note: Groups, roles, and privileges are defined in the Jacada Management Console. For details, refer to the Jacada WorkSpace Administrator Guide, Using the Jacada Management Console (Authentication and Authorization).



Dashboard Tab Frames

The *Dashboard* tab consists of the following frames:

- **Group General Info:** Provides a snapshot view of agent status, including:
 - Total number of agents (in the selected group)
 - Number of agents online
 - Number of agents in call status (In Call)
 - Number of agents in disposition status (After Call Work)
- **Agents List:** Provides more detailed information about each agent in the group, including:
 - **Name:** User name of the agent
 - **Status:** Work status of the agent (Offline, Online, In Call, After Call Work)
 - **Groups:** All Jacada WorkSpace groups to which the agent is assigned
 - **Extension:** Telephone extension from which the agent is working (relevant only to agents connected to CTI)

Selecting the row of an agent and clicking the *Chat* icon (at the bottom of the *Agents List* frame) opens the *Chat* dialog, allowing the supervisor to initiate a chat with the agent.

Note: The Chat option is functional only when the agent is online, and when the Chat feature is supported.

- **Agent Task:** Opens the Task Manager of the agent selected in the *Agents List* frame, allowing you to view and manage agent tasks. For details, refer to [Using the Agent Task Manager](#).

Using the Agent Task Manager

The Agent Task Manager lists the tasks of the agent selected in the Agents List frame, and enables you to manage these tasks. The Task Manager grid provides the following information about each task:

- **Description:** Name of task
- **Due Date:** Date and time at which the task needs to be done
- **Reminder:** The timeframe (before the due date/time of the task) at which the agent gets a reminder about the task.
- **Status:** Current status of the task (*Completed* or *Not Completed*)

John

Status: All

Description	Due Date	Reminder	Status ▲
Call John	12-01-2011 15:05	15 minutes	Not Completed

Add To Group | Add | Refresh | Reassign

Call John

Name	Value ▲
Task Reassigned on Nov 30, 2011 3:12:10 PM	Reassigned from agent guest

Agent Task Manager Interface

The *Status* drop-down list, in the upper left corner of the Agent Task Manager, allows you to filter the Task Manager grid according to task status (*All*, *Completed* or *Not Completed*).

When a row in the Task Manager grid is selected, any key/value pairs that are entered in the project API are displayed in the lower frame of the Agent Task Manager. (For information about customizing the *Agent Tasks* portlet, refer to [The Agent Tasks Portlet](#).)

The following buttons in the Agent Task Manager enable you to manage tasks:

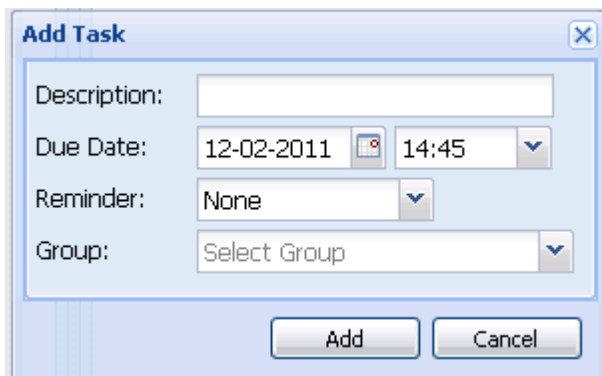
- **Add To Group:** Assigns a task to all members of a selected group. For details, refer to [Adding a Group Task](#).
- **Add:** Assigns a task to a selected agent. For details, refer to [Adding an Agent Task](#).
- **Refresh:** Refreshes the Task Manager grid.
- **Reassign:** Transfers a selected task from the one agent to another agent. For details, refer to [Reassigning a Task](#).

Adding a Group Task

The Add To Group feature allows you to assign a new task to the task lists of all members of a specified group.

To add a group task:

1. At the bottom of the Agent Task Manager, click *Add To Group*.
The *Add Task* dialog appears.



The screenshot shows a standard Windows-style dialog box titled "Add Task". It contains the following elements:

- Description:** A text input field.
- Due Date:** A date picker showing "12-02-2011" and a time dropdown menu showing "14:45".
- Reminder:** A dropdown menu currently set to "None".
- Group:** A dropdown menu currently set to "Select Group".
- Buttons:** "Add" and "Cancel" buttons at the bottom right.

2. In the *Description* textbox, enter an appropriate name for the task.
3. If desired, in the *Due Date* fields, specify the required date and time that the task is due. By default, the task is due 24 hours from the time that the task is assigned.
4. If desired, from the *Reminder* drop-down list, select the timeframe (before the due date/time) at which agents will receive a reminder about the task. By default, no reminder is sent.
5. From the *Groups* drop-down list, select the group to which the task will be assigned.
6. Click *Add*.

The *Add Task* dialog disappears, and the new task is displayed in a new row of the Task Manager grid.

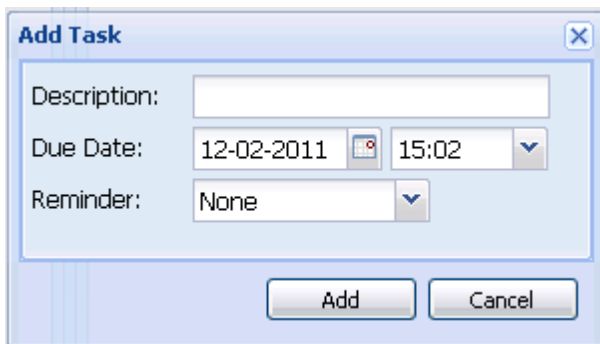
Adding an Agent Task

The Add feature allows you to assign a new task to the task list of a specific agent.

To add an agent task:

1. From the Agents List on the left side of the *Dashboard* tab, verify that the row of the relevant agent is selected.
2. At the bottom of the Agent Task Manager, click *Add*.

The *Add Task* dialog appears.



3. In the *Description* textbox, enter an appropriate name for the task.
4. If desired, in the *Due Date* fields, specify the required date and time that the task is due. By default, the task is due 24 hours from the time that the task is assigned.
5. If desired, from the *Reminder* drop-down list, select the timeframe (before the due date/time) at

which agents will receive a reminder about the task. By default, no reminder is sent.

6. Click *Add*.

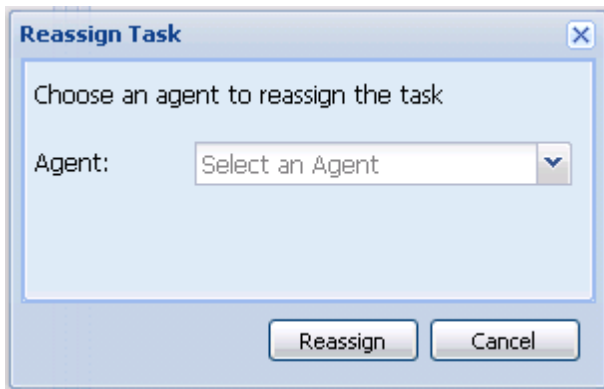
The *Add Task* dialog disappears, and the new task is displayed in a new row of the Task Manager grid.

Reassigning a Task

The Reassign feature allows you to transfer a task from the task list of one agent to the task list of another agent.

To reassign a task:

1. From the Agents List on the left side of the *Dashboard* tab, verify that the row of the agent from whom you want to transfer the task is selected.
2. In the Task Manager grid, select the row of the relevant task.
The *Reassign* button is enabled.
3. At the bottom of the Agent Task Manager, click *Reassign*.
The *Reassign Task* dialog appears.



4. From the *Agent* drop-down list, select the agent to whom you want to assign the task, and then click *Reassign*.
The *Reassign Task* dialog disappears, and the task is transferred to the Task Manager grid of the target agent.

Customizing the Supervisor View

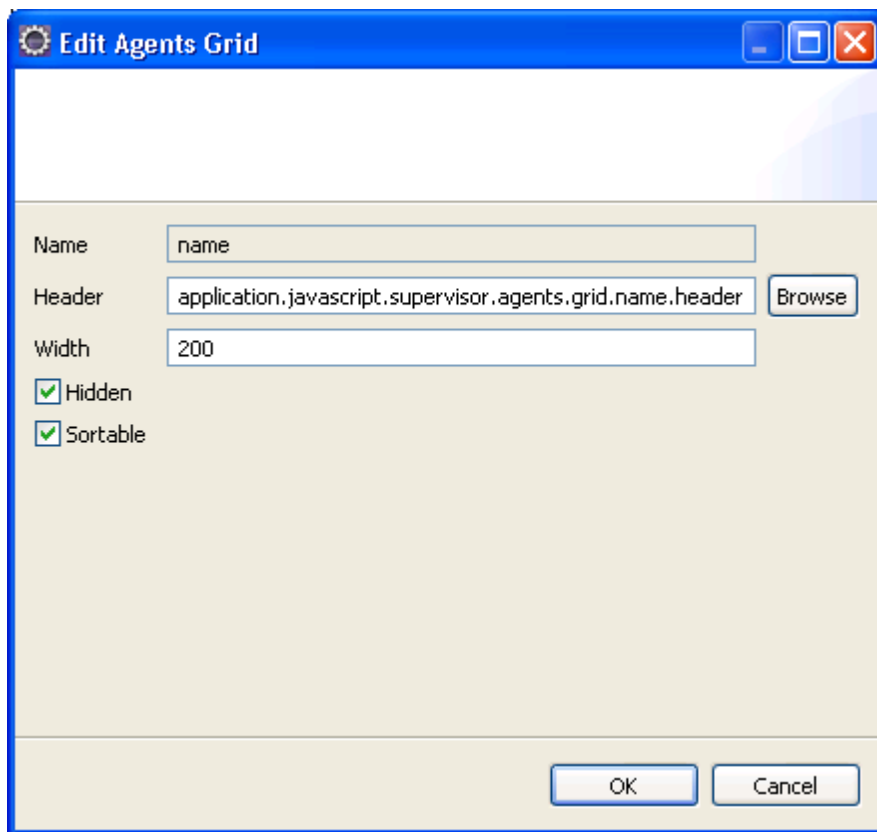
The Supervisor wizard, which is accessed from the development environment, enables you to change the default configuration of the components of the *Dashboard* tab. Using the wizard, you can:

- Update the localization keys of columns in the Agent List grid, and change the order of the columns in the grid
- Add sub-tabs to the *Dashboard* tab, and update the content of sub-tabs
- Edit the text of the buttons in the Task Manager grid, and change the order in which the buttons appear

To use the Supervisor wizard:

1. From the Eclipse IDE, right-click on the relevant project name, and select *Jacada Work Space > Supervisor View Wizard*.

The Supervisor wizard appears.



b. Update the following parameters, as required:

- In the *Header* field, enter the localization key for the content of the column. The header key must begin with **application.javascript**.
- In the *Width* textbox, enter the desired width (in pixels) of the column.
- To hide the column in the UI, check the *Hidden* checkbox.
- To enable the user to sort the grid according to the column, check the *Sortable* checkbox.

c. Click *OK*.

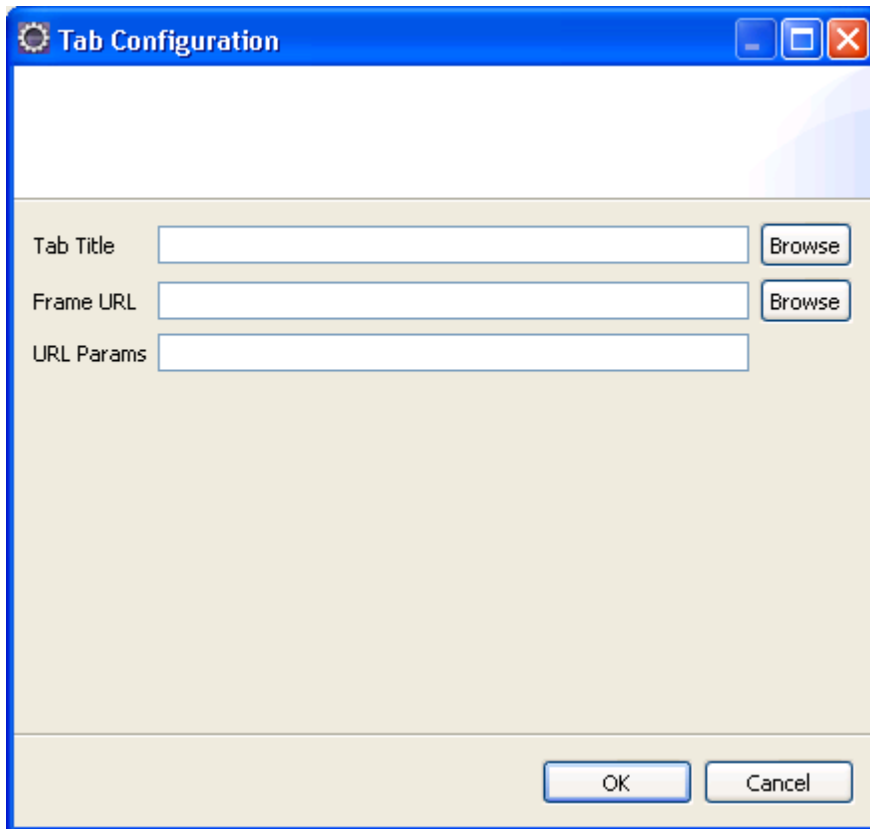
The *Edit Agents Grid* dialog disappears, and updated parameters appear in the *Agents View Grid Configuration* page of the wizard.

3. To change the order of the columns in the Agents List grid, select the relevant column, and use the *Move Up/Move Down* buttons to put the column in the desired location in the grid.
4. Click *Next*.

The *Supervisor Tabs* page of the wizard appears.

Tab Header	Tab Relative URL	Tab Params
application.javascript.s...	/SYSTEM/portlets/supervisor/tabFra...	foo=bar&wtf=1
application.addContent...	/SYSTEM/portlets/search/search.jsp	

- 5. If desired, add a new sub-tab to the *Dashboard* tab:
 - a. At the right side of the *Supervisor Tabs* page, click *New*.
The *Tab Configuration* dialog appears.



- b. In the *Tab Title* field, enter the header key of the relevant sub-tab. Clicking *Browse* enables you to select an existing header key.
- c. In the *Frame URL* field, enter the URL of the subtab's content. Clicking *Browse* enables you to navigate to the relevant URL.
- d. If relevant, in the *URL Params* textbox, enter appropriate URL parameters.
- e. Click *OK*.

The *Tab Configuration* dialog disappears, and updated parameters appear in the *Supervisor Tabs* page of the wizard.

6. If required, edit the grid on the *Supervisor Tabs* page:
 - To update parameters of a sub-tab, select the relevant row in the grid, and then click *Edit*. Alternatively, double-click the relevant row.
In the *Tab Configuration* dialog, update parameters as required. (Refer to Step 5b-e above.)

- Note:** The default sub-tab cannot be deleted.

7. Click *Next*.

Supervisor Wizard

Tasks Actions Configuration

Agent Task Actions configuration

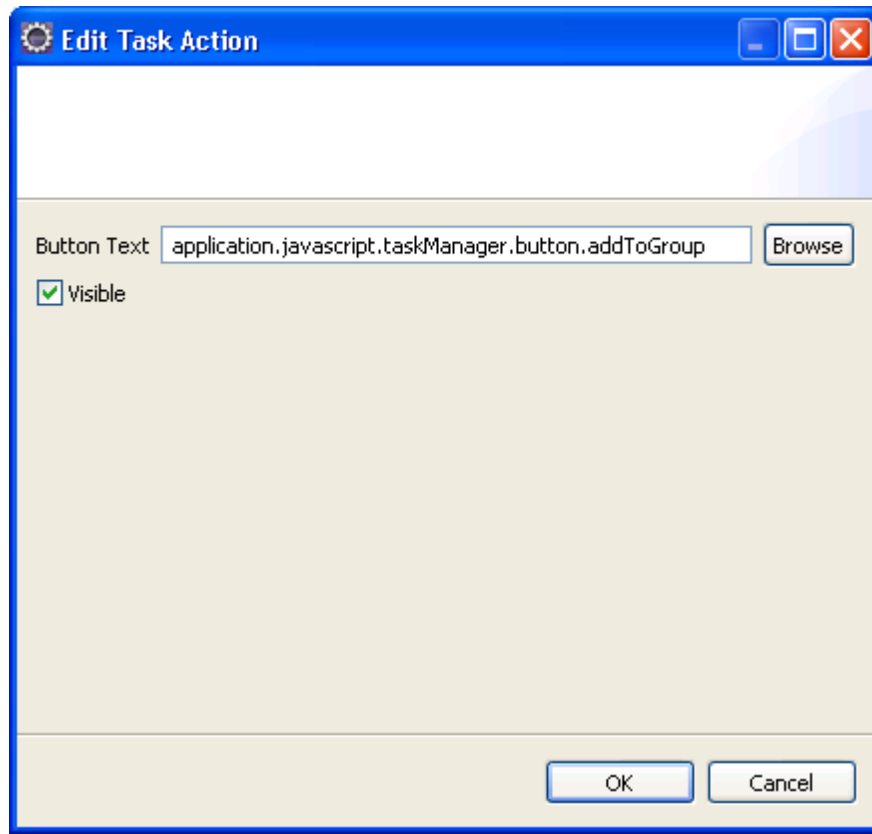
Administer the actions you want to be enabled in the Task View

Button Description	Button Text	Visible
Add task to group	application.javascript.taskMana...	true
Add Task To an Agent	application.javascript.taskMana...	true
Refresh Tasks	application.javascript.taskMana...	true
Reassign Task	application.javascript.taskMana...	true

EditMove UpMove Down

< BackNext >FinishCancel

- The *Edit Task Action* dialog appears.



b. Update the following parameters, as required:

- In the *Button Text* field, enter the localization key for the text of the button. The header key must begin with **application.javascript**.
- To hide the button in the UI, clear the *Visible* checkbox.

c. Click *OK*.

The *Edit Task Action* dialog disappears, and updated parameters appear in the *Tasks Action Configuration* page of the wizard.

9. To change the order of the buttons, select the relevant row in the grid, and use the *Move Up/Move Down* buttons to put the button in the desired location in the grid.

10. Click *Finish*.

The wizard disappears, and the project is republished. The Supervisor View is reconfigured.

Messaging Capabilities

The *Message Center* tab enables creating messages for broadcast to either all Jacada WorkSpace users or to a specific Jacada WorkSpace group. When creating a message, supervisors define its attributes, which include message display location (*Message Board*, ticker tape, welcome screen), required read frequency and ticker tape display appearance.

Jacada WorkSpace instant messaging capability is supplied by the *Chat* portlet. Agents can interactively chat with a supervisor or an expert agent. Also supported are agent to agent chat sessions and agent to peer agent chat sessions. Jacada WorkSpace user authorization controls each users chat privileges; one's ability to chat with a specific category of users and one's ability to initiate chat sessions.

The following sections describe:

- [Message Center Methods](#)
- [Instant Messaging Methods](#)
- [Customizing Date and Time Display](#)
- [Chat Session History](#)

Message Center Methods

Jacada WorkSpace provides a coding infrastructure that is used to develop the project's message center-related processing. This infrastructure makes available the `MessagingManager` interface that exposes the following methods:

Method	Description
<code>public void addMessage(Message message);</code>	Adds a new message to the Message Center.

Method	Description
<code>public void deleteMessage(Integer messageID);</code>	Deletes an existing message, identified by its <code>messageID</code> , from the Message Center.
<code>public void deleteMessages(List<Integer> messageIDs);</code>	Deletes a list of existing messages, each identified by its <code>messageID</code> , from the Message Center.
<code>public long getMessageCount();</code>	Returns the current number of messages.
<code>public void updateMessage(Message message);</code>	Updates an existing message in the Message Center.

Code Example

The following sample code includes the `myNewMessage()` method to add a message:

```
public class MyMessage {
    private MessagingManager messagingManager;
    public void setMessagingManager(MessagingManager messagingManager) {
        this.messagingManager = messagingManager;
    }
    public void myNewMessage() {
        Message msg = new Message();
        msg.setSubject("Subject");
        msg.setContent("Content");
        // configure the message location
        msg.setShowOnWelcomeScreen(true);
        msg.setShowOnTickerTape(true);
        msg.setShowOnMessageBoard(true);
        // configure ticker tape properties
        msg.setFlashOnTickerTape(true);
        msg.setPauseOnTickerTape(true);
        msg.setColorOnTickerTape("#0C0107");
        msg.setActivationDate(new Date());
        msg.setExpiryDate(new Date());
        msg.setMustRead(true);
        msg.setReadOnce(true);
        msg.setRepeating(true); //
        msg.setRepeatingPeriodType(RepeatingPeriodType.ONE_DAY);
        // RepeatingPeriodType is an enum, see the definition below
        msg.setPriorityLevel(3);
        // RecipientType is an enum, has two options ALL or GROUP
        // if equals GROUP, then use setRecipient() for the group name
        // otherwise the setRecipient() is ignored
        msg.setRecipient(null);
        msg.setRecipientType(RecipientType.ALL);
        messagingManager.addMessage(msg);
    }
}
```

The location and expiration date, shown in red above, must be specified.

Note: If a recipient type is not specified in the API, it will be assigned to *All*. If the ticker tape color is not specified in the API, it will be white. No other default values are assigned.

Register the class in the file `WebContent/WEB-INF/applicationContext-user.xml`.

```
<bean id="MyMessage" class="MyMessage">
  <property name="messagingManager" ref="messagingManager"/>
</bean>
```

Access the `myNewMessage` bean. For example, using code similar to the following:

```
ApplicationContext applicationContext =
    WebApplicationContextUtils.
        getWebApplicationContext(getSessionInformation().getSession().
                                   getServletContext());
MyMessage myNewMessage1 = (MyMessage)applicationContext.getBean("MyMessage");
myNewMessage1.myNewMessage();
```

Instant Messaging Methods

When Jacada WorkSpace instant messaging is supported by an XMPP server other than the Openfire XMPP Server, you may need to override, in your project code, some or all of the methods listed in the table that follows.

The relevant bean is `IMContactListProvider`, which is defined in the file `WebContent/WEB-INF/applicationContext-user.xml`. The class to which these methods belong is responsible for Jacada WorkSpace communication with the XMPP server.

In the code, inherit the default implementation and then override any of the following methods:

Method	Description
<code>void beforeConnect(ConnectionConfiguration connectionConfiguration)</code>	Called before connecting to the XMPP server.
<code>void beforeLogin()</code>	Called before logging in to the XMPP server.

Method	Description
<code>void beforeDisconnect()</code>	Called before disconnecting from the XMPP server.
<code>void beforeSetPresence(Presence presence)</code>	Called before a new agent presence is sent to the XMPP server. The <code>presence</code> parameter holds the new presence data.
<code>void prepareUserAttributes(String userName, String password, Map<String, String> userAttributes)</code>	Called before the user registers with the XMPP server. The <code>userAttributes</code> parameter is an empty map that this method populates. Jacada WorkSpace sends the map to the XMPP server in the registration request.

Customizing Date and Time Display

By default, Jacada WorkSpace formats all date and time displays using the `DATE_FORMAT` and the `TIME_FORMAT` properties that are defined in the `WebContent/index.jsp` file. Customize Jacada WorkSpace's display of date and time information for both Instant Messaging related messages (*Chat*) and *Message Center* related messages.

Accomplish this by overriding the default date and time display formats. In the `WebContent/USER/rsources/js/override.js`, file define any of the following properties:

For Instant Messaging (*Chat*) related messages, use:

- `CHAT_DATE_FORMAT`
- `CHAT_TIME_FORMAT`

For *Message Center* related messages, use:

- `MESSAGES_DATE_FORMAT`
- `MESSAGES_TIME_FORMAT`

Each property that is defined, customizes Jacada WorkSpace's display of the associated information. Define these properties using a set of standard JavaScript notations for the supported

date and time formats. For a listing of the supported, format notations, see [Customizing Date and Time Displays](#).

For example, incorporating the following format property definition in `override.js`:

```
MESSAGES_DATE_FORMAT = 'm/j/Y';
```

results in the *Message Board* tab displaying its message entries where the *Sent On* field displays the custom formatted date in the following manner: 03/5/2009. Note that in this example, the time display format has not been customized.

Chat Session History

Jacada WorkSpace provides a *Chat History* tab. *Chat History* provides the supervisor with sub-tabs:

- *Chat Activity: 24 Hour Window*.
- *Session Log*.

Chat History enables supervisors to view the current number of active chat sessions. Via its sub-tabs, supervisors view statistical data regarding agent chat sessions, as follows:

- The *Chat Activity* tab provides a bar graph, with a rolling window display, showing you the total number of chat sessions that occurred in a hour of the day, during the previous 24 hours.
- The *Session Log* tab provides a listing of chat session entries where each displays the following information: chat initiator, chat recipient, chat session start time and the length of the chat session.

The size of the table displayed in the *Session Log* tab is set by the `SESSIONS_ON_PAGE` value. You can change the size of the table by adjusting the value of `SESSIONS_ON_PAGE` in `USER/resources/js/override.js`.

Logging Capabilities

Jacada WorkSpace provides a mechanism for logging Jacada WorkSpace session information. The following sections describe:

- [Producing Log Files](#)
- [Jacada WorkSpace Log Configuration File](#)
- [Log Configuration File Settings](#)
- [Logging Information in a Project](#)
- [Supported Appenders](#)
- [Session-Related Logging](#)

Producing Log Files

Jacada WorkSpace uses the Apache Logging Services APIs to produce log files. The main components of the Apache logging APIs are *loggers*, *appenders*, and *layouts*.

Loggers are the logic components that describe the target for log messages. Appenders are the components that define where and how to write the message, for example, to a console or to a log file. Layouts define the formats of the loggers output.

A single logger can have several appenders. An important feature of loggers is that they have a hierarchical structure. This means that they inherit a property from the first ancestor for which this property is defined.

Loggers, appenders, and layouts are specified in XML log configuration files. By changing the values of their parameters, you can customize the logging to create different log files.

Jacada WorkSpace uses `log4j` as the logging infrastructure. For more information on the `log4j` APIs, follow the links to the individual projects in the Apache Logging Services Project home page at: <http://logging.apache.org/>

In addition to the logging related to the project deployed in the application server, Jacada WorkSpace provides the following:

- A session dump log file. A session dump is generated when a Jacada WorkSpace session terminates due to an unexpected error. The session dump contains session-specific information to help understand the cause of the error.
- A nested application log file. Nested applications – any Windows-based or Web-based application that is incorporated into a Jacada WorkSpace tab – write to their own, common log file. The logging performed by nested applications occurs in and resides on the agents' desktop.

Jacada WorkSpace Log Configuration File

A log configuration file consists of `<logger>` and `<appender>` sections, that define the properties of each logger and each appender. Each logger is associated with one or more appenders. An appender includes where to which to write the messages.

The log configuration can be accessed and modified through the Jacada Management Console. For details, refer to the Jacada WorkSpace Administrator Guide.

The Jacada WorkSpace log configuration is saved in the `JacadaLogConfiguration.xml` file, located in the application server directory.

The default Jacada WorkSpace log configuration is as follows:

Logger	Appender	Log File
Aila	AilAppender	logs/Ail.log
Jacada.Admin	FusionFile	logs/Fusion.log
Jacada.Debug	FusionFile	logs/Fusion.log
Jacada.Admin.JFAP.ADMIN	JFAPSingleFileAdmin	logs/JFAP-Admin.log
Jacada.Debug.JFAP.CTI	JFAPCTIFile	logs/CTI.log
Jacada.Debug.JFAP.CTIGateway	JFAPCTIGatewayFile	logs/CTIGateway.log

Logger	Appender	Log File
<code>Jacada.Debug.JFAP.SCRIPT</code>	<code>JFAPSingleFileScript</code>	<code>logs/JFAP-Script.log</code>

Note: The `JacadaLogConfiguration.xml` contains loggers, for 3rd party software, such as Beehive, Hibernate, and Spring, and `Jacada.Debug.JFAP.DEV_INTERNAL`. These loggers are for internal use only. Do not modify these loggers.

Logger Names and Appender Additivity

The logger names implement a hierarchical structure by which a logger is said to be an ancestor of another logger if its name followed by a dot is a prefix of the descendant logger name. A logger is said to be a parent of a child logger if there are no ancestors between itself and the descendant logger.

For example, `Jacada.Debug` is a parent of `Jacada.Debug.JFAP`, which in turn is a parent of `Jacada.Debug.JFAP.SCRIPT`.

Log4j implements an “appender additivity” feature, by which appender accumulation is additive. This means that when the additivity flag is set to true, each logging request for a given logger is forwarded to all the appenders in that logger as well as the appenders of the parent logger in the hierarchy.

For example, in `Jacada Workspace` all the `Jacada.Debug.JFAP.<NAME>` loggers are set to use additivity, which means that in addition to writing to the log files defined by their own appenders, they all also write to the `JFAP-Debug.log` which is defined by the appender of their parent logger `Jacada.Debug.JFAP`. In a similar manner, all the `Jacada` loggers write to the `Fusion.log`.

Note: If an ancestor of logger E, for example C, has the additivity flag set to false, then E's output will be directed to all the appenders in E and it's ancestors up to and including C but not the appenders in any of the ancestors of C.

Log Files

The log files are used as follows:

File Name	Contains
Fusion.log	All log messages.
JFAP-Debug.log	Debug-specific information.
JFAP-Admin.log	Application information.
JFAP-Script.log	Script-specific information.
CTI.log	CTI information.
CTIGateway.log	CTI Gateway information.

By default, the log files are created in the application server `logs` directory.

Log Configuration File Settings

The following sections describe in detail the settings in a logger section and in an appender section of the log configuration file.

Logger Section

A `<logger>` section contains the following settings:

Setting Name	Description
name	The name of the logger. The logger name has a hierarchical structure, which together with the <code>additivity</code> setting, determines which appenders are used and which log files the logger writes to. For more information, see Logger Names and Appender Additivity .
additivity	A boolean parameter which specifies whether the logger output is sent to the appenders of the parent logger in addition to its appenders. When set to <code>false</code>, the logger outputs to its appenders only. When the parameter is not specified it is implied to be <code>true</code>.
level	The debug level of the logger. See "Debug Levels", below, for details.
appender-ref	The name of an appender to which the logger sends its output. Since this is the setting that links the logger to the appender, an appender with this name must be specified in the file.

Debug Levels

The different debug levels determine what information is written to the logs. You can set the debug level of the file to one of the levels listed in the table below.

The **ALL** level allows output of all log messages. At the other end of the scale, the **OFF** level turns off all output. The levels between **ALL** and **OFF** place increasingly growing limitations on the output of messages. When going from one level to the next, less messages are output and they are a result of error situations of increasing severity.

Level	Description
ALL	Turns on all logging.
TRACE	Logs all the server HTTP requests.

Level	Description
DEBUG	Logs informational events that are most useful to debug an application.
INFO	Logs informational messages that highlight the progress of the application.
WARN	Logs potentially harmful situations.
ERROR	Logs error events that might still allow the application to continue running.
FATAL	Logs severe error events that lead the application to abort.
OFF	Turns off logging.

Example

The following is an example of a logger definition from the `JacadaLogConfiguration.xml`:

```
<logger name="Jacada.Admin.JFAP.ADMIN" additivity="true">
  <level value="INFO"/>
  <appender-ref ref="JFAPSingleFileAdmin" />
</logger>
```

TRACE Debug Level

When using the `TRACE` level, you must add the `com.jacada.logging.TraceLevel` class attribute in the logger definition. For example:

```
<logger name="Jacada.Debug.JFAP.ADMIN" additivity="true">
  <level value="TRACE" class="com.jacada.logging.TraceLevel"/>
  <appender-ref ref="JFAPSingleFileAdmin" />
</logger>
```

For all other debug levels, the class attribute is not specified.

Appender Section

Appenders are the output destinations of loggers.

All `<appender>` sections contain the following basic settings:

Setting	Description
name	The name of the appender.
class	The Java class that imports the appender. In <code>JacadaLogConfiguration.xml</code> , the full class name is specified (for example, <code>org.apache.log4j.FileAppender</code>).
layout	A section that defines the log's format.

For a list of the appenders supported by Jacada WorkSpace, see [Supported Appenders](#).

Layout Section

Each appender contains a `<layout>` section, that defines the output format of the appender.

Jacada WorkSpace uses different layouts defined by the following classes:

Class Name	Description
PatternLayout	Formats a logging event and returns the result as a string.
TTCCLayout	Format consists of time, thread, category and nested diagnostic context information.

Note: In `JacadaLogConfiguration.xml`, the layout is specified by the full class name (for example, `<class="org.apache.log4j.PatternLayout">`).

PatternLayout Properties

The `PatternLayout` returns a logging event as a string. The string's format is specified in the `ConversionPattern` parameter. This parameter is composed of literal text and format control

expressions (conversion specifiers).

Each conversion specifier starts with a percent sign (%) and is followed by optional format modifiers and a conversion character. The conversion character specifies the type of data, such as category, priority, date, thread name, etc. The format modifiers control such things as field width, padding, left and right justification.

The following conversion characters are used in the log configuration file:

Conversion Character	Effect
c	Used to output the category of the logging event. The category conversion specifier can be optionally followed by a precision specifier, that is a decimal constant in brackets. If a precision specifier is given, only the corresponding number of right most components of the category name will be printed. By default the category name is printed in full. For example, for the category name “a.b.c” the pattern %c{2} will output “b.c”
d	Used to output the date of the logging event. The date conversion specifier may be followed by a date format specifier enclosed between braces. For example, %d{HH:mm:ss,SSS} or %d{dd MMM yyyy HH:mm:ss,SSS}. If no date format specifier is given then ISO8601 format is assumed. For best results it is recommended to use the log4j date formatters. These can be specified using one of the strings “ABSOLUTE”, “DATE” and “ISO8601” for specifying AbsoluteTimeDateFormat, DateTimeDateFormat and respectively ISO8601DateFormat. For example, %d{ISO8601} or %d{ABSOLUTE}.
m	Used to output the application supplied message associated with the logging event.
n	Outputs the platform dependent line separator character or characters.
p	Used to output the priority of the logging event.

Conversion Character	Effect
t	Used to output the name of the thread that generated the logging event.

For more information, go to <http://logging.apache.org/>.

Logging Information in a Project

In a Jacada WorkSpace project, you can log information from different parts of the project. For example, logging information from a page (*.JSP) is done differently than logging information from a file that uses Java code.

The following sections describe:

- [Adding Logging to a Page](#)
- [Excluding Sensitive Information From the Log](#)
- [Creating a Session Dump](#)
- [Adding Logging to Java Code](#)
- [Specifying the Error Message to Write to the Log](#)

Adding Logging to a Page

Jacada WorkSpace provides a set of logging tags, such as *Trace*, *Debug*, *Info*, *Warn*, *Error*, *Fatal*, to enable adding to a page log messages of different debug levels. The log messages can be directed either to the `JFAP-Script.log` file, for script-related logging, or to the `JFAP-Admin.log` file, for application-related logging. For more information, see [Log Files](#).

To use the logging tabs, the following line must be included in the JSP page:

```
<%@ taglib uri="http://jakarta.apache.org/taglibs/log-1.0" prefix="log"%>
```

To add logging to a page:

- Use a logging tag, such as `<log:debug>`, at the point at which you want the log message to be generated, specifying the following attributes:

loggerType – Specify the logger name, either `ADMIN` (for application related logging) or `SCRIPT` (for script related logging).

message – The message text to be written to the log file.

Excluding Sensitive Information From the Log

You can set the logging for a specific page to exclude from the log sensitive information, such as credit card numbers. You can exclude all elements from the page or specific elements on the page.

To exclude information from the log, the following line must be included in the JSP page:

```
<%@ taglib uri="http://jakarta.apache.org/taglibs/log-1.0" prefix="log"%>
```

To exclude sensitive information from the log:

- Use the logging tag `<log:exclude>` at the point at which you want the log message to be generated, specifying the following attributes:

elementsToExclude – Specify the elements that should not be logged, separated by spaces.

excludeEntirePage – Specify `true` to exclude all elements in the page from logging. The default value is `false`.

Creating a Session Dump

You can generate session-specific information that is written to a log file, in order to help understand the reason for an error. The log file, referred to as a session dump, contains environment information, session information and lists session resources as well. For further details, refer to [The Session Dump](#).

To create a session dump, the following line must be included in the JSP page:

```
<%@ taglib uri="http://jakarta.apache.org/taglibs/log-1.0" prefix="log"%>
```

To create a session dump:

- Use the logging tag `<log:dump>` at the point at which you want the dump to be generated, specifying the following attribute:

scope – Specify the scope of the dump.

Adding Logging to Java Code

Jacada WorkSpace enables writing log messages on the server side from Java code using the following class:

```
com.jacada.jad.logging.LogWrapper
```

The `LogWrapper` class contains the following functions for each debug level:

```
public static void fatal(int type, String msg, Throwable ex)
public static void fatal(int type, Throwable ex)
public static void fatal(int type, String msg)
public static void fatal(String msg, Throwable ex)
public static void fatal(Throwable ex)
public static void fatal(String msg)
public static void error(int type, String msg, Throwable ex)
public static void error(int type, Throwable ex)
public static void error(int type, String msg)
public static void error(String msg, Throwable ex)
public static void error(Throwable ex)
public static void error(String msg)
public static void warn(int type, String msg)
public static void warn(String msg)
public static void info(int type, String msg)
public static void info(String msg)
public static void debug(int type, String msg)
public static void debug(String msg)
public static void trace(int type, String msg)
public static void trace(String msg)
```

where:

msg – The text to write to the log

ex – The exception that was caught within a `try/catch` block

type – One of the following logger types, each of which maps to a Jacada logger:

Logger Type	Maps to Jacada Logger:
<code>LogWrapper.GENERAL</code>	<code>Jacada.Debug.JFAP</code>
<code>LogWrapper.SCRIPT</code>	<code>Jacada.Debug.JFAP.SCRIPT</code>
<code>LogWrapper.ADMIN</code>	<code>Jacada.Admin.JFAP.ADMIN</code>
<code>LogWrapper.CTI</code>	<code>Jacada.Debug.JFAP.CTI</code>
<code>LogWrapper.CTI_GATEWAY</code>	<code>Jacada.Debug.JFAP.CTIGateway</code>

The `LogWrapper.GENERAL` logger is the default, if type is not specified.

In addition, the `LogWrapper` class contains another function that enables checking the current debug level of a specific logger, before sending messages to the logger:

```
isEnabledFor(int type, Priority level);
```

The Jacada loggers are defined in the `JacadaLogConfiguration.xml` file, located in the application server directory.

Code Example

In the following example, an error message level is written. The message includes the text of the exception that was thrown. Since the `type` parameter is not specified, the default `LogWrapper.GENERAL` is used.

```
try {  
    ...  
} catch (Exception ex) {  
    LogWrapper.error("Incorrect selection", ex);  
}
```

Specifying the Error Message to Write to the Log

The error messages generated by Jacada WorkSpace have the following format:

<ID><LEVEL><#>=<message><params>

where:

ID – Identifies the relevant component for the error, such as CTI or PSH for push-based errors.

LEVEL – The error level where 1 is fatal, 2 is error, etc.

– The error message number.

message – The error message text that can include parameters such as {0}, {1}. All parameters are replaced at run time with their real values.

For example:

```
CTI11004="Error running method {0} of channel type {1} in manager class {2}: {3};  
Cause: {4}"  
Or,  
PSH10007="Duplicate push handlers for session {0}"
```

All Jacada WorkSpace supplied error messages are specified in `WebContent/SYSTEM/resources/resourceBundles/LogMessages.properties`. Define your own Jacada WorkSpace project specific error messages and add them to the file `WebContent/USER/resources/resourceBundles/LogMessages.properties`.

Supported Appenders

Jacada WorkSpace supports the following appenders:

- `ConsoleAppender`: Causes log events to be written to the console.
- `FileAppender`: Causes log events to be written to a file.
- `RollingFileAppender`
- `DailyRollingFileAppender`
- `ExternallyRolledFileAppender`
- Cluster Appenders

RollingFileAppender

The `RollingFileAppender` extends `FileAppender` to backup the log file when it reaches a certain size and create a new log file. The following settings determine how big the log file is before it is rolled over and how many backups are kept:

MaxFileSize – Specifies the maximum file size of the log file. When the value is reached, the file is backed up and a new log file is created. The default value is 10 Mb.

MaxBackupIndex – Specifies how many backups of the log file are kept. There is one backup file by default.

DailyRollingFileAppender

The `DailyRollingFileAppender` extends `FileAppender`. It backs up the log file at a specified frequency, as opposed to the `RollingFileAppender` which depends on the file size. The rolling schedule is specified by a `DatePattern` parameter. This pattern follows simple date format conventions such as `yyyy-MM-dd` or `dd-MM-yy`. The date pattern is used as the suffix for the rolled file name.

For example, if the `File` parameter is set to `/foo/bar.log` and the `DatePattern` set to `'yyyy-MM-dd'`, on 2001-02-16 at midnight, the logging file `/foo/bar.log` will be copied to `/foo/bar.log.2001-02-16` and logging for 2001-02-17 will continue in `/foo/bar.log` until it rolls over the next day. Note the period separator is specified in quotes.

ExternallyRolledFileAppender

The `ExternallyRolledFileAppender` extends `RollingFileAppender`.

This appender listens on a socket on the port specified by the `Port` property for a “RollOver” message. When such a message is received, the underlying log file is rolled over and an acknowledgment message is sent back to the process initiating the roll over.

Cluster Appenders

In a clustered environment all the managed servers on a given machine use the same `JacadaLogConfiguration.xml` file. Therefore all the servers try to write to the same log files. In order to avoid this situation, Jacada provides an appender that enables each managed server in a cluster environment to write its own log by creating a separate log directory for each managed server.

Each appender extends a different log4j file appender. The name of the cluster appender is identical to the name of the log4j appender with the prefix `Cluster` added. For example, the `ClusterDailyRollingFileAppender` extends the log4j `DailyRollingFileAppender`.

These appenders add the name of the server to the path of the log file in the `File` parameter as the last directory name, so that every server writes its logs to a different directory.

Session-Related Logging

In addition to the logging related to the deployed project, Jacada WorkSpace provides the following, Jacada WorkSpace session-related logging:

- [The Session Dump](#)
- [Nested Application Logging](#)

The Session Dump

When a Jacada WorkSpace session terminates due to an unexpected error, session-specific information is written to a log file, in order to help understand the reason for the error. The log file, referred to as a session dump, contains environment information, session information and lists session resources as well.

Each session dump is written to a separate log file in the logs directory under the application server.

The name of the log begins with the `'ws_'` prefix followed by the WorkSpace session ID.

The generated file will be listed in an index file (`index.html`, in the same directory), that will contain pointers to all the session dump log files generated on the same server.

The `IsSessionDumpEnabled` field, which is available in the *Global Settings* page of the Jacada Management Console, is used to determine whether to generate the session dump log files. By default, the field's value is `true`, meaning that session dump log files are generated. For information about the *Global Settings* page, see the *Jacada WorkSpace Administrator Guide*.

Session Dump Content

The session dump consists of the following segments:

- General environment information
- General session information
- Accumulated session information

The following tables describe each segment.

General Environment Information

Information	Description
Server information	The IP address and operating system details of the server, and the server memory consumption.
Jacada WorkSpace information	The version of Jacada WorkSpace, and the name of the running application.
Java version	The JDK version and additional details.

General Session Information

Information	Description
Session ID	The session ID and application ID.
Agent ID	The user name of the logged-in agent and related details.

Information	Description
Login time	The time when the agent logged in to Jacada WorkSpace.
Session termination time	The time when the session terminated.
CTI agent details	The CTI agent name, extension number and other CTI details. If CTI is not in use (working in manual mode), a message indicating this will be shown instead.

Accumulated Session Information

This segment includes information accumulated during the session lifetime, to help understand the cause of the session dump:

Information	Description
Dump title	A short description of the cause for the dump, such as “An unexpected exception occurred”.
Dump description	A more detailed description of the dump.
Dump date	The date and time when the dump started.
Log messages	A list of recent log messages for the session, including messages not written to the log because of the log level.
Last exceptions thrown	A list of recent exceptions, which were thrown by the session.
Last accessed pages	A list of recent <code>URLs</code> , of pages which were visited by the session.
Fatal exceptions	A list of the exceptions that caused the error condition which generated the dump.
Last push messages sent	A list of recent push messages, with keyword and parameters, which were sent by the session.

Nested Application Logging

In each agent desktop that has installed the `JacadaApplicationHolder` ActiveX control, nested applications, running in Jacada WorkSpace tabs, will perform logging.

During a Jacada WorkSpace session, each nested application that is running in a Jacada WorkSpace tab writes entries to the following log file, which is located in the agent desktop:

`${ALLUSERSPROFILE}\Jacada\JAD\LogFiles\Nesting.log`. Nested applications write to this file via the `JacadaApplicationHolder` ActiveX control.

Upon an agent logging in to a Jacada WorkSpace session and the start of the ActiveX control, the Jacada WorkSpace project issues the current application server time and the WorkSpace session ID, which the ActiveX control then records in the `Nesting.log` file.

Each entry provides the following information about the nested application:

Local time in the agent desktop – for example, 26 Nov 2008 19:06:44,723.

Information debug level – for example, `INFO`.

Source file and line number – for example, `jadnesting.cpp: 343`.

Thread and thread ID – for example, 5,520.

Time in the application server – for example, 2008/11/26-18:05:30.

Recorded log message – for example, `Effective desktop Rect {205,354,662,451}`.

The following is an example of an entry recorded in the `Nesting.log`:

```
26 Nov 2008 19:06:44,723 [INFO] [jadnesting.cpp: 343] [5,520] [2008/11/26-18:05:30] Effective desktop Rect {205,354,662,451}
```

Configuration of Nested Application Logging

The `NestingLogConfiguration.xml` file contains the configuration definitions necessary for the logging performed by nested applications in Jacada WorkSpace. `NestingLogConfiguration.xml` exists in each agent desktop that has installed the `JacadaApplicationHolder` ActiveX control, which supports the nesting of applications in Jacada WorkSpace tabs. By default, this configuration file resides in `<Jacada_installation_folder>/JacadaApplicationHolder/Configuration/Logs`.

`NestingLogConfiguration.xml` establishes the use of the `TimeRollingFile` appender with the `DailyRollingFileAppender` class for the `Nesting.log` file. Nested application back-up log files are located in the same folder as the `Nesting.log` file and are uniquely identified by the naming convention of `Nesting.log.yyyy-MM-dd`. The default `Debug level` of information written to the log

file is `ERROR`.

Update any of the provided logging configurations directly in the `NestingLogConfiguration.xml` file; the Jacada Management Console does not currently support updating nested application logging configuration. In order that updates made in the configuration file take effect for the logging performed by a nested application, it is necessary to close and then re-open the nested application in the tab.

Authentication and Authorization

Jacada WorkSpace includes project authentication and authorization. *Authentication* is the control of user login access to Jacada WorkSpace. *Authorization* is the control of agent and supervisor utilization of Jacada WorkSpace's features.

The following sections present:

- [Overview](#)
- [Predefined Authorization in Jacada WorkSpace](#)
- [Setting Up Authorization in Jacada WorkSpace](#)
- [Methods Available for Use with Authorization](#)

Overview

Jacada WorkSpace includes project authentication and authorization to enable or restrict access to Jacada WorkSpace – authentication – and use of its features – authorization.

During development, both authentication and authorization is implemented based on information in the Jacada Management Console. In a production environment, an LDAP server is used to authenticate users, while authorization is still implemented using the Jacada Management Console.

Jacada WorkSpace has the following settings for authenticating user log in, set in the Jacada Management Console:

Standalone – The information defined in the Jacada WorkSpace database is used to authenticate user log in. This setting is used during development.

External – An external LDAP server is used to authenticate user log in. Users and groups are owned and managed by the LDAP server. Jacada WorkSpace imports this information from the LDAP server. This setting is used during production. When transitioning from Standalone mode to

External mode, all users and groups must be deleted from the database. Perform this activity using the Jacada Management Console.

A project implements authentication and authorization using the following entities:

User – A user login (name and password) that undergoes *authentication* to access a Jacada WorkSpace session. Once establishing a Jacada WorkSpace session, each user's ability to work with Jacada WorkSpace, that is their roles and privileges, is determined by their group affiliations. Users are assigned to one or more Groups.

Group – A set containing one or more *Users*. Additionally, each group has a profile consisting of one or more of the following assignments:

- Roles – The roles assigned to a group determine the usage rights for the users in the group, via privileges.
- Supervisors of the group – Specific supervisors from the pool of supervisors who supervise this group.
- Expert groups – Whether the group, and thus the users in the group are experts in the contact center. This enables individuals groups to be defined which can be treated differently using APIs. The APIs available to use with expert groups are described in [Client-Side Methods](#).
- Peer groups – Groups where the users in the group have similar status. This enables individuals groups to be defined which can be treated differently using APIs. The APIs available to use with peer groups are described in [Client-Side Methods](#).

Privilege – The usage rights defined for a feature. Privileges are specified in the project for each feature, for example, access to a specific tab or portlet or control in a JSP page such as a button or label. If a privilege is not specified for a feature then there is no restriction on any agent using the feature. Privileges are assigned to *Roles*.

Role – A set containing one or more *Privileges*. Each privilege assigned to a role can be either enabled or disabled. When the privilege is enabled, the user has full access to the specific feature. When the privilege is disabled, the user has read-only access to the specific feature.

Note: Enabling or disabling a privilege only applies to input and selection controls in a JSP page, defined using specific Jacada tags. If the privilege is disabled on a Label control, the label text is not displayed.

Skill – A set defining one or more areas of expertise related to the customer's business and the Jacada WorkSpace project. Skills are assigned to Groups.

These entities are set in the Jacada Management Console. For details, refer to the Jacada WorkSpace Administrator Guide.

Predefined Authorization in Jacada WorkSpace

Jacada WorkSpace provides a base set of users, groups, roles and privileges.

Entity	Value
Users	guest, workspace (both with password workspace)
Groups	Agents, CTIUsers, Supervisors
Roles	AgentRole, CTIUserRole, SupervisorRole
Privileges	Jacada WorkSpace provides privileges for the various features of Jacada WorkSpace, including portlets, menu options, CTI options and tabs. For a list of the Jacada WorkSpace-provided privileges, see the Jacada WorkSpace Administrator Guide.

The user `guest` is authorized to use basic Jacada WorkSpace features. The user `workspace` is authorized to use the entire range of Jacada WorkSpace features. Look in the Jacada Management Console for details of what groups, roles and privileges are assigned to these users.

Setting Up Authorization in Jacada WorkSpace

The `privilege` attribute is used to specify the privilege for a feature defined in Jacada WorkSpace. The following features include this attribute:

- Menus
- Portlets
- Tabs
- Toolbar buttons

For example, the Message Board tab has the following definition in `tabs.jspf`:


```
<jacada:tab id="messagesTab" accessKey="2" privilege="TabMessageBoard"
    removable="true" maximizable="true"
    header="application.tab.label.messageBoard" >
  <jacada:frame id="MessageBoardFrame" height="100%" width="100%"
    fusion="auto"
    url="SYSTEM/portlets/messaging/messageBoard.jsp" />
</jacada:tab>
```

Note: The `role` attribute in these files, `tabs.jspf`, `menus.xml`, etc. is deprecated. It is supported only for backward compatibility.

In addition, the `privilege` attribute can be set on the following controls used in a JSP page, for example in a call handling script or portlet:

- Button
- Checkbox and checkbox group
- Combo box
- Label
- Radio button and radio button group
- Text input field and text input area

For example, to define a button with authorization:

```
<jacada:button privilege="a_privilege" id="id1"
    style="background-color: #ffffff" type="button" value="val1"
    title="title2" />
```

Authorizing the Display of JSP Pages

For JSP pages, the Spring framework provides the `authorize` tag. This tag is used to authorize access to the code contained within it. If the user has the required role, the code in the tag is processed and the relevant portion of the page is displayed. The following code is an example of how to use the `authorize` tag in a JSP page:

```

<sec:authorize ifAllGranted="role_name">
  /* Only users assigned the above role access this code section.
   * Unauthorized users do not see the code processing result. */

  Restricted code here

</sec:authorize>

```

Methods Available for Use with Authorization

Jacada WorkSpace provides the following methods to use with the authorization defined in the Jacada Management Console:

- [Server Side Methods](#)
- [Client-Side Methods](#)

Server Side Methods

The following methods can be used in the project Java code:

Supervisor Related Methods

The `UsersManager` interface exposes the following supervisor related methods:

Method	Description
<pre>String[] getSupervisorsForUser(String userName);</pre>	Returns the array of supervisor user names for the specified agent <code>userName</code>. The method returns all the supervisors for all the groups to which the specified agent belongs.
<pre>String[] getSupervisorsForUser();</pre>	Returns the array of supervisor user names for the current agent. The method returns all the supervisors for all the groups to which the current agent belongs.

Method	Description
<code>String getSupervisorForUser(String userName);</code>	Returns a single supervisor user name for the specified agent <code>userName</code> .
<code>String getSupervisorForUser();</code>	Returns a single supervisor user name for the current agent.
<code>boolean isUserSupervisor(String userName);</code>	Returns <code>true</code> if the specified <code>userName</code> is a supervisor, based on the following criteria: • The user is a member of a group that is assigned the role <code>SupervisorsRole</code>. • The user is assigned as a supervisor to some group.
<code>boolean isUserSupervisor();</code>	Returns <code>true</code> if the current user is a supervisor, based on the following criteria: • The user is a member of a group that is assigned the role <code>SupervisorsRole</code>. • The user is assigned as a supervisor to some group.
<code>Set<String> getUserGroups(String userName);</code>	Returns the set of group names to which the specified user <code>userName</code> belongs.
<code>Set<String> getUserGroups();</code>	Returns the set of group names to which the current agent belongs.
<code>List<String> getAllGroups();</code>	Returns the list of all the groups.

Expert Related Methods

The `ExpertsInfoProvider` interface exposes the following expert related methods:

Method	Description
<code>Set<String> getExpertsGroupsNames();</code>	Returns the set of expert group names for the current agent.
<code>Set<String> getExpertsNames();</code>	Returns the set of expert user names for the current agent.
<code>Set<String> getExpertsGroupsNames(String userName);</code>	Returns the set of expert group names for the specified agent <code>userName</code> .
<code>Set<String> getExpertsNames(String userName);</code>	Returns the set of expert user names for the specified agent <code>userName</code> .
<code>Set<String> getGroupSkills(String groupName);</code>	Returns the set of skills for the specified group <code>groupName</code> .
<code>Set<String> getUserSkills();</code>	Returns the set of skills for the current agent.
<code>Set<String> getUserSkills(String userName);</code>	Returns the set of skills for the specified agent <code>userName</code> .

Peer Related Methods

The `PeersInfoProvider` interface exposes the following peer related methods:

Method	Description
<code>Set<String> getPeersGroupsNames();</code>	Returns the set of peer groups for the current agent.
<code>Set<String> getPeersGroupsNames(String userName);</code>	Returns the set of peer groups for the specified agent <code>userName</code> .
<code>boolean peers(String peerName);</code>	Returns true if the specified <code>peerName</code> is a peer of the current agent.

Method	Description
<code>boolean peers(String userName, String peerName);</code>	Returns true if the specified <code>peerName</code> is a peer of the specified agent <code>userName</code> .

Role and Privilege Related Methods

The `Authorization` interface exposes the following role and privilege related methods:

Method	Description
<code>public boolean isPermitted(String privilege, PermissionType permission);</code>	Returns <code>true</code> if the user has the type of permission, specified in <code>permission</code>, for the specified <code>privilege</code>. Permission types for privileges are: • Full use (privilege assigned to role and enabled). • Read-only (privilege assigned to role and disabled).
<code>public boolean isPrivileged(String privilege);</code>	Returns <code>true</code> if the user has the specified <code>privilege</code> .
<code>public boolean isUserInRole(HttpServletRequest request, String role);</code>	Returns <code>true</code> if the user has the specified <code>role</code> .
<code>public List<String> getRoles();</code>	Returns the list of all the roles of the user.

Authorizing Use of a Method

Spring security framework provides the ability to restrict the invocation of any server-side method, based on a specific role. In project code, annotate server-side methods, on a class or an interface, to limit user access to them. The following code is an example of adding annotation to restrict method use:

```
public interface BankService {
    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account readAccount(Long id);
    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account[] findAccounts();
    @Secured("ROLE_TELLER")
    public Account post(Account account, double amount);
}
```

Client-Side Methods

The following JavaScript functions can be used in the project HTML-based code:

- Supervisor Related Functions
- Expert Related Functions
- Peer Related Functions
- Role and Privilege Related Functions

In order to prevent network communication overload being caused by client to server inquiries, a client-side cache is used. Caching in the client is initiated upon user log in to a Jacada WorkSpace session and is performed by the following, most commonly used, client-side methods:

- `isUserSupervisor`
- `isPermitted`
- `isPrivileged`
- `isUserInRole`
- `getRoles`

To override the client-side caching function `isRAPCacheEnabled()`, define the identically named function in the `WebContent/USER/resources/js/override.js` file. The following code is an example of overriding this function in `override.js`:

```
function isRAPCacheEnabled() {
    return false;
}
```

Supervisor Related Functions

The `RAPManager.js` class exposes the following supervisor related functions:

Method	Description
<code>function getSupervisorsForUser();</code>	Returns the array of supervisor user names for the current agent. The method returns all the configured supervisors for all the groups to which the current agent belongs.
<code>function getSupervisorForUser();</code>	Returns a single supervisor user name for the current agent.
<code>function isUserSupervisor();</code>	Returns true if the current agent is a supervisor, based on the following criteria: • The agent is a member of a group that is assigned the role <code>SupervisorsRole</code>. • The agent is assigned as a supervisor to some group.
<code>function getUserGroups();</code>	Returns the set of group names to which the current agent belongs.

Expert Related Functions

The `RAPManager.js` class exposes the following expert related functions:

Method	Description
<code>function getExpertsGroupsNames();</code>	Returns the set of expert groups names for the current agent.
<code>function getExpertsNames();</code>	Returns the set of expert user names for the current agent.
<code>function getGroupSkills(groupName);</code>	Returns the set of skills for the specified group <code>groupName</code> .
<code>function getUserSkills();</code>	Returns the set of skills for the current agent.

Method	Description
<code>function isUserExpert();</code>	Returns <code>true</code> if the current agent is an expert and is a member of a group that is assigned as an expert group to another group.

Peer Related Functions

The `RAPManager.js` class exposes the following peer related functions:

Method	Description
<code>function getPeersGroupsNames();</code>	Returns the set of peer groups names for the current agent.
<code>function peers(String peerName);</code>	Returns <code>true</code> if the specified user <code>peerName</code> is a peer of the current agent.

Role and Privilege Related Functions

The `RAPManager.js` class exposes the following role and privilege related functions:

Method	Description
<code>function isPermitted(privilege, permission);</code>	Returns <code>true</code> if the user has the type of permission, specified in <code>permission</code>, for the specified <code>privilege</code>. Permission types for privileges are: • Full use (privilege assigned to role and enabled). • Read-only (privilege assigned to role and disabled).
<code>function isPrivileged(privilege);</code>	Returns <code>true</code> if the authenticated user has the specified <code>privilege</code> .
<code>function isUserInRole(role);</code>	Returns <code>true</code> if the user has the specified <code>role</code> .

Method	Description
<code>function getRoles();</code>	Returns the list of all the roles of the user.

Event Auditing

Jacada WorkSpace provides for the auditing of project events. Events are enabled for audit in the following ways:

- Using the Jacada Management Console, administer the pre-defined set of project events. From this set of events, assign (enable) events to be processed by the Jacada auditing manager.
- Project events, other than the Jacada WorkSpace predefined set of events, are explicitly coded to be processed by the Jacada auditing manager.

When the auditing feature is enabled, then database auditing can be enabled and any custom project auditing can be invoked during runtime of the project.

Database Auditing – During a Jacada WorkSpace session, events identified for audit are recorded in the database by the DB Auditor process. For a description of the relevant tables and the event information recorded with database auditing of project events, see the Jacada WorkSpace Administrator Guide.

Custom Project Auditing – During project development, you can override any of the supplied auditing mechanisms, for example, override database auditing of project events by extending the DBAuditor. Also, you can develop new auditors that provide custom auditing treatment of events.

The following sections present:

- [Auditing Data Model](#)
- [Set of Events for Audit](#)
- [Developing Auditing](#)

Auditing Data Model

Jacada WorkSpace provides the following types of data auditing models:

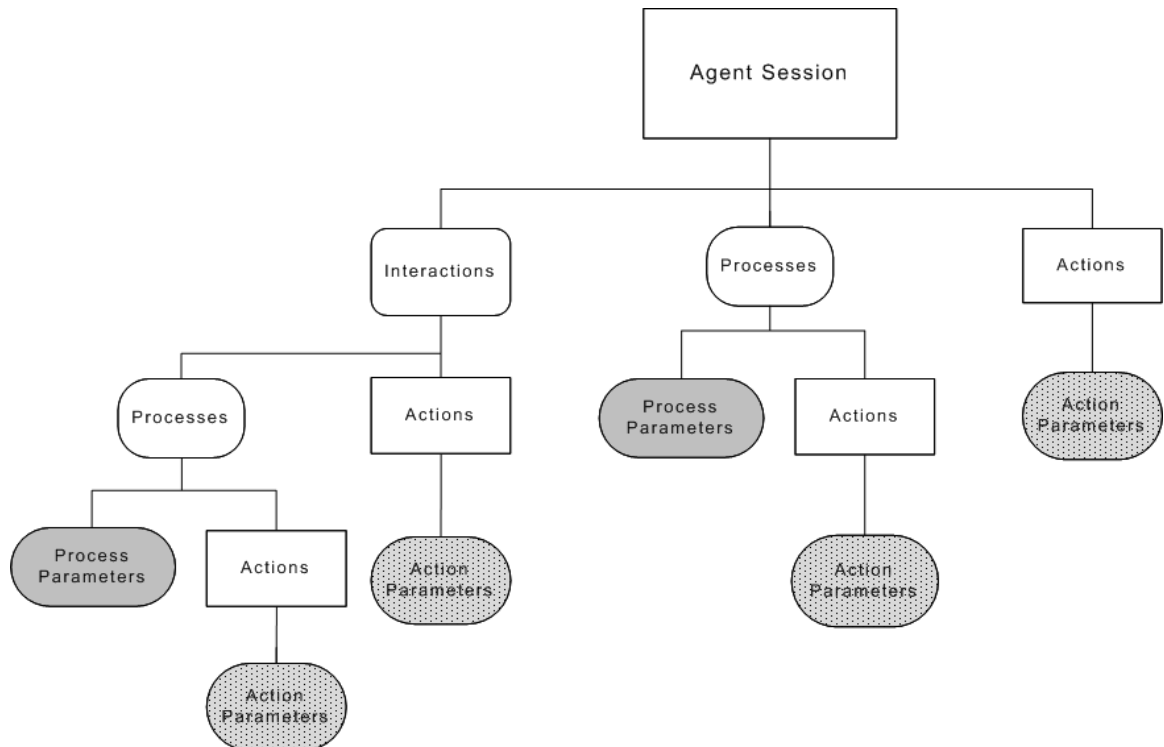
- **Session:** Involves auditing of events that take place during an agent session. For details, refer to [Session Auditing Data Model](#).

- **Application:** Involves auditing of events outside the context of an agent session. For details, refer to [Application Auditing Data Model](#).

Session Auditing Data Model

This model audits events that occur within an agent session. Jacada WorkSpace supplies a set of events that can be enabled for auditing, using the Jacada Management Console. For a list of events, refer to [Set of Events for Audit](#).

The structure of the session auditing data context is diagrammed as follows:



This structure models the auditing-related data objects of a project in production. This structure also represents the database schema that stores audited events, when such events are processed by the DBAuditor.

Types of Audited Entities

The session auditing data model consists of the following types of audited entities:

Agent Session – represents a complete agent session from login through logout.

Interaction – represents specific interactions between an agent and a client that occur, for example, as part of a telephone conversation or a fax correspondence. An interaction occurs within the context of an agent session.

Process – representing initiated Jacada WorkSpace business processes that occur within the context of either an agent session or an interaction. For example, the security check of a client that is performed by an agent at the outset of an interaction.

Process Parameters – parameters providing operational information that are integral to the process with which it is associated. For example, `Chat`, a Jacada WorkSpace-provided event available for audit, is a *Process* event and has the following process parameters: `Initiator` and `Recipient`.

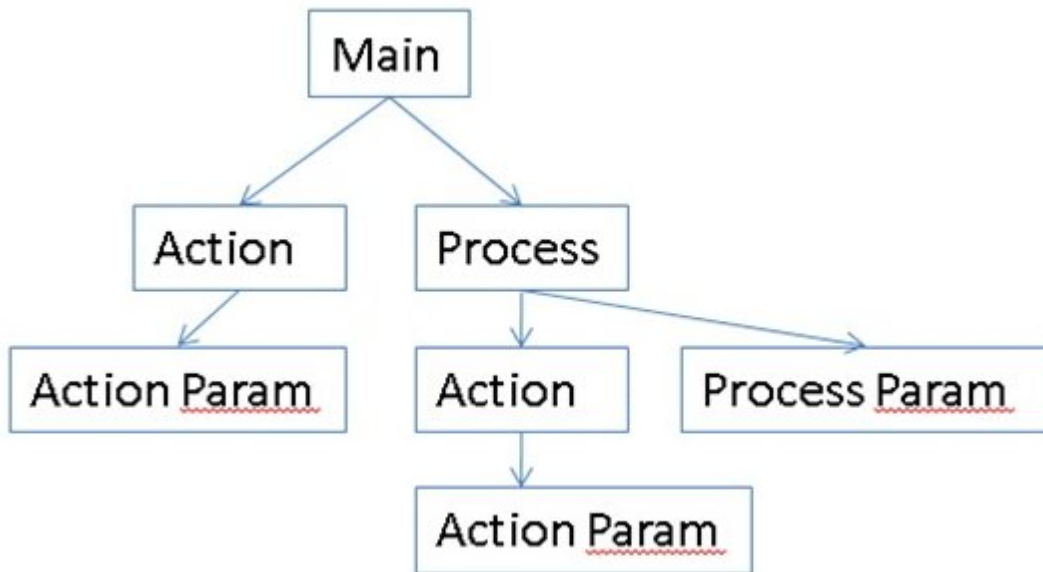
Action – representing operations performed by the agent that occur within the context of either an agent session, an interaction or a process.

Action Parameters – parameters providing operational information that are integral to the action with which it is associated. For example, `warmTransferClicked`, a Jacada WorkSpace-provided event available for audit, is an *Action* event and has the following action parameter: `dn`.

Application Auditing Data Model

In this type of model, the application is deployed in the server, and an agent session is not required for auditing. The application auditing data model is enabled by calling the Audit Manager API: `auditingManager.audit(action);` ([Using the Audit Manager API](#)).

The structure of the application auditing data context is diagrammed as follows:



The session auditing data model consists of the following types of audited entities:

Process – representing initiated Jacada WorkSpace business processes that occur within the context of an application.

Process Parameters – parameters providing operational information that are integral to the process with which it is associated.

Action – representing operations that occur within the context of an application.

Action Parameters – parameters providing operational information that are integral to the action with which it is associated.

Using the Audit Manager API

The Audit Manager is responsible for all auditing operations in the project. Developers use the Audit Manager's API to audit their own processes and actions. This manager is registered as a spring bean, so it is accessible throughout the code. The runtime component holds all the auditors, and delegates the auditing event to all the enabled auditors.

The workflow for using the Audit Manager API involves defining and implementing a class for the *ApplicationAuditingManager* property. Once the class is implemented, you will be able to use the business methods in the *ApplicationAuditingManager*.

To use the Audit Manager API:

1. In your spring configuration file, define your class, and set the *ApplicationAuditingManager* property.

```
<bean id="myBeanId" class="myPackage.MyClass" init-method="init">
    <property name="auditingManager" ref="applicationAuditingManager"/>
</bean>
```

2. Create a new java class (as defined in your spring configuration file), and create a setter method for the *ApplicationAuditingManager* interface.

```
public class MyClass {

    private ApplicationAuditingManager auditingManager;

    public void init(){

        Action action =
auditingManager.getAuditApplication().addAction(pointName);
action.addActionParam("my key" ,"my value");
Process process =
auditingManager.getAuditApplication().addProcess("processName");
        process.addProcessParam("key1", "value1");

        try {
            auditingManager.audit(action);
auditingManager.audit(process);
        } catch (AuditingException e) {
            // handle exception here
        }
    }

    public void setAuditingManager(ApplicationAuditingManager auditingManager) {
        this.auditingManager = auditingManager;
    }
}
```

Set of Events for Audit

Jacada WorkSpace supplies a set of events that can be enabled for auditing, using the Jacada Management Console. Enabled events become auditable by all registered, project auditors, for example, the DBAuditor, and any custom developed auditors.

Set of Events

The following table describes the Jacada WorkSpace set of events:

Event Name	Description	Event Type	Associated Event Parameters
Add Task to BPM	The event generated when Jacada WorkSpace adds a task to an external BPM system.	Action	title, assigneeRoleOrGroup, assigneeUser
After Call Work	The code generated event that moves the agent, currently in Ready status, directly to After Call Work status.	Action	
Answer	The event generated by the agent answering an incoming, business call and auto-answer is not enabled.	Action	
Answer Custom Item	The event generated by the agent answering an incoming, custom item call and custom item auto-answer is not enabled.	Action	
Busy	The event generated when the agent, in CTI inbound mode, requests to move to the Busy status.	Action	

Event Name	Description	Event Type	Associated Event Parameters
Chat	The event generated by the agent or supervisor beginning a chat session. This event contains information that reports about both the beginning and the ending of the chat session.	Process	Initiator, Recipient
Chat Message	The event generated by the agent or supervisor sending a chat message.	Action	MsgContent, MsgSender
Cold Transfer	The event generated by the agent's cold transfer of a call.	Action	dn (dialed number)
Complete Transfer	The event generated by the agent completing the warm transfer of a call.	Action	
Consult	The event generated by the agent moving a call into a two-party consult call, which occurs between either two agents or an agent and a supervisor.	Action	dn (dialed number)
Dial	The event generated by the agent dialing a call.	Action	dn (dialed number)
End Call	The event generated by the agent ending a call.	Action	
End Disposition	The event generated when a call disposition script ends.	Action	

Event Name	Description	Event Type	Associated Event Parameters
End Item	The event generated by the agent ending a custom item call.	Action	
Handshake Transfer	The event generated by the agent's handshake transfer of a call.	Action	dn (dialed number)
Handshake Transfer Connect	(Genesys only) The event generated by the agent's handshake transfer connect of a call.	Action	dn (dialed number)
Hold	The event generated by the agent placing a call on hold.	Action	
Join	The event generated by the agent joining the caller to the consult call, thereby establishing a three-party conference call.	Action	
Load Script	The event generated when either a call handling script loads in the Script tab or an external script, for example a Jacada Interaction Manager script, loads in the workspace area.	Action	ScriptId
Login	The event generated by an agent or a supervisor logging into Jacada WorkSpace.	Process	

Event Name	Description	Event Type	Associated Event Parameters
Logout	The event generated by an agent or a supervisor logging out from Jacada WorkSpace.	Process	Reason
Off Hold	The event generated by the agent removing a call from hold.	Action	
Ready	The event generated when the agent, in CTI inbound mode, requests to move to the Ready status.	Action	
Resynch	(Genesys only) The event generated by the agent re-establishing the connection between a Jacada WorkSpace session and CTI.	Action	
Return	The event generated by the agent returning to the caller, during either a warm transfer of a call or a two-party consult call.	Action	
Start	The event generated by the agent, in manual mode, starting a call.	Action	dn (dialed number)
Start Disposition	The event generated by the starting of the call disposition script.	Action	
Start Outbound	The event generated by the agent starting an outbound call.	Action	

Event Name	Description	Event Type	Associated Event Parameters
Switch Tab	The event generated by the agent switching, while in a business call, between two Jacada WorkSpace tabs.	Action	ToTab (the Tab Id of the tab being switched to)
Transfer From Consult	The event generated by the agent transferring the caller, during a two-party consult call, directly to a supervisor.	Action	
Warm Transfer	The event generated by the agent's warm transfer of a call.	Action	dn (dialed number)
Warm Transfer Connect	(Genesys only) The event generated by the agent's warm transfer connect of a call.	Action	dn (dialed number)

Developing Auditing

The major components for developing auditing code are:

The Auditing Manager: The runtime component responsible for all auditing related processing in a Jacada WorkSpace project. `auditingManager` is one of the registered WorkSpace managers and is accessible throughout project code. It holds all the auditors and dispatches audited events to all enabled auditors, for example the `DBAuditor`, for treatment.

`auditingManager` exposes the following operations:

- Get the current agent session object.

```
public AgentSession getAgentSession();
```

- Get the current `login` and `logout` process objects. In doing so, the `auditingManager` enables adding, via code, process parameters and action objects to the `login` and the `logout` process objects.

```
public Process getLoginProcess();
public Process getLogoutProcess();
```

- Get the current interaction object of any of the following types: inbound, outbound, manual, internal or custom item.

```
public Interaction getCurrentInteraction(InteractionType
interactionType);
```

- Audit one or more than one process and action.

```
public void audit(Process process) throws AuditingException;
public void audit(Action action) throws AuditingException;
```

Each of these audit methods delegates its audited entities to all registered, project auditors/all the auditors that are available to the project. Auditing of a process object does not automatically perform a recursive audit of all of the process object's associated action objects.

Auditors: An auditor is a mechanism that provides specific auditing treatment for a project's audited entities. For example, the Jacada WorkSpace-provided auditor DBAuditor supplies.database auditing of audited events. Custom developed auditor mechanisms implement unique, auditing treatments of audited events.

To develop a project auditor, perform the following in your code:

1. Create a class that implements the Auditor interface.

This interface provides the following methods to audit the audited entities of a project:

Audited Entity	Auditor Interface Method
Agent Session	<code>void audit(AgentSession agentSession) throws AuditingException;</code>
Interaction	<code>void audit(Interaction interaction) throws AuditingException;</code>
Process	<code>void audit(Process process) throws AuditingException;</code>
Action	<code>void audit(Action action) throws AuditingException;</code>

2. Replace existing auditors and add custom auditors by editing the configuration file `WebContent/WEB-INF/applicationContext-user.xml`. All project auditors, including Jacada WorkSpace-

provided auditors, are registered by spring in this configuration file.

In the following example, a custom auditor named `MyProjectAuditor` is added to the `applicationContext-user.xml` file:

```
<bean id="auditors" class="com.jacada.spring.types.MapWrapper">
  <property name="content">
    <map>
      <entry key="DBAuditor" value-ref="dbAuditor"/>
      <entry key="MyProjectAuditor" value-ref="myprojectAuditor"/>
    </map>
  </property>
</bean>
```

To override the processing of an existing auditor, perform the following in your code:

- 1. Extend the relevant Jacada WorkSpace-provided auditor.
- 2. Replace the relevant auditor’s existing value-ref definition by editing the auditor’s bean in the configuration file `WebContent/WEB-INF/applicationContext-user.xml`.

Auditing of the Agent Session Context

When the auditing feature is enabled for a project, upon each user log in to Jacada WorkSpace, the Auditing Manager automatically creates an *Agent Session* entity for that specific Jacada WorkSpace session. The Auditing Manager stores the following information about the *Agent Session* it creates:

Agent Session Information	Description
ID	The ID assigned to this agent session.
AGENT ID	The user name used, by the agent, to log in to a Jacada WorkSpace session.
PROJECT ID	The name of the deployed Jacada WorkSpace project.

Agent Session Information	Description
SESSION ID	The Jacada WorkSpace session ID.

Per agent session, both new *Process* and *Action* type events can be defined in a project. An *Agent Session* entity exposes the following methods to create *Process* and *Action* entities that occur within the context of an agent session:

```
public Process addProcess(String processName)
public Action addAction(String actionName)
```

Auditing of the Interaction Context

When the auditing feature is enabled for a project and an interaction begins between an agent and a caller, the `Auditing Manager` automatically creates an *Interaction* entity of the relevant interaction type. Jacada WorkSpace recognizes the occurrence, during an agent session, of the following interaction types; other types of interaction cannot be created:

- Inbound Call
- Outbound Call
- Manual Call
- Internal Call
- Custom Item

Note: Only one instance of each *Interaction* type can simultaneously occur during an agent session.

Auditing Manager Interaction Information

The Auditing Manager stores the following information about the *Interaction* it creates:

Interaction Information	Description
ID	The ID assigned to this interaction.
INTERACTION TYPE	The type assigned to this interaction. Audited interaction types are: <i>Inbound</i> , <i>Outbound</i> , <i>Manual</i> , <i>Internal</i> and <i>Custom Item</i> .
INTERACTION NUMBER	For interactions of types <i>Inbound</i> , <i>Outbound</i> , <i>Manual</i> and <i>Internal</i> , this number is the telephone number of the call. For interactions of type <i>Custom Item</i> , this number is the Genesys Interaction ID.
START TIME	The date and time when the interaction started, in the format YYYY-MM-DD HH:MM:SS.
END TIME	The date and time when the interaction ended, in the format YYYY-MM-DD HH:MM:SS.
AGENT SESSION ID	The ID assigned to the agent session, which is the parent of this interaction.

Per interaction, both new *Process* and *Action* type events can be defined in a project. An *Interaction* entity exposes the following methods to create *Process* and *Action* entities that occur within the context of an interaction:

```
public Process addProcess(String processName)
public Action addAction(String actionName)
```

Auditing of the Process Context

Within the context of either a current agent session or interaction, processes can be initiated. For each new process, a developer adds a *Process* entity.

Process Information

The Auditing Manager stores the following information about the added *Process*:

Process Information	Description
PROCESS ID	The ID assigned to this process.
PROCESS NAME	The name given to this process.
START TIME	The date and time when the process started, in the format YYYY-MM-DD HH:MM:SS.
END TIME	The date and time when the process ended, in the format YYYY-MM-DD HH:MM:SS.

Per process, any associated process parameters, new *Action* type events or both can be added in a project. A *Process* entity exposes the following methods to a) add the process parameters that are integral to the operation of the specific process and b) add the *Action* entities that can occur within the context of a process:

```
public void addProcessParam(String name, String value)
public Action addAction(String actionName)
```

Process Parameter Information

The Auditing Manager stores the following information about each added process parameter:

Parameter Information	Description
ID	The ID assigned to this process parameter.

Parameter Information	Description
NAME	The name given to this process parameter. For example, Initiator.
VALUE	The information content of the process parameter.
PROCESS ID	The ID assigned to the process, which is the parent of this process parameter.

Auditing of the Action Context

Within the context of either a current agent session, interaction or process, actions can be initiated. For each new action, a developer adds an *Action* entity. The Auditing Manager stores the following information about the added *Action*:

Action Information	Description
ACTION ID	The ID assigned to this action.
ACTION NAME	The name given to this action.
ACTION TIME	The date and time when the action occurred, in the format YYYY-MM-DD HH:MM:SS.

Per action, any associated action parameters can be added in a project. An *Action* entity exposes the following method to add the action parameters that are integral to the operation of the specific action:

```
public void addActionParam(String name, String value)
```

Action Parameter Information

The Auditing Manager stores the following information about each added action parameter:

Action Parameter Information	Description
ID	The ID assigned to this action parameter.
NAME	The name given to this action parameter. For example, dn (dialed number).
VALUE	The information content of the action parameter.
ACTION ID	The ID assigned to the action, which is the parent of this action parameter.

Audit Code Examples

Code Example: single-sign-on

In the following code example, a new action, `single-sign-on`, is created and added to the process `login`. After adding the new action and its action parameters, the new action is audited:

```
AuditingManager auditingManager = getGlobalApp().getAuditingManager();

// Get the login process from the auditing manager:
Process loginProcess = auditingManager.getLoginProcess();

// Add an action to the login process:
Action ssoAction = loginProcess.addAction("single-sign-on");

// Add two new action params to the action:
ssoAction.addActionParam("sap", "ok");
ssoAction.addActionParam("another system", "ok");

try {
    // Audit the new action:
    auditingManager.audit(ssoAction);
}

// Handle exceptions resulting from entity auditing.
// During development, implement your own custom exception handling:
catch (AuditingException e) {
    e.printStackTrace();
}
```

Code Example: securityCheck

In the following code example, a new process, `securityCheck`, is created and added to the current `INBOUND` interaction. After adding the new process, a new action and its action parameter, the new process and then its new action are audited:

```
AuditingManager auditingManager = getGlobalApp().getAuditingManager();

// Get the current INBOUND interaction from the auditing manager:
Interaction inInteraction =
    auditingManager.getCurrentInteraction(InteractionType.INBOUND);

// Create new process under the current inbound call interaction:
Process securityProcess = inInteraction.addProcess("securityCheck");

// To the new process, add the id check action:
Action idCheckAction = securityProcess.addAction("idCheck");
idCheckAction.addActionParam("id", "111111111");

try {
    // Audit the new process:
    auditingManager.audit(securityProcess);

    // Audit the new action (must be audited after auditing the parent
    // process):
    auditingManager.audit(idCheckAction);
}

// Handle exceptions resulting from entity auditing.
// During development, implement your own custom exception handling:
catch (AuditingException e) {
    e.printStackTrace();
}
```

Failover Support

Failover enables work to continue even after part of the architecture, such one of the servers, stops working. Jacada WorkSpace supports failover in production for IBM WebSphere by enabling session replication, whereby Jacada WorkSpace session data, which is stored in the memory of the active application server, is copied by the active server and stored in another application server in the server cluster.

The following sections describe how to implement session replication in a Jacada WorkSpace project:

- [Enable Session Replication in Project Code](#)
- [Re-establishing a JSF Script After a Failure](#)
- [Controlling the Impact of Infrastructure Failures](#)

Note: As session replication may impact performance, Jacada recommends consulting your Jacada representative before implementing session replication.

Enable Session Replication in Project Code

An application server supports failover by replicating the current state of session data. In a clustered server environment, the active application server instance copies its session data to the designated, standby server instance in the cluster. When the active server fails, the replicated HTTP session state on the designated, standby server is used to continue the processing of the agent session.

In order for the application servers to be able to perform session replication of data per Jacada WorkSpace session, incorporate the following in the project code:

- Session data must be **serializable**:
 - Every object stored in the `HTTP` session or `globalApp` must implement the interface

```
java.io.Serializable.
```

- Every member variable of an object stored in the session has to be serializable or be defined as transient.
- When a member variable is defined as transient, the code must provide a mechanism for recreating the variable content when a replication occurs.
- Establish the session data that requires replication to be an attribute of the Jacada WorkSpace session. Use `setAttribute` to change the session state.

In an HTTP servlet that implements `javax.servlet.http.HttpSession`, use `HttpSession.setAttribute` to change attributes in a session object. Using `setAttribute` ensures that the object and its attributes are replicated in a cluster.

For example, add a line of code, such as the one indicated, to ensure replication in a cluster:

```
public class MyUrl {
    public String url;
    ...
    MyUrl myUrl = (MyUrl)getSession().getAttribute("MyUrl");
    myUrl.url = getRequest().getContextPath();//update internal state
    getSession().setAttribute("MyUrl",myUrl);//ensure state replication
    ...
}
```

- Follow the coding guideline: specify the essential minimum of an agent's session data for session replication.

Recoverable Session Data

When session replication is implemented in the Jacada WorkSpace project code, essential, agent session data, that is data that **does not** exist in any other data source, can be recovered following the failover from the active application server host to the standby application server host. A Jacada WorkSpace project must also be equipped to recover its session data that **does** exist in some other data source, when a failover occurs in the product architecture. This type of session data is called *recoverable* session data. Jacada WorkSpace enables the recovery of *recoverable* session data, based on the particular technology used to develop your projects.

Data Recovery with Spring

If your project is developed using Spring technology, implement the following in your project code to enable the recovery of *recoverable* session data.

In any of the Spring definition files `/WebContent/WEB-INF/applicationContext-
<Workspace_componentarea_name>.xml`, any class that uses *session context*, must do the

following:

- Implement the serializable interface.
- Include the `aop:scoped-proxy` definition in the bean definition of the class. For example:

```
<bean id="sessionInfo" class="com.jacada.service.beans.SessionInfo"
      scope="session">
  <aop:scoped-proxy/>
</bean>
```

Data Recovery with Beehive

Jacada WorkSpace supports backward compatibility for project code developed using the Beehive technology regarding enabling the recovery of *recoverable* session data. If your project is developed using the Beehive technology, implement in each controller the following:

Include the class member:

```
protected transient SharedData globalApp;
```

Deploy the functions:

```
public SharedData getGlobalApp() {
    if (globalApp == null) {
        globalApp = (SYSTEM.global.SharedData)
            PageFlowUtils.getSharedFlow(SYSTEM.global.SharedData.class.getName(),
                this.getRequest(), getRequest().getSession().getServletContext());
    }
    return globalApp;
}

protected synchronized void beforeAction() throws Exception {
    super.beforeAction();
    getGlobalApp().failoverRecovery(getRequest(), getResponse());
    jFAPDataBase = getGlobalApp().getJFAPDataBase();
}
```

Re-establishing a JSF Script After a Failure

JSF components of a script are not serializable and therefore are not replicated in a failover situation. To ensure that the script continues, that is make the JSF state persist, add the following parameter assignment in your project's `WebContent/WEB-INF/web.xml` file:

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

The `javax.faces.STATE_SAVING_METHOD` parameter instructs an application server about where it must save the JSF state. The default is to save the JSF state on the server. However, for failover support purposes, you need the JSF state saved on the client.

Controlling the Impact of Infrastructure Failures

To control the impact of part of the system failing, you need to re-establish push messaging between Jacada WorkSpace sessions and Jacada WorkSpace on the server when the Big-IP Local Traffic Management (LTM) system reroute traffic to a functioning application server, following the detection of a loss of communications with another application server.

Both a client-side setting and a server-side timeout setting exist to control timeout for push messages. To re-establish push messaging, reduce both settings.

The server-side setting is modified using the Jacada Management Console. See the *Jacada WorkSpace Administrator Guide* for the procedure required to reduce the server-side timeout setting for push messages.

To reduce the client-side timeout setting for push messages:

1. In your project development environment, open the `WebContent/SYSTEM/resources/js/push.js` file.
2. In the `push.js` file, locate the following line of code:

```
this.currentTimer = setTimeout('Push._abort();',180000);
```

3. Modify the timeout value provided to `setTimeout()` from 180000 milliseconds (3 minutes) to 31000 milliseconds (31 seconds).

4. Save the change.

The project's client-side timeout setting for push messages is updated with value you provided to the `setTimeout()` method.

5. Rebuild your project and then republish it to the application server.

Localization

Jacada WorkSpace provides support for localization. Localization is the capability to provide multiple language display options for the deployed project's Jacada WorkSpace.

The developer and the administrator determine the extent of localization in use in Jacada WorkSpace. The language in use affects the entire Jacada WorkSpace, for example, menu items, portlet names, tab names, the CTI-handling bar, displayed UI messages, call-flow scripts and dialogs such as the welcome dialog and the login screen.

The following sections describe:

- [Setting Up Localization](#)
- [Localization Functions](#)

Setting Up Localization

Jacada uses localization property files in which you specify the actual text to display in Jacada WorkSpace. English is the default language of Jacada WorkSpace and Jacada WorkSpace is provided with an English localization property file. To specify text in other languages, you create language specific localization property files.

The default language for Jacada WorkSpace is defined in the *Default Locale* field, which is available in the *Global Settings* page of the Jacada Management Console.

To define a language to be used:

1. Copy the `WebContent/USER/resources/resourceBundles/UI_Texts_en.properties` file and change the Java Locale code to the Java Locale code of language you want.
For example, for French, rename the copied file to `WebContent/USER/resources/resourceBundles/UI_Texts_fr.properties`.
2. In the new localization property file, provide the keys and their associated language specific text that Jacada WorkSpace must display. It is recommended to copy the relevant keys from

`WebContent/SYSTEM/resources/resourceBundles/UI_Texts_en.properties` and translate the text associated with each key as appropriate.

Note: Any key defined in a language specific localization property file located under the `USER` directory overrides that same key definition in the parallel, localization property file located under the `SYSTEM` directory.

Separate groups of entries in the `UI_Texts_<language>.properties` file by inserting a line before each group. Starts this line with a hash (`#`) symbol.

3. In the Jacada Management Console navigation menu, select the *Configuration > Global Settings* option.
The *Global Settings* page displays.
4. In the `Default Locale` field, enter the necessary Java Locale code and click `Apply`.
5. Where appropriate, change any graphics to be localized. For example, the *Chat History* tab includes graphics that include text that can be replaced with graphics including localized text. Changing a graphic might also require overriding a css class: in the case of the *Chat History* tab, the `.chatSessionsImage` needs overriding in `USER/resources/css/override.css`:

```
.chatSessionsImage {  
    background-image: url("images/ChatSessions.gif");  
    background-repeat:no-repeat;  
    display:inline;  
    background-position:0 60px;  
}
```

Localization Functions

Jacada Workspace provides functions to return localized text. You can retrieve localized text both for Jacada Workspace and call-handling scripts. Thus, Jacada Workspace can use one language while the call-handling script can be in another language.

Java Methods

Jacada WorkSpace provides the following Java language methods:

`LocaleManager.getApplicationLocalizationValue(<key_name>)` – Gets the text to use in Jacada WorkSpace based on the language key supplied.

`LocaleManager.getScriptLocalizationValue(<key_name>)` – Gets the text to use in scripts based on the language key supplied.

If the language is not defined, `???<key_name>???` is displayed instead of the text.

JSP Pages

Use the following, standard JSP localization tags:

- `<fmt:message bundle="{applicationMessages}" key="{key}">`

This JSP localization tag gets the text to use in portlets, based on the provided text key. If this key is not defined, `???<key_name>???` is displayed instead of the text.

- `<fmt:message bundle="{messages}" key="{key}"/>`

This JSP localization tag gets the text to use in call-handling scripts, based on the provided text key. If this key is not defined, `???<key_name>???` is displayed instead of the text.

To use this `fmt` tag within a call-handling script, include one of the following at the beginning of the script page:

- Incorporate the following include declaration:

```
<%@ include file = "../resources/jspf/includes/scriptHeader.jspf" %>
```

Jacada recommends using this include declaration.

- Incorporate the following code:

```
<%
    ApplicationContext applicationContext =
        WebApplicationContextUtils.getWebApplicationContext(application);
    LocaleManager localeManager =
        (LocaleManager)applicationContext.getBean("localeManager");
    String locale = localeManager.getScriptLocale();
%>
<fmt:setLocale value="{%=locale%}"/>
<fmt:setBundle basename="{%=localeManager.getResourceBundleBaseName()%}"
    var="messages"/>
```

JavaScript Functions

Jacada WorkSpace provides the following JavaScript language functions:

- `Var text_value = $W().localeManager.getLocalizationValue(<key_name>)` – Gets the text to use in portlets based on the language key supplied. The `LocaleManager` object is loaded when the portlet is loaded.
- `$W().scriptLocaleManager.getLocalizationValue(<key_name>)` – Gets the text to use in call-handling scripts based on the language key supplied. The `scriptLocaleManager` object is loaded when the script is loaded.

Note: To access a localized value in the client side, using the JavaScript language functions, the value's key must start with either one of the following:

- `application.javascript` for portal level keys.
- `script.javascript` for script level keys.

OK, *Cancel* and *Browse* buttons in JavaScript popup, alert, confirm and prompt dialog boxes are browser buttons and cannot be localized.

If the language is not defined, `???<key_name>???` is displayed instead of the text.

Preparing a Project for Production

The following sections explain how to deploy a project:

- [Deploying to an IBM WebSphere/Oracle WebLogic Application Server](#)
- [Deploying to an Apache Tomcat Server](#)

Deploying to an IBM WebSphere/Oracle WebLogic Application Server

To deploy a project to an IBM WebSphere application server you must first prepare the project for the application server and then either deploy the project directly to the application server, or export the EAR file to be deployed at a later date.

To prepare a project for an IBM WebSphere/Oracle WebLogic application server:

- Right-click on the project, and select *Jacada WorkSpace > Configure Project for WebSphere/ WebLogic Runtime*.

Exporting a Project EAR File From the IDE for Production

Instead of directly deploying the project to the application server, you can export the EAR file to be deployed at a later date.

To export a project EAR file:

1. In the IDE, right-click on the *<project_name>EAR* you want to export.
2. In the displayed menu, select the *Export > Export....*
3. In the opened Export dialog, select an export destination. Valid selections are either of the following:

- a. The *Other* folder > the *EAR file with JSP pre-compilation* option.
 - b. The *J2EE* folder > the *EAR file* option.
4. Click *Next*.
5. In the *Destination* field of the opened EAR Export dialog, enter the destination to export the EAR file.

The destination can be either a location in a local or a networked file system. The field contains the complete path to the file to be exported, with the EAR file in the form
`<project_nameEAR>.ear`.
6. Click *Finish*.

The EAR file, `<project_nameEAR>.ear`, is exported to the specified location for deployment in a production environment. For details about deploying an application to a production server, see the Jacada WorkSpace Administrator Guide.

Deploying to an Apache Tomcat Server

To deploy a project to an Apache Tomcat server, you must first prepare the project for the server, and then either deploy the project directly to the server (using Eclipse), or export the WAR file to be deployed at a later date.

To prepare a project for an Apache Tomcat server:

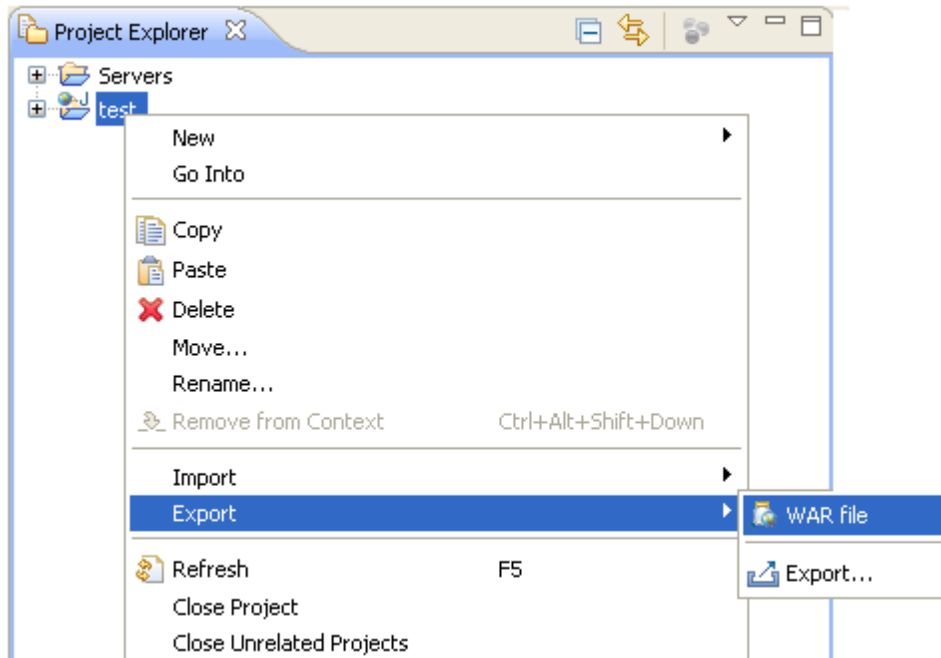
- Right-click on the project, and select *Jacada WorkSpace > Configure Project for Tomcat Runtime*.

Exporting a Project WAR File From the IDE for Production

Instead of directly deploying the project to the application server, you can export the WAR file to be deployed at a later date.

To export a project WAR file:

1. In the IDE, right-click on the `<project_name>` you want to export.
2. In the displayed menu, select the *Export > Export...*



3. Click *Next*.

4. In the *Destination* field of the opened WAR Export dialog, enter the destination to export the WAR file.

The destination can be either a location in a local or a networked file system. The field contains the complete path to the file to be exported, with the WAR file in the form `<project_name>.war`.

5. Click *Finish*.

The WAR file, `<project_name>.war`, is exported to the specified location for deployment in a production environment. For details about deploying an application to a production server, see the Jacada WorkSpace Administrator Guide.