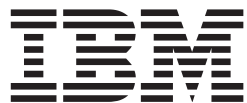
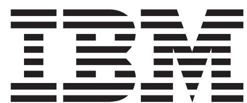
Sterling Selling and Fulfillment Foundation



Customizing the Rich Client Platform Interface

Version 9.1.0.4.0

Sterling Selling and Fulfillment Foundation



Customizing the Rich Client Platform Interface

Version 9.1.0.4.0

Note

Before using this information and the product it supports, read the information in "Notices" on page 201.

Copyright

This edition applies to the 9.1 Version of IBM Sterling Selling and Fulfillment Foundation and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1999, 2013.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Checklist for Customization Projects . . . 1
Customization Projects . . . 1
Prepare Your Development Environment . . . 1
Plan Your Customizations . . . 1
Extend the Database . . . 1
Make Other Changes to APIs . . . 2
Customize the UI . . . 2
Extend Transactions . . . 3
Build and Deploy your Customizations or Extensions 3

Chapter 2. The Rich Client Platform . . . 5
About Customizing the Rich Client Platform Interface 5
Rich Client Platform Architecture . . . 6
Benefits of Using the Rich Client Platform Interface . 7
Rich Client Platform and Desktop Applications . . . 8
XML Binding for Rich Client Platform Applications . 8
Localizing Rich Client Platform Applications . . . 9
Themes for Rich Client Platform Applications . . . 9
Related Tasks for Rich Client Platform Applications . 9
Shared Tasks for Rich Client Platform Applications 10
Navigator Tasks for Rich Client Platform Applications . . . 10
Wizards for Rich Client Platform Applications . . . 10
Hot Keys for Rich Client Platform Applications . . 13
Debug Mode for Rich Client Platform Applications 13
Running Rich Client Platform Applications in Debug Mode . . . 14
Running the Standalone Rich Client Platform Application in Debug Mode . . . 14
Running the Rich Client Platform Application in Eclipse in Debug Mode . . . 15
Prototype Mode for Rich Client Platform Applications . . . 15
Running Standalone Rich Client Platform Applications in Prototype Mode . . . 15
Running Rich Client Platform Applications in Eclipse in Prototype Mode . . . 16
Tracing a Rich Client Platform Application . . . 16
Tracing a Standalone Rich Client Platform Application . . . 17
Tracing a Rich Client Platform Application in Eclipse . . . 18
Masking Sensitive Information During Trace . . 18
Capitalizing the Text Entered in Rich Client Platform Applications . . . 19
Fetching Images for Rich Client Platform Applications . . . 20
Security Handling for Rich Client Platform Applications . . . 20
Output Templates for Rich Client Platform Applications . . . 20
Commands for Rich Client Platform Applications. . 20
Log Files for Rich Client Platform Applications . . 21
Masking Sensitive Information During Logging 21
Data Caching for Rich Client Platform Applications 22
Error Handling for Rich Client Platform Applications . . . 22
Table Filtering for Rich Client Platform Applications 23
Clearing the Sort Order in a Table . . . 23
Scheduling Jobs for Rich Client Platform Applications . . . 24
Scheduling a Generic Job . . . 24
Scheduling an Alert-Related Job . . . 24
Preventing the Deactivation of Alert Notification 25
Audio Files for Rich Client Platform Applications . 26
Low Resolution Display for Rich Client Platform Applications . . . 26
Displaying Panel Tasks on the Menu Bar for Rich Client Platform Applications. . . 28
Switching Locales for Rich Client Platform Applications . . . 29
Using a VM Login for Rich Client Platform Applications . . . 29
Using a VM JRE for Rich Client Platform Applications . . . 29
Supervisory Overrides for Rich Client Platform Applications . . . 30
Using the Pop-Up Method to Perform Supervisory Overrides. . . 30
Starting a Supervisory Transaction to Perform Supervisory Overrides. . . 31
Running Rich Client Platform Applications in POS Mode . . . 31
Version-Based Communication between Client and Server . . . 32
Integrating Web Applications with Rich Client Platform . . . 33
Create an Extension . . . 34

Chapter 3. The Development Environment for Rich Client Platform Applications . . . 37
Installing Prerequisite Software Components . . . 37
Clean Cached Build Information in Eclipse . . . 37
Installing the Rich Client Platform Plug-In . . . 38
Installing the Rich Client Platform Tools Plug-In 38
Rich Client Platform Tools . . . 38
Creating and Configuring Locations . . . 40
Creating a Plug-In Project . . . 41
Rich Client Platform Plug-In Wizard . . . 42
Running the Rich Client Platform Plug-In Wizard 43
Launching the Rich Client Platform Application in Eclipse . . . 45

Chapter 4. Customizing the Log In Screen . . . 49
Customizing the Login Screen . . . 49

© Copyright IBM Corp. 1999, 2013

Chapter 5. Customizing Rich Client Platform Applications 51
Overview of Customizing Rich Client Platform Applications 51
Localizing Rich Client Platform Applications . . . 51
Defining Themes for Rich Client Platform Applications 51
Extending Rich Client Platform Applications . . . 51
Building and Deploying Extended Rich Client Platform Applications 52
Building Rich Client Platform Extensions 52
Deploying Rich Client Platform Extensions 54

Chapter 6. Customizing the About Box 55
Customizing the About Box 55

Chapter 7. Masking Sensitive Customer Information 57
Methods for Masking Sensitive Customer Information 57

Chapter 8. Modifying Existing Screens and Wizards 59
Modifying Existing Rich Client Platform Screens . . 59
Validating or Capturing Data During API or Service Calls. 59
Modifying Existing Rich Client Platform Wizards. . 60
Retrieve Wizard and Namespace Information . . . 62
Creating an Extended Wizard Definition. 62
Registering the Wizard Extension File 63
Creating the Wizard Entity 63
Modifying the Wizard Extension Behavior 63

Chapter 9. Creating and Adding Screens 65
About Creating a Rich Client Platform Composite 65
Creating a Rich Client Platform Composite Using the Rich Client Platform Composite Wizard . . . 65
About Designing a Rich Client Platform Composite 66
Creating the Search Criteria Panel for a Rich Client Platform Composite 67
Adding Controls to the Search Criteria Panel for a Rich Client Platform Composite 69
Creating the Search Result Panel for a Rich Client Platform Composite 71
Displaying Paginated Results in a Rich Client Platform Composite 72
Creating Tables for Rich Client Platform Screens . . 74
  Creating Standard Tables 74
  Adding Columns to the Standard Table 75
  Creating Editable Tables 75
Naming Controls for Rich Client Platform Screens 76
  Creating a Binding Object 76
  Naming a Control 76
Setting Data On Controls for Rich Client Platform Screens 76
Binding Controls and Classes for Rich Client Platform Screens. 77
Source Binding for Controls on Rich Client Platform Screens 77
  Multiple Source Bindings 78
Target Binding for Controls on Rich Client Platform Screens 79
Checked Binding for Controls on Rich Client Platform Screens. 80
Unchecked Binding for Controls on Rich Client Platform Screens. 80
List Binding for Controls on Rich Client Platform Screens 81
Code Binding for Controls on Rich Client Platform Screens 81
Description Binding for Controls on Rich Client Platform Screens. 81
Attribute Binding for Controls on Rich Client Platform Screens. 82
Key Binding for Controls on Rich Client Platform Screens 82
Binding Input to Custom Controls on Rich Client Platform Screens. 83
About Setting Bindings for Controls on Rich Client Platform Screens. 83
Creating a Binding Object for a Label. 84
  Bind a Label 84
Creating a Binding Object for Text Boxes 85
  Bind a Text Box 85
Creating a Binding Object for StyledText Components 86
  Bind a StyledText Component 86
Creating a Binding Object for Combo Boxes . . . 87
  Bind a Combo Box 87
Version-Specific Data in Combo Boxes 88
  Populating Version-Specific Data in Combo Boxes 88
Creating a Binding Object for List Boxes. 89
  Bind a List Box 89
Creating a Binding Object for Checkboxes 90
  Bind a Check Box 90
Creating a Binding Object for Radio Buttons . . . 91
  Bind a Radio Button 91
Creating a Binding Object for Links 92
  Bind a Link 92
Creating a Binding Object for a Standard Table . . 93
  Creating a Binding Object for a Column 93
  Bind a Standard Table and Column 93
Setting Bindings for an Editable Table 96
  Binding Combo Box Cell Editors 96
Setting Bindings for an Extended Table 97
  Creating a Binding Object for an Extended Table 97
  Create a Map of the Advanced Column Binding Data 97
  Bind an Extended Table and Advanced Column 98
Setting Bindings for Extended Editable Tables . . 100
  Binding Combo Box Cell Editors 101
Creating a Binding Object for a File Upload Column in a Table in the Rich Client Platform . . 102
Creating a Binding Object for a File Upload Text Box in the Rich Client Platform 103
Localizing Controls and Defining Themes for Rich Client Platform Applications 105
  Defining Themes for Controls 105

Calling APIs and Services for Rich Client Platform Applications . . . 105
Calling the Same API/Service Multiple Times 106
Calling Multiple APIs/Services . . . 107
Adding New Rich Client Platform Screens as Pop-ups . . . 108
Adding New Rich Client Platform Screens to Menu Commands . . . 109
Displaying New Rich Client Platform Screens in an Editor . . . 109

Chapter 10. Configuring File Uploads and Downloads . . . 113
Uploading and Downloading . . . 113
Using yfs.properties to Configure File Uploads . . 113
Configuring File Uploads . . . 114
Securing Uploaded Files . . . 116
Upload Error Messages . . . 117
Configuring File Downloads . . . 118
Securing Downloaded Files . . . 119
Download Error Messages . . . 120
Structuring the File Upload and Download . . . 121
Uploading and Downloading Using Interface Contracts without the Sterling Application Platform . . . 124

Chapter 11. Creating and Adding Wizards . . . 127
Phase 1: Create Wizard Definitions . . . 127
Creating a Wizard Definition with the Rich Client Platform Wizard Editor . . . 127
Phase 2: Create Components to Implement a Wizard Definition . . . 128
Creating Wizard Class . . . 128
Creating Wizard Behavior Class . . . 130
Phase 3 Adding Components to Wizard Definition 132
Adding a Rule to a Wizard Definition . . . 132
Adding a Page to a Wizard Definition . . . 133
Adding a Sub-task to a Wizard Definition . . . 134
Adding a Transition to a Wizard Definition . . . 134
Creating Wizard Page Components . . . 135
Creating Wizard Page Class . . . 135
Creating Wizard Page Behavior Class . . . 137
Creating Wizard Rule Components . . . 139
Registering the Wizard Command File . . . 141
Adding Wizards as Pop-ups in Rich Client Platform Applications . . . 141
Adding Wizards to Menu Commands in Rich Client Platform Applications . . . 141
Adding Wizards to Editors in Rich Client Platform Applications . . . 142

Chapter 12. Creating Related Tasks 145
About Related Tasks . . . 145
Extending the YRCRelatedTasks Extension Point 145
Extending the YRCRelatedTaskCategories Extension Point . . . 147
Extending the YRCRelatedTaskGroups Extension Point . . . 148
Extending the YRCRelatedTasksDisplayer Extension Point . . . 149
Access Editor Information . . . 150
Extending the YRCRelatedTasksExtensionContributor Extension Point . . . 150
Enabling Custom Dialog Boxes Through an Extension Point for Rich Client Platform Applications . . . 152

Chapter 13. Creating Commands . . . 153
About Commands . . . 153
Defining Namespaces . . . 154
Overriding Commands . . . 155

Chapter 14. Defining and Overriding Hot Keys . . . 157
Phase 1: Defining a Hot Key Command . . . 157
Phase 2: Defining a Hot Key Binding . . . 158
Phase 3: Defining a Hot Key Action . . . 159
Overriding Hot Keys . . . 159
Disabling Related Task Hot Keys . . . 160

Chapter 15. Merging Templates . . . 163
Merging Input and Output Templates . . . 163

Chapter 16. Related and Shared Tasks 165
Adding New Related Tasks . . . 165
Hiding Existing Related Tasks . . . 165
Registering Shared Tasks . . . 165
Using Shared Tasks . . . 167

Chapter 17. Defining Themes . . . 169
Defining New Themes . . . 169
Defining Themes for Controls . . . 170
Applying Themes to Non-editable Text Boxes 171

Chapter 18. Menus and Custom Controls . . . 173
Adding and Removing Menus in Rich Client Platform Applications . . . 173
Customizing the Menu View Through the YRCMenuDisplayer Extension Point . . . 173
Creating Custom Controls for Rich Client Platform Applications . . . 174
Extending the YRCCustomControl Extension Point . . . 174
Using Custom Controls in RCP Applications . . . 175

Chapter 19. Setting the Extension Model and Configuring SSL and SSO . 177
Setting the Extension Model for Rich Client Platform Applications . . . 177
Configuring SSL for Rich Client Platform Applications . . . 177
Configuring SSO for Rich Client Platform Applications . . . 178
Client Settings for SSO Configuration . . . 178
Server Settings for SSO Configuration . . . 179

Chapter 20. General Concepts Reference 181
Rich Client Platform Architecture. 181
Eclipse and its Rich Client Platform 182
Workbench 183
Plug-In Manifest Editor 183
YRCPluginAutoLoader Extension Point. 184
YRCAutoUpdateExtn Extension Point 184
YRCApplicationInitializer Extension Point. . . . 185
YRCContainerToolbar Extension Point 186
YRCPostWindowOpenInitializer Extension Point 187
YRCJasperReport Extension Point 188
YRCContainerTitleProvider Extension Point . . . 188
YRCMessageDisplayer Extension Point 189
Creating New Actions 190
Registering a Plug-In 192
Registering Plug-In Files. 193
Validating Controls 195
Custom Data Formatting 195
Siblings 198
Rich Client Platform Utilities 198
  Viewing Screen Models 198
  Saving Models as Templates 199

Notices 201

Chapter 1. Checklist for Customization Projects

Customization Projects

Projects to customize or extend Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales vary with the type of changes that are needed. However, most projects involve an interconnected series of changes that are best carried out in a particular order. The checklist identifies the most common order of customization tasks and indicates which guide in the documentation set provides details about each stage.

The items identified for extension and/or modification in the documentation are Source Components (to the extent such item involves source code) and Sample Materials for purposes of the License Information file associated with this product.

Prepare Your Development Environment

Set up a development environment that mirrors your production environment, including whether you deploy your application on a WebLogic, WebSphere®, or JBoss application server. Doing so ensures that you can test your extensions in a real-time environment.

You install and deploy your application in your development environment following the same steps that you used to install and deploy it in your production environment. Refer to your system requirements and installation documentation for details.

You have an option to customize your application with Microsoft COM+. Using Microsoft COM+ has advantages such as increased security, better performance, increased manageability of server applications, and support for clients of mixed environments. If this is your choice, see the *Customization Basics Guide* about additional installation instructions.

Plan Your Customizations

Are you adding a new menu entry? Or customizing the sign-in screen or logo? Or customizing views or wizards? Or creating new themes or new screens? Each type of customization varies in scope and complexity.

For background, see the *Customization Basics Guide*, which summarizes the types of changes that you can make and provides important guidelines about file names, keywords, and other general conventions.

Extend the Database

For many customization projects, the first task is to extend the database so that it supports the other UI or API changes that you make later. For instructions, see the *Extending the Database Guide*, which includes information about the following topics:

- Important guidelines about what you can and cannot change in the database.

© Copyright IBM Corp. 1999, 2013

- Information about modifying APIs. If you modify database tables so that any APIs are impacted, you must extend the templates of those APIs or you cannot store or retrieve data from the database. This step is required if table modifications impact an API.
- How to generate audit references so that you improve record management by tracking records at the entity level. This step is optional.

Make Other Changes to APIs

Your application can call or invoke standard APIs or custom APIs. For background about APIs and the services architecture of service types, behavior, and security, see the *Customizing APIs Guide*. This guide includes information about the following types of changes:

- Invoke standard APIs for displaying data in the UI and for saving changes made in the UI to the database.
- Invoke customized APIs for executing your custom logic in the extended service definitions and pipeline configurations.
- APIs use input and output XML to store and retrieve data from the database. If you don't extend these API input and output XML files, you may not get the results you want in the UI when your business logic is executing.
- Every API input and output XML file has a DTD and XSD associated to it. Whenever you modify input and output XML, you must generate the corresponding DTD and XSD to ensure data integrity. If you don't generate the DTD and XSD for extended XMLs, you may get inconsistent data.

Customize the UI

IBM® applications support several UI frameworks. Depending on your application and the customizations you want to make, you may work in only one or in several of these frameworks. Each framework has its own process for customizing components such as menu items, logos, themes, and so on.

Depending on the framework you want, consult one of the following guides:

- *Customizing the Console JSP Interface Guide*
- *Customizing the Swing Interface Guide*
- *Customizing User Interfaces for Mobile Devices Guide*
- *Customizing the Rich Client Platform Guide* and *Using the RCP Extensibility Tool Guide*
- *Customizing the Web UI Framework Guide*

Depending on the framework you want, consult one of the following guides:

- *Customizing the Console JSP Interface Guide*
- *Customizing the Swing Interface Guide*
- *Customizing User Interfaces for Mobile Devices Guide*
- *Customizing the Rich Client Platform Guide* and *Using the RCP Extensibility Tool Guide*
- *Customizing the Web UI Framework Guide*

Extend Transactions

You can extend and enhance the standard functionality of your application by extending the Condition Builder and by integrating with external systems. For background about transaction types, security, dynamic variables, and extending the Condition Builder, see the *Extending Transactions Guide* and *Extending the Condition Builder Guide*. These guides includes information about the following types of changes:

- Extend the Condition Builder to define complex and dynamic conditions for executing your custom business logic and using a static set of attributes.
- Define variables to dynamically configure properties belonging to actions, agents, and services configurations.
- Set up transactional data security for controlling who has access to what data, how much they can see, and what they can do with it.
- Create custom time-triggered transactions. You can invoke and schedule custom time-triggered transactions in much the same manner as you invoke and schedule the time-triggered transactions supplied by your application.
- Coordinate your custom, time-triggered transactions with external transactions and run them either by raising an event, calling a user exit, or invoking a custom API or service.

Build and Deploy your Customizations or Extensions

After performing the customizations that you want, you must build and deploy your customizations or extensions.

1. Build and deploy your customizations or extensions in the test environment so you can verify them.
2. When you are ready, repeat the same process to build and deploy your customizations and extensions in your production environment.

For instructions about this process, see the *Customization Basics Guide* which includes information about the following topics:

- Building and deploying standard resources, database extensions, and other extensions (such as templates, user exits, and Java™ interfaces).
- Building and deploying enterprise-level extensions.

Chapter 2. The Rich Client Platform

About Customizing the Rich Client Platform Interface

This section explains the prerequisites for customizing the Rich Client Platform application UIs easily, quickly, and with fewer errors.

Rich Client Platform Concepts

Before customizing the Rich Client Platform application UIs, it is important that you understand the various concepts of the Rich Client Platform.

Extensibility Capability Summary

Before extending any Rich Client Platform application UI, it is necessary to understand the extensibility capabilities provided by the Rich Client Platform.

Guidelines for Smooth Updates and Easy Maintenance

When customizing applications that use the Rich Client Platform UI, do not modify:

- Plug-in files
- Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales-related resource files
- JAR files

These files are shipped as part of the standard default configuration. You can, however, create new files or copy the existing files and modify them.

Setting Up the Development Environment

To customize applications, set up the development environment to accommodate modifications that you make to the Rich Client Platform application UI. For more information about setting up the development environment, see "The Development Environment for Rich Client Platform Applications".

Extending Rich Client Platform Applications

The Rich Client Platform provides various extension points that you can implement to extend the Rich Client Platform application as needed. Using features such as Localization and Theming, you can further extend the Rich Client Platform application. The Rich Client Platform also provides a Rich Client Platform Extensibility tool with which you can extend the Rich Client Platform application's UI.

You can extend the Rich Client Platform application by:

- Adding or Removing Menus—You can add or remove menus from the Rich Client Platform screens by defining a new resource in the resources of Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales . For more information about adding or removing menus, see "Menus and Custom Controls for Rich Client Platform Applications".

© Copyright IBM Corp. 1999, 2013

- Creating and Adding New Screens—You can create new Rich Client Platform screens for a Rich Client Platform application. You can also add the newly created Rich Client Platform screens to a Rich Client Platform application. For more information about creating and adding new screens, see "Creating and Adding New Screens".
- Adding New Related Tasks and Hiding Existing Related Tasks—You can add new related tasks and hide existing related tasks from the Rich Client Platform applications. For more information about adding and hiding related tasks, see "Creating Related Tasks for Rich Client Platform Applications".
- Modifying Existing Screens—You can modify existing screens of a Rich Client Platform application using the Rich Client Platform Extensibility Tool. For information about modifying existing screens, see "Modifying the Existing Rich Client Platform Screens and Wizards".
- Modifying Existing Wizards—You can modify the existing wizards of a Rich Client Platform application. For more information about modifying the existing wizards, see "Modifying Existing Rich Client Platform Wizards".
- Localizing—You can localize the Rich Client Platform application for different languages based on the user's locale. The user can localize the Rich Client Platform application by defining locale-specific entries and translating the text. For more information about localizing the Rich Client Platform application, see "Customizing Rich Client Platform Application".
- Theming—You can customize the Rich Client Platform application by using custom themes. You can change the font type and color scheme for controls, graphical text, messages, and so forth. For more information about theming, see "Defining Themes for Rich Client Platform Applications".

Rich Client Platform Architecture

The Rich Client Platform provides a highly interactive Rich Client Platform, which can be remotely deployed, updated, and easily managed. A Rich Client Platform is a client that processes the bulk of data operations without depending on the server to which it is connected. However, it is dependent on the server, primarily for data storage. The Rich Client Platform is rich in features and functionality and has complete access to the programming functions of the operating system.

Rich Clients are designed in such a way that you can work over low bandwidth network connections and still efficiently utilize the client-side capabilities to avoid costly round trips to the central server. You can also work offline.

The Rich Client Platform is built on the Eclipse Rich Client Platform. The Rich Client Platform has extended the Eclipse Rich Client Platform to provide additional features and functionality. In addition to the features provided by the Eclipse Rich Client Platform, the Rich Client Platform provides features such as localizing, theming, binding, and so forth. The UI forSterling Call Center and Sterling Store application is developed using the Rich Client Platform. The following figure depicts the Rich Client Platform architecture.

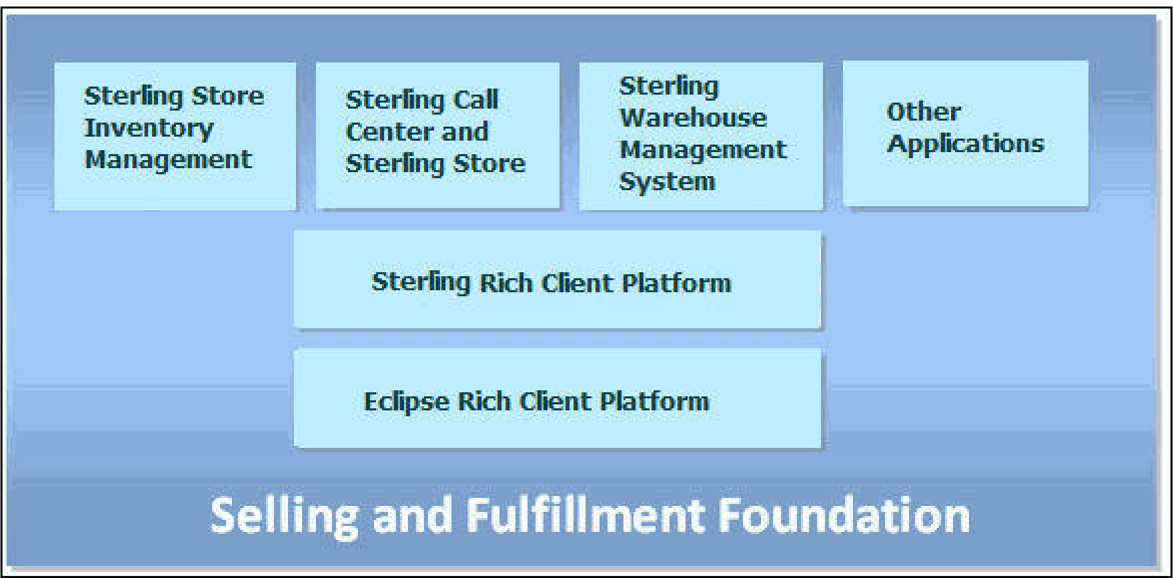


Figure 1. Rich Client Platform Architecture

Benefits of Using the Rich Client Platform Interface

The benefits of using the Rich Client Platform are:

- Rich User Experience
  - High responsiveness during information retrieval. This does not work in a "page-at-a-time" paradigm of Hyper Text Markup Language (HTML).
  - Ability to validate client side data, if needed.
  - Highly interactive and graphical user interface.
  - Provides visibility to multiple pages on the screen without refreshing any page.
  - Ability to locally store data in memory.
  - Batch server operations.
  - Ability to interact with other Desktop applications such as e-mail and spreadsheets.
- Lower Total Cost of Ownership (TCO)
  - Ability to automatically update the Rich Client Platform applications to remote clients based on the server-side update information.
  - Centralized administration, setup, and client updates.
  - Ability to work across Wide Area Network (WAN) through multiple security infrastructures such as proxies, Firewalls, and so forth.
  - Built-in support for data compression and batch command processing results in optimal network utilization.
  - Ability to work with standard protocols such as Hyper Text Transfer Protocol (HTTP) and Hyper Text Transfer Protocol Secure (HTTPS).
- Deployment Strategy
  - Rich Client Platform applications are self contained Desktop applications, and depend on the Java Runtime Environments (JREs). The Rich Client Platform applications can be copied to theSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales directory and point them to the

specific JRE. This provides a reliable and coexisting application deployment strategy without disrupting other existing Java installations. For information about JRE versions that the Rich Client Platform supports, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Installation GuideInstalling the Platform*.

- Subsequent upgrades can be automated with theSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales auto-update feature, which checks and updates the existing configuration on a server. If you do not want to use the auto-update feature, you can turn it off and remotely perform a manual file copy based update.
- Since basic installation and upgrade involves a "file copy" operation, administration and maintenance can be done locally or remotely.
- Rich Clients support standard input devices such as keyboard, mouse, stylus, barcode scanner, and so forth. Rich Clients also support standard printers within the supported operating systems.
- Since Rich Client Platform is a Desktop application, any standard mechanism such as desktop shortcut or program file links can be used to launch the application.

Rich Client Platform and Desktop Applications

The Rich Client Platform supports desktop applications. A desktop application or Multiple Document Interface (MDI) application contains standard menus, views, editors, and so forth. You can work with multiple views and editors simultaneously. You can switch from one editor to another without closing any editor that is already open. You can use any standard mechanism (such as desktop shortcut or program file links) to launch the desktop application. A desktop application allows you to open one or more documents at the same time and displays each document in a separate window. The menu bar for a desktop application is displayed on the application frame. Some examples of desktop applications includeSterling Call Center and Sterling Store application, which is developed using the Rich Client Platform.

XML Binding for Rich Client Platform Applications

To easily create UIs, the core classes in the Rich Client Platform support XML Binding for different types of controls to an XML Distributed Object Model (DOM). This allows the UI developer to bind different controls on a form to various parts of the DOM. The advantage of using XML Binding is that the developer has to write limited code for displaying or retrieving data from a specific field in any screen or both. The developer has to only set and get the model of a screen to set and get the data for the entire screen.

The XML Binding is performed to map the input XML to the screen and back from the screen to an output XML. XML Binding in the Rich Client Platform is XML driven. A binding definition is an XPath (XML Path), which defines the rules for retrieving data from one XML and sending it to another XML. You can set the XML Bindings for various controls such as text boxes, combo boxes, buttons, tables, and so forth. The XML Bindings are specified to associate controls on the screen with a model (an XML document that stores information). To associate the controls to a model, XML Bindings are specified. Usually, it is an XPath specifying the attribute or an element in the document. The XML Binding used depends on the type of control that is used. The Rich Client Platform provides various XML

Binding data classes for different controls. For more information about XML binding classes, see the Binding Controls and Classes for Rich Client Platform Screens topic.

Note:

For a Label, the XPath expression is not supported in the binding definition.

Localizing Rich Client Platform Applications

The Rich Client Platform applications are all internationalized. This means they can handle multiple languages and cultural conventions transparently. The Rich Client Platform enables you to customize the Rich Client Platform applications in such a way that the extensions are also internationalized. The user can localize all the graphical text and messages. You can localize the Rich Client Platform application by defining locale-specific entries in the bundle file. For more information about localizing the Rich Client Platform application, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales : *Localization Guide*.

Database Localization

In addition to storing the transaction data, the database also stores configuration data, such as error codes and item descriptions of various attributes. This means that the database may need to store values in a language-specific format. If these database literals are not localized, screen literals displays inconsistently, with some displaying in the localized language and others displaying in English. You can store item descriptions in your database in multiple languages. If localizing Rich Client Platform application UIs, you may want to localize the factory setup. For more information about database localization, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Localization Guide*.

Themes for Rich Client Platform Applications

The theming feature enables users to define different fonts, colors used within the applications by creating a custom theme. For more information about theming controls, see "Defining Themes".

Related Tasks for Rich Client Platform Applications

This feature allows you to extend the Rich Client Platform applications by adding (or hiding) end user tasks in the UI. These tasks are available to the end user through a common related tasks view. Related Tasks feature enables you to perform the tasks that are related to a particular operation. You can group a set of related tasks by associated them with a group. You can also define a category, which can contain multiple tasks from multiple groups. For example, if you are viewing the details of an order, then all the related tasks for this operation such as Cancel Order, Add Order Line, and so forth are displayed in the Related Tasks view under the Order group. You can provide the implementation for displaying these related tasks on the screen. You can also provide the implementation for opening extensible related tasks in the application-provided editor or in your own custom editor. For creating the related tasks you need to extend the following extension points:

- YRCRelatedTasks extension point

- YRCRelatedTaskCategories extension point
- YRCRelatedTaskGroups extension point
- YRCRelatedTasksDisplayer extension point
- YRCRelatedTasksExtensionContributor extension point

For more information about adding new related tasks and hiding existing related tasks, see "Related and Shared Tasks".

Shared Tasks for Rich Client Platform Applications

Rich Client Platform-based applications may contain some reusable UI components such as lookup screens. In such cases, the other Rich Client Platform-based applications or extension plug-ins do not have to recreate the same UI components. Instead, they can use the available UI components as a shared task.

To maintain the backward compatibility and to avoid multiple plug-in dependencies, the shared tasks are registered with the Rich Client Platform plug-in. The other Rich Client Platform-based applications or extension plug-ins can directly invoke these shared tasks using the utility methods provided by the Rich Client Platform plug-in.

You can register the shared tasks through the YRCSharedTasks extension point defined in the Rich Client Platform plug-in. To use these registered shared tasks in your application, invoke them by clicking a button or menu item. For more information about registering and using shared tasks, see "Related and Shared Tasks".

Navigator Tasks for Rich Client Platform Applications

This feature allows you to extend the Rich Client Platform applications by adding or hiding the navigator task in the UI. These tasks are available to the end user in the navigator view.

For example, to modify the navigator tasks menu in Sterling Store Inventory Management PCA, perform the following sequence of actions:

1. Create a new Resources as required under Sterling Store Inventory Management PCA resource hierarchy.
2. Create a new Menu Hierarchy or modify the existing Sterling Store Inventory Management PCA menu hierarchy.
3. Assign the desired Menu Group to Sterling Store Inventory Management PCA user.

For more information about Defining Menus and Resources, see "Defining Menus" and "Defining Resources" sections of *Sterling Selling and Fulfillment Foundation: Application Platform Configuration Guide*.

Wizards for Rich Client Platform Applications

A wizard definition defines the flow of a wizard. You can define new wizard rules to control the flow of a wizard. The flow of a wizard depends on the output value of a wizard rule. The output of a wizard rule is compared with a transition value. Transition lines are used to transfer control from one wizard entity to another wizard entity.

A wizard definition contains:

- Wizard Entity—There are three types of wizard entities:
  - Wizard Page - A wizard page takes care of the UI in order to take inputs from a user.
  - Wizard Rule - A wizard rule is a logical step that performs computations to evaluate different output values. Based on these output values, wizard transitions are defined to decide the flow of the wizard.
  - Sub-task - This is a separate individual task that can be embedded into a wizard. A sub-task can be utilized in the wizard flow. When the execution of a sub-task is complete, the control moves to the next defined wizard entity in the wizard flow. For example, a sub-task can be a wizard that can be inserted between two wizard entities, or it can be the last entity in the wizard flow. If a sub-task is inserted between two wizard entities, the sub-task should display the **Next** button for navigation to the next wizard entity. If the sub-task is the last entity in the wizard flow, it should display the **Finish** button to end the wizard. This information must be passed to the context object, which is used to control the flow of data between the parent wizard and the sub-task, and contains the input to the sub-task. If there is an output of the sub-task, it can be set in the context and passed back to the parent wizard. The context object utility methods will display the appropriate buttons for navigation. However, the context object utility must have its position information in the parent wizard to display the correct navigation buttons.
- Wizard Transition—This is used to transfer control from one wizard entity to another wizard entity. Wizard transition connects wizard entity sequences with each other.

You can start your wizard with any wizard entity (a wizard rule or a wizard page or a sub-task).

For the wizard entity from where the wizard definition starts, the Starting property should be set to "true". For the wizard entity at which the wizard definition ends, the isLast property should be set to "true".

You can add transition lines to transfer the control from one wizard entity to another wizard entity based on the output values of the wizard rule. The transitions originating from a wizard page can have only one target. Transitions starting from a wizard rule can have multiple targets based on the rule output. The output of a wizard rule is compared with the transition values defined for a rule. Based on this value, control is transferred to the appropriate wizard entity.

All the new wizard definitions are created in the *Plug-in_id_wizard_name*.ycml file.

Note: Use a separate *Plug-in_id_wizard_name*.ycml file for each wizard definition.

Consider, for example, that you have a wizard definition for the following wizard flow:

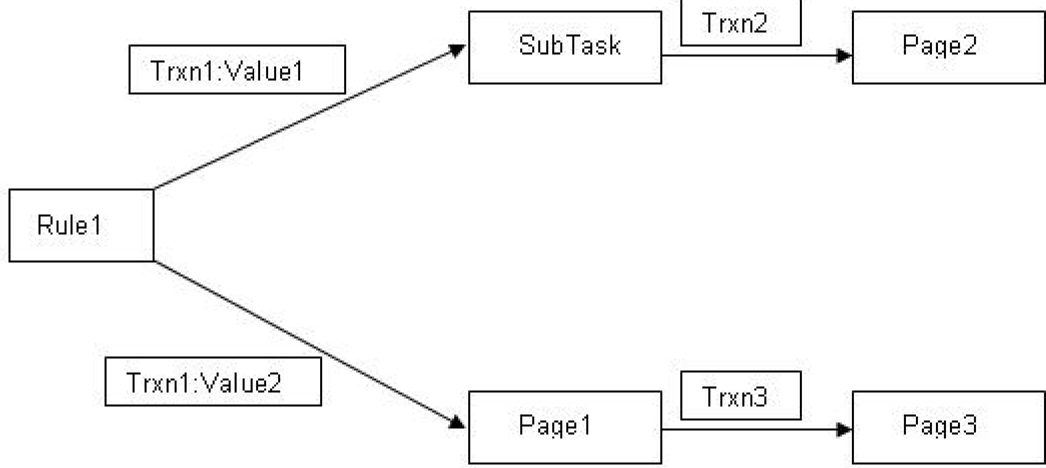


Figure 2. Sample Wizard Flow

In this case, the wizard starts from the wizard rule (Rule1). From the wizard rule (Rule1), the wizard transitions either to a wizard page (Page1) or a sub-task (Task1) based on the output values of the wizard rule (Rule1).

Depending on the transition of the wizard from the wizard rule (Rule1), the wizard ends at two different wizard pages (Page2 or Page3). For the wizard entity from where the wizard definition started, the start property must be set to "true". For the wizard entity at which the wizard definition ends, the isLast property must be set to "true".

The Rich Client Platform supports three types of wizard entities:

- A wizard rule (Rule1) contains a wizard rule identifier (ID), implementation class (Impl), namespace definition for the rule (Namespace), and the list of output values, one of which is the output of the wizard rule.

  Note: Multiple transitions can take place from a wizard rule. Therefore, a wizard rule can return multiple output values, one value for each transition that starts from the wizard rule.
- A sub-task (SubTask) contains a sub-task identifier (ID), implementation class (Impl), namespace definition for the sub-task (Namespace), and the flags isLast and Starting, to indicate whether the sub-task is the starting entity or the last entity in the wizard flow.

  Note: A sub-task cannot have multiple transitions. You can have only one transition starting from a sub-task and ending at another wizard entity.
- A wizard page (Page1) contains a wizard page identifier (ID) and implementation class (Impl).

  Note: A wizard page cannot have multiple transitions. You can have only one transition starting from a wizard page and ending at another wizard entity.

The application supports three types of transitions:

- Transition from a wizard rule—You can have multiple transitions from a wizard rule. The wizard transition that starts from a wizard rule contains a wizard transition identifier (ID), source wizard entity (source), multiple target wizard entities (target), and the output value of the wizard rule for which the transition

occurs. The source contains the identifier of the wizard entity from where the transition starts. The target contains the identifier of the wizard entity at which the transition ends.

For example, in the wizard flow illustrated earlier, two transitions start from a wizard rule (Rule1). These two transitions end at two different wizard entities, such as wizard page (Page1) and sub-task (SubTask), based on the output value returned by the wizard rule (Rule1).

Note: Because a wizard rule can have multiple transitions, when a transition starts from a wizard rule, you can define multiple targets, with values associated with each target.

Note: The transition identifier (ID) for all the transitions that start from a wizard rule are the same. The transition occurs based on the value defined for a given transition.

For example, in the wizard flow illustrated earlier, the transition starts from a wizard page (Page1) and ends in a wizard page (Page3).

- Transition from a wizard page—You can have only a single transition from a wizard page. The wizard transition that starts from a wizard page contains a wizard transition identifier (ID), source wizard entity (source), and target wizard entity (target). The source contains the identifier of the wizard entity from where the transition starts. The target contains the identifier of the wizard entity at which the transition ends.

  For example, in the wizard flow illustrated earlier, the transition starts from a wizard page (Page1) and ends in a wizard page (Page3).
- Transition from a sub-task—You can have only a single transition from a sub-task. A wizard transition that starts from a sub-task contains a wizard transition identifier (ID), source wizard entity (source), and target wizard entity (target). The source contains the identifier of the wizard entity from where the transition starts. The target contains the identifier of the wizard entity at which the transition ends.

  For example, in the wizard flow illustrated earlier, the transition starts from a sub-task (SubTask) and ends in a wizard page (Page2).

Hot Keys for Rich Client Platform Applications

Hot keys are keyboard shortcuts that perform a predefined function. For example, if you want to perform an operation, you can either click the **Search** button or press **F7**. The Rich Client Platform enables you to define hot keys for the new screens you create for Rich Client Platform applications. The Rich Client Platform also enables you to override the hot keys defined for the existing screens.

For more information about defining new hot keys and overriding existing hot keys, see "Defining and Overriding Hot Keys".

Debug Mode for Rich Client Platform Applications

Rich Client Platform enables you to run a Rich Client Platform application in the debug mode and performs additional validations on the Rich Client Platform application to reduce the number of errors created when performing extensions. Also, when you run an application in debug mode, the Rich Client Platform provides a comprehensive visibility to information about errors and missing parameters.

Note: If you do not specify the control name for a control, the background of that particular control is highlighted in red color, when the application is run in debug mode.

The Rich Client Platform performs the following validations in debug mode:

- Control Name Validation—The Rich Client Platform checks whether or not a unique control name has been specified for each control in the Rich Client Platform application. If this is not specified, when you move the cursor on that control, the tool tip displays "Specify the name of the control". Additionally, the control is displayed with a red background. This is useful in extending a Rich Client Platform application. You can easily extend a screen if you know the name of all controls on the screen.
- Bundle Entry Validation—The Rich Client Platform checks whether you have specified the bundle entry in the *bundle.properties file for each string in the Rich Client Platform application. If you do not specify the bundle entry for a string, it is considered as a non-localized string. Such non-localized strings always displays within the exclamation marks (!). For example, if you do not specify the bundle entry for the "Order No" string, the string display as !Order No!. This bundle entry validation helps the developers to understand whether the strings on the UI are localized or non-localized.

Note: The localized strings are sometimes displayed within the exclamation marks.

Running Rich Client Platform Applications in Debug Mode

About this task

You can run the Rich Client Platform application in debug mode in order to perform some extra validations and traces. Additionally, debug mode provides much more visual information to tell you what is wrong and where. You can run both a standalone application and an application within Eclipse in the debug mode.

Running the Standalone Rich Client Platform Application in Debug Mode

About this task

You can run the standalone Rich Client Platform application in debug mode. The standalone application can be a shipped application or an extended application.

To run the standalone Rich Client Platform application in debug mode:

Procedure

1. Modify the Rich Client Platform application's *.ini file to provide the appropriate VM arguments to run the application in debug mode. You can find the *.ini file for the Rich Client Platform application in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/*PCA_DIR*/ directory.

   For example, to run the Sterling Call Center and Sterling Store application in debug mode, edit the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/com/com.ini file.
2. In the *.ini file, add the following VM arguments:

   ```
   -vmargs
   -Ddebugmode=true
   ```

3. Run the EXE file of the Rich Client Platform application.

Running the Rich Client Platform Application in Eclipse in Debug Mode

About this task

When launching the Rich Client Platform application in Eclipse, in the VM Arguments field, enter the following arguments:

```
-Ddebugmode=true.
```

Prototype Mode for Rich Client Platform Applications

The advantage of running a Rich Client Platform application in the prototype mode enables you to quickly test UIs that you develop, without having to communicate with the server for APIs or services output. During an API or service call, the Rich Client Platform application uses the sample output XML files that are located in the prototype directory. The output of the sample output XML file is hard-coded and does not reflect any real-time data.

Running Rich Client Platform Applications in Prototype Mode

You can run the Rich Client Platform application in the prototype mode to test UIs. The Rich Client Platform enables you to run any standalone application or applications within Eclipse in prototype mode.

Running Standalone Rich Client Platform Applications in Prototype Mode

About this task

You can run the standalone Rich Client Platform application in prototype mode. The standalone application can be an extended application or any application that is already shipped.

To run the standalone Rich Client Platform application in prototype mode:

Procedure

1. Modify the Rich Client Platform application's *.ini file stored in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/*PCA_DIR*/ directory to provide appropriate VM arguments.

   For example, to run the Sterling Call Center and Sterling Store application in the prototype mode, modify the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/com/com.ini file.
2. In the *.ini file, add the following VM arguments:

   ```
   -vmargs
   -DProtoTypeDir=C:/EclipseInfrastructure/com.yantra.yfc.rcp.ri/prototype
   ```

   where ProtoTypeDir property refers to the prototype directory that contains the sample output XML files.
3. Verify that the name of all sample output XML files stored in the prototype directory are same as the command name for which they are used. For example, if the command name is getOrderDetails, the sample output XML file used for this command must be named as getOrderDetails.xml.

4. To run a command in the prototype mode, in the commands file, set the value of the prototype attribute for that particular command to "true". For more information about creating commands, see "Creating Commands".

   Note: The prototype mode is always set at the command level. It is important that you set the value of the prototype attribute to "true". This invokes the API or service in prototype mode.
5. Run the EXE file of the appropriate Rich Client Platform application.

Running Rich Client Platform Applications in Eclipse in Prototype Mode

About this task

When launching the Rich Client Platform application in Eclipse, in the VM Arguments field, enter the following argument:

```
-DProtoTypeDir=C:/EclipseInfrastructure/com.yantra.yfc.rcp.ri/prototype
```

Where ProtoTypeDir property refers to the prototype directory that contains the sample output XML files.

Note: Make sure that the name of all sample output XML files stored in the prototype directory are same as the command name for which they are used. For example, if the command name is getOrderDetails, the sample output XML file used for this command must be named as getOrderDetails.xml.

Tracing a Rich Client Platform Application

The Rich Client Platform enables you to trace a specific Rich Client Platform application. This is useful in checking operations such as API or service calls, warning or error messages (if any), bindings, and so forth. When you start tracing an application, the system writes all the information in the log file.

Timer Tracing

The Rich Client Platform provides a lightweight time tracing mechanism using which a Rich Client Platform application can be traced based on time. The timer tracing mechanism can be used to check the performance of an application.

The Timer Tracing mechanism can be used to measure the time taken by the application to perform the following operations:

- Screen Loading—Total time taken for completely loading a particular screen in order to be ready for use.
- Data Displaying and Fetching—Total time taken to populate the data on the screen and total time taken to fetch the data from the screen.
- API or Service Calling—Total time taken for an API or Service call to complete.
- Execute Wizard Rule—Total time taken to execute a "Wizard Rule" of a wizard.
- Execute Individual tasks—In addition a Rich Client Platform application can trace the total time taken for completion of a particular task such as Cancel Order, Search Order, Search Item, and so forth by setting the timer. You can set the timer using the TimerTraceEnabled(), startTimer(String formid,String identifier), endTimer(String formid ,String identifier,String operation ,String message) methods of the YRCPlatformUI class. For more information about YRCPlatformUI class and its methods, refer to *Javadocs*.

To use the lightweight timer tracing, in addition to the "trace" and "debugfile" arguments you must also pass the "tracelevel" as VM argument. The "tracelevel" VM argument can have following values:

- TIMER—Indicates that the timer trace information needs to be captured. The Rich Client Platform stores the timings traced in the log file that you have passed in the "debugfile" VM argument. The log file is renamed and is appended with "_timer" suffix. For example, you must pass the VM arguments as:

```
-Dtrace=true
-Dtracelevel=timer
-Ddebugfile=d:\RuntimeInfo.log
```

Since the tracelevel is "timer", the log file which contains the timer trace information will be named as RuntimeInfo_timer.log.

- DEBUG—Indicates that the debug tracer information needs to be captured. The Rich Client Platform stores the debug information in the log file that you have passed in the "debugfile" VM argument. The log file is renamed and is appended with "_debug" suffix. For example, you must pass the VM arguments as:

  ```
  -Dtrace=true
  -Dtracelevel=debug
  -Ddebugfile=d:\RuntimeInfo.log
  ```

  Since the tracelevel is "debug", the log file which contains the debug trace information will be named as RuntimeInfo_debug.log.
- ALL—Indicates that both debug as well as timer tracer information needs to be captured. In this case, the Rich Client Platform generates two separate files: one for debug trace information and the other one for timer trace information. The Rich Client Platform renames the log file that you have passed in the "debugfile" VM argument into two separate files with the appropriate suffix. The debug trace information is stored in the file with "_debug" suffix and timer trace information is stored in the file with "_timer" suffix. For example, you must pass the VM arguments as:

  ```
  -Dtrace=true
  -Dtracelevel=all
  -Ddebugfile=d:\RuntimeInfo.log
  ```

  Now since the tracelevel is "all", the system will generate two different log files. The file which contains the debug trace information will be named as RuntimeInfo_debug.log and the file which contains timer trace information will be named as RuntimeInfo_timer.log.

Note: By default, the Rich Client Platform takes ALL as the value for the "tracelevel" VM argument. This means that by default the system captures both timer and debug tracer details and stores them in their respective log files.

You can trace both the standalone application and any application within Eclipse.

Tracing a Standalone Rich Client Platform Application

About this task

You can set the trace on for a standalone Rich Client Platform application. The standalone application can be any application that is shipped or an extended application.

To start tracing a standalone Rich Client Platform application:

Procedure

1. Locate the *application_id*.ini file of the Rich Client Platform application stored in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/*PCA_DIR*/ directory to add the appropriate VM arguments.

   For example, to trace the Sterling Call Center and Sterling Store application, locate the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/com/com.ini file.
2. Edit the *application_id*.ini file, add the following parameter to the list of VM arguments:

   ```
   -vmargs
   -Ddebugfile=C:/debug.log
   ```

   where debugfile property refers to the directory in which the log file is created.

   Note: If the -vmargs parameter already exists in the file, add the -Ddebugfile parameter anywhere after the -vmargs parameter. Do not add another -vmargs parameter.
3. Run the EXE file of the appropriate Rich Client Platform application.
4. After you successfully log in to the application, the application window displays. To start or stop tracing the application, press **Ctrl+F2**.

   Note: When you press **Ctrl+F2** to start the trace, the information that is present in the log file is deleted. In the title bar of the application "Trace is On" displays:

   IBM Sterling Call Center - Trace is On

   Note:

   If you want the trace to be ON by default for an application, set the trace property to "true" in the *.ini file. For example, `-Dtrace=true`

   To stop tracing the application, press **Ctrl+F2.**

   IBM recommends not to set the trace ON as the default option. Instead, use the **Ctrl+F2** key combination to turn the trace ON, if needed.

Tracing a Rich Client Platform Application in Eclipse

About this task

When launching the Rich Client Platform application in Eclipse, in the VM Arguments field, enter the following argument:

```
-Ddebugfile=C:/debug.log
```

where debugfile property refers to the directory in which the log file is created.

Masking Sensitive Information During Trace

You can mask sensitive information such as credit card information, CVV numbers and passwords in the generated log files by using the message filters provided by the Application Platform. The message filters are used before logging. You can

substitute messages in the log files to protect sensitive information by using the message filters. To use the message filters, you must first register them during plugin initialization.

To hide sensitive information by using message filters, do the following:

1. The Application Platform provides a new interface IYRCTraceMessageFilter for filtering sensitive information.

2. You must implement the IYRCTraceMessageFilter interface to provide message filters as required for hiding sensitive information, before writing them to the log files.

The IYRCTraceMessageFilter interface returns a message string which is written to the log file. For more information on message filters and the IYRCTraceMessageFilter interface, refer to the Javadocs.

Note: Hiding sensitive information using the message filters is applicable only for the debug file, not for the timer file.

Registering a Message Filter

The message filters are used to hide or filter sensitive information such as credit card information, passwords or CVV numbers in the log files. A new method addTraceMessageFilter is added in the YRCPlatformUI class. To register your message filter file use the addTraceMessageFilter method within the plugin constructor. For more information on registering the message filter, refer to the Javadocs.

Note: Before calling the addTraceMessageFilter() method, the plugin must be registered using the registerPlugin() method of the YRCPlatformUI class.

Capitalizing the Text Entered in Rich Client Platform Applications

About this task

You can force all capital letters in text fields for a Rich Client Platform application. When you enable capital letters for text fields, the value in the text field is automatically converted to capital letters even if the value entered is in lowercase letters.

To enable capital letters in text fields:

Procedure

1. Locate the *application_id*.ini file of the Rich Client Platform application stored in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/*PCA_DIR*/ directory to add the appropriate VM arguments.

   For example, to enable capital letters in text fields for Sterling Call Center and Sterling Store application, locate the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/com/com.ini file.
2. Edit the *application_id*.ini file, add the following parameter to the list of VM arguments:

   ```
   -vmargs
   -Denablecapsintextboxes=true
   ```

Fetching Images for Rich Client Platform Applications

About this task

The Rich Client Platform allows you to fetch images from the server to the labels and table columns. To fetch images from the server, define a Config element in the location.ycfg file and set the connection settings for fetching images. You can define multiple Config elements to fetch images of different formats. The Rich Client Platform supports various image formats such as GIF, BMP, ICO, JPEG, PNG, and TIFF. For information about configuring the connection settings for fetching images from the server, see theSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales *Installation GuideInstalling the Platform*.

Note: You can fetch images from the server only for labels and table columns.

Security Handling for Rich Client Platform Applications

The Rich Client Platform enables you to securely communicate with the servers. It allows you to connect to servers using the HTTPS protocol. This provides an authenticated and encrypted way of running the resources from the server on the client machine. The authentication and encryption is handled using certificates, which help you run the authorized resources on client machines. These certificates must be stored in the truststore folder of the Rich Client Platform plug-in.

By default, during handshake, if there is a mismatch between the URL's host name and the server's identification host name, the Rich Client Platform allows the HTTPS connection.

Note: You must provide your own custom verification logic by adding the host name verifier.

You can add your own custom verification logic by extending the YRCHostNameVerifier extension point.

To connect to the server using the HTTPS protocol, specify the protocol as HTTPS and also specify the port number for the HTTPS connection in the configuration file. For more information about configuring the connection settings for HTTPS connection, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Installation GuideInstalling the Platform*.

Output Templates for Rich Client Platform Applications

Output templates are used during an API call or service to ensure that the data is retrieved in the desired format (for example, if you want to display few attributes in a particular order.)

You can also merge output templates to retrieve the additional data from an API or service. For more information about template merging, see "Merging Templates".

Commands for Rich Client Platform Applications

The Rich Client Platform has modeled API calls as commands. Commands are defined to call APIs or services to retrieve data in the desired format. Whenever you call an API, specify the name of the command associated with the API. The Rich Client Platform supports the creation of commands at a form level. You can

also override commands if necessary. This is useful when you want to call a custom API for a particular form. For more information about creating commands, see "Creating Commands".

Log Files for Rich Client Platform Applications

Log files contain information such as warnings and errors (if any). When you run a Rich Client Platform application, the following log files get generated:

- Eclipse log file—Whenever you run a Rich Client Platform application, the Eclipse automatically creates a log file. This log file contains high-level error messages and/or warnings logged by Eclipse.

  Using the osgi.instance.area.default property, you can configure the location to store this log file in the *INSTALL_DIR*/repository/rcpdrop/ *OPERATING_SYSTEM*/*PCA_DIR*/configuration/config.ini file. By default, this log file is stored in the @user.home/*application_workspace* directory. The Rich Client Platform does not allow you to rename the log file.

  For example, if you run the Sterling Call Center and Sterling Store application, the config.ini file is stored in the *INSTALL_DIR*/repository/rcpdrop/ *OPERATING_SYSTEM*/COM/configuration directory.

  In the config.ini file, the osgi.instance.area.default property is set to:

  ```
  osgi.instance.area.default=@user.home/comworkspace
  ```

  Here, @user.home refers to a user's home directory. You can only change the directory where the log file gets created. You cannot change the name of the file.
- Rich Client Platform Infrastructure log file—Whenever you run a Rich Client Platform application for which the tracing is turned ON, the Rich Client Platform plug-in creates a log file. This log file provides information about API or service calls, warning or error messages (if any), bindings, and so forth. You can specify the location in the *.ini file of the appropriate application that you want to trace. The *.ini file is located in the *INSTALL_DIR*/repository/rcpdrop/ *OPERATING_SYSTEM*/*PCA_DIR*/ directory.

  Note: Make sure that the tracing is turned ON in the application for the information to be written in the log file.

Masking Sensitive Information During Logging

The Rich Client Platform enables you to prevent sensitive information such as credit card number, passwords, and so forth from being logged in the log messages. You can mask sensitive information by defining a set of named regular expressions in the customer_override.properties file.

The Rich Client Platform matches the log messages against a set of well defined regular expressions and discards the message if it matches.

To mask sensitive information during logging, define a set of named regular expressions against which you want to match the log messages in the *INSTALL_DIR*/properties/customer_overrides.properties file using the following properties:

```
filterset.name.pattern.num=pattern
[optional]filterset.name.replace.num=replace
```

where *pattern* is a Java-style regular expression and defines the regular expression against which you want to match the message string. The *replace* property is

optional, and defines the string which will be used to replace the expression. The filter is used to suppress a log message and layout is used to replace the log message with some other string.

You can set the default FilterSet parameters by setting the following properties:

```
default.rcp.filter.filterset=logfilter.filter_name
default.rcp.layout.filterset=logfilter.layout_name
```

You can also define a common set of regular expression patterns across multiple filter sets as following:

```
filterset.name.includes=name1,name2,...
```

You can view the *INSTALL_DIR*/properties/logfilter.properties.in file to see some sample entries for defining these properties.

Data Caching for Rich Client Platform Applications

The Rich Client Platform allows you to cache data at the plug-in level for later use. The cache is maintained separately for each plug-in, with each plug-in cache having a MAX SIZE. When the cache size for a plug-in reaches the MAX SIZE, the Least Recently Used (LRU) algorithm is used to remove the old cache and add the new cache. To cache the data for a plug-in, call the add() method of the YRCCacheManager class. This method accepts the following input arguments:

- pluginId (String)—Pass the identifier of the plug-in for which you want to cache data.
- key (Object)—Pass the key object that you want to associate with the data to be cached.
- value (Object)—Pass the value object that you want to associate with the data to be cached.

  To access the cached data for a plug-in, call the get() method of the YRCCacheManager class. This method accepts the following input arguments:
- pluginId (String)—Pass the identifier of the plug-in for which you want to access the cache data.
- key (Object)—Pass the key object associated with the cached data that you want to access.

Note: After you access the cached data for a plug-in, that particular cached data is moved to the top of the Most Recently Used (MRU) list of the plug-in.

Clearing Data Cache

You can clear all the existing cache for a particular plug-in by calling the clearCache() method of the YRCCacheManager class. This method accepts the plug-in identifier as input argument. In the pluginId (String) argument, pass the identifier of the plug-in for which you want to clear all the existing cache.You can also clear the entire existing cache for all the plug-ins at the same time by calling the clearAllCache() method of the YRCCacheManager class.

Error Handling for Rich Client Platform Applications

The Rich Client Platform enables you to handle the errors encountered in an API call. Because it allows you to add an error handler extension for an error code, all the related error codes are handled and routed to the error handler.

Configure the error handler to show the appropriate error message. The current context used for an API call can be passed to the error handler to process the errors.

The extension point definition for the error handler is as follows:

```
com.yantra.yfc.rcp.YRCErrorHandler
```

A sample extension defined in the plugin.xml can be as follows:

```
<extension id="" point="com.yantra.yfc.rcp.YRCErrorHandler">
   <ErrorHandler errorCode="<error_code_to_be_handled>"
      class="<error_handler_class_name>"/>
</extension>
```

The class mentioned in the attribute `class` should implement interface com.yantra.yfc.rcp.IYRCErrorHandler.

The method handleError(String errorCode, Document errorDocument, YRCApiContext context) is invoked when an error is encountered for the error code defined in the extension.

Table Filtering for Rich Client Platform Applications

About this task

The Rich Client Platform enables you to filter the records in a table based on custom criteria. For example, if a table contains 100 records, you may want to filter the records in the table based on some value for one or more columns. You can achieve this by using the Table Filter functionality.

To filter records in a table:

Procedure

1. Right-click the table and select **Filter** from the pop-up menu. Depending on the table you are filtering, the filtering options provided in the Filter pop-up window vary for each table column.
2. Enter the criteria for one or more columns based on how you want to filter table records.
3. Click **OK**.

Clearing the Sort Order in a Table

About this task

The Rich Client Platform enables you to clear the existing sort order in a table when necessary. For example, you may want clear the default sort order.

To clear the sort order, call the clearSort(String tableName) method of the YRCBehavior class and pass the table name for which you want to clear the sort order. For example,

```
btnReset.clearSort("tblSearchCriteria");
```

where tblSearchCriteria is the table name.

Scheduling Jobs for Rich Client Platform Applications

Jobs are reusable units of work that can be scheduled to run with the Job Manager. When a job is completed, it can be scheduled to run again. The Rich Client Platform supports scheduling of:

- Generic jobs
- Alert-related jobs

Scheduling a Generic Job

About this task

The Rich Client Platform enables you to schedule a job by registering all the generic jobs.

To create and register a generic job:

Procedure

1. Create a new object of the YRCJobData class. This class accepts the following arguments as input:
   - proceedEvenIfIdle (Boolean)—This flag indicates whether the job should be suspended or run if the application is idle.

     Note: You can set the idle time (in minutes) for the job by providing the VM arguments. For example, `-Dideltime=3`

     By default, idle time is set to 5 minutes.
   - scheduleIntervalInMinutes (int)—Contains the time interval (in minutes) after which you need to reschedule the job.
2. Create a new job. This job must extend the YRCJob class. The YRCJob class accepts the following arguments as input:
   - name (String)—Contains the name of the job.
   - YRCJobData (Object)—Job data object, which contains the configuration of the job.
3. Override the execute() method to write the code to perform the appropriate operation when the job is scheduled.
4. Register the job you created with the Rich Client Platform using the registerJob (YRCJob job) method.

Scheduling an Alert-Related Job

About this task

The Rich Client Platform enables you to schedule alert-related jobs. You can configure the alert-related jobs to run at a desired time interval. For example, you may want a message to pop up in an Alert Pop-up window panel every two minutes when an alert is assigned to the user who has logged in.

2 alerts have been assigned to you.
Details ExceptionType1
Type Prototype Exception Do Not Notify

You can register such alert-related jobs with the Rich Client Platform, which in turn schedules the jobs at the desired interval.

To create and register an alert-related job:

Procedure

1. Create a new object of the YRCJobData class. This class accepts the following arguments as input:
   - proceedEvenIfIdle (Boolean)-A boolean flag that indicates whether the job has to run even if the application is idle.

     Note: You can set the idle time (in minutes) for the job by providing the VM arguments. For example, `-Dideltime=3`

     By default, idle time is set to 5 minutes.
   - scheduleIntervalInMinutes (int)-Contains the time interval (in minutes) after which the job must be rescheduled.
2. Register the job you created with the Rich Client Platform using the registerAlertJob() method. This method accepts the following parameters:
   - IYRCAlertPopUpHandler-This interface provides visibility to alert details when you click the alert message hyperlink. This is an optional parameter. If this parameter is passed as "null", the alert message displays as a label instead of a hyperlink.
   - YRCJobData-Job data object that contains the job configuration.

Preventing the Deactivation of Alert Notification

About this task

Alert related jobs configure alerts to pop up at scheduled intervals. Users can turn off notification of alerts by selecting the **Do Not Notify** check box in the Alert Notification Panel. However, this may not be desired for certain alerts, which have to be mandatorily run and displayed.

To prevent disabling of such alerts, the system provides a method to hide the **Do Not Notify** check box.

To hide the **Do Not Notify** check box:

Procedure

1. A static utility method is added in the YRCAlertMessageController class with the following format:
   ```
   public static void hideDoNotNotifyCheckbox(true)
   ```
2. Set the flag to true to hide the check box.

   This method must be called before registering the Alert job. Based on the flag set in the utility method, the **Do Not Notify** check box is hidden or displayed. By default, the check box is displayed.

Audio Files for Rich Client Platform Applications

About this task

In an Rich Client Platform-based application, you can load and play an audio file when a specific event occurs. For example, you may want to play an audio file whenever an alert is raised.

To play an audio file:

Procedure

1. First, load the audio file into an Rich Client Platform-based application by defining the theme entries in the *Plug-in_id_theme_name*.ythm file at the plug-in level. To set a theme for loading an audio file, define the entries in the theme file for the audio file. For example:

   ```
   <ThemeEntry Name="AlertPopupAudio">
       <Audio Path="/audio/alertpopup.wav" LoopCount="3" />
   </ThemeEntry>
   ```

   Here, the Name attribute indicates the name of the theme entry that is used for theming an audio file. The Path attribute contains the path of the audio file to be loaded, and the LoopCount attribute indicates the number of times the audio file should be played repeatedly.

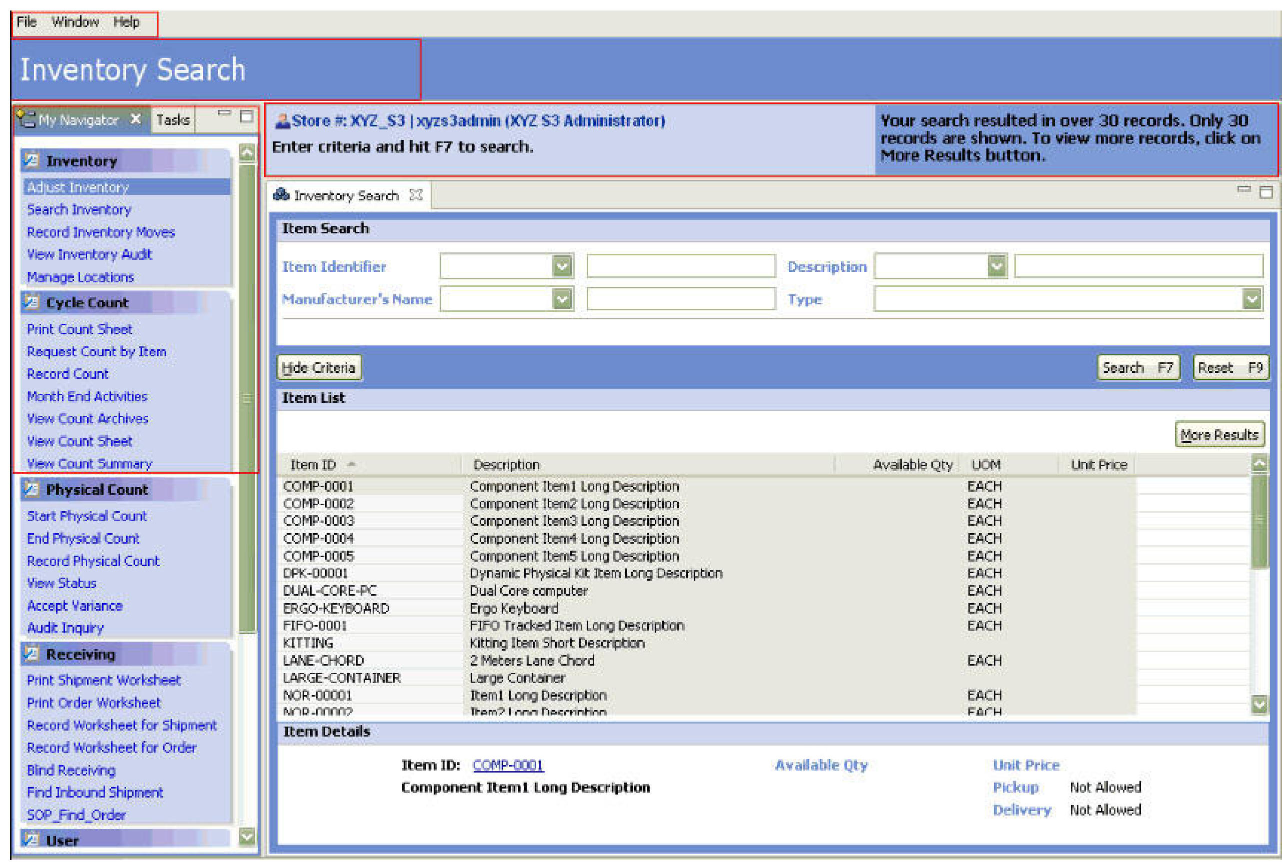
   Note: Sterling Rich Client Platform Infrastructure supports only *.wav files.
2. Play the audio file by calling the playAudio(String themeEntry) method of the YRCPlatformUI class. For example:

   ```
   YRCPlatformUI.playAudio("AlertPopupAudio");
   ```

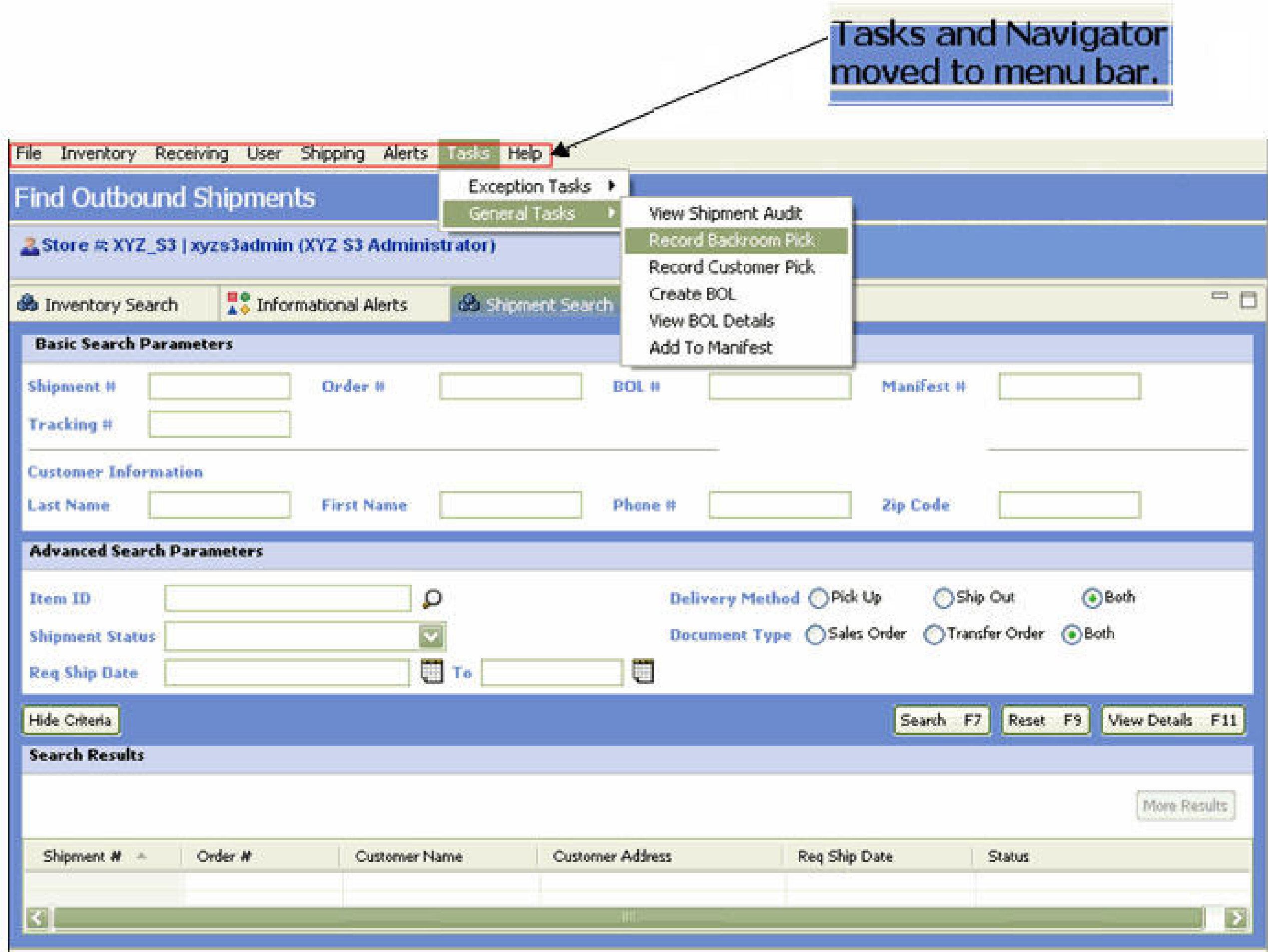
   In this case, the AlertPopupAudio is the name of the theme entry defined in the theme file.

Low Resolution Display for Rich Client Platform Applications

The following figure depicts one of the Rich Client Platform PCA applications that displays on a screen for which the system resolution is set to greater than 800 X 600 pixels (high resolution display).



If you set the screen resolution to less than or equal to 800 X 600 pixels and relaunch the application, the left panel (Navigator and Tasks panel) is not visible and the menu items are placed in the menu bar. The font size in the theme entries defined for a particular screen also reduces by one point. The following figure depicts one of the Rich Client Platform PCA applications that displays on a screen whose system resolution is set to less than or equal to 800 X 600 pixels.



Displaying Panel Tasks on the Menu Bar for Rich Client Platform Applications

About this task

In high resolution, the Navigator tasks are displayed in a panel on the left side. You can display these Navigation panel tasks as menu bar entries in the Rich Client Platform application.

To display the Navigation panel tasks as menu bar entries:

Procedure

1. Modify the Rich Client Platform application's *.ini file to provide the appropriate VM argument. You can find the *.ini file for the Rich Client Platform application in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/*PCA_DIR*/ directory.

   For example, to run theSterling Call Center and Sterling Store application in debug mode, edit the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/com/com.ini file.
2. In the *.ini file, add the following VM argument:

```
-vmargs
-Dshownavigatorasmenu=true
```

3. Run the *.exe file of the Rich Client Platform application.

Switching Locales for Rich Client Platform Applications

The Rich Client Platform application enables users to switch locales based on the locale configuration.

To switch locales, pass the locale code as `-Dlocalecode=<LOCALE_CODE>`.

Note: If the passed locale is not defined on the server, the system locale is used.

Using a VM Login for Rich Client Platform Applications

The Rich Client Platform enables you to log in to a Rich Client Platform application by passing the location name, user ID, and password as VM arguments.

Note: You can pass the appropriate VM arguments in the *.ini file of a Rich Client Platform application. You can find the *.ini files in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/*PCA_DIR*/ directory.

You can pass "location" as a VM argument to log in to a Rich Client Platform application. For example, if you want to log in to a Rich Client Platform application using "DEFAULT" as the location, pass the following VM argument:

```
-Dlocation=DEFAULT
```

Note: If you pass "location" as a VM argument, the Location Preference pop-up window does not display.

You can pass "userid" and "password" as VM arguments to log in to a Rich Client Platform application. For example, to log in to a Rich Client Platform application using "storeop" as the user ID and "admin" as the password, pass the following VM arguments:

```
-Duserid=storeop
-Dpassword=admin
```

Note: If you pass "userid" and "password" as VM arguments, the Log In pop-up window does not display.

By default the Sterling Rich Client log-in window is displayed at the top and cannot be minimized. To change the log-in window behavior, you need to set the value of the "logindialogontop" system property as “false” in application ini file. If you want the log-in window to be displayed at the top, set the value of the "logindialogontop" property as "true" (default behavior).

Using a VM JRE for Rich Client Platform Applications

In case multiple Java Runtime Environments (JREs) are installed on the system, the Rich Client Platform enables you to specify which Java Runtime Environment (JRE) to use to launch the Rich Client Application by passing the Path as VM argument.

You can pass the VM arguments in the *.ini file of a Rich Client Platform application. You can find the *.ini files in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/*PCA_DIR*/ directory.

For example, you can find the com.ini file forSterling Call Center and Sterling Store application in the *INSTALL_DIR*/repository/rcpdrop/*OPERATING_SYSTEM*/com/com.ini file.

Edit the *application_id*.ini file, and add the following parameter to the list of VM arguments:

```
-vmargs
<path_to_the_JRE>
```

Supervisory Overrides for Rich Client Platform Applications

The Supervisory Override functionality of the Rich Client Platform enables a user with no permissions to perform a particular task or operation. For example, if a user logs in to a Rich Client Platform application to modify the value of a field, the user must have permission to perform this task. Otherwise, you can perform supervisory overrides to allow the user to modify the field value.

Using the Pop-Up Method to Perform Supervisory Overrides

About this task

This section explains how to use the pop-up method to perform supervisory overrides for the currently logged in user. The advantage of using this method is that the user does not need to manually log out of the application after performing the task. As soon as the user closes the pop-up window, the system automatically logs the user out of the application.

To perform supervisory overrides using the pop-up method:

Call the openSupervisorShell() utility method in the YRCPlatformUI class. This method considers the following input arguments:

- pnlRoot (Composite)-Specifies the screen to display as a pop-up window.
- permissionID (String)-Specifies the resource identifier of the task or operation for which the user must have permission.
- titleKey (String)-Specifies the title of the pop-up window.
- iconTheme (String)-Specifies the theme entry of the image to display in the pop-up window.
- width (int)-Specifies the default width of the pop-up window.
- height (int)-Specifies the default height of the pop-up window.

When the openSupervisorShell() method is called, the Rich Client Platform performs the following actions:

Procedure

1. The Login pop-up window displays.
2. After successfully logging in to a Rich Client Platform application, the system verifies whether the user has permission to perform the task.
3. The appropriate screen opens in a pop-up window.
4. When the user closes the pop-up window, the system automatically logs the user out of the application.

Starting a Supervisory Transaction to Perform Supervisory Overrides

About this task

Another method of performing supervisory overrides is to start a supervisory transaction. However, if you use this method, the user must manually log out off the application after performing a task or operation.

To perform supervisory overrides by starting a supervisory transaction:

Procedure

1. Call the handleSupervisorTransaction() utility method in the YRCPlatformUI class. This method considers the following input arguments:
   - permissionID (String)-Specifies the resource identifier of the task or operation for which the user must have permission.
   - actionID (String)-Specifies the identifier of the action to invoke.

   When the handleSupervisorTransaction() method is called, Rich Client Platform performs the following actions:
   a. The Login pop-up window displays.
   b. After the user successfully logs in to a Rich Client Platform application, the system verifies whether the user has permission to perform the task.
   c. Rich Client Platform invokes the task.
2. Call the logoffSupervisor() method to log the user out of the application.

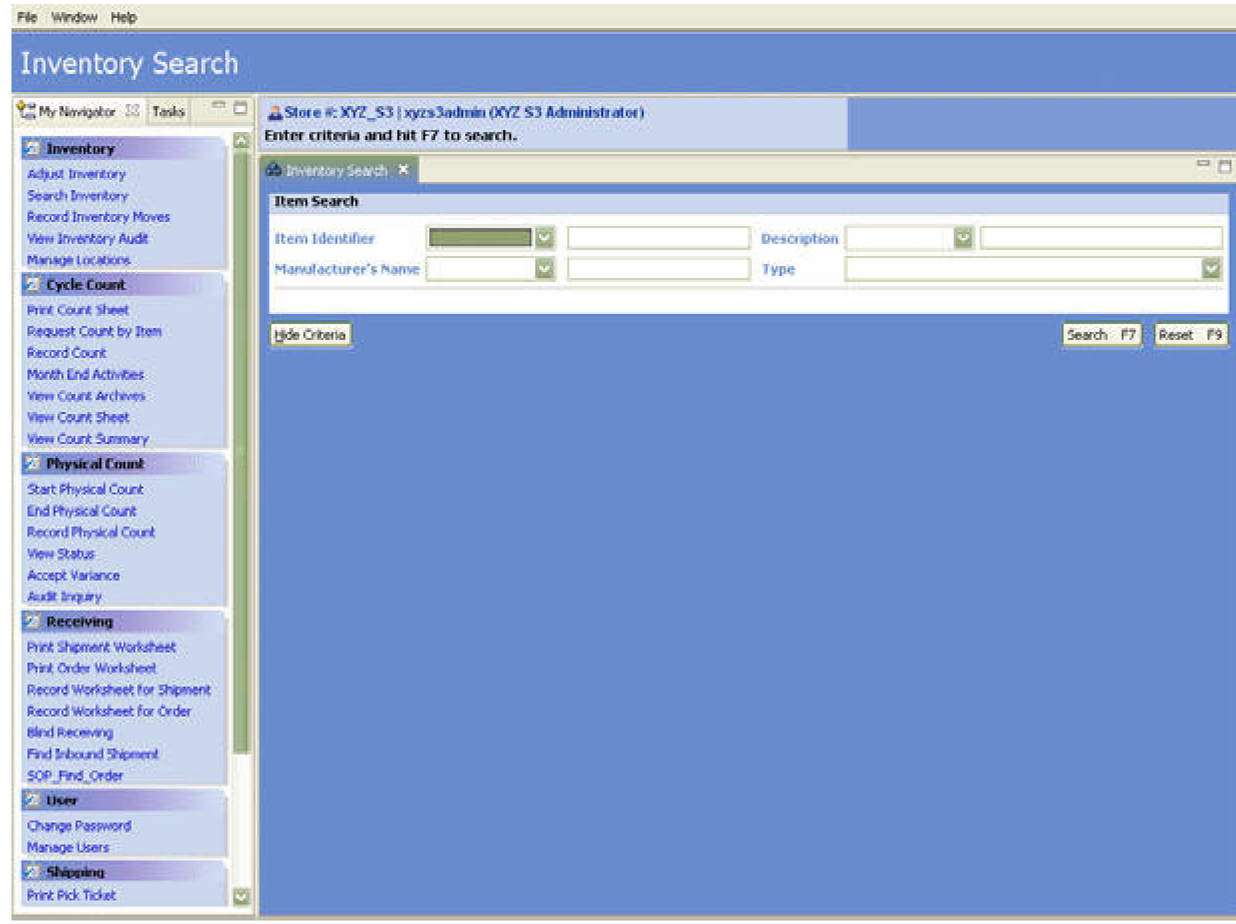
Running Rich Client Platform Applications in POS Mode

About this task

The Rich Client Platform enables you to run the Rich Client Platform application in Point of Sales (POS) mode. When you run the Rich Client Platform application in POS mode, the title bar of the application window is removed.

To run a Rich Client Platform application in POS mode, set the value of the posmode parameter to true by passing `-Dposmode=true` as the VM argument.

The following figure depicts the application layout in POS mode.



Version-Based Communication between Client and Server

When server components are migrated from an older version to a later version, all the server components are deployed in the server in a single exercise. Similarly, the Rich Client Platform client is also updated with the latest version. The old Rich Client Platform client must have the functionality to communicate with the migrated or the new application server.

- The Rich Client Platform application is built using the Rich Client Platform plug-in, the PCA plug-in and a custom extension plug-in.
- A client.properties file is provided in resources.jar (com.yantra.yfc.rcp plug-in), which is modified by a PCA for PCA-specific properties. The client.properties file contains the version information and other details as key-value pairs, which are available at the server in an environment object.
- You can add your own properties in another client.properties file in the extn directory (in the resources.jar file).
- To change any of the existing properties, add the new value (for the property key) in the same client.properties file mentioned in the previous step.

  Note: Custom properties override the Rich Client Platform or PCA properties.
- All the PCA-specific keys should start with their respective module code. For example, the version key for COM PCA entry should be com_Version=8.0.

- When creating an HTTP connection from the Rich Client Platform application to an application server, all the keys are set into the request header.
- In the server, the Rich Client Platform Servlet reads the request header and sets them into the YFSEnvironment.
- The environment object is passed as input to all the services and APIs. A Java HashMap of client properties can be obtained from this YFSEnvironment object, which contains the value of a key pertaining to the client version. This value can be used appropriately on the server.

Client Component

The following methods are provided in the YRCClientPropertiesManager:

- void setClientProperty(String key, String value, boolean overrideIfExists)sets additional properties dynamically.
- Properties getClientProperties ()returns all the client property registered through the client.properties file dynamically using the void setClientProperty method.

Server Component

A new ClientVersionSupport interface has been added to enable version-based communication between client and server. This interface is implemented by the out-of-the-box YCPContext and the InteropEnvStub.

Any class implementing YFSEnvironment should also implement the ClientVersionSupport interface. From the YFSEnvironment, you can get the value for a specific key from the hashmap.

The sample code for server class is shown here:

```
If (env instanceof ClientVersionSupport)
 {
   ClientVersionSupport
     clientVersionSupport = (ClientVersionSupport) env;
   HashMap
     map = clientVersionSupport.getClientProperties();
    If(map != null) {
       String value = (String)map.get(key);
             }
  }
```

To enable multiple Rich Client Platform clients for communicating with the corresponding server components, multiple commands.ycml files are supported, one for each Sterling Application Platform version of the client connecting to it:

- A utility class file YRCCommandsMergeUtils.java is added to Sterling Application Platform.
- Use this utility java class in build scripts for merging all commands into one single file named Commands_VERSION.ycml. The version information is read from the client.properties file.

Integrating Web Applications with Rich Client Platform

The Rich Client Platform provides a mechanism to integrate multiple Web applications based on different domains, which enables applications on Rich Client Platform to seamlessly connect to one or more Sterling Web application without actually logging into the other application.

To integrate other Web applications with Rich Client Platform, an extension point YRCWebAppIntegrator and an interface IYRCWebAppHandler are added to Rich Client Platform and a number of utilities are exposed in the class YRCWebAppUtils.

The required configuration details (used for logging in to the Web application) must be provided in the locations.ycfg file by specifying a config element (name, application ID, protocol etc) for each application that must be integrated with Rich Client Platform, as follows. Each config element name, which identifies the Web application to connect to, must be unique. The ApplicationID is the Web Application ID.

```
<Config Name=WebApp1
ApplicationID=""
Protocol="http"
BaseUrl="10.11.26.99"
PortNumber="7007"
WebAppContext="/smcfs<application_name>"
NoUILoginURL="/NoUILoginServlet">
</Config>
```

Note: Include a config element for each Web application to be integrated. The attributes, Protocol, BaseUrl, NoUILoginServlet and ApplicationID are mandatory. The system creates the URL by concatenating the following values provided under each config element:

```
Protocol + :// + BaseUrl + : PortNumber + WebAppContext +
   NoUILoginServlet
```

Create an Extension

About this task

To create an extension:

Procedure

1. Each extension has a number of elements, corresponding to the number of Web applications that Rich Client Platform wants to connect to.
2. Each extension element must contain the following mandatory attributes:
   - id -This ID should correspond to the config element name that contains the configuration details required for a particular Web application.
   - classToLoad - Specifies the class to be loaded to implement the interface IYRCWebAppHandler.
   - The IYRCWebAppHandler has the following format:

```
public interface IYRCWebAppHandler {
     /**
      * This method implementation will have to store the browser
      * configuration details so that the application
      * implementation can access the same when required.
      *
      * @param configElement is the config element which provides
      * browser configuration details for the web application
      * which the Rich Client Platform Application intends to switch.
      */
       public void init(Element configElement );
     /**
      * This method implementation will have to handle login to the
      * application, Rich Client Platform application intends to
      * connect to. Add any listeners that need to be added to
      * the browser including the one to handle session timeout .
      *
```

```
      * @param browser is the browser instance provided by the application
      * @param handler has to provide implementation as to
      * what data has to posted.
      */
       public void handleBrowser(Browser browser,
IYRCBrowserHandler handler);
      /**
       * This method implementation will have to store the login
       * information of the user who has logged in to the application.
       *
       * @param info provides information about the current user
       * userid and session information
       */
        public void setUserInfo(YRCLoginInfo info);
```

3. The following utilities related to the Web application integration are exposed in the class YRCWebAppUtils:
   - setUpBrowser - Must be called when a Web application has to be integrated with Rich Client Platform.

     ```
     public static void setupBrowser(String configName,
     Browser browser, IYRCBrowserHandler handler)
     ```

   - loginToWebApp - Must be called when a Rich Client Platform application that is already launched needs to log the user in to another application by using the session ID.

     ```
     public static YRCWebAppLoginInfo loginToWebApp(String configName,
     YRCLoginInfo info,HashMap<String, String> paramsMap)
     ```

   - addCookiesToBrowserSynchronously - Must be called to set cookies to the Web browser synchronously.

     ```
     public static YRCWebAppStatus addCookiesToBrowserSynchronously
     (String webAppconfigName, final Browser browser)
     ```

   - addCookiesToBrowserAsynchronously - Must be called to set cookies to the Web browser asynchronously.

     ```
     public static void addCookiesToBrowserAsynchronously
     (final String webAppConfigName,final Browser browser,
     final YRCWebAppLoginInfo webAppLoginInfo,
     final IYRCBrowserHandler handler)
     ```

   - formNoUILoginURL - Must be called to create an URL from the Web application config element.

     ```
     public static String formNoUILoginURL(Element webAppConfigElement)
     ```

Chapter 3. The Development Environment for Rich Client Platform Applications

Installing Prerequisite Software Components

This section describes the various software components required to customize the Rich Client Platform application. Before deploying the Rich Client Platform application, make sure that you have already installed the following software components:

- Eclipse SDK

  Install the Eclipse SDK version that the Rich Client Platform supports.
- Eclipse-related Plug-ins

  Install the following Eclipse-related plug-ins that the Rich Client Platform supports.
- VE (Visual Editor) plug-in
- GEF (Graphical Editor Framework) plug-in
- EMF (Eclipse Modeling Framework) plug-in
- Java Development Kit (JDK)

  Install the JDK version that the Rich Client Platform supports.
- Rich Client Platform plug-ins

  Install the following Rich Client Platform plug-ins that the Rich Client Platform supports. For more information about the Rich Client Platform plug-ins version, see the *Installation Guide*.
  - Rich Client Platform plug-in
  - Rich Client Platform Tools plug-in

These plug-ins are shipped along withSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales and are located in the *INSTALL_DIR*/rcpclient directory

Note: If you are installing a new version of the Rich Client Platform plug-ins or updating the earlier versions you must clean the cached build information in Eclipse.

Clean Cached Build Information in Eclipse

About this task

To clean cached build information in Eclipse, start the Eclipse SDK with the "-clean" option:

Procedure

1. Right-click the Eclipse's shortcut and select **Properties** from the pop-menu. The Properties window displays.
2. In Target, enter the command-line argument `-clean` at the end. For example, `"C:\Eclipse\eclipse\eclipse.exe" -clean`.
3. In Target, enter the command-line argument `-clean` at the end. For example, `"C:\Eclipse 3.2\eclipse\eclipse.exe" -clean`.
4. Start the Eclipse SDK.

© Copyright IBM Corp. 1999, 2013

Installing the Rich Client Platform Plug-In

About this task

To install the Rich Client Platform plug-ins:

Copy the contents of the folder <*INSTALL_DIR*>/rcpclient to the <*ECLIPSE_HOME*>/plugins folder. <*ECLIPSE_HOME*> refers to the Eclipse SDK installation directory.

The rcpclient folder contains the following plug-ins:
- com.yantra.ide.rcptools.core_1.1.0 - This plug-in is used to enable the Rich Client Platform extensibility tool.
- com.yantra.ide.rcptools.rcpextn_1.1.0 - This plug-in is used to enable the Rich Client Platform extensibility tool.
- com.yantra.ide.rcptools.uieditor_1.1.0 - This plug-in is used to enable the Rich Client Platform UI Editor for creating Rich Client Platform Composite, and Rich Client Platform plug-in.
- com.yantra.yfc.rcp.common_1.0.0 - This is the base plug-in and is common for all applications of Rich Client Platform. This plug-in is required.
- com.yantra.yfc.rcp.libs - This is the base plug-in of Rich Client Platform libraries and is common for all applications of Rich Client Platform.
- com.yantra.yfc.rcp_1.0.0 - This is the base plug-in and is common for all applications of Rich Client Platform. This plug-in is required.

  Note: You must first copy the required plug-ins in the <*ECLIPSE_HOME*>/plugins folder and then the application-specific plug-ins. Extensibility plug-ins can be included for application extensibility, if required.

Installing the Rich Client Platform Tools Plug-In

About this task

To install the Rich Client Platform Tools plug-in, copy the contents of the folder *INSTALL_DIR*/rcpclient to the *ECLIPSE_HOME*/plugins folder.

ECLIPSE_HOME refers to the Eclipse SDK installation directory.

Rich Client Platform Tools

The Rich Client Platform Tools plug-in contains the following tools:
- Rich Client Platform Command XML Editor—The Rich Client Platform Command XML Editor provides a way to conveniently edit the *Plug-in_id*_commands.ycml file. The Commands XML file is used to create or modify commands and namespaces.
- Rich Client Platform Config XML Editor—The Rich Client Platform Config XML Editor tool provides a way to conveniently edit the locations.ycfg file. The locations.ycfg file contains configuration information for the Rich Client Platform applications. A location configuration and server configuration must be defined to connect the Rich Client Platform application to the server.
- Rich Client Platform Theme Editor—The Rich Client Platform Theme Editor tool provides a convenient way to edit the *Plug-in_id_theme name*.ythm file. The theme file is used to define theme entries for a particular theme.

- Rich Client Platform Wizard Editor—Rich Client Platform Wizard Editor tool is used to conveniently edit the *Plug-in_id*_commands.ycml for creating or modifying the wizard definition. The wizard definition specifies the flow of a wizard.

  Note: To understand how to use Rich Client Platform tools such as the Rich Client Platform Command XML Editor, Rich Client Platform Config XML Editor, and so forth in Eclipse, see the cheat sheets provided by the Rich Client Platform.
- Rich Client Platform UI Wizards—The Rich Client Platform UI Editor is a plug-in that includes several development time database utilities for the Rich Client Platform application. The Rich Client Platform UI Editor provides various wizards for creating these utilities, such as:
  - Rich Client Platform Composite—The Rich Client Platform Composite wizard is used to create standalone Rich Client Platform screens for the Rich Client Platform application.
  - Rich Client Platform Plug-in—The Rich Client Platform Plug-in wizard is used to extend an Eclipse plug-in so that it can be recognized by the Rich Client Platform. The Rich Client Platform Plug-in wizard includes a plug-in file and various Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales-specific resource files such as theme file, configuration file, command file, bundle file, and extension file. The plug-in Java file is used to register the Eclipse plug-in and the Rich Client Platform-specific resource files with the Rich Client Platform. Whenever you run the Rich Client Platform Plug-in wizard on top of an Eclipse plug-in, the newly created bundle activator gets updated. Also, the bundle activator for the plug-in file is placed in the plugin.xml file.
  - Rich Client Platform Extensibility Tool—The Rich Client Platform Extensibility Tool is used to modify the existing screens of a Rich Client Platform application. Using this tool you can add new controls, modify existing controls, and so forth.
- Rich Client Platform Application Plug-in—Extract the Rich Client Platform application compressed file that you want to customize to any directory. You can find the compressed file for a PCA in the

  INSTALL_DIR/rcp/*PCA_DIR*/rcpclient/ directory. *PCA_NAME* refers to the PCA installation directory.

  For example, if you want to customize theSterling Call Center and Sterling Store application, extract the *INSTALL_DIR*/rcp/COM/rcpclient/com.zip file to any directory.

  After you extract all files, copy the content of the Rich Client Platform Application's plug-in folder to the *ECLIPSE_HOME*/plugins folder.

  For example, if you want to customize the Sterling Call Center and Sterling Store application, copy the *TEMP_DIR*/plugins/com.yantra.pca.ycd.rcp_*version* to the *ECLIPSE_HOME*/plugins folder.

  where *TEMP_DIR* is the name of the directory where you have extracted the com.zip file. *ECLIPSE_HOME* refers to the directory where you have installed Eclipse SDK.

View Rich Client Platform Cheat Sheets

About this task

To view the cheat sheets in Eclipse:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar select **Help** > **Cheat Sheets**. The Cheat Sheet Selection window displays.

   Expand **Rich Client Platform: UI Editor** from the list and open the appropriate cheat sheet.

Open the Rich Client Platform UI wizards

About this task

To open the UI wizards:

Procedure

1. Launch the Rich Client Platform application in Eclipse.
2. Press **Ctrl+N**.

   Expand Rich Client Platform Wizards from the list of wizards and open the appropriate wizard.

Creating and Configuring Locations

About this task

To configure locations, ensure that you create the locations.ycfg XML file.

To configure a location, follow these steps:

Procedure

1. In the locations.ycfg XML file, define a Locations root element.
2. Under the Locations root element, define the Location element. In the id attribute of the Location element, specify the location identifier such as DEFAULT, LOCAL, REMOTE, and so forth.
3. Configure the proxy server and application server URL settings for the location. For more information about location configuration settings, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Installing the PlatformInstallation Guide*.

   Note: You must have one Location element with id attribute value of `DEFAULT` and this Location element must have a Config element whose Name attribute is `DEFAULT`.

   When you log in to a Rich Client Platform application using a particular location, the system checks whether or not the loaded location has a "DEFAULT" Config element defined for it. If the selected location has `DEFAULT` Config element, the system loads the that configuration. Otherwise the system loads the `DEFAULT` configuration defined in the `DEFAULT` location.
4. Add locations.ycfg XML file to resources.jar file.
5. Copy the resources.jar file to the *ECLIPSE_HOME*/plugins/ com.yantra.yfc.rcp_*version* directory

   where *ECLIPSE_HOME* refers to the Eclipse SDK installation directory.

Creating a Plug-In Project

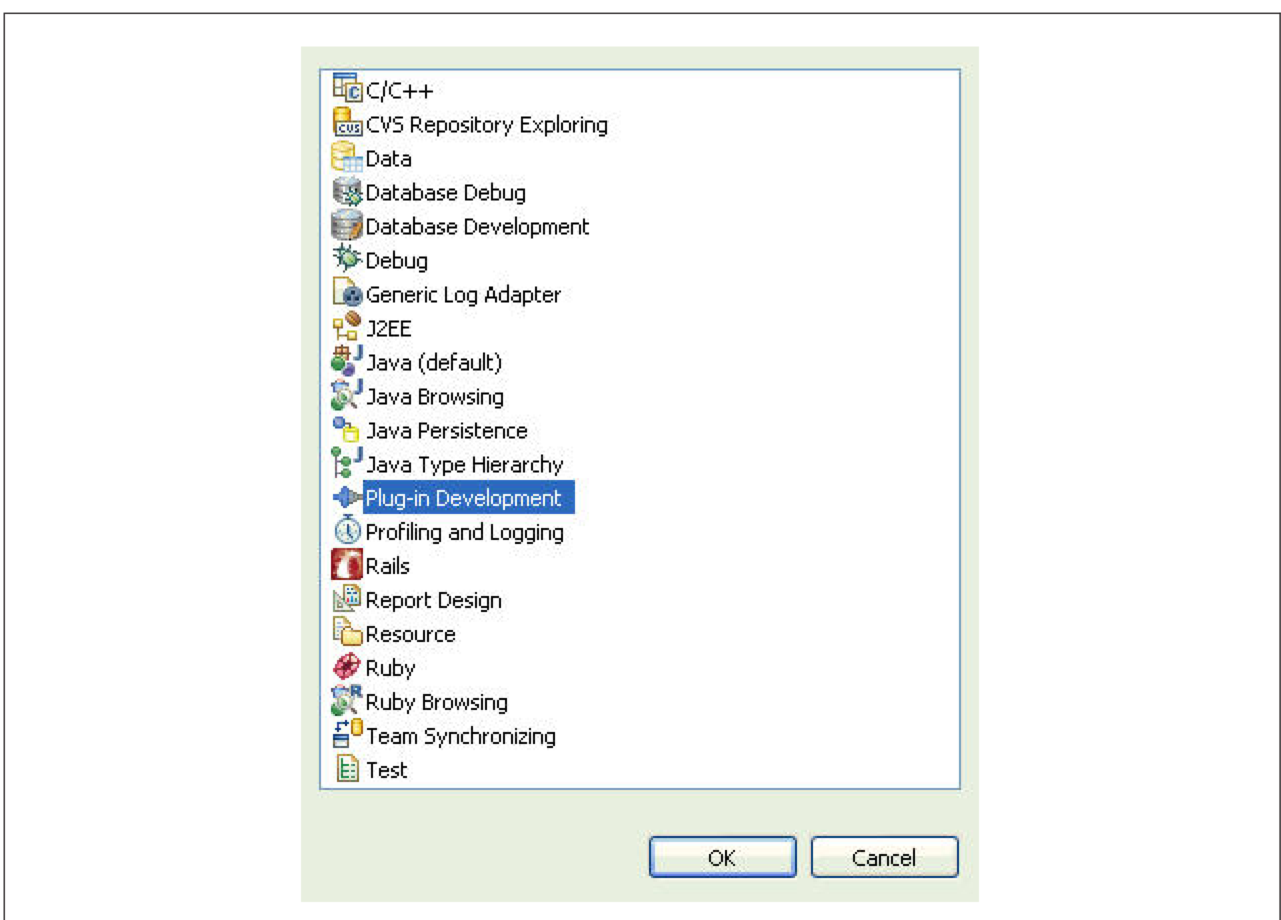
About this task

This section explains how to create a plug-in project.

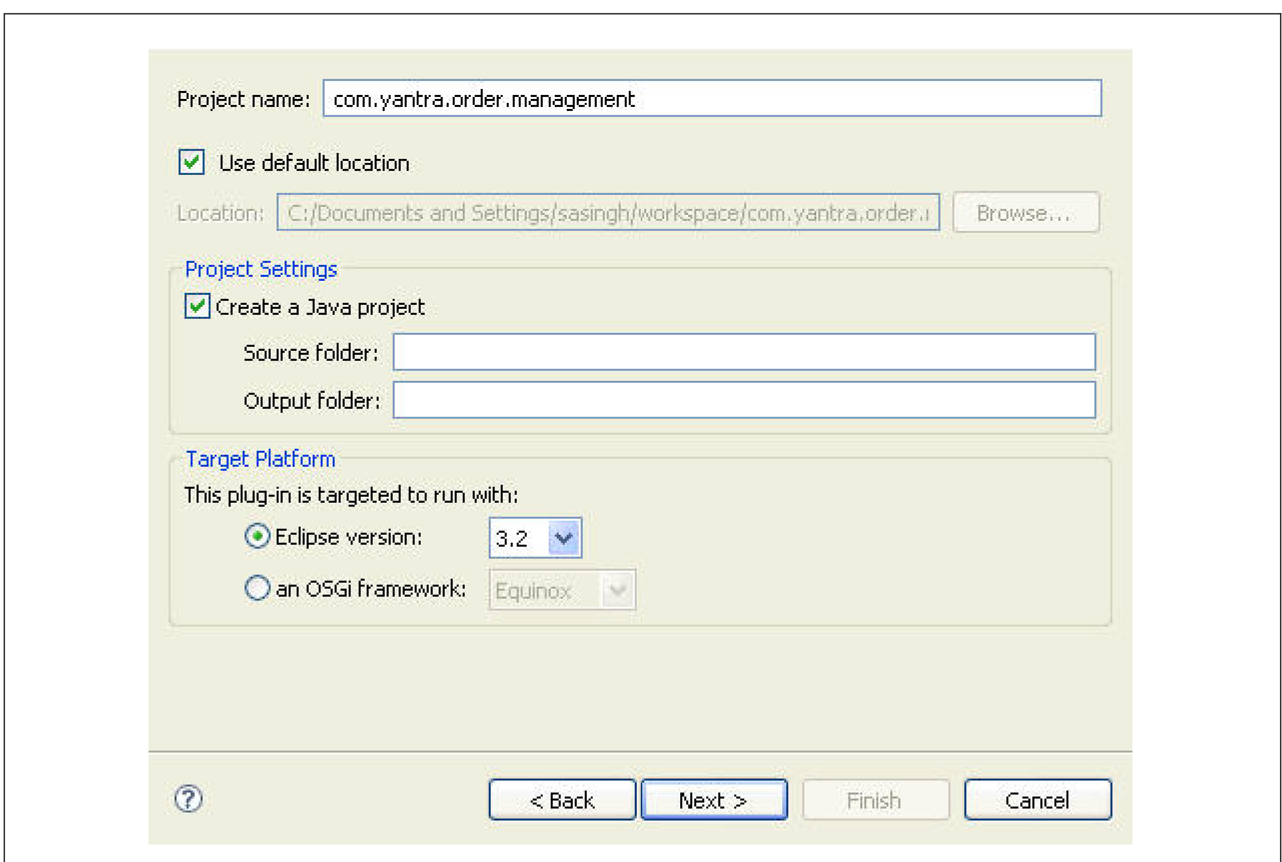
To create a plug-in project:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **WindowOpen PerspectiveOther.....**
   The Select Perspective window displays.



3. From the list of wizards, select **Plug-in Development**.
4. Click **OK**. The Eclipse Workbench opens in Java perspective.
5. From the menu bar, select **File** > **New** > **New Project....** The New Project window displays.
6. From the list of wizards, under Plug-in Development category, select the **Plug-in Project**.
7. Click **Next**. The New Plug-in Project window displays.



Field	Description
Project name:	Enter the name of the new plug-in project.
Use default location	Uncheck this box if you want to specify the path where you want to store the new plug-in project. By default, this box is always checked. IBM recommends that you use the default directory to store the new plug-in project.
Project Settings	
Make sure that the Source folder: and Output folder: text boxes are empty.	
Target Platform	
Eclipse version:	Select **3.3** from the drop-down list.

8. Click **Next**. The Plug-in Content page displays.
9. Click **Next**. The Templates page displays.
10. Click **Finish**. The new plug-in project gets created.

Rich Client Platform Plug-In Wizard

After creating the new plug-in project, run the Rich Client Platform Plug-in wizard on top of the new plug-in project that you created. The Rich Client Platform Plug-in wizard enables you to extend an Eclipse plug-in so that it is easily understood by the Rich Client Platform. The Rich Client Platform Plug-in wizard creates a plug-in Java file and the following Rich Client Platform-specific resource files:

- Bundle files such as *bundle.properties—Used to define bundle entries for internationalizing a Rich Client Platform application.
- Command files such as *commands.ycml—Used to define commands for calling APIs or services to get the required data.
- Theme files such as *theme.ythm—Used to define theme entries for theming a Rich Client Platform application.
- Extension files such as *extn.yuix—Used to store all the extensions made to a Rich Client Platform application.

These resource files allow you to extend the UI and control the behavior of a Rich Client Platform application. The plug-in Java file is used to register the Eclipse plug-in and the Rich Client Platform-specific resource files with the Rich Client Platform.

Running the Rich Client Platform Plug-In Wizard

About this task

To run the Rich Client Platform Plug-in wizard:

Procedure

1. Start the Eclipse SDK.
2. From the menu, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. Expand the plug-in project that you created.
4. Right-click the source folder where you want to store the Rich Client Platform extension plug-in Java file and select **New** > **Other** from the pop-up menu. The New window displays.
5. From the list of wizards, select **Rich Client Platform Wizards** > **Rich Client Platform Plug-in**.
6. Click **Next**. The New Plug-in to Rich Client Platform UI window displays.

Source Folder: \com.yantra.order.management\com\yantra\order\management\plugin Browse...

Plugin Id com.yantra.order.management

Package: com.yantra.order.management.plugin

Plugin File Name NewPlugin.java

This wizard can also optionally generate an application file. If this plugin is also an independent RCP application, choose this option by entering an application file name.

Application File Name

Override if there are existing files with same name?

< Back Next > Finish Cancel

Field	Description
Source Folder:	The folder path that you selected displays. Click Browse to browse to the source folder where you want to store the plug-in java file, if necessary.
Plugin Id	The identifier of the plug-in project which contains the source folder displays.
Package:	The package name displays. If this field is empty, the system considers the source folder as the default package. Note: It is recommended that you do not use a default package with this wizard. The plug-in name is created and prefixed with a dot or period. Therefore, you will encounter an error when you run the application within Eclipse.
Plugin File Name	By default, the NewPlugin.java plugin file name displays. Enter a new plug-in file name, if necessary. This plug-in file registers your resource files such as bundles, themes, commands, and extension files.
Application File Name	Enter the name of an application file name, if necessary.

7. Click **Finish**.

   After you run the Rich Client Platform Plugin wizard on top of a plug-in project, the Rich Client Platform performs the following tasks:

   - Loads the dependent plug-ins. The dependent plug-ins are the plug-ins whose extension points are extended by another plug-in to extend the functionality provided by the Eclipse platform.
   - Implements the YRCPluginAutoLoader extension point. The YRCPluginAutoLoader extension point is provided by the Rich Client

Platform, which defines the order in which plug-ins need to be loaded. The YRCPluginAutoLoader extension point automatically loads the classes within a plug-in during startup in the specified order. All classes that need to be automatically loaded are sorted in ascending order and loaded one at a time.

- Creates the plug-in Java file. The plug-in Java file is stored in the folder you specified. This file is used to register the plug-in you created and the Rich Client Platform-specific resource files.
- Creates the following Rich Client Platform-specific resource files:
- Bundle file of format *bundle.properties
- Commands file of format *commands.ycml
- Theme file of format *theme.ythm
- Extension file of format *extn.yuix

The following illustrates a typical folder structure that has both the plug-in Java file and the Rich Client Platform-specific resource files stored under the package that you specified.

Launching the Rich Client Platform Application in Eclipse

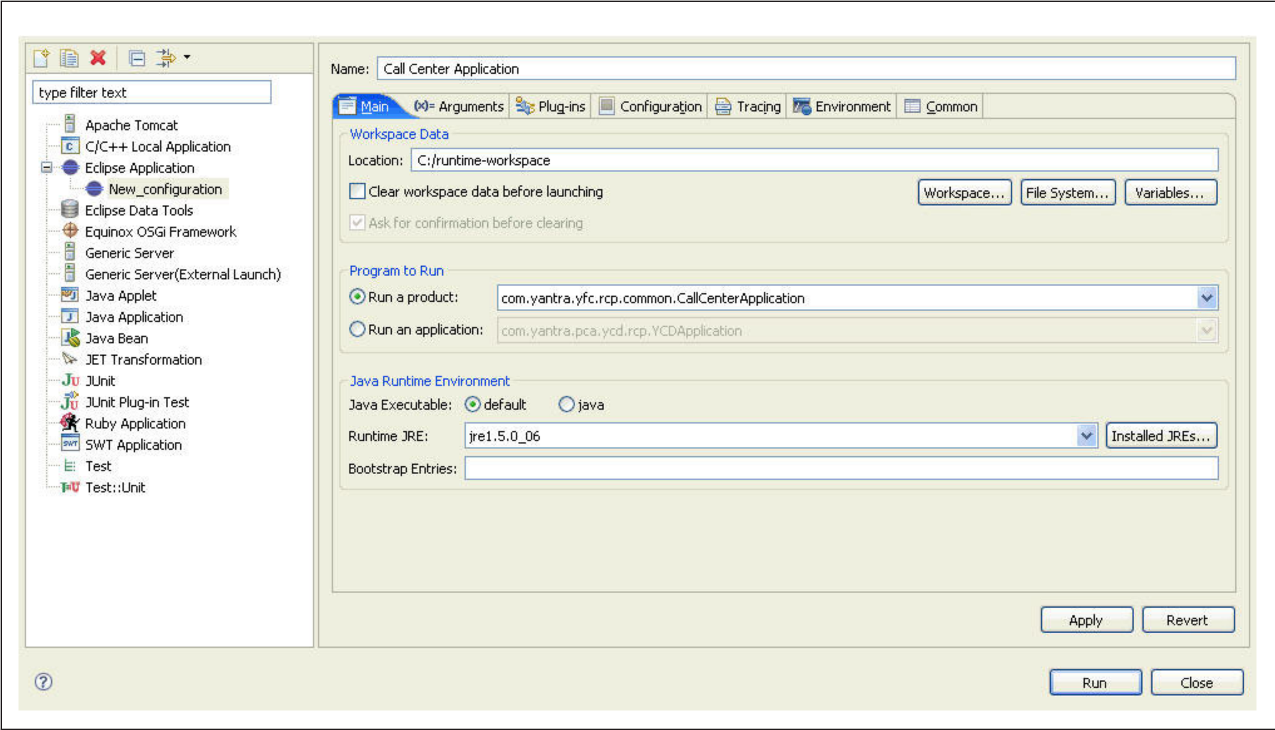
About this task

After you run the Rich Client Platform Plugin wizard on the plug-in project, launch the Rich Client Platform application that you want to customize within Eclipse.

To launch the Rich Client Platform application in Eclipse:

Procedure

1. Start the Eclipse SDK. In the Package Explorer view, you can see the plug-in project that you created.
2. From the menu bar, select **Run** > **Run...**. The Run window displays.



3. In the left panel, right-click and select **New_configuration** from the pop-up menu.
4. In Name, enter the name for the new run configuration. For example, Call Center Application.
5. In the Workspace Data panel, in Location, enter the location where you want to create the runtime workspace. Click **File System...** to browse to the directory where you want to create the runtime workspace.
6. In the Program to Run panel, select **Run a product**: and from the drop-down list select the product identifier of the application you want to customize. For example, if you want to customize theSterling Call Center and Sterling Store application, select the product identifier of the Sterling Call Center and Sterling Store application. For more information about the product identifier for each PCA, see the respective Rich Client Platform based applications *Installation Guide.*
7. In the Plug-ins tab, choose **Choose plug-ins and fragments** to launch from the list.
8. Click **Deselect All**.
9. From the list of plug-ins, expand **Workspace Plug-ins** and select the plug-in project that you created.
10. Expand **Target Platform**. Select the Rich Client Platform application plug-in that you want to customize.

    Note: Ensure you select the plug-in is the same as the Rich Client Platform application whose ID you selected in Step 6.
11. Select the **com.yantra.ide.rcptools.rcpextn** plug-in.
12. Click **Add Required Plug-ins**.
13. Click **Validate plug-in Set**. If you have correctly performed all steps, the Plug-in Validation window displays the message "No problems were detected in the selected set of plug-ins."

14. Click **OK**.
15. Select Configuration tab and check the **Clear the configuration area before launching** box. This clears the cached configuration data saved by Eclipse.
16. Click **Apply**.
17. Click **Run**. The Rich Client Platform application now runs.

    Note: The Rich Client Platform Extensibility tool plug-in depends on some of the Eclipse plug-ins. When you add the Rich Client Platform extensibility tool plug-in, these dependent Eclipse plug-ins are automatically added. Therefore, when you launch a Rich Client Platform application such asSterling Store and Sterling Store application within Eclipse, the system throws the following error messages:

    ```
    Invalid Menu Extension (Path is invalid): org.eclipse.ui.actions.showKeyAssistHandler.
    Invalid Menu Extension (Path is invalid): org.eclipse.update.ui.updateMenu.
    Invalid Menu Extension (Path is invalid): org.eclipse.update.ui.configManager.
    Invalid Menu Extension (Path is invalid): org.eclipse.update.ui.newUpdates.
    ```

    These are known issues and have no bearing on the functioning of an application.

Chapter 4. Customizing the Log In Screen

Customizing the Login Screen

About this task

You can customize login screen for the following:
- Adding additional fields.
- Modifying background color.
- Customizing images

What to do next

Adding Additional Fields to the Login Screen

The Sterling RCP provides the YRCloginDialogExtn extension point, which enables a user to customize the Login Dialog with additional fields. An implementation has to be provided for the YRCLoginDialogExtn extension point using the IYRCLoginDialog interface. The implementation should update the login input document with the requisite attributes or add additional attributes or perform both tasks.

Every extension of the Login Dialog must have the following mandatory attributes:
- ModuleId—The module ID of the application for which the Login Dialog is being extended, for example, the module ID of Sterling Call Center client application is ycd, and the module ID of theSterling Store client application is sop.
- ClassToLoad—The class to be loaded to extend the IYRCLoginDialog interface.
- LoadOrder—When multiple Login Dialog implementations exist for the same application, the LoadOrder attribute decides which Login Dialog implementation must be picked up. The Login Dialog extension with the highest value of the LoadOrder attribute is selected and loaded.

If the implementation for the YRCLoginDialogExtn extension point using the IYRCLoginDialog interface is absent, the application will display the default Login Dialog.

Note: Following are the limitations of the YRCLoginDialogExtn extension point implementation:
- An API cannot be called prior to logging into an application. For example, if a drop-down list is used when customizing the Login screen, the drop-down list cannot be populated by calling an API. In this scenario, the drop-down list can be populated with hard-coded items, or with contents from a file.
- The customization of the Login Dialog must be performed in a separate and independent plug-in that does not include any other customizations, such as those pertaining to bundles, commands, and so on.

Modifying Background Color of the Login Screen

© Copyright IBM Corp. 1999, 2013

You can change the background color of the login dialog using the setLoginDialogueColor (int red, int green, int blue) method of the YRCPlatformUI class.

Customizing Login Screen Images

You can customize the login dialog images using the setLoginDialogueImages(Image leftImage, Image rightImage, Image topImage, Image bottomImage, String moduleId) method of the YRCPlatformUI class.

Chapter 5. Customizing Rich Client Platform Applications

Overview of Customizing Rich Client Platform Applications

The Rich Client Platform supports various ways of customizing a Rich Client Platform application.

When customizing the Rich Client Platform application, copy the standard Rich Client Platform-specific resource files and modify them or create new resource files. Do not modify the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales-specific resource files that are shipped with Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales.

Localizing Rich Client Platform Applications

You can localize a Rich Client Platform application's locale-specific files based on the user's locale. The Rich Client Platform supports bundle and theme locale-specific files. The Rich Client Platform application plug-ins contain bundle file such as *Plug-in_id_name*.properties and theme file such as *Plug-in_id*_theme_name.ythm. For more information about localizing bundle and theme files, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Localization Guide*.

Defining Themes for Rich Client Platform Applications

You can theme the Rich Client Platform application based on the custom theme. To theme your application, at the plug-in level, define new theme entries for controls, text, strings, images, and so forth in the *Plug-in_id_theme name*.ythm file. For more information about theming the Rich Client Platform application, see "Defining Themes".

Extending Rich Client Platform Applications

You can extend the Rich Client Platform application's UI to address specific needs of your business. Extending the Rich Client Platform application can be as simple as defining some additional fields or as advanced as defining an entire new plug-in.

IBM recommends that you extend the Rich Client Platform application by modifying existing screens.

Before you can start extending the Rich Client Platform application using any one of the given ways, make sure that you set up the development environment for performing customizations. For more information about setting up development environment, see the "The Development Environment for Rich Client Platform Applications".

Note: In some screens or editors, the layout of a screen or editor may have changed because of a HF or upgrade. For example, new controls being added, existing controls being hidden, and so forth. To ensure that your extensions are applied on such screens or editors, you will have to rework on positioning these new custom controls, for example, labels, text boxes, radio buttons, and so forth (if necessary).

© Copyright IBM Corp. 1999, 2013

Modifying Existing Screens

Use the Rich Client Platform Extensibility Tool to modify or extend existing screens of the Rich Client Platform application. This tool allows you to modify existing screens by adding or removing text boxes, labels, combo boxes, buttons, table columns, and so forth from the existing forms. You can also add or modify composites and groups. For more information about modifying existing screens, see "Modifying Existing Screens and Wizards".

Note: Whenever out-of-the-box logic for any Application shipped control is overridden through extensibility (i.e. by writing custom logic in extension behavior class of the control), the overridden logic is implemented first followed by the out-of-the-box logic. For example, if the Order Search button logic is overridden to open the customer screen, the customer screen will be opened first and when that screen is closed the order search screen will be opened.

But if out-of-the-box logic for any Application shipped control is overridden to display an error validation dialog, only the error dialog will be opened and out-of-the-box logic will not be implemented.

Modifying Existing Wizards

You can modify the wizard definition XML of the existing wizards by adding new wizard entities (wizard page, wizard rule, or sub-task). You can also modify the flow of a wizard by adding new transitions or overriding the existing transitions. A wizard rule is used to control the flow of a wizard based on certain criteria. The flow of a wizard depends on the output value of a wizard rule. Use the transition lines to transfer control from one wizard page or rule to another wizard page or rule. The system compares the output of the wizard rule with the transition value. Based on the transition value, the system transfers the control to the appropriate wizard page or rule. The sub-task is used to perform a separate sub-task within the wizard flow. For example, you can add a sub-task to the wizard flow. That sub-task can be a separate wizard within the existing wizard.

Creating and Adding New Screens

You can create new Rich Client Platform screens for a Rich Client Platform application in Eclipse using the Visual Editor (VE).

After creating new screens, you can add these to the Rich Client Platform application.

For more information, see *Creating and Adding Screens*.

Building and Deploying Extended Rich Client Platform Applications

After making extensions to a Rich Client Platform application, make sure that you build and deploy the new extensions. You should build and deploy the Rich Client Platform application with all the new plug-ins that you created, resource files that you synchronized, and SSL certificates.

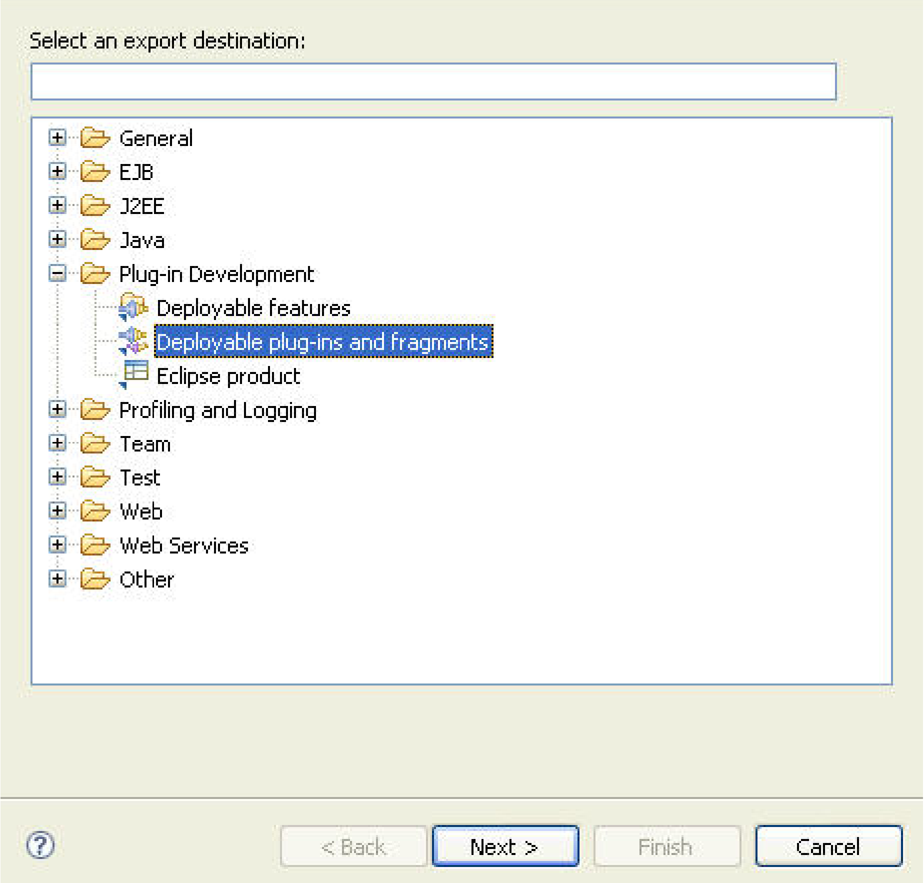
Building Rich Client Platform Extensions

About this task

Building the Rich Client Platform extensions is as follows:

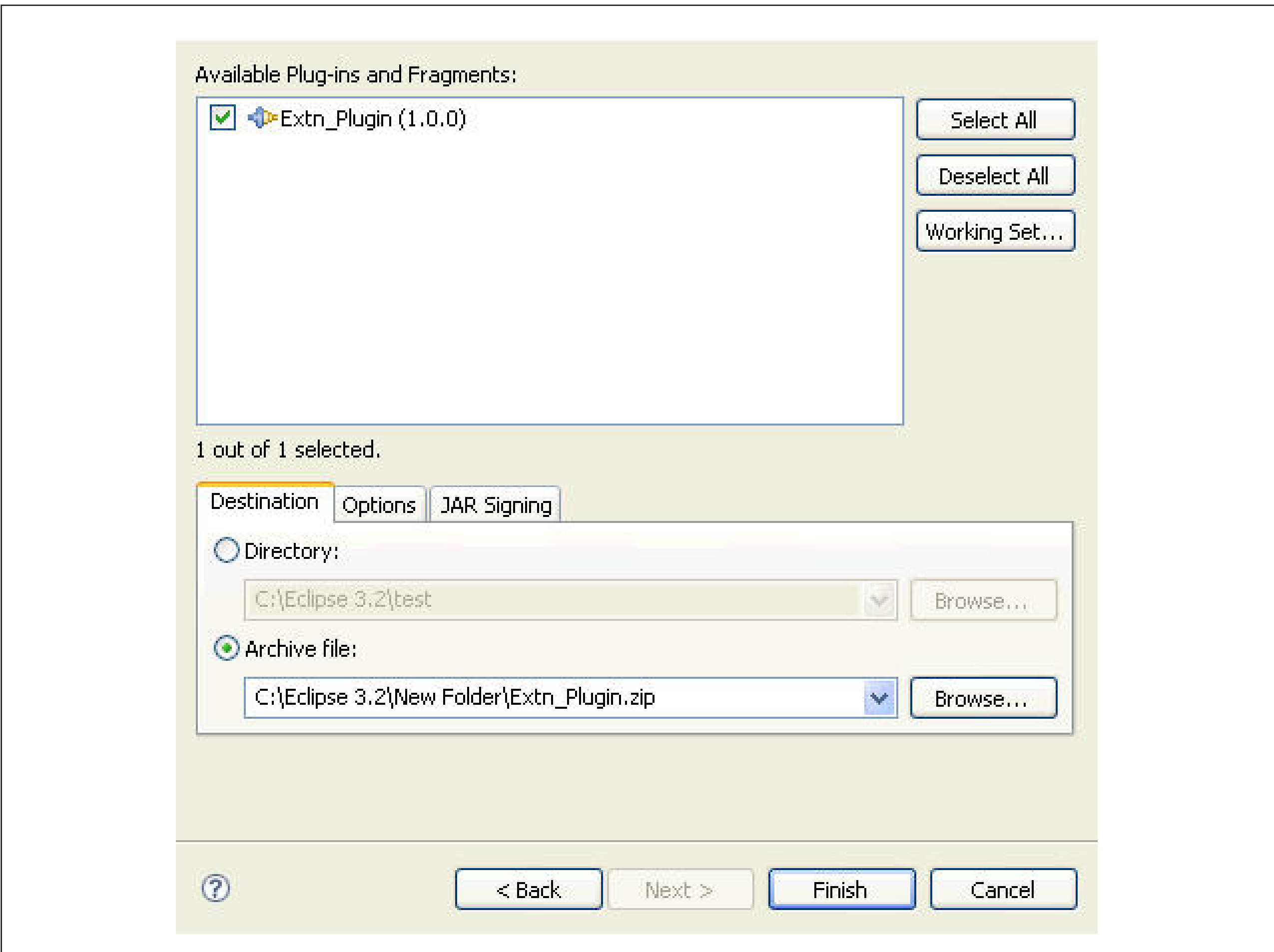
Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. Right-click on the plug-in project that you want to build and deploy.
4. Select **Export...** from the pop-up menu. The Export window displays.



5. From the list of export destinations, under Plug-in Deployment, select Deployable plug-ins and fragments.
6. Click **Next**.

   The Available Plug-ins window displays.



7. In the Destination tab, Choose **Archive file:**.
8. Click **Browse** and browse to the folder where you want to store the exported plug-in compressed file.
9. In the Options tab, make sure that the Package plug-ins as individual JAR archives box is checked.
10. Click **Finish**. The plug-in jar is generated and stored in the plugins folder in the compressed file as specified in step 8.

Deploying Rich Client Platform Extensions

About this task

After you build the Rich Client Platform extensions plugin jar, you must deploy this plug-in.

To deploy the Rich Client Platform extensions, copy the plugin jar that you built to the plugins directory of the *RCP_EXTN_FOLDER* folder and follow the steps as described in the *Deploying and Updating Rich Client Platform Application* chapter of theSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales *Installation GuideInstalling the Platform*.

Chapter 6. Customizing the About Box

Customizing the About Box

About this task

The About Box of a Rich Client application indicates the name of the application and also the version number of the application.

To customize the About Box:

Procedure

1. Create a custom about.properties file in the *INSTALL_DIR*/extensions/plugins/*plug-in-id*/ directory.
2. Edit the about.properties file and all your custom entries in the *name=value* pair format.

   Note: Your custom about.properties file must contain the following entries:
   - Name
   - Version
   - Build

   For example, if you are customizing Sterling Call Center and Sterling Store About Box, the custom about.properties file will look like this:
   ```
   Name=Sterling Call Center and Sterling Store
   Version=8.5
   Build=1201
   ```
3. Register your custom about.properties file with your plug-in. To register your about.properties file, call the registerAboutPluginProperties() method within the plug-in's constructor. For example,
   ```
   YRCPlatformUI.registerAboutPluginProperties("about", ID);
   ```
   where ID is a unique identifier of the plug-in that registers this about.properties file.

   Note: Make sure that the properties file being registered is present in the plug-in project.

   Note: Before calling the registerAboutPluginProperties() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class. For more information, see the *Registering a Plug-In* topic.

© Copyright IBM Corp. 1999, 2013

Chapter 7. Masking Sensitive Customer Information

Methods for Masking Sensitive Customer Information

The Rich Client Platform provides two methods to prevent sensitive information such as credit card numbers, CVV numbers, or passwords from being visible in log messages:

- To mask sensitive information in logging messages, add entries to the customer_overrides.properties file specifying the information to be masked. Because the properties file resides on the server, it is always available to mask information on the client side. See the *Masking Sensitive Information During Logging* topic.
- To obtain greater control over the filtering process, register and implement a custom message filter class. This method eliminates the need to define multiple properties in the customer_overrides.properties file. However, the custom class is stored on the client, so if the class is removed from the client, filtering will no longer occur. It applies only to debug (not timer) files. *See the Masking Sensitive Information During Trace* topic.

© Copyright IBM Corp. 1999, 2013

Chapter 8. Modifying Existing Screens and Wizards

Modifying Existing Rich Client Platform Screens

This section explains how to modify the existing screens of a Rich Client Platform application.

Starting the Rich Client Platform Extensibility Tool

After you set up the development environment, start the Rich Client Platform Extensibility Tool.

Customizing the User Interface

After you start the Rich Client Platform Extensibility Tool, you can customize the existing screen by adding or removing text boxes, labels, combo boxes, buttons, table columns, and so forth. You can also add composites and groups to the screen.

Synchronizing Differences

Whenever you customize an existing screen, you must synchronize the resource files.

Building and Deploying Extensions

After you extend the existing screens, make sure that you build and deploy the new extensions. .

Validating or Capturing Data During API or Service Calls

You can validate or capture additional data during API or Service calls by overriding the preCommand() method of the YRC ExtensionBehavior class. You can also ensure the receipt of notification upon completion of an API or Service call by overriding the preCommand() method of the YRC ExtensionBehavior class.

- preCommand(YRCApiContext apiContext)—To validate the data or capture additional data before calling an API or Service or both, override the preCommand(YRCApiContext apiContext) method in the behavior class. The preCommand(YRCApiContext apiContext) method returns a boolean value. The valid values are "true" or "false". If the value returned is "false", the Rich Client Platform terminates the API or Service call. For example:

```
public boolean preCommand(YRCApiContext apiContext) {
          if("getOrderDetails".equals(apiContext.getApiName())) {
               return false;
          } else {
               return true;
          }
      }
```

- postCommand(YRCApiContext apiContext)—To ensure the receipt of notification upon completion of an API or Service call, override the postCommand(YRCApiContext apiContext) method in the behavior class. Using postCommand() method you can store the API or Service call output and use it at later point in time for incorporating customizations on the screen. For example:

© Copyright IBM Corp. 1999, 2013

```
public void postCommand(YRCApiContext apiContext) {
      System.out.println("Finished api call:"+apiContext.getApiName());
  }
```

Note: The postCommand() method does not prevent the default handling of the API output on the screen.

Modifying Existing Rich Client Platform Wizards

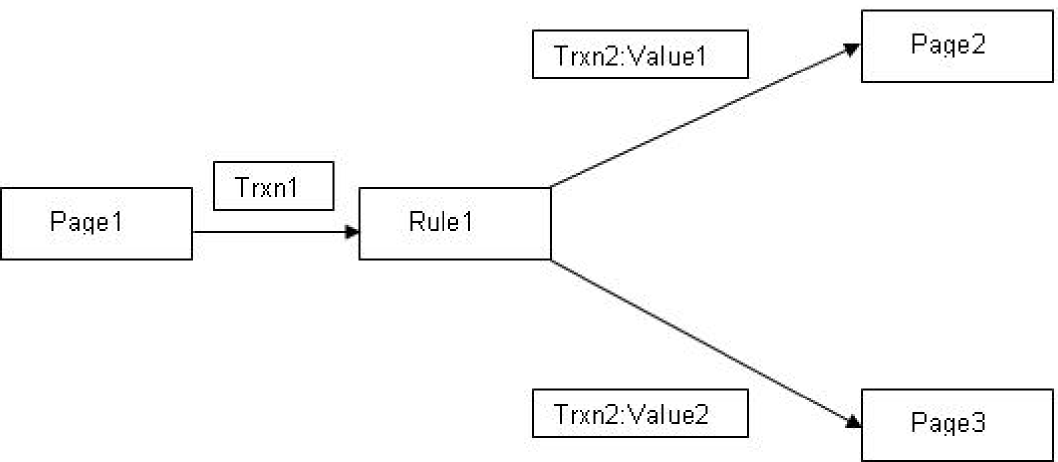
You can modify the existing wizards by creating new wizard entities such as wizard page, wizard rule, or sub-task in the new wizard definition. Define the new wizard definitions in the plug-in project by creating the *Plug-in_id_wizard_name*.ywx file.

Before modifying an existing wizard:

- You must know the form identifier of the wizard you want to extend. After you have identified the form identifier, define the same form identifier in the extended wizard definition using the id attribute of the wizard element.
- To add new wizard rules, you must know the namespace for defining new rules and their values. After you have identified the namespace, define the new rules and their values in the *Plug-in_id_wizard_name*.ywx file.
- To add new sub-tasks, you must know the namespace for defining new sub-tasks. After you have identified the namespace, define the new sub-tasks in the *Plug-in_id_wizard_name*.ywx file.

An example of how to extend an existing wizard is described here.

In the *Plug-in_id_wizard_name*.ycml file, you have an existing wizard definition defined for the following wizard flow:

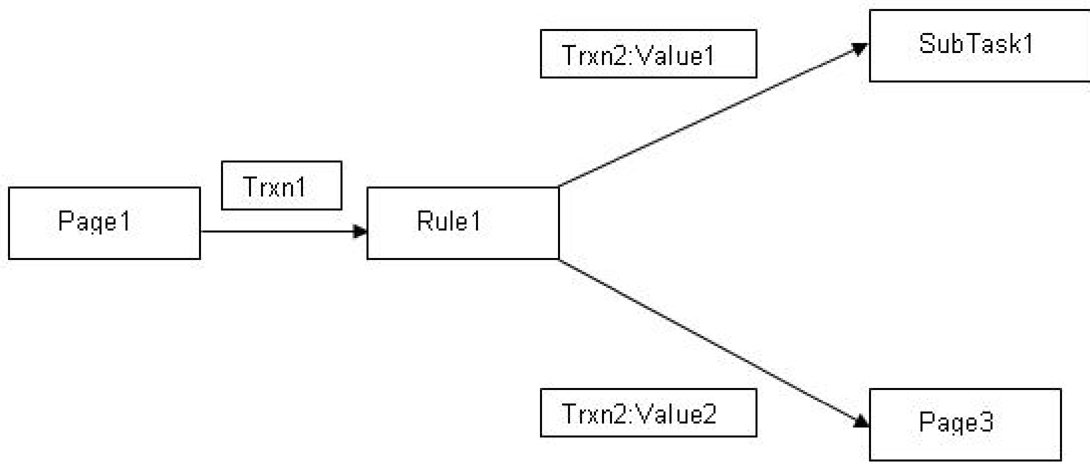


In this wizard flow, the wizard starts from a wizard page (Page1) and transitions to a wizard rule (Rule1). The wizard rule (Rule1) computes some values and returns these values, based on which the control is transferred to two different wizard pages (Page2 and Page3). For Value1, the wizard transitions from Rule1 to Page2, and for Value2, the wizard transitions from Rule1 to Page3.

The sample *Plug-in_id_wizard_name*.ycml XML for the existing wizard definition is as follows:

```
<forms>
 <form Id="com.yantra.pca.ycd.rcp.alert.wizard.YCDAlertWizard">
  <namespaces>
  <namespace type="input" name="Rule" templateName="getRule"/>
  <namespaces>
 <Wizard>
  <WizardEntities>
    <WizardEntity id="Page1">
          impl="java:com.yantra.yfc.rcp.wizard.pages.AlertWizPage1"
          type="PAGE" xPos="340" yPos="200"  start="true">
   </WizardEntity>
   <WizardEntity id="Rule1">
           impl="java:com.yantra.yfc.rcp.wizard.rules.AlertWizRule1"
           type="RULE" xPos="40" yPos="200">
        <Namespace name="Rule"/>
        <Output value="value1"/>
        <Output value="value2"/>
   </WizardEntity>
   <WizardEntity id="Page2">
          impl="java:com.yantra.yfc.rcp.wizard.pages.AlertWizPage2"
          type="PAGE" xPos="140" yPos="200" last="true">
   </WizardEntity>
   <WizardEntity id="Page3">
          impl="java:com.yantra.yfc.rcp.wizard.pages.AlertWizPage3"
          type="PAGE" xPos="140" yPos="200" last="true">
   </WizardEntity>
   </WizardEntities>
   <WizardTransitions>
      <WizardTransition id="Trxn1" source="Page1" target="Rule1"/>
      <WizardTransition id="Trxn2" source="Rule1">
         <Output target="Page2" value="value1">
      </WizardTransition>
      <WizardTransition id="Trxn2" source="Rule1">
          <Output target="Page3" value="value2">
      </WizardTransition>
   <WizardTransitions>
  <Wizard>
 </form>
</forms>
```

To extend the existing wizard flow, for Value1, replace the existing transition from Rule1 to Page2 with the transition from Rule1 to SubTask1 as follows:



Create the extended wizard definition in the *Plug-in_id_wizard_name*.ywx file.

Retrieve Wizard and Namespace Information

About this task

To get the wizard and namespace information:

Procedure

1. In the Rich Client Platform application, navigate to the wizard you want to extend.
2. In the Rich Client Platform Extensibility Tool, view the screen information. The wizard and namespace information is displayed in the screen information window.

Creating an Extended Wizard Definition

About this task

This section explains how to create an extended wizard definition in a Rich Client Platform application.

To create an extended wizard definition:

Procedure

1. Create a new *.ywx XML file and save it in the plug-in project that you created when setting up the development environment, for example, *Plug-in_idwizard_name*.ywx.
2. Start the Eclipse SDK.
3. In the Navigator view, expand the plug-in project that you created.
4. Right-click the newly created *.ywx file and select **Open With** > **Text Editor** from the pop-up menu.
5. Create the Wizards root element.
6. In the applicationId attribute, specify the application identifier of the Rich Client Platform application whose wizard you want to extend.

   For more information about the application IDs of a Rich Client Platform application, see the corresponding Rich Client Platform application documentation.
7. Create the Wizard element under the Wizards root element.
8. In the id attribute, specify the form identifier of the wizard you are extending.
9. Create the WizardEntities element under the Wizard element.
10. Create the required new wizard entities, such as wizard rule, wizard page, or sub-task under the WizardEntities element. For example, create a new sub-task (SubTask1).
11. Create and override the required wizard transitions under the WizardEntities element. For example, override the existing transition, Trxn2, with Value1 for transition from Rule1 to SubTask1.
12. Close the Wizard element.
13. Close the Wizards root element.

    The sample *Plug-in_idwizard_name*.ywx XML file for the extended wizard definition is as follows:

```
<Wizards applicationId="YFSSYS00011">
  <Wizard id="com.yantra.pca.ycd.rcp.alert.wizard.YCDAlertWizard">
  <WizardEntities>
```

```
    <WizardEntity id="SubTask1">
             ⌂impl="com.yantra.yfc.rcp.wizard.subtasks.AlertSubTask1"
              type="WIZARD" xPos="340" yPos="200" last="true"/>
  </WizardEntities>
  <WizardTransitions>
    <WizardTransition id="Trxn2" source="Rule1">
      <Output target="SubTask1" value="value1">
  <WizardTransitions>
 </Wizard>
</Wizards>
```

The id attribute of the wizard entity contains the form identifier of the wizard that you extended. For the new sub-task (SubTask1), a new WizardEntity element in the WizardEntities element is created.

In the existing wizard definition, the Trxn2 with value1 defines the transition from Rule1 to Page2. In the new wizard definition, override this transition with the new target, which is SubTask1.

Registering the Wizard Extension File

About this task

After creating the extended wizard definition in the newly created *<Plug-in_id><wizard_name>*.ywx file, you must register this file with your plug-in. To register your *.ywx file, call the registerWizardExtensions() method within the plug-in's constructor. For example,

```
YRCPlatformUI.registerWizardExtensions("<Plug-in_id>_<wizard_name>",
ID)
```

where *Plug-in_idwizard_name* is the name of your wizard extension file without the ".ywx" extension. ID is a unique identifier of the plug-in that registers this wizard extension file.

Note: Before calling the registerWizardExtensions() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class.

Creating the Wizard Entity

You must create the implementation Java class for the new wizard entity that you add to the extended wizard definition. This can be a wizard rule, a wizard page, or a sub-task. This implementation class is specified in the wizard extension file using the impl attribute of the WizardEntity element.

- If you are adding a new wizard page, you must create the implementation Java class for the wizard page. For more information about creating a new wizard page class, see "Adding a Page to a Wizard Definition".
- If you are adding a new wizard rule, you must create the implementation Java class for the wizard rule. For more information about creating a new wizard rule, see "Adding a Rule to a Wizard Definition".
- If you are adding a sub-task, the implementation class specified in the Impl property of the sub-task should point to a separate subtask that can be run independently as a task.

Modifying the Wizard Extension Behavior

If you have already created the wizard extension behavior class, do the following:

- If you are adding a new wizard page, return an instance of the new wizard page in the createPage(String pageIdToBeShown, Composite pnlRoot) method of the wizard extension behavior class. For example:

```
public IYRCComposite createPage(String pageIdToBeShown) {
IYRCComposite page=null;
If(pageIdToBeShown.equalsIgnoreCase(AlertWizPage2.FORM_ID))
{ AlertWizPage2 temp = new AlertWizPage2(new Shell(Display.getDefault(), SWT.NONE);
  page = temp;
}
  return page;
}
```

- If you are adding a new sub-task, return an instance of the new sub-task in the createChildWizard(String wizardPageFormId, Composite pnlRoot, YRCWizardContext wizardContext) method of the wizard extension behavior class.

  A sub-task can be a wizard that can either be inserted between two wizard entities or the last entity in the wizard flow. If a sub-task is inserted between two wizard entities, the sub-task should display the **Next** button for navigation to the next wizard entity. If the sub-task is the last entity in the wizard flow, the sub-task should display the **Finish** button to end the wizard. This information must be passed to the context object (YRCWizardContext). A context object is used to control the flow of data between the parent wizard and the sub-task, and contains the input to the sub-task. If there is an output to the sub-task, it can be set in context and passed back to the parent wizard. Because the context object utility methods display the appropriate buttons for navigation, these methods must have the position information in the parent wizard to display the proper navigation buttons. For example:

```
public YRCWizard createChildWizard(String wizardPageFormId,
Composite pnlRoot, YRCWizardContext wizardContext){
return
null;
}
```

Chapter 9. Creating and Adding Screens

About Creating a Rich Client Platform Composite

After you set up the development environment, start creating the new Rich Client Platform screen. The Rich Client Platform provides features that enable you to create Rich Client Platform screens.

The Rich Client Platform composite consists of:

- Composite File—The composite Java file handles the UI. In the composite java file, write the code for naming, binding, localizing, and theming controls.
- Behavior File—The behavior Java file handles the functionality or behavior of the screen. In the behavior Java file, write the code for calling APIs or services and getting or setting the XML model for populating the bound controls.

  The Rich Client Platform provides the Rich Client Platform Composite wizard for creating the Rich Client Platform composite. The Rich Client Platform Composite wizard creates an empty composite. You need to design the composite by adding appropriate controls as needed. After designing the composite, name, bind, localize, and theme controls that you add to the composite.

Creating a Rich Client Platform Composite Using the Rich Client Platform Composite Wizard

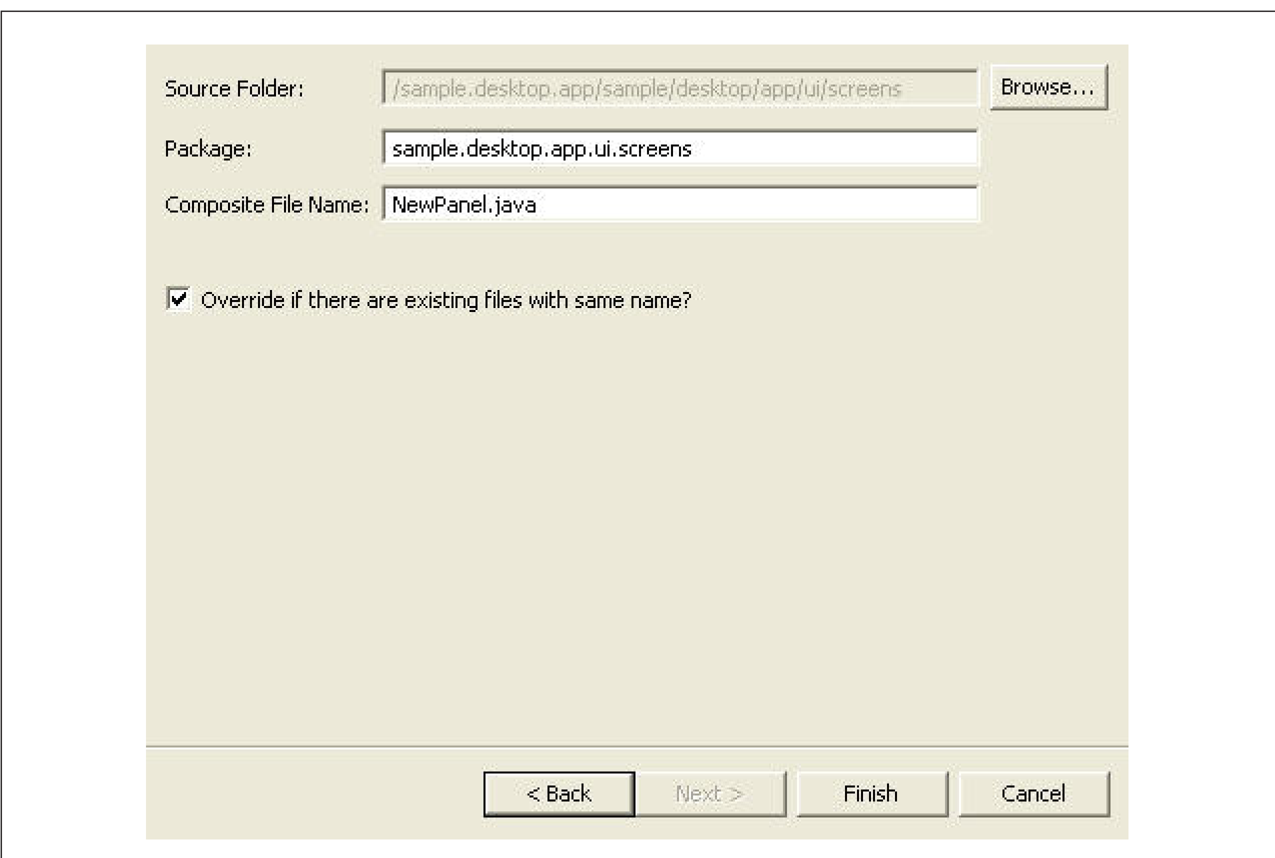
About this task

To create a Rich Client Platform composite using the Rich Client Platform Composite wizard:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To store the screens, right-click on the folder.
4. Select **New** > **Other...** from the pop-up menu. The New window displays.
5. From the list of wizards, select **Rich Client Platform Wizards category** > **UI Wizards** > **Rich Client Platform Composite**.
6. Click **Next**. The New Rich Client Platform Composite window displays.

© Copyright IBM Corp. 1999, 2013



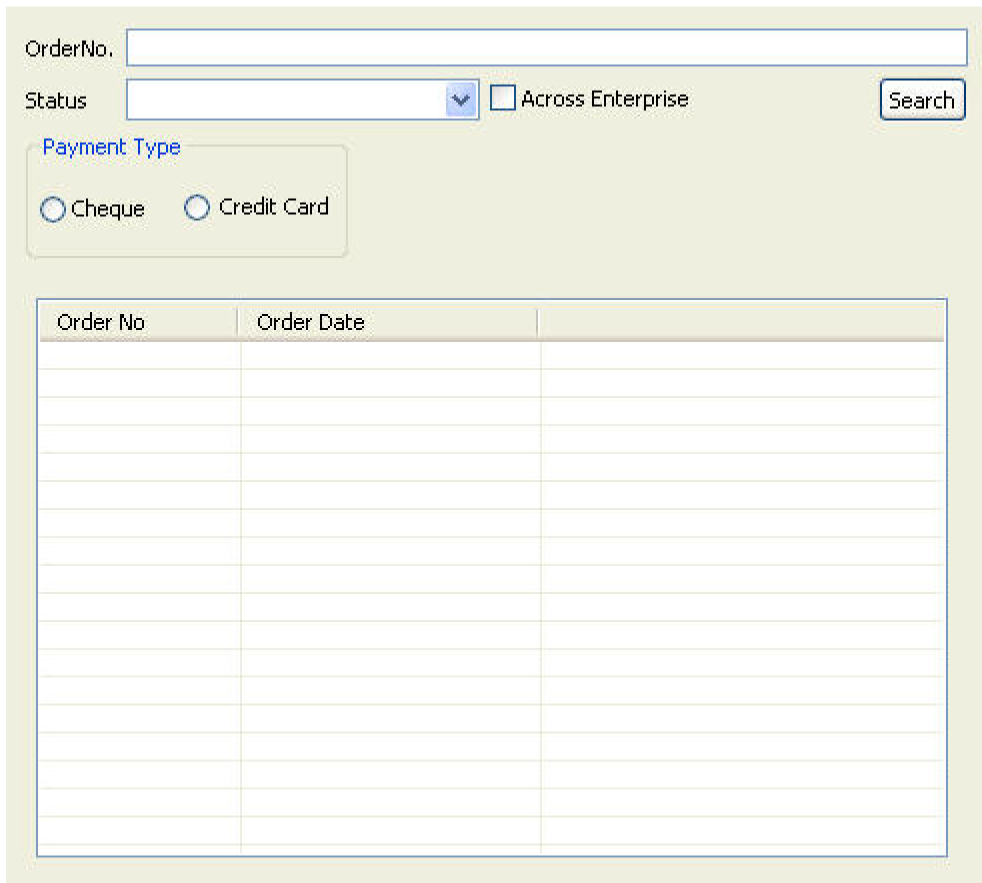
Field	Description
Source Folder:	The path of the folder that you selected automatically displays. Click Browse to browse to the source folder where you want to store the composite and behavior java files.
Package:	Enter the name of the package in which you want to store the composite and behavior java files (if necessary). This helps you to easily manage the directory structure of your plug-in project. If not specified, the system automatically creates the composite java file with the default package name.
Composite File Name:	By default, the NewPanel.java composite file name displays. Enter a new composite file name, if necessary.

7. Click **Finish**. The system creates a composite file and behavior file in the specified source folder. These files are stored in the package that you specified. The following figure illustrates a typical folder structure that has both Java files stored under the package name.

About Designing a Rich Client Platform Composite

You can customize the layout and alignment of your screen as needed. In Visual Editor (Eclipse plug-in), use the Standard Widget Toolkit (SWT) to design UIs.

The Visual Editor enables you to work with layout managers and easily design your screen. This section explains how to create a simple Search and List screen with an example. The following figure depicts a sample Search and List screen.



You can divide the Search and List screen into the following panels:

- Search Criteria panel—This panel contains controls that are used to get input from the user. This may include text boxes, combo boxes, radio buttons, checkboxes, and so forth. When you click the search button, the appropriate API is called with contents in the controls as input to the API. The API output is displayed in the Search Results panel.
- Search Results panel—This panel displays the search results. If the results returns multiple items, you can show the items in a table. Otherwise, you can display data using suitable controls.

Creating the Search Criteria Panel for a Rich Client Platform Composite

About this task

To create the Search Criteria panel for the Search and List screen:

Procedure

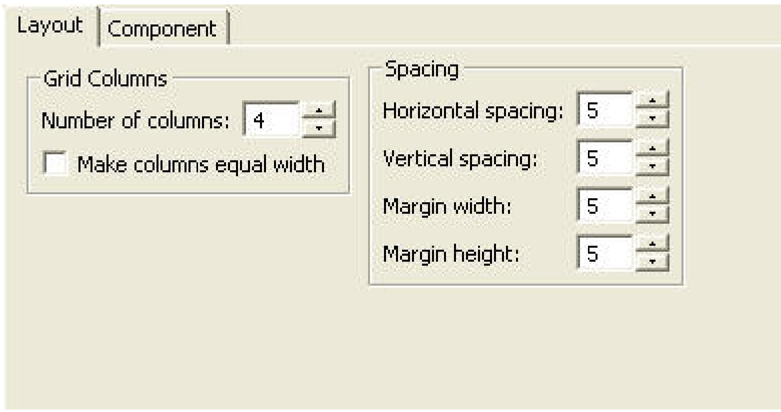
1. Start the Eclipse SDK.
2. In the navigator window, expand the plug-in project that you created.

3. Expand the folder where you have stored the Rich Client Platform composite file.
4. Right-click the Rich Client Platform composite file and select **Open With** > **Visual Editor** from the pop-up menu. The composite file opens in the Visual Editor UI. The Java Beans view automatically opens on the left-hand side of the Eclipse workbench along with other views. Otherwise, manually open the Java Beans View by selecting **Window** > **Show View** > **Other....** From the list of views under Java, select Java Beans.

   In the Java Beans view, you can view the hierarchy of SWT containers and controls.

   In the Properties view, you can view properties and values of the selected containers (composite and group) and controls (labels, text boxes, combo boxes, and so forth).

   You can select containers or controls from the Visual Editor UI or Java Beans View.
5. From the Java Beans view, select **this composite**.
6. From the Properties view, in the layout property, select **FillLayout** from the drop-down list.
7. From the Java Beans view, select **pnlRoot** composite.
8. From the Properties view, select the **GridLayout** value from the drop-down list for the layout property.
9. From the Palette, click **SWT Containers**.
10. Select **Composite** and place it in the pnlRoot composite. The Name pop-up window displays.
11. Enter the name for the Composite. For example, `cmpSearchCriteria`.
12. From the Properties view, select the **GridLayout** value from the drop-down list for the layout property.
13. Right-click the cmpSearchCriteria composite, and select the **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.



14. In the Grid Columns panel, in Number of columns:, enter `4`.
15. Select the Component tab. In the Fill panel, click to fill the excess horizontal space.
16. In the Grab Excess panel, click to grab the excess horizontal space.

17. Now add various controls to the cmpSearchCriteria composite. For more information about adding controls to the cmpSearchCriteria composite, see the *Adding Controls to the Search Criteria Panel for a Rich Client Platform Composite* topic.
18. Bind the controls to display the required data. For more information about binding controls, see the *Binding Controls and Classes for Rich Client Platform Screens* topic.

Adding Controls to the Search Criteria Panel for a Rich Client Platform Composite

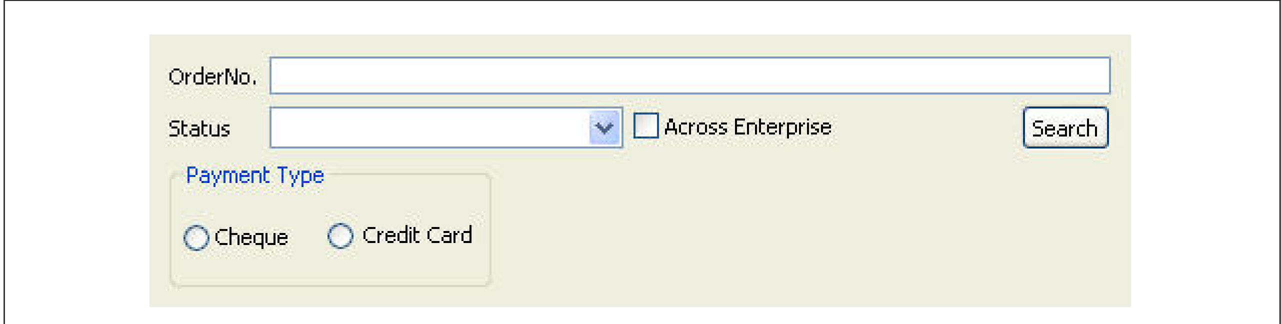
About this task

To add various controls to the composite follow these steps:

Procedure

1. From the Palette, click **SWT Controls**.
2. Select **Label** and place it in the cmpSearchCriteria composite. The Name pop-up window displays.
3. Enter the name of the Label. For example, `lblOrderNo`.
4. In the Properties view, enter the text property value as `OrderNo`.
5. Right-click the lblOrderNo label and select **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.
6. Select the Component tab. In the Fill panel, click to fill the excess horizontal space. Click .
7. From SWT Controls, select **Text** and place it after the lblOrderNo label. The Name pop-up window displays.
8. Enter the name for the Text. For example, `txtOrderNo`.
9. Right-click the txtOrderNo text box and select **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.
10. Select the Component tab. In Span panel, in Horizontal, enter `3`.
11. In the Fill panel, click to fill the excess horizontal space and click .
12. From SWT Controls, select **Label** and place it after the txtOrderNo text box. The Name pop-up window displays.
13. Enter the name for the Label. For example, `lblStatus.`
14. In the Properties view, enter the text property value as `Status`.
15. Right-click the lblStatus label and select **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.
16. Select the Component tab. In the Fill panel, click to fill the excess horizontal space and click .
17. From SWT Controls, select **Combo** and place it after the lblStatus label. The Name pop-up window displays.
18. Enter the name for the Combo. For example, `cmbStatus`.
19. Right-click the cmbStatus composite, and select the **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.

20. Select the Component tab. In Grab Excess panel, click to grab the excess horizontal space.
21. In the Fill panel, click to fill the excess horizontal space. Click .
22. From SWT Controls, select **CheckBox** and place it after the cmbStatus combo box. The Name pop-up window displays.
23. Enter the name of the CheckBox. For example, `chkAcrossEnterprise`.
24. In the Properties view, enter the text property value as `Across Enterprise`.
25. Select the Component tab. In the Fill panel, click to fill the excess horizontal space. Click .
26. From SWT Controls, select **Button** and place it after the chkAcrossEnterprise check box. The Name pop-up window displays.
27. Enter the name of the Button. For example, `btnSearch`.
28. In the Properties view, enter the text property value as `Search`.
29. Right-click the btnSearch button and select **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.
30. Select the Component tab. In the Alignment panel, click to right align the btnSearch button.
31. In the Grab Excess panel, click to grab the excess horizontal space.
32. From SWT Containers, select **Group** and place it after the btnSearch command button. The Name pop-up window displays.
33. Enter the name of the Group. For example, `grpPaymentType`.
34. In the Properties view, select the GridLayout value from the drop-down list for the layout property.
35. Enter the text property value as `Payment Type`.
36. Right-click the grpPaymentType group and select **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.
37. In the Grid Columns panel, in Number of columns:, enter `2`.
38. Select the Component tab. In Span panel, in Horizontal, enter `4`.
39. From SWT Containers, select **RadioButton** and place it inside the grpPaymentType group. The Name pop-up window displays.
40. Enter the name for the RadioButton. For example, `rdbtnCheck`.
41. In the Properties view, enter the text property value as `Check`.
42. Add another radio button and enter the name for the RadioButton. For example, `rdbtnCreditCard`.
43. In the Properties view, enter the text property value as `Credit Card`.
44. Click . The Search Criteria panel is created as shown:



Creating the Search Result Panel for a Rich Client Platform Composite

About this task

To create the Search Results panel for the Search and List screen:

Procedure

1. From the Palette, click **SWT Containers**.
2. Select **Composite** and place it under the pnlRoot composite. The Name pop-up window displays.
3. Enter the name for the Composite. For example, `cmpSearchResult`.
4. From the Properties view, select the **GridLayout** value from the drop-down list for the cmpSearchResult composite.
5. Right-click the **cmpSearchResult** composite, and select the **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.
6. Select the Layout tab. In Grid Columns panel, in Number of columns, enter 1.
7. Select the Component tab. In Fill panel:
   - Click to fill the excess horizontal space.
   - Click to fill the excess vertical space.
8. In Grab Excess panel:
   - Click to grab the excess horizontal space.
   - Click to grab the excess vertical space.
   - Click .
9. Create a standard table in the cmpSearchResult composite.
10. Click . The following Search Results panel is created:

11. Bind the table and table columns with the required data.

Displaying Paginated Results in a Rich Client Platform Composite

You can display paginated results in a Search and List composite. The Search and List composite that wants to display the paginated results must implement the IYRCPaginatedSearchAndListComposite interface and return an instance of the YRCPaginationData object. This YRCPaginationData object should contain the pagination strategy that you want to use, along with the name of the table in which you want to display the paginated results. The YRCPaginatedData is internally used by the system to make a pagination call to the getPage API in order to get the pagination results. Use one of the following pagination strategies to get the paginated results:

- GENERIC
- ROWNUM
- RESULTSET
- NEXTPAGE

By default, the GENERIC pagination strategy is used to get the paginated results.

Note: Screens that make pagination calls should set the YRCApiContext.setPaginationRequired property to "true". If not set to "true", a normal API call is performed.

If you try to use a feature that is not supported in a particular pagination strategy, the system throws an YRCPaginationException exception.

For more information about the getPage API and various pagination strategies, see the Javadocs.

Page Size

To configure the page size for displaying the paginated data, use the *INSTALL_DIR*/properties/customer_overrides.properties file to set the yfc.ui.ListPageSize property.

Note: If in the customer_overrides.properties file, the yfc.ui.ListPageSize attribute is not set. The system defaults the page size to 50.

You can also set this property during application initialization by calling the setpageSize() of the YRCPaginationData class.

YRCPaginatedData

Return an instance of the YRCPaginationData class with the following parameters:

- paginationStrategy(int)—The pagination strategy that you want to use to get the paginated results.
- resultsTable(Table)—The name of the table in which you want to display the paginated results.

YRCPaginationException

Return an instance of the YRCPaginationException class to throw an exception to indicate that a particular pagination strategy does not support this feature. The exception is thrown when the system attempts to call the getPage API, and either the pagination is not supported for that particular screen or composite, or the pagination data is null.

IYRCPageNavigator

To get a handle for the various navigation operations for paginated results, you must implement the page navigation methods. To implement page navigation:

1. The screen for which the pagination is required must implement the IYRCPaginatedSearchAndListComposite interface.
2. Set the value of the setPaginationRequired attribute to true for YRCApiContext of the API that is being called for pagination, which populates the paginated data to a table.
3. Users must create a user interface for any of the following, depending on the pagination strategy and the functionality:
   - Next Page—To navigate to the next page in the paginated result set, use the showNextPage() method of the YRCPaginationNavigator class. Pass the pagination data to this method.
   - Previous Page—To navigate to the previous page in the paginated result set, use the showPreviousPage() method of the YRCPaginationNavigator class. Pass the pagination data to this method.
   - Goto Page *PAGE_NO*—To navigate to a particular page in the paginated result set, use the gotoPage() method of the YRCPaginationNavigator class. Pass the pagination data and the page number to this method.
4. To obtain page navigation controls such as next page, or previous page, or goto page, call the corresponding method in the IYRCPageNavigator interface.
5. For example, To obtain the next page, use the following:

```
btnNext.addSelectionListener(new org.eclipse.swt.events.SelectionAdapter()
{ public void widgetSelected(org.eclipse.swt.events.SelectionEvent e)
{ try { myBehavior.getPageNavigator().showNextPage(getPaginationData()
```

```
);
} catch (YRCPaginationException e1) { myBehavior.trace(e1);
}
}
});
```

Here, btnNext is the control which navigates to the next page, on click.

Note: Since the pagination API is called asynchronously, the showNextPage() method will not return any output till the API call is completed. The method, handleAPICompletion() for the pagination API returns the pagination data.

Note: Make sure that the pagination API is called prior to calling any of the methods in the IYRCPageNavigator interface. Also, the getPaginatedData() method implementation should return the same instance of YRCPaginatedData method every time, since it contains the cumulative information pertaining to the corresponding screen pagination. For example,

```
YRCPaginationData data;
   public YRCPaginationData getPaginationData() {
     if(data == null)
        data = new YRCPaginationData(IYRCPaginationConstants.YRC_GENERIC_
        PAGINATION_STRATEGY, tblSearchResults);
     return data;
}
```

Server-Side Sorting

You can perform server-side sorting for a table by calling the performSort() method of the IYRCPageNavigator interface. Pass the pagination data to this method.

You can also perform server-side sorting for a table by right-clicking the Table column and selecting Sort from the pop-up menu.

Note: To get the pop-up menu for server side sorting, you must call the setServerSortBinding() method of the YRCTblClmBindingData class and pass the XPath of the attribute (on which you want perform the sort operation) to the method.

Creating Tables for Rich Client Platform Screens

The Rich Client Platform supports two types of tables, standard tables and editable tables. As the name suggests, you can modify the data in an editable table, but not in a standard table.

Creating Standard Tables

About this task

You can create a standard table and add columns to this table.

To create a standard table:

Procedure

1. From the Palette, click **SWT Controls**.
2. Select **Table** and place it in a composite. The Name pop-up window displays.
3. Enter the name for the Table. For example, `tblSearchResults`.
4. Right-click the tblSearchResults table, and select the **Customize Layout...** from the pop-up menu. The Customize Layout pop-up window displays.

5. Set the layout properties such as Fill, Grab Access, and so forth as needed.

Adding Columns to the Standard Table

About this task

To add columns to a table:

Procedure

1. From the Palette, click **SWT Controls**.
2. Select **TableColumn** and place it in a table. The Name pop-up window displays. You can add as many columns as you want in a table.
3. Enter the name for each TableColumn that you add to a table. For example, `tblcolOrderNo`.
4. Bind the table and table columns with the data. For more information about binding a standard table, see the *Creating a Binding Object for a Standard Table* topic.

Creating Editable Tables

About this task

To create an editable table:

Procedure

1. Create a standard table.
2. To change the standard table to an editable table, associate each table column to a specific cell editor.

   Note: You must write the code for creating an editable table in the Rich Client Platform composite class.

   Create an array of cell editors [ ] of size that is equal to number of columns. For example:

   ```
   String[] editors = new String[noOfColumns];
   ```

   The Rich Client Platform supports the following cell editors that are defined in YRCInternalConstants class:

   - YRCInternalConstants.YRCComboBoxCellEditor
   - YRCInternalConstants.YRCTextCellEditor
   - YRCInternalConstants.YRCCheckBoxCellEditor
3. Create a cell editor and associate with a column. This column acts as an editable cell. For example:

   ```
   editors[columnIndex1] = YRCConstants.YRC_COMBO_BOX_CELL_EDITOR;
   editors[columnIndex2] = YRCConstants.YRC_TEXT_BOX_CELL_EDITOR;
   ```

   Note: When creating a combo box cell editor you must create a YRCComboBindingData binding object and set the appropriate bindings.
4. After creating all cell editors, set the CellTypes for the table with the cell editor array as the input argument. For example:

   ```
   tableBindingData.setCellTypes(editors);
   ```
5. Bind the table and table columns with the required data.

Naming Controls for Rich Client Platform Screens

To name a control, invoke the setName() method on the binding object of that particular control. You must always set a unique name for each control on the screen so that it is easy to refer this control in other files.

To name a control, you must create a binding object.

Creating a Binding Object

About this task

To create a binding object for naming a control, create a new instance of binding class for a specific control. For example, to name a text box, create the following:

```
YRCTextBindingData oData = new YRCTextBindingData();
```

where YRCTextBindingData is the class to set bindings for the text box and oData is the binding object.

Naming a Control

About this task

Use the binding object that you created to name the control.

To name a control:

Procedure

1. Set the name of the control as follows:

   ```
   oData.setName("txtOrderNo");
   ```

   where *txtOrderNo* is the name of the text box.
2. Set the binding data for the control by associating the binding object to the key for that control. For example:

   ```
   txtOrderNo.setData(YRCConstants.YRC_TEXT_BINDING_DEFINATION,oData);
   ```

   where txtOrderNo is the reference variable name of the text box, which you specified in the visual editor and YRCConstants.YRC_TEXT_BINDING_DEFINATION is the key used for identifying the text box binding object.

   Note: IBM recommends that you do not use the same binding object for multiple controls.

   If the binding object for a control such as composite or group does not exist, or if you want to name a control without creating the binding object, you can directly set the name for that control using the setData() method. For example,

   ```
   grpSearchCriteria.setData(YRCConstants.YRC_CONTROL_NAME,
   "grpSearchCriteria");
   ```

   where grpSearchCriteria is the reference variable name of the group, which you specified in the visual editor.

Setting Data On Controls for Rich Client Platform Screens

About this task

You can set additional data on out of the box or custom controls for performing additional operations. This provides you the ability to read the controls on a screen

and determine whether or not the additional data has been set for a given control through extensibilty. You can perform custom operations on the control based on the data set for that control.

You can set the data for a control using the setControlData(String fieldName,String key, Object value) method of the YRCExtensionBehavior class.

where *fieldName* is the name of the control on which data needs to be set, *key* is the key for the data, and *value* is the value set for the key.

Binding Controls and Classes for Rich Client Platform Screens

Bindings are defined to map an input XML model to the screen and back from the screen to an target XML model.

Binding Classes

The Rich Client Platform allows you to create binding objects of the following class types for different controls:

- YRCLabelBindingData class for binding labels.
- YRCTextBindingData class for binding text boxes.
- YRCStyledTextBindingData class for binding styledtext components.
- YRCComboBindingData class for binding combo boxes.
- YRCListBindingData class for binding list boxes.
- YRCButtonBindingData class for binding checkboxes and radio buttons.
- YRCLinkBindingData class for binding links.
- YRCTableBindingData class for binding tables.
- YRCTblClmBindingData class for binding table columns.
- YRCCstmCtrlBindingData class for binding custom controls.

Types of Bindings Required for Controls on Rich Client Platform Screens

Each control is associated with a set of bindings. The Rich Client Platform supports the following types of bindings for the control types:

- Label
- Check Box
- Radio Button
- Text Box, StyledText component, and Link
- Combo Box and List Box
- Table
- Table Column—For Table columns, set the Attribute Binding.
- Custom Control

Source Binding for Controls on Rich Client Platform Screens

Source binding displays XML data in the screen returned by an API by mapping the XML attributes to the screen components. Use the source binding to specify the XML path of an attribute whose value you want to get from an XML model and display in a control. For example, consider the following XML model:

```
<OrderList>
     <Order OrderNo="Y00102495" Status="Accepted">
</OrderList>
```

If you want to get the value of the OrderNo attribute from the XML model and display in a text box, set the source binding for the text box as:

```
txtBindingData.setSourceBinding("OrderDetails:OrderList/Order/@OrderNo")
```

where txtBindingData is the text box binding object and OrderDetails is the namespace of the XML model.

When you set the source binding for a table, specify only the repeating element of the XML model. For example, consider the following XML model:

```
<OrderList>
    <Order OrderNo="Y00102495" Status="Accepted">
    <Order OrderNo="Y00992495" Status="Scheduled">
    <Order OrderNo="Y00990195" Status="Shipped">
</OrderList>
```

Now, set the source binding for the table as:

```
tblBindingData.setSourceBinding("Results:Orderlist/Order")
```

where tblBindingData is the table binding object, Results is the namespace of the XML model, and Order is the repeating element in the XML model.

Multiple Source Bindings

About this task

The Rich Client Platform supports multiple source bindings that allow you to display the values of multiple attributes in the same control. You can separate multiple source bindings by a semicolon. Using the key binding, you can change the format of the multiple source-binding values.

The Rich Client Platform allows you to set multiple source bindings for a control. However, the XML model should have the same namespace for multiple binding attributes.

For example, consider the XML model as specified in the *Source Binding for Controls on Rich Client Platform Screens* topic. To display the values of both OrderNo and OrderDate attributes of the XML model in a text box, do the following:

Procedure

1. Set the multiple source bindings for the text box as:

   ```
   txtBindingData.setSourceBinding("OrderDetails:OrderList/Order/@OrderNo;OrderDetails:OrderList/Ord
   ```

   where txtBindingData is the text box binding object and OrderDetails is the namespace of the XML model.
2. Set the key binding for the text box as:

   ```
   txtBindingData.setKey("orderno_and_status_description")
   ```

   where orderno_and_status_description is the key.
3. In the *Plug-in id*_bundle.properties file, enter the key value pair (*key* = *value*) bundle entry for the multiple source binding as:

   ```
   orderno_and_status_description = The Order No. {0} was booked on {1}
   ```

   where orderno_and_status_description is the key. {0} and {1} are the positions of the binding attributes in the XML path.

The value in the text box displays as: The Order No. Y00102495 was booked on 2005-04-07.

Target Binding for Controls on Rich Client Platform Screens

Target binding allows you to create an input XML for an API that contains data entered on the screen. You can use the target binding to specify the XML path of an attribute whose value you want to get from a control and set in an XML model. For example, consider the following XML model:

```
<OrderList>
     <Order OrderNo="Y00102495" Status="Accepted">
</OrderList>
```

If you want to set the value entered in the text box for the OrderNo attribute in the XML model, set the target binding for the text box as:

```
txtBindingData.setTargetBinding("OrderDetails:OrderList/Order/@OrderNo")
```

where txtBindingData is the text box binding object and OrderDetails is the namespace of the XML model.

When you set the target binding for a table, specify only the repeating element of the XML model. For example, consider the following XML model:

```
<OrderList>
    <Order OrderNo="Y00102495" Status="Accepted">
    <Order OrderNo="Y00992495" Status="Scheduled">
     <Order OrderNo="Y00990195" Status="Shipped">
</OrderList>
```

Now, set the target binding for the table as:

```
tblBindingData.setTargetBinding("Results:Orderlist/Order")
```

where tblBindingData is the table binding object, Results is the namespace of the XML model, and Order is the repeating element in the XML model.

Multiple Target Bindings

The Rich Client Platform supports multiple target bindings, allowing you to set the value of the attributes at multiple locations in the XML models. This is useful when you want to pass the value of a single control on the screen as an input to multiple APIs. You can also specify multiple target bindings for a control by separating them by a semicolon. For example, consider the following XML models:

OrderListDetails is the namespace of the following model.

```
<OrderList OrderNo="Y00102495" OrderDate="2005-04-07">
   <Order OrderNo="Y00102495" Status="Accepted">
</OrderList>
```

OrderLineDetails is the namespace of the following model.

```
<OrderLine>
   <OrderLineList>
      <Order OrderNo="Y00102495" ItemID="MOUSE"/>
   </OrderLineList>
</OrderLine>
```

If you want to set the value entered in the text box for the OrderNo attribute in the XML models, set the target binding for the text box as:

```
txtBindingData.setTargetBinding("OrderListDetails:OrderList/
@OrderNo;OrderListDetails:OrderList/Order/@OrderNo;OrderLineDetails:
OrderLine/OrderLineList/Order/@OrderNo)
```

where txtBindingData is the text box binding object. OrderListDetails and OrderLineDetails are the namespaces of the XML models.

Checked Binding for Controls on Rich Client Platform Screens

Checked binding is used only for checkboxes and radio buttons. Checked binding is used to specify the value based on which radio button gets selected or check box gets checked or unchecked. Use the checked binding to specify a string to get and set the value of an attribute in an XML model.

When getting the attribute value, the system compares the string value with the attribute value in the XML model. If the value matches, the check box of the corresponding attribute is automatically checked.

When you check a box, the system sets the string value specified in the Checked binding as the attribute value in the XML model.

For example, consider the following XML model:

```
<OrderList>
     <Order OrderNo="Y001" Status="Accepted"
       IsAccrossEnterprise="Y" FromHistory="N"/>
</OrderList>
```

For example, to get and set the value of the IsAccrossEnterprise attribute value as Y, set the checked binding as follows:

```
btnBindingData.setCheckedBinding("Y")
```

where btnBindingData is the button binding object.

Unchecked Binding for Controls on Rich Client Platform Screens

The unchecked binding is used to specify a string to get and set the value of an attribute in an XML model.

When comparing the value of an attribute, the value of the specified string is compared with the attribute value. If the value matches, the check box of the corresponding attribute gets automatically unchecked.

When setting the value of an attribute, the system sets the string value as the attribute value in the XML model when you uncheck the box.

For example, consider the XML model as specified in the *Checked Binding for Controls on Rich Client Platform Screens* topic. If you want to get and set the value of the IsAccrossEnterprise attribute as N, set the unchecked binding as:

```
btnBindingData.setUnCheckedBinding("N")
```

where btnBindingData is the button binding object.

List Binding for Controls on Rich Client Platform Screens

Use the list binding to specify the XML path of the repeating element to populate the list box or combo box with a list of attribute values.

For example, consider the following XML model:

```
<Order>
   <OrderStatusList>
     <OrderStatus Status="1001" StatusDesc="Created"/>
     <OrderStatus Status="1002" StatusDesc="Packed"/>
     <OrderStatus Status="1003" StatusDesc="Released"/>
     <OrderStatus Status="1004" StatusDesc="Shipped"/>
   </OrderStatusList>
</Order>
```

If you want to populate the list box or combo box with the `StatusDesc` attribute values, set the list binding as:

```
cmbBindingData.setListBinding
    ("OrderStatusDetails:Order/OrderStatusList/OrderStatus")
```

where cmbBindingData is the combo binding object, OrderStatusDetails is the namespace of the XML model, and OrderStatus is the repeating element in the XML model.

Note: You must not specify an XML attribute in the list binding.

Code Binding for Controls on Rich Client Platform Screens

Use the code binding to specify the XML path of an attribute. The value assigned to the code binding attribute is based on the value selected from the list box or combo box.

For example, consider the XML model as specified in the *List Binding for Controls on Rich Client Platform Screens* topic. To get the value of the Status attribute based on the value selected for the StatusDesc attribute, set the code binding as:

```
cmbBindingData.setCodeBinding("Status")
```

where Status is the attribute whose value is picked from the XML model based on the value of the StatusDesc attribute, which is specified in the *Description Binding for Controls on Rich Client Platform Screens* topic.

Description Binding for Controls on Rich Client Platform Screens

Use the description binding for displaying the attribute's value on the screen. To display the attribute value, specify the attribute corresponding to the repeating element that you specified in the list binding.

For example, consider the XML model as specified in the *List Binding for Controls on Rich Client Platform Screens* topic. If you want to display the value of StatusDesc attribute, then set the Description Binding as:

```
cmbBindingData.setDescriptionBinding("StatusDesc")
```

where cmbBindingData is the combo binding object and StatusDesc is the attribute corresponding to the repeating element.

Attribute Binding for Controls on Rich Client Platform Screens

Use the attribute binding to specify the XML path of the attribute whose value you want to display in a table column. For example, consider the following XML model:

```
<OrderList>
    <Order ItemID="MOUSE" ItemDesc="Pointing device"/>
    <Order ItemID="KEYBOARD" ItemDesc="Keyboard Device"/>
    <Order ItemID="PENCIL" ItemDesc="7HB Bold Pencil"/>
    <Order ItemID="PEN" ItemDesc="Super Pen"/>
</OrderList>
```

To display the value of `ItemID` attribute in the table column:

- Set the source binding for the table as:

  `ItemDetails:OrderList/Order`

  where Order is the repeating element in the XML model.
- Specify the attribute binding as:

  `ItemID`

  where ItemID is the attribute corresponding to the repeating element as specified in source binding.

Multiple Attribute Bindings

The Rich Client Platform supports multiple attribute bindings, allowing you to display the values of multiple attributes in a table column. You can separate multiple attribute bindings by a semicolon. Using the key binding, you can change the format of the multiple attribute-binding values.

The Rich Client Platform allows you to set multiple attribute bindings for a control. But the XML model must have the same namespace for multiple binding attributes.

For example, to display the values of ItemID and ItemDesc attributes in a table column:

- Set the attribute binding for the table column as:

  `ItemID;ItemDesc`
- Set the key binding for the table column as:

  `item_description`

  where *item_description* is the key.
- In the bundle file, enter the *key* = *value* pair bundle entry for the previously specified source binding as:

  `item_description = {0} : {1}`

  where *item_description* is the key. {0} and {1} is the position of the binding attributes in the XML path as specified in the source binding.

  As a result, the table column displays the value as: MOUSE : Pointing Device.

Key Binding for Controls on Rich Client Platform Screens

Use the key binding to specify a resource bundle key, which you want to use to format and display the XML data within a localizable sentence or combined with another XML data attribute. The key binding is used in conjunction with the source binding or attribute binding as described in the previous sections. For example, consider the following XML model:

```
<OrderList>
   <Order ItemID="MOUSE" ItemDesc="Pointing device"/>
</OrderList>
```

To format the value of the ItemID attribute:
- Specify the source binding as:
  `ItemDetails:OrderList/Order/@ItemID`
- Specify the key binding as: `item_description`

The bundle file contains the following <key>=<value> pair bundle entry for the previously specified key:

```
item_description= The item ordered is : {0}
```

where item_description is the key. {0} is the position of the binding attributes in the XML path.

The value displayed is: The item ordered is MOUSE.

Binding Input to Custom Controls on Rich Client Platform Screens

About this task

Custom Control Input binding allows you to configure following parameters for a custom control:
- BorderRequiredOnInputControls—Whether you want to have borderaround the custom control or not.
- Editable—Whether you want to make the custom control editable or not.
- NoOfColumns—Number of columns you want to have in the custom control.
- Style(int)—Style you want to have for the custom control. For example, SWT.LEFT, SWT.WRAP, and so forth.

For example, you can set the Custom Control Input Binding for a custom control as follows:

Procedure

1. Set the border for the custom control as:
   `cstmCtrlInputBindingData.setBorderRequiredOnInputControls(true);`
   where cstmCtrlInputBindingData is the custom control input binding object.
2. To make the custom control editable, set the editable parameter as:
   `cstmCtrlInputBindingData.setEditable(false);`
3. To define multiple columns for a custom control, set the NoOfColumns parameter as:
   `cstmCtrlInputBindingData.NoOfColumns (3);`
   where (3) is the number of columns you want to have in the custom control.
4. Set the custom control style as:
   `cstmCtrlInputBindingData.setStyle(SWT.LEFT );`

About Setting Bindings for Controls on Rich Client Platform Screens

Consider the following input and target XML models for specifying the different bindings for controls:

Input XML Model

```
<Order OrderNo="Y001" Status="Included In Shipment"
       IsAccrossEnterprise="Y" FromHistory="N" Link
       Binding="A Link Binding Example : Click Me">
   <OrderLineList>
      <OrderLine ItemID="MOUSE" CodeDescription="First
                 Class" Code="A"/>
      <OrderLine ItemID="PEN" CodeDescription="Second
                 Class" Code="B"/>
      <OrderLine ItemID="PENCIL" CodeDescription="First
                 Class" Code="A"/>
   </OrderLineList>
</Order>
```

Target XML Model

```
<OrderList>
  <Order OrderNo="Y001" Status="Accepted"
         CodeDescription="First Class" Code="A"
         IsAccrossEnterprise="Y" FromHistory="N"/>
  <Order OrderNo="Y002" Status="Released"
         CodeDescription="Second Class" Code="B"
         IsAccrossEnterprise="N" FromHistory="Y"/>
  <Order OrderNo="Y003" Status="Shipped"
         CodeDescription="Third Class" Code="C"
         IsAccrossEnterprise="Y" FromHistory="Y"/>
</OrderList>
<OrderStatus>
    <Order OrderNo="Y001" Status="Accepted"/>
    <Order OrderNo="Y002" Status="Released"/>
    <Order OrderNo="Y003" Status="Shipped"/>
</OrderStatus>
```

Creating a Binding Object for a Label

About this task

To set bindings for a label, create a binding object for the label.

To create a binding object for a label, create a new instance of the YRCLabelBindingData binding class. For example:

```
YRCLabelBindingData lblBindingData = new YRCLabelBindingData();
```

where YRCLabelBindingData is the class to set bindings for the label and lblBindingData is the binding object.

Bind a Label

Procedure

1. Set the name of the label using the binding object that you created. For example:

   ```
   lblBindingData.setName("lblOrderNo");
   ```

   where lblOrderNo is the name of the text box and lblBindingData is the binding object.
2. Set the source binding for the label. For example:

   ```
   lblBindingData.setSourceBinding("OrderDetails:Order/@OrderNo");
   ```

   where OrderDetails is the namespace of the model.
3. (Optional) Set the multiple source binding for the label. For example:

   ```
   lblBindingData.setSourceBinding("OrderDetails:Order/@OrderNo;Order/@Status");
   ```

where OrderDetails is the namespace of the model.

4. (Optional) Set the key binding for the label. For example:

   ```
   lblBindingData.setKey("order_details");
   ```

   where order_details is the key.

   Note: If you are specifying multiple source binding for the label, this step is mandatory.

5. (Optional) If you want to display an image for this label, set the server image configuration for the label to display the image from the server. For example:

   ```
   lblBindingData.setServerImageConfiguration(YRCConstants.IMAGE_SMALL);
   ```

   where IMAGE_SMALL is the value of the Name attribute of the Config element, which is defined in the configuration file. For more information about configuring server images, see theSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales *Installation GuideInstalling the Platform*.

6. Set the binding data for the label by associating the binding object to the key. For example:

   ```
   lblOrderNo.setData(YRCConstants.YRC_LABEL_BINDING_DEFINITION,
   lblBindingData);
   ```

   where lblOrderNo is the reference variable name of the label that you specified in the visual editor and YRCConstants.YRC_LABEL_BINDING_DEFINITION is the key used for identifying the label binding object.

Creating a Binding Object for Text Boxes

About this task

To set bindings for a text box, you must create a binding object for the text box.

To create a binding object for a text box, create a new instance of the YRCTextBindingData binding class. For example:

```
YRCTextBindingData txtBindingData = new YRCTextBindingData();
```

where YRCTextBindingData is the class to set bindings for the text box and txtBindingData is the binding object.

Bind a Text Box

Procedure

1. Set the name of the text box by using the binding object that you created. For example:

   ```
   txtBindingData.setName("txtOrderNo");
   ```

   where txtOrderNo is the name of the text box and txtBindingData is the binding object.

2. Set the source binding for the text box. For example:

   ```
   txtBindingData.setSourceBinding("OrderDetails:Order/@OrderNo");
   ```

   where OrderDetails is the namespace of the model.

3. (Optional) Set the multiple source binding for the text box. For example:

   ```
   txtBindingData.setSourceBinding("OrderDetails:Order/@OrderNo;Order/@Status");
   ```

   where OrderDetails is the namespace of the model.

4. (Optional) Set the key binding for the text box. For example:

   ```
   txtBindingData.setKey("order_details");
   ```

where order_details is the key.

Note: If you are specifying multiple source binding for the text box, this step is mandatory.

5. Set the target binding for the text box. For example:
```
txtBindingData.setTargetBinding("OrderListDetails:OrderList/Order/@OrderNo");
```
where OrderListDetails is the namespace of the model.
6. (Optional) Set the multiple target binding for the text box. For example:
```
txtBindingData.setTargetBinding("OrderDetails:Order/@OrderNo;OrderStatus/Order/@Status");
```
where OrderDetails is the namespace of the model.
7. Set the binding data for the text box by associating the binding object to the key. For example:
```
txtOrderNo.setData(YRCConstants.YRC_TEXT_BINDING_DEFINATION,
txtBindingData);
```
where txtOrderNo is the reference variable name of the text box, which you specified in the visual editor and YRCConstants.YRC_TEXT_BINDING_DEFINATION is the key used for identifying the text box binding object.

Creating a Binding Object for StyledText Components

About this task

To set bindings for a styledtext component, create a binding object for the styledtext component.

To create a binding object for a styledtext component, create a new instance of the YRCStyledTextBindingData binding class. For example:

```
YRCStyledTextBindingData styledTextBindingData = new
YRCStyledTextBindingData();
```

where YRCStyledTextBindingData is the class to set bindings for the text box and styledTextBindingData is the binding object.

Bind a StyledText Component

Procedure

1. Set the name of the styledtext component using the binding object that you created. For example:
```
styledTextBindingData.setName("styledTextOrderNo");
```
where styledTextOrderNo is the name of the text box and styledTextBindingData is the binding object.
2. Set the source binding for the styledtext component. For example:
```
styledTextBindingData.setSourceBinding("OrderDetails:Order/@OrderNo");
```
where OrderDetails is the namespace of the model.

For more information about source binding, see "Source Binding for Controls on Rich Client Platform Screens".
3. (Optional) Set the multiple source binding for the styledtext component. For example:
```
styledTextBindingData.setSourceBinding("OrderDetails:Order/
@OrderNo;Order/@Status");
```
where OrderDetails is the namespace of the model.

For more information about multiple source binding, see "Multiple Source Bindings".

4. (Optional) Set the key binding for the styledtext component. For example:

   ```
   txtBindingData.setKey("order_details");
   ```

   where order_details is the key.

   Note: If you are specifying multiple source binding for the styledtext component, this step is mandatory.

   For more information about key binding, see "Key Binding for Controls on Rich Client Platform Screens".

5. Set the target binding for the styledtext component. For example:

   ```
   styledTextBindingData.setTargetBinding("OrderListDetails:OrderList/
   Order/@OrderNo");
   ```

   where OrderListDetails is the namespace of the model.

   For more information about target binding, see "Target Binding for Controls on Rich Client Platform Screens".

6. (Optional) Set the multiple target binding for the styledtext component. For example:

   ```
   styledTextBindingData.setTargetBinding("OrderDetails:Order/
   @OrderNo;OrderStatus/Order/@Status");
   ```

   where OrderDetails is the namespace of the model.

   For more information about multiple target binding, see "Multiple Target Bindings".

7. Set the binding data for the styledtext component by associating the binding object to the key. For example:

   ```
   styledTextOrderNo.setData(YRCConstants.YRC_STYLED_TEXT_BINDING_DEFINATION,
   styledTextBindingData);
   ```

   where styledTextOrderNo is the reference variable name of the styledText component, which you specified in the visual editor and YRCConstants.YRC_STYLED_TEXT_BINDING_DEFINATION is the key used for identifying the styledtext component binding object.

Creating a Binding Object for Combo Boxes

About this task

To set bindings for a combo box, create a binding object for the combo box.

To create a binding object for a combo box, create a new instance of the YRCComboBindingData binding class. For example:

```
YRCComboBindingData cmbBindingData = new YRCComboBindingData();
```

where YRCComboBindingData is the class to set bindings for the combo box and cmbBindingData is a binding object.

Bind a Combo Box

Procedure

1. Set the name of the combo box using the binding object that you created. For example:

   ```
   cmbBindingData.setName("cmbCode");
   ```

   where cmbCode is the name of the combo box and cmbBindingData is the binding object.

2. Set the source binding for the combo box. For example:

```
cmbBindingData.setSourceBinding("OrderDetails:Order/OrderLineList/OrderLine/@Code");
```

   where OrderDetails is the namespace of the model.

   Note: For combo box, source binding is used to specify the default value that should get selected in the combo box. The value of the source binding attribute is compared with the code binding attribute and the corresponding value of the description binding attribute gets selected in the combo box.
3. Set the list binding for the combo box. For example:

```
cmbBindingData.setListBinding("OrderListDetails:OrderList/Order");
```

   where OrderListDetails is the namespace of the model.
4. Set the description binding for the combo box. For example:

```
cmbBindingData.setDescriptionBinding("CodeDescription");
```

5. Set the code binding for the combo box. For example:

```
cmbBindingData.setCodeBinding("Code");
```

6. Set the target binding for the combo box. For example:

```
cmbBindingData.setTargetBinding("OrderListDetails:OrderList/Order/@Code");
```

   where OrderListDetails is the namespace of the model.

   Note: For combo box, target binding is used to specify the attribute whose value is set in the target XML model when user selects a value from the combo box. The value of the code binding attribute is set as the value of the target binding attribute in the target XML model.
7. Set the binding data for the combo box by associating the binding object with the key. For example:

```
cmbCommonCode.setData(YRCConstants.YRC_COMBO_BINDING_
      DEFINATION,cmbBindingData);
```

   where cmbCommonCode is the reference variable name of the combo box, which you specified in the visual editor and YRCConstants.YRC_COMBO_BINDING_DEFINATION is the key used for identifying the combo box binding object.

Version-Specific Data in Combo Boxes

The Rich Client Platform supports multiple versions of Rich Client Platform clients on a single server. In such a scenario, data populated in combo boxes such as common codes vary between versions and do not correspond to the version of the client launched. To overcome this problem, combo binding is enabled for version awareness.

The getCommonCodeList API and YRCComboBindingData are enhanced to include additional parameters. The method comboBindingData.setApplicationVersionSpecific(true) is called for displaying version-specific data in select combo boxes for the required application.

Populating Version-Specific Data in Combo Boxes

About this task

To populate the combo boxes with version-specific data:

Procedure

1. The method setApplicationVersionSpecific is added to the YRCComboBindingData to specify version-specific information in a combo box, in the following format:

```
public void setApplicationVersionSpecific(boolean versionSpecific)
```

2. Set versionSpecific to "true". If this is set to "true", the combo boxes are populated with version-specific information such as common codes. Only data pertaining to the version of the client launched is populated in the combo box.
3. Combo boxes which are populated with version-specific data bear a different theme, versionedComboTheme. Applications can override this theme, if required. The Control Info Panel is updated to include the version specific information as well as the theme applied for a combo box as follows:
   - Is Application Version Specific: true
   - Theme Name: VersionedComboTheme
4. If data selected in the combo box is not compatible with the version of the client launched, a default key DifferentVersionPreffix = **{0}** is added to the Sterling Application Platform Bundle, where {0} represents incompatible data. For example, if status information "Chained Order Created" in a combo box is selected, but does not correspond to the version of the client launched, it is displayed in the following format:

   Chained Order created.

   The application can override this key.

Creating a Binding Object for List Boxes

About this task

To set bindings for a list box, create a binding object for the list box.

To create a binding object for a list box, create a new instance of the YRCListBindingData binding class. For example:

```
YRCListBindingData lstBindingData = new YRCListBindingData();
```

where YRCListBindingData is the class to set bindings for the list box and lstBindingData is a binding object.

Bind a List Box

Procedure

1. Set the name of the list box using the binding object that you created. For example:

```
lstBindingData.setName("lstCommonCode");
```

   where lstCommonCode is the name of the list box and lstBindingData is the binding object.
2. Set the source binding for the list box. For example:

```
lstBindingData.setSourceBinding("OrderDetails:Order/OrderLineList/OrderLine/@Code");
```

   where OrderDetails is the namespace of the model.

   Note: For list box, source binding is used to specify the default value that should get selected in the list box. The value of the source binding attribute is compared with the code binding attribute and the corresponding value of the description binding attribute gets selected in the list box.

3. Set the list binding for the list box. For example:

```
lstBindingData.setListBinding("OrderListDetails:OrderList/Order");
```

where OrderListDetails is the namespace of the model.

4. Set the description binding for the list box. For example:

```
lstBindingData.setDescriptionBinding("CodeDescription");
```

5. Set the code binding for the list box. For example:

```
lstBindingData.setCodeBinding("Code");
```

6. Set the target binding for the list box. For example:

```
lstBindingData.setTargetBinding("OrderListDetails:OrderList/Order/@Code");
```

where OrderListDetails is the namespace of the model.

Note: For list box, target binding is used to specify the attribute whose value is set in the target XML model when user selects a value from the list box. The value of the code binding attribute is set as the value of the target binding attribute in the target XML model.

7. Set the binding data for the list box by associating the binding object with the key. For example:

```
lstCommonCode.setData(YRCConstants.YRC_LIST_BINDING_DEFINITION,
lstBindingData);
```

where lstCommonCode is the reference variable name of the list box, which you specified in the visual editor and YRCConstants.YRC_LIST_BINDING_DEFINITION is the key used for identifying the combo box binding object.

Creating a Binding Object for Checkboxes

About this task

To set bindings for a check box, create a binding object for the check box.

To create a binding object for a check box, create a new instance of the YRCButtonBindingData binding class. For example:

```
YRCButtonBindingData chkBindingData = new YRCButtonBindingData();
```

where YRCButtonBindingData is the class to set bindings for the check box and chkBindingData is a binding object.

Bind a Check Box

Procedure

1. Set the name of the check box using the binding object that you created. For example:

```
chkBindingData.setName("chkAcrossEnterprice");
```

where chkAccrossEnterprice is the name of the check box and chkBindingData is the binding object.

2. Set the source binding for the check box. For example:

```
chkBindingData.setSourceBinding("OrderDetails:Order/@IsAccrossEnterprice");
```

where OrderDetails is the namespace of the model.

3. Set the target binding for the check box. For example:

```
chkBindingData.setTargetBinding("OrderDetails:Order/@IsAccrossEnterprice");
```

where OrderDetails is the namespace of the model.

4. Set the checked binding for the check box. For example:

```
chkBindingData.setCheckedBinding("Y");
```

When getting the IsAcrossEnterprise field value from the input XML model, the string "Y" is compared with the IsAcrossEnterprise field value in the input XML model. If the value matches, the check box is automatically checked. When setting the field value in the target XML model, the string "Y" is set as the value for IsAcrossEnterprise field when you check the box.

5. Set the unchecked binding for the check box. For example:

   ```
   chkBindingData.setUnCheckedBinding("N");
   ```

   When getting the IsAcrossEnterprise field value from the input XML model, the string "N" is compared with the IsAcrossEnterprise field value in the input XML model. If the value matches, the check box is automatically unchecked. When setting the field value in the target XML model, the string "N" is set as the value for IsAcrossEnterprise field when you uncheck the box.

6. Set the binding data for the check box by associating the binding object to the key. For example:

   ```
   chkAcrossEnterprice.setData(YRCConstants.YRC_BUTTON_BINDING_DEFINATION,chkBindingData);
   ```

   where chkAcrossEnterprice is the reference variable name of the check box, which you specified in the visual editor and YRCConstants.YRC_BUTTON_BINDING_DEFINATION is the key used for identifying the check box binding object.

Creating a Binding Object for Radio Buttons

About this task

To set bindings for a radio button, create a binding object for the radio button.

To create a binding object for a radio button, create a new instance of the YRCButtonBindingData binding class. For example:

```
YRCButtonBindingData rdBindingData = new YRCButtonBindingData();
```

where YRCButtonBindingData is the class to set bindings for the radio button and rdBindingData is a binding object.

Bind a Radio Button

Procedure

1. Set the name of the radio button using the binding object that you created. For example:

   ```
   rdBindingData.setName("rdOpen");
   ```

   where rdOpen is the name of the radio button and rdBindingData is the binding object.

2. Set the source binding for the radio button. For example:

   ```
   rdBindingData.setSourceBinding("OrderDetails:Order/@FromHistory");
   ```

   where OrderDetails is the namespace of the model.

3. Set the target binding for the radio button. For example:

   ```
   rdBindingData.setTargetBinding("OrderDetails:Order/@FromHistory");
   ```

   where OrderDetails is the namespace of the model.

4. Set the checked binding for the radio button to specify the value used to get the FromHistory field value from the input XML model. Set the specified value for the FromHistory field in the target XML model. For example:

   ```
   rdBindingData.setCheckedBinding("S001");
   ```

where S001 is the value of the rdOpen radio button.

For example, if there are three radio buttons, create binding for each of the radio buttons. Set name, source binding and target binding for each radio button. Set the checked binding for each radio button with different values such as S001, S002, and S003. Therefore, when getting the value for a particular field from the input XML model, the value "S001" is compared with the value of that field in the input XML model. If the value matches, then the radio button corresponding to that field is automatically selected. When setting the value in the target XML model, the value "S001" is set as the value for that field in the target XML model when you select the radio button corresponding to that field.

5. Set the binding data for the radio button by associating the binding object to the key. For example:

```
rdOpen.setData(YRCConstants.YRC_BUTTON_BINDING_DEFINATION,rdBindingData);
```

where rdOpen is the reference variable name of the radio button, which you specified in the visual editor and YRCConstants.YRC_BUTTON_BINDING_DEFINATION is the key used for identifying the check box binding object.

Creating a Binding Object for Links

About this task

To set bindings for a link, you must create a binding object for the link.

To create a binding object for a link, create a new instance of the YRCLinkBindingData binding class. For example:

```
YRCLinkBindingData linkBindingData = new YRCLinkBindingData();
```

where YRCLinkBindingData is the class to set bindings for the link and linkBindingData is the binding object.

Bind a Link

Procedure

1. Set the name of the link using the binding object that you created. For example:

```
linkBindingData.setName("lnkClickHere");
```

where lnkClickHere is the name of the link and linkBindingData is the binding object.

2. Set the source binding for the link. For example:

```
linkBindingData.setSourceBinding("OrderDetails:Order/@Binding");
```

where OrderDetails is the namespace of the model.

3. Set the binding data for the link by associating the binding object to the key. For example:

```
lnkClickHere.setData(YRCConstants.YRC_LINK_BINDING_DEFINATION,linkBindingData);
```

where lnkClickHere is the reference variable name of the link, which you specified in the visual editor and YRCConstants.YRC_LINK_BINDING_DEFINATION is the key used for identifying the link binding object.

Creating a Binding Object for a Standard Table

About this task

To set bindings for a standard table, you must create a binding object for the standard table. Also, you must create a binding object for a column.

To create a binding object for a standard table, create a new instance of YRCTableBindingData binding class. For example:

```
YRCTableBindingData tblBindingData =
new YRCTableBindingData();
```

where YRCTableBindingData is the class to set bindings for the standard table and tblBindingData is a binding object.

Creating a Binding Object for a Column

About this task

To create a binding object for a column, create an array of YRCTblClmBindingData[] with an array size equal to the number of columns in the table. For example:

```
YRCTblClmBindingData clmBindingData[] =
new YRCTblClmBindingData[no. of columns in the table];
```

Bind a Standard Table and Column

Procedure

1. Set the name of the table using the table binding object that you created. For example:

   ```
   tblbBindingData.setName("tblSearchResults");
   ```

   where tblSearchResults is the name of the table and tblbBindingData is the binding object.
2. Set the name of the table column using the table column binding object that you created. For example:

   ```
   clmBindingData[0].setName("clmItemID");
   ```

   where tblSearchResults is the name of the table column and clmBindingData is the binding object.
3. Associate the YRCTblClmBindingData() attribute to each column. For example:

   ```
   clmBindingData[0] = new YRCTblClmBindingData();
   ```
4. Set the attribute binding for the column. For example:

   ```
   clmBindingData[0].setAttributeBinding("ItemID");
   ```
5. (Optional) Set the multiple attribute binding for the column. For example:

   ```
   clmBindingData[0].setAttributeBinding("ItemID;Code");
   ```

   where OrderDetails is the namespace of the model.
6. (Optional) Set the key binding for the column. For example:

   ```
   clmBindingData[0].setKey("item_details");
   ```

   where item_details is the key.

   Note: If you are specifying multiple attribute binding for the column, this step is mandatory.
7. Set the title of the table column. For example:

```
clmBindingData[0].setColumnBinding("item_id");
```

8. Set the server image configuration for the column to display the image from the server. For example,

   ```
   clmBindingData[0].setServerImageConfiguration(YRCConstants.IMAGE_SMALL);
   ```

   where IMAGE_SMALL is the value of the Name attribute of the Config element, which is defined in the configuration file. For more information about configuring server images, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Installation GuideInstalling the Platform.*
9. To sort a column, set the SortReqd attribute value to "true". For example:

   ```
   clmBindingData[0].setSortReqd(true);
   ```

10. To make a column data localized, set the DbLocaliseReqd attribute value to "true". For example:

    ```
    clmBindingData[0].setDbLocaliseReqd(true);
    ```

    For more information about database localization, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Localization Guide.*
11. Repeat steps 2 to 10 to set bindings for all columns in the table.
12. To allow navigation through the keys in a table, set the KeyNavigationRequired attribute value to "true". For example:

    ```
    tblBindingData.setKeyNavigationRequired(true);
    ```

13. To sort a table, set the SortReqd attribute value to "true". For example:

    ```
    tblBindingData.setSortRequired(true);
    ```

14. To filter the table based on some value, set the FilterReqd attribute value to "true". For example:

    ```
    tblBindingData.setFilterRequired(true);
    ```

15. Set the source binding for the column. For example:

    ```
    tblBindingData.setSourceBinding("Results:/OrderLineList/OrderLine");
    ```

    where Results is the namespace for this model.
16. Set the column binding data on the table binding data using the setTblClmBindings() method. For example:

    ```
    tblBindingData.setTblClmBindings(clmBindingData);
    ```

17. (Optional) Use the setLinkProvider() method to create links in the table. The setLinkProvider() method takes the IYRCTableLinkProvider interface as input, which contains two methods getLinkTheme() and linkSelected(). You must implement these methods to create links in the table. The linkSelected() method is called when you select any link in the column. For example:

    ```
    tblBindingData.setLinkProvider(new IYRCTableLinkProvider() {
    public String getLinkTheme(Object element, int columnIndex) {
    return "TableLink"; }
    public void linkSelected(Object element, int columnIndex) {
    }});
    ```

    In the getLinkTheme() method, add the logic to set themes for links in a column. This method returns the name of the link theme. If it returns null it is assumed that a link is not required.

    In the linkSelected() method, add the logic to perform the required operation, when the link on the table column cell gets selected.

    Note: To create links in your table, set the LinkRequired flag of the table column binding object to "true". For example:

    ```
    clmBindingData[0].setLinkReqd(true);
    ```

where clmBindingData is the object of YRCTblClmBindingData class and 0 is the column index.

18. (Optional) Use the setImageProvider() method to add images for a table column. The setImageProvider() method takes the IYRCTableImageProvider interface as input, which contains getImageThemeForColumn() method. You must implement this method to add images in columns. For example:

```
tblBindingData.setImageProvider(new IYRCTableImageProvider(){
public String getImageThemeForColumn(Object element, int columnIndex){
return null;
}});
```

In the getImageThemeForColumn() method, add the logic for setting a unique image theme for the table column cell based on some condition. This method returns the unique image theme set. If it returns null, the default image theme is applied.

19. (Optional) Use the setColorProvider() method to set different colors for the table columns. The setColorProvider() method takes the IYRCTableColorProvider interface as input, which contains getColorTheme() method. You must implement this method to provide different colors for the table columns. For example:

```
tblBindingData.setColorProvider(new IYRCTableColorProvider(){
public String getColorTheme(Object element, int columnIndex) {
return null;
}});
```

In the getColorTheme() method, add the logic for setting different colors for the table column cells based on some condition. For example, you may want to set different color for non-editable cells that displays data for the status field, and different color for editable cells that displays data for the amount field. This method returns the name of the color theme. If it returns null, the default color theme is applied.

20. (Optional) Use the setFontProvider() method to set different font types for the table columns. The setFontProvider() method takes the IYRCTableFontProvider interface as input, which contains getFontTheme() method. You must implement this method to provide different colors for the table columns. For example:

```
tblBindingData.setFontProvider(new IYRCTableFontProvider(){
public String getFontTheme(Object element, int columnIndex) {
return null;
}});
```

In the getFontTheme() method, add the logic for setting different font types for the table column cells based on some condition. For example, you may want to set different font type for non-editable cells that displays data for the status field and different font type for editable cells that displays data for the amount field. This method returns the name of the font theme. If it returns null, the default font theme is applied.

21. After setting the binding properties for the YRCTableBindingData object, set the binding data for the table by associating the binding object to the key. For example:

```
tblSearchResult.setData(YRCConstants.YRC_TABLE_BINDING_DEFINATION,
tblBindingData);
```

where YRCConstants.YRC_BUTTON_BINDING_DEFINATION is the key used for the table binding object.

Setting Bindings for an Editable Table

Binding editable tables is same as binding standard tables except that when you bind editable tables, you must handle the editable table columns. To bind an editable table, follow the steps as described in the *Creating a Binding Object for a Standard Table* topic.

To handle the editable table columns, use the setCellModifier() method. The setCellModifier() method takes the IYRCCellModifier interface as input, which contains three methods allowModifiedValue(), allowModify() and getModifiedValue(). You must implement these methods to control editable features of different columns in the table. For example:

```
tblBindingData.setCellModifier(new IYRCCellModifier() {
protected boolean allowModify(String property, String value,
Element element)
{
return true;
}
protected int allowModifiedValue(String property, String value,
Element element)
{
return 0;
}
protected String getModifiedValue(String property, String value,
Element element)
{
return value;
}});
```

In the allowModify() method, add the logic to check whether you want to allow modifications in an editable cell of a table column. For example, you may want to allow modifications for an editable cell, which displays data for the discount field. This method returns a boolean value, "true" or "false". If the method returns a "false" value, it indicates that modifications are not allowed for that cell.

In the allowModifiedValue() method, add the logic for adding further validation constraints to check whether the new value entered is valid or not. This method returns an integer value. If it returns "0", then the existing value is not replaced with the new value.

In the getModifiedValue() method, add the logic to set the modified value for a cell of a table column that you are currently editing. You can use this method to update some other property based on the current one or to change the format of the property.

Binding Combo Box Cell Editors

About this task

Binding combo box cell editors means binding a combo box inside an editable table. To set bindings for a combo box cell editor, do the following:

Procedure

1. Create a binding object for the combo box.
2. Set the list binding, description binding, and code binding for the combo box.

   Note: Only set the list binding, description binding, and code binding for the combo box.

3. Set the binding data of the table column with the YRCComboBindingData binding object as an argument. For example:

   ```
   clmBindingData[columnIndex].setBindingData(cmbBindingData);
   ```

   where cmbCommonCode is the reference variable name of the combo box, which you specified in the visual editor and YRCConstants.YRC_COMBO_BINDING_DEFINATION is the key used for identifying the combo box binding object.

Setting Bindings for an Extended Table

To set bindings for an extended table, you must create a binding object for the extended table.

Note: Make sure that you write the code for binding extended tables in the extension behavior class that you created. In the extension behavior class, override the getExtendedTableBindingData() method. In this method create and return the extended table binding object. For example:

```
YRCExtendedTableBindingData extntblBindingData = new
YRCExtendedTableBindingData("tableSearch");
// Create and get the advanced column binding map for the extended table.
HashMap advclmBindingData = getTableColumnBindingData ("tableSearch");
extntblBindingData.setTableColumnBindingsMap ("advclmBindingData");
.
.
    //Set Bindings for Extended Table and Advanced Columns
.
return extntblBindingData;
```

where tableSearch is the name of the extended table.

Creating a Binding Object for an Extended Table

About this task

To create a binding object for an extended table, create a new instance of YRCExtendedTableBindingData binding class. For example:

```
YRCExtendedTableBindingData extntblBindingData = new
YRCExtendedTableBindingData();
```

where YRCExtendedTableBindingData is the class to set bindings for the extended table and extntblBindingData is a binding object.

Create a Map of the Advanced Column Binding Data

About this task

To create a map of the binding data for the advanced column that you added using the Rich Client Platform Extensibility Tool, create a new instance of HashMap binding class. For example:

```
HashMap bindingDataMap = new HashMap();
```

where HashMap is the class to create a map of the advanced column binding data and bindingDataMap is the hash map. The HashMap contains the name of the advanced column as the key and the corresponding binding data as the value.

Bind an Extended Table and Advanced Column

About this task

Procedure

1. Create a binding object for an advanced column by creating a new instance of the YRCTblClmBindingData binding class. For example:
```
YRCTblClmBindingData advclmBindingData = new YRCTblClmBindingData();
```
where YRCTblClmBindingData is the class to set bindings for the advanced column and advclmBindingData is a binding object.

Note: Only if you have added an advanced column through extensibility, you need to create the binding object and set bindings for that advanced column.
2. Set the attribute binding for the advanced column. For example:
```
advclmBindingData.setAttributeBinding("ItemID");
```
3. (Optional) Set multiple attribute binding for the advanced column. For example:
```
advclmBindingData.setAttributeBinding("ItemID;Code");
```
where OrderDetails is the namespace of the model.
4. (Optional) Set the key binding for the advanced column. For example:
```
advclmBindingData.setKey("item_details");
```
where item_details is the key.

Note: If you specify multiple attribute binding for the column, step is mandatory.
5. Set the server image configuration for the advanced column to display the image from the server. For example,
```
advclmBindingData.setServerImageConfiguration(YRCConstants.IMAGE_SMALL);
```
where IMAGE_SMALL is the value of the Name attribute of the Config element, which is defined in the configuration file. For more information about configuring server images, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Installation GuideInstalling the Platform*.
6. To sort the advanced column, set the SortReqd attribute value to "true". For example:
```
advclmBindingData.setSortReqd(true);
```
7. To localize the advanced column data, set the DbLocaliseReqd attribute value to "true". For example:
```
advclmBindingData.setDbLocaliseReqd(true);
```
For more information about localizing the database, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Localization Guide*.
8. To filter an advanced column based on some value, set the FilterReqd attribute value to "true". For example:

advclmBindingData.setFilterRequired(true);
9. Add the advanced column binding data object to the data map object. For example:
```
bindingDataMap.put("extn_AdvClm1", advclmBindingData);
```

where bindingDataMap is the hash map binding object, extn_AdvCml is the name of the advanced column added using the Rich Client Platform Extensibility Tool, and advclmBindingData is the advanced column binding object.

10. Repeat Step 1 through Step 9 to set bindings for all advanced columns that you add to the extended table using the Rich Client Platform Extensibility Tool.
11. (Optional) To sort an extended table, set the SortReqd attribute value to "true". For example:

```
extntblBindingData.setSortRequired(true);
```

12. (Optional) To filter an extended table based on some value, set the FilterReqd attribute value to "true". For example:

```
extntblBindingData.setFilterRequired(true);
```

13. Set the source binding for the table. For example:

```
extntblBindingData.setSourceBinding("Results:/OrderLineList/OrderLine");
```

where Results is the namespace for this model.

14. (Optional) Use the setLinkProvider() method to create links in the advanced columns that you added using the Rich Client Platform Extensibility Tool. The setLinkProvider() method takes the YRCExtendedTableLinkProvider class as input, which contains two methods getLinkTheme() and linkSelected(). You must implement these methods to create links in the advanced columns of an extended table. The linkSelected() method is called when you select any link in the column. For example:

```
YRCExtendedTableLinkProvider extntblLinkProvider
= new YRCExtendedTableLinkProvider() {
public String getLinkTheme(Object element, String property) {
return "TableLink"; }
public void linkSelected(Object element, String property) {
}};
extntblBindingData.setLinkProvider(extntblLinkProvider);
```

In the getLinkTheme() method, add the logic to set themes for links in a column. This method returns the name of the link theme. If it returns null it is assumed that a link is not required.

In the linkSelected() method, add the logic to perform the required operation, when you click the link in the advanced column cell.

Note: To create links in your extended table, set the LinkRequired flag of the advanced column binding object to "true". For example:

```
advclmBindingData.setLinkReqd(true);
```

where advclmBindingData is the advanced column binding object.

Note: On UNIX or Linux, to activate a link in an editable table, double-click on the link.

15. (Optional) Use the setImageProvider() method to add images for an advanced column. The setImageProvider() method takes the YRCExtendedTableImageProvider class as input, which contains getImageThemeForColumn() method. You must implement this method to add images in advanced columns. For example:

```
YRCExtendedTableImageProvider extntblImageProvider
= new YRCExtendedTableImageProvider() {
public String getImageThemeForColumn(Object element, String property) {
        if (property.equals("@ItemID")) {
             Element e = (Element)element;
             String strImageTheme = e.getAttribute("TableFilter");
```

```
                return strImageTheme;
            } return null;
}}; extntblBindingData.setImageProvider (extntblImageProvider);
```

In the getImageThemeForColumn() method, add the logic for setting a unique image theme for the advanced column cell based on some condition. This method returns the unique image theme set. If it returns null, the default image theme is applied.

16. (Optional) Use the setColorProvider() method to set different colors for the advanced columns. The setColorProvider() method takes the YRCExtendedTableColorProvider class as input, which contains getColorTheme() method. You must implement this method to provide different colors for the advanced columns. For example:

```
YRCExtendedTableColorProvider extntblColorProvider
= new YRCExtendedTableColorProvider() {
public String getColorTheme(Object element, String property) {
           if (property.equals("@Price")) {
                Element e = (Element)element;
                int price = YRCXmlUtils.getIntAttribute(e,"Price");
          If (price < 50) {
              return "ValidationOK";
          else
{
              return "ValidationERROR";
          }
      } return null;
}};
extntblBindingData.setColorProvider (extntblColorProvider);
```

In the getColorTheme() method, add the logic for setting different colors for the advanced column cells based on some condition. For example, you may want to apply a different color for non-editable cells that displays data for the status field, and a different color for editable cells that displays data for the amount field. This method returns the name of the color theme. If it returns null, the default color theme is applied.

Setting Bindings for Extended Editable Tables

Binding extended editable tables is same as binding extended tables. The only difference is that when you bind extended editable tables, you must handle the editable advanced columns. To bind an extended editable table, follow the steps as described in “Setting Bindings for an Extended Table” on page 97.

To handle the editable advanced columns, use the setCellModifier() method. The setCellModifier() method takes the YRCExtendedCellModifier class as input, which contains three methods allowModifiedValue(), allowModify(), and getModifiedValue(). You must implement these methods to control editable features of different columns in the table. For example:

```
YRCExtendedCellModifier extntblCellModifier =
   new YRCExtendedCellModifier() {
public boolean allowModify(String property, String value,
   Element element) {
   return true;
}
public YRCValidationResponse validateModifiedValue(String property,
 String value, Element element) {
 return new YRCValidationResponse(YRCValidationResponse.YRC_VALIDATION_OK,
 "Status message");
}
public String getModifiedValue(String property, String value,
Element element) {
```

```
return value;
}});
extntblBindingData.setCellModifier (extntblCellModifier);
```

In the allowModify() method, add the logic to check whether you want to allow modifications in an editable cell of an advanced column. For example, you may want to allow modifications for an editable cell, which displays data for the discount field. This method returns a boolean value, "true" or "false". If the method returns a "false" value, it indicates that modifications are not allowed for that cell.

In the validateModifiedValue() method, add the logic for adding further validation constraints to check whether the new value entered is valid or not. This method returns an instance of YRCValidationResponse object with an appropriate status code and status message. The status code can be one of the following:

- YRCValidationResponse.YRC_VALIDATION_OK
- YRCValidationResponse.YRC_VALIDATION_WARNING
- YRCValidationResponse.YRC_VALIDATION_ERROR

In the getModifiedValue() method, add the logic to set the modified value for a cell of an advanced column that you are currently editing. You can use this method to update some other property based on the current one or to change the format of the property.

Binding Combo Box Cell Editors

About this task

Binding combo box cell editors indicates binding a combo box inside an editable extended table. To set bindings for a combo box cell editor:

Procedure

1. Create a binding object for the combo box.
2. Set the list binding, description binding, and code binding for the combo box.

   Note: Only set the list binding, description binding, and code binding for the combo box.
3. Set the binding data of the advanced column with the YRCComboBindingData binding object as an argument. For example:

   ```
   advclmBindingData.setBindingData(cmbBindingData);
   ```

   where cmbBindingData is the combo box binding object and YRCConstants. YRC_COMBO_BINDING_DEFINITION is the key used for identifying the combo box binding object.
4. Add the advanced column binding data object to the advanced column data map object. For example:

   ```
   bindingDataMap.put("extn_AdvClm1", advclmBindingData);
   ```

   where bindingDataMap is the hash map binding object, extn_AdvCml is the name of the advanced column that you added using the Rich Client Platform Extensibility Tool, and advclmBindingData is the advanced column binding object.

Creating a Binding Object for a File Upload Column in a Table in the Rich Client Platform

Procedure

1. Create a new instance of the YRCTblClmBindingData binding class. For example:

```
YRCTblClmBindingData advclmBindingData = new YRCTblClmBindingData();
```

   where `YRCTblClmBindingData` is the class to set bindings for the advanced column and `advclmBindingData` is the binding object.

   Note: You need to create the binding object and set bindings for the advanced column only if you have added an advanced column through extensibility.

2. Set the attribute binding for the advanced column. For example:

```
advclmBindingData.setAttributeBinding("ItemID");
```

3. (Optional) Set multiple attribute binding for the advanced column. For example:

```
advclmBindingData.setAttributeBinding("ItemID;Code");
```

   where `ItemID` is the namespace of the model.

4. (Optional) Set the key binding for the advanced column. For example:

```
advclmBindingData.setKey("item_details");
```

   where `item_details` is the key.

   Note: If you are specifying multiple attribute binding for the column, this step is mandatory.

5. Set the server image configuration for the advanced column to display the image from the server. For example:

```
advclmBindingData.setServerImageConfiguration("YRCConstants_IMAGE_SMALL");
```

   where `IMAGE_SMALL` is the value of the Name attribute of the Config element, which is defined in the configuration file.

6. To sort the advanced column, set the SortReqd attribute value to **true**. For example:

```
advclmBindingData.SortReqd(true);
```

7. To localize the advanced column data, set the DbLocaliseReqd attribute value to **true**. For example:

```
advclmBindingData.DbLocaliseReqd(true);
```

8. To filter an advanced column based on some value, set the setFilterRequired attribute value to **true**. For example:

```
advclmBindingData.setFilterRequired(true);
```

9. Add the advanced column binding data object to the data map object. For example:

```
bindingDataMap.put("extn_AdvClml", advclmBindingData);
```

   where `bindingDataMap` is the hash map binding object, `extn_AdvClml` is the name of the advanced column added using the Rich Client Platform Extensibility tool, and `advclmBindingData` is the advanced column binding object.

10. Set the name of the back end table, which is the parent table of the file attachment table. For example:

    ```
    advclmBindingData.setFileUploadTable("<table_name>");
    ```

    where <table_name> is the back end table.
11. Specify that the table column is being used for a file upload. For example:

    ```
    advclmBindingData.setFile(true);
    ```

    where advclmBindingData is the table column being used for a file upload.
12. Specify the method to set the file upload type. For example:

    ```
    advclmBindingData.setFileUploadType("ONSUBMIT");
    ```

    where ONSUBMIT indicates that the file needs to be uploaded to the server before the calling of the API which persists the file to the database.
13. Indicate if the model update will be handled by the application and/or a single table column is used to upload files for multiple entities. For example:

    ```
    advclmBindingData.setModelUpdateDynamic(boolean modelUpdateDynamic)
    ```

    where modelUpdateDynamic=true if the model update is performed by the application.

Creating a Binding Object for a File Upload Text Box in the Rich Client Platform

Procedure

1. Create a new instance of the YRCTextbindingData binding class. For example:

   ```
   YRCTextbindingData txtBindingData = new YRCTextbindingData();
   ```

   where YRCTextbindingData is the class to set bindings for the text box and txtBindingData is the binding object.
2. Set the name of the text box by using the binding object that you created. For example:

   ```
   txtBindingData.setName("txtOrderNo");
   ```

   where txtOrderNo is the name of the text box and txtBindingData is the binding object.
3. Set the source binding for the text box. For example:

   ```
   txtBindingData.setSourceBinding("OrderDetails:Order/@OrderNo");
   ```

   where OrderDetails is the namespace of the model.
4. (Optional) Set the multiple source binding for the text box. For example:

   ```
   txtBindingData.setSourceBinding("OrderDetails:Order/@OrderNo;Order/@Status");
   ```

   where OrderDetails is the namespace of the model.
5. (Optional) Set the key binding for the text box. For example:

   ```
   txtBindingData.setKey("order_details");
   ```

   where order_details is the key.

   Note: If you are specifying multiple source binding for the text box, this step is mandatory.
6. Set the target binding for the text box. For example:

```
txtBindingData.setTargetBinding("OrderListDetails:OrderList/Order/@OrderNo");
```

where `OrderListDetails` is the namespace of the model.

7. Set the multiple target binding for the text box. For example:

```
txtBindingData.setTargetBinding("OrderDetails:Order/@OrderNo;OrderStatus/Order/@Status");
```

where `OrderDetails` is the namespace of the model.

8. Set the binding data for the text box by associating the binding object to the key. For example:

```
txtOrderNo.setData(YRCConstants.YRC_TEXT_BINDING_DEFINATION,txtBindingData);
```

where `txtOrderNo` is the reference variable name of the text box, which you specified in the visual editor, and `YRCConstants.YRC_TEXT_BINDING_DEFINATION` is the key used for identifying the text box binding object.

9. Set the type of the field to **true** for file upload. For example:

```
txtOrderNo.setFile(true);
```

where `txtOrderNo` is the field value which is the location of the file which needs to get uploaded.

10. Specify that the file validations for the maximum allowed size and file types which will be allowed for for file upload are table-specific, given by the table name. For example:

```
txtOrderNo.setFileUploadTable("<table_name>");
```

where `<table_name>` is the table for which the file validations are allowed.

11. Specify the method to set the file upload type.

Note: The application only has to use this method, and does not have to use the getFileUploadType method.
For example:

```
txtOrderNo.setFileUploadType("ONSELECT");
```

where `ONSELECT` indicates that the file needs to be uploaded as soon as the file is registered.

```
txtOrderNo.setFileUploadType("ONSUBMIT");
```

where `ONSUBMIT` indicates that the file needs to be uploaded to the server before the API which persists the file to the database is called. For one control at any time, the file upload type can be either `ONSELECT` or `ONSUBMIT`

12. Indicate if the model update will be handled by the application and/or a single text control is used to upload files for multiple entities. For example:

```
txtOrderNo.setModelUpdateDynamic(boolean modelUpdateDynamic)
```

where modelUpdateDynamic=true if the model update is performed by the application.

Note: For file validation to occur, the field should be associated with a data type.

Localizing Controls and Defining Themes for Rich Client Platform Applications

About this task

To localize controls, text, or strings:

Procedure

1. Specify the *key* = *value* pair in your *Plug-in id*_bundle.properties file at the plug-in level. Here, key is the resource key and value is the literal displayed for the corresponding locale.
2. Replace *value* with the translated value. For example, to localize a label:
   a. Set the key value pair bundle file for the label. For example:
      ```
      Customer_Address=Customer Address
      ```
      where Customer_Address is the key and Customer Address is the value for the key.
   b. Set the text of the label with the key as the input argument. For example:
      ```
      lblCustAdd.setText("Customer_Address");
      ```
      where lblCustAdd is the reference variable name of the label, which you specified in the visual editor.

      Note: The Rich Client Platform automatically localizes labels, buttons, group headers, tab folder items, and table column headers. Therefore, the literals used in the binding object must be resource bundle keys if they need to be translated to different languages.

Defining Themes for Controls

For theming controls, define the new theme entries in the *Plug-in_id_theme_name*.ythm file.

Calling APIs and Services for Rich Client Platform Applications

About this task

Calling an API or service is as follows:

Procedure

1. Create a command in the *Plug-in id*_commands.ycml file and associate the API or services to be called with the command. Make sure that the code used for calling an API or service is written in the behavior class. For example, to call the getOrderList API, you must create a command with the name as getOrderList and in the APIName attribute enter getOrderList. For more information about creating commands, see "Creating Commands".
2. Create a YRCApiContext class object. For example:
   ```
   YRCApiContext context = new YRCApiContext();
   ```
3. Set the command name for the context. For example:
   ```
   context.setApiName("getOrderList");
   ```
   where getOrderList is the command name that you created in the *Plug-in id*_commands.ycml file.
4. Set the form id for the context. For example,
   ```
   context.setFormId(getFormId());
   ```

5. Set the input XML document that is passed to an API or service. For example:

   ```
   context.setInputXml(getTargetModel("Order").getOwnerDocument());
   ```

   where Order is the namespace of the XML model.
6. (Optional) Set the key for the context that you created. For example:

   ```
   context.setUserData("InitialData","1");
   ```

   where InitialData is the key and 1 is the value for this key. The value of the key is used to uniquely identify the context. This step is mandatory, if you are calling the same API multiple times. For more information about calling same API multiple times, see the *Calling the Same API/Service Multiple Times* topic
7. Call the API or service. For example,

   ```
   callApi(context);
   ```

8. After the API or service call is complete, the Rich Client Platform calls the handleApiCompletion() method of behavior class to validate the output and process it. Therefore, you can write the API completion logic in this method. For example:

   ```
   public void handleApiCompletion (YRCApiContext context) {
      if(context.getInvokeAPIStatus() < 0) {
          // Add logic for the failure condition
      }
      else {
       if(YRCPlatformUI.equals(context.getApiName(),"getOrderList")){
    setOrderList(context);                        }
              }
   }
   ```

   Note: If an API or service call fails, the Rich Client Platform throws an exception.

Calling the Same API/Service Multiple Times

About this task

The Rich Client Platform enables you to call the same API or service multiple tiles. For example, if you want to call the getOrderList API three times with a different input XML model as input to the API:

Procedure

1. Create three objects of the YRCApiContext class. For example,

   ```
   YRCApiContext context1 = new YRCApiContext();
   YRCApiContext context2 = new YRCApiContext();
   YRCApiContext context3 = new YRCApiContext();
   ```

2. Set the same command name for each context. For example,

   ```
   context1.setApiName("getOrderList");
   context2.setApiName("getOrderList");
   context3.setApiName("getOrderList");
   ```

3. Set the form id for each context. For example,

   ```
   context1.setFormId(getFormId());
   context2.setFormId(getFormId());
   context3.setFormId(getFormId());
   ```

4. Set the different input XML document for each context.

   ```
   context1.setInputXml(getTargetModel("Order").getOwnerDocument());
   context2.setInputXml(getTargetModel("OrderDetail").getOwnerDocument());
   context3.setInputXml(getTargetModel("OrderList").getOwnerDocument());
   ```

   where Order, OrderDetail, and OrderList are the namespaces of the different XML model.

5. Set the key for each context using the UserData key.

```
context1.setUserData("InitialData","1");
context2.setUserData("InitialData","2");
context3.setUserData("InitialData","3");
```

   where InitialData is the key and 1,2,and 3 are the values for this key based on the each context. The value of the key is used to uniquely identify each context.
6. Call the API for each context.

```
callApi(context1);
callApi(context2);
callApi(context3);
```

7. In the handleApiCompletion() method, get the context.getUserData() to identify each context. Then, validate and process the output at each API level. For example,

```
public void handleApiCompletion(YRCApiContext context) {
   if(YRCPLatformUI.equals(context.getUserData("InitialData"),"1")) {
//Add your own logic for validating and processing the
//output at each API level.
    }
else if(YRCPLatformUI.equals(context.getUserData("InitialData"),"2")) {
//Add your own logic for validating and processing the
//output at each API level.
    }
else if(YRCPLatformUi.equals(context.getUserData("InitialData"),"3")) {
//Add your own logic for validating and processing the
//output at each API level.
}
```

Calling Multiple APIs/Services

About this task

The Rich Client Platform enables you to call multiple APIs. To call multiple APIs, define multiple commands in the <*Plug-in id*>_commands.ycml file.

Note: IBM recommends that you call all APIs at the same time to reduce the network traffic.

For example, if there are three combo boxes: cmbStatus, cmbEnterprise, and cmbCountry, you must call APIs or services to display a list of values for these combo boxes. For instance, the values displayed for the cmbStatus combo box depends on the output of the getOrderList API. The values displayed for the cmbEnterprise combo box depends on the output of the getShipNodeList API. The values displayed for the cmbCountry combo box depends on the output of the getCommonCodeList API output.

To call multiple APIs or services:

Procedure

1. In the *Plug-in id*_commands.ycml file, define three commands with names as getOrderList, getShipNodeList, and getCommonCodeList. Associate an API or service with each of these commands using the APIName attribute. Make sure that the code used for calling an API or service is written in the behavior class.

   For more information about creating commands, see "Creating Commands".
2. Create a YRCApiContext class object. For example:

```
YRCApiContext context = new YRCApiContext();
```

3. To call multiple APIs, set the command names for the commands that you created in the *Plug-in id*_commands.ycml file. For example:

```
context.setApiNames(new
String[]{"getOrderStatusList","getShipNodeList","getCommonCodeList"});
```

4. Set the form id for the context. For example,

```
context.setFormId(getFormId());
```

5. Set input XMLs for multiple APIs. For example:

```
context.setInputXmls(new
Document[]{"getOrderStatusList","getShipNodeList","getCommonCodeList"});
```

6. (Optional) Set Unique key for multiple commands. For example:

```
context.setUserData("InitialData","1");
```

7. Call the API or service. For example:

```
callApi(context);
```

8. Invoke the handleApiCompletion() method to validate and process the output at each API level. You must call this method after executing the callApi() method. For example:

```
public void handleApiCompletion(YRCApiContext context) {
String[] sAPINames = context.getApiNames();
if(YRCPlatform.equals(sAPINames[0],"getOrderStatusList")){
setOrderList(context);                                          }
else if(YRCPlatform.equals(sAPINames[1],"getShipNodeList")){
setShipNodeList(context); }
else if(YRCPlatform.equals(sAPINames[2],"getCommonCodeList")){
setCommonCodeList(context); }}
```

Adding New Rich Client Platform Screens as Pop-ups

About this task

You can display the new screen as a pop-up screen, when you click on a button. You need to associate the new screen with the button. To display a new screen as a pop-up screen:

Procedure

1. Add a new button to an existing screen.

   Note: When adding the new button, make sure that you check the "Validation Required?" box.
2. Synchronize the extension behavior for the screen.
3. In the navigator view, expand the plug-in project that you created when setting up the development environment.
4. Expand the package and open the extension behavior class, which you specified in Step 2.
5. In the validateButtonClick() method, add the logic to display the new screen in a pop-up window or dialog window, when you click on the newly added button. For example,

```
ViewOrderDetails screen  = new ViewOrderDetails(new
Shell(Display.getDefault()), SWT.NONE, bindingData, filterObjectList);
YRCDialog oDialog = new YRCDialog(screen,400,400,"OrderDetails",null);
oDialog.open();
```

   where ViewOrderDetails is the class name of the screen and OrderDetails is the title of the dialog window that displays this screen.

Adding New Rich Client Platform Screens to Menu Commands

About this task

You can display the new screen as a menu item. The menu items are connected to the actions by specifying the action identifier for a specific menu item. Configure the action which gets invoked, when you click on the menu item or a related task. To add new screens to a Rich Client Platform application menu, define screens in the resources. All the resources of have a set of primary properties that are common to all types of resources. For example, all resources have a Resource ID. These resources are used to define screens. In addition to primary properties, each type of resource has a set of unique properties that is specific to a particular type of resource.

For adding new screens to an application in the resources, define the Resource ID, URL, and Resource Type. The Resource ID is a unique identifier for each resource. The URL contains the Rich Client Platform ActionId of the class that invokes the screen, which is defined in the plugin.xml file.

Note: The action identifiers are not specific to menus. The Related Tasks can also invoke these actions.

The class that invokes the newly created screen must be created by extending the YRCAction class. In the YRCAction class, the execute() method invokes the action configured by you when you click on a menu item. In the execute() method you can write a code to open the new screen either in a pop-up window or an editor.

For more information about defining resources, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales Configuration Guide *Application Platform Configuration Guide*.

Displaying New Rich Client Platform Screens in an Editor

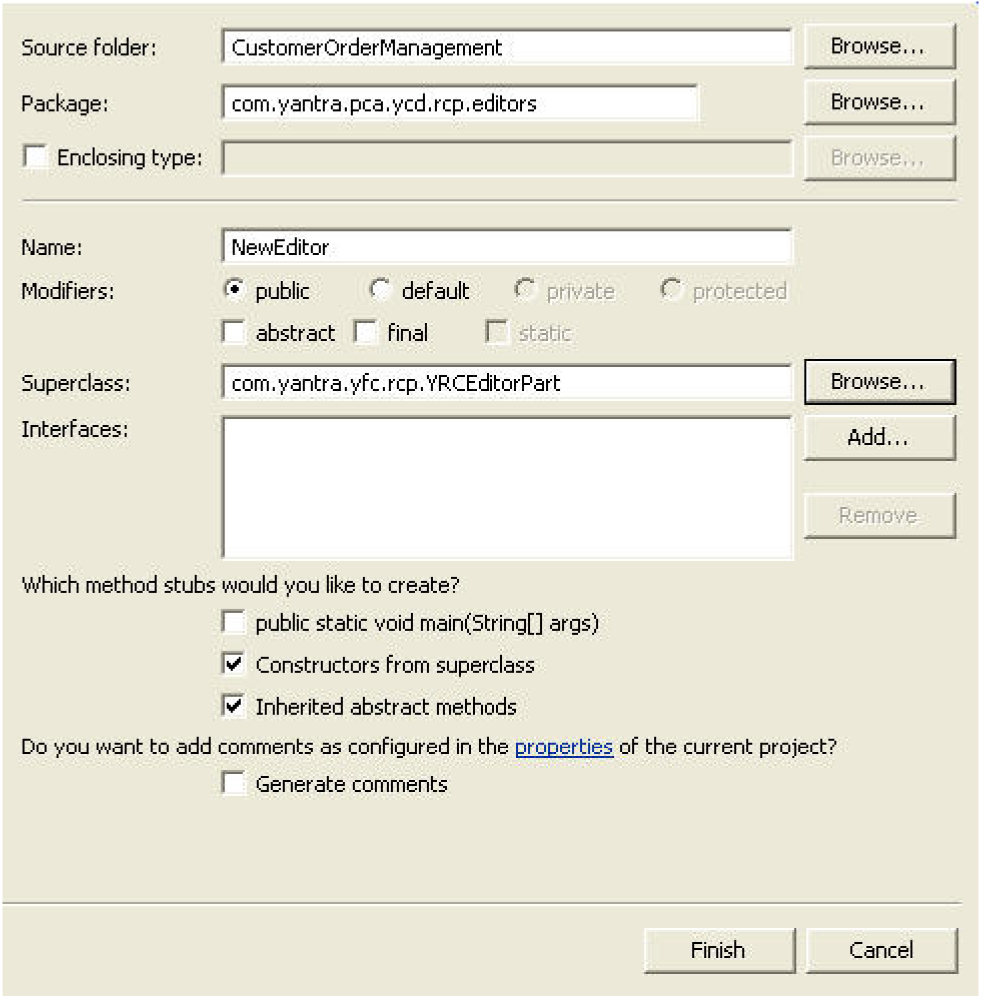
About this task

You can display the new screen in an editor when you click on a button or a menu item or a related task. To display a new screen in an editor:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click **Add**. From the New Extension window, select org.eclipse.ui.editors extension point from the list.
6. Click **Finish**.
7. Select the **org.eclipse.ui.editors** extension point. The Extension Details panel displays.

8. In the Extension Details panel, enter the properties of org.eclipse.ui.editors extension point.
9. Right-click on the **org.eclipse.ui.editors** extension and select **New** > **editor**. The editor extension element gets created.
10. Select the editor extension element. The Extension Element Details panel displays.
11. Enter the properties of the editor extension element.
12. In id*, enter the identifier for the editor.
13. In icon, browse to the path of the icon that you want to associate with this editor.
14. In class, to specify the implementation class, do any of the following:
    - Click **Browse**. The Select Type pop-up window displays. Select the class that extends the YRCEditorPart class.
    - Click on the **class:** hyperlink. The Java Attribute Editor window displays.



Field	Description
Source folder:	The name of the source folder that you selected to store the editor class automatically displays. Click **Browse** to browse to the folder that you want to specify as the source folder.

Field	Description
Package:	The name of the package that you selected to store the editor class automatically displays. Click **Browse** to browse to the package where you want to store the editor class.
Name	Enter the name of the editor class.
Superclass:	Click **Browse**. The Superclass Selection window displays. In Choose a type, enter `YRCEditorPart` and click **OK**.
Constructors from superclass	Check this box. The system automatically creates the constructor for the YRCEditorPart superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the YRCEditorPart superclass.
Finish	When you click on this button, the system creates the new editor class in the selected folder or package.

15. To open the new screen in the specified editor using the menu item, define a new resource in the resources for the new menu item.
16. In the execute() method of the action set that you associated with the menu item in the previous step do the following:
    - Create a new input element to pass to the YRCEditorInput object.
    - Create a new input object to pass to the YRCEditorInput object, if required.
    - Create a new YRCEditorInput object. Pass the input element and the input object that you created (if required). Also pass the array of strings, which contains the attribute of the input element, and the related task.
    - Open the editor that you created for the new screen by passing the Id of the editor to the YRCPlatformUI.openEditor() method.

    Note: Make sure that the editor identifier that you pass to the YRCPlatformUI.openEditor() method is same as specified in Step 12.

    For example,

```
Element inputElement = YRCXmlUtils.createFromString
("<Order OrderNo=\"YCD001\"/>").getDocumentElement();
Object inputObject = new String("");
YRCEditorInput editorInput = new YRCEditorInput(inputElement,
inputObject, new String[]{"OrderNo"}, "YCD_TASK_QUICK_ACCESS");
YRCPlatformUI.openEditor("com.yantra.qa.editors.QAEdito", editorInput);
```

Chapter 10. Configuring File Uploads and Downloads

Uploading and Downloading

You can set up your Web UI Framework (WUF) and Rich Client Platform (RCP) applications to attach files to records of a table, upload files, and download attached files. Uploads and downloads flow between internal file systems and the application database.

With this feature, you can do the following:
- Perform a file upload or download in a single transaction.
- Have multiple file uploads within a single API call.
- Restrict the types and sizes of files being uploaded.
- Perform authentication during a file download.
- Perform authorization during a file download.
- Download one file at a time.
- Attach more than one document per record.

This feature has the following limitations:
- Two or more files with the same name cannot be uploaded and attached to a particular record of a database table. The application must have customized validation to prevent two or more files with the same name from being uploaded.
- It is not supported for the Console UI Framework, HTTPTester, End Point, and remote clients.
- In Windows, file names are case-insensitive within a particular directory, but in Linux, file names are case-sensitive. At the Sterling Application Platform level, the case insensitivity of file names will not be checked before the files are imported into the database. That is, at the Sterling Application Platform level, abc.txt and ABC.txt are two different files. Application developers will have to decide if the application allows files of the same name but in different cases to be uploaded for a record of a database table.

Using yfs.properties to Configure File Uploads

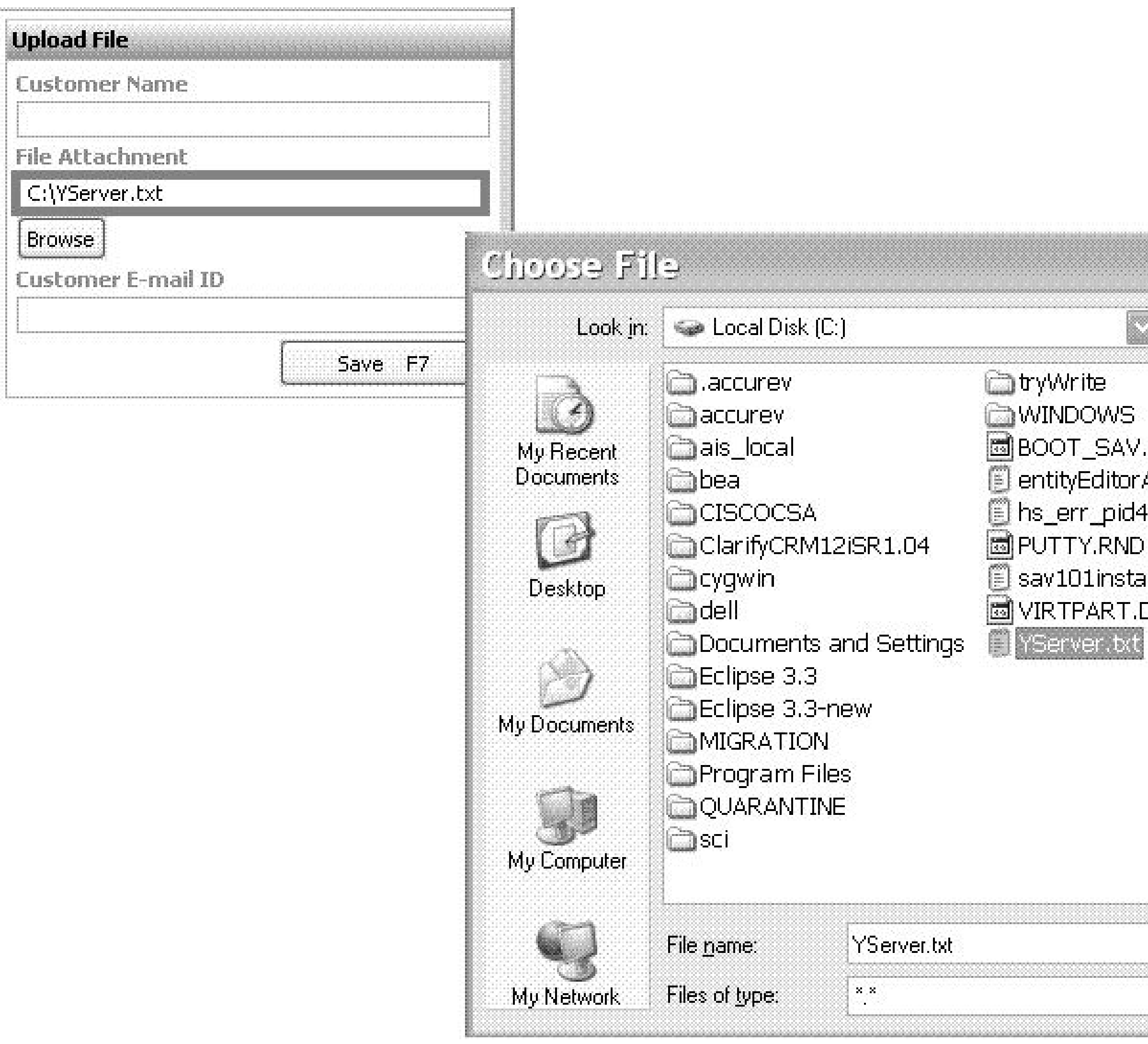
The yfs.properties file includes several properties that you need to set to upload files in the Web UI Framework and the Rich Client Platform. For more information, refer to other upload-related tasks and the inline documentation in the actual yfs.properties file.

Note: Do not directly edit or change the yfs.properties* files. To make changes to the properties in these files, you must use the customer_overrides.properties file. IBM does not recommend that you modify or change any properties in files ending with .in because newer versions or patches of the product will overwrite your changes. IBM also does not recommend that you change a property file that has a corresponding .in file because the setupfiles script will re-create the properties file again, thus causing you to lose your changes.

© Copyright IBM Corp. 1999, 2013

Configuring File Uploads

For file uploads in the Rich Client Platform, you can use only text fields to capture the file location. Text binding data and table column binding data have methods to indicate that a particular control might contain file data. The following graphic shows how a file could be uploaded:



You also need to do the following to configure file uploads.

Note: For more information about the yfs.properties settings, refer to the information about using yfs.properties to configure file uploads in the Rich Client Platform.

- Use the yfs.properties file to specify whether to start uploading a file as soon as a field gets file location information and the validation passes, or after the file is selected and validations are successful.
- Enable RCP uploads using the following methods in the YRCTextBindingData class. For more information, refer to the Javadocs.`public boolean isFile()`

  Checks to see if a field is the file data type to upload a file.`public void setFile(boolean isFile)`

Sets a field as the location of the file that needs to get uploaded.`public void setFileUploadTable(String tableName)`

Sets the name of the back end table.`public String getFileUploadTable()`

Gets the name of the back end table.`public String getFileUploadType()`

Gets the file upload type, which determines when a file is uploaded to the server.`public void setFileUploadType(String fileUploadType)`

Sets the file upload type, which determines when a file is uploaded to the server.`public boolean isModelUpdateDynamic()`

Indicates if the model update will be handled by the application and/or a single text control is used to upload files for multiple entities.`public void setModelUpdateDynamic(boolean modelUpdateDynamic)`

Indicates that the application will handle the model update. It sets the model update property to **dynamic**, which indicates that the application will handle the model update.

- Specify the allowed file types in the yfs.properties file.
- Specify the maximum allowed file size in the yfs.properties file.

The preference in which the maximum allowed size and allowed types are picked is:

1. Table-specific properties.
2. Application-specific properties.
3. Properties from yfs.properties which are not table-specific.
4. Default properties.

You can configure the user interface to tell users that a file upload is in progress or when the file upload is complete, using event-related code. Also, utility methods are available for canceling file uploads that are in progress and for deleting files that have been uploaded.

When you move your cursor out of a field that is enabled for file upload, the following validation occurs to determine if a file being uploaded is valid or not. For validation to occur, the field should be associated with a datatype.

1. Whether a file is present.
2. Whether it is a file and not a directory.
3. Whether the file is an allowed file type.
4. Whether the file is within the maximum allowed size.

On the successful streaming of files to the server for temporary storage, a unique file ID is generated for each of the files uploaded. The format of the ID is `<UserID>_<System Time in Milliseconds>_<CurrentThreadID>_<Counter>`. This ID information is used when file information has to be persisted in the database as part of the API call.

The following example code shows XML code that uses this ID. The example ID is `consoleadmin_1273071853344_16_1`. This is the model update after the files were persisted temporarily. In the example, FileLocation is the location at which the files were persisted temporarily.

```
<Organization>
    <OrgList>
        <Org Organization_Key="XYZ" OrgAttachmentDoc="admin_1267599660674_1" />
        <FileAttachments>
            <FileAttachment FileLocation="C:\bea\user_projects\domains\<domain_name>\
             AdminServer\consoleadmin\2297091940258334\consoleadmin_1273071853344_16_1"
```

```
                FileSize=123 FileName="abc.txt" FileContentType="text/plain"/>
          </FileAttachments>
     </OrgList>
     <User User_Key="admin" UserAttachmentTxt="admin_1267599660675_2"/>
     <FileAttachments>
          <FileAttachment FileLocation="C:\bea\user_projects\domains\<domain_name>\
           AdminServer\consoleadmin\2297091940258334\consoleadmin_1273071853344_16_1"
           FileSize=345 FileName="xyz.txt" FileContentType="text/plain"/>
     </FileAttachments>
</Organization>
```

An API failure occurs when:

- There is a failure to send across files to the server.
- There is an exception while trying to persist files in the database as part of the API call (that is, after the files have been placed successfully on the server in temporary storage).

To rename an already existing file in the database, the attribute value **NewFileName=new_file_name** has to be set to the file attachment element to update an already existing file attachment record with the new file name.

Securing Uploaded Files

Interfaces for the following tasks help with securing uploaded files in the Web UI Framework and the Rich Client Platform:

- Virus scan
- Encoding/decoding of files

Virus Scan

You can plug in logic to scan an uploaded file for viruses. Use the yfs.properties file to plug in the scanner. A default implementation of virus scanning is not available.

The virus scanner interface includes the following methods. For more information, refer to the Javadocs.

- public PLTVirusScanResponse scan(InputStream stream, PLTFileHandlerObj fileObject)

  Called first during uploads and downloads to scan the file input stream as is during the upload/download request.
- public PLTVirusScanResponse scan(PLTFileHandlerObj fileObject)

  Called only during uploads to enable the scanning of the file written to temporary directory.

Encoding/Decoding of Files

Files that are uploaded can be encoded before they are streamed to a temporary server folder as part of the upload process. These files are encoded by objects of the IFileEncoderDecoder interface, which has two methods (encode and decode). A default implementation is provided for this interface. Use the sc.file.upload.encoder property of the yfs.properties file to plug in the encoder/decoder. For more information about the sc.file.upload.encoder property, refer to the inline documentation in the actual yfs.properties file.

Note: The default encoder increases file sizes significantly.

The following includes basic information about each method. For more information, refer to the Javadocs.

- public void encode(InputStream iStream, OutputStream oStream)

  Encodes the input stream and places it in the output stream. Called during file upload.
- public void decode(InputStream iStream, OutputStream oStream)

  Decodes the input stream and places it in the output stream. Called during file download.

Encoding and decoding can negatively affect performance. If the sc.file.upload.encoder property is not set, then encoding and decoding are skipped.

Compression/Decompression of BLOB Data

You can compress and decompress data that is stored in columns and tables. The actual file content, stored as bytes in the PLT_FILE_DATA table, follows this mechanism by marking the column as **CompressionSupported=true** and **UseCompression=true**.

Upload Error Messages

Number	Error Code	Error Condition	Error-Related More Information or Any Message
1	PLTF001	If sc.file.upload.dir is not set	Mandatory temporary directory not configured in properties file.
2	PLTF007	If user is not authenticated.	File Upload - Authentication Failed.
3	PLTF006	Session is not valid.	InvalidSession
4	PLTF009	If the file size exceeds the allowed size.	Cannot proceed with file upload. Maximum file size exceeded for file :<filename>. Maximum file size allowed(bytes):<max_size>
5	PLTF011	If the file type is not allowed.	Cannot proceed with file upload. File type not allowed. File :<filename>. Allowed file types:<allowed_types>
6	PLTF004	If file is not found after virus scan in quarantine directory.	File is not found in the directory specified after virus scan.
7	PLTF008	If virus scan fails.	Error message sent by the virus scan implementation should be shown.

Number	Error Code	Error Condition	Error-Related More Information or Any Message
8		If file cannot be deleted after it is placed in the temporary directory.	Error: Cannot delete file at location:
9		If file is not found at the specified location, during file delete.	Error: File not present at location:
10		If during file delete, if yfs.properties is not configured or file location is not provided.	Error: File upload temporary is not configured in yfs.properties or file location is not provided.
11		If file is successfully deleted in temporary location.	File deleted at location:
12	PLTF013	Virus scan implementation class cannot be found or is incorrect.	Error message sent will be available on the client.
13	PLTF014	Encoder-decoder implementation class cannot be found or is incorrect.	Error message sent will be available on the client.
14	PLTF015	File upload temporary directory is not found.	Error message sent will be available on the client.
15	PLTF016	File upload quaranti ne directory is not found.	Error message sent will be available on the client.
16	PLTF017	Processing of multipart/form-data request failed.	Error message sent will be available on the client.
17	PLTF018	File upload property is not defined correctly.	Error message sent will be available on the client.

Configuring File Downloads

A utility method is used to download files in the Rich Client Platform. The method call will take in the API name, API input XML, and the file download input XML with the FileAttachmentKey. The key is mandatory and points to the file that has to be downloaded from the database. Only one file can be downloaded at a time. This request is posted to the servlet (PLTFileDownloadServlet) that handles the download of files.

The syntax of the method is `public static void downloadFile(String APIName, Document inputXML, Document fileDownloadinput)`, with these parameter definitions:

- APIName

The name of the API that needs to be called. This is for authorization. On the success of this API, files are fetched.

- inputXML

  The input XML required for the execution of the API. This is for authorization.
- fileDownloadinputThis is used after authorization finishes and the API call succeeds. This XML must contain the FileAttachmentKey, which is like a primary key to get the file attached. It points to the file that has to be downloaded from the database. It is mandatory for the application to pass this information to download a file.

Files are downloaded to the location specified by the VM argument `-DFileDownloadDir` that is specified in the *application*.ini file. This argument can contain `@user.home` in the path, which is the user's home directory. If the argument is not specified, then the default location will be `@user.home/RCP/FileDownloads`.

Events are fired to indicate that a download is in progress and when downloading is complete. You need to configure the UI to indicate when the download begins and ends.

Securing Downloaded Files

When downloading files in the Web UI Framework and the Rich Client Platform, to authenticate and authorize the file download request, an API and API input has to be passed along with the file download input. The API will be called with the API input passed. If the API is successful, the file specified in the file download input will be downloaded.

The API which has to be called for file download authorization has to be registered with the class PLTFileDownloadAPIRegistry using the method API(String appCode, String appName). The client request should provide the API name and input. The following is an example of the API input (for the getUserList API):

```
<User UserKey="">
  <FileAttachments>
    <FileAttachment FileName=""/>
  </FileAttachments>
</User>
```

As part of the API invocation, authorization and other security tasks are handled by API security, token validation, and other tasks. This API is invoked to check if the API succeeds. If it does, then the user is authorized to download files. Then the file for which FileAttachmentKey is provided in the file download input XML file will be downloaded.

The API template is provided inside the template/filedownloadapi folder in the resources.jar file. This is the template for the API which has to be called for authorization. It should be as small as possible, for example:

```
<User UserKey="">
  <FileAttachments>
    <FileAttachment FileName="" FileAttachmentKey=""/>
  </FileAttachments>
</User>
```

Authorization for the file download servlet (PLTFileDownloadServlet) is configured via web.xml. The context parameter is `sc-file-download-authorization-required`. By default, this property value is **TRUE**.

For download, APIName,APIInputXML, and FileDownloadInputXML are all required and have to be passed from the client. If the web.xml entry `sc-file-download-authorization-required` is set to **FALSE**, then the FileAttachmentKey in FileDownloadInputXML is used to fetch the file. If `sc-file-download-authorization-required` is set to **TRUE**, then the API passed by the client is called with the API input passed. The template for this API call is described earlier in this topic. If the API call succeeds, then the file is downloaded.

The user invoking the servlet should have permission for the API. On the successful completion of this API, the file data is decompressed, decoded, and the client is served with the file it requested for download.

Download Error Messages

Number	Error Code	Error Condition	Error-Related More Information or Any Message
1	PLTF006	Session is not valid.	InvalidSession
2	PLTF002	File download input is not provided.	File download input is not provided.
3	PLTF012	FileAttachment key is not provided in file download input.	FileAttachment key is not provided
4	PLTF003	File download authorization is required, but API input is not provided.	API input to download file is not provided.
5	PLTF003	File download authorization is required, but API name is not registered.	API provided for file download file authorization is not registered.
6	PLTF003	File download authorization is required, but API template is not found at location template/ FileDownloadapi.	Template not found for API authorization.
7	YCP0045	If after authorization, FileAttachment key cannot be found in the output.	File attachment record does not exist or the user is not authorized to access it.
8	YCP0045	If the file attachment record is not found in the file attachment table.	File attachment record does not exist.
9	PLTF008	If virus scan implementation finds virus.	Error message sent by the virus scan implementation should be shown.

Number	Error Code	Error Condition	Error-Related More Information or Any Message
10	PLTF004	If after virus scan, the file is not found within the quarantine directory.	File is not found in the directory specified after the virus scan.
11	PLTF013	Virus scan implementation class cannot be found or is incorrect.	Error message sent will be available on the client.
12	PLTF014	Encoder-decoder implementation class cannot be found or is incorrect.	Error message sent will be available on the client.

Structuring the File Upload and Download

You can implement file upload/download functionality through interface contracts in the Web UI Framework and the Rich Client Platform. The upload/download implementations can be plugged in either through web.xml as context parameters or programmatically using exposed methods.

This enables you to do the following:

- Customize upload/download functionality in the same way that you can customize authorization, authentication, and other tasks.
- Implement upload/download functionality for applications that do not use the Sterling Application Platform.

These tasks use the following base classes which are present both in the Sterling Application Platform (the platform_afc.jar file) and in the base file attachment jar file (platform_fa.jar) that is used when not using the Sterling Application Platform:

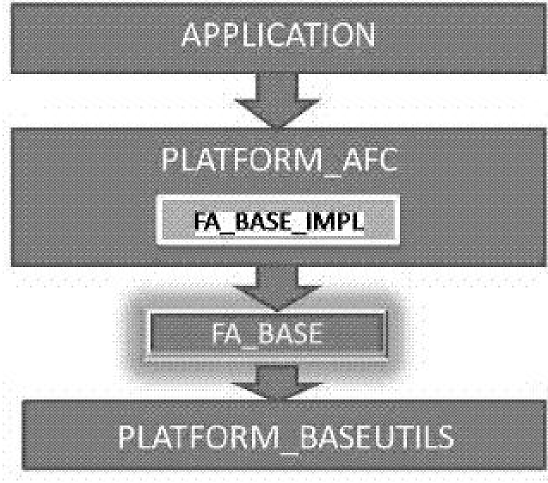
- The interfaces IFileUploadProvider and IFileDownloadProvider
- The abstract class PLTFileUploadProvider
- The servlets PLTFileUploadservlet and PLTFileDownloadServlet

Note: For applications consuming the platform_afc.jar file, the platform_fa.jar file does not have to be added in the classpath since the file attachment base classes of the platform_fa.jar file are included in the platform_afc.jar file.

Default File Upload/Download Implementations

The following graphic shows the default implementation of upload/download functionality in applications based on the Sterling Application Platform but which do not use the Web UI Framework (like the Rich Client Platform):

In the graphic, Platform and Platform_AFC refer to the Sterling Application Platform.



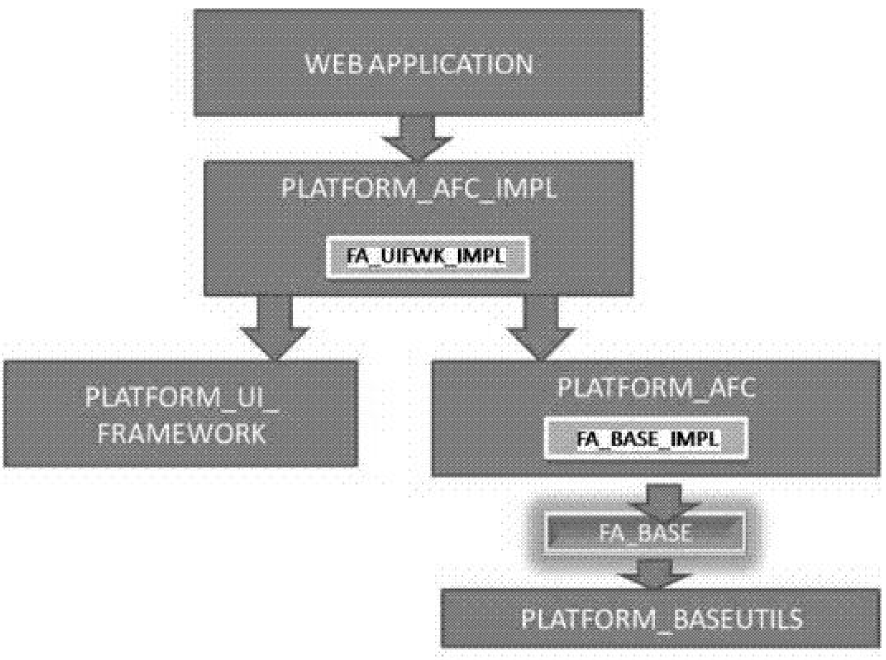
FA_BASE → FILE ATTACHMENT BASE FRAMEWORK
FA_BASE_IMPL → FILE ATTACHMENT BASE IMPLEMENTATION

Default implementation classes are as follows:

- For upload - com.sterlingcommerce.woodstock.util.frame.file.impl.PLTFileUploadProviderImpl
- For download - com.sterlingcommerce.woodstock.util.frame.file.impl.PLTFileDownloadProviderImpl

The following graphic shows the default implementation of upload/download functionality in applications based on the Sterling Application Platform and use the Web UI Framework:

In the graphic, Platform and Platform_AFC refer to the Sterling Application Platform.



FA_BASE → FILE ATTACHMENT BASE FRAMEWORK
FA_BASE_IMPL → FILE ATTACHMENT BASE IMPLEMENTATION (xml input/output)
FA_UIFWK_IMPL → UIFWK specific FILE ATTACHMENT IMPLEMENTATION (json input/output)

Default implementation classes are as follows:

- For upload - com.sterlingcommerce.ui.web.platform.file.SCUIFileUploadProviderImpl
- For download - com.sterlingcommerce.ui.web.platform.file.SCUIFileDownloadProviderImpl

Plugging in File Upload/Download Implementations through web.xml

The following context parameters are used when plugging in file upload/download implementations through web.xml:

- For upload - sc-file-upload-provider
- For download - sc-file-download-provider

The value of the above mentioned context parameters should be a qualified Java class path which implements IFileUploadProvider and IFileDownloadProvider for upload and download, respectively.

Sample context parameters to be plugged in applications based on the Sterling Application Platform but which do not use the WUF (note the PLT prefix in the method name):

```
<context-param>
     <param-name>sc-file-upload-provider</param-name>
     <param-value>
com.sterlingcommerce.woodstock.util.frame.file.impl.PLTFileUploadProviderImpl
     </param-value>
</context-param>
<context-param>
```

```
        <param-name>sc-file-download-provider</param-name>
        <param-value>
com.sterlingcommerce.woodstock.util.frame.file.impl.PLTFileDownloadProviderImpl
        </param-value>
   </context-param>
```

Sample context parameters to be plugged in applications based on the Sterling Application Platform and are on WUF (note the SCUI prefix in the method name):

```
   <context-param>
        <param-name>sc-file-upload-provider</param-name>
        <param-value>
com.sterlingcommerce.ui.web.platform.file.SCUIFileUploadProviderImpl
        </param-value>
   </context-param>
   <context-param>
        <param-name>sc-file-download-provider</param-name>
        <param-value>
com.sterlingcommerce.ui.web.platform.file.SCUIFileDownloadProviderImpl
        </param-value>
   </context-param>
```

Note: By default, the context parameters required to plug in the default implementations will not be provided. Consuming applications will have to add the context parameters in the web.xml file or set it using setFileUploadProvider and setFileDownloadProvider methods exposed in the PLTFileUploadDownloadHelper class.

Plugging in File Upload/Download Implementations Programmatically Using Exposed Methods

The methods setFileUploadProvider and setFileDownloadProvider are exposed in the PLTFileUploadDownloadHelper class to plug in file upload/download implementations programmatically using interface contracts.

Sample usage of the above mentioned set methods:

```
PLTFileUploadDownloadHelper.setUploadProviderImpl(uploadImpl);
PLTFileUploadDownloadHelper.setDownloadProviderImpl(downloadImpl);
```

Here, uploadImpl and downloadImpl are class objects which implement IFileUploadProvider and IFileDownloadProvider, respectively.

Uploading and Downloading Using Interface Contracts without the Sterling Application Platform

About this task

In the Web UI Framework and the Rich Client Platform, you can plug in file upload/download implementations without the Sterling Application Platform using the file attachment base framework (platform_fa.jar):

Procedure

1. Code the file upload provider to implement IFileUploadProvider or extend PLTFileUploadProvider, whichever is appropriate.
2. Code the file download provider to implement IFileDownloadProvider.

3. Add the context parameters `sc-file-upload-provider` and `sc-file-download-provider` to web.xml or set the file upload/download providers in the initialization servlet of the application, with the qualified Java class path of the implementations.
4. Add file upload/download servlet and servlet mapping entries in the web.xml file.

   Sample servlet and servlet mapping entries for file upload and download servlet:

```
<servlet id="Servlet_55">
    <description>File Upload Servlet</description>
    <display-name>File Upload Servlet</display-name>
    <servlet-name>FileUploadServlet</servlet-name>
    <servlet-class>
com.sterlingcommerce.woodstock.util.frame.file.base.servlets.PLTFileUploadServlet
    </servlet-class>
</servlet>
<servlet id="Servlet_56">
    <description>File Download Servlet</description>
    <display-name>File Download Servlet</display-name>
    <servlet-name>FileDownloadServlet</servlet-name>
    <servlet-class>
com.sterlingcommerce.woodstock.util.frame.file.base.servlets.PLTFileDownloadServlet
    </servlet-class>
</servlet>
<servlet-mapping id="ServletMapping_30">
    <servlet-name>FileUploadServlet</servlet-name>
    <url-pattern>/FileUploadServlet/*</url-pattern>
</servlet-mapping>
<servlet-mapping id="ServletMapping_31">
    <servlet-name>FileDownloadServlet</servlet-name>
    <url-pattern>/FileDownloadServlet/*</url-pattern>
</servlet-mapping>
```

Chapter 11. Creating and Adding Wizards

Phase 1: Create Wizard Definitions

A wizard is used for any task consisting of many steps, which must be completed in a specific order. A wizard acts as an interface to lead a user through a complex task, using step-by-step pages. It can also be used for the execution of any task involving a sequential series of steps.

Wizard behavior means that each wizard page in a sequence contains a "Next" button, which the user clicks to move to the next wizard page after entering data or configuring information in the current wizard page. If the user decides to go back and change any information entered in a previous wizard page, each wizard page contains a "Previous" button that the user clicks to go back. At the end of the wizard sequence, the user clicks a Finish button to begin the particular process.

Note: Before you can start creating wizards, you must set up the development environment. For more information about setting up development environment, see "The Development Environment for Rich Client Platform Applications".

Creating a Wizard Definition

You can create a new or modify an existing wizard definition by adding wizard entities and wizard transitions. The flow of the wizard depends on the output value of a wizard rule. The wizard definition is created in the *Plug-in_id*_commands.ycml file.

Note: You must use a separate *Plug-in_id_wizard_name*.ycml file for each wizard definition you create.

Creating a Wizard Definition with the Rich Client Platform Wizard Editor

About this task

The Rich Client Platform Wizard Editor is used for creating or modifying the wizard definition. To open the *Plug-in_id*_commands.ycml file in the Rich Client Platform Wizard Editor:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. In the navigator window, expand the plug-in project that you created when setting up the development environment.
4. Right-click the *Plug-in_id_wizard_name*.ycml file, select **Open With** > **Rich Client Platform Wizard Editor** from the pop-up menu.
5. The Rich Client Platform Wizard Editor displays. A Palette is available on the right-hand side, containing a list of tools that can be used to create or modify wizard definition, for example, Marquee, Transition, Rule, Page, and ChildWizard.

© Copyright IBM Corp. 1999, 2013

6. In the Properties view, in Wizard Description, enter the description for the new wizard.

Phase 2: Create Components to Implement a Wizard Definition

After creating the new wizard definition, you need to create the individual wizard components that provide implementation for the new wizard.

A wizard contains a wizard class and a wizard behavior class.

- Wizard Class—The container class that controls the UI of the wizard.
- Wizard Behavior Class—The container class that controls the behavior of the wizard. Primary function of this class is to display the appropriate wizard pages in a wizard.

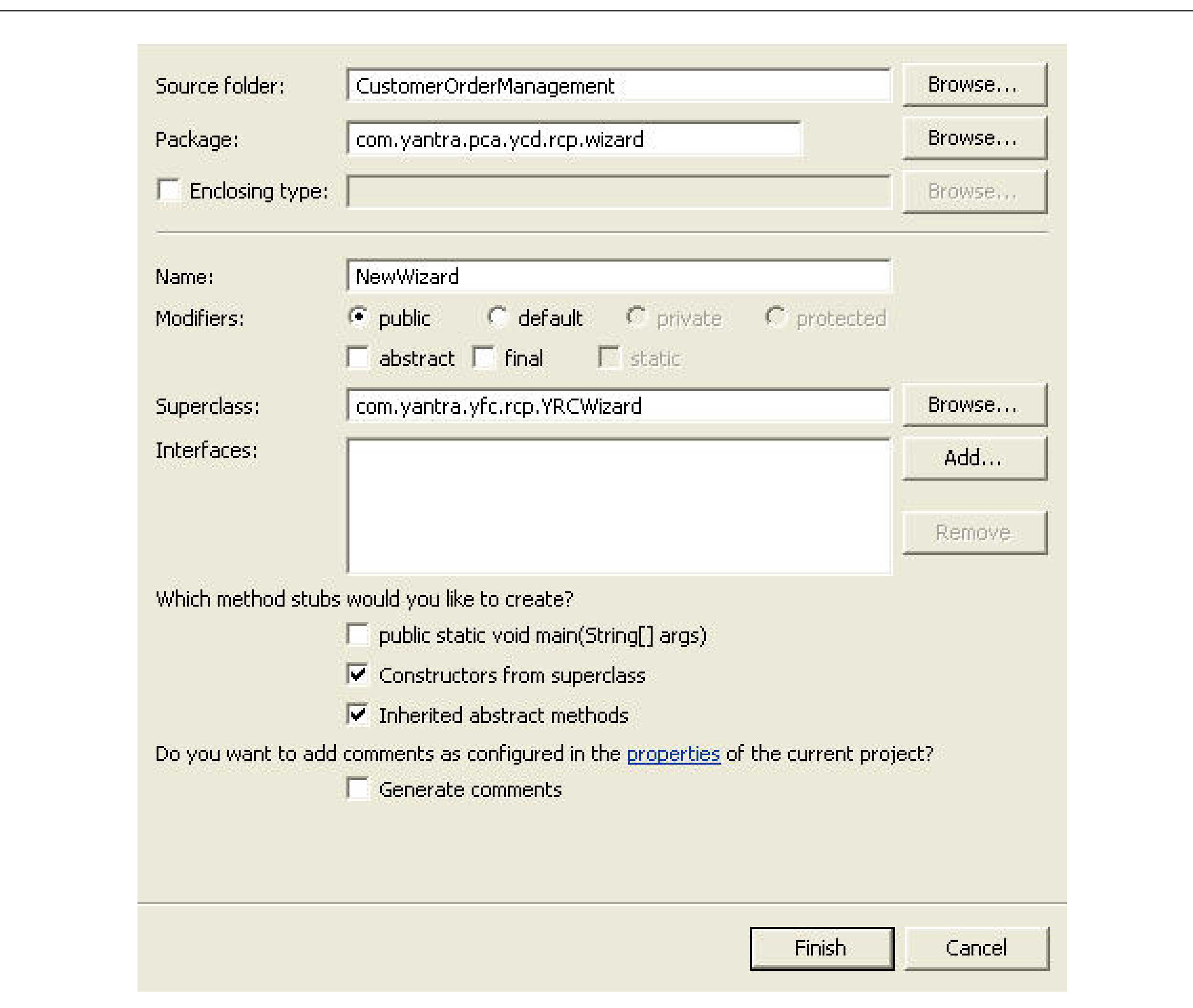
Creating Wizard Class

About this task

To create a wizard class:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. In the navigator window, expand the plug-in project that you created when setting up the development environment.
4. To store the wizard class, right-click on a folder or package and select **New** > **Class** from the pop-up menu. The New Java Class window displays.



Field	Description
Source folder:	The name of the source folder that you selected to store the wizard class automatically displays. Click **Browse** to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the wizard class automatically displays. Click **Browse** to browse to the package where you want to store the wizard class.
Name	Enter the name of the wizard class.
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter YRCWizard and click OK.
Constructors from superclass	Check this box. The system automatically creates the constructor for the YRCWizard superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the YRCWizard superclass.

5. Click **Finish**. The system creates the new wizard class in the folder or package selected by you.
6. Open the newly created wizard class in the Java Editor.

7. Right-click in the editor window, select **Source** > **Override/Implement Methods...** from the pop-up menu. The Override/Implement Methods window displays.
8. Select getFormId(), getHelpId(), and createBehavior() methods from the list of methods provided in the YRCWizard class and click OK.
9. Create the field FORM_ID and specify the identifier of the wizard in this field. For example,

   ```
   public static final String FORM_ID = "com.yantra.pca.ycd.rcp.wizard.NewWizard";
   ```

   Override the getFormId() method and return this form id field.

   Note: The identifier specified in the FORM_ID field should be the same form id that you specified in the wizard definition.
10. In the wizard class constructor initialize the wizard by calling the initializeWizard() method. For example,

    ```
    public NewWizard(String wizardId, Composite parent,
    Object wizardInput, int style) {
           super(wizardId, parent, wizardInput, style);
           initializeWizard();
        }
    ```
11. Override the createBehavior() method. Create and return an instance of the wizard behavior class. For example,

    ```
    protected YRCWizardBehavior createBehavior() {
    myBehavior = new RCPRIWizardBehavior(this, FORM_ID);
    return myBehavior;
        }
    ```

Creating Wizard Behavior Class

About this task

To create a wizard behavior class:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. In the navigator window, expand the plug-in project that you created when setting up the development environment.
4. To store the wizard behavior class, right-click on a folder or package and select **New** > **Class** from the pop-up menu. The New Java Class window displays.

Field	Description
Source folder:	The name of the source folder that you selected to store the wizard behavior class automatically displays. Click **Browse** to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the wizard behavior class automatically displays. Click **Browse** to browse to the package where you want to store the wizard behavior class.
Name	Enter the name of the wizard behavior class.
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter YRCWizardBehavior and click OK.
Constructors from superclass	Check this box. The system automatically creates the constructor for the YRCWizardBehavior superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the YRCWizardBehavior superclass.

5. Click **Finish**. The system creates the new wizard behavior class in the folder or package selected by you.
6. Open the newly created wizard behavior class in the Java Editor.
7. Right-click in the editor window, select **Source** > **Override/Implement Methods...** option from the pop-up menu. The Override/Implement Methods window displays.

8. Select initPage(String) method from the list of methods provided in the YRCWizardBehavior class and click **OK**.
9. In the initPage(String) method, write the code for performing wizard page specific operations. For example, setting the model, calling API or service, and so forth.
10. In the createPage(String pageIdToBeShown, Composite pnlRoot) method, return an instance of a wizard page corresponding to the pageId. This method is called internally.

```
public IYRCComposite createPage(String pageIdToBeShown,
  Composite pnlRoot) {
IYRCComposite page=null;
If(pageIdToBeShown.equalsIgnoreCase(NewWizardPage1.FORM_ID))
{ NewWizardPage1 temp = new NewWizardPage1(pnlRoot, SWT.NONE);
  page = temp;
} else if(pageIdToBeShown.equalsIgnoreCase(NewWizardPage2.FORM_ID)) {NewWizardPage2
temp = new NewWizardPage2(pnlRoot,     SWT.NONE);
  page = temp;
}
return page;
}
```

For more information about creating wizard page class, see "Creating Wizard Page Class".

Phase 3 Adding Components to Wizard Definition

About this task

After creating the individual wizard components, you need to add all these components to the wizard definition. You can add the following wizard components to the wizard definition:

- Wizard Rule
- Wizard Page
- Sub-task
- Wizard Transition

Adding a Rule to a Wizard Definition

About this task

To add a new wizard rule:

Procedure

1. Open the *Plug-in_id_wizard_name*.ycml file using the Rich Client Platform Wizard Editor. For more information about opening the Rich Client Platform Wizard Editor, see "Creating a Wizard Definition with the Rich Client Platform Wizard Editor".
2. From the Palette, click **Rule** and select **Rule**.
3. Place the Rule in the Wizard Definition editor where you want to add it.
4. In the Properties view, in Description, enter the description for the new wizard rule.
5. In the Properties view, in Id, enter the unique identifier for the wizard rule.
6. In Impl, enter the fully qualified path of the implementation class for this wizard rule. For example:

```
java:com.yantra.pca.ycd.rcp.wizard.rules.NewWizardRule1
```

Here, com.yantra.pca.ycd.rcp.wizard.rules is the package name and NewWizardRule1 is the wizard rule class name that provides the implementation for this wizard rule.

In a wizard rule, you can also specify a Greex rule you want to evaluate. To specify the Greex rule, in Impl, enter the relative path of the *.greex file, which contains the Greex rule you want to evaluate. For example:

```
greex:greexRules/test1.greex
```

Here, test1.greex is the name of the Greex file. greexRules is the directory in your plug-in project containing the *.greex file.

Note: You can use only those Greex rules whose return type is either string or Boolean.

7. In isLast, enter `true` if the wizard rule is the last entity in the wizard flow.
8. In Namespaces, enter the namespaces of the XML model based on which the output of a rule is computed. You can enter more than one namespace for a rule by separating them with a semi-colon. These namespaces are defined in the *Plug-in_id*_command.ycml file.
9. In Outputs, enter one or more output values that will be returned by the wizard rule. Based on the output values returned by the wizard rule, the control is transferred to a wizard entity. You can define more than one output value for a wizard rule by separating them with semi-colon.
10. In Starting, enter `true` if the wizard rule is the starting entity in the wizard flow.
11. In X and Y, enter the X and Y co-ordinates for this wizard rule. These co-ordinates are relative to the (0,0) co-ordinates of the top-left corner.

Adding a Page to a Wizard Definition

About this task

To add a new wizard page:

Procedure

1. Open the Plug-in_id_*wizard_name*.ycml file using the Rich Client Platform Wizard Editor.
2. From the Palette, click **Page** and select **Page**.
3. Place the page in the Wizard Definition editor where you want to add it.
4. In the Properties view, in Description, enter the description for the new wizard page.
5. In the Properties view, in Id, enter the unique identifier for the wizard page.
6. In Can Be Hidden, enter `true` if you want to hide the wizard page in the wizard flow.
7. In Impl, enter the fully qualified path of the implementation class for the wizard page that you created. For example:

   ```
   com.yantra.pca.ycd.rcp.wizard.pages.NewWizardPage1
   ```

   Here, com.yantra.pca.ycd.rcp.wizard.pages is the package name and NewWizardPage1 is the wizard page class name that provides the implementation for this wizard page.
8. In isLast, enter `true` if the wizard page is the last entity in the wizard flow.
9. In isLast, enter `true` if the wizard page is the starting entity in the wizard flow.

10. In X and Y, enter the X and Y co-ordinates for this page. These co-ordinates are relative to the (0,0) co-ordinates of the top-left corner.

Adding a Sub-task to a Wizard Definition

About this task

To add a new sub-task:

Procedure

1. Open the *Plug-in_id_wizard_name*.ycml file using the Rich Client Platform Wizard Editor.
2. From the Palette, click **ChildWizard** and select **ChildWizard**.
3. Place the ChildWizard in the Wizard Definition editor where you want to add it.
4. In the Properties view, in Description, enter the description for the new sub-task.
5. In the Properties view, in the Id field, enter the unique identifier for the sub-task.
6. In Impl, enter the fully qualified path of the implementation class for this sub-task. For example:

   ```
   java:com.yantra.pca.ycd.rcp.wizard.subtasks.NewSubTask1
   ```

   Here, the com.yantra.pca.ycd.rcp.wizard.subtasks is the package name and NewSubTask1 is the sub-task class name that provides the implementation for this sub-task.
7. In isLast, enter `true` if the sub-task is the last entity in the wizard flow.
8. In Namespaces, enter the namespaces of the XML model that will be used for the sub-task. You can enter more than one namespace for a sub-task by separating them with a semi-colon. These namespaces are defined in the *Plug-in_id*_command.ycml file.
9. In Starting, enter `true` if the sub-task is the starting entity in the wizard flow.
10. In X and Y, enter the X and Y co-ordinates for this sub-task. These co-ordinates are relative to the (0,0) co-ordinates of the top-left corner.

Adding a Transition to a Wizard Definition

About this task

Wizard transition is used to transfer control from one wizard entity to another wizard entity. The wizard transition value is compared with the output of the wizard rule, and based on this value, the control is transferred to the next wizard entity. You can define same wizard transition identifier for multiple wizard transitions. However, they must have different values associated with them.

To add a new wizard transition:

Procedure

1. Open the *Plug-in_id*_commands.ycml file using the Rich Client Platform Wizard Editor..
2. From the Palette, select **Transition**.
3. Click the wizard entity from which you want to transfer the control and then click the wizard entity to which you want to transfer the control.

4. In the Properties view, in Transition Id, enter the identifier for this wizard transition.

   Note: Multiple wizard transitions originating from a wizard rule must have same wizard Transition ID. There can only be one transition from a wizard page.

Creating Wizard Page Components

A wizard page contains a wizard page class and a wizard page behavior class.

- Wizard Page Class—The container class that controls the UI of the wizard page to take inputs from the user. In addition, this class takes care of binding controls, and so forth.
- Wizard Page Behavior Class—The container class that controls the behavior of the wizard page.

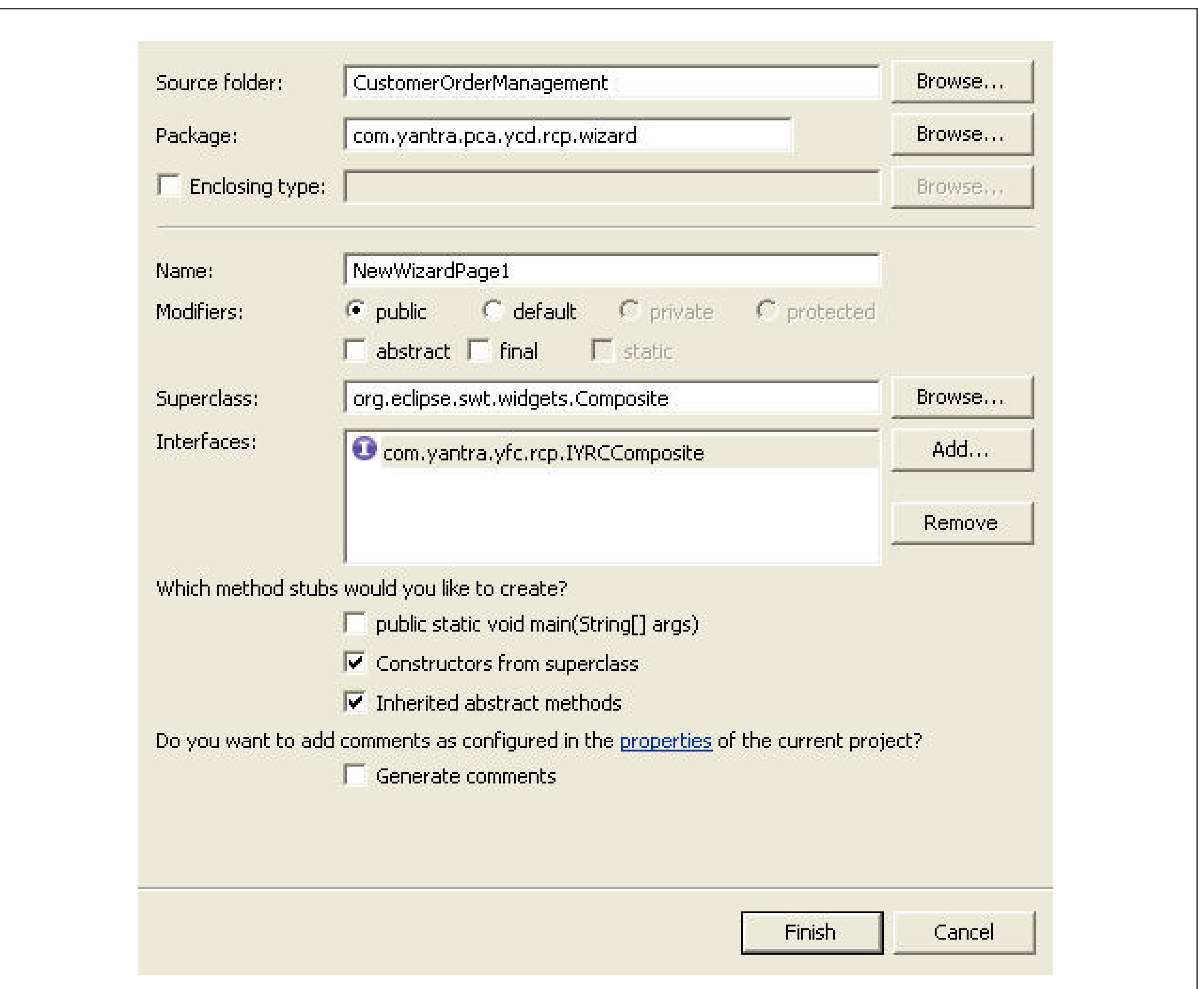
Creating Wizard Page Class

About this task

To create a wizard page class:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. In the navigator window, expand the plug-in project that you created when setting up the development environment.
4. To store the wizard page class, right-click on a folder or package and select **New** > **Class** from the pop-up menu. The New Java Class window displays.



Field	Description
Source folder:	The name of the source folder that you selected to store the wizard page class automatically displays. Click Browse to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the wizard page class automatically displays. Click Browse to browse to the package where you want to store the wizard page class.
Name	Enter the name of the wizard page class.
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter Composite and click OK.
Interfaces:	Click **Add**, the Implemented Interfaces Selection window displays. In Choose a type, enter IYRCComposite and click OK.
Constructors from superclass	Check this box. The system automatically creates the constructor for the Composite superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the Composite superclass.

5. Click **Finish**. The system creates the new wizard page class in the folder or package selected by you.

6. Open the wizard page java class in the java editor and design the UI to take the inputs from the user as per the requirements.
7. In the getFormId() method return the unique FORM_ID of this wizard page.

   Note: The string identifier specified in the FORM_ID field should be the same form id that you specified for the wizard page in the wizard definition.
8. In the constructor of the wizard page class, create an instance of wizard page behavior class and store is as a field. For example,

```
public NewWizardPage1(Composite parent, int style) {
super(parent, style);
this.setData("FORMID", FORM_ID);
myBehavior = new NewWizardPage1Behavior(this);
}
```

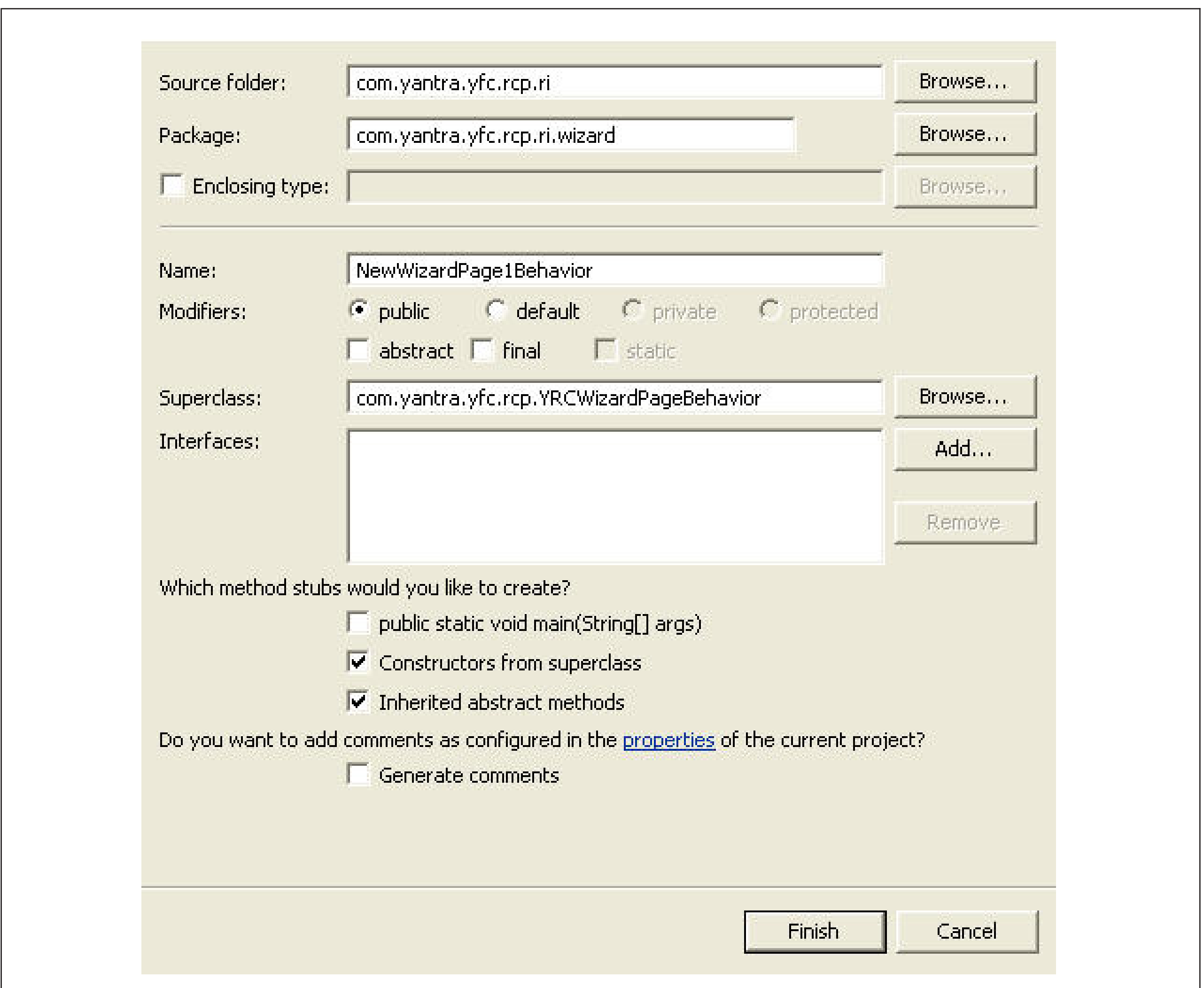
Creating Wizard Page Behavior Class

About this task

To create a wizard page behavior class:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. In the navigator window, expand the plug-in project that you created.
4. To store the wizard page behavior class, right-click on a folder or package and select **New** > **Class** from the pop-up menu. The New Java Class window displays.



Field	Description
Source folder:	The name of the source folder that you selected to store the wizard page behavior class automatically displays. Click **Browse** to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the wizard page behavior class automatically displays. Click **Browse** to browse to the package where you want to store the wizard page behavior class.
Name	Enter the name of the wizard page behavior class.
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter YRCWizardPageBehavior and click OK.
Constructors from superclass	Check this box. The system automatically creates the constructor for the YRCWizardPageBehavior superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the YRCWizardPageBehavior superclass.

5. Click **Finish**. The system creates the new wizard page behavior class in the folder or package selected by you.

6. In the initPage(String) method, write the code for performing wizard page specific operations. For example, setting the model, calling API or service, and so forth.

Creating Wizard Rule Components

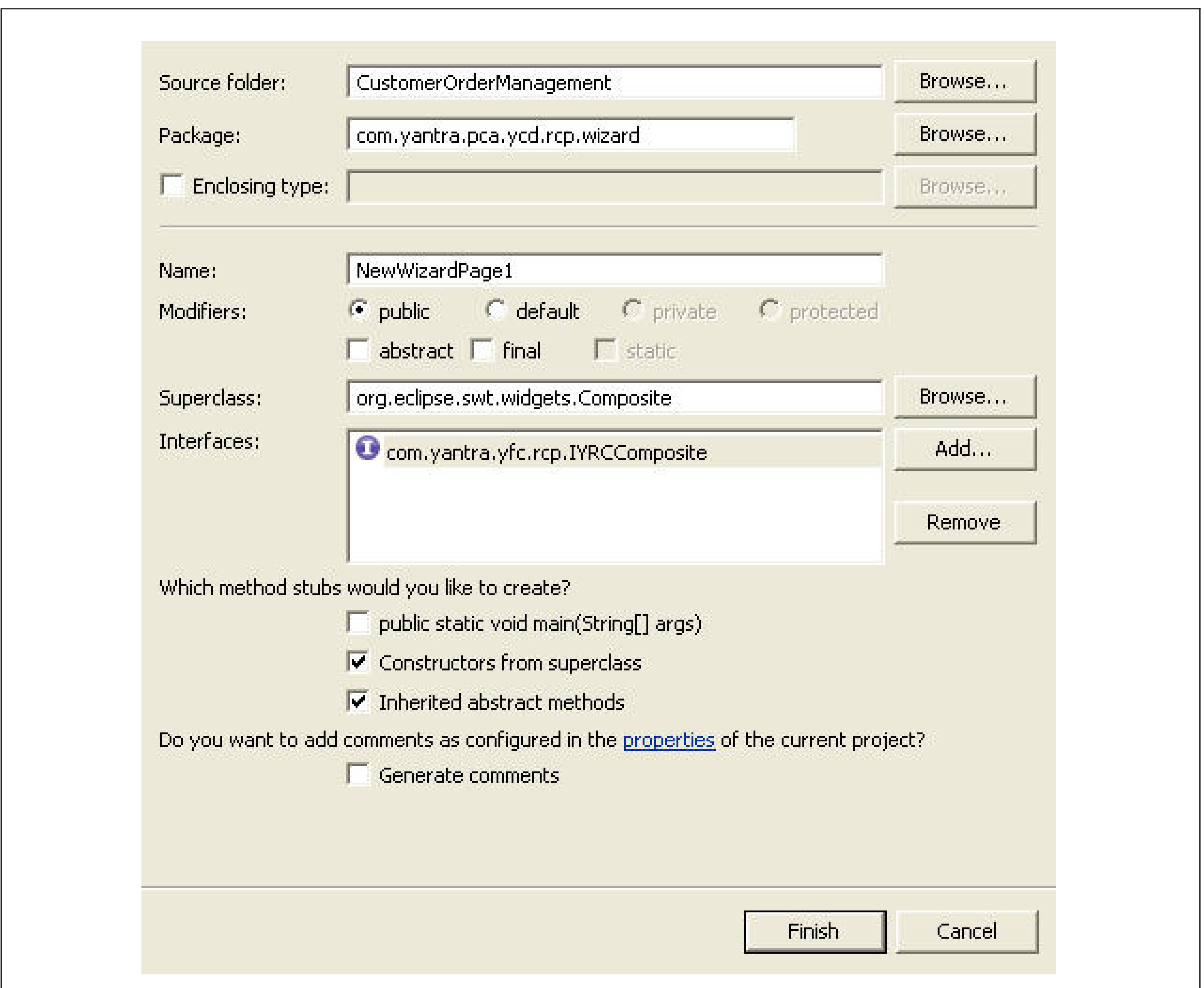
About this task

A wizard rule contains a wizard rule class. This class performs logical computations to evaluate various output values. Based on these output values flow of the wizard is decided.

To add a new wizard rule:

Procedure

1. Start the Eclipse SDK.
2. From the menu bar, select **Window** > **Show View** > **Navigator**. The plug-in project is displayed in the Navigator view.
3. In the navigator window, expand the plug-in project that you created when setting up the development environment.
4. To store the wizard rule class, right-click on a folder or package and select **New** > **Class** from the pop-up menu. The New Java Class window displays.



Field	Description
Source folder:	The name of the source folder that you selected to store the wizard rule class automatically displays. Click **Browse** to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the wizard rule class automatically displays. Click **Browse** to browse to the package where you want to store the wizard rule class.
Name	Enter the name of the wizard rule class.
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter `Object` and click **OK.**
Interfaces:	Click Add, the Implemented Interfaces Selection window displays. In Choose a type, enter `IYRCWizardRule` and click **OK.**
Constructors from superclass	Check this box. The system automatically creates the constructor for the superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the superclass.

5. Click **Finish**. The system creates the new wizard rule class in the folder or package selected by you.

6. In the execute(HashMap namespaceModelMap) method, write the logic for computing the output value using the model that is passed in the namespaceModelMap parameter and return the output value of the rule. This method is called when the wizard flow needs the output value of this rule. Wizard flow is based on the output of this rule, as defined in the wizard definition. The HashMap contains a list of all namespaces and the corresponding models. These namespaces are defined in the *Plug-in_Id_wizard_name*.ycml file.

Registering the Wizard Command File

After you create the wizard definition in the *Plug-in_id_wizard_name*.ycml commands file, you must register this command file with the plug-in project that you created, if required. This is required in order to make your new wizard work according to the flow that you have defined in the commands file.

Adding Wizards as Pop-ups in Rich Client Platform Applications

About this task

You can display the new wizards as a pop-up screen, when you click on a button. You need to associate the new wizard with the button. To display a new wizard as a pop-up screen:

Procedure

1. Add a new button to an existing screen.

   Note: When adding the new button, make sure that you check the "**Validation Required?**" box.
2. Synchronize the extension behavior for the screen.
3. In the navigator view, expand the plug-in project that you created when setting up the development environment.
4. Expand the package and open the extension behavior class, which you specified in Step 2.
5. In the validateButtonClick() method, add the logic to display the new screen in a pop-up window or dialog window, when you click on the newly added button. For example,

```
Object WizardInput = YRCXmlUtils.createfromString("<WizardInput/>");
NewWizard wizard  = new NewWizard(NewWizard.FORM_ID,
Shell(Display.getDefault()), WizardInput, SWT.NONE);
wizard.start();
YRCDialog oDialog = new YRCDialog(wizard,400,400,"AddLine",null);
oDialog.open();
```

Adding Wizards to Menu Commands in Rich Client Platform Applications

You can display the new wizard as a menu item. The menu items are connected to the actions by specifying the action identifier for a specific menu item. Configure the action which gets invoked, when you click on the menu item or a related task. To add new wizards to a Rich Client Platform application menu, define wizards in the your application. All the resources have a set of primary properties that are common to all types of resources. For example, all resources have a Resource ID.

These resources are used to define wizards. In addition to primary properties, each type of resource has a set of unique properties that is specific to a particular type of resource.

For adding new wizards to an application in the Resources, define the Resource ID, URL, and Resource Type. The Resource ID is a unique identifier for each resource. The URL contains the Rich Client Platform ActionId of the class that invokes the wizard, which is defined in the plugin.xml file.

Note: The action identifiers are not specific to menus. The Related Tasks can also invoke these actions.

The class that invokes the newly created wizard must be created by extending the YRCAction class. In the YRCAction class, the execute() method invokes the action configured by you when you click on a menu item. In the execute() method you can write a code to open the new wizard either in a pop-up window or an editor.

For more information about defining resources, see theSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales *Configuration Guide Application Platform Configuration Guide*.

Adding Wizards to Editors in Rich Client Platform Applications

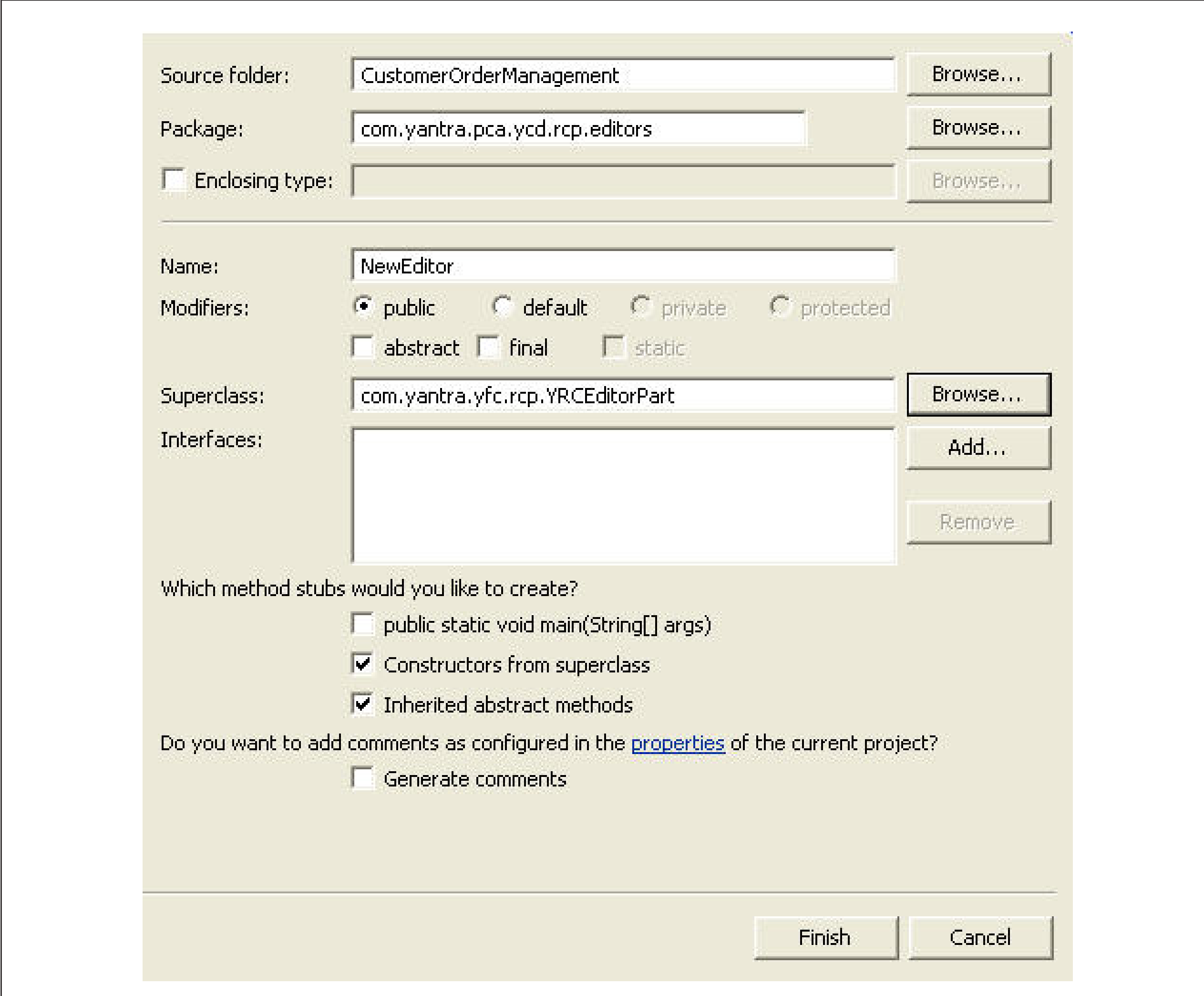
About this task

You can display the new wizard in an editor when you click on a button or a menu item or a related task. To display a new wizard in an editor:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Window** > **Show View** > **Navigator**.
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **org.eclipse.ui.editors extension point** from the list.
7. Click **Finish**.
8. Select the org.eclipse.ui.editors extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of org.eclipse.ui.editors extension point.
10. Right-click on org.eclipse.ui.editors extension and select **New** > **editor**. The editor extension element gets created.
11. Select the editor extension element. The Extension Element Details panel displays.
12. Enter the properties of the editor extension element.
13. In id*, enter the identifier for the editor.

14. In icon, browse to the path of the icon that you want to associate with this editor.
15. In class, to specify the implementation class, do any of the following:
    - Click **Browse**. The Select Type pop-up window displays. Select the class that extends the YRCEditorPart class.
    - Click on the class: hyperlink. The Java Attribute Editor window displays.



Field	Description
Source folder:	The name of the source folder that you selected to store the editor class automatically displays. Click Browse to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the editor class automatically displays. Click Browse to browse to the package where you want to store the editor class.
Name	Enter the name of the editor class.
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter `YRCEditorPart` and click **OK**.
Constructors from superclass	Check this box. The system automatically creates the constructor for the YRCEditorPart superclass.

Field	Description
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the YRCEditorPart superclass.
Finish	When you click on this button, the system creates the new editor class in the selected folder or package.

16. Open the newly created editor class in the Java Editor.
17. In the createPartControl() method create and return the instance of the new wizard that you created. For example,

```
public Composite createPartControl(Composite parent, String task) {
Object WizardInput = YRCXmlUtils.createfromString("<WizardInput/>");
NewWizard wizard  = new NewWizard(NewWizard.FORM_ID, parent,
WizardInput, SWT.NONE);
wizard.start();
return wizard;
}
```

18. To open the new wizard in the specified editor using the menu item, define a new resource in the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales Resources for the new menu item.
19. In the execute() method of the action set that you associated with the menu item in the previous step do the following:
    - Create a new input element to pass to the YRCEditorInput object.
    - Create a new input object to pass to the YRCEditorInput object, if required.
    - Create a new YRCEditorInput object. Pass the input element and the input object that you created (if required). Also pass the array of strings, which contains the attribute of the input element, and the related task.
    - Open the editor that you created for the new screen by passing the Id of the editor to the YRCPlatformUI.openEditor() method.

    Note: Make sure that the editor identifier that you pass to the YRCPlatformUI.openEditor() method is same as specified in Step 12.

    For example:

```
Element inputElement = YRCXmlUtils.createFromString
("<Order OrderNo=\"YCD001\" />").getDocumentElement();
Object inputObject = new String("");
YRCEditorInput editorInput = new YRCEditorInput(inputElement,
inputObject, new String[]{"OrderNo"}, "YCD_TASK_QUICK_ACCESS");
YRCPlatformUI.openEditor("com.yantra.qa.editors.QAEdito", editorInput);
```

Chapter 12. Creating Related Tasks

About Related Tasks

The Rich Client Platform provides the ability to create related tasks by grouping a set of appropriate tasks based on the functionality. You must define the group and category for each related task. All related tasks can belong to multiple categories, but limited to one group.

In addition, you can move the existing related tasks from one group to another group by creating a new related task in the second group with same the action ID as given in the first group and then hiding the existing related task in the first group.

Extending the YRCRelatedTasks Extension Point

About this task

The Rich Client Platform provides the YRCRelatedTasks extension point for defining related tasks. This extension point needs to be used when you want to display a new Related task on the Related tasks view.

Each related task is associated with a group. You can also define multiple categories for each related task. This extension point is defined in the com.yantra.yfc.rcp plug-in. Any plug-in that is dependent on the com.yantra.yfc.rcp plug-in can extend this extension point to define its own related tasks. The YRCRelatedTasks extension has an extension element called tasks. The tasks extension element also has an extension element called task.

Prior to implementing this extension point the following information is required:

- Category Information—When a new related task is being associated to the active task running in the current editor, you need to know the categories which the active task is interested in, so that the new related task can be shown on the Related tasks view. Once you have identified the category id to which the new related task should belong to, you must define the same category definition using the YRCRelatedTaskCategories extension point. Also, make sure that the new related task is defined in the previously mentioned category.

  Note: You can only add new related task to an existing category.

  You must define this category in your plugin.xml file, and make sure that the new related task is defined under this category.
- Group Information—To display a new related task, you can either use an existing group or create a new group.

To get the category and the group information for adding the new related task to a existing task:

Procedure

1. In the Rich Client Platform application, navigate to the task you want to extend.

© Copyright IBM Corp. 1999, 2013

2. Through the Rich Client Platform Extensibility Tool, view the screen information. The category and group information is displayed in the screen information window.
3. Start the Eclipse SDK.
4. In the navigator view, expand the plug-in project that you created.
5. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
6. Select the Extensions tab.
7. Click **Add**.
8. From the New Extension window, select **com.yantra.yfc.rcp.YRCRelatedTasks** extension point from the list.
9. Click **Finish**. The com.yantra.yfc.rcp.YRCRelatedTasks extension point gets created. The tasks, categories, and permissions extension elements gets created automatically in a tree structure.
10. Select the **com.yantra.yfc.rcp.YRCRelatedTasks** extension point. The Extension Details panel displays.
11. In the Extension Details panel, enter the properties of YRCRelatedTasks extension point.
12. Select the tasks extension element. The Extension Element Details panel displays.
13. In the Extension Element Details panel, enter the properties of the tasks extension element.
14. Expand the tasks extension element and select the task extension element. The Extension Element Details panel displays.
15. In the Extension Element Details panel, enter the properties of the task extension element. To create a new task extension element, right-click on tasks extension you created and select **New** > **task**. The task extension element gets created. You can create multiple task elements under the tasks extension element.
16. In groupId, enter the identifier of the group to which the related task belongs to. The groups are defined by extending the YRCRelatedTaskGroups extension point.
17. In actionId, enter the identifier of the action that gets invoked when you click on the related task. The action class of the actionId that you specified should extend the YRCRelatedTaskAction class.
18. In permissionId, enter the resource identifier from theSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales Resources that provide implementation for checking permissions to perform the related task. For example, a customer support representative may not have permissions to change the price of an item. The resource identifier for a given resource is defined in the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales Configuration Guide Application Platform Configuration Guide
19. Set the isExtension property to true if you want to mark the related task as a extended related task.

    Note: Whenever you set the value of the isExtension property to `true` for a related task, it indicates that you want to open the extended related task in an

existing editor. Therefore, you must define an extension contributor to open the extended related task in an existing editor.
20. Set the filterRequired property to `true`, if you want to filter related tasks based on custom criteria. For example, the Cancel Order related task should not be displayed after you ship an order.
21. Select the categories extension element. The Extension Element Details panel displays.
22. In the Extension Element Details panel, enter the properties of the categories extension element.
23. Expand the categories extension element and select the category extension element. The Extension Element Details panel displays.
24. In the Extension Details panel, enter the properties of the category extension element.
25. In id, enter the identifier of the category to which the related task belongs to. These categories are defined by extending the YRCRelatedTaskCategories extension point.
26. Select the permissions extension element. The Extension Element Details panel displays.
27. In the Extension Element Details panel, enter the properties of the permissions extension element.
28. Expand the permissions extension element and select the permission extension element. The Extension Element Details panel displays.
29. In the Extension Element Details panel, enter the properties of the permission extension element. To create a new permission extension element, right-click the permissions extension element that you created and select **New** > **permissions**. The permission extension element gets created. Just as the Rich Client Platform supports the definition of multiple permissions for each related task, you can define multiple permission extension elements under the category extension element.
30. In permissionId, enter the unique resource identifier that you defined for this resource in the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales Configuration Guide Application Platform Configuration Guide
31. In applicationid, enter the unique identifier of the RCP application.

Extending the YRCRelatedTaskCategories Extension Point

About this task

The Rich Client Platform provides the YRCRelatedTaskCategories extension point for defining categories, which can contain multiple related tasks from multiple groups. This extension point is defined in the com.yantra.yfc.rcp plug-in. Any plug-in that is dependent on the com.yantra.yfc.rcp plug-in can extend this extension point to define its own categories for the related tasks. The YRCRelatedTaskCategories extension has an extension element called categories. The categories extension element also has an extension element called category.

To extend the YRCRelatedTaskCategories extension point:

Procedure

1. Start the Eclipse SDK.

2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **New** > **tasks**.
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **com.yantra.yfc.rcp.YRCRelatedTaskCategories** extension point from the list.
7. Click **Finish**. The com.yantra.yfc.rcp.YRCRelatedTaskCategories extension point gets created. The categories and tasks extension elements gets created automatically in a tree structure.
8. Select the **com.yantra.yfc.rcp.YRCRelatedTaskCategories** extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of the YRCRelatedTaskCategories extension point.
10. Select the categories extension element. The Extension Element Details panel displays.
11. In the Extension Element Details panel, enter the properties of the categories extension element.
12. Expand the categories extension element and select the category extension element. The Extension Element Details panel displays.
13. In the Extension Element Details panel, enter the properties of the category extension element.
14. Expand the category extension element and select the tasks extension element. The Extension Element Details panel displays.
15. In the Extension Element Details panel, enter the properties of the tasks extension element.
16. Expand the tasks extension element and select the task extension element. The Extension Element Details panel displays.
17. In the Extension Details panel, enter the properties of the task extension element.
18. In id, enter the id of the related task that you want to have in this particular category.

Extending the YRCRelatedTaskGroups Extension Point

About this task

The Rich Client Platform provides the YRCRelatedTaskGroups extension point for defining group for a set of related tasks. This extension point is defined in the com.yantra.yfc.rcp plug-in. Any plug-in that is dependent on the com.yantra.yfc.rcp plug-in can extend this extension point to define its own groups for the related tasks. The YRCRelatedTaskGroups extension has an extension element called groups. The groups extension element also has a extension element called group.

To extend the YRCRelatedTaskGroups extension point:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **com.yantra.yfc.rcp.YRCRelatedTaskGroups** extension point from the list.
7. Click **Finish**.
8. Select the **com.yantra.yfc.rcp.YRCRelatedTaskGroups** extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of the YRCRelatedTaskGroups extension point.
10. Select the groups extension element. The Extension Element Details panel displays.
11. In the Extension Element Details panel, enter the properties of the groups extension element.
12. Expand the groups extension element and select the group extension element. The Extension Element Details panel displays.
13. In the Extension Details panel, enter the properties of the group extension element.
14. In sequence, enter the number to indicate that the groups should display in the ascending order of the sequence number in the Related Tasks view. The groups are displayed in the ascending order of their sequence number.

Extending the YRCRelatedTasksDisplayer Extension Point

About this task

The Rich Client Platform provides the YRCRelatedTasksDisplayer extension point for specifying the class that implements the com.yantra.yfc.rcp.IYRCRelatedTasksDisplayer interface.

This extension point is used to display the Related tasks view, implement this extension point only if you need to override the way the current view is displayed.

This extension point is defined in the com.yantra.yfc.rcp plug-in. Any plug-in that is dependent on the com.yantra.yfc.rcp plug-in can extend this extension point to provide its own implementation. The YRCRelatedTasks element has an extension element called relatedTasksDisplayer.

To extend the YRCRelatedTasksDisplayer extension point:

Procedure

1. Start the Eclipse SDK.

2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click Add. From the New Extension window, select **com.yantra.yfc.rcp.YRCRelatedTasksDisplayer** extension point from the list.
6. Click **Finish**.
7. Select the **com.yantra.yfc.rcp.YRCRelatedTasksDisplayer** extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of the YRCRelatedTasksDisplayer extension point.
9. Select the **relatedTasksDisplayer** extension element. The Extension Element Details panel displays.
10. To specify the implementation class, do any of the following
    - Click **Browse**. The Select Type pop-up window displays. Select the class that implements the com.yantra.yfc.rcp.IYRCRelatedTasksDisplayer interface. The specified class must return the list of all the related tasks that you want to display in a panel as ArrayList.
    - Click on the **class*** hyperlink. The Java Attribute Editor window displays.
      a. Enter the name of the class that implements the com.yantra.yfc.rcp.IYRCRelatedTasksDisplayer interface.
      b. Click **Finish**. The new class is automatically created.

Access Editor Information

About this task

Before implementing the YRCRelatedTasksExtensionContributor extension point the following information is required:

- Editor Information—To open the newly created related task within an application shipped editor, you need to know the identifier of that particular editor.

To access the editor information:

Procedure

1. Navigate to the task you want to extend in the Rich Client Platform application.
2. View the screen information through the Rich Client Platform Extensibility Tool. The editor information is displayed in the screen information window.

Extending the YRCRelatedTasksExtensionContributor Extension Point

About this task

The Rich Client Platform provides the YRCRelatedTasksExtensionContributor extension point for specifying the class that implements the com.yantra.yfc.rcp.IYRCRelatedTasksExtensionContributor interface. Use this

extension point only when you want to open a newly created related task within an application shipped editor. This extension contributor is called when a extended related task needs to be invoked in an application shipped editor.

This extension point is defined in the com.yantra.yfc.rcp plug-in. Any plug-in that is dependent on the com.yantra.yfc.rcp plug-in can extend this extension point to provide its own implementation for extended related tasks. The YRCRelatedTasksExtensionContributor element has an extension element called relatedTasksExtensionContributor.

To extend the YRCRelatedTasksExtensionContributor extension point:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **com.yantra.yfc.rcp.YRCRelatedTasksExtensionContributor** extension point from the list.
7. Click **Finish**.
8. Select the **com.yantra.yfc.rcp.YRCRelatedTasksExtensionContributor** extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of the YRCRelatedTasksExtensionContributor extension point.
10. Select the **relatedTasksExtensionContributor extension** element. The Extension Element Details panel displays.

    Make sure that for each relatedTasksExtensionContributor extension element you specify a unique editor. Otherwise, the system randomly selects an extension contributor from the specified extension contributors.
11. In editorId, specify the Id of the editor in which the extensible related tasks need to be opened. You can use the Rich Client Platform-provided editor or your own custom editor to open the related task. For one relatedTasksExtensionContributor elements, you can specify only one editor Id.
12. To specify the implementation class, do any of the following
    - Click **Browse**. The Select Type pop-up window displays. Select the class that implements the com.yantra.yfc.rcp.IYRCRelatedTasksExtensionContributor interface.
    - Click on the **class*** hyperlink. The Java Attribute Editor window displays.
      a. Enter the name of the class, that implements the com.yantra.yfc.rcp.IYRCRelatedTasksExtensionContributor interface.
      b. Click **Finish**. The new class is automatically created.
13. Override the createPartControl(Composite parent, YRCEditorInput editorInput, YRCRelatedTask currentTask) and return the panel to open the current editor. For example,

```
public Composite createPartControl(Composite parent,
YRCEditorInput editorInput, YRCRelatedTask currentTask) {
YCDAlertScreen NewScreen  = new YCDAlertScreen(parent, SWT.NONE);
      return NewScreen;
}
```

Enabling Custom Dialog Boxes Through an Extension Point for Rich Client Platform Applications

About this task

The Rich Client Platform provides four types of dialog boxes, namely, Error, Warning, Information and Confirmation, which have default and standard theme (font, size, background color and foreground color) settings. To use different settings for dialog boxes, an application can create its own custom dialog boxes with suitable themes and/or other parameters, as required.

To enable an application to create its own custom dialog boxes, an extension point YRCMessageDialog and an interface IYRCMessageDialog to implement the class are added to com.yantra.yfc.rcp plug-in.

An application can extend all or any of the message dialog boxes using the interface. If the extension is not specified, default settings are applied to the dialog boxes.

Note: The application must handle the entire creation and rendering of custom dialog boxes. However, required information for a dialog box such as a suitable title and message are provided by Rich Client Platform.

To create an extension:

Procedure

1. Use the extension point YRCMessageDialog for implementation.
2. Select this extension point and provide the following details in the Extension Elements Detail panel as explained subsequently:
   - Each extension element consists of one or more Message Dialog elements, each with a mandatory attribute, ModuleID. The ModuleID must correspond to the application's ModuleID to identify the application, for which, the dialog box changes are required.
   - Each Message Dialog element may contain one or more Dialog elements. For each Dialog element, provide the following mandatory attributes:
     - **type** - Indicates the type of dialog box to be extended [Error, Warning, Information, Confirmation or API Error].
     - classToLoad - Specifies the class to be loaded for implementing the interface IYRCMessageDialog.
     - The IYRCMessageDialog interface is defined in the following format
       ```
       public interface IYRCMessageDialog {
             Object show(Object ... objects);
       }
       ```

   Apart from the four dialog boxes, Applications can also extend dialog boxes used for displaying errors encountered during an API execution. The API error message displays error code and error description. Applications can extend this message box by using the error document provided by Rich Client Platform.

Chapter 13. Creating Commands

About Commands

You can create commands to call APIs or services to retrieve data in the required format. To create commands at the form level, use the *Plug-in id*_commands.ycml command file. Each form is a self-contained panel in itself. A self-contained panel has its own behavior class that extends the YRCBehavior class. Therefore, you must specify the identifier of the form in the Id attribute of the form element.

Note: In case of wizards, although wizard page has its own behavior, it is not a self-contained panel. This is because the wizard page behavior is internally dependent on the wizard behavior. Therefore, to create commands for a wizard page, you must create commands at the wizard level. You must specify the identifier of the wizard in the Id attribute of the form element.

The various attributes of command element are:

Field	Description
Name	Specify a unique name for the command. Command names are unique across the system and redefining a command with the same name overrides an existing definition.
APIName	Specify the name of an API or service. Associate each command you create with an API or service name.
APIType	Specify the API type. The valid values are API and SERVICE.
outputNamespace	Specify the namespace of the output template. This namespace is defined in the namespaces element.
inputNamespace	Specify the namespace of the input XML model. **Note:** If a Rich Client Platform application does not set the input namespace programmatically when calling a command, the system will, by default, create the input namespace in the following manner: 1. The target XML model for the screen is retrieved. 2. The input namespace attributes are then retrieved from the target XML model. However, if the Rich Client Platform application sets the input namespace programmatically when calling a command, the input namespace specified is ignored. The latter is the more commonly used approach in Rich Client Platform applications.
URL	If you want to invoke your own API instead of Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales APIs or services, specify the URL path of the server in the URL attribute. This URL must contain the value of the Name attribute of the Config element from the *.ycfg file. The complete path of the URL is defined in *.ycfg file. For more information about defining server URL in the *.ycfg file, see the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales: *Installing the PlatformInstallation Guide*.

© Copyright IBM Corp. 1999, 2013

Field	Description
prototype	Set the value of prototype attribute equal to "true" to run the commands in prototype mode. In prototype mode, the application uses XMLs stored in the prototype folder on the client machine as an output of an API.
version	The Rich Client Platform also supports versioning of APIs to ensure backward compatibility. Specify the version of the API that you want to call in the version attribute.

The following code is from a typical *.ycml file that is used to create commands:

```
<forms>
   <form Id = "com.yantra.order.capture.ui.screens.OrderSearchandList">
       <commands>
            <command Name="getOrderDetails"
                    APIName="getOrderDetails"
                    APIType="API"
                    outputNamespace="OrderDetails"
                    inputNamespace=""
                    URL="LOCAL"
                    prototype="true"
                    version=""/>
             <command Name = "getOrderList"
                    APIName = "getOrderList"
                    APIType="API"
                    outputNamespace="OrderList"
                    inputNamespace=""
                    URL=""
                    prototype=""
                    version=""/>
       </commands>
  </form>
</forms>
```

Note: IBM recommends that you do not make changes to the configuration file shipped with Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales. Always create your own configuration file or use the default configuration file that gets created whenever you create a new Rich Client Platform plug-in.

Every plug-in must invoke the command files during plug-in initialization to register its own set of commands. For more information about registering a commands file and other plug-in files, see the *Registering a Plug-In* topic.

Defining Namespaces

Namespaces are defined to uniquely identify an XML model. Use the *Plug-in id*_commands.ycml file to define namespaces. You can define namespaces at the form level. Specify the unique identifier of the form in the Id attribute of the form element. The various attributes of namespace element are:

Field	Description
name	Specify a unique name for the namespace.

Field	Description
type	Specify the type of namespace, depending on whether the template is to be used as input or output. For example, "input" or "output". **Note:** If you are creating the namespaces for the wizard rules: • Specify type attribute as "input" if you want to take inputs from the user as the target model of the screen. • Specify type attribute as "output" if you want to populate the controls with the values from an existing model.
templateName	Specify the name of the XML file that is to be picked from the server. For example, getOrderDetails. The system searches for this file on the server in the template/*Plug-in_id*/*form_id*/namespaces directory **Note:** Plug-in_id is the ID of the plug-in, which will register the ycml file.

The following code is from a typical *.ycml file that is used to define namespaces:

```
<forms>
   <form Id = "com.yantra.order.capture.ui.screens.OrderSearchandList">
      <commands>
           <command Name="getOrderDetails"
                  APIName="getOrderDetails"
                  APIType="API"
                  Namespace="OrderDetails"
                  URL=""
                  prototype=""
                  version=""/>
       </commands>
       <namespaces>
           <namespace name="OrderDetails"
                    type="output"
                    templateName="getOrderDetails"/>
            <namespace name="OrderList"
                    type="output"
                    templateName="getOrderList"/>
       </namespaces>
   </form>
</forms>
```

Overriding Commands

The Overriding Commands feature enables a user to call its custom API instead of APIs provided by the application for a particular form. To override a command, you must enter your own custom API name and use the same form Id and command name. For example, on OrderSearchandList form you want to call your own custom customGetOrderDetails API instead of the getOrderDetails API provided by the application.

The sample code from the *.ycml file to override commands:

```
<forms>
  <form Id = "com.yantra.order.capture.ui.screens.OrderSearchandList">
    <commands>
         <command Name="getOrderDetails"
                 APIName="customGetOrderDetails"
                 APIType="SERVICE"
                 outputNamespace="custOrderDetails"
                 inputNamespace="custOrderDetails"
                 URL="LOCAL"
                 prototype=""
```

```
                 version=""/>
      </commands>
   </form>
</forms>
```

Chapter 14. Defining and Overriding Hot Keys

Phase 1: Defining a Hot Key Command

About this task

The Rich Client Platform enables you to define new hot keys for new screens, and override the hot keys defined for the existing screens

To define a new command:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **org.eclipse.ui.commands** extension point from the list.
7. Click **Finish**.
8. Select the **org.eclipse.ui.commands** extension point. The Extension Details panel displays.
9. In the Extension Details panel, set the properties of the org.eclipse.ui.commands extension point.
10. Create the category extension element, if applicable. The category element is used to logically group a set of commands. To create a new category extension element, right-click on org.eclipse.ui.commands extension point and select **New** > **category**. The category extension element is created.
11. Select the category extension element. The Extension Element Details panel displays.
12. In id*, enter the unique identifier of the category.
13. In name*, enter the name of the category.
14. Create a new command extension element, right-click on **org.eclipse.ui.commands** extension point and select **New** > **command**. The command extension element is created.
15. Select the command extension element. The Extension Element Details panel displays.
16. In id*, enter the unique identifier of the command.
17. In name*, enter the name of the command.
18. In categoryId, enter the identifier of the category to which the command belongs, if applicable.
19. Click to save the changes.

© Copyright IBM Corp. 1999, 2013

Phase 2: Defining a Hot Key Binding

About this task

To define a new key binding for the category that you created in the org.eclipse.ui.commands extension point:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click **Add**. From the New Extension window, select **org.eclipse.ui.bindings** extension point from the list.
6. Click **Finish**.
7. Select the **org.eclipse.ui.bindings** extension point. The Extension Details panel displays.
8. In the Extension Details panel, set the properties of the org.eclipse.ui.bindings extension point.
9. Create a new key extension element, right-click on **org.eclipse.ui.bindings** extension point and select **New** > **key**. The key extension element is created.
10. Select the key extension element. The Extension Element Details panel displays.
11. In sequence*, enter a valid key sequence of the hot key for the command.
    - Use M1 to specify the **Ctrl** key
    - Use M2 to specify the **Shift** key
    - Use M3 to specify the **Alt** key
    - Use Esc to specify the **Escape** key

    To specify a combination of keys use the " + " operator. For example, to specify the hot key for a control as **Ctrl+Alt+K**, enter the key sequence as `M1+M3+K`.
12. In schemeId*, enter `defaultYantraKeyConfigurations`.
13. Set the context for the hot key either as local or global. In a local context, you can use the hot key for a specific screen in the application. In a global context, you can use the hot key for any screen in the application.
    - If you want to set the context of the hot key as local, in contextId, enter the identifier of the form used to identify the screen.

      To retrieve information for a specific screen, in a Rich Client Platform application, navigate to the screen for which you are defining the new hot keys. Using the Rich Client Platform Extensibility Tool, you can view the screen information.
    - If you want to set the context of the hot key as global, in contextId, enter the global context identifier of the Rich Client Platform. This context identifier is defined in the plugin.xml file of Rich Client Platform plug-in, for example, com.yantra.rcp.contexts.global.

14. In commandId, enter the identifier of the command that you defined.
15. Click [save icon] to save the changes.

Phase 3: Defining a Hot Key Action

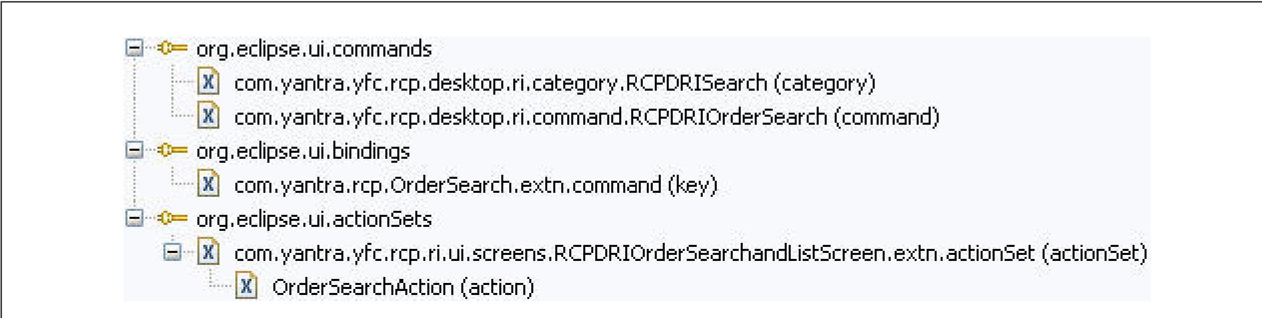
About this task

After defining the command and hot key binding, define the action to invoke when you press the hot key.

For more information about defining or creating new actions, see the *Creating New Actions* topic.

Note: In the definitionId field, enter the identifier of the hot key command that you created in Phase1.

After defining the command, key binding, and action, the structure of the plugin.xml file of the plug-in project is shown in the following figure.



Overriding Hot Keys

About this task

You can override the hot key bindings defined for existing screens. To override an existing hot key, you need to know the identifier of the command.

To override a hot key:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **Open With** > **Plug-in Manifest Editor**
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **org.eclipse.ui.bindings** extension point from the list.

7. Click **Finish**.
8. Select the org.eclipse.ui.bindings extension point. The Extension Details panel displays.
9. In the Extension Details panel, set the properties of the org.eclipse.ui.bindings extension point.
10. Create a new key extension element, right-click on org.eclipse.ui.bindings extension point, and select **New** > **category**. The key extension element is created.
11. Select the key extension element. The Extension Element Details panel displays.
12. In sequence*, enter the new valid key sequence of the hot key that you want to override.
    - Use M1 to specify the **Ctrl** key
    - Use M2 to specify the **Shift** key
    - Use M3 to specify the **Alt** key
    - Use Esc to specify the **Escape** key

    To specify a combination of keys use the " + " operator. For example, to specify the hot key for a control as **Ctrl+Alt+K**, enter the key sequence as `M1+M3+K`.
13. In schemeId*, enter `defaultYantraKeyConfigurations.`
14. Set the context for the hot key. You can either set the context as local or global. Local context means that the hot key can be used only for a particular screen in the application. Global context means that the hot key can be used for any screen in the application.
    - If you want to set the context of the hot key as local—In contextId, enter the identifier of the form that is used to identify the screen. To get the information about a particular screen:
    - In the Rich Client Platform application, navigate to the screen for which you are defining the new hot keys.
    - Through the Rich Client Platform Extensibility Tool view the screen information.
    - If you want to set the context of the hot key as global—In contextId, enter the global context identifier of the Rich Client Platform. This context identifier is defined in the `plugin.xml` file of the Rich Client Platform plug-in. For example, com.yantra.rcp.contexts.global.
15. In commandId, enter the identifier of the command whose hot key you want to override.
16. Click to save the changes.

Disabling Related Task Hot Keys

About this task

By default, the hot keys defined for related tasks are always enabled. You can globally disable hot keys defined for the related tasks in a Rich Client Platform application.

To disable the related task hot keys, call the enableRelatedTasksHotKeys() utility method of the YRCAppShellConfiguration class and pass "false" as the input argument. For example,

```
YRCAppShellConfiguration.enableRelatedTasksHotKeys(false);
```

Note: You can disable the hot key for a particular related task by changing the hot key configurations using the Rich Client Platform Extensibility Tool.

Chapter 15. Merging Templates

Merging Input and Output Templates

About this task

The Rich Client Platform allows you to merge the input and output templates as per your needs. Template merging can be used to get additional data from an API or Service. For example, you may want to get the values of additional attributes from an API or service. All the templates that are shipped withSterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales are stored on the server in the namespaces folder of the Rich Client Platform plug-in and PCA plug-in directories. For getting the values of additional attributes, you may have to first extend the Application database by creating a new column and then updating the template.

The PCA templates are located at *INSTALL_DIR*/repository/xapi/template/merged/*PCA_plug-in_id*/*form_id*/namespaces

To extend the PCA templates, place the extended templates for the PCA in *INSTALL_DIR*/extensions/global/template/*plug-in-id*/*form_id*/namespaces

The Rich Client Platform templates are located at: *INSTALL_DIR*/repository/xapi/template/merged/com.yantra.yfc.rcp/namespaces

Note: You cannot extend the Rich Client Platform templates.

For APIs or services for which no form identifier is specified, the output templates are stored in the following folder of the Rich Client Platform plug-in: *INSTALL_DIR*/repository/xapi/template/merged/template/*Plug-in_id*/namespaces

You can create new output templates and store them in the following folder of the Rich Client Platform plug-in:

```
<INSTALL_DIR>/extensions/global/template/<plug-in-id>/<form_id>/namespaces
```

As an example, the following is a getOrderLineDetails output template:

```
<OrderLine>
   <OrderLineList>
        <Order OrderNo="Y00102495" ItemID="MOUSE"/>
   </OrderLineList>
</OrderLine>
```

To add a new attribute called Status to the OrderNo element in this output template:

Procedure

1. Create the XML file with the same name as the existing output template and store it in the /extensions/global/template/*Plug-in_id*/*form_id*/namespaces folder of the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales PCA plug-in. For example, getOrderLineDetails.xml.

© Copyright IBM Corp. 1999, 2013

2. Add a new attribute called Status in the Order element. Add only the additional attributes that you require. The new output template looks as follows:

```
<OrderLine>
   <OrderLineList>
       <Order Status=" "/>
   </OrderLineList> <
/OrderLine>
```

The new getOrderLineDetails output template looks as follows:

```
<OrderLine>
  <OrderLineList>
     <Order OrderNo="Y00102495" ItemID="MOUSE" Status=" "/>
  </OrderLineList>
</OrderLine>
```

Chapter 16. Related and Shared Tasks

Adding New Related Tasks

You can add new related tasks to the Rich Client Platform application by extending the following extension points provided by the Rich Client Platform.

- YRCRelatedTasks
- YRCRelatedTaskCategories
- YRCRelatedTaskGroups
- YRCRelatedTasksDisplayer
- YRCRelatedTasksExtensionContributor

For more information about creating related tasks, see "Creating Related Tasks".

Hiding Existing Related Tasks

If you want to hide the related tasks panel on the screen, remove the related tasks from the list specified in the YRCRelatedTasksDisplayer extension point.

If you want to hide the individual related tasks from the related tasks panel, revoke the permissions for that particular related task resource from the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales Configuration Guide Application Platform Configuration Guide

Registering Shared Tasks

About this task

You can register the new shared tasks with the Rich Client Platform plug-in using the YRCSharedTasks extension point.

To register the new shared tasks:

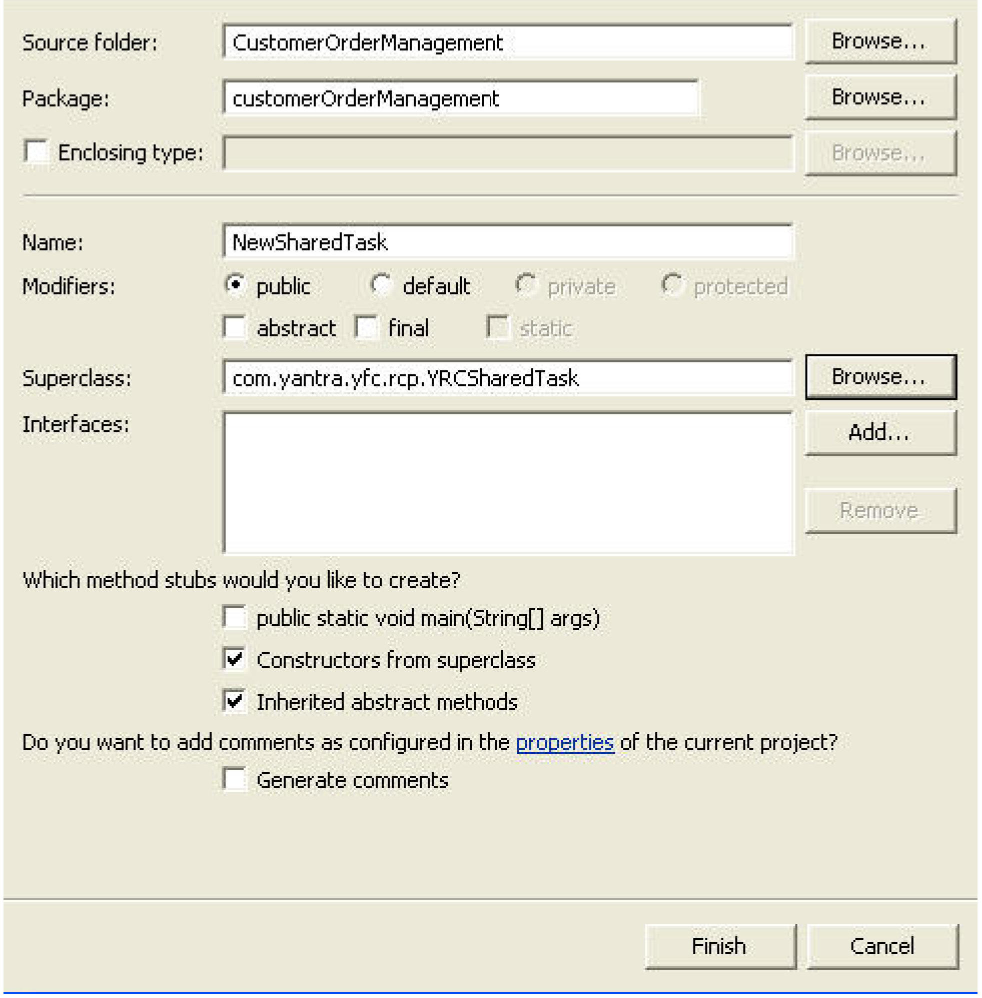
Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file, and select **Open With** > **Plug-in Manifest Editor**.
4. Click the Extensions tab.
5. Click Add. From the New Extension window, select **com.yantra.yfc.rcp.YRCSharedTasks** extension point from the list.
6. Click **Finish**.
7. Select the **com.yantra.yfc.rcp.YRCSharedTasks** extension point. The Extension Details panel displays.

© Copyright IBM Corp. 1999, 2013

8. In the Extension Details panel, enter the properties of the YRCSharedTasks extension point.
9. In id*, enter the unique identifier for the shared task. This shared task identifier should be unique across all the applications and plug-ins.
10. In name*, enter the name for the shared task.
11. In description*, enter the description of the shared task.
12. In class*, specify the implementation class for the shared task.

    To specify the implementation class, do any of the following:
    - Click **Browse**. The Select Type pop-up window displays. Select the class to use to extend the YRCSharedTask class.
    - Click the **class*** hyperlink. The Java Attribute Editor window displays.



Field	Description
Source folder:	The name of the source folder that you selected to store the shared task class automatically displays. Click Browse to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the shared task class automatically displays. Click **Browse** to browse to the package where you want to store the shared task class.
Name	Enter the name of the shared task class.

Field	Description
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter YRCSharedTask and click **OK**.
Constructors from superclass	Check this box. The system automatically creates the constructor for the YRCSharedTask superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the YRCSharedTask superclass.
Finish	When you click on this button, the system creates the new shared task class in the selected folder or package.

13. Open the newly created editor class in the Java Editor and implement the abstract methods of the YRCSharedTask class.

Using Shared Tasks

About this task

You can invoke a shared task by clicking a button, menu item, and so forth. You can also invoke a shared task by calling the launchSharedTask(String taskId, Element input) method provided by the YRCPlatformUI utility class of the Rich Client Platform.

To invoke a shared task within an application, you must know the complete details of the shared task, such the identifier of the shared task, structure of the input XML template, and structure of the output XML template.

To view the shared task information:

Procedure

1. Navigate to the Rich Client Platform application.
2. In the Rich Client Platform Extensibility Tool, view the shared task information.

   After getting the required information invoke the shared task by calling the launchSharedTask(String taskId, Element input) method. For example:

   ```
   YRCPlatformUI.launchSharedTask("com.yantra.rcp.SharedTask1",input);
   ```

   where com.yantra.rcp.SharedTask1 is the identifier of the shared task that you want to invoke and input is an input XML element that exist in the input XML to the shared task.

   The YRCPlatformUI class provides more methods, which you can call to invoke a particular shared task. For example, launchSharedTask(String taskId), launchSharedTask(Composite parent, String taskId), and so forth.

Chapter 17. Defining Themes

Defining New Themes

About this task

For theming the Rich Client Platform application, define the new theme entries in the *Plug-in_id_theme_name*.ythm theme file. After you register the theme file, it is loaded using the user-defined locale. For more information about registering the theme file and other plug-in files, see "Registering Plug-In Files" on page 193. The system loads all theme entries into a common repository and automatically applies them to the controls on the UI. The last theme definition that is loaded overrides the previous theme definitions.

To define the new theme entries for theming the Rich Client Platform application:

Procedure

1. Before you can start theming your Rich Client Platform application, you must set up the development environment. For more information about setting up the development environment, see "The Development Environment for Rich Client Platform Applications".
2. In the navigator window, expand the plug-in project that you created.
3. Open the *.ythm file in the text editor.
4. Create the root element Theme.
5. In the id attribute, specify the unique identifier for the theme.
6. Create ThemeEntry element under the Theme element.
7. In the Name attribute specify the unique name for this theme entry.
8. Create the Font element under ThemeEntry and set the its attributes. For Font element attribute list, see the following table.

Attribute	Description
Name	Specify the name of the font you want to use. For example, Tahoma, Courier, Arial, and so forth.
Height	Specify the height of the font.
Style	Specify the font style that you want to use. For example, NORMAL, BOLD, ITALIC, and so forth.

9. Create the BackgroundColor element under ThemeEntry and set the its attributes. For BackgroundColor element attribute list, see the following table.

Attribute	Description
Red	Specify the decimal color code for the red color. Valid values range from 0 to 255.
Green	Specify the decimal color code for the green color. Valid values range from 0 to 255.
Blue	Specify the decimal color code for the blue color. Valid values range from 0 to 255.

10. Create the ForegroundColor element under ThemeEntry and set the its attributes. Use the same attribute list as the BackgroundColor elements.

© Copyright IBM Corp. 1999, 2013

11. Create the Image element under the ThemeEntry element, if applicable.
12. In the Path attribute, specify the path of the image you want to display.

    Note: You can create multiple ThemeEntry elements to define themes for various resources such as control text, user info, error text, error icons, logos, and so forth.
13. Rename the *.ythm file to: *file_name*_theme_name.ythm. For example, comapp_jade.ythm.

    where comapp is the *file_name* and jade is the *theme_name*.
14. Register the theme file in the plugin java file of the plug-in project using the registerTheme() method. For example,

    ```
    YRCPlatformUI.registerTheme("<file_name>_<themename>", ID);
    ```

    The sample theme entries from the *.ythm file are as follows:

    ```
    <Theme id="jade">
        <ThemeEntry Name="Label">
            <Font Name="Tahoma" Height="9" Style="NORMAL"/>
            <ForegroundColor Red="0" Green="0" Blue="0"/>
            <BackgroundColor Red="245" Green="245" Blue="245"/>
       </ThemeEntry>
        <ThemeEntry Name="Text">
          <Font Name="Tahoma" Height="8" Style="NORMAL"/>
          <ForegroundColor Red="0" Green="0" Blue="0"/>
          <BackgroundColor Red="255" Green="255" Blue="255"/>
      </ThemeEntry>
        <ThemeEntry Name="Table">
          <Font Name="Tahoma" Height="8" Style="NORMAL"/>
          <BackgroundColor Red="245" Green="245" Blue="245"/>
          <ForegroundColor Red="0" Green="0" Blue="0"/>
      </ThemeEntry>
        <ThemeEntry Name="ErrorColor">
          <Font Name="Tahoma" Height="10" Style="BOLD"/>
          <ForegroundColor Red="255" Green="0" Blue="0"/>
          <BackgroundColor Red="245" Green="245" Blue="245"/>
      </ThemeEntry>
        <ThemeEntry Name="ErrorIcon">
          <Image Path="/icons/error.gif"/>
      </ThemeEntry>
        <ThemeEntry Name="HeaderLogo">
          <Image Path="/icons/yantra_header.jpg"/>
      </ThemeEntry>
    </Theme>
    ```

Defining Themes for Controls

About this task

For theming controls, define the theme entries in the *Plug-in_id_theme_name*.ythm file at the plug-in level. For example, let us consider that you have created a new label and you want to have a specific font and color for that label. To set a theme for the label:

Procedure

1. Define entries in the theme file for the label. For example:

   ```
   <Theme id="sapphire">
      <ThemeEntry Name="MyLabel">
           <Font Height="8" Name="Tahoma" Style="NORMAL"/>
   ```

```
        <ForegroundColor Blue="0" Green="0" Red="0"/>
        <BackgroundColor Blue="245" Green="245" Red="245"/>
    </ThemeEntry>
</Theme>
```

where id attribute is the unique identifier for the *Plug-in_id_theme_name*.ythm file. The Name attribute indicates the name of the theme entry, which is used for theming controls.

Note: The theme file corresponding to the theme specified within the user configuration is loaded. For example, if you log on to the Rich Client Platform application as user that is configured to use the theme with id as "sapphire", then the theme file with id "sapphire" gets loaded.

Therefore, if you are creating new screens and adding new entries for the "sapphire" theme, the Id attribute of this extension theme file should be "sapphire".

2. Set the binding data for the control by associating the binding object with the key. For example,

```
lblDate.setData(YRCConstants.YRC_CONTROL_CUSTOMTYPE,"MyLabel");
```

where lblDate is the reference variable name of the label, which you specified in the visual editor, YRCConstants.YRC_CONTROL_CUSTOMTYPE is the key used for identifying the custom theme entry, and MyLabel is the name of the ThemeEntry element in the theme file.

What to do next

Note: You cannot define theme for the following RCP UI controls. This is an eclipse SWT limitation:

- Button—You can only change the font (size , name , and style) for a button. The background and foreground color cannot be changed as it is OS specific.
- Combo Box—You can only change the foreground , background and font of the contents of a combo box. The combo box color cannot be changed as it is OS specific.
- Table Column—Theme cannot be applied for a table column.

Applying Themes to Non-editable Text Boxes

About this task

Labels do not support text edits and cannot display lengthy text. To overcome this problem, non-editable text boxes are used (without the border). Such non-editable text boxes do not have any theme set and are indistinguishable from labels.

To distinguish between labels and non-editable text boxes, a new theme NoneditableTextboxTheme as the ThemeEntry Name is included in the *.ythm files, by default. The Rich Client Platform applies this default theme to all non-editable text boxes that do not already have a theme.

To override the default settings:

Add a theme entry with the same name, NoneditableTextboxTheme, in the Application plugins, *.ycml file as follows:

```
<ThemeEntry Name="NoneditableTextboxTheme">
 <Font Height="8" Name="Tahoma" Style="NORMAL"/>
 <ForegroundColor Blue="0" Green="179" Red="0"/>
 <BackgroundColor Blue="255" Green="255" Red="255"/>
</ThemeEntry>
```

Note: This theme cannot be applied to text boxes that are made non-editable dynamically.

Chapter 18. Menus and Custom Controls

Adding and Removing Menus in Rich Client Platform Applications

Menu configuration contains the standard application resources and also the extended resources that you define when configuring resources.

All menus are grouped into a menu group. The default menu group contains the standard menu configuration of the Application Console, which is linked to the default Administrator user. When creating your own users, you can reuse this menu group or create a new menu group. The custom menus may contain different menu items.

For more information about adding or removing menus, see the Defining Menus topic in the Sterling Business CenterSterling Selling and Fulfillment FoundationSterling Field Sales Configuration Guide Application Platform Configuration Guide.

Customizing the Menu View Through the YRCMenuDisplayer Extension Point

About this task

The Rich Client Platform enables you to extend the menu view for specific modules in Rich Client Platform applications through an extension point, YRCMenuDisplayer. This extension point is provided in the com.yantra.yfc.rcp plugin. An interface IYRCMenuDisplayer is also provided, which must be implemented by the class specified in the extension.

To customize the menu view for specific modules that contain a menu view in Rich Client Platform applications, perform the following steps:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the **Extensions** tab.
5. Click **Add**.
6. From the New Extension window, select **YRCMenuDisplayer** extension point from the list.
7. Click **Finish**.
8. Select the **com.yantra.yfc.rcp.YRCMenuDisplayer** extension point. The Extension Details panel is displayed.
9. In the Extension Details panel, enter the properties of YRCMenuDisplayer extension.

© Copyright IBM Corp. 1999, 2013

10. The extension point has a defined sequence, which consists of the following attributes:
    - id: The extension is identified by a unique ID which must be specified.
    - name: This is the name given to the extension. The name is optional. For example, mymenu.
    - MenuDisplayer: The MenuDisplayer element defines the class to be loaded for customizing the menu view on the workbench window. This extension point consists of the following mandatory attributes:
    - class: Specify a fully qualified path to the Java class that must implement the IYRCMenuDisplayer interface.
    - moduleId: Specify the module ID of the application for which the menu view must be customized. For example, ycd (for Sterling Call Center and Sterling Store).

Creating Custom Controls for Rich Client Platform Applications

The Rich Client Platform enables you to create custom controls to handle various business requirements. For example, capturing date and time from a single control can be difficult as it is prone to errors and it is not intuitive. To solve this issue, you can model date and time as a single custom control.

Note: By default, the Rich Client Platform provides custom control implementation for Date and Time control. But Rich Client Platform allows you to create new custom controls based on your business requirement. For example, you may want to create a custom control with specific look and feel, control type, order of controls.

Extending the YRCCustomControl Extension Point

About this task

To extend the YRCCustomControl extension point:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select the **com.yantra.yfc.rcp.YRCCustomControl** extension point from the list.
7. Click **Finish**.
8. Select the **com.yantra.yfc.rcp.YRCCustomControl** extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of YRCCustomControl extension point.
10. In controlId, enter the unique identifier for the custom control. You can create different custom controls using a single extension point but different controlId.

11. In class, to specify the implementation class, do any of the following:
    - Click **Browse**. The Select Type pop-up window displays. Select the class that extends the org.eclipse.swt.widgets.Composite class.
    - Click on the class: hyperlink. The Java Attribute Editor window displays.

Field	Description
Source folder:	The name of the source folder that you selected to store the custom control class automatically displays. Click **Browse** to browse to the folder that you want to specify as the source folder.
Package:	The name of the package that you selected to store the custom control class automatically displays. Click **Browse** to browse to the package where you want to store the custom control class.
Name	Enter the name of the custom control class.
Superclass:	Click **Browse**, the Superclass Selection window displays. In Choose a type, enter Composite and click **OK**.
Interfaces:	Click **Add**, the Implemented Interfaces Selection window displays. In Choose a type, enter IYRCCustomControl and click **OK**.
Constructors from superclass	Check this box. The system automatically creates the constructor for the Composite superclass.
Inherited abstract methods	Check this box. The system automatically adds the abstract methods inherited by the Composite superclass.

12. Open the newly created custom control class in the Java Editor and implement the abstract methods of the Composite class.

Using Custom Controls in RCP Applications

To use a custom control you need to implement the IYRCCustomControl interface in your custom class. The IYRCCustomControl interface provides the ability to display or fetch data in any format as per your business requirement. For example, you may want to display date and time in AM or PM format. To display data in a custom control, you need to implement the following abstract methods of the IYRCCustomControl interface:

- setBehavior(YRCBehavior)—Pass the Parent behavior to this method to handle errors, refresh views, and toggle Editor dirty status. To handle error on tab out, use the addFieldInError() and removeFieldInError() methods. Also, whenever a text is entered in the custom control, you should mark editor dirty status, using the setDirty() method.

  Note: If you want to invoke the custom control after behavior creation, then you should set the behavior explicitly. Otherwise Rich Client Platform will set the behavior for the custom control. In case of extensibility, the Rich Client Platform automatically takes care of setting the behavior.
- setText(namespace, value)—Whenever the setModel() method is called, depending on the source binding and the namespace, the text is set on the controls. In case of a custom control, this method is called whenever the setModel() method is called for the given namespace. Implementor of this class should set the text on the custom control.
- setFocusOnErrorControl()—This method is used to set focus on Error control, if the custom control has some validation error.

  Whenever showError() method is called and the custom control is in error (i.e you have called the addFieldInError() method in behavior on some validation), this method is called by Rich Client Platform to set focus on the control in error.

- getText(namespace)—This method is called on the getTargetModel() method for each namespace defined in the target Binding. The value that you return should be a deformatted value.
- getBindingData()—Pass the binding object of the YRCCustomControlBindingData class that you created for the custom control.
- setInput(input)—Pass the custom control input binding object that you created for the custom control.

You can invoke a custom control by calling the getCustomControl(parentComposite, customControlId, CustomControlBindingData) method provided by the YRCUIUtils utility class of the Rich Client Platform.

Note: Rich Client Platform also provides an overrided method which does not take parent composite as argument. Using this method or passing a null parent will return the custom control with parent as a new shell. But in this case, you should call the setParent() method on the custom control.

For example:

```
IYRCCustomControl dateTimeCtrl = YRCUIUtils.getCustomControl("DateTime",
customCtrlBindingData);
```

where DateTime is the identifier of the custom control that you want to invoke and customCtrlBindingData is the custom control binding object.

The YRCUIUtils class also provides following additional methods, which you can call for a custom control:

- localizeControl(parent, formId, pluginId)—Used to handle localization for the text on all controls, hot key on links, and buttons on parent control and all its children.
- applyTheme(parent, formId)—Used to apply a particular theme on parent control and all its children.
- getCustomControl(controlId)—Returns a custom control which is an instance of IYRCCustomControl class

Chapter 19. Setting the Extension Model and Configuring SSL and SSO

Setting the Extension Model for Rich Client Platform Applications

Extension model is set to populate the newly added fields on the form with the required data. Extension model must be used in case you are not getting the required data from the existing model or existing APIs or services called on the screen.

Before you set the extension model, do the following:

- Creating Commands—In the *Plug-in_id*_commands.ycml file, create new commands for calling an API or service for a screen. For more information, about creating commands, see "Creating Commands".
- Defining Namespaces—In the *Plug-in_id*_commands.ycml file, define the new namespaces for a screen.

Note: All the new namespaces that you define must start with "Extn_".

After creating new commands and namespaces for a screen, call an API or service. After API or service call completes, call the setExtensionModel() method to populate the newly added fields on the screen. You must pass the namespace of the model and the target element as arguments to the setExtensionModel() method.

Note: Use the setExtensionModel() method only if the a specific API is not returning the required data for the newly added field and hence you want to call your own API.

Configuring SSL for Rich Client Platform Applications

About this task

The Rich Client Platform allows you to connect to servers using the HTTPS protocol.

You can add your own custom hostname verification logic by adding the hostname verifier. To add the hostname verifier, you must extend the YRCHostNameVerifier extension point.

To extend the YRCHostNameVerifier extension point:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.
3. To open the plugin.xml file in the Plug-in Manifest Editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.

© Copyright IBM Corp. 1999, 2013

5. Click **Add**.
6. From the New Extension window, select **com.yantra.yfc.rcp.YRCHostNameVerifier** extension point from the list.
7. Click **Finish**.
8. Select the **com.yantra.yfc.rcp.YRCHostNameVerifier** extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of the YRCHostNameVerifier extension point.
10. To create a new hostNameVerifier extension element, right-click on com.yantra.yfc.rcp.YRCHostNameVerifier extension and select **New** > **hostNameVerifier**. The hostNameVerifier extension element gets created.
11. Select the **hostNameVerifier** extension element. The Extension Element Details panel displays.
12. To specify the implementation class, do any of the following
    - Click **Browse**. The Select Type pop-up window displays. Select the class that implements the javax.net.ssl.HostnameVerifier interface.
    - Click the **class*** hyperlink. The Java Attribute Editor window displays.
    - Enter the name of the class that implements the javax.net.ssl.HostnameVerifier interface.
    - Click **Finish**. The new class gets created.
13. Implement the verify (String hostName, SSLSession session) method and return the value "true" if the host name is acceptable. Otherwise, return the value "false".

Configuring SSO for Rich Client Platform Applications

Single Sign-on (SSO) enables a user to perform an authentication once and gain access to the of multiple applications' resources without having to login to the applications again and again. To set up an SSO for a Rich Client Platform application, you need to configure both client-side and server side settings.

Note: SSO must be implemented as a separate plug-in. If not implemented as a separate plug-in, the out-of-the-box *key= value* bindings will be ignored.

Client Settings for SSO Configuration

About this task

Perform the following steps:

Procedure

1. Create a new plug-in for SSO authentication.
2. Start the Eclipse SDK.
3. In the navigator view, expand the plug-in project that you created.
4. To open the plugin.xml file in the Plug-in Manifest Editor, perform one of the following tasks:
    - Double-click the plugin.xml file.
    - Right-click the plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
5. Select the Extensions tab.
6. Click **Add**.

7. From the New Extension window, select **com.yantra.yfc.rcp.YRCSSOAuthenticator** extension point from the list.
8. Click **Finish**.
9. Select the com.yantra.yfc.rcp.YRCSSOAuthenticator extension point. The Extension Details panel is displayed.
10. In the Extension Details panel, enter the properties of the com.yantra.yfc.rcp.YRCSSOAuthenticator extension point.
11. To specify the implementation class, perform one of the following tasks:
    - Click **Browse**. In the Select Type pop-up window that is displayed, select the class that implements the com.yantra.yfc.rcp.YRCSSOAuthenticator interface.
    - Click the class* hyperlink. The Java Attribute Editor window is displayed.
    - – Enter the name of the class that implements the com.yantra.yfc.rcp.YRCSSOAuthenticator interface.
    - Click **Finish**. The new class is automatically created.
12. Implement the isAuthTokenAvailable() method. If the SSO Authentication is available in the Rich Client Platform Application, the isAuthTokenAvailable() method should return True. Otherwise, it should return False.
13. Implement the setAuthToken(URLConnection connection) method and return the *key* and *value* pairs of the connection request property. Following is an example of this:
    ```
    public void setAuthToken(URLConnection connection){
    connection.setRequestProperty(key,value);
    }
    ```
14. Override the getBrowserAuthParams() method and return the map of the connection request parameters. The map should contain the string objects as *key* and *value* pairs. Following is an example of this:
    ```
    public Map getBrowserAuthParams() {
    Map map = new HashMap();
    map.put(key, value);
    return map;
    }
    ```
15. Edit the Rich Client Platform application's *.ini file and add the following VM arguments:
    ```
    -vmargs
    -Dssomode=Y
    ```

Server Settings for SSO Configuration

Procedure

1. Open the *INSTALL_DIR*/repository/eardata/platform/descriptors/weblogic/WAR/WEB-INF/web.xml file and search for the *servlet-name* tag.
2. Inside the RcpSSOServlet *servlet-name* tag, add the following init parameter entry:
   ```
   <init-param>
        <param-name>rcpssomanager</param-name>
        <param-value>com.yantra.SsoManager</param-value>
   </init-param>
   ```

   Note: To use SSO, the client should be configured to SSO and should have the authentication token. The rcpssomanger init parameter set on the server is used to validate the user session.

Chapter 20. General Concepts Reference

Rich Client Platform Architecture

Rich Internet Clients have advantages of both Client-Server and thin-client applications. Rich Internet Client applications are developed on open standards and have strong integration with the Desktop Operating System (OS), resulting in rich interaction. Rich Internet Client applications provide immediate feedback to users when they interact with the application. Rich Internet Client applications use modern UI controls, such as tree controls or tabbed panels. Also, Rich Internet Client applications allow users to perform interactive operations such as drag and drop.

User Interfaces (UI) have been an integral part of any software application. For the last few decades, a wide range of architectures and technologies have been used to deliver user interfaces. The Total Cost of Ownership (TCO) and Usability, Responsiveness, and Performance (URP) have been the two balancing factors for choice of technologies.

TCO covers all the upfront and ongoing costs of an application, which includes: purchase price, equipment, installation, training, and ongoing maintenance.

URP measures the performance of an application, its usability, and user's productivity.

The ideal application would be one with a low TCO and a high URP.

UI architectures can be classified as:

- Green screen (or Character User Interfaces (CUI))
- Client-Server
- Browser based
- Rich Internet Client

The CUI provided users with basic user interfaces. CUI did not have the capability of displaying information such as product images due to lack of graphic capabilities. With the advent of Graphical User Interfaces (GUI) and operating systems such as Windows, applications can support more sophisticated user interfaces along with alternate input devices. For example, mouse.

The GUI applications developed using Client-Server technologies resulted in Dynamic Link Library (DLL) conflicts and heavy network usage with respect to TCO.

In mid-90's, internet technologies such as Hyper Text Markup Language (HTML) started emerging very fast. Due to its simplicity and very low TCO, internet technologies had tremendous impact in the way business applications were delivered. Initially, HTML was only used for displaying information. However, its potential for applications was soon exploited.

HTMLs performance in an interactive mode is highly limited, with high number of trips required back and forth from the server. In addition, standard HTML was

© Copyright IBM Corp. 1999, 2013

never intended to produce the high-quality and high-performance user interfaces that excelled under the Client-Server model.

The reduction in TCO of these browser-based applications was at the expense of the users, as the URP was severely reduced.

The following figure illustrates the Rich Client Platform Architecture.

Figure 3. Rich Client Platform Architecture

Today, technologies exists to create what is commonly known as Rich Internet Clients that have advantages of a Client-Server model in terms of URP and thin-client applications in terms of TCO. Rich Internet Client applications are developed on open standards and have strong integration with the desktop Operating System (OS), which results in highly rich interaction. Rich Internet Client applications are also designed to provide server-based updates and are designed to work with low bandwidth networks and standard security protocols.

Eclipse and its Rich Client Platform

Eclipse is an open source software development environment dedicated to provide a robust, full-featured, and commercial-quality industry platform for the development of highly integrated tools. The Eclipse Platform is designed for building Integrated Development Environments (IDEs) that are used to create diverse applications.

Eclipse Rich Client Platform helps in building java applications that are Application Platform independent. Eclipse Rich Client Platform can also be used for building non-IDE applications. Eclipse Rich Client Platform provides a general UI, which can be extended by developers to suit their business needs.

www.eclipse.org is an association of software development tool vendors. The Eclipse community was formed in order to create better development environments and product integration. The community shares an interest in creating products that are easily interoperable, because they are based on plug-in technology and a common platform.

The Eclipse platform, which is a part of the Eclipse project, is an open extensible Integrated Development Environment (IDE). The Eclipse platform provides building blocks and a foundation for constructing and running integrated software development tools. Primarily, Eclipse platform is driven by International Business Machines (IBM). Eclipse technology is widely accepted within the Java community.

The Eclipse Rich Client Platform addresses the need for a single cross-platform environment to create highly-interactive business applications. Essentially, Rich Client Platform provides a generic Eclipse workbench that developers can extend to construct their own applications. Eclipse Rich Client Platform is a part of the Eclipse 3.23.3release. Eclipse Rich Client Platform enables application developers to deliver rich internet applications that run on platforms such as Windows, Linux, and so forth.

Workbench

Workbench refers to the desktop development environment. Workbench window contains one or more perspectives. A perspective defines the initial set and layout of views in the Workbench window. Perspectives contain views, editors, menus, and tool bars. You can customize a perspective by defining a set of actions. More than one Workbench window can exist on the desktop at any given time.

Plug-In Manifest Editor

The Plug-in Manifest Editor provides a single UI for editing the manifest and other plug-in rel.ted files. The Plug-in Manifest Editor contains following sections.

Overview

The Overview section provides plug-in details such as plug-in identifier, version, and so forth. It also specifies the class that is called when the user runs a plug-in.

Dependencies

The Dependencies section provides a list of dependant plugins required by the plug-in to compile its code. If a plug-in is using the extension points of some other plugins, then the plug-in must list those plugins as dependant plugins.

Runtime

The Runtime section provides a list of libraries in which the plug-in code is packaged. For example, sop.jar. The class loader searches these libraries during runtime to load the plug-in's classes. You can set the library's type, visibility, and content in the runtime section.

Extensions

The Extensions section describes the functionality that a plug-in contributes to the Eclipse platform by extending other plugins extension points. The extension declaration must adhere to the schema defined by the extension point it extends. You can add new menus and menu items along with toolbar by extending the org.eclipse.ui.actionSets extension point.

Extension Points

The Extension Points section provides a list of new extension points that are defined by a plug-in, which can be extended by other plugins to add the new functionality. For example, the Rich Client Platform plug-in provides a YRCPluginAutoLoader extension point which other plugins can extend to load their plug-in.

Build

The Build section provides a list of libraries that are required at the runtime. It also lists the source folder where these libraries are located. You can select the folders and/or files you want to include in the source build and binary build.

Manifest.mf

The manifest.mf file contains a list of plugins that are loaded dynamically. The Bundle-Activator entry specifies the name of the plug-in. For example, com.yantra.yfc.rcp.

Plugin.xml

The plugin.xml file contains all information that is required to run a plug-in. The plugin.xml file is used for defining Eclipse extension points, and other dependent plug-in's extension points. However, if you are not using any extension points, you can exclude this file.

Build.properties

The build.properties file contains all files and directories that are required by a plug-in at the runtime.

YRCPluginAutoLoader Extension Point

The Rich Client Platform provides YRCPluginAutoLoader extension point, which defines the order in which the plugins needs to be loaded. The YRCPluginAutoLoader is an extension point, which is defined in the com.yantra.yfc.rcp plug-in. Any plug-in that is dependent on the com.yantra.yfc.rcp plug-in can extend this extension point to automatically load a class in the specified order when starting the Rich Client Platform application. The YRCPluginAutoLoader automatically loads the classes within a plug-in during startup in a specified order. All classes that need to be automatically loaded are sorted in ascending order and loaded one at a time. The YRCPluginAutoLoader has a extension element called AutoLoad, which has two properties ClassToLoad and LoadOrder.

Note: Loading a class within a plug-in may load the plug-in itself, resulting in initialization of the class used for registering plug-in and other resource files. Therefore, the YRCPluginAutoLoader extension point is used for initialization purposes.

YRCAutoUpdateExtn Extension Point

The Rich Client Platform provides an extension point, YRCUpdateExtn and an interface IYRCClientUpdater, which can be used to prevent users from logging on to the system when application updates are being automatically installed.

The interface IYRCClientUpdater contains downloadingUpdate and downloadComplete methods, which must be implemented to inform users about the updates being installed and when they are complete.

The method downloadingUpdate must be called after the Updatecheck method and returns one of the following:

- DO_NOT_UPDATE—Skips the update.
- BLOCK_UI—Downloads the update and prevents user from logging on.
- CONTINUE—Continues to download the update but allows users to log in.

The method downloadComplete must be called after the method DownloadingUpdate and returns one of the follwing:

- INSTALL_EXIT—Installs updates and closes the application.
- INSTALL_RESTART—Installs updates and restarts the application. The login credentials must be passed again when the application is restarted.
- CONTINUE—Displays Update Downloaded message and installs updates when user restarts the application.

YRCApplicationInitializer Extension Point

About this task

The Rich Client Platform provides an extension point, YRCApplicationInitializer, and an interface, IYRCApplicationInitializer, which can be used to define the classes that are initialized and invoked during application startup. These classes are called after login but before the application workbench window is created or opened.

The YRCApplicationInitializer extension point is defined in the com.yantra.yfc.rcp plug-in and must implement the IYRCApplicationInitializer interface.

To define the initialization class, perform the following steps:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**
4. Select the **Extensions** tab.
5. Click **Add**.
6. From the New Extension window, select **YRCApplicationInitializer** extension point from the list.
7. Click **Finish**.
8. Select the com.yantra.yfc.rcp.YRCApplicationInitializer extension point. The **Extension Details** panel is displayed.
9. In the **Extension Details** panel, enter the properties of YRCApplicationInitializer extension.
10. The extension point has a defined sequence, which consists of the following attributes:
    - id—The extension is identified by a unique ID which must be specified.
    - name—This is the name given to the extension. The name is optional. For example, myappinitializer.
    - Initializer—The Initializer element defines the initializer class to be loaded before the workbench window is created or opened. This consists of the following mandatory attribute that must be defined:
      – class—Specify a fully classified path to a Java class that must implement the IYRCApplicationInitializer interface.

YRCContainerToolbar Extension Point

About this task

YRCContainerToolbar extension point is a resource provider extension point for PCAs, provided by the Rich Client Platform in the com.yantra.yfc.rcp plugin. An interface IYRCContainerToolbarProvider is provided, which must be implemented by the class specified in the extension.

The YRCContainerToolbar extension point can be used to display a toolbar or customize the existing toolbar on the application workbench window.

By default, the application container layout consists of the following elements:
- Container Header
- Container Toolbar
- Related Task/menu
- Main Editor

You can use this extension point to customize the toolbar on the container layout.

To display or customize the toolbar, perform the following steps:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the **Extensions** tab.
5. Click **Add**. From the New Extension window, select **YRCContainerToolbar** extension point from the list.
6. Click **Finish**.
7. Select the com.yantra.yfc.rcp.YRCContainerToolbar extension point. The **Extension Details** panel is displayed.
8. In the **Extension Details** panel, enter the properties of YRCContainerToolbar extension.
9. The extension point has a defined sequence, which consists of the following attributes:
   - id—The extension is identified by a unique ID which must be specified.
   - name—This is the name given to the extension point. The name is optional. For example, mycontainertitle.
   - ApplicationToolbarProvider—This element defines the class to be loaded before the workbench window is created or opened. This consists of the following mandatory attributes:
     - moduleId—Specify the module ID of the PCA for which you want to display or customize the toolbar. For example, ycd (forSterling Call Center and Sterling Store application).
     - class—Specify fully classified path to a Java class that must implement the interface IYRCContainerToolbarProvider interface.

YRCPostWindowOpenInitializer Extension Point

About this task

The YRCPostWindowOpenInitializer extension point is provided in the com.yantra.yfc.rcp plug-in for initialization operations. The extension point can be used to open the required editors and menus after the application workbench window is open. The YRCPostWindowOpenInitializer extension point can also be used to display or hide views.

An interface IYRCPostWindowOpenInitializer is provided, which must be implemented by the class specified in the extension.

To create an extension for YRCPostWindowOpenInitializer, perform the following steps:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the **Extensions** tab.
5. Click **Add**.
6. From the New Extension window, select **YRCPostWindowOpenInitializer** extension point from the list.
7. Click **Finish**.
8. Select the com.yantra.yfc.rcp.YRCPostWindowOpenInitializer extension point. The **Extension Details** panel is displayed.
9. In the **Extension Details** panel, enter the properties of YRCPostWindowOpenInitializer extension.
10. The extension point has a defined sequence, which consists of the following attributes:
    - id—The extension is identified by a unique ID which must be specified.
    - name—This is the name given to the extension point. The name is optional. For example, mywindowinitializer.
    - Initializer—This element defines the class to be loaded after the workbench window is created or opened, for post-window initialization. This consists of the following mandatory attribute:
      - class—Specify a fully classified path to a Java class that must be called after the workbench window is opened. This class must implement the IYRCPostWindowOpenInitializer interface.

YRCJasperReport Extension Point

About this task

The YRCJasperReport extension point is a report definition extension point provided in the com.yantra.yfc.rcp plug-in to define or register definitions of Jasper reports. Application plugins can use this extension point to override the default report definitions.

To register your own Jasper report definitions, perform the following steps:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the **Extensions** tab.
5. Click **Add**.
6. From the New Extension window, select **YRCJasperReport** extension point from the list.
7. Click **Finish**.
8. Select the com.yantra.yfc.rcp.YRCJasperReport extension point. The **Extension Details** panel is displayed.
9. In the **Extension Details** panel, enter the properties of YRCJasperReport extension.
10. The extension point has a defined sequence, consisting of one or more elements called JasperReport, for registering or defining each report:
    - id—The extension is identified by a unique report ID which must be specified. Based on this report ID, applications can override or execute Jasper reports.
    - description—Specify the report description, which is required.
    - permissionId—Permission ID is the permission given to a user for launching reports. This is optional.
    - file—Specify the path and the file name of the Jasper report for which you want to register the definitions. Only files with the extension, .jasper, must be specified.

YRCContainerTitleProvider Extension Point

About this task

YRCContainerTitleProvider is a resource provider extension point for PCAs, provided in the com.yantra.yfc.rcp plug-in. This extension point can be used to create and display a title header on the application workbench window.

An interface IYRCContainerTitleHeader, is provided which must be implemented by the Java class specified in the extension point.

To display the title header, perform the following steps:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the **Extensions** tab.
5. Click **Add**.
6. From the New Extension window, select **YRCContainerTitleProvider** extension point from the list.
7. Click **Finish**.
8. Select the com.yantra.yfc.rcp.YRCContainerTitleProvider extension point. The **Extension Details** panel is displayed.
9. In the **Extension Details** panel, enter the properties of YRCContainerTitleProvider extension.
10. The extension point has a defined sequence, which consists of the following attributes:
    - id—The extension is identified by a unique ID which must be specified.
    - name—This is the name given to the extension. The name is optional. For example, containertitle.
    - ApplicationTitleProvider—This element defines the class to be loaded for displaying the title header on the workbench window. This consists of the following mandatory attributes:
      - class—Specify a fully classified path to a Java class that must implement the interface IYRCContainerProvider. This class must create the title control, set user and title information for the title header that must be displayed on the application workbench window.
      - moduleId—Specify the module ID of the PCA for which you want to display the title header. For example, sop.

YRCMessageDisplayer Extension Point

About this task

The YRCMessageDisplayer extension point is a resource provider extension point for PCAs, provided in the com.yantra.yfc.rcp plug-in. The extension point can be used to customize the message view on the application workbench window of the PCAs. An interface, IYRCMessageDisplayer is provided which must be implemented by the class specified in the extension.

The standard message view contains the following:
- Customer name
- Customer message
- Status or error message

You can add or modify messages on the message view by using this extension point.

To customize the message view, perform the following steps:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the **Extensions** tab.
5. Click **Add**.
6. From the New Extension window, select **YRCMessageDisplayer** extension point from the list.
7. Click **Finish**.
8. Select the com.yantra.yfc.rcp.YRCMessageDisplayer extension point. The **Extension Details** panel is displayed.
9. In the **Extension Details** panel, enter the properties of YRCMessageDisplayer extension.
10. The extension point has a defined sequence, which consists of the following attributes:
    - id—The extension is identified by a unique ID which must be specified.
    - MessageDisplayerList—The MessageDisplayerList group element consists of one or more MessageDisplayer elements, each corresponding to the module ID of an application. For example, MessageDisplayerList1 can contain one or more MessageDisplayer elements corresponding to different module IDs of applications such as COM,SOM, or SOP.
    - name—This is the name given to the extension. The name is optional. For example, mymessageDisplayer.
11. MessageDisplayer element: Each MessageDisplayer element belongs to the MessageDisplayerList element and consists of the following mandatory attributes:
    - moduleId—Specify the module ID of the PCA for which you want to customize the message view. For example, ycd (forSterling Call Center and Sterling Store application).
    - class—Specify the Java class that must implement the interface IYRCMessageDisplayer interface.

Creating New Actions

About this task

This section explains how to create new actions and invoke them on clicking of a menu item or button in a Rich Client Platform application.

To create a new action:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:

- Double-click on plugin.xml file.
- Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.

4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **org.eclipse.ui.actionSets** extension point from the list.
7. Click **Finish**. The ActionSet extension element is created.
8. Select the **org.eclipse.ui.actionSets** extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of org.eclipse.ui.actionSets extension point.
10. Select the **ActionSet** extension element. The Extension Element Details panel displays.
11. In the Extension Details panel, enter the properties of the actionSet extension element.
12. In the visible, select **true** from the drop-down menu to make the actions defined in this action set visible.
13. To create a new action extension element, right-click on **actionSet** extension element and select **New** > **action**. The action extension element is created.
14. Select the action extension element. The Extension Element Details panel displays.
15. In id*, enter a unique identifier for the action. This action identifier corresponds to the identifier of the action that gets invoked when you click on a menu item or related task. This action identifier also corresponds to the action identifier specified in the URL field, which is defined within a resource.
16. In class, to specify the implementation class, do any of the following:

    Note: This implementation class can extend either YRCAction class or YRCRelatedTaskAction class. If you are creating an normal action, which is used for menu items then extend the YRCAction class and if you want to create an related task action, which is used for related tasks then extend the YRCRelatedTaskAction class.

    - If you want to select an existing action class, click **Browse**, . The Select Type pop-up window displays. Select the action class that extends the YRCAction class or YRCRelatedTaskAction class. Click on the **class** hyperlink.
    - If you want to create a new action class, click on the **class hyperlink**. The Java Attribute Editor window displays.
      a. In Source Folder, the name of the source folder that you selected to store the action class automatically displays. You can also browse to the folder that you want to specify as the source folder.
      b. In Package, the name of the package that you selected to store the action class automatically displays. This helps you to easily manage your directory structure.
      c. In Name, enter the name of the action class.
      d. In Superclass, click **Browse**, the Superclass Selection window displays.
      e. Select the YRCAction class or YRCRelatedTaskAction class from the list and click OK.

f. In Interfaces:, remove the interface org.eclipse.ui.IWorkbenchWindowActionDelegate, which is resent by default.
g. Check the Constructors from superclass box. The system automatically creates the constructor for the superclass that you specified.
h. Check the Inherited abstract methods box. The system automatically adds the abstract methods inherited by the superclass that you specified.
i. Click **Finish**. The system creates the new action class in the folder or package selected by you.

17. Open the newly created action class in the Java Editor.
18. Depending on the action class that you are extending, write the code to perform the required operation in the inherited abstract execute() method. For example,
    - If you are extending the YRCAction class then write the code for performing the required operation in the execute() method. The execute() method internally checks if the action can be run or not. This check criteria depends on the following criteria:
      – Whether or not the current editor has errors.
      – Whether or not the current editor has been modified.

      Therefore, following methods must be overridden in the class which is extending the YRCAction class:
      – checkForErrors()—This method is used to check if the current editor has errors or not. If you want to skip this check, then return false.
      – checkForModifications()—This method is used to check if the current editor has been modified or not. If you want to skip this check, then return false.
    - If you are extending theYRCRelatedTaskAction class then write the code for performing the required operation in the executeTask() method.

Registering a Plug-In

Every plug-in that is a part of the Rich Client Platform application must be registered. Various features such as localization, theming, configuration, UI extension depend on a plug-in being registered. To register a plug-in in the Rich Client Platform application, you must invoke the registerPlugin() method of the YRCPlatformUI class.

Note: When you create a Rich Client Platform plug-in using the **Rich Client Platform Wizards** > **UI wizards** > **Rich Client Platform Plug-in**, the class for registering a plug-in and other Rich Client Platform-specific resource files is automatically created. Therefore, you need not explicitly register the plug-in and other Rich Client Platform-specific resource files.

A sample code for registering a plug-in is as follows:

```
public class TestPlugin extends AbstractUIPlugin {
    private static TestPlugin plugin;
    public static final String ID="com.mycompany.test.rcp";
    public TestPlugin() {
        super();
        plugin=this;
        try {
            YRCPlatformUI.registerPlugin(ID, this);
           } catch (Exception ex) {
```

```
                YRCPlatformUI.trace(ex);
            }
        }
}
```

Registering Plug-In Files

Every plug-in that wants to use its own resource files such as bundle, theme, configuration files, and so forth must register these files with Rich Client Platform application. You can register all resource files together within the plug-in constructor.

A sample code that can be used to register a plug-in and all its resource files is as follows:

```
public class TestPlugin extends AbstractUIPlugin {
      private static TestPlugin plugin;
      public static final String ID="com.mycompany.test.rcp";
      public TestPlugin() {
             super();
             plugin=this;
             try {
YRCPlatformUI.registerPlugin(ID, this);
YRCPlatformUI.registerConfiguration("com.mycompany.test.rcp_config", ID);
YRCPlatformUI.registerBundle("com.mycompany.test.rcp_bundle", ID);
YRCPlatformUI.registerCommands("com.mycompany.test.rcp_commands", ID);
YRCPlatformUI.registerExtensions("com.mycompany.test.rcp_extn", ID);
YRCPlatformUI.registerTheme("com.mycompany.test.rcp_sapphire", ID);
                } catch (Exception ex) {
                     YRCPlatformUI.trace(ex);
        }
}
```

Registering Bundle File

The bundle file is used for localizing Rich Client Platform applications. Every plug-in that requires its own bundle file should invoke the registerBundle() method of the YRCPlatformUI class during plug-in initialization, preferably within the plug-in constructor to register its bundle file. After the bundle file is registered, it gets loaded using the users current locale. The bundle file must have "properties" extension.

To register the bundle file within the plug-in constructor, for example:

```
YRCPlatformUI.registerBundle("com.yantra.pca.ycd_bundle", ID)
```

where com.yantra.pca.ycd_bundle is the name of the bundle file without ".properties" extension. ID is a unique identifier of the plug-in that registers this bundle file.

Note: Before calling the registerBundle() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class.

Registering Theme File

The theme file is used for setting the color scheme and font properties of Rich Client Platform applications. Every plug-in that requires its own theme file should invoke the registerTheme() method of the YRCPlatformUI class during plug-in initialization, preferably within the plug-in constructor to register its theme file.

To register the theme file within the plug-in constructor, for example:

```
YRCPlatformUI.registerTheme("com.mycompany.test.rcp_skyblue", ID)
```

where com.mycompany.test.rcp_skyblue is the name of the your theme file without ".ythm" extension. ID is a unique identifier of the plug-in that registers this theme file.

Note: Before calling the registerTheme() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class.

Registering Configuration File

The configuration file is used to set the URL path parameters for connecting Rich Client Platform applications to the server. Every plug-in that requires its own configuration file should invoke the registerConfiguration() method of the YRCPlatformUI class during plug-in initialization, preferably within the plug-in constructor to register its configuration file. Configuration file must have extension "ycfg". Plugins can use any custom XML configuration file.

To register your configuration file within the plug-in constructor, for example:

```
YRCPlatformUI.registerConfiguraton("com.mycompany.test.rcp_config", ID)
```

where com.mycompany.test.rcp_config is the name of the your configuration file without ".ycfg" extension. ID is a unique identifier of the plug-in that registers this configuration file.

Note: Before calling the registerConfiguration() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class.

Registering Commands File

The commands file is used to create commands to call different APIs or services. Every plug-in that requires its own set of commands should invoke the registerCommands() method of the YRCPlatformUI class during plug-in initialization, preferably within the plug-in constructor to register its commands file. Commands file must have extension "ycml". The command names are unique, and reusing a command name overrides an existing definition. To register your commands file within the plug-in constructor:

```
YRCPlatformUI.registerCommands("com.mycompany.test.rcp_commands", ID)
```

where com.mycompany.test.rcp_commands is the name of the your commands file without ".ycml" extension. ID is a unique identifier of the plug-in that registers this commands file.

Note: Before calling the registerCommands() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class.

Registering Extension File

The extension file is used to store information about Rich Client Platform applications UI extensibility such as addition of new fields, modification of existing fields, and so forth. Every plug-in that requires its own extension file should invoke the registerExtensions() method of the YRCPlatformUI class during plug-in initialization, preferably within the plug-in constructor to register its extension file. The extension file must have "yuix" extension. To register your extension file within the plug-in constructor:

```
YRCPlatformUI.registerExtensions("com.yantra.order.capture_extn.yuix",ID)
```

where com.yantra.order.capture_extn.yuix is the name of the your extension file with ".extn" extension. ID is a unique identifier of the plug-in that registers this extension file.

Note: Before calling the registerExtensions() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class.

Validating Controls

The Rich Client Platform provides methods to validate various controls. When the controls have target binding, the associated data type is retrieved and appropriate validation is performed at the infrastructure level. You can validate the data entered in the controls such as text box, combo box, button, and so forth by implementing the appropriate validate method. The data type validation can be performed for the value entered by the user. Validations can also be performed for custom criteria. If the data type validation for a control fails, the validate method for that control is not called and an error message is displayed.

The methods for validating the following controls are:

- Text Control—When the text control loses focus, the data type validation and other mandatory validations are performed first. If the validation succeeds, the control is passed to the validateTextField() method.
- Combo Control—When a different item is selected from the combo control, the data type validation and other mandatory validations are performed first. If the validation succeeds, the control is passed to the validateComboField() method.
- Button Control—When the controls such as button, check box, radio button whether selected or unselected, the validateButtonClick() method is invoked.

  Note: In case of radio button, if the validation fails, the focus is set on the selected radio button and not on the original radio button. You will have to explicitly set the focus on the original radio button.

You can extend the default validations using the previously mentioned methods and define custom validations in your action classes. However, the default validations will always be performed after the custom validations and the default validations cannot be suppressed.

If you want to suppress the default validations, hide the control associated with the action that is performing the validation. Add a custom control and define a new action for the custom control with the custom validations that are required. However, ensure that the new action contains the code which performs the same logic as the default validations in addition to the custom validations.

Custom Data Formatting

About this task

Rich Client Platform enables you to perform custom data formatting.

For example, say that the user enters 6175677890 in the Phone Number field and presses the **Tab** key. You want to format this number and display it as 617-567-7890.

To display the formatted value, you must associate the formatted logic with the Phone Number field. You can perform custom formatting for a field by extending the YRCDataFormatter extension point.

Note: Rich Client Platform supports custom data formatting for label, text, and styled text controls.

To extend the YRCDataFormatter extension point:

Procedure

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created.
3. To open the plugin.xml file in the Plug-in Manifest editor, do any of the following:
   - Double-click on plugin.xml file.
   - Right-click on plugin.xml file and select **Open With** > **Plug-in Manifest Editor**.
4. Select the Extensions tab.
5. Click **Add**.
6. From the New Extension window, select **com.yantra.yfc.rcp.YRCDataFormatter** extension point from the list.
7. Click **Finish**.
8. Select the **com.yantra.yfc.rcp.YRCDataFormatter** extension point. The Extension Details panel displays.
9. In the Extension Details panel, enter the properties of the com.yantra.yfc.rcp.YRCDataFormatter extension point.
10. In ID, enter a unique identifier for the com.yantra.yfc.rcp.YRCDataFormatter extension point. This is a mandatory field.
11. To create a new dataFormatter extension element, right-click on **com.yantra.yfc.rcp.YRCDataFormatter** and select **New** > **dataFormatter**. The dataFormatter extension element is created.
12. Select the dataFormatter extension element. The Extension Element Details panel displays.
13. In the Extension Details panel, enter the properties of the dataFormatter extension element.
14. In attributeBinding*, enter the name of the XPath attribute whose value you want to custom format. For example, the attribute binding can be set as DayPhone.
15. In class, specify the implementation class by doing any of the following:
    - Click **Browse**. The Select Type pop-up window displays. Select the implementation class that contains the formatting logic for the field.
    - Click on the **class*** hyperlink. The Java Attribute Editor window displays.
      - In Source Folder, the name of the source folder that you selected to store the implementation class displays. You can also browse to the folder that you want to specify as the source folder.
      - In Package, the name of the package that you selected to store the implementation class displays. This enables you to easily manage your directory structure.
      - In Name, enter the name of the implementation class.
      - In Superclass, click Browse. The Superclass Selection window displays.

- Enter the YRCDataFormatter class and click **OK.**
- Check the Constructors from superclass box. The system creates the constructor for the superclass that you specified.
- Check the Inherited abstract methods box. The system automatically adds the abstract methods inherited by the superclass that you specified.
- Click **Finish**. The system creates the new implementation class in the folder or package selected by you.

16. Open the newly created implementation class in the Java editor.
17. Override the inherited abstract getFormattedValue() method. Write the formatting code for displaying the field value and return the formatted value. For example, the formatting logic can be:

```
public YRCFormatResponse getFormattedValue(String attributBinding,
String value) {
       YRCFormatResponse response = null;
//validForDataType(String)method can be used to do custom validation
//on the value of the field.
//Based on the validation we can set the response.
if(validForDataType(value)) {
     String retVal =  value.substring(0, 3)+"-"+value.substring(3, 6)+
"-"+ value.substring(6);
        response =
new YRCFormatResponse(YRCFormatResponse.YRC_VALIDATION_OK,
"Valid Format", retVal);
         }
else{
         response =
new YRCFormatResponse(YRCFormatResponse.YRC_VALIDATION_ERROR,
"InValid Format", null);
     }
     return response;
}
```

If you want to perform some custom validation on the field value, you can write your own logic to validate the value. For example, in the following code the validForDataType(String) method is used to perform custom validation on the field value.

```
private boolean validForDataType(String value) {
        if(value.length()==10){
           return true;
        }
        return false;
}
```

18. Override the inherited abstract getDeformattedValue() method. Write the deformatted value of the field you want to store in the XML and return the deformatted value. For example, the deformatting logic can be:

```
public
String getDeformattedValue(String attributBinding, String value) {
   String retVal=null;
   String [] retValArray = value.split("-");
   for(int i=0;i<retValArray.length;i++){
       if (i==0) {
          retVal = retValArray[i];
      }else {
          retVal = retVal+retValArray[i];
             }
       }
   return retVal;
}
```

Siblings

Siblings are the first level children of the parent. For example, let us consider the following scenario:

Here, the siblings of the OrderNo label are text box (Y001), button (Search), and Group2, which are the first level children of Group1. Similarly, the sibling of ItemID label is the text box (SKU-1001).

Rich Client Platform Utilities

Rich Client Platform provides a utility tool using which you can gather information for a particular UI or form such as form id, models used, and so forth.

Viewing Screen Models

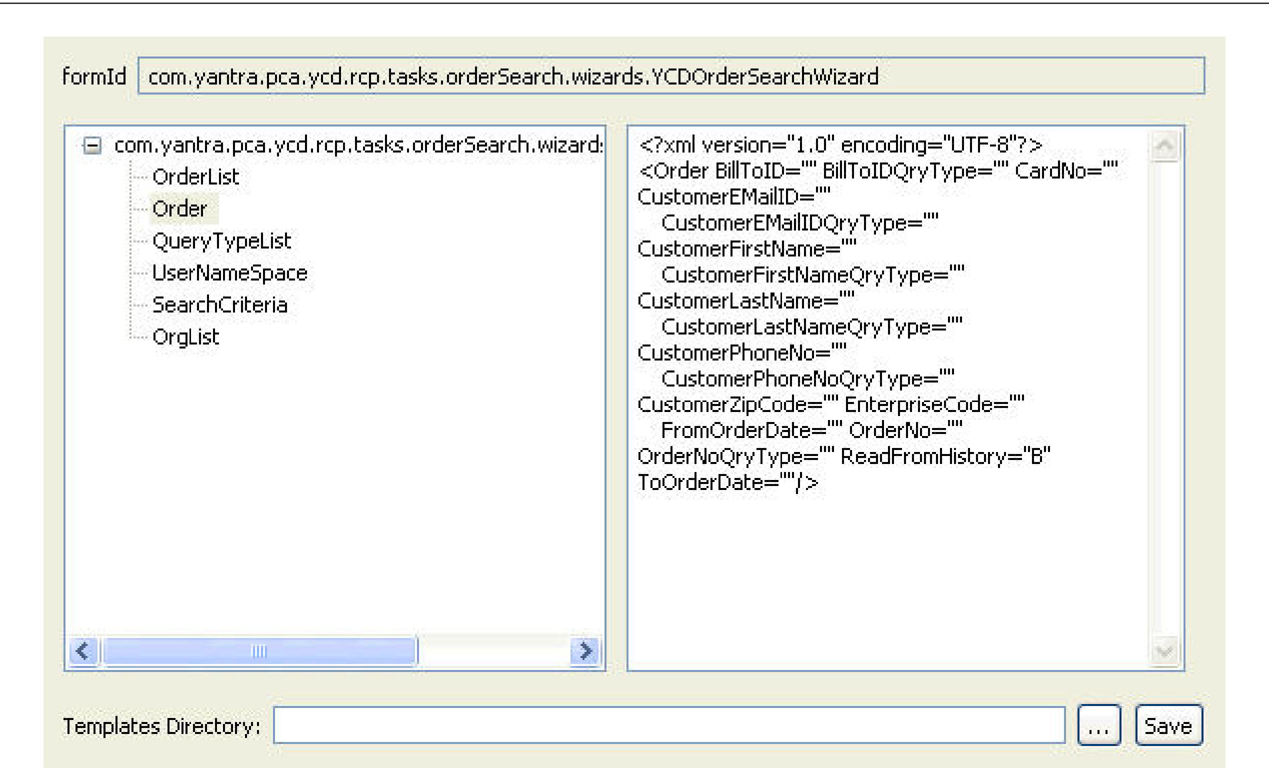
About this task

By using the Screen Model utility tool, you can view the form identifier and all models used in any UI, along with the various elements and attributes used in a model. You can also save all screen models as templates.

To view a screen model:

Procedure

1. Run the appropriate Rich Client Platform application.
2. After you successfully log in to the application, the application window displays.
3. Open the screen for which you want to view models.
4. Press **CTRL+SHIFT+M** to view screen model. The Screen Models window displays.



In the formId field, the identifier of the form displays, which is used to identify the screen.

The left-hand side panel displays a list of models used in the screen as a tree structure with root being the form identifier of the screen. After you select a specific model, you can view a list of elements and attributes defined in the model on the right-hand side panel. If a screen contains embedded screens in it, then you can view a list of models used for each screen.

Saving Models as Templates

About this task

To save existing models as templates:

Procedure

1. Click the button next to the Templates Directory field. The Choose Directory pop-up window displays.
2. Select the directory where you want to store the models of the screen as templates, and click **OK**.
3. Click **Save**. The system stores the models of each screen as templates in their respective folders.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing

Legal and Intellectual Property Law

IBM Japan Ltd.

1623-14, Shimotsuruma, Yamato-shi

Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be

© Copyright IBM Corp. 1999, 2013

incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

J46A/G4

555 Bailey Avenue

San Jose, CA 95141-1003

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© IBM 2013. Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 2013.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium and the Ultrium Logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Connect Control Center®, Connect:Direct®, Connect:Enterprise®, Gentran®, Gentran®:Basic®, Gentran:Control®, Gentran:Director®, Gentran:Plus®, Gentran:Realtime®, Gentran:Server®, Gentran:Viewpoint®, Sterling Commerce™, Sterling Information Broker®, and Sterling Integrator® are trademarks or registered trademarks of Sterling Commerce®, Inc., an IBM Company.

Other company, product, and service names may be trademarks or service marks of others.

IBM®

Product Number: xxxx-xxx

Printed in USA