Sterling Commerce
An IBM Company

# Selling and Fulfillment Foundation: Customizing the Rich Client Platform Interface Guide

Release 8.5

*Last updated in HF20*

August 2010

# Copyright Notice



## STERLING COMMERCE SOFTWARE



---

Sterling Commerce, Inc.
4600 Lakehurst Court
Dublin, Ohio 43016-2000

## Third-Party Software

Portions of the Sterling Commerce Software may include products, or may be distributed on the same storage media with products, ("Third Party Software") offered by third parties ("Third Party Licensors"). Sterling Commerce Software may include Third Party Software covered by the following copyrights: Copyright © 2006-2008 Andres Almiray. Copyright © 1999-2005 The Apache Software Foundation. Copyright (c) 2008 Azer Koçulu http://azer.kodfabrik.com. Copyright © Einar Lielmanis, einars@gmail.com. Copyright (c) 2006 John Reilly (www.inconspicuous.org) and Copyright (c) 2002 Douglas Crockford (www.crockford.com). Copyright (c) 2009 John Resig, http://jquery.com/. Copyright © 2006-2008 Json-lib. Copyright © 2001 LOOX Software, Inc. Copyright © 2003-2008 Luck Consulting Pty. Ltd. Copyright 2002-2004 © MetaStuff, Ltd. Copyright © 2009 Michael Mathews micmath@gmail.com. Copyright © 1999-2005 Northwoods Software Corporation. Copyright (C) Microsoft Corp. 1981-1998. Purple Technology, Inc. Copyright (c) 2004-2008 QOS.ch. Copyright © 2005 Sabre Airline Solutions. Copyright © 2004 SoftComplex, Inc. Copyright © 2000-2007 Sun Microsystems, Inc. Copyright © 2001 VisualSoft Technologies Limited. Copyright © 2001 Zero G Software, Inc. All rights reserved by all listed parties.

The Sterling Commerce Software is distributed on the same storage media as certain Third Party Software covered by the following copyrights: Copyright © 1999-2006 The Apache Software Foundation. Copyright (c) 2001-2003 Ant-Contrib project. Copyright © 1998-2007 Bela Ban. Copyright © 2005 Eclipse Foundation. Copyright © 2002-2006 Julian Hyde and others. Copyright © 1997 ICE Engineering, Inc./Timothy Gerard Endres. Copyright 2000, 2006 IBM Corporation and others. Copyright © 1987-2006 ILOG, Inc. Copyright © 2000-2006 Infragistics. Copyright © 2002-2005 JBoss, Inc. Copyright LuMriX.net GmbH, Switzerland. Copyright © 1998-2009 Mozilla.org. Copyright © 2003-2009 Mozdev Group, Inc. Copyright © 1999-2002 JBoss.org. Copyright Raghu K, 2003. Copyright © 2004 David Schweinsberg. Copyright © 2005-2006 Darren L. Spurgeon. Copyright © S.E. Morris (FISH) 2003-04. Copyright © 2006 VisualSoft Technologies. Copyright © 2002-2009 Zipwise Software. All rights reserved by all listed parties.

Certain components of the Sterling Commerce Software are distributed on the same storage media as Third Party Software which are not listed above. Additional information for such Third Party Software components of the Sterling Commerce Software is located at: installdir/mesa/studio/plugins/SCI_Studio_License.txt.

Third Party Software which is included, or are distributed on the same storage media with, the Sterling Commerce Software where use, duplication, or disclosure by the United States government or a government contractor or subcontractor, are provided with RESTRICTED RIGHTS under Title 48 CFR 2.101, 12.212, 52.227-19, 227.7201 through 227.7202-4, DFAR 252.227-7013(c) (1) (ii) and (2), DFAR 252.227-7015(b)(6/95), DFAR 227.7202-3(a), FAR 52.227-14(g)(2)(6/87), and FAR 52.227-19(c)(2) and (6/87) as applicable.

Additional information regarding certain Third Party Software is located at installdir/SCI_License.txt.

Some Third Party Licensors also provide license information and/or source code for their software via their respective links set forth below:

http://danadler.com/jacob/

http://www.dom4j.org

This product includes software developed by the Apache Software Foundation (http://www.apache.org). This product includes software developed by the Ant-Contrib project (http://sourceforge.net/projects/ant-contrib). This product includes software developed by the JDOM Project (http://www.jdom.org/). This product includes code licensed from RSA Data Security (via Sun Microsystems, Inc.). Sun, Sun Microsystems, the Sun Logo, Java, JDK, the Java Coffee Cup logo, JavaBeans , JDBC, JMX and all JMX based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. All other trademarks and logos are trademarks of their respective owners.

## THE APACHE SOFTWARE FOUNDATION SOFTWARE

The Sterling Commerce Software is distributed with or on the same storage media as the following software products (or components thereof) and java source code files: Xalan version 2.5.2, Cookie.java, Header.java, HeaderElement.java, HttpException.java, HttpState.java, NameValuePair.java, CronTimeTrigger.java, DefaultTimeScheduler.java, PeriodicTimeTrigger.java, Target.java,

TimeScheduledEntry.java, TimeScheduler.java, TimeTrigger.java, Trigger.java, BinaryHeap.java, PriorityQueue.java, SynchronizedPriorityQueue.java, GetOpt.java, GetOptsException.java, IllegalArgumentException.java, MissingOptArgException.java (collectively, "Apache 1.1 Software"). Apache 1.1 Software is free software which is distributed under the terms of the following license:

## License Version 1.1



Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgement: "This product includes software developed by the Apache Software Foundation (http://www.apache.org)." Alternatively, this acknowledgement may appear in the software itself, if and whenever such third-party acknowledgements normally appear.

4. The names "Commons", "Jakarta", "The Jakarta Project", "HttpClient", "log4j", "Xerces "Xalan", "Avalon", "Apache Avalon", "Avalon Cornerstone", "Avalon Framework", "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without specific prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without the prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMIPLIED WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTIBILITY, AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTIAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. The GetOpt.java, GetOptsException.java, IlligalArgumentException.java and MissingOptArgException.java software was originally based on software copyright (c) 2001, Sun Microsystems., http://www.sun.com. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

The preceding license only applies to the Apache 1.1 Software and does not apply to the Sterling Commerce Software or to any other Third-Party Software.

The Sterling Commerce Software is also distributed with or on the same storage media as the following software products (or components thereof): Ant, Antinstaller, Apache File Upload Package, Apache Commons Beans, Apache Commons BetWixt, Apache Commons Collection, Apache Commons Digester, Apache Commons IO, Apache Commons Lang., Apache Commons Logging, Apache Commons Net, Apache Jakarta Commons Pool, Apache Jakarta ORO, Lucene, Xerces version 2.7, Apache Log4J, Apache SOAP, Apache Struts and Apache Xalan 2.7.0, (collectively, "Apache 2.0 Software"). Apache 2.0 Software is free software which is distributed under the terms of the Apache License Version 2.0. A copy of License Version 2.0 is found in the following directory files for the individual pieces of the Apache 2.0 Software: installdir/jar/commons_upload/1_0/ CommonsFileUpload_License.txt,
installdir/jar/jetspeed/1_4/RegExp_License.txt,
installdir/ant/Ant_License.txt
<install>/jar/antInstaller/0_8/antinstaller_License.txt,
<install>/jar/commons_beanutils/1_7_0/commons-beanutils.jar (/META-INF/LICENSE.txt),
<install>/jar/commons_betwixt/0_8/commons-betwixt-0.8.jar (/META-INF/LICENSE.txt),

<install>/jar/commons_collections/3_2/LICENSE.txt,
<install>/jar/commons_digester/1_8/commons-digester-1.8.jar (/META-INF/LICENSE.txt),
<install>/jar/commons_io/1_4/LICENSE.txt,
<install>/jar/commons_lang/2_1/Commons_Lang_License.txt,
<install>/jar/commons_logging/1_0_4/commons-logging-1.0.4.jar (/META-INF/LICENSE.txt),
<install>/jar/commons_net/1_4_1/commons-net-1.4.1.jar (/META-INF/LICENSE.txt),
<install>/jar/smcfs/8.5/lucene-core-2.4.0.jar (/META-INF/LICENSE.txt),
<install>/jar/struts/2_0_11/struts2-core-2.0.11.jar (./LICENSE.txt),
<install>/jar/mesa/gisdav/WEB-INF/lib/Slide_License.txt,
<install>/mesa/studio/plugins/xerces_2.7_license.txt,
<install>/jar/commons_pool/1_2/Commons_License.txt,
<install>/jar/jakarta_oro/2_0_8/JakartaOro_License.txt,
<install>/jar/log4j/1_2_15/LOG4J_License.txt,
<install>/jar/xalan/2_7/Xalan_License.txt,
<install>/jar/soap/2_3_1/Apache_SOAP_License.txt

Unless otherwise stated in a specific directory, the Apache 2.0 Software was not modified. Neither the Sterling Commerce Software, modifications, if any, to Apache 2.0 Software, nor other Third Party Code is a Derivative Work or a Contribution as defined in License Version 2.0. License Version 2.0 applies only to the Apache 2.0 Software which is the subject of the specific directory file and does not apply to the Sterling Commerce Software or to any other Third Party Software. License Version 2.0 includes the following provision:

"Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License."

NOTICE file corresponding to the section 4 d of the Apache License, Version 2.0, in this case for the Apache Ant distribution. Apache Ant Copyright 1999-2008 The Apache Software Foundation. This product includes software developed by The Apache Software Foundation (http://www.apache.org/). This product includes also software developed by :

- the W3C consortium (http://www.w3c.org) ,

- the SAX project (http://www.saxproject.org)

The <sync> task is based on code Copyright (c) 2002, Landmark Graphics Corp that has been kindly donated to the Apache Software Foundation.

Portions of this software were originally based on the following:

- software copyright (c) 1999, IBM Corporation., http://www.ibm.com.

- software copyright (c) 1999, Sun Microsystems., http://www.sun.com.

- voluntary contributions made by Paul Eng on behalf of the Apache Software Foundation that were originally developed at iClick, Inc., software copyright (c) 1999.

NOTICE file corresponding to the section 4 d of the Apache License, Version 2.0, in this case for the Apache Lucene distribution. Apache Lucene Copyright 2006 The Apache Software Foundation. This product includes software developed by The Apache Software Foundation (http://www.apache.org/). The snowball stemmers in contrib/snowball/src/java/net/sf/snowball were developed by Martin Porter and Richard Boulton. The full snowball package is available from http://snowball.tartarus.org/

## Ant-Contrib Software

The Sterling Commerce Software is distributed with or on the same storage media as the Anti-Contrib software (Copyright (c) 2001-2003 Ant-Contrib project. All rights reserved.) (the "Ant-Contrib Software"). The Ant-Contrib Software is free software which is distributed under the terms of the following license:

The Apache Software License, Version 1.1

## DOM4J Software

THIS SOFTWARE IS PROVIDED BY METASTUFF, LTD. AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL METASTUFF, LTD. OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



The preceding license only applies to the Dom4j Software and does not apply to the Sterling Commerce Software, or any other Third-Party Software.

## THE ECLIPSE SOFTWARE FOUNDATION

The Sterling Commerce Software is also distributed with or on the same storage media as the following software:

com.ibm.icu.nl1_3.4.4.v200606220026.jar, org.eclipse.ant.core.nl1_3.1.100.v200606220026.jar,
org.eclipse.ant.ui.nl1_3.2.0.v200606220026.jar, org.eclipse.compare.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.boot.nl1_3.1.100.v200606220026.jar,
org.eclipse.core.commands.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.contenttype.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.expressions.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.filebuffers.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.filesystem.nl1_1.0.0.v200606220026.jar,
org.eclipse.core.jobs.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.resources.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.runtime.compatibility.auth.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.runtime.compatibility.nl1_3.1.100.v200606220026.jar,
org.eclipse.core.runtime.nl1_3.2.0.v200606220026.jar,
org.eclipse.core.variables.nl1_3.1.100.v200606220026.jar,
org.eclipse.debug.core.nl1_3.2.0.v200606220026.jar,
org.eclipse.debug.ui.nl1_3.2.0.v200606220026.jar,
org.eclipse.equinox.common.nl1_3.2.0.v200606220026.jar,
org.eclipse.equinox.preferences.nl1_3.2.0.v200606220026.jar,
org.eclipse.equinox.registry.nl1_3.2.0.v200606220026.jar,
org.eclipse.help.appserver.nl1_3.1.100.v200606220026.jar,
org.eclipse.help.base.nl1_3.2.0.v200606220026.jar, org.eclipse.help.nl1_3.2.0.v200606220026.jar,
org.eclipse.help.ui.nl1_3.2.0.v200606220026.jar, org.eclipse.jdt.apt.core.nl1_3.2.0.v200606220026.jar,
org.eclipse.jdt.apt.ui.nl1_3.2.0.v200606220026.jar,
org.eclipse.jdt.core.manipulation.nl1_1.0.0.v200606220026.jar,
org.eclipse.jdt.core.nl1_3.2.0.v200606220026.jar,
org.eclipse.jdt.debug.ui.nl1_3.2.0.v200606220026.jar,
org.eclipse.jdt.doc.isv.nl1_3.2.0.v200606220026.jar,
org.eclipse.jdt.doc.user.nl1_3.2.0.v200606220026.jar,
org.eclipse.jdt.junit4.runtime.nl1_1.0.0.v200606220026.jar,
org.eclipse.jdt.launching.nl1_3.2.0.v200606220026.jar, org.eclipse.jdt.nl1_3.2.0.v200606220026.jar,
org.eclipse.jdt.ui.nl1_3.2.0.v200606220026.jar,
org.eclipse.jface.databinding.nl1_1.0.0.v200606220026.jar,
org.eclipse.jface.nl1_3.2.0.v200606220026.jar, org.eclipse.jface.text.nl1_3.2.0.v200606220026.jar,
org.eclipse.ltk.core.refactoring.nl1_3.2.0.v200606220026.jar,
org.eclipse.ltk.ui.refactoring.nl1_3.2.0.v200606220026.jar,
org.eclipse.osgi.nl1_3.2.0.v200606220026.jar, org.eclipse.osgi.services.nl1_3.1.100.v200606220026.jar,
org.eclipse.osgi.util.nl1_3.1.100.v200606220026.jar, org.eclipse.pde.core.nl1_3.2.0.v200606220026.jar,
org.eclipse.pde.doc.user.nl1_3.2.0.v200606220026.jar,
org.eclipse.pde.junit.runtime.nl1_3.2.0.v200606220026.jar,
org.eclipse.pde.nl1_3.2.0.v200606220026.jar, org.eclipse.pde.runtime.nl1_3.2.0.v200606220026.jar,
org.eclipse.pde.ui.nl1_3.2.0.v200606220026.jar,
org.eclipse.platform.doc.isv.nl1_3.2.0.v200606220026.jar,
org.eclipse.platform.doc.user.nl1_3.2.0.v200606220026.jar,

org.eclipse.rcp.nl1_3.2.0.v200606220026.jar, org.eclipse.search.nl1_3.2.0.v200606220026.jar,
org.eclipse.swt.nl1_3.2.0.v200606220026.jar, org.eclipse.team.core.nl1_3.2.0.v200606220026.jar,
org.eclipse.team.cvs.core.nl1_3.2.0.v200606220026.jar,
org.eclipse.team.cvs.ssh.nl1_3.2.0.v200606220026.jar,
org.eclipse.team.cvs.ssh2.nl1_3.2.0.v200606220026.jar,
org.eclipse.team.cvs.ui.nl1_3.2.0.v200606220026.jar, org.eclipse.team.ui.nl1_3.2.0.v200606220026.jar,
org.eclipse.text.nl1_3.2.0.v200606220026.jar, org.eclipse.ui.browser.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.cheatsheets.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.console.nl1_3.1.100.v200606220026.jar,
org.eclipse.ui.editors.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.externaltools.nl1_3.1.100.v200606220026.jar,
org.eclipse.ui.forms.nl1_3.2.0.v200606220026.jar, org.eclipse.ui.ide.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.intro.nl1_3.2.0.v200606220026.jar, org.eclipse.ui.navigator.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.navigator.resources.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.presentations.r21.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.views.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.views.properties.tabbed.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.workbench.nl1_3.2.0.v200606220026.jar,
org.eclipse.ui.workbench.texteditor.nl1_3.2.0.v200606220026.jar,
org.eclipse.update.configurator.nl1_3.2.0.v200606220026.jar,
org.eclipse.update.core.nl1_3.2.0.v200606220026.jar,
org.eclipse.update.scheduler.nl1_3.2.0.v200606220026.jar,
org.eclipse.update.ui.nl1_3.2.0.v200606220026.jar,
com.ibm.icu_3.4.4.1.jar,
org.eclipse.core.commands_3.2.0.I20060605-1400.jar,
org.eclipse.core.contenttype_3.2.0.v20060603.jar,
org.eclipse.core.expressions_3.2.0.v20060605-1400.jar,
org.eclipse.core.filesystem.linux.x86_1.0.0.v20060603.jar,
org.eclipse.core.filesystem_1.0.0.v20060603.jar, org.eclipse.core.jobs_3.2.0.v20060603.jar,
org.eclipse.core.runtime.compatibility.auth_3.2.0.v20060601.jar,
org.eclipse.core.runtime_3.2.0.v20060603.jar, org.eclipse.equinox.common_3.2.0.v20060603.jar,
org.eclipse.equinox.preferences_3.2.0.v20060601.jar, org.eclipse.equinox.registry_3.2.0.v20060601.jar,
org.eclipse.help_3.2.0.v20060602.jar, org.eclipse.jface.text_3.2.0.v20060605-1400.jar,
org.eclipse.jface_3.2.0.I20060605-1400.jar, org.eclipse.osgi_3.2.0.v20060601.jar,
org.eclipse.swt.gtk.linux.x86_3.2.0.v3232m.jar, org.eclipse.swt_3.2.0.v3232o.jar,
org.eclipse.text_3.2.0.v20060605-1400.jar,
org.eclipse.ui.workbench.texteditor_3.2.0.v20060605-1400.jar,
org.eclipse.ui.workbench_3.2.0.I20060605-1400.jar, org.eclipse.ui_3.2.0.I20060605-1400.jar,
runtime_registry_compatibility.jar, eclipse.exe, eclipse.ini, and startup.jar
(collectively, "Eclipse Software").
All Eclipse Software is distributed under the terms and conditions of the Eclipse Foundation Software User Agreement (EFSUA) and/or terms and conditions of the Eclipse Public License Version 1.0 (EPL) or other license agreements, notices or terms and conditions referenced for the individual pieces of the Eclipse Software, including without limitation "Abouts", "Feature Licenses", and "Feature Update Licenses" as defined in the EFSUA .

A copy of the Eclipse Foundation Software User Agreement is found at
<install_dir>/SI/repository/rcp/rcpdependencies/windows/eclipse/notice.html,
<install_dir>/SI/repository/rcp/rcpdependencies/windows/eclipse/plugins/notice.html,
<install_dir>/SI/repository/rcp/rcpdependencies/gtk.linux_x86/eclipse/notice.html, and
<install_dir>/SI/repository/rcp/rcpdependencies/gtk.linux_x86/eclipse/plugins/notice.html.

A copy of the EPL is found at
<install_dir>/SI/repository/rcp/rcpdependencies/windows/eclipse/plugins/epl-v10.htm,
<install_dir>/SI/repository/rcp/rcpdependencies/windows/eclipse/epl-v10.htm,
<install_dir>/SI/repository/rcp/rcpdependencies/gtk.linux.x86/eclipse/plugins/epl-v10.html, and
<install_dir>/SI/repository/rcp/rcpdependencies/gtk.linux.x86/eclipse/epl-v10.html.

The reference to the license agreements, notices or terms and conditions governing each individual piece of the Eclipse Software is found in the directory files for the individual pieces of the Eclipse Software as described in the file identified as installdir/SCI_License.txt.

These licenses only apply to the Eclipse Software and do not apply to the Sterling Commerce Software, or any other Third Party Software.

The Language Pack (NL Pack) piece of the Eclipse Software, is distributed in object code form. Source code is available at http://archive.eclipse.org/eclipse/downloads/drops/L-3.2_Language_Packs-200607121700/index.php. In the event the source code is no longer available from the website referenced above, contact Sterling Commerce at 978-513-6000 and ask for the Release Manager. A copy of this license is located at <install_dir>/SI/repository/rcp/rcpdependencies/windows/eclipse/plugins/epl-v10.htm and

<install_dir>/SI/repository/rcp/rcpdependencies/gtk.linux.x86/eclipse/plugins/epl-v10.html.

The org.eclipse.core.runtime_3.2.0.v20060603.jar piece of the Eclipse Software was modified slightly in order to remove classes containing encryption items. The org.eclipse.core.runtime_3.2.0.v20060603.jar was modified to remove the Cipher, CipherInputStream and CipherOutputStream classes and rebuild the org.eclipse.core.runtime_3.2.0.v20060603.jar.

## Ehcache Software

The Sterling Commerce Software is also distributed with or on the same storage media as the ehache v.1.5 software (Copyright © 2003-2008 Luck Consulting Pty. Ltd.) ("Ehache Software"). Ehcache Software is free software which is distributed under the terms of the Apache License Version 2.0. A copy of License Version 2.0 is found in <install>/jar/smcfs/8.5/ehcache-1.5.0.jar (./LICENSE.txt).

The Ehcache Software was not modified. Neither the Sterling Commerce Software, modifications, if any, to the Ehcache Software, nor other Third Party Code is a Derivative Work or a Contribution as defined in License Version 2.0. License Version 2.0 applies only to the Ehcache Software which is the subject of the specific directory file and does not apply to the Sterling Commerce Software or to any other Third Party Software. License Version 2.0 includes the following provision:

"Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License."

## EZMorph Software

The Sterling Commerce Software is also distributed with or on the same storage media as the EZMorph v. 1.0.4 software (Copyright © 2006-2008 Andres Almiray) ("EZMorph Software"). EZMorph Software is free software which is distributed under the terms of the Apache License Version 2.0. A copy of License Version 2.0 is found in <install>/jar/ezmorph/1_0_4/ezmorph-1.0.4.jar (./LICENSE.txt).

The EZMorph Software was not modified. Neither the Sterling Commerce Software, modifications, if any, to the EZMorph Software, nor other Third Party Code is a Derivative Work or a Contribution as defined in License Version 2.0. License Version 2.0 applies only to the EZMorph Software which is the subject of the specific directory file and does not apply to the Sterling Commerce Software or to any other Third Party Software. License Version 2.0 includes the following provision:

"Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License."

## Firebug Lite Software

The Sterling Commerce Software is distributed with or on the same storage media as the Firebug Lite Software which is free software distributed under the terms of the following license:

Redistribution and use of this software in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of Azer Koçulu. nor the names of any other contributors may be used to endorse or promote products derived from this software without specific prior written permission of Parakey Inc.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## ICE SOFTWARE

The Sterling Commerce Software is distributed on the same storage media as the ICE Software (Copyright © 1997 ICE Engineering, Inc./Timothy Gerard Endres.) ("ICE Software"). The ICE Software is independent from and not linked or compiled with the Sterling Commerce Software. The ICE Software is a free software product which can be distributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License or any later version.

A copy of the GNU General Public License is provided at installdir/jar/jniregistry/1_2/ICE_License.txt. This license only applies to the ICE Software and does not apply to the Sterling Commerce Software, or any other Third Party Software.

The ICE Software was modified slightly in order to fix a problem discovered by Sterling Commerce involving the RegistryKey class in the RegistryKey.java in the JNIRegistry.jar. The class was modified to comment out the finalize () method and rebuild of the JNIRegistry.jar file.

Source code for the bug fix completed by Sterling Commerce on January 8, 2003 is located at: installdir/jar/jniregistry/1_2/RegistryKey.java. Source code for all other components of the ICE Software is located at http://www.trustice.com/java/jnireg/index.shtml.

The ICE Software is distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

## JBOSS SOFTWARE

The Sterling Commerce Software is distributed on the same storage media as the JBoss Software (Copyright © 1999-2002 JBoss.org) ("JBoss Software"). The JBoss Software is independent from and not linked or compiled with the Sterling Commerce Software. The JBoss Software is a free software product which can be distributed and/or modified under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License or any later version.

A copy of the GNU Lesser General Public License is provided at:
<install_dir>\jar\jboss\4_2_0\LICENSE.html

This license only applies to the JBoss Software and does not apply to the Sterling Commerce Software, or any other Third Party Software.

The JBoss Software is not distributed by Sterling Commerce in its entirety. Rather, the distribution is limited to the following jar files: el-api.jar, jasper-compiler-5.5.15.jar, jasper-el.jar, jasper.jar, jboss-common-client.jar, jboss-j2ee.jar, jboss-jmx.jar, jboss-jsr77-client.jar, jbossmq-client.jar,

jnpserver.jar, jsp-api.jar, servlet-api.jar, tomcat-juli.jar.

The JBoss Software was modified slightly in order to allow the ClientSocketFactory to return a socket connected to a particular host in order to control the host interfaces, regardless of whether the ClientSocket Factory specified was custom or note. Changes were made to org.jnp..server.Main. Details concerning this change can be found at http://sourceforge.net/tracker/?func=detail&aid=1008902&group_id=22866&atid=376687.

Source code for the modifications completed by Sterling Commerce on August 13, 2004 is located at: http://sourceforge.net/tracker/?func=detail&aid=1008902&group_id=22866&atid=376687. Source code for all other components of the JBoss Software is located at http://www.jboss.org.

## JGO SOFTWARE

The Sterling Commerce Software is distributed with, or on the same storage media, as certain redistributable portions of the JGo Software provided by Northwoods Software Corporation under a commercial license agreement (the "JGo Software"). The JGo Software is provided subject to the disclaimers set forth above and the following notice:

U.S. Government Restricted Rights

The JGo Software and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (C)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (C)(1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable. Contractor / manufacturer of the JGo Software is Northwoods Software Corporation, 142 Main St., Nashua, NH 03060.

## JSLib Software

The Sterling Commerce Software is distributed with or on the same storage media as the JSLib software product (Copyright (c) 2003-2009 Mozdev Group, Inc.) ("JSLib Software"). The JSLib Software is distributed under the terms of the MOZILLA PUBLIC LICENSE Version 1.1. A copy of this license is located at <install>\repository\eardata\platform_uifwk_ide\war\designer\MPL-1.1.txt. The JSLib Software code is distributed in source form and is located at http://jslib.mozdev.org/installation.html. Neither the Sterling Commerce Software nor any other Third-Party Code is a Modification or Contribution subject to the Mozilla Public License. Pursuant to the terms of the Mozilla Public License, the following notice applies only to the JSLib Software (and not to the Sterling Commerce Software or any other Third-Party Software):

"The contents of the file located at http://www.mozdev.org/source/browse/jslib/ are subject to the Mozilla Public License Version 1.1 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/MPL-1.1.html.

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

The Original Code is Mozdev Group, Inc. code. The Initial Developer of the Original Code is Mozdev Group, Inc. Portions created by_Mozdev Group, Inc. are Copyright © 2003 Mozdev Group, Inc. All Rights Reserved. Original Author: Pete Collins <pete@mozdev.org>one Contributor(s):_____none listed_____.

Alternatively, the contents of this file may be used under the terms of the ____ license (the "[___] License"), in which case the provisions of [___] License are applicable instead of those above. If you wish to allow use of your version of this file only under the terms of the [___] License and not allow others to use your version of this file under the MPL, indicate your decision by deleting the provisions above and replace them with the notice and other provisions required by the [___] License. If you do not delete the provisions above, a recipient may use your version of this file under either the MPL or the [___] License."

The preceding license only applies to the JSLib Software and does not apply to the Sterling Commerce Software, or any other Third-Party Software.

## Json Software

The Sterling Commerce Software is also distributed with or on the same storage media as the Json 2.2.2 software (Copyright © 2006-2008 Json-lib) ("Json Software"). Json Software is free software which is distributed under the terms of the Apache License Version 2.0. A copy of License Version 2.0 is found in <install>/jar/jsonlib/2_2_2/json-lib-2.2.2-jdk13.jar.

This product includes software developed by Douglas Crockford (http://www.crockford.com).

The Json Software was not modified. Neither the Sterling Commerce Software, modifications, if any, to the Json Software, nor other Third Party Code is a Derivative Work or a Contribution as defined in License Version 2.0. License Version 2.0 applies only to the Json Software which is the subject of the specific directory file and does not apply to the Sterling Commerce Software or to any other Third Party Software. License Version 2.0 includes the following provision:

"Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License."

## Purple Technology

The Sterling Commerce Software is distributed with or on the same storage media as the Purple Technology Software (Copyright (c) 1995-1999 Purple Technology, Inc.) ("Purple Technology Software"), which is subject to the following license:

Copyright (c) 1995-1999 Purple Technology, Inc. All rights reserved.

PLAIN LANGUAGE LICENSE: Do whatever you like with this code, free of charge, just give credit where credit is due. If you improve it, please send your improvements to alex@purpletech.com. Check http://www.purpletech.com/code/ for the latest version and news.

LEGAL LANGUAGE LICENSE: Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The names of the authors and the names "Purple Technology," "Purple Server" and "Purple Chat" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact server@purpletech.com.

THIS SOFTWARE IS PROVIDED BY THE AUTHORS AND PURPLE TECHNOLOGY "AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR PURPLE TECHNOLOGY BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The preceding license only applies to the Purple Technology Software and does not apply to the Sterling Commerce Software, or any other Third Party Software.

## Rico Software

The Sterling Commerce Software is also distributed with or on the same storage media as the Rico.js software (Copyright © 2005 Sabre Airline Solutions) ("Rico Software"). Rico Software is free software

which is distributed under the terms of the Apache License Version 2.0. A copy of License Version 2.0 is found in <install>/repository/eardata/platform/war/ajax/scripts/Rico_License.txt.

The Rico Software was not modified. Neither the Sterling Commerce Software, modifications, if any, to the Rico Software, nor other Third-Party Code is a Derivative Work or a Contribution as defined in License Version 2.0. License Version 2.0 applies only to the Rico Software which is the subject of the specific directory file and does not apply to the Sterling Commerce Software or to any other Third-Party Software. License Version 2.0 includes the following provision:

"Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License."

## Rhino Software

The Sterling Commerce Software is distributed with or on the same storage media as the Rhino js.jar (Copyright (c) 1998-2009 Mozilla.org.) ("Rhino Software"). A majority of the source code for the Rhino Software is dual licensed under the terms of the MOZILLA PUBLIC LICENSE Version 1.1. or the GPL v. 2.0. Additionally, some files (at a minimum the contents of toolsrc/org/Mozilla/javascript/toolsdebugger/treetable) are available under another license as set forth in the directory file for the Rhino Software.

Sterling Commerce's use and distribution of the Rhino Software is under the Mozilla Public License. A copy of this license is located at <install>/3rdParty/rico license.doc. The Rhino Software code is distributed in source form and is located at http://mxr.mozilla.org/mozilla/source/js/rhino/src/. Neither the Sterling Commerce Software nor any other Third-Party Code is a Modification or Contribution subject to the Mozilla Public License. Pursuant to the terms of the Mozilla Public License, the following notice applies only to the Rhino Software (and not to the Sterling Commerce Software or any other Third-Party Software):

"The contents of the file located at <install>/jar/rhino/1_7R1/js.jar are subject to the Mozilla Public License Version 1.1 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/.

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

The Original Code is Rhino code, released May 6, 1999. The Initial Developer is Netscape Communications Corporation. Portions created by the Initial Developer are Copyright © 1997-1999. All Rights Reserved. Contributor(s):_____none listed.

The preceding license only applies to the Rico Software and does not apply to the Sterling Commerce Software, or any other Third-Party Software.

## Sun Microsystems

The Sterling Commerce Software is distributed with or on the same storage media

as the following software products (or components thereof): Sun JMX, and Sun JavaMail (collectively, "Sun Software"). Sun Software is free software which is distributed under the terms of the licenses issued by Sun which are included in the directory files located at:

SUN COMM JAR - <install>/Applications/Foundation/lib

SUN ACTIVATION JAR - <install>/ Applications/Foundation/lib

SUN JavaMail - <install>/jar/javamail/1_4/LICENSE.txt

The Sterling Commerce Software is also distributed with or on the same storage media as the Web-app_2_3.dtd software (Copyright © 2007 Sun Microsystems, Inc.) ("Web-App Software"). Web-App Software is free software which is distributed under the terms of the Common Development

and Distribution License ("CDDL"). A copy of the CDDL is found in http://kenai.com/projects/javamail/sources/mercurial/show.

The source code for the Web-App Software may be found at: <install>/3rdParty/sun/javamail-1.3.2/docs/JavaMail-1.2.pdf

Such licenses only apply to the Sun product which is the subject of such directory and does not apply to the Sterling Commerce Software or to any other Third Party Software.

The Sterling Commerce Software is also distributed with or on the same storage media as the Sun Microsystems, Inc. Java (TM) look and feel Graphics Repository ("Sun Graphics Artwork"), subject to the following terms and conditions:

Copyright 2000 by Sun Microsystems, Inc. All Rights Reserved.

Sun grants you ("Licensee") a non-exclusive, royalty free, license to use, and redistribute this software graphics artwork, as individual graphics or as a collection, as part of software code or programs that you develop, provided that i) this copyright notice and license accompany the software graphics artwork; and ii) you do not utilize the software graphics artwork in a manner which is disparaging to Sun. Unless enforcement is prohibited by applicable law, you may not modify the graphics, and must use them true to color and unmodified in every way.

This software graphics artwork is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE GRAPHICS ARTWORK.

IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE GRAPHICS ARTWORK, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

If any of the above provisions are held to be in violation of applicable law, void, or unenforceable in any jurisdiction, then such provisions are waived to the extent necessary for this Disclaimer to be otherwise enforceable in such jurisdiction.

The preceding license only applies to the Sun Graphics Artwork and does not apply to the Sterling Commerce Software, or any other Third Party Software.

## WARRANTY DISCLAIMER

This documentation and the Sterling Commerce Software which it describes are licensed either "AS IS" or with a limited warranty, as set forth in the Sterling Commerce license agreement. Other than any limited warranties provided, NO OTHER WARRANTY IS EXPRESSED AND NONE SHALL BE IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE OR FOR A PARTICULAR PURPOSE. The applicable Sterling Commerce entity reserves the right to revise this publication from time to time and to make changes in the content hereof without the obligation to notify any person or entity of such revisions or changes.

The Third Party Software is provided "AS IS" WITHOUT ANY WARRANTY AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. FURTHER, IF YOU ARE LOCATED OR ACCESSING THIS SOFTWARE IN THE UNITED STATES, ANY EXPRESS OR IMPLIED WARRANTY REGARDING TITLE OR NON-INFRINGEMENT ARE DISCLAIMED.

Without limiting the foregoing, the ICE Software and JBoss Software are distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

# Contents

## 8 Creating and Adding Wizards to Rich Client Platform Applications

## 17 Rich Client Platform General Concepts Reference

## Index

# Preface

This manual provides a brief glimpse into the Rich Client Platform and explains how to customize the different components of a Rich Client Platform application.

## Intended Audience

This manual is intended for use by those who are responsible for customizing Selling and Fulfillment Foundation.

## Structure

This document contains the following chapters:

**Chapter 1, "Checklist for Customization Projects"**

This chapter describes a checklist of the tasks you need to perform to customize the different components of Selling and Fulfillment Foundation.

**Chapter 2, "Rich Client Platform"**

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

**Chapter 3, "The Development Environment for Rich Client Platform Applications"**

This chapter describes the various software components required to customize the Rich Client Platform application.

### Chapter 4, "Customizing Rich Client Platform Application"

This chapter describes various ways of customizing a Rich Client Platform application.

### Chapter 5, "Customizing the About Box"

This chapter describes how to customize the About Box of a Rich Client Platform application.

### Chapter 6, "Modifying the Existing Rich Client Platform Screens and Wizards"

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

### Chapter 7, "Creating and Adding Screens to Rich Client Platform Applications"

This chapter explains how to create Rich Client Platform screens.

### Chapter 8, "Creating and Adding Wizards to Rich Client Platform Applications"

This chapter explains how to create and add wizards to Rich Client Platform applications.

### Chapter 9, "Creating Related Tasks for Rich Client Platform Applications"

This chapter explains how to create related tasks for Rich Client Platform applications.

### Chapter 10, "Creating Commands for Rich Client Platform Applications"

This chapter explains how to create commands to call APIs or services to retrieve data.

### Chapter 11, "Defining and Overriding Hot Keys in Rich Client Platform Applications"

This chapter explains how to define new hot keys for new screens, and override the hot keys defined for the existing screens.

### Chapter 12, "Merging Templates for Rich Client Platform Applications"

This chapter explains how to merge the input and output templates to get additional data from an API or Service.

### Chapter 13, "Related and Shared Tasks in Rich Client Platform Applications"

This chapter explains how to add new related tasks and shared tasks to the Rich Client Platform application.

### Chapter 14, "Defining Themes for Rich Client Platform Applications"

This chapter explains how to define a new theme for theming the Rich Client Platform application.

### Chapter 15, "Menus and Custom Controls for Rich Client Platform Applications"

This chapter explains how to add menus and custom controls for Rich Client Platform applications.

### Chapter 16, "Setting the Extension Model, Configuring SSL and SSO for Rich Client Platform Applications"

This chapter explains how to set the extension model to populate the newly added fields on the form with the required data. It also describes how to configure SSL for Rich Client Platform applications.

### Chapter 17, "Rich Client Platform General Concepts Reference"

This chapter provides information about general Rich Client Platform concepts.

## Selling and Fulfillment Foundation Documentation

For more information about the Selling and Fulfillment Foundation components, see the following manuals:

- *Selling and Fulfillment Foundation: Release Notes*
- *Selling and Fulfillment Foundation: Installation Guide*
- *Selling and Fulfillment Foundation: Upgrade Guide*

- *Selling and Fulfillment Foundation: Configuration Deployment Tool Guide*
- *Selling and Fulfillment Foundation: Performance Management Guide*
- *Selling and Fulfillment Foundation: High Availability Guide*
- *Selling and Fulfillment Foundation: System Management Guide*
- *Selling and Fulfillment Foundation: Localization Guide*
- *Selling and Fulfillment Foundation: Customization Basics Guide*
- *Selling and Fulfillment Foundation: Customizing APIs Guide*
- *Selling and Fulfillment Foundation: Customizing Console JSP Interface for End User Guide*
- *Selling and Fulfillment Foundation: Customizing the RCP Interface Guide*
- *Selling and Fulfillment Foundation: Customizing User Interfaces for Mobile Devices Guide*
- *Selling and Fulfillment Foundation: Customizing Web UI Framework Guide*
- *Selling and Fulfillment Foundation: Customizing Swing Interface Guide*
- *Selling and Fulfillment Foundation: Extending the Condition Builder Guide*
- *Selling and Fulfillment Foundation: Extending the Database Guide*
- *Selling and Fulfillment Foundation: Extending Transactions Guide*
- *Selling and Fulfillment Foundation: Using Sterling RCP Extensibility Tool Guide*
- *Selling and Fulfillment Foundation: Integration Guide*
- *Selling and Fulfillment Foundation: Product Concepts Guide*
- *Sterling Warehouse Management™ System: Concepts Guide*
- *Selling and Fulfillment Foundation: Application Platform Configuration Guide*
- *Sterling Distributed Order Management™: Configuration Guide*

- *Sterling Supply Collaboration: Configuration Guide*
- *Sterling Global Inventory Visibility™: Configuration Guide*
- *Catalog Management™: Configuration Guide*
- *Sterling Logistics Management: Configuration Guide*
- *Sterling Reverse Logistics™: Configuration Guide*
- *Sterling Warehouse Management System: Configuration Guide*
- *Selling and Fulfillment Foundation: Application Platform User Guide*
- *Sterling Distributed Order Management: User Guide*
- *Sterling Supply Collaboration: User Guide*
- *Sterling Global Inventory Visibility: User Guide*
- *Sterling Logistics Management: User Guide*
- *Sterling Reverse Logistics: User Guide*
- *Sterling Warehouse Management System: User Guide*
- *Selling and Fulfillment Foundation: Mobile Application User Guide*
- *Selling and Fulfillment Foundation: Business Intelligence Guide*
- *Selling and Fulfillment Foundation: Javadocs*
- *Sterling Selling and Fulfillment Suite™: Glossary*
- *Parcel Carrier: Adapter Guide*
- *Selling and Fulfillment Foundation: Multitenant Enterprise Guide*
- *Selling and Fulfillment Foundation: Password Policy Management Guide*
- *Selling and Fulfillment Foundation: Properties Guide*
- *Selling and Fulfillment Foundation: Catalog Management Concepts Guide*
- *Selling and Fulfillment Foundation: Pricing Concepts Guide*
- *Business Center: Item Administration Guide*
- *Business Center: Pricing Administration Guide*
- *Business Center: Customization Guide*

- *Business Center: Localization Guide*

# Conventions

In this manual, Windows refers to all supported Windows operating systems.

The following conventions may be used in this manual:

| Convention | Meaning |
|---|---|
| . . . | Ellipsis represents information that has been omitted. |
| < > | Angle brackets indicate user-supplied input. |
| `mono-spaced text` | Mono-spaced text indicates a file name, directory path, attribute name, or an inline code example or command. |
| / or \ | Slashes and backslashes are file separators for Windows, UNIX, and Linux operating systems. The file separator for the Windows operating system is "\" and the file separator for UNIX and Linux systems is "/". The UNIX convention is used unless otherwise mentioned. |
| <INSTALL_DIR> | User-supplied location of the Selling and Fulfillment Foundation installation directory. This is only applicable for Release 8.0 or later. |
| <INSTALL_DIR_OLD> | User-supplied location of the Selling and Fulfillment Foundation installation directory (for Release 8.0 or later).<br>**Note:** This is applicable only for users upgrading from Release 8.0 or later. |
| <YANTRA_HOME> | User-supplied location of the Sterling Supply Chain Applications installation directory. This is only applicable for Releases 7.7, 7.9, and 7.11. |
| <YANTRA_HOME_OLD> | User-supplied location of the Sterling Supply Chain Applications installation directory (for Releases 7.7, 7.9, or 7.11).<br>**Note:** This is applicable only for users upgrading from Releases 7.7, 7.9, or 7.11. |

| Convention | Meaning |
|---|---|
| <YFS_HOME> | For Releases 7.3, 7.5, and 7.5 SP1, this is the user-supplied location of the Sterling Supply Chain Applications installation directory.<br>For Releases 7.7, 7.9, and 7.11, this is the user-supplied location of the <YANTRA_HOME>/Runtime directory.<br>For Release 8.0 or above, the <YANTRA_HOME>/Runtime directory is no longer used and this is the same location as <INSTALL_DIR>. |
| <YFS_HOME_OLD> | This is the <YANTRA_HOME>/Runtime directory for Releases 7.7, 7.9, or 7.11.<br>**Note:** This is only applicable for users upgrading from Releases 7.7, 7.9, or 7.11. |
| <ANALYTICS_HOME> | User-supplied location of the Sterling Analytics installation directory.<br>**Note:** This convention is used only in the *Selling and Fulfillment Foundation: Business Intelligence Guide*. |
| <COGNOS_HOME> | User-supplied location of the IBM Cognos 8 Business Intelligence installation directory.<br>**Note:** This convention is used only in the *Selling and Fulfillment Foundation: Business Intelligence Guide*. |
| <MQ_JAVA_INSTALL_PATH> | User-supplied location of the IBM WebSphere® MQ Java components installation directory.<br>**Note:** This convention is used only in the *Selling and Fulfillment Foundation: System Manangement and Administration Guide*. |
| <DB> | Refers to Oracle®, IBM DB2®, or Microsoft SQL Server® depending on the database server. |
| <DB_TYPE> | Depending on the database used, considers the value oracle, db2, or sqlserver. |

**Note:** The Selling and Fulfillment Foundation documentation set uses the following conventions in the context of the product name:

- Yantra is used for Release 7.7 and earlier.
- Sterling Supply Chain Applications is used for Releases 7.9 and 7.11.
- Sterling Multi-Channel Fulfillment Solution is used for Releases 8.0 and 8.2.
- Selling and Fulfillment Foundation is used for Release 8.5.

# 1

# Checklist for Customization Projects

This chapter provides a high-level checklist for the tasks involved in customizing or extending Selling and Fulfillment Foundation.

## 1.1 Customization Projects

Projects to customize or extend Selling and Fulfillment Foundation 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.

1. Prepare your development environment

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

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

   An option is to customize Selling and Fulfillment Foundation with Microsoft COM+. Using COM+ provides you with 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 *Selling and Fulfillment Foundation: Customization Basics Guide* about additional installation instructions.

2. Plan your customizations

   Are you adding a new menu entry, customizing the Sign In screen and logo, creating new themes, customizing views and wizards, or adding new screens? Each type of customization varies in scope and complexity. For background, see the *Selling and Fulfillment Foundation: Customization Basics Guide*, which summarizes the types of changes that you can make.

   Important guidelines about file names, keywords, and other conventions are found in the *Selling and Fulfillment Foundation: Customization Basics Guide*.

3. 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 *Selling and Fulfillment Foundation: Extending the Database Guide* which include information about the following topics:

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

4. Make other changes to APIs

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

- How to invoke standard APIs for displaying data in the UI and also how to save the changes made to the UI in 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 Application XMLs, you may get inconsistent data.

5. Customize the UI

   Sterling Commerce 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 like menu items, logos, themes, and etc. Depending on the framework you want, consult one of the following guides:

   - *Selling and Fulfillment Foundation: Customizing Console JSP Interface for End User Guide*
   - *Selling and Fulfillment Foundation: Customizing the Swing Interface Guide*
   - *Selling and Fulfillment Foundation: Customizing User Interfaces for Mobile Devices Guide*
   - *Selling and Fulfillment Foundation: Customizing the RCP Interface Guide* and *Selling and Fulfillment Foundation: Using the Sterling RCP Extensibility Tool Guide*
   - *Customizing the Web UI Framework Guide*

6. Extend Transactions

   You can extend the standard Selling and Fulfillment Foundation to enhance the functionality of your implementation of Selling and Fulfillment Foundation and to integrate with external systems. For background about transaction types, security, dynamic variables, and extending the

Condition Builder, see the *Selling and Fulfillment Foundation: Extending Transactions Guide Selling and Fulfillment Foundation: Extending the Condition Builder Guide* . These guides includes information about the following types of changes:

- How to extend Selling and Fulfillment Foundation Condition Builder to define complex and dynamic conditions for executing your custom business logic and using a static set of attributes.
- How to define variables to dynamically configure properties belonging to actions, agents, and services configurations.
- How to 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.
- How to create custom time-triggered transactions. You can invoke and schedule these custom time-triggered transactions in much the same manner as you invoke and schedule Selling and Fulfillment Foundation standard time-triggered transactions. Finally, you can 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.

7. Build and deploy your customizations or extensions

   After performing the customizations that you want, you must build and deploy your customizations or extensions. First, build and deploy these customizations or extensions in the test environment for verification. When you are ready, repeat the same process to build and deploy your customizations and extensions in the production environment. For instructions, see the *Selling and Fulfillment Foundation: Customization Basics Guide* which includes information about the following topics:

   - How to build and deploy standard resources, database, and other extensions (such as templates, user exits, java interfaces).
   - How to build and deploy Enterprise-Level extensions.

# 2

# Rich Client Platform

## 2.1 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.

### 2.1.1 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.

### 2.1.2 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. For more information about extensibility capabilities, see Section 2.1.5, "Extending Rich Client Platform Applications".

### 2.1.3 Guidelines for Smooth Updates and Easy Maintenance

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

- Plug-in files
- Selling and Fulfillment Foundation-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.

### 2.1.4 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 Chapter 3, "The Development Environment for Rich Client Platform Applications".

### 2.1.5 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 Selling and Fulfillment Foundation. For more information about adding or removing menus, see Chapter 15, "Menus and Custom Controls for Rich Client Platform Applications".
- 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 Section 4.1.3.3, "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 Chapter 9, "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 Chapter 6, "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 Section 6.3, "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 Chapter 4, "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 Chapter 14, "Defining Themes for Rich Client Platform Applications".

## 2.2 Rich Client Platform Architecture

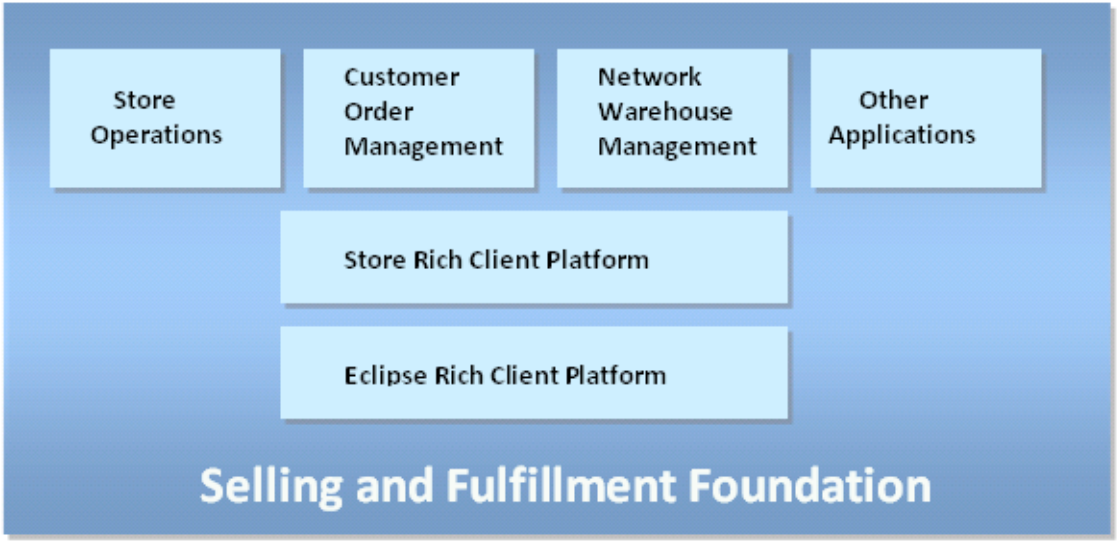
The Rich Client Platform provides a highly interactive Rich Client Platform, which can be remotely deployed, updated, and easily managed. A Rich Client Platform is a client that processes the bulk of data operations without depending on the server to which it is connected. However, it is dependent on 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 UIs for the Selling and Fulfillment Foundation Package Composite

Applications (PCAs), such as Selling and Fulfillment Foundation Store Operations (SOP) and Selling and Fulfillment Foundation Sterling Call Center and Sterling Store (SCCS), are developed using the Rich Client Platform. Figure 2–1 depicts the Rich Client Platform architecture.

***Figure 2–1*** **Rich Client Platform Architecture**




## 2.3 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 the Selling and Fulfillment Foundation 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 *Selling and Fulfillment Foundation: Installation Guide*.
  - Subsequent upgrades can be automated with the Selling and Fulfillment Foundation 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.

## 2.4 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 include Sterling COM PCA and Sterling SOP PCA, which are developed using the Rich Client Platform.

## 2.5 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 Section 7.11.1, "Binding Classes".

## 2.6 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 *Selling and Fulfillment Foundation: Localization Guide*.

### 2.6.1 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 *Selling and Fulfillment Foundation: Localization Guide*.

## 2.7 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 Chapter 14, "Defining Themes for Rich Client Platform Applications".

## 2.8 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 Selling and Fulfillment Foundation-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 Chapter 13, "Related and Shared Tasks in Rich Client Platform Applications".

## 2.9 Shared Tasks for Rich Client Platform Applications

Rich Client Platform-based applications such as Sterling COM PCA 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 Section 13.3, "Registering Shared Tasks" and Section 13.4, "Using Shared Tasks".

## 2.10 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".

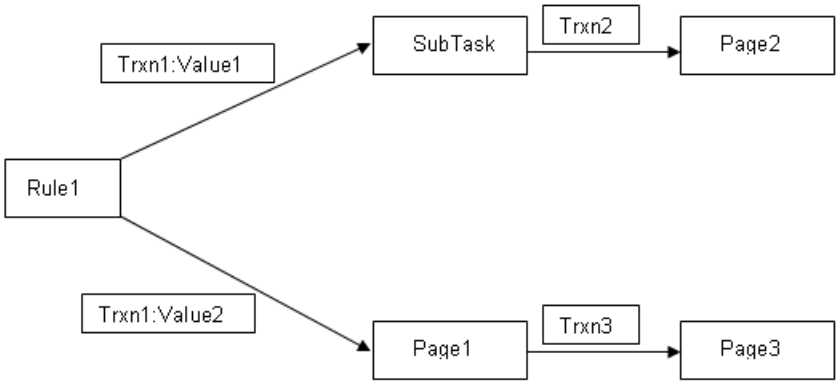
You can add transition lines to transfer the control from one wizard entity to another wizard entity based on the output values of the wizard rule. The transitions originating from a wizard page can have only one target. Transitions starting from a wizard rule can have multiple targets based on the rule 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 2–2 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.

Selling and Fulfillment Foundation 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).

## 2.11 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 Chapter 11, "Defining and Overriding Hot Keys in Rich Client Platform Applications".

## 2.12 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.

## 2.12.1 Running Rich Client Platform Applications in Debug Mode

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. This section explains the following:

- Running the Standalone Rich Client Platform Application in Debug Mode
- Running the Rich Client Platform Application in Eclipse in Debug Mode

## 2.12.2 Running the Standalone Rich Client Platform Application in Debug Mode

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:

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 COM PCA 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.

### 2.12.3 Running the Rich Client Platform Application in Eclipse in Debug Mode

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

```
-Ddebugmode=true
```

For more information about launching the Rich Client Platform application in Eclipse, see Section 3.5, "Launching the Rich Client Platform Application in Eclipse".

## 2.13 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.

### 2.13.1 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.

### 2.13.2 Running Standalone Rich Client Platform Applications in Prototype Mode

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:

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 COM PCA 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 Chapter 10, "Creating Commands for Rich Client Platform Applications".

   **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.

## 2.13.3 Running Rich Client Platform Applications in Eclipse in Prototype Mode

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.

# 2.14 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.

- *Selling and Fulfillment Foundation: Javadocs*

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

## 2.14.1 Tracing a Standalone Rich Client Platform Application

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:

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 COM PCA, 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.

***Figure 2–3 Application Title Bar***


> **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.
>
> Sterling Commerce 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.

## 2.14.2 Masking Sensitive Information in Logs 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. For information on registering message filters, refer to the Section 17.14.6, "Registering a Message Filter" .

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.
3. 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.

## 2.14.3 Tracing a Rich Client Platform Application in Eclipse

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.

## 2.15 Capitalizing the Text Entered in Rich Client Platform Applications

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:

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 COM PCA, 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
   ```

## 2.16 Fetching Images for Rich Client Platform Applications

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 the *Selling and Fulfillment Foundation: Installation Guide*.

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

## 2.17 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 *Selling and Fulfillment Foundation: Installation Guide*.

## 2.18 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 Chapter 12, "Merging Templates for Rich Client Platform Applications".

## 2.19 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 Chapter 10, "Creating Commands for Rich Client Platform Applications".

## 2.20 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 COM PCA, 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.

For more information about tracing an application, see Section 2.14, "Tracing a Rich Client Platform Application".

### 2.20.1 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.

## 2.21 Table Filtering for Rich Client Platform Applications

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:

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.

### 2.21.1 Clearing the Sort Order in a Table

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.

## 2.22 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

### 2.22.1 Scheduling a Generic Job

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

To create and register a generic job:

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`

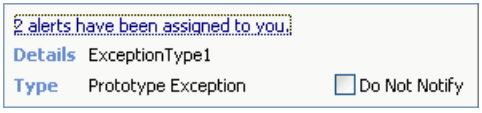
     By default, idle time is set to 5 minutes.

   - 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.

## 2.22.2 Scheduling an Alert-Related Job

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 a panel every two minutes when an alert is assigned to the user who has logged in.

***Figure 2–4 Alert Pop-up Window***



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:

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.

## 2.22.3 Preventing the Deactivation of Alert Notification

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:

1. A static utility method is added in the *YRCAlertMessageController* class with the following format:

   ```
   public static void hideDoNotNotifyCheckbox(true)
   ```

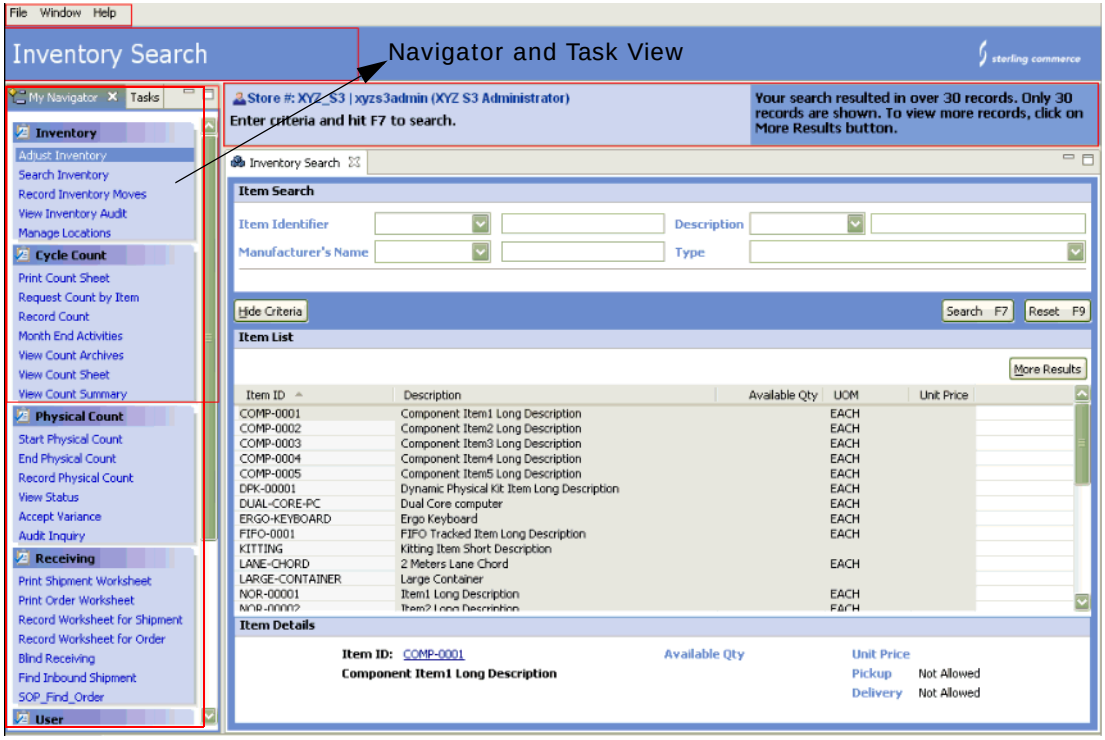
2. Set the flag to true to hide the check box.

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.

## 2.23 Low Resolution Display for Rich Client Platform Applications

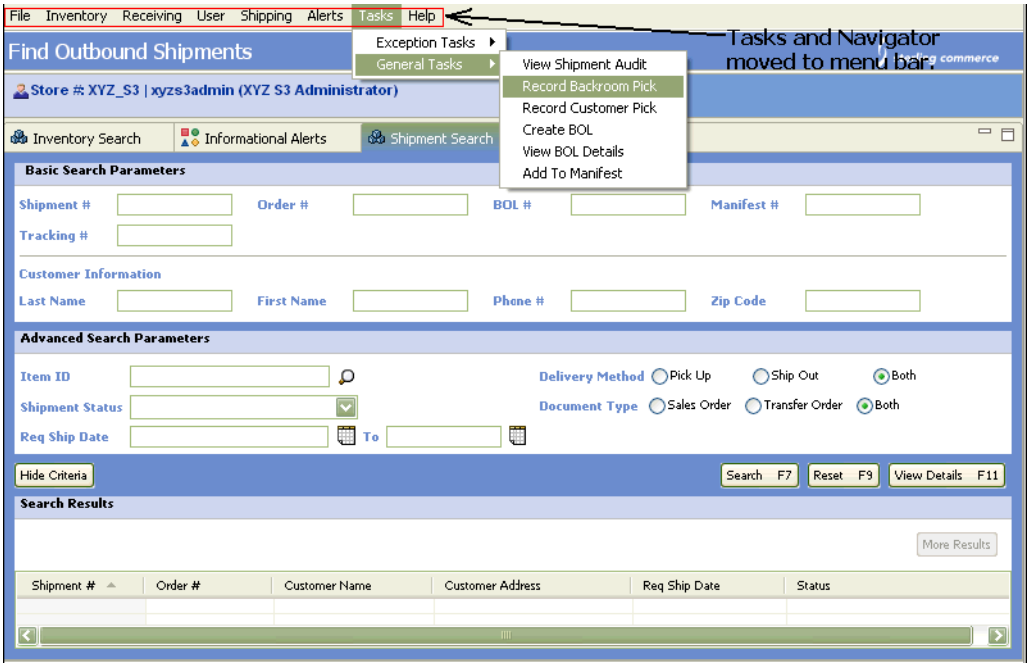
Figure 2–5 depicts one of the Rich Client Platform PCA applications that displays on a screen for which the system resolution is set to greater than 800 X 600 pixels.

*Figure 2–5 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) will 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. Figure 2–6 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.

*Figure 2–6 Low Resolution Display*




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

In high resolution, the Navigator tasks are displayed in a panel on the 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:

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 the Sterling COM PCA 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.

## 2.25 Switching Locale for Rich Client Platform Applications

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

To switch the locale, you need to pass the locale code as -Dlocalecode=<LOCALE_CODE>.

> **Note:** If the passed locale is not defined on the server, in such case the system locale is used.

## 2.26 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.

# 2.27 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 for Sterling COM PCA 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>
```

# 2.28 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.

## 2.28.1 Using the Pop-Up Method

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:

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.

### 2.28.2 Starting a Supervisory Transaction

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:

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.

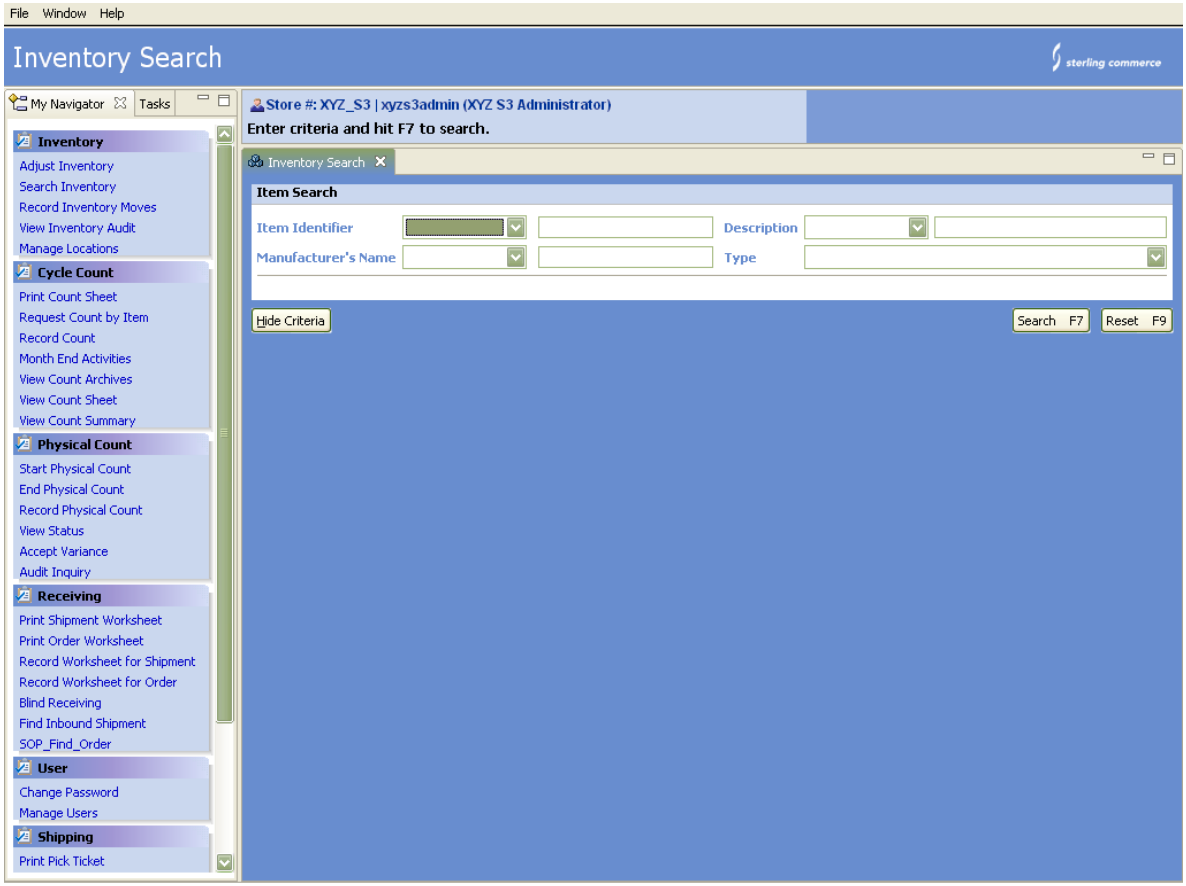
## 2.29 Running Rich Client Platform Applications in POS Mode

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.

Figure 2–7 depicts the application layout in POS mode.

***Figure 2–7 POS Mode Layout***

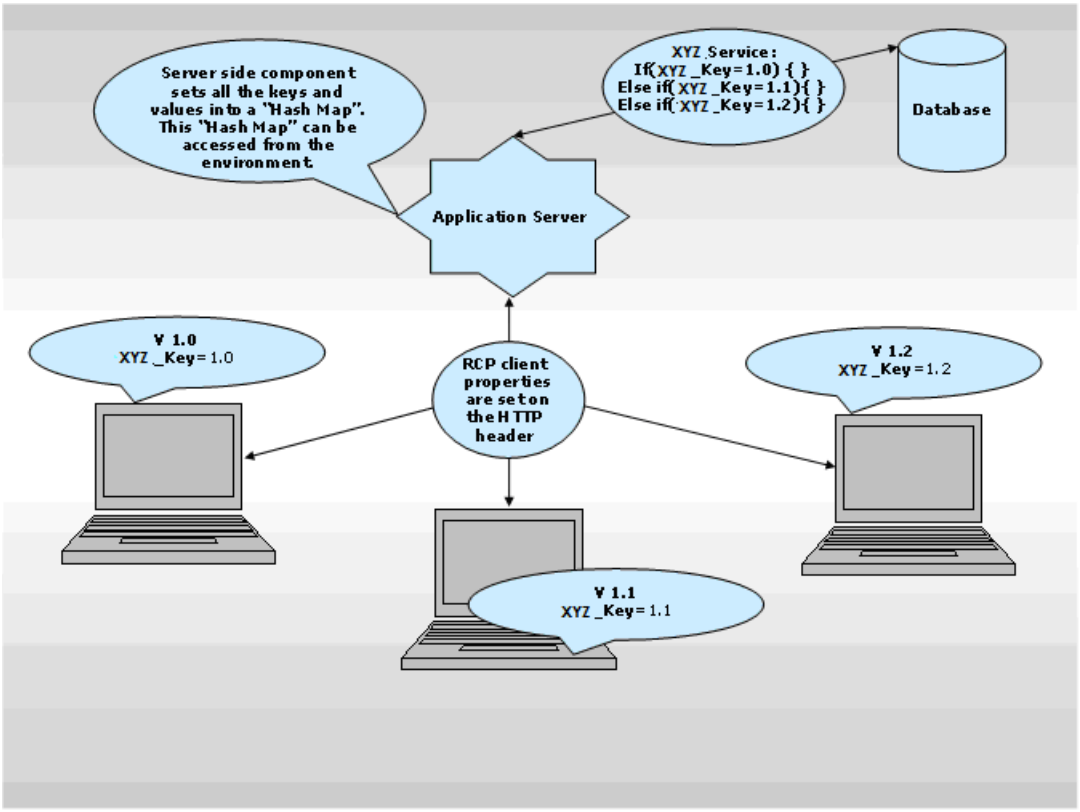


# 2.30 Version-Based Communication between Client and Server

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.

Figure 6–5 depicts the layout of version-based communication between client and server.

*Figure 2–8 Layout of the client-server communication process*




- 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.

> **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.

## 2.30.1 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.

## 2.30.2 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 Platform version of the client connecting to it:

- A utility class file `YRCCommandsMergeUtils.java` is added to 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.

## 2.31 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 applications 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, appication 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"
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`

To create an extension:

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)
  ```

# 3

# The Development Environment for Rich Client Platform Applications

## 3.1 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. For more information about the Eclipse SDK version, see the *Selling and Fulfillment Foundation: Installation Guide*.

- Eclipse-related Plug-ins

  Install the following Eclipse-related plug-ins that the Rich Client Platform supports. For more information about the Eclipse plug-in versions, see the *Selling and Fulfillment Foundation: Installation Guide*.

  - VE (Visual Editor) plug-in
  - GEF (Graphical Editor Framework) plug-in
  - EMF (Eclipse Modelling Framework) plug-in

- Java Development Kit (JDK)

  Install the JDK version that the Rich Client Platform supports. For more information about the JDK version, see the *Selling and Fulfillment Foundation: Installation Guide*.

- 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 *Selling and Fulfillment Foundation: Installation Guide*.

- Rich Client Platform plug-in
- Rich Client Platform Tools plug-in

These plug-ins are shipped along with Selling and Fulfillment Foundation and are located in the following directory:

`<INSTALL_DIR>/rcpclient`

**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.

To clean this information, start the Eclipse SDK with the "-clean" option:

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 3.2\eclipse\eclipse.exe" -clean`.
3. Start the Eclipse SDK.

## 3.1.1 Installing the Rich Client Platform Plug-In

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, Rich Client Platform plug-in, and Rich Client Platform Search List Composite. For details, see Section 3.1.3, "Rich Client Platform Tools" .
- 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. For more information on Extensibility tool plug-ins, refer to the section Section 3.5, "Launching the Rich Client Platform Application in Eclipse".

## 3.1.2 Installing the Rich Client Platform Tools Plug-In

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.

## 3.1.3 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.
>
> To view the cheat sheets in Eclipse, follow these steps:
>
> 1. Start the Eclipse SDK.
> 2. From the menu bar select Help > Cheat Sheets. The Cheat Sheet Selection window displays.
> 3. Expand "Rich Client Platform: UI Editor" from the list and open the appropriate cheat sheet.

- 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 Selling and Fulfillment Foundation-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 Search List Composite—The Rich Client Platform Search List Composite wizard is used to create the sample Search and List screen for the Rich Client Platform application. The Rich Client Platform Search and List screens are divided into two panels: Search panel and List panel. The Search screen accepts the search criteria entered by a user and performs the search operation. The List screen displays the results of the search operation.

**Note:** To open the UI wizards:

1. Launch the Rich Client Platform application in Eclipse. For more information, see Section 3.5, "Launching the Rich Client Platform Application in Eclipse".
2. Press Ctrl+N.
3. Expand Rich Client Platform Wizards from the list of wizards and open the appropriate wizard.

- 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—Unzip the Rich Client Platform application zip file that you want to customize to any directory. You can find the zip file for a PCA in the following directory:

  <INSTALL_DIR>/rcp/<PCA_DIR>/rcpclient/

  Here, <PCA_NAME> refers to the PCA installation directory.

  For example, if you want to customize the Sterling COM PCA, unzip 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 COM PCA, copy the: <TEMP_UNZIP_DIR>/plugins/com.yantra.pca.ycd.rcp_<version> to the <ECLIPSE_HOME>/plugins folder.

  where <TEMP_UNZIP_DIR> is the name of the directory where you have extracted the com.zip files. <ECLIPSE_HOME> refers to the directory where you have installed Eclipse SDK.

## 3.2 Creating and Configuring Locations

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

To configure a location, follow these steps:

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 *Selling and Fulfillment Foundation: Installation 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.

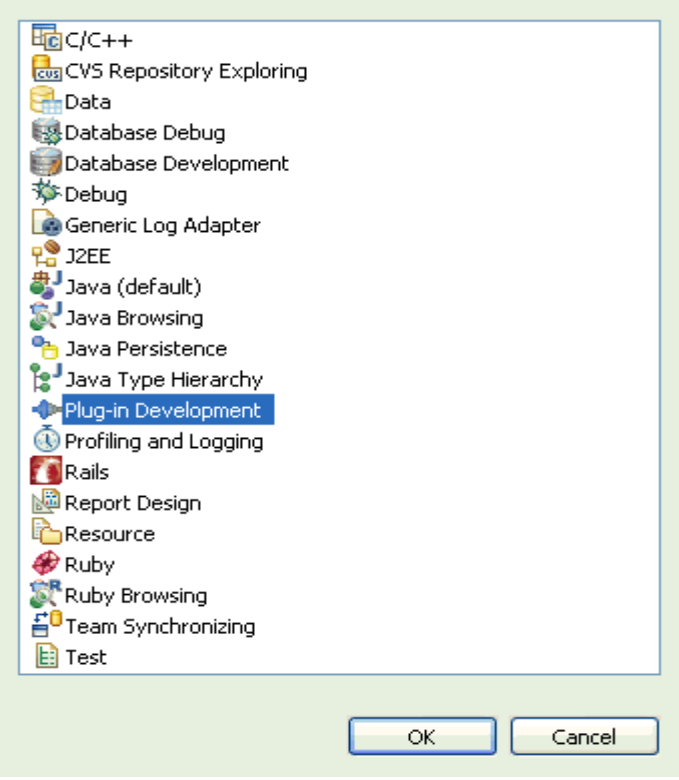
## 3.3 Creating a Plug-In Project

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

To create a plug-in project:

1. Start the Eclipse SDK.
2. From the menu bar, select Window > Open Perspective > Other.... The Select Perspective window displays.

***Figure 3–1 Select Perspective Window***




3. From the list of wizards, select Plug-in Development.
4. Click OK. The Eclipse Workbench opens in Java perspective. For more information about the Eclipse workbench, see Appendix 17.3, "Workbench".
5. From the menu bar, select File > 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.

***Figure 3–2 New Plug-in Project Window***

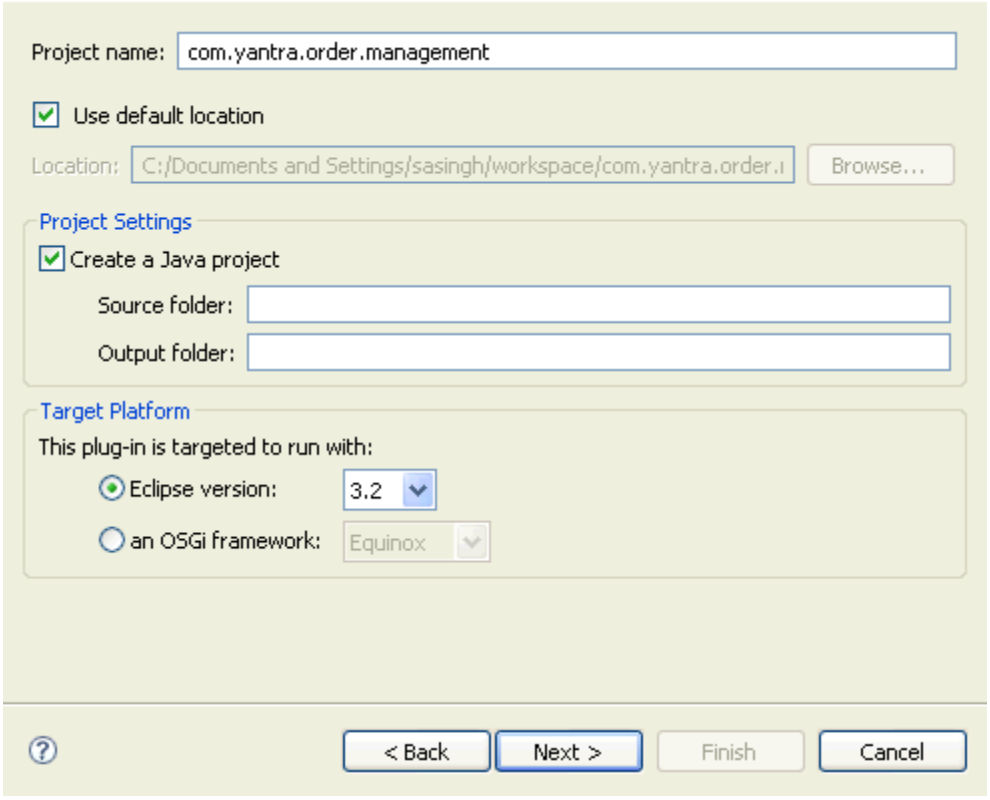



***Table 3–1 New Plug-in Project Window***

| 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.<br>Sterling Commerce 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. | |

*Table 3–1 New Plug-in Project Window*

| Field | Description |
|---|---|
| Target Platform | |
| Eclipse version: | Select 3.2 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.

## 3.4 Running the 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.

To run the Rich Client Platform Plug-in wizard:

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. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.

*Figure 3–3 New Plugin to Rich Client Platform UI Window*

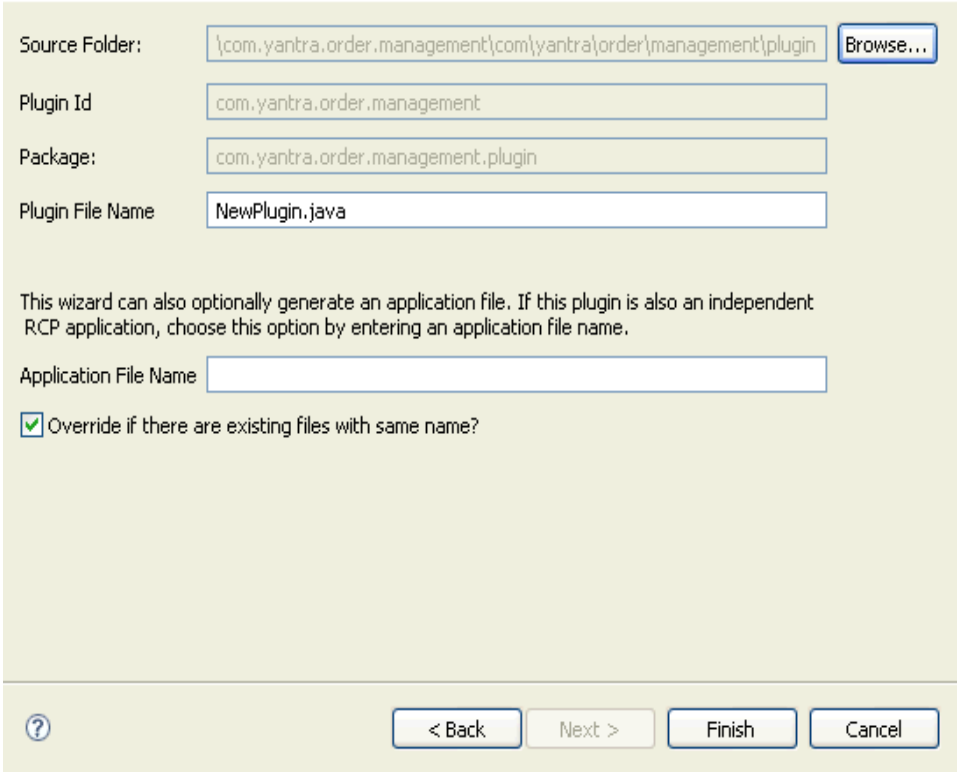



*Table 3–2 New Plugin to Rich Client Platform UI Window*

| 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. |

*Table 3–2 New Plugin to Rich Client Platform UI Window*

| Field | Description |
|---|---|
| Package: | The package name displays. If this field is empty, the system considers the source folder as the default package.<br>**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.

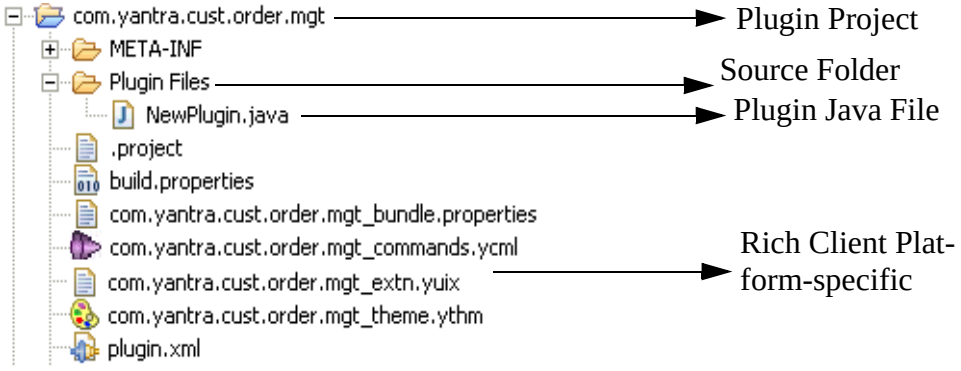
After you run the Rich Client Platform Plugin wizard on top of a plug-in project, the Rich Client Platform performs the following tasks:

- 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. For more information about the YRCPluginAutoLoader extension point, see Appendix 17.5, "YRCPluginAutoLoader Extension Point".
- 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`

Figure 3–4 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.

***Figure 3–4 Typical Folder Structure in Eclipse***



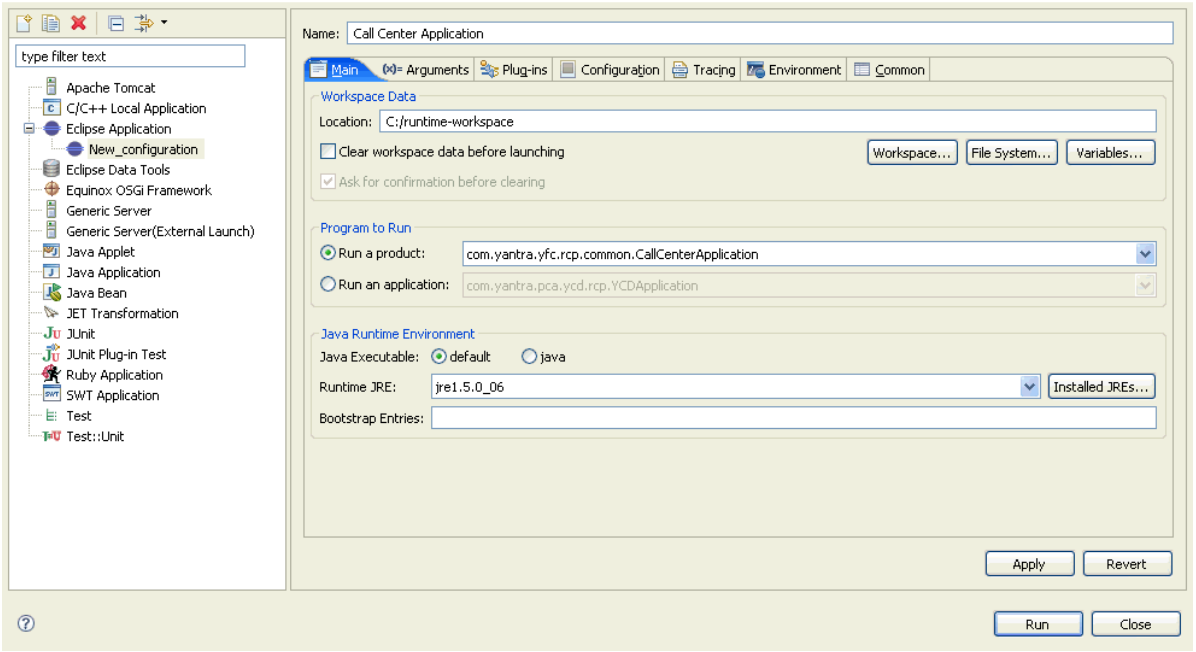

# 3.5 Launching the Rich Client Platform Application in Eclipse

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:

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.

***Figure 3–5 Run Window***




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 the Sterling COM PCA, select the product identifier of the Sterling COM PCA. For more information about the product identifier for each PCA, see the respective PCA 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 that you selected in Step 6 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 as Sterling COM PCA 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.

# 4

# Customizing Rich Client Platform Application

## 4.1 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 Selling and Fulfillment Foundation-specific resource files that are shipped with Selling and Fulfillment Foundation.

### 4.1.1 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 *Selling and Fulfillment Foundation: Localization Guide*.

### 4.1.2 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 Chapter 14, "Defining Themes for Rich Client Platform Applications".

## 4.1.3 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.

Sterling Commerce 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 Chapter 3, "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).

### 4.1.3.1 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 Chapter 6, "Modifying the Existing Rich Client Platform Screens and Wizards".

#### 4.1.3.2 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. For more information about modifying an existing wizard, see Section 6.3, "Modifying Existing Rich Client Platform Wizards".

#### 4.1.3.3 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). For more information about creating new screens, see Section 7, "Creating and Adding Screens to Rich Client Platform Applications".

After creating new screens, you can add these to the Rich Client Platform application. For more information about adding new screens to a Rich Client Platform application, see Chapter 7, "Creating and Adding Screens to Rich Client Platform Applications".

## 4.2 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.

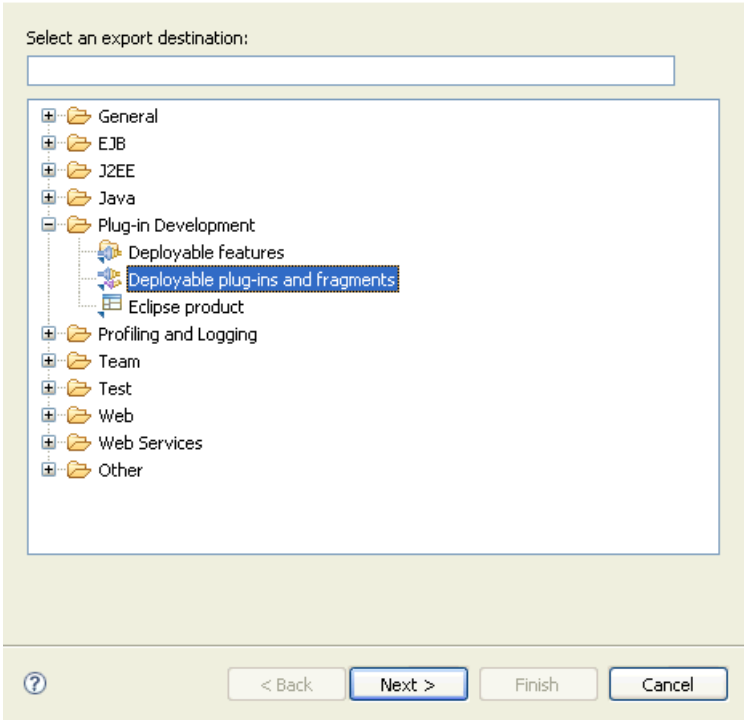
### 4.2.1 Building Rich Client Platform Extensions

Building the Rich Client Platform extensions is as follows:

1. Start the Eclipse SDK.

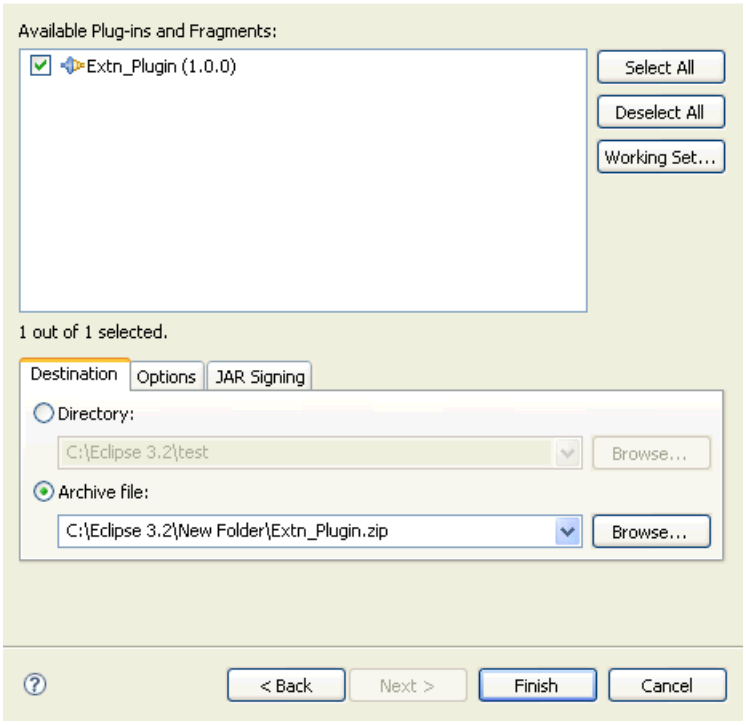
2. From the menu bar, select Window > Show View > Navigator. The plug-in project is displayed in the Navigator view.
3. Right-click on the plug-in project that you want to build and deploy. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
4. Select Export... from the pop-up menu. The Export window displays.

***Figure 4–1 Export Window***




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

*Figure 4–2 Available Plug-ins Window*




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 zip 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 zip file as specified in Step 8.

## 4.2.2 Deploying Rich Client Platform Extensions

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 the *Selling and Fulfillment Foundation: Installation Guide*.

# 5

# Customizing the About Box

## 5.1 Customizing the About Box

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

**To customize the About Box:**

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",plugin_ID);
```

where plugin_ID is a unique identifier of the plug-in that registers this about.properties file.

> **Note:** Before calling the registerAboutPluginProperties() method, the plug-in must be registered using the registerPlugin() method of the YRCPlatformUI class. For more information on how to register a plug-in, see Appendix 17.13, "Registering a Plug-In".

# 6

# Modifying the Existing Rich Client Platform Screens and Wizards

## 6.1 Modifying Existing Rich Client Platform Screens

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

### 6.1.1 Starting the Rich Client Platform Extensibility Tool

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

### 6.1.2 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.

### 6.1.3 Synchronizing Differences

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

### 6.1.4 Building and Deploying Extensions

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

# 6.2 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:

```
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.

## 6.3 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.

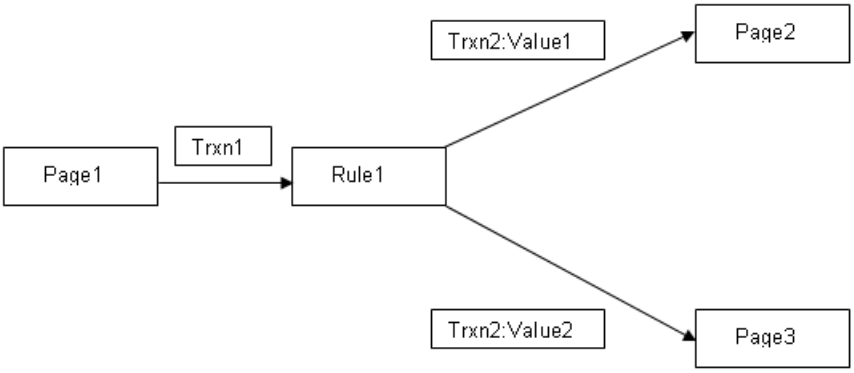
To get the wizard and namespace information:

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.

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

Let us consider that in the `<Plug-in_id>_<wizard_name>.ycml` file, you have an existing wizard definition defined for the following wizard flow:

***Figure 6–1 Sample 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.

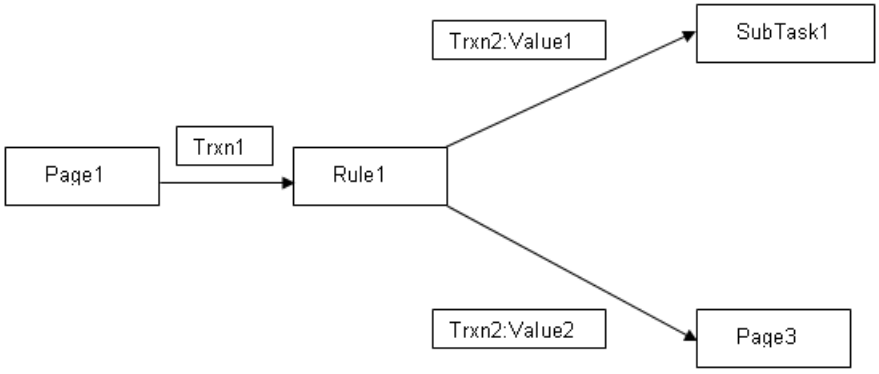
The sample `<Plug-in_id>_<wizard_name>.ycml` XML for the existing wizard definition is as follows:

```
<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:

***Figure 6–2 Sample Wizard Flow***




Create the extended wizard definition in the `<Plug-in_id>_<wizard_name>.ywx` file.

# 6.4 Creating an Extended Wizard Definition

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

To create an extended wizard definition:

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_id><wizard_name>.ywx`.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".

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_id>_<wizard_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.

## 6.5 Registering the Wizard Extension File

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_id>_<wizard_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. For more information about registering a plug-in, see Appendix 17.13, "Registering a Plug-In".

## 6.6 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 Section 8.4, "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 Section 8.3, "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.

## 6.7 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;
  }
  ```

# 7

# Creating and Adding Screens to Rich Client Platform Applications

## 7.1 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:

1. Composite File—The composite java file handles the UI. In the composite java file, write the code for naming, binding, localizing, and theming controls.
2. 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 following wizards for creating the Rich Client Platform composite:

- Rich Client Platform Search List Composite Wizard—The Rich Client Platform Search List Composite wizard creates a sample search and list screen. Multiple pages of the wizard allows you to name, bind, localize, and theme the controls separately for the search panel and list panel.
- Rich Client Platform Composite Wizard—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.

## 7.2 Creating a Rich Client Platform Composite Using the Rich Client Platform Search List Composite Wizard

The Rich Client Platform Search List Composite wizard automatically creates the composite Java file and the behavior Java file.

> **Note:** All new commands that you create are stored in the `<Plug-in_id>_commands.ycml` file. All new theme entries that you define are stored in the `<Plug-in_id>_<theme_name>.ythm` file. All new bundle entries that you define are stored in the `<Plug-in_id>_bundle.properties` file.

To call your own API or service instead of the Rich Client Platform-specific API or service, see Section 7.36, "Calling APIs and Services for Rich Client Platform Applications".

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

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
3. To store or create the sample search and list screen, 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 > Rich Client Platform Search List Composite.
6. Click Next. The Search List Screen Preferences window displays.

***Figure 7–1 Search List Screen Preferences Window***

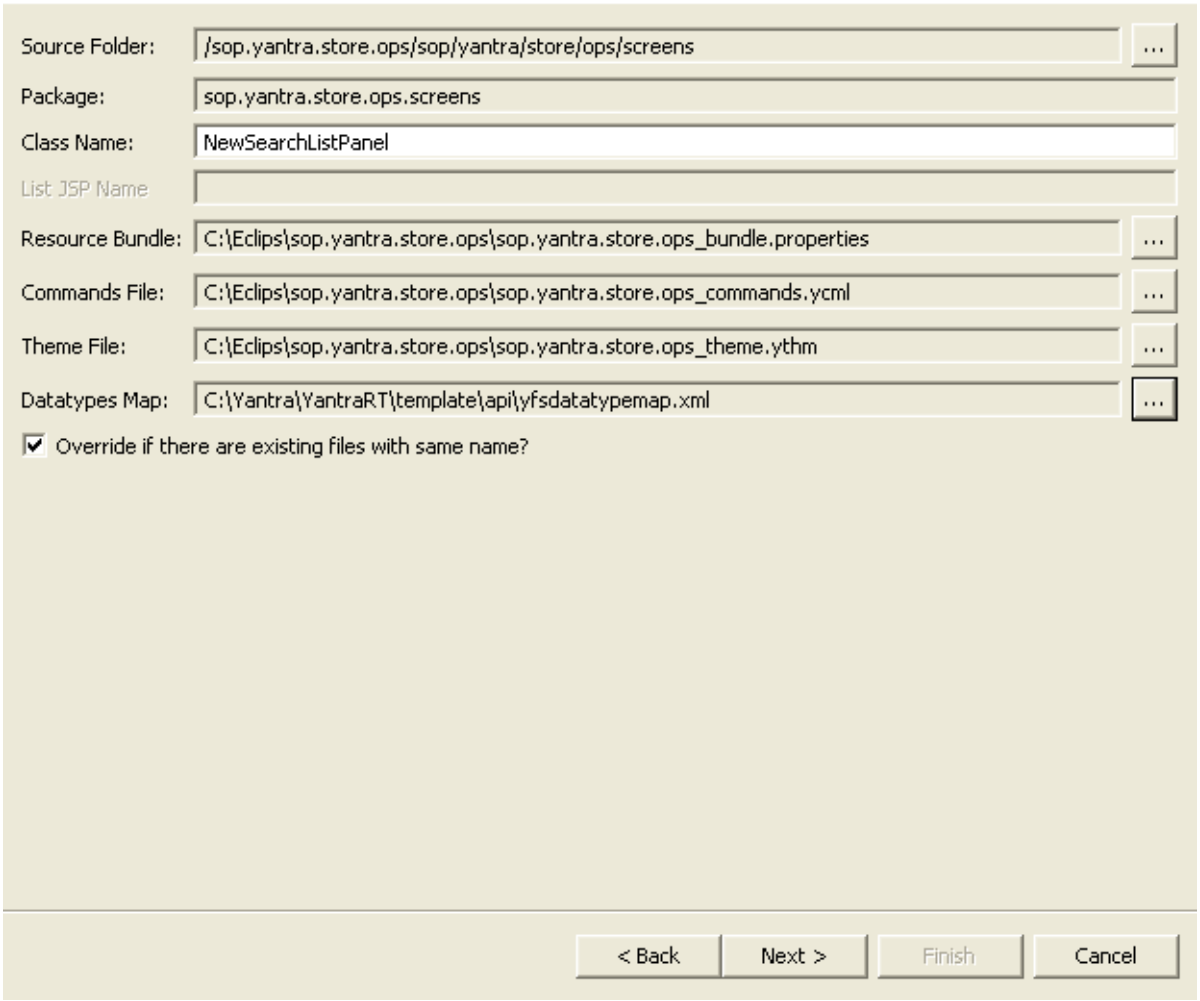



***Table 7–1 Search List Screen Preferences Window***

| Field | Description |
|---|---|
| Source Folder | The path of the selected folder automatically displays. You can browse to the folder where you want to store the sample search list screen files. |
| Package | The name of the package automatically displays. |

***Table 7–1 Search List Screen Preferences Window***

| Field | Description |
|---|---|
| Class Name | By default, the NewSearchListPanel.java class file name displays. Enter a new class file name, if necessary. |
| Resource Bundle | The path of the bundle file that exists in the plug-in project automatically displays, if any. You can browse to the <Plug-in_id>_bundle.properties bundle file in which new bundle entries for the search list screen are added. |
| Commands File | The path of the commands file that exists in the plug-in project automatically displays, if any. You can browse to the <Plug-in_id>_commands.ycml commands file in which new commands for the search list screen are added. |
| Theme File | The path of the theme file that exists in the plug-in project automatically displays, if any. You can browse to the <Plug-in_id>_<theme_name>.ythm theme file in which new theme entries for the search list screen are added. |
| Datatypes Map | Browse to the yfsdatatypemap.xml file in which the new data types are stored. This file is stored in the following location:<br><INSTALL_ DIR>/repository/xapi/template/merged/resource/yfsdatatypemap.xml |

7. Click Next. The Search Screen Input window displays.

*Figure 7–2 Search Screen Input Window*

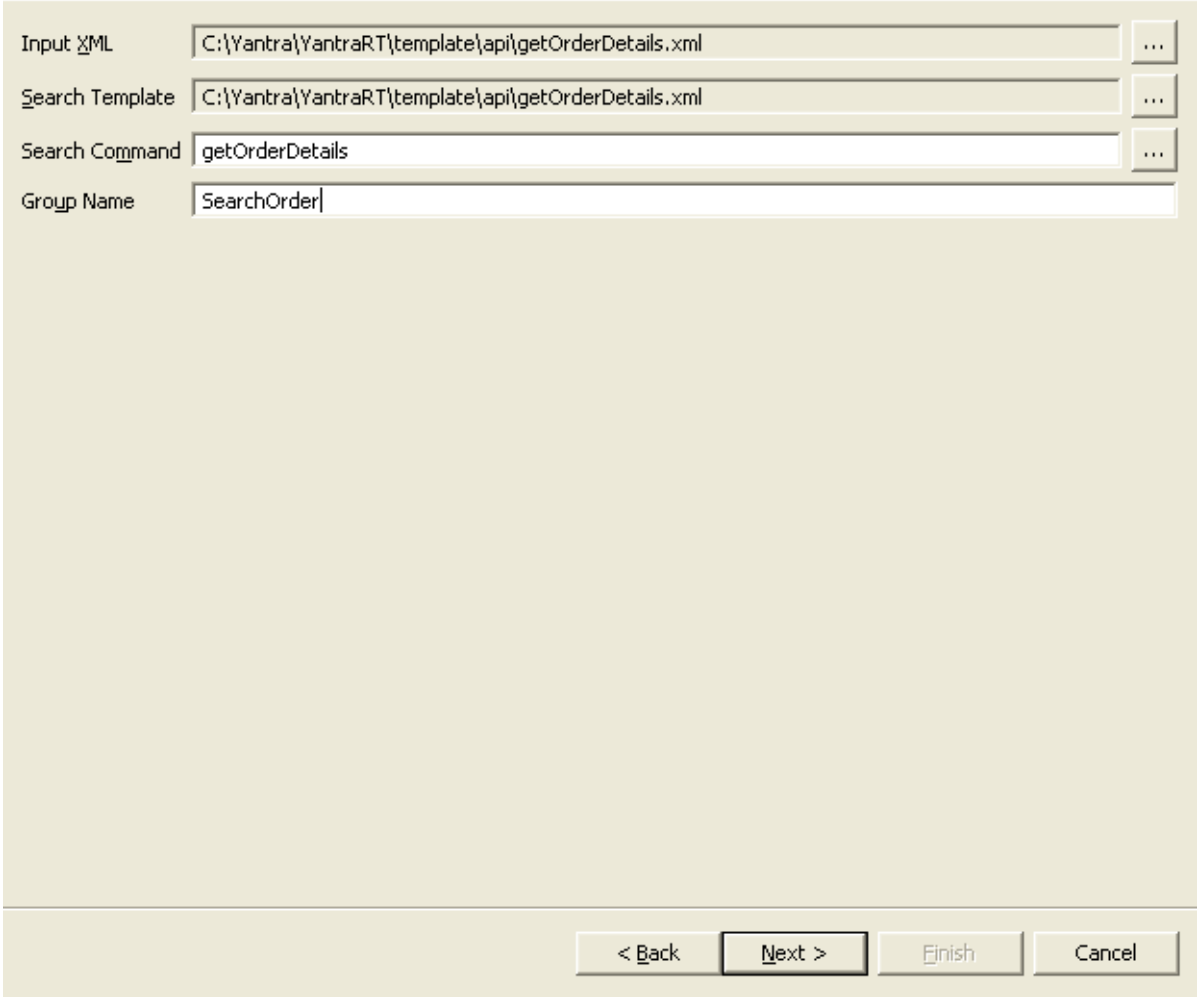
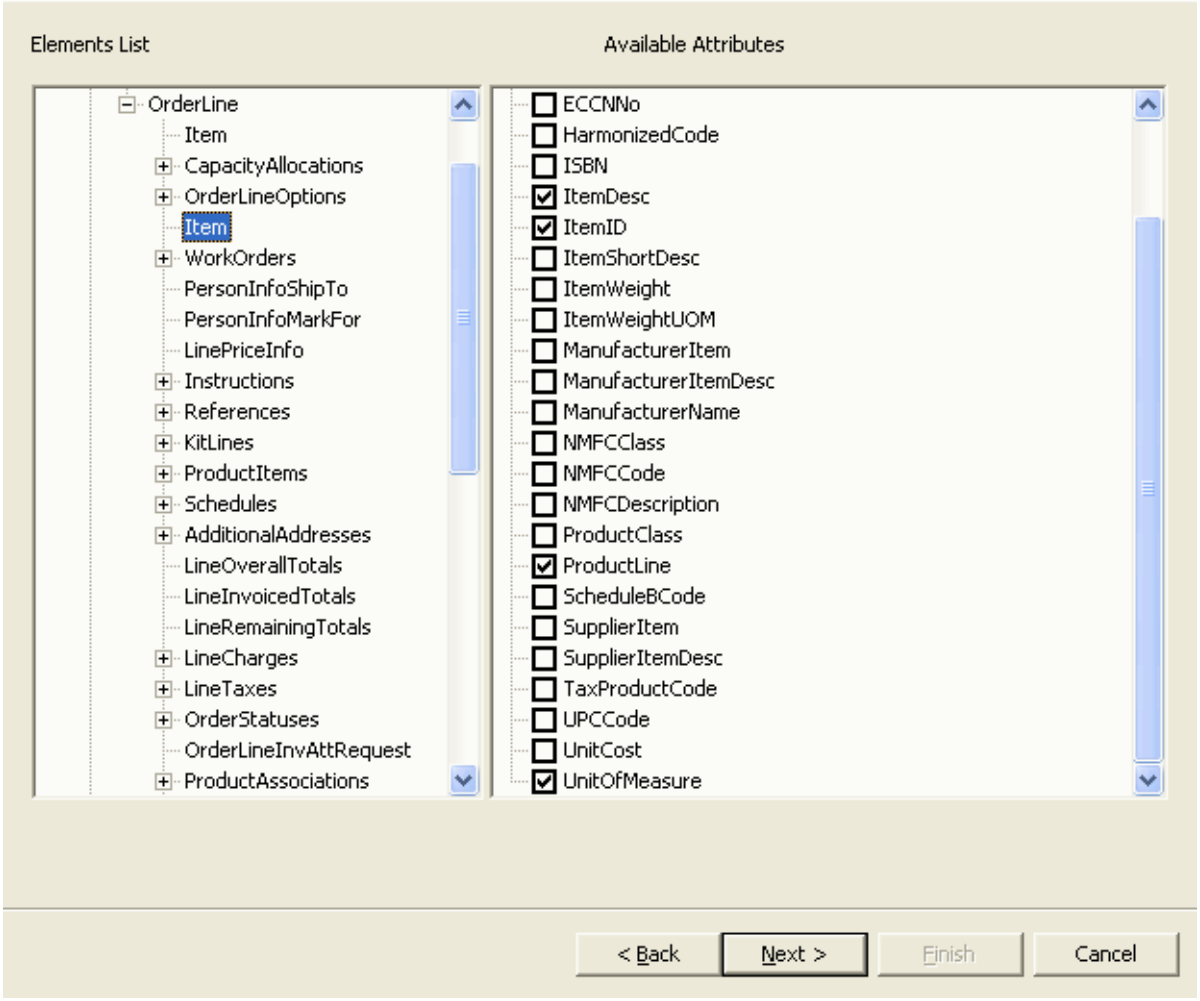


***Table 7–2 Search Screen Input Window***

| Field | Description |
|---|---|
| Input XML | Select the XML file, which is passed as input to a command or API to populate the search screen. |
| Search Template | Select the search template file, which is passed as output template to a command or API to populate the list screen. |
| Search Command | Enter the name of the command that is called during the search operation. Depending on the commands in the `<Plug-in_id>_commands.ycml` commands file, you can select the command on which you want the search to be based. You can also create a new command, if needed. If this command is not in the file, entry for the command is created in the command file when the wizard completes. |
| Group Name | Enter the name of the group that encloses the search screen, if applicable. All controls are positioned in a group box. You can name the group box by entering the name in "Group Name" text box. |

**8.** Click Next. The XML Attribute Selection window displays.

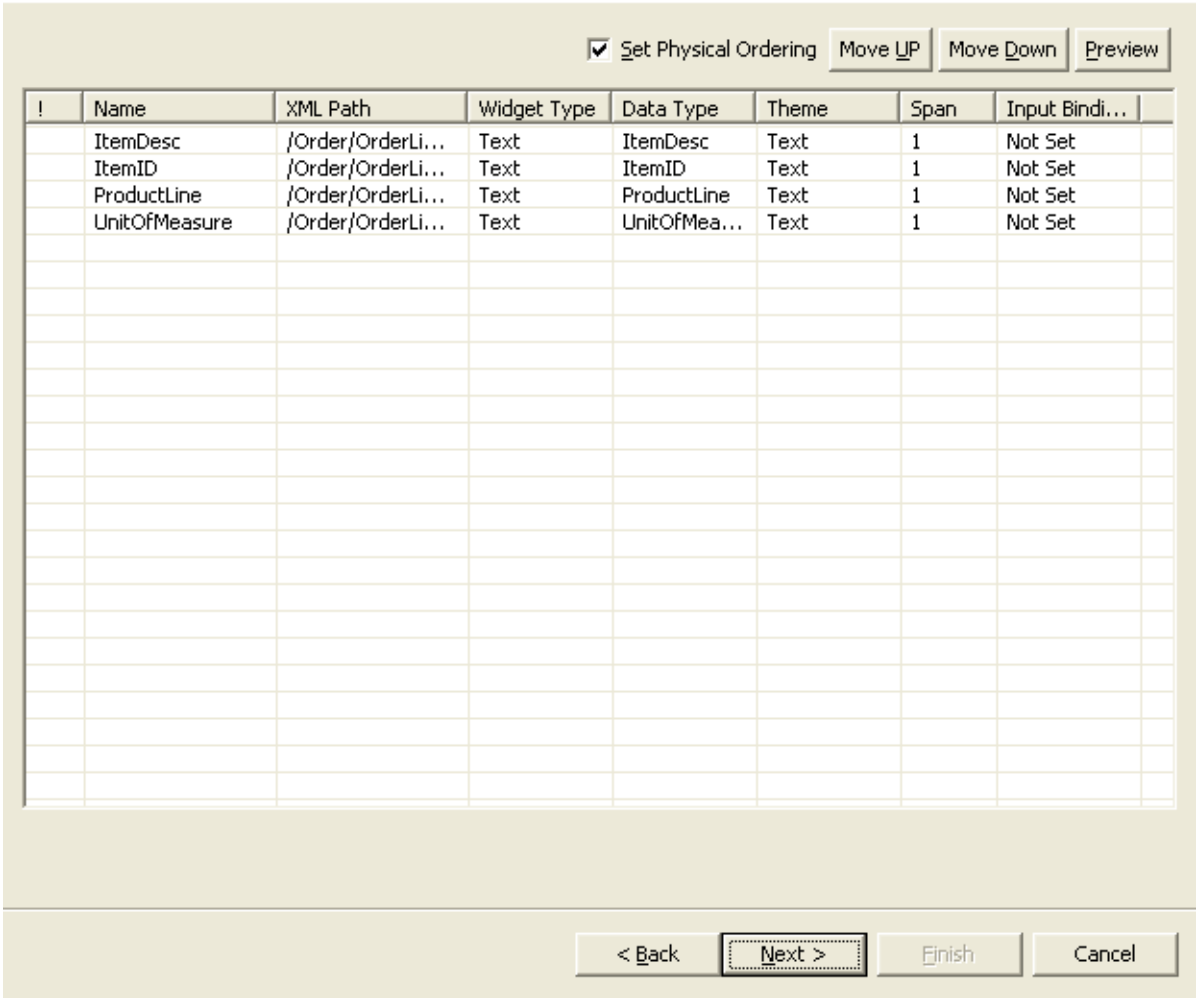
***Figure 7–3 XML Attribute Selection Window***




9. Select the XML attributes from the input XML, based on which the search screen is created. The search screen contains fields based on which the search operation is performed. The screen fields are created based on the selected attributes. The Element List panel displays the hierarchy of elements and their attributes for the specified input XML file. The Available Attribute panel displays the list of attributes corresponding to the element selected in the Element List panel.

**10.** Click Next. The Configure Individual Fields window displays.

***Figure 7–4 Configure Individual Fields Window***



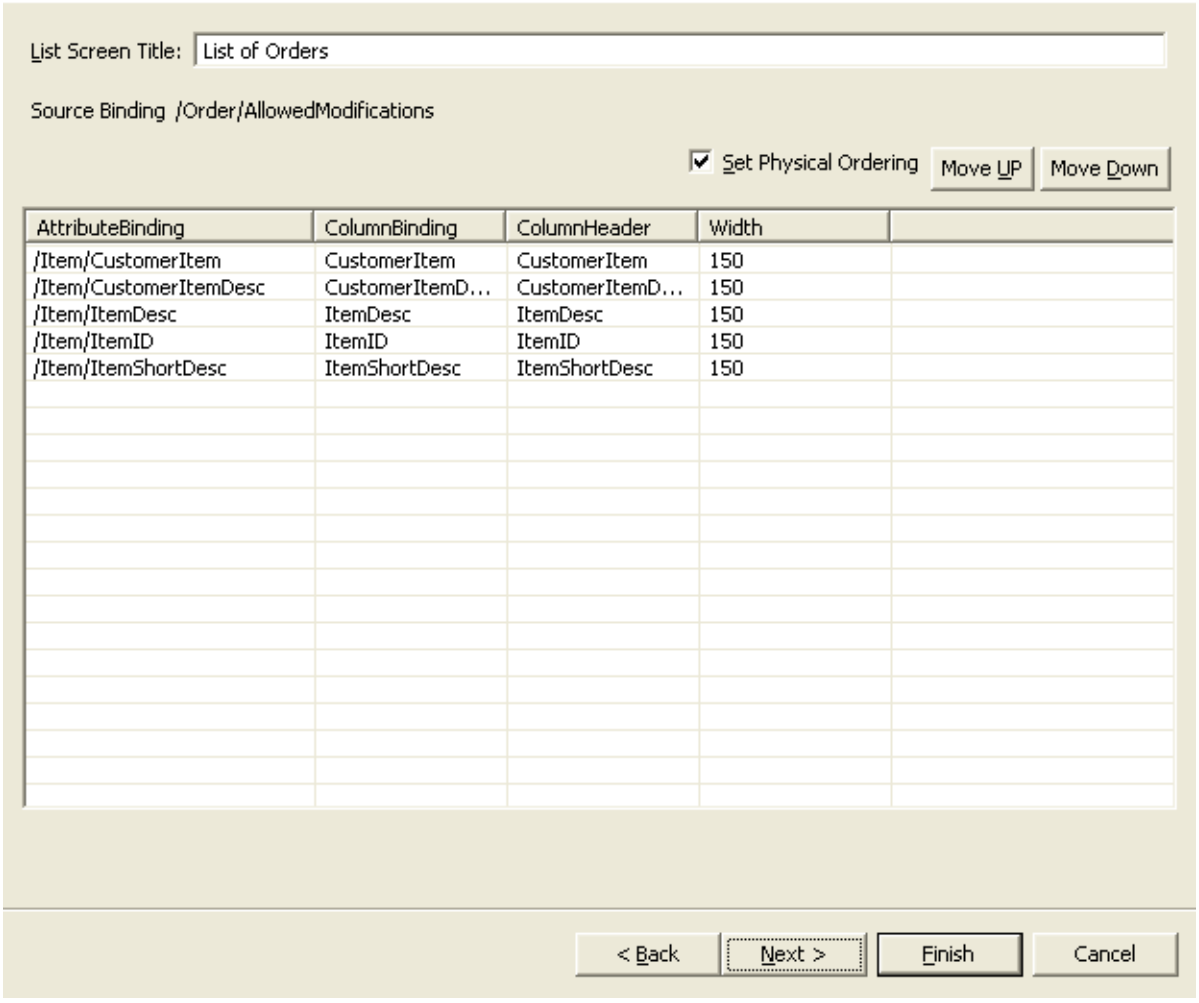

**11.** Configure the Search screen fields selected in the previous page. Specify the Name, Widget Type, Data Type, Theme, Span, and Input Binding to use on each individual control. You can change the physical ordering of the Search screen controls by clicking the Move Up or

Move Down button. This screen also provides visibility to the Search screen fields.

> **Note:** By default, a label is associated with each control unless the control itself is a label.

12. Click Next. The XML Attribute Selection window displays. Figure 7–3 illustrates XML Attribute Selection window.
13. Select the XML attributes from the input XML. Based on these XML attribute values, the fields in the List screen gets populated. The list screen has a table and all attributes displays in the appropriate columns. The Element List panel displays the hierarchy of elements and their attributes for the specified output template. The Available Attribute panel displays a list of attributes corresponding to the element selected in the Element List panel.
14. Click Next. The List Screen Fields window displays.

***Figure 7–5 List Screen Fields Window***




15. Configure the List screen fields selected in the previous page. In List Screen Title, enter the title of the List screen. Specify the AttributeBinding, ColumnBinding, ColumnHeader, and width to use on each field. You can change the physical ordering of the List screen controls by clicking the Move Up or Move Down button.
16. Click Next. The Select Template window displays.

17. From the Generator Template combo box, select the template. Based on the template, the Search List screen is created. Each template supports certain configurable attributes. The template is inbuilt with the default layout colspan, width, columnheader, and so forth. To change the configurable attribute, click Configure.
18. Click Finish. The system automatically creates the composite java file such as `NewSearchListPanel.java` and behavior java file such as `NewSearchListPanelBehavior.java` in the specified source folder. These files are stored under the specified package.

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

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

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.

***Figure 7–6 New Rich Client Platform Composite Window***

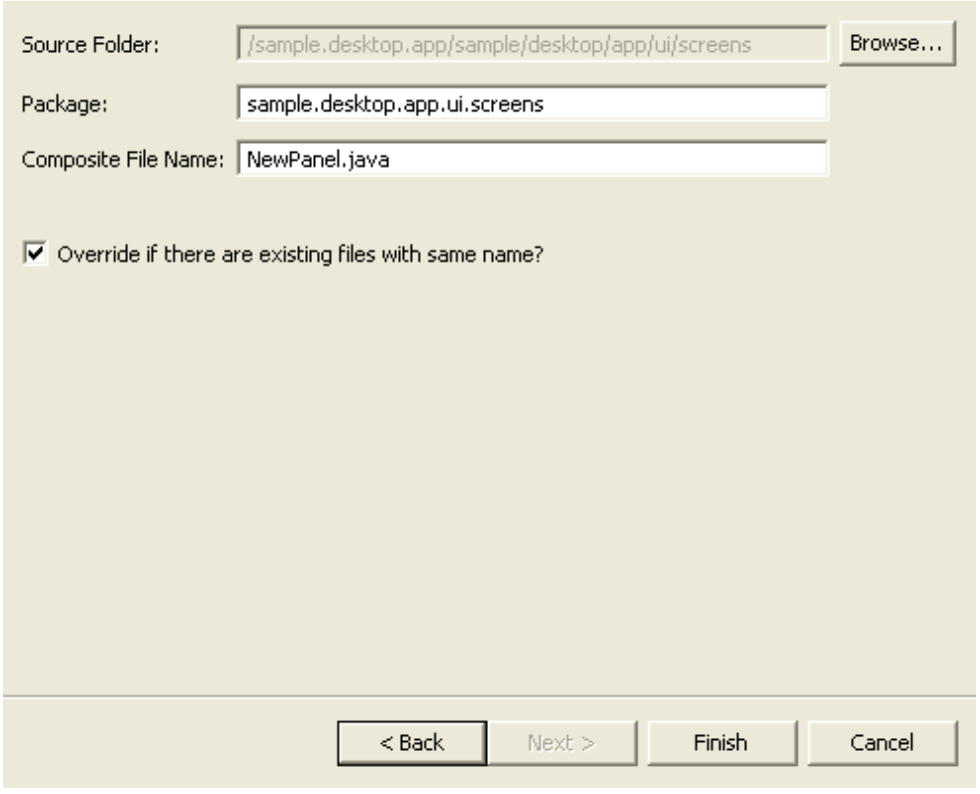


*Table 7–3 New Rich Client Platform Composite Window*

| 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. Figure 7–7 illustrates a typical folder structure that has both Java files stored under the package name.

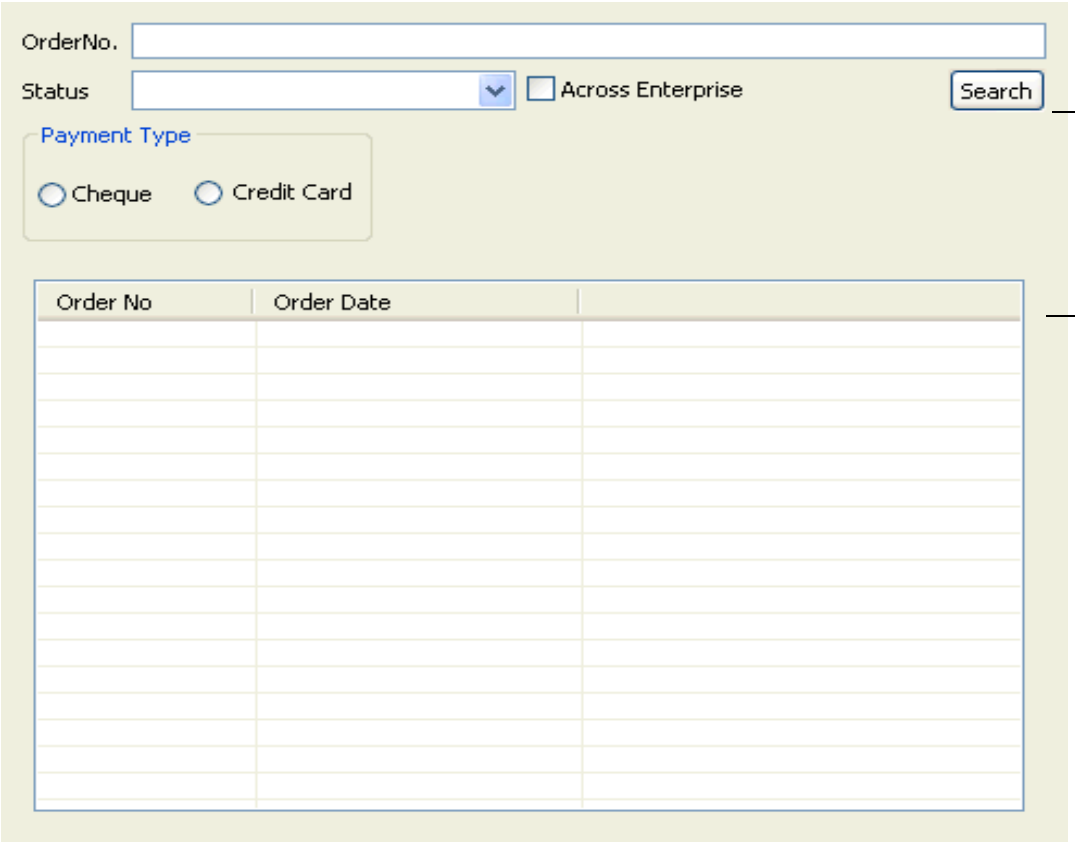
*Figure 7–7 Typical Folder Structure in Eclipse*


# 7.4 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. Figure 7–8 depicts a sample Search and List screen.

***Figure 7–8 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.

  For more information about creating the search criteria panel, see Section 7.5, "Creating the Search Criteria Panel for a Rich Client Platform Composite".

- 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.

  For more information about creating the search results panel, see Section 7.7, "Creating the Search Result Panel for a Rich Client Platform Composite".

## 7.5 Creating the Search Criteria Panel for a Rich Client Platform Composite

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

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. For more information about creating the Rich Client Platform composite, see Section 7.3, "Creating a Rich Client Platform Composite Using the Rich Client Platform Composite Wizard".
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.

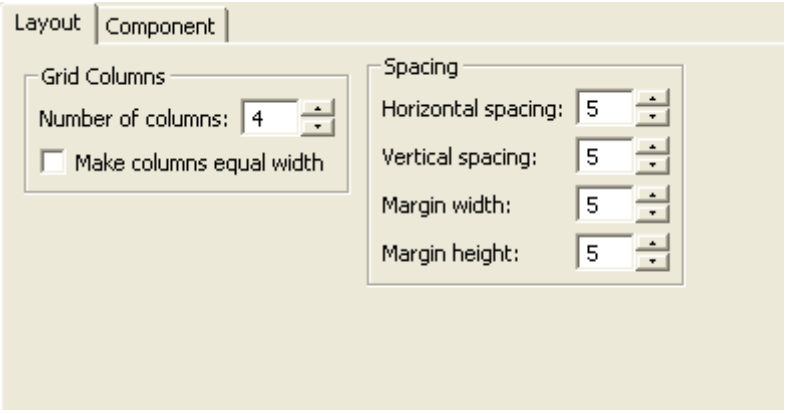
   In the Java Beans view, you can view the hierarchy of SWT containers and controls.

   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.

***Figure 7–9 Customize Layout***




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 Section 7.6, "Adding Controls to the Search Criteria Panel for a Rich Client Platform Composite".

18. Bind the controls to display the required data. For more information about binding controls, see Section 7.11, "Binding Controls and Classes for Rich Client Platform Screens".

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

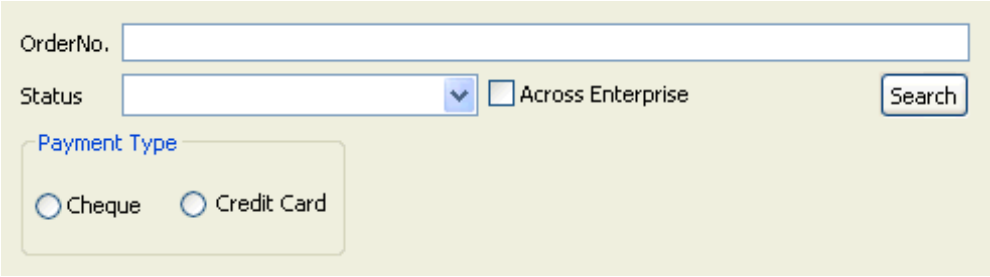
To add various controls to the composite follow these steps:

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 as shown in Figure 7–9.
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 as shown in Figure 7–9.
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 as shown in Figure 7–9.
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 as shown in Figure 7–9.
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 as shown in Figure 7–9.
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 as shown in Figure 7–9.
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 gets created as shown:

*Figure 7–10 Search Criteria Panel*




# 7.7 Creating the Search Result Panel for a Rich Client Platform Composite

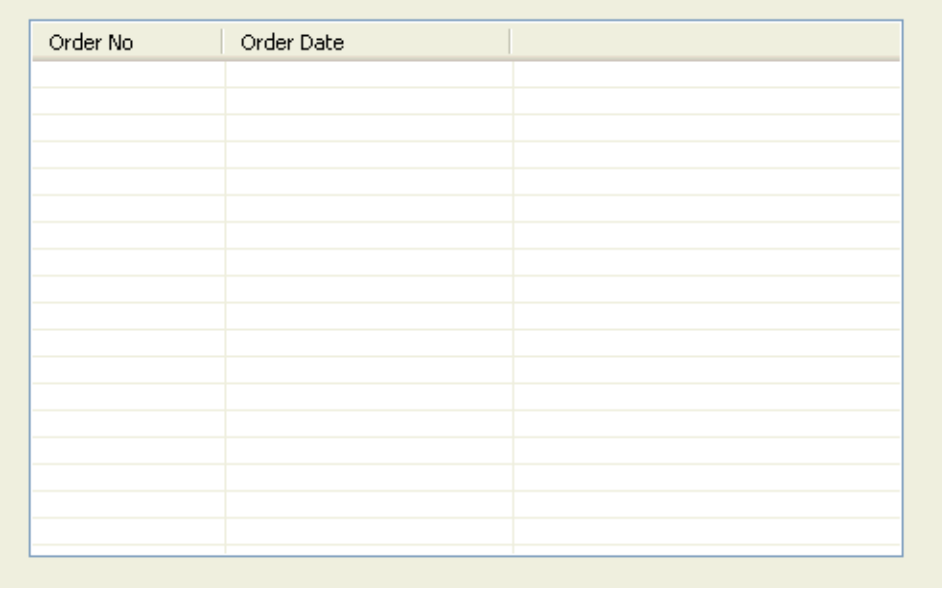
To create the Search Results panel for the Search and List screen:

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 as shown in Figure 7–9.
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. For more information about creating standard tables, see Section 7.9.1, "Creating Standard Tables".

10. Click . The following Search Results panel is created:

***Figure 7–11 Search Results Panel***




11. Bind the table and table columns with the required data. For more information about binding tables, see Section 7.31, "Setting Bindings for Standard Tables".

# 7.8 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.

**Note:** 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 *Selling and Fulfillment Foundation: Javadocs*.

## 7.8.1 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. For additional information about overriding properties using the `customer_overrides.properties` file, see the *Selling and Fulfillment Foundation: Properties Guide*.

**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.

### 7.8.2 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.

### 7.8.3 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.

### 7.8.4 IYRCPageNavigator

To get a handle for the various navigation operations for paginated results, call the getPageNavigator() method available in the behavior class. This method returns the IYRCPageNavigator interface, which can be used to handle the following navigation options for the paginated result set:

- 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—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.

### 7.8.5 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.

## 7.9 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.

### 7.9.1 Creating Standard Tables

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

To create a standard table:

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 as shown in Figure 7–9.
5. Set the layout properties such as Fill, Grab Access, and so forth as needed.

## 7.9.2 Adding Columns to the Standard Table

To add columns to a table:

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 Section 7.31, "Setting Bindings for Standard Tables".

## 7.9.3 Creating Editable Tables

To create an editable table:

1. Create a standard table. For more information about creating a standard table, see Section 7.9.1, "Creating Standard Tables".
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. For more information about binding a combo box cell editor, see Section 7.32.1, "Binding Combo Box Cell Editors".

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. For more information about binding an editable table, see Section 7.32, "Setting Bindings for an Editable Table".

# 7.10 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.

## 7.10.1 Creating a Binding Object

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.

### 7.10.2 Naming a Control

Use the binding object that you created to name the control. For more information about creating binding objects, see Section 7.10.1, "Creating a Binding Object".

To name a control:

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:** Sterling Commerce 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.

## 7.11 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.

## 7.11.1 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.

## 7.11.2 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.

# 7.12 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.

## 7.12.1 Multiple Source Bindings

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 Section 7.12, "Source Binding for Controls on Rich Client Platform Screens". To display the values of both OrderNo and OrderDate attributes of the XML model in a text box, do the following:

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

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

   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.

   For more information about the key binding, see Section 7.20, "Key Binding for Controls on Rich Client Platform Screens".

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.

# 7.13 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.

### 7.13.1 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;OrderListDe
tails:OrderList/Order/@OrderNo;OrderLineDetails:OrderLine/OrderLineList/Order/@O
rderNo")
```

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

## 7.14 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.

## 7.15 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 Section 7.14, "Checked Binding for Controls on Rich Client Platform Screens". 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.

## 7.16 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.

# 7.17 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 Section 7.16, "List Binding for Controls on Rich Client Platform Screens". 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 Section 7.18, "Description Binding for Controls on Rich Client Platform Screens".

## 7.18 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 Section 7.16, "List Binding for Controls on Rich Client Platform Screens". 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 specified in the List Binding for Controls on Rich Client Platform Screens.

## 7.19 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.

## 7.19.1 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.

  For more information about key binding, see Section 7.20, "Key Binding for Controls on Rich Client Platform Screens".

- 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. For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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

# 7.20 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.

# 7.21 Binding Input to Custom Controls on Rich Client Platform Screens

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 following:

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 );
```

# 7.22 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:

## 7.22.1 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>
```

### 7.22.2 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>
```

## 7.23 Setting Bindings for Labels

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

### 7.23.1 Creating a Binding Object

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.

## 7.23.2 Steps to Bind a Label

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.

   For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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.

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

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.

   For more information about key binding, see Section 7.20, "Key Binding for Controls on Rich Client Platform Screens".

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 the *Selling and Fulfillment Foundation: Installation Guide*.

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_DEFINATION` is the key used for identifying the label binding object.

# 7.24 Setting Bindings for Text Boxes

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

## 7.24.1 Creating a Binding Object

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.

## 7.24.2 Steps to Bind a Text Box

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.

For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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.

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

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.

For more information about key binding, see Section 7.20, "Key Binding for Controls on Rich Client Platform Screens".

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.

For more information about target binding, see Section 7.13, "Target Binding for Controls on Rich Client Platform Screens".

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.

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

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.

# 7.25 Setting Bindings for StyledText Components

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

## 7.25.1 Creating a Binding Object

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.

## 7.25.2 Steps to Bind a StyledText Component

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 Section Section 7.12, "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/@S
tatus");
```

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 Section 7.20, "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/@Or
derNo");
```

where `OrderListDetails` is the namespace of the model.

For more information about target binding, see Section 7.13, "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;OrderSta
tus/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.

# 7.26 Setting Bindings for Combo Boxes

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

## 7.26.1 Creating a Binding Object

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.

## 7.26.2 Steps to Bind a Combo Box

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.

For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

> **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.

For more information about list binding, see Section 7.16, "List Binding for Controls on Rich Client Platform Screens".

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

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

For more information about description binding, see Section 7.18, "Description Binding for Controls on Rich Client Platform Screens".

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

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

For more information about code binding, see Section 7.17, "Code Binding for Controls on Rich Client Platform Screens".

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.

For more information about target binding, see Section 7.13, "Target Binding for Controls on Rich Client Platform Screens".

> **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.

## 7.26.3 Populating 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 parameteres. The method `comboBindingData.setApplicationVersionSpecific(true)`is called for displaying version-specific data in select combo boxes for the required application.

To populate the combo boxes with version-specific data:

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 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.

# 7.27 Setting Bindings for List Boxes

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

## 7.27.1 Creating a Binding Object

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.

## 7.27.2 Steps to Bind a List Box

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.

   For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

   > **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.

   For more information about list binding, see Section 7.16, "List Binding for Controls on Rich Client Platform Screens".

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

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

   For more information about description binding, see Section 7.18, "Description Binding for Controls on Rich Client Platform Screens".

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

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

For more information about code binding, see Section 7.17, "Code Binding for Controls on Rich Client Platform Screens".

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.

For more information about target binding, see Section 7.13, "Target Binding for Controls on Rich Client Platform Screens".

> **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.

# 7.28 Setting Bindings for Checkboxes

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

## 7.28.1 Creating a Binding Object

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.

## 7.28.2 Steps to Bind a Check Box

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.

   For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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

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

   where OrderDetails is the namespace of the model.

   For more information about target binding, see Section 7.13, "Target Binding for Controls on Rich Client Platform Screens".

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.

   For more information about checked binding, see Section 7.14, "Checked Binding for Controls on Rich Client Platform Screens".

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.

For more information about unchecked binding, see Section 7.15, "Unchecked Binding for Controls on Rich Client Platform Screens".

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.

# 7.29 Setting Bindings for Radio Buttons

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

## 7.29.1 Creating a Binding Object

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.

## 7.29.2 Steps to Bind a Radio Button

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.

For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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

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

where OrderDetails is the namespace of the model.

For more information about target binding, see Section 7.13, "Target Binding for Controls on Rich Client Platform Screens".

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.

For more information about checked binding, see Section 7.14, "Checked Binding for Controls on Rich Client Platform Screens".

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.

# 7.30 Setting Bindings for Links

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

## 7.30.1 Creating a Binding Object

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.

## 7.30.2 Steps to Bind a Link

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.

For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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.

# 7.31 Setting Bindings for Standard Tables

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.

## 7.31.1 Creating a Binding Object for a Standard Table

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.

## 7.31.2 Creating a Binding Object for a Column

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];
```

## 7.31.3 Steps to Bind a Standard Table and Column

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");
   ```

   For more information about attribute binding, see Section 7.19, "Attribute Binding for Controls on Rich Client Platform Screens".

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.

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

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.

For more information about key binding, see Section 7.20, "Key Binding for Controls on Rich Client Platform Screens".

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 *Selling and Fulfillment Foundation: Installation Guide*.

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 localizing the database, see Section 2.6.1, "Database Localization".

11. Repeat Step 2 to Step 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.

    For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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.

# 7.32 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. Therefore, to bind an editable table, follow the steps as described in Section 7.31, "Setting Bindings for Standard Tables".

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.

## 7.32.1 Binding Combo Box Cell Editors

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:

1. Create a binding object for the combo box. For more information about creating a binding object, see Section 7.26.1, "Creating a Binding Object".
2. Set the list binding, description binding, and code binding for the combo box. For more information about setting these bindings, see Section 7.26, "Setting Bindings for Combo Boxes".

**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.

## 7.33 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.

## 7.33.1 Creating a Binding Object for an Extended Table

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.

## 7.33.2 Create a Map of the Advanced Column Binding Data

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.

## 7.33.3 Steps to Bind an Extended Table and Advanced Column

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");
   ```

   For more information about attribute binding, see Section 7.19, "Attribute Binding for Controls on Rich Client Platform Screens".

3. (Optional) Set multiple attribute binding for the advanced column. For example:

   ```
   advclmBindingData.setAttributeBinding("ItemID;Code");
   ```

   where OrderDetails is the namespace of the model.

   For more information about multiple target binding, see Section 7.19.1, "Multiple Attribute Bindings".

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.

For more information about key binding, see Section 7.20, "Key Binding for Controls on Rich Client Platform Screens".

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 *Selling and Fulfillment Foundation: Installation Guide*.

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 Section 2.6.1, "Database Localization".

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.

For more information about source binding, see Section 7.12, "Source Binding for Controls on Rich Client Platform Screens".

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.

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.

# 7.34 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. Therefore, to bind an extended editable table, follow the steps as described in Setting Bindings for an Extended Table.

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.

## 7.34.1 Binding Combo Box Cell Editors

Binding combo box cell editors indicates binding a combo box inside an editable extended table. To set bindings for a combo box cell editor:

1. Create a binding object for the combo box. For more information about creating a binding object, see Section 7.26.1, "Creating a Binding Object".

2. Set the list binding, description binding, and code binding for the combo box. For more information about setting these bindings, see Section 7.26, "Setting Bindings for Combo Boxes".

   **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_DEFINATION 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.

# 7.35 Localizing Controls and Defining Themes for Rich Client Platform Applications

This Section explains how to localize controls, text, or strings.

To localize controls, text, or strings:

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:

1. 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.

2. 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.

### 7.35.1 Defining Themes for Controls

For theming controls, define the new theme entries in the `<Plug-in_id>_<theme_name>.ythm` file.

## 7.36 Calling APIs and Services for Rich Client Platform Applications

Calling an API or service is as follows:

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 Chapter 10, "Creating Commands for Rich Client Platform Applications".

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 Section 7.36.1, "Calling the Same API/Service Multiple Times".

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.

## 7.36.1 Calling the Same API/Service Multiple Times

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:

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.
}
```

## 7.36.2 Calling Multiple APIs/Services

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:** Sterling Commerce 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.

Therefore, to call multiple APIs or services:

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 Chapter 10, "Creating Commands for Rich Client Platform Applications".

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); }}
```

# 7.37 Adding New Rich Client Platform Screens as Pop-ups

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:

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.

# 7.38 Adding New Rich Client Platform Screens to Menu Commands

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 Selling and Fulfillment Foundation 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. For more information about Rich Client Platform actions, see Appendix 17.12, "Creating New 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 opening a screen using a pop-up windows, see Section 7.37, "Adding New Rich Client Platform Screens as Pop-ups". For information on how to open a screen in an editor, see Section 7.39, "Displaying New Rich Client Platform Screens in an Editor".

For more information about defining resources, see the *Selling and Fulfillment Foundation: Application Platform Configuration Guide*.

# 7.39 Displaying New Rich Client Platform Screens in an Editor

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 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.

***Figure 7–12 Java Attribute Editor Window***

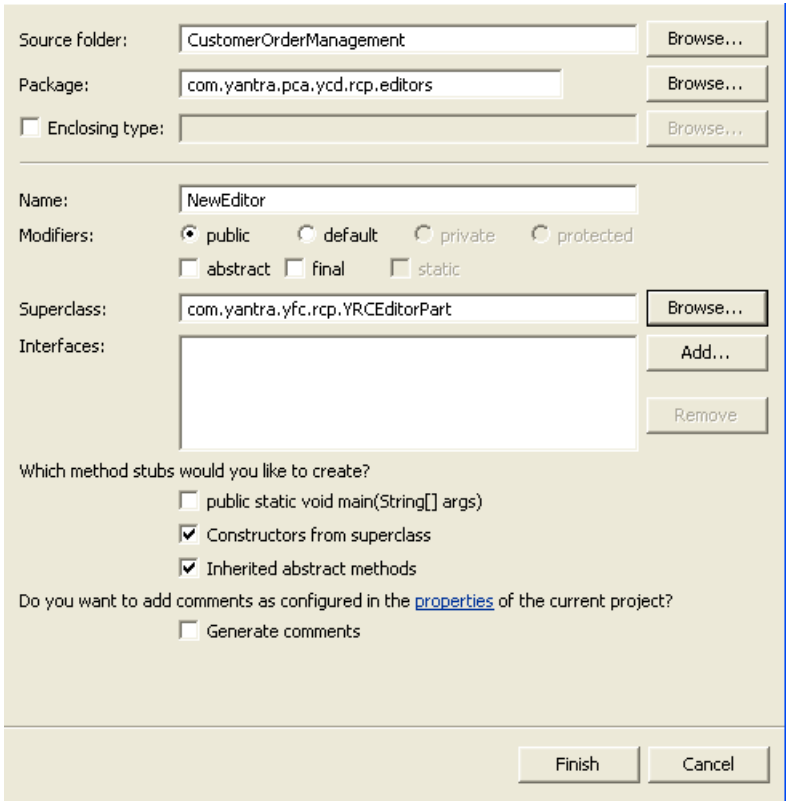



***Table 7–4 Java Attribute Editor Window***

| 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. |

*Table 7–4 Java Attribute Editor Window*

| Field | Description |
|---|---|
| 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. For more information about opening new screens using menu, see Section 7.38, "Adding New Rich Client Platform Screens to Menu Commands".

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);
```

# 8

# Creating and Adding Wizards to Rich Client Platform Applications

## 8.1 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 Chapter 3, "The Development Environment for Rich Client Platform Applications".

### 8.1.1 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.

## 8.2 Creating a Wizard Definition with the Rich Client Platform Wizard Editor

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:

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. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.
6. In the Properties view, in Wizard Description, enter the description for the new wizard.

## 8.3 Adding a Rule to a Wizard Definition

To add a new wizard rule:

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 Section 8.2, "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.

   For more information about Greex rule or an advanced XML condition, see the *Selling and Fulfillment Foundation: Extending the Condition Builder Guide* .
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. For more information about defining namespaces, see Section 10.2, "Defining Namespaces".

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.

## 8.4 Adding a Page to a Wizard Definition

To add a new wizard page:

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 Section 8.2, "Creating a Wizard Definition with 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.

## 8.5 Adding a Sub-task to a Wizard Definition

To add a new sub-task:

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 Section 8.2, "Creating a Wizard Definition with 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. For more information about defining namespaces, see Section 10.2, "Defining Namespaces".

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.

## 8.6 Adding a Transition to a Wizard Definition

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:

1. Open the `<Plug-in_id>_commands.ycml` file using the Rich Client Platform Wizard Editor. For more information about opening the Rich Client Platform Wizard Editor, see Section 8.2, "Creating a Wizard Definition with 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.

5. In Value, enter the value for which this transition is to be performed. This value is compared with the output value returned by a wizard rule, and depending on this value, the control is transferred to the appropriate wizard entity.

**Note:** Each Transition ID should have a unique value associated with it.

## 8.7 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.

## 8.8 Creating Wizard Components

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.

This section explains the following:

- Creating Wizard Class
- Creating Wizard Behavior Class

### 8.8.1 Creating Wizard Class

To create a wizard class:

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.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".

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.

***Figure 8–1 New Java Class window***

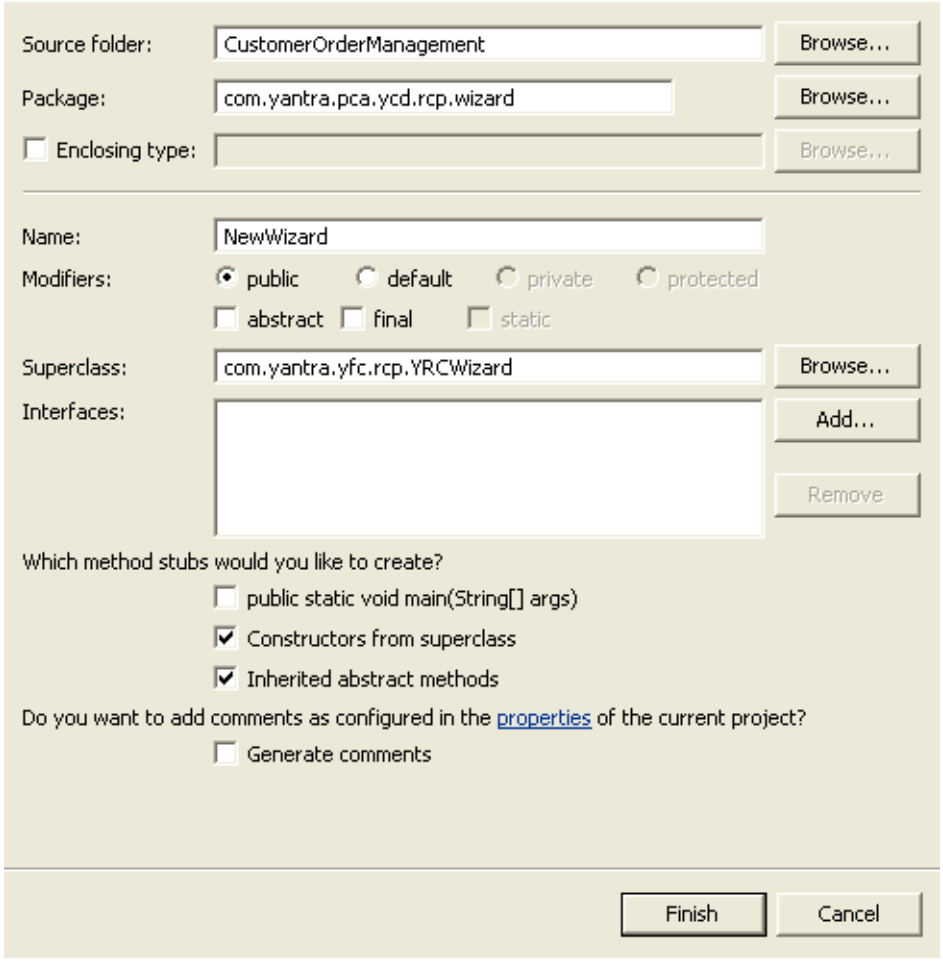


***Table 8–1 New Java Class Window***

| 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... option 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;
}
```

For more information about creating wizard behavior class, see Section 8.8.2, "Creating Wizard Behavior Class".

## 8.8.2 Creating Wizard Behavior Class

To create a wizard behavior class:

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.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.

***Figure 8–2 New Java Class Window***

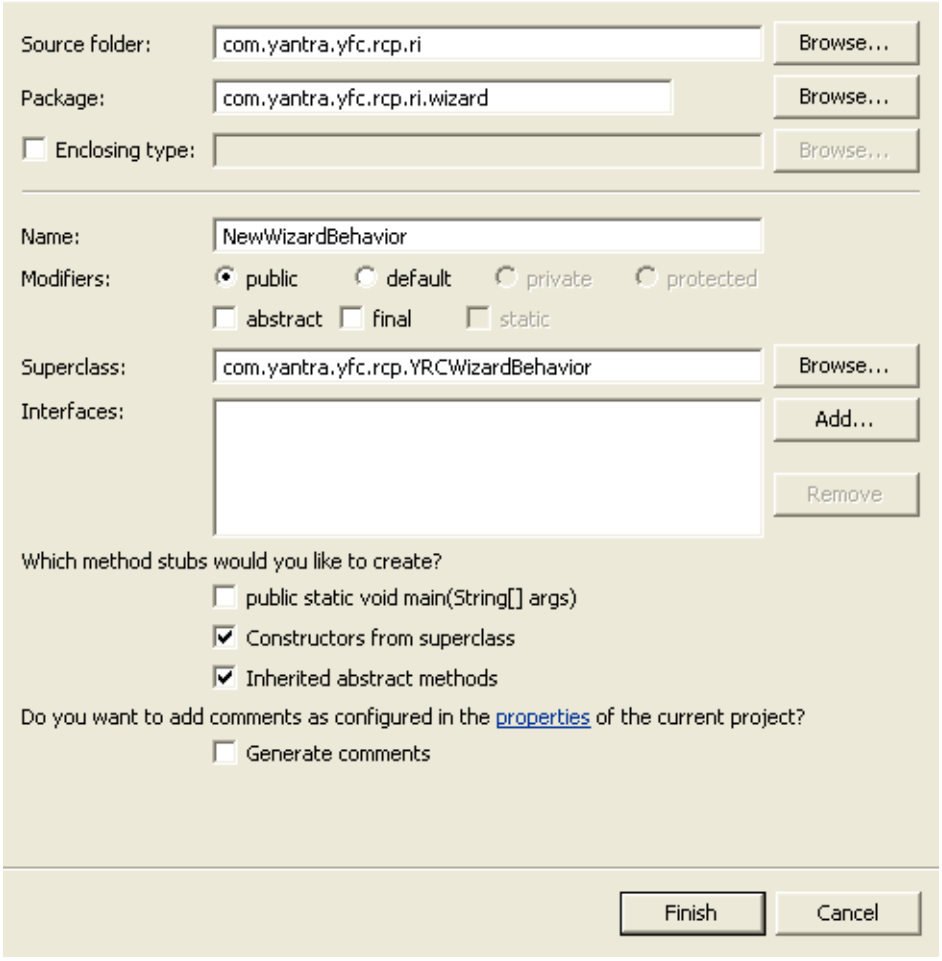


***Table 8–2 New Java Class Window***

| 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 Section 8.9.1, "Creating Wizard Page Class".

# 8.9 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.

## 8.9.1 Creating Wizard Page Class

To create a wizard page class:

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.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".

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.

***Figure 8–3 New Java Class Window***

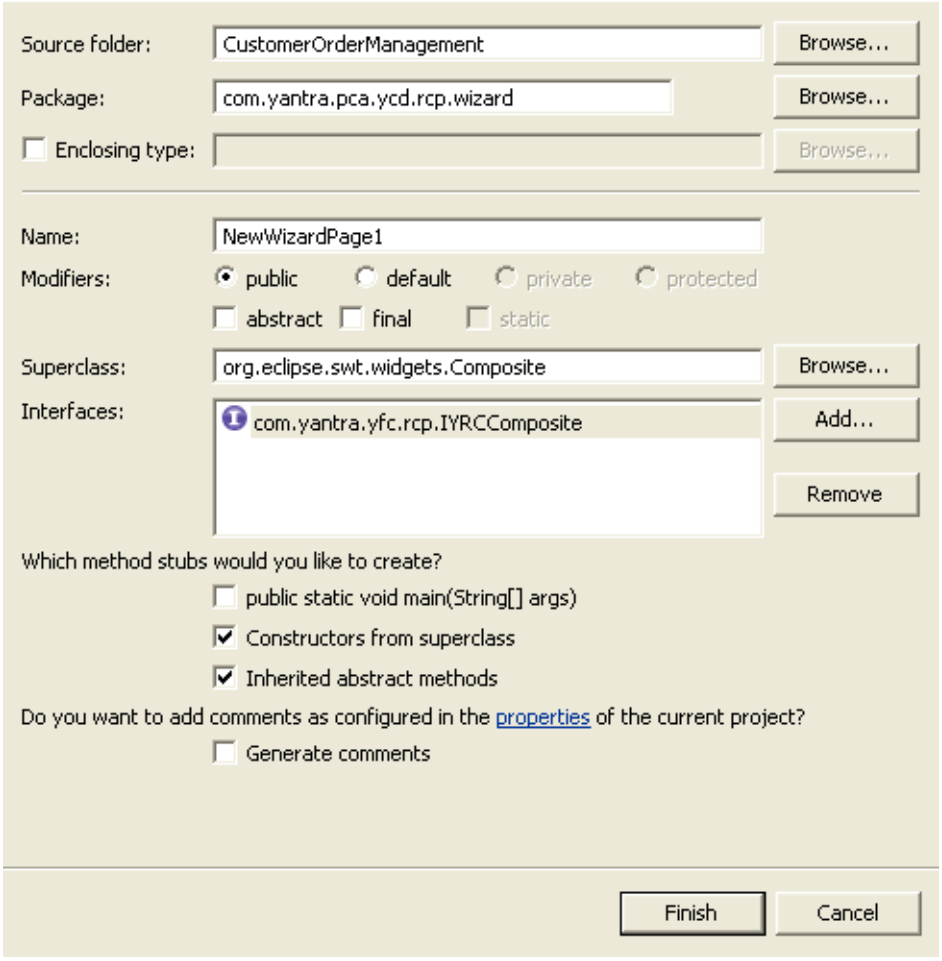


*Table 8–3 New Java Class Window*

| 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. For more information about designing a Rich Client Platform composite, see Section 7.4, "About Designing a Rich Client Platform Composite".
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);
}
```

For more information about creating wizard page behavior class, see Section 8.9.2, "Creating Wizard Page Behavior Class".

## 8.9.2 Creating Wizard Page Behavior Class

To create a wizard page behavior class:

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. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.

***Figure 8–4 New Java Class Window***

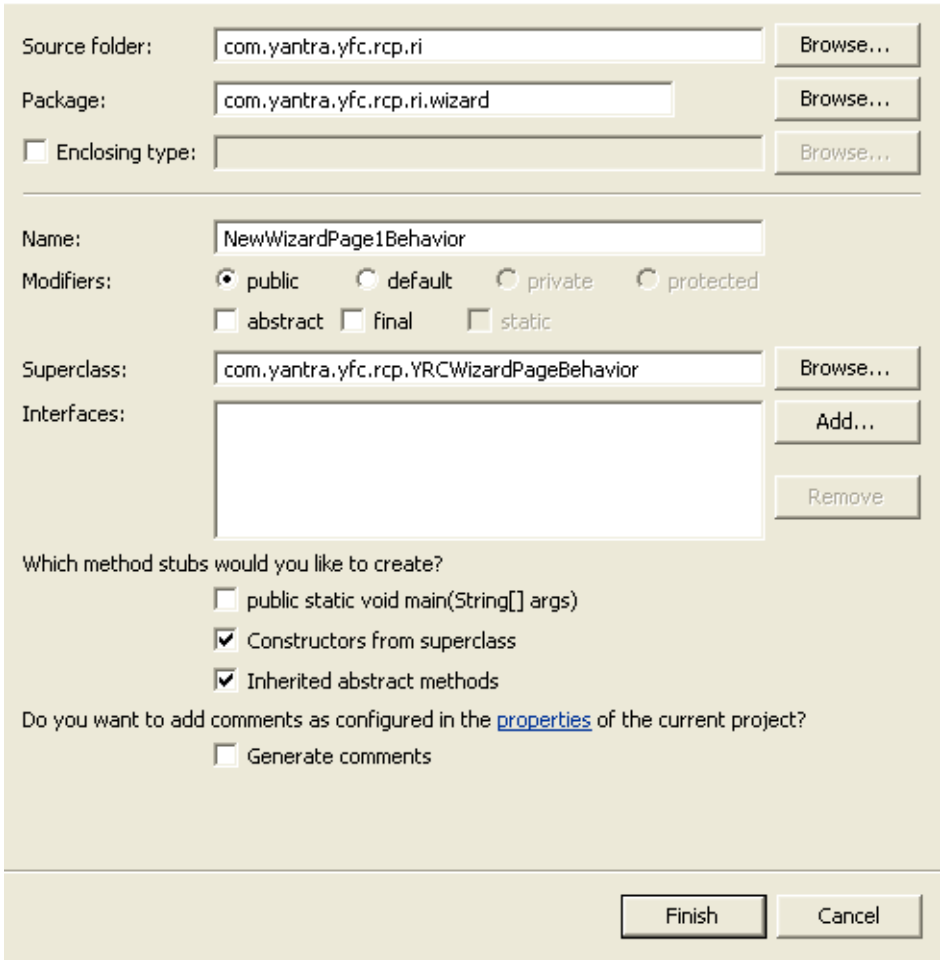


*Table 8–4 New Java Class Window*

| 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.

## 8.10 Creating Wizard Rule Components

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:

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. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.

***Figure 8–5 New Java Class Window***

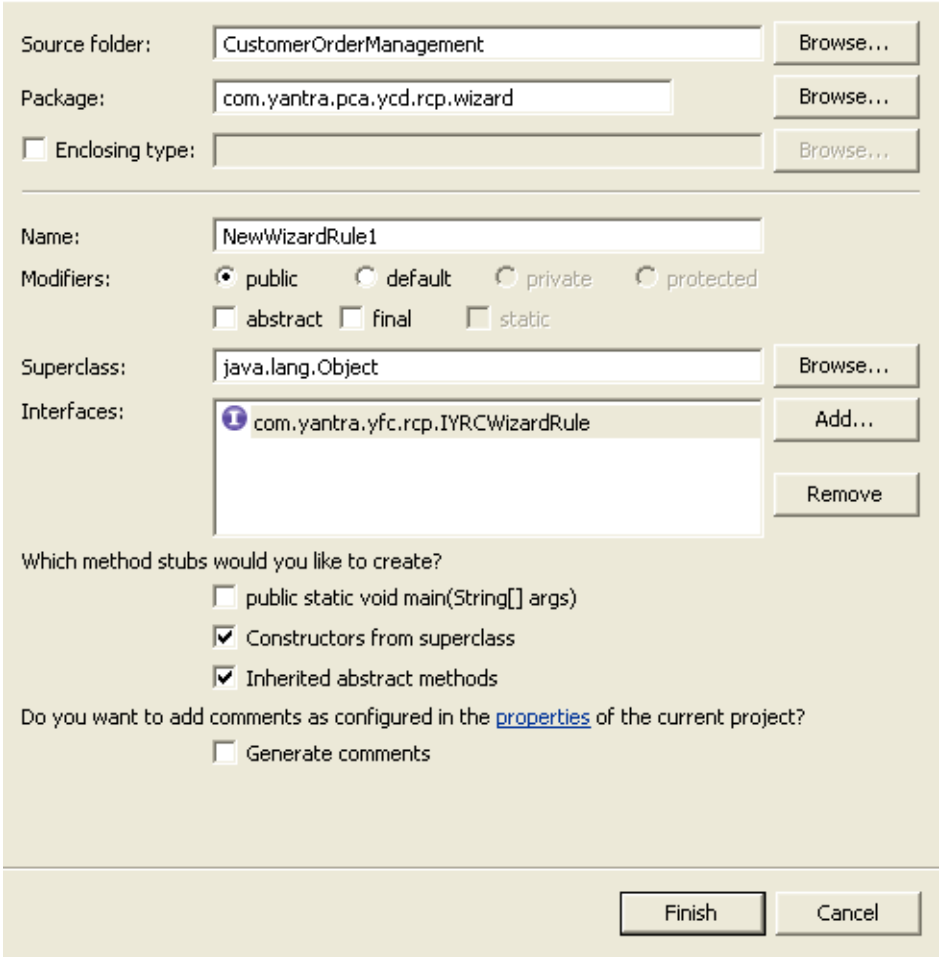


*Table 8–5 New Java Class Window*

| 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. For more information about defining namespaces, see Section 10.2, "Defining Namespaces".

### 8.10.1 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. For more information about registering commands file, see Appendix 17.14.4, "Registering Commands File".

## 8.11 Adding Wizards as Pop-ups in Rich Client Platform Applications

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:

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();
```

# 8.12 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 Selling and Fulfillment Foundation Resources. All the Selling and Fulfillment Foundation 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 Selling and Fulfillment Foundation 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. For more information about Rich Client Platform actions, see Appendix 17.12, "Creating New 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 opening a wizard in a pop-up window, see Section 8.11, "Adding Wizards as Pop-ups in Rich Client Platform Applications". For information about opening a wizard in an editor, see Section 8.13, "Adding Wizards to Editors in Rich Client Platform Applications".

For more information about defining resources, see the *Selling and Fulfillment Foundation: Application Platform Configuration Guide*.

# 8.13 Adding Wizards to Editors in Rich Client Platform Applications

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 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.

***Figure 8–6 Java Attribute Editor Window***

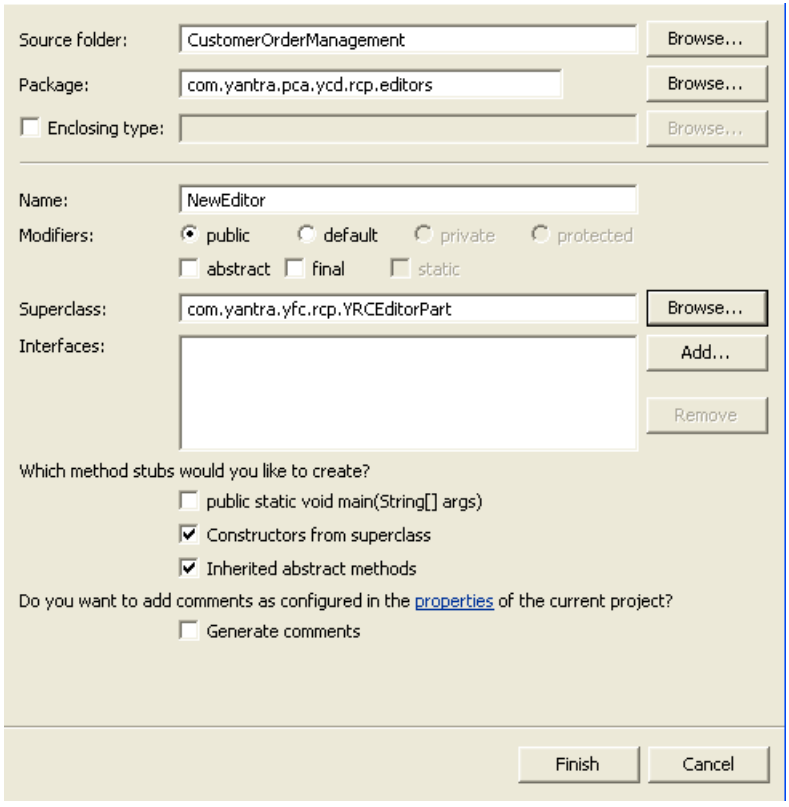



***Table 8–6 Java Attribute Editor Window***

| 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. |

***Table 8–6 Java Attribute Editor Window***

| Field | Description |
|---|---|
| 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. Open the newly created editor class in the Java Editor.
16. 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;
}
```

17. To open the new wizard in the specified editor using the menu item, define a new resource in the Selling and Fulfillment Foundation Resources for the new menu item. For more information about opening new wizards using menu, see Section 8.12, "Adding Wizards to Menu Commands in Rich Client Platform Applications".
18. 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);
```

# 9

# Creating Related Tasks for Rich Client Platform Applications

## 9.1 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. For more information about the YRCRelatedTasks extension point, see Section 2.8, "Related Tasks for Rich Client Platform Applications".

## 9.2 Extending the YRCRelatedTasks Extension Point

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:

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

To extend the YRCRelatedTasks extension point:

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

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.YRCRelatedTasks extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCRelatedTasks extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of YRCRelatedTasks extension point.
9. Right-click on com.yantra.yfc.rcp.YRCRelatedTasks extension and select New > tasks. The `tasks` extension element gets created.
10. Select the tasks extension element. The Extension Element Details panel displays.
11. In the Extension Element Details panel, enter the properties of the tasks extension element.
12. 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.
13. In the Extension Details panel, enter the properties of the task extension element.
14. In permissionId, enter the resource identifier from the Selling and Fulfillment Foundation 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.
15. 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.
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. For more information about extending the YRCRelatedTaskGroups extension point, see Section 9.4, "Extending the YRCRelatedTaskGroups Extension Point".
17. 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. For more information about extending the YRCRelatedTaskExtensionContributor extension point, see Section 9.6, "Extending the YRCRelatedTasksExtensionContributor Extension Point".

18. 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.
19. To create a new categories extension element, right-click on the related task for which you want to define the category and select New > categories. The categories extension element gets created.
20. Select the categories extension element. The Extension Element Details panel displays.
21. In the Extension Element Details panel, enter the properties of the categories extension element.
22. To create a new category extension element, right-click the categories extension element that you created and select New > category. The category extension element gets created. Just as the Rich Client Platform supports the definition of multiple categories for each related task, you can define multiple category extension elements under the category extension element.
23. In the Extension Details panel, enter the properties of the category extension element.
24. 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. For more information about extending the YRCRelatedTaskCategories extension point, see Section 9.3, "Extending the YRCRelatedTaskCategories Extension Point".

# 9.3 Extending the YRCRelatedTaskCategories Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.YRCRelatedTaskCategories extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCRelatedTaskCategories extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of the YRCRelatedTaskCategories extension point.
9. To create a new categories extension element, right-click on com.yantra.yfc.rcp.YRCRelatedTaskCategories extension and select New > categories. The categories extension element gets created.

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. To create a new category extension element, right-click on categories extension element that you created in the previous step and select New > category. The category extension element gets created. You can create multiple category elements under the categories extension element.
13. In the Extension Details panel, enter the properties of the category extension element.
14. To create a new tasks extension element, right-click on category extension element that you created and select New > tasks. The tasks extension element gets created.
15. Select the tasks extension element. The Extension Element Details panel displays.
16. In the Extension Element Details panel, enter the properties of the tasks extension element.
17. To create a new task extension element, right-click on tasks extension that you created in the previous step and select New > task. The task extension element gets created. You can create multiple task elements under the tasks extension element. You can associate multiple related tasks to the same category.
18. In the Extension Details panel, enter the properties of the task extension element.
19. In id, enter the id of the related task that you want to have in this particular category. These categories are defined by extending the YRCRelatedTaskCategories extension point. For more information about extending the YRCRelatedTaskCategories extension point, see Section 9.3, "Extending the YRCRelatedTaskCategories Extension Point".

# 9.4 Extending the YRCRelatedTaskGroups Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.YRCRelatedTaskGroups extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCRelatedTaskGroups extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of the YRCRelatedTaskGroups extension point.
9. To create a new `group` extension element, right-click on com.yantra.yfc.rcp.YRCRelatedTaskGroups extension and select New > groups. The groups extension element gets created.

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. To create a new group extension element, right-click on groups extension element that you created and select New > group. The group extension element gets created. You can create multiple group elements under the groups extension element.
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.

## 9.5 Extending the YRCRelatedTasksDisplayer Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".

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. To create a new relatedTasksDisplayer extension element, right-click on com.yantra.yfc.rcp.YRCRelatedTasksDisplayer extension and select New > relatedTasksDisplayer. The relatedTasksDisplayer extension element gets created.
10. Select the relatedTasksDisplayer extension element. The Extension Element Details panel displays.
11. To specify the implementation class, do any of the following
    a. 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.
    b. Click on the class* hyperlink. The Java Attribute Editor window displays.
       - Enter the name of the class that implements the com.yantra.yfc.rcp.IYRCRelatedTasksDisplayer interface.
       - Click Finish. The new class gets automatically created.

# 9.6 Extending the YRCRelatedTasksExtensionContributor Extension Point

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.

Prior to implementing this 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 get the editor information for opening the new related task in the application shipped editor:

1. Navigate to the task you want to extend, in the Rich Client Platform application.
2. Through the Rich Client Platform Extensibility Tool view the screen information. The editor information is displayed in the screen information window.

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:

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

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".

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.YRCRelatedTasksExtensionContributor extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCRelatedTasksExtensionContributor extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of the YRCRelatedTasksExtensionContributor extension point.
9. To create a new relatedTasksExtensionContributor extension element, right-click on com.yantra.yfc.rcp.YRCRelatedTasksExtensionContributor extension and select New > relatedTasksExtensionContributor. The relatedTasksExtensionContributor extension element gets created. You can create multiple relatedTasksExtensionContributor elements but make sure that for one relatedTasksExtensionContributor extension element you specify a unique editor. Otherwise, the system randomly selects an extension contributor from the specified extension contributors.
10. Select the relatedTasksExtensionContributor extension element. The Extension Element Details panel displays.
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.
  - Enter the name of the class, that implements the com.yantra.yfc.rcp.IYRCRelatedTasksExtensionContributor interface.
  - Click Finish. The new class gets 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;
}
```

# 9.7 Enabling Custom Dialog Boxes Through an Extension Point for Rich Client Platform Applications

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:

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.

# 10

# Creating Commands for Rich Client Platform Applications

## 10.1 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:

***Table 10–1 Command Element's Attribute List***

| 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. For more information about overriding commands, see Section 10.3, "Overriding Commands". |
| 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. For more information about defining namespaces, see Section 10.2, "Defining Namespaces". |
| inputNamespace | Specify the namespace of the input XML model.<br>**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:<br>1. The target XML model for the screen is retrieved.<br>2. The input namespace attributes are then retrieved from the target XML model.<br>However, if the Rich Client Platform application sets the input namespace programmatically when calling a command, the input namespace specified is ignored.<br>The latter is the more commonly used approach in Rich Client Platform applications. |

***Table 10–1 Command Element's Attribute List***

| Field | Description |
|---|---|
| URL | If you want to invoke your own API instead of APIs or services of the Selling and Fulfillment Foundation, 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 *Selling and Fulfillment Foundation: Installation Guide*. |
| 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. For more information about prototype mode, see Section 2.13, "Prototype Mode for Rich Client Platform Applications". |
| 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:** Sterling Commerce recommends that you do not make changes to the configuration file shipped with Selling and Fulfillment Foundation. 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, see Appendix 17.14.4, "Registering Commands File".

# 10.2 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:

***Table 10–2 Namespace Element's Attribute List***

| Field | Description |
|---|---|
| name | Specify a unique name for the namespace. |
| type | Specify the type of namespace, depending on whether the template is to be used as input or output. For example, "input" or "output".<br>**Note:** If you are creating the namespaces for the wizard rules:<br>• Specify type attribute as "input" if you want to take inputs from the user as the target model of the screen.<br>• 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.<br>**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>
```

# 10.3 Overriding Commands

The Overriding Commands feature enables a user to call its custom API instead of APIs provided by Selling and Fulfillment Foundation 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 Selling and Fulfillment Foundation.

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>
```

# 11

# Defining and Overriding Hot Keys in Rich Client Platform Applications

## 11.1 Phase 1: Defining a Hot Key Command

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.commands extension point from the list.
6. Click Finish.
7. Select the org.eclipse.ui.commands extension point. The Extension Details panel displays.

8. In the Extension Details panel, set the properties of the org.eclipse.ui.commands extension point.
9. 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.
10. Select the category extension element. The Extension Element Details panel displays.
11. In id*, enter the unique identifier of the category.
12. In name*, enter the name of the category.
13. 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.
14. Select the command extension element. The Extension Element Details panel displays.
15. In id*, enter the unique identifier of the command.
16. In name*, enter the name of the command.
17. In categoryId, enter the identifier of the category to which the command belongs, if applicable.
18. Click  to save the changes.

## 11.2 Phase 2: Defining a Hot Key Binding

To define a new key binding for the category that you created in the org.eclipse.ui.commands extension point:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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

    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. For information about defining commands, see Section 11.2, "Phase 2: Defining a Hot Key Binding".
15. Click  to save the changes.

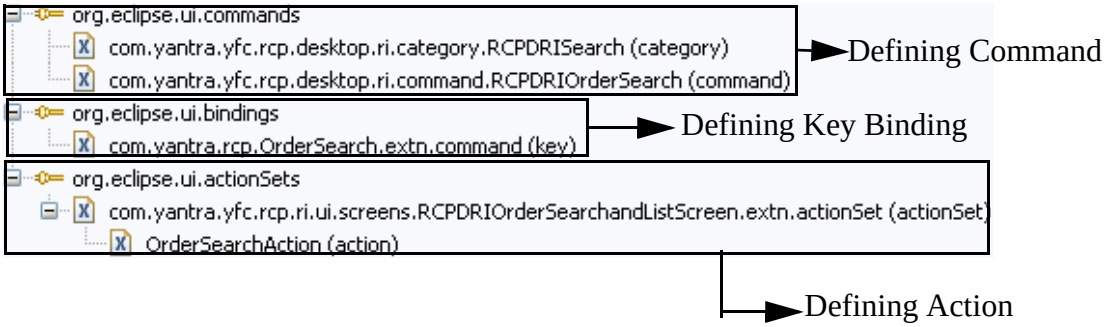
## 11.3 Phase 3: Defining a Hot Key Action

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 Appendix 17.12, "Creating New Actions".

**Note:** In the definitionId field, enter the identifier of the command that you created. For more information about defining commands, see Section 11.2, "Phase 2: Defining a Hot Key Binding".

After defining the command, key binding, and action, the structure of the `plugin.xml` file of the plug-in project is shown in Figure 11–1.

***Figure 11–1 Plugin.xml File***




## 11.4 Overriding Hot Keys

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 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

    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. 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.
14. In commandId, enter the identifier of the command whose hot key you want to override.

15. Click  to save the changes.

## 11.4.1 Disabling Related Task Hot Keys

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.

# 12

# Merging Templates for Rich Client Platform Applications

## 12.1 Merging Input and Output Templates

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 with Selling and Fulfillment Foundation are stored on the server in the namespaces folder of the Rich Client Platform plug-in and PCA plug-in directories.

The PCA templates are located at:

```
<INSTALL_DIR>/repository/xapi/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, let us consider the following 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:

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 Selling and Fulfillment Foundation PCA plug-in. For example, getOrderLineDetails.xml.
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>
```

# 13

# Related and Shared Tasks in Rich Client Platform Applications

## 13.1 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 Chapter 9, "Creating Related Tasks for Rich Client Platform Applications".

## 13.2 Hiding Existing Related Tasks

You can hide the related tasks on the screen by removing the related tasks from the list specified in the YRCRelatedTasksDisplayer extension point. For more information about the YRCRelatedTasksDisplayer extension point, see Section 9.5, "Extending the YRCRelatedTasksDisplayer Extension Point".

## 13.3 Registering Shared Tasks

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.
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.

***Figure 13–1 Java Attribute Editor Window***

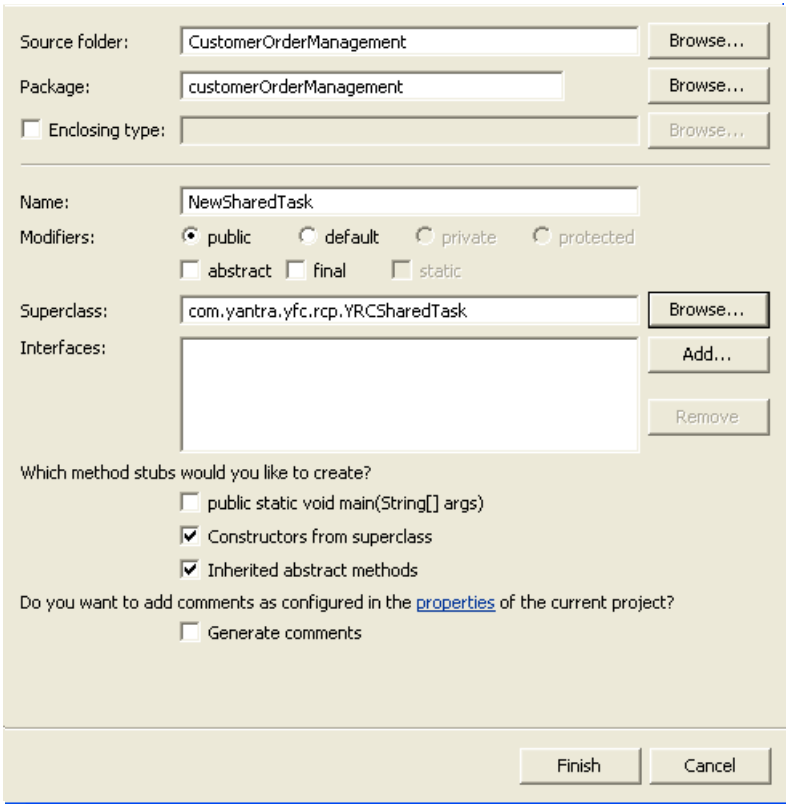



***Table 13–1 Java Attribute Editor Window***

| 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. |

*Table 13–1 Java Attribute Editor Window*

| 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.

## 13.4 Using Shared Tasks

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:

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.

# 14

# Defining Themes for Rich Client Platform Applications

## 14.1 Defining New Themes

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, see Appendix 17.14.2, "Registering Theme File". 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:

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 Chapter 3, "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 Table 14–1.

***Table 14–1 Font Element Attribute List***

| 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 Table 14–2.

***Table 14–2 BackgroundColor Element Attribute List***

| 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. For ForegroundColor element attribute list, see Table 14–2.
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 is 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>
```

## 14.2 Defining Themes for Controls

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:

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.

## 14.2.1 Applying Themes to Non-editable Text Boxes

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.

# 15

# Menus and Custom Controls for Rich Client Platform Applications

## 15.1 Adding and Removing Menus in Rich Client Platform Applications

Menu configuration contains the standard Selling and Fulfillment Foundation 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 from the screen, see the *Selling and Fulfillment Foundation: Application Platform Configuration Guide*.

## 15.2 Customizing the Menu View Through the YRCMenuDisplayer Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 YRCMenuDisplayer extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCMenuDisplayer extension point. The Extension Details panel is displayed.
8. In the Extension Details panel, enter the properties of YRCMenuDisplayer 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. 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).

# 16

# Setting the Extension Model, Configuring SSL and SSO for Rich Client Platform Applications

## 16.1 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 Chapter 10, "Creating Commands for Rich Client Platform Applications".
- Defining Namespaces—In the `<Plug-in_id>_commands.ycml` file, define the new namespaces for a screen. For more information about defining namespaces, see Section 10.2, "Defining Namespaces".

> **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.

## 16.2 Configuring SSL for Rich Client Platform Applications

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created when setting up the development environment.

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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.YRCHostNameVerifier extension point from the list.
6. Click Finish.

7. Select the com.yantra.yfc.rcp.YRCHostNameVerifier extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of the YRCHostNameVerifier extension point.
9. 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.
10. Select the hostNameVerifier extension element. The Extension Element Details panel displays.
11. 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.
12. Implement the verify (String hostName, SSLSession session) method and return the value "true" if the host name is acceptable. Otherwise, return the value "false".

## 16.3 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.

## 16.3.1 Client Settings for SSO Configuration

Perform the following steps:

1. Create a new plug-in for SSO authentication. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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
```

## 16.3.2 Server Settings for SSO Configuration

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.

# 17

# Rich Client Platform General Concepts Reference

## 17.1 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.

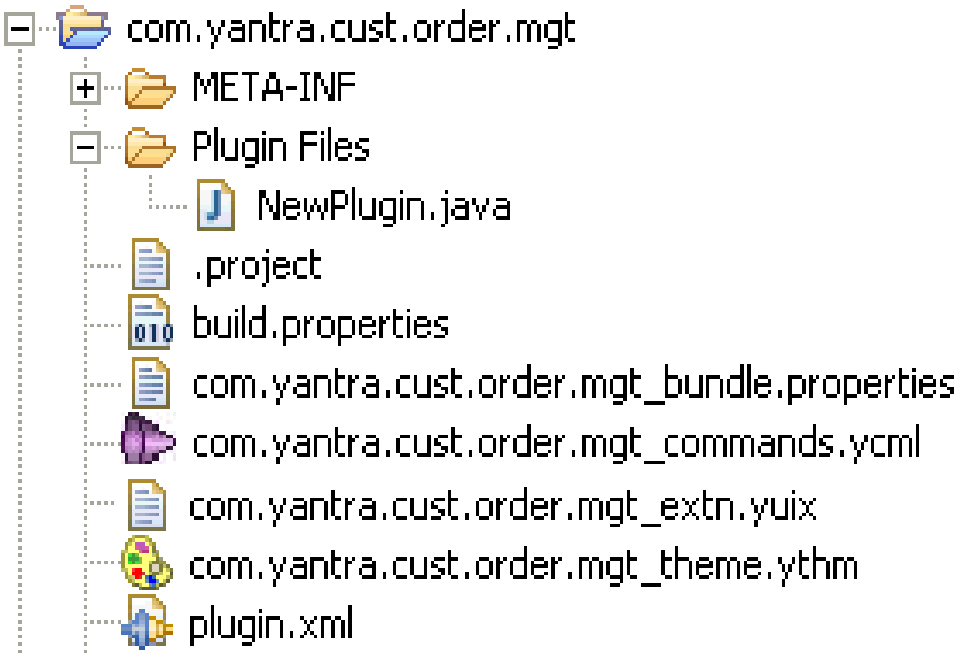
In mid-90's, internet technologies such as Hyper Text Markup Language (HTML) started emerging very fast. Due to its simplicity and very low TCO, internet technologies had tremendous impact in the way business applications were delivered. Initially, HTML was only used for displaying information. However, 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 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.

Figure 0–1 illustrates the Rich Client Platform Architecture.

*Figure 0–1 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.

## 17.2 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 Eclipse 3.2 release. Eclipse Rich Client Platform enables application developers to deliver rich internet applications that run on platforms such as Windows, Linux, and so forth.

## 17.3 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.

## 17.4 Plug-In Manifest Editor

The Plug-in Manifest Editor provides a single UI for editing the manifest and other plug-in related files. The Plug-in Manifest Editor contains following sections:

### 17.4.1 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.

### 17.4.2 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.

### 17.4.3 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.

### 17.4.4 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.

### 17.4.5 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.

### 17.4.6 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.

### 17.4.7 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`.

### 17.4.8 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.

### 17.4.9 Build.properties

The `build.properties` file contains all files and directories that are required by a plug-in at the runtime.

## 17.5 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.

## 17.6 YRCApplicationInitializer Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 YRCApplicationInitializer extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCApplicationInitializer extension point. The Extension Details panel is displayed.
8. In the Extension Details panel, enter the properties of YRCApplicationInitializer 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. 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.

# 17.7 YRCContainerToolbar Extension Point

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:

1. Start the Eclipse SDK.

2. In the navigator view, expand the plug-in project that you created. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 (for Sterling Call Center and Sterling Store)
     - class: Specify fully classified path to a Java class that must implement the interface IYRCContainerToolbarProvider interface.

# 17.8 YRCPostWindowOpenInitializer Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 YRCPostWindowOpenInitializer extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCPostWindowOpenInitializer extension point. The Extension Details panel is displayed.
8. In the Extension Details panel, enter the properties of YRCPostWindowOpenInitializer 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, 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.

## 17.9 YRCJasperReport Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 YRCJasperReport extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCJasperReport extension point. The Extension Details panel is displayed.
8. In the Extension Details panel, enter the properties of YRCJasperReport extension.

9. 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.

## 17.10 YRCContainerTitleProvider Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 YRCContainerTitleProvider extension point from the list.

6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCContainerTitleProvider extension point. The Extension Details panel is displayed.
8. In the Extension Details panel, enter the properties of YRCContainerTitleProvider 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. 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.

## 17.11 YRCMessageDisplayer Extension Point

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:

1. Start the Eclipse SDK.
2. In the navigator view, expand the plug-in project that you created. For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 YRCMessageDisplayer extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCMessageDisplayer extension point. The Extension Details panel is displayed.
8. In the Extension Details panel, enter the properties of YRCMessageDisplayer 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.
   - 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.

10. 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 (for Sterling Call Center and Sterling Store).
    - class: Specify the Java class that must implement the interface IYRCMessageDisplayer interface.

## 17.12 Creating New Actions

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:

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

   For more information on how to create a plug-in project, see Section 3.3, "Creating a Plug-In Project".
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 > Plugin Manifest Editor.
4. Select the Extensions tab.
5. Click Add. From the New Extension window, select `org.eclipse.ui.actionSets` extension point from the list.
6. Click Finish.
7. Select the `org.eclipse.ui.actionSets` extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of `org.eclipse.ui.actionSets` extension point.
9. Right-click on `org.eclipse.ui.actionSets` extension and select New > actionSet. The "actionSet" extension element gets created.

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 gets 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.

- Click Browse. The Select Type pop-up window displays. Select the class that extends the `YRCAction` class or `YRCRelatedTaskAction` class.
- 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 action class automatically 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 action class automatically displays. This helps you to easily manage your directory structure.
- In Name, enter the name of the action class.
- In Superclass, click Browse, the Superclass Selection window displays.
- Select the YRCAction class or YRCRelatedTaskAction class from the list and click OK.
- Check the Constructors from superclass box. The system automatically 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 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.

## 17.13 Registering a Plug-In

Every plug-in that is a part of the Rich Client Platform application must be registered. Various features of Selling and Fulfillment Foundation such as localization, theming, configuration, UI extension, and so forth 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. For more information about creating Rich Client Platform plug-in, see Section 3.4, "Running the Rich Client Platform Plug-In Wizard".

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);
            }
     }
}
```

## 17.14 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);
	}
}
```

### 17.14.1 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. For more information on how to register a plug-in, see Appendix 17.13, "Registering a Plug-In".

## 17.14.2 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. For more information on how to register a plug-in, see Appendix 17.13, "Registering a Plug-In".

### 17.14.3 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. For more information on how to register a plug-in, see Appendix 17.13, "Registering a Plug-In".

### 17.14.4 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. For more information on how to register a plug-in, see Appendix 17.13, "Registering a Plug-In".

## 17.14.5 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. For more information on how to register a plug-in, see Appendix 17.13, "Registering a Plug-In".

### 17.14.6 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. For more information on how to register a plugin, see Section 17.13, "Registering a Plug-In" .

## 17.15 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.

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.

## 17.16 Custom Data Formatting

Rich Client Platform enables you to perform custom data formatting.

For example, let us consider that in the Phone Number field, the user enters the number as 6175677890 and presses the Tab key. You want to format this number and display 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:

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

   For more information about creating a plug-in project, see Section 3.3, "Creating a Plug-In Project".

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.YRCDataFormatter extension point from the list.
6. Click Finish.
7. Select the com.yantra.yfc.rcp.YRCDataFormatter extension point. The Extension Details panel displays.
8. In the Extension Details panel, enter the properties of the com.yantra.yfc.rcp.YRCDataFormatter extension point.
9. In ID, enter a unique identifier for the com.yantra.yfc.rcp.YRCDataFormatter extension point. This is a mandatory field.
10. 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.
11. Select the dataFormatter extension element. The Extension Element Details panel displays.
12. In the Extension Details panel, enter the properties of the dataFormatter extension element.
13. In attributeBinding*, enter the name of the XPath attribute whose value you want to custom format. For example, the attribute binding mentioned in Example can be set as DayPhone.
14. 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.

15. Open the newly created implementation class in the Java editor.
16. 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 as explained in Example can be:

```
public YRCFormatResponse getFormattedValue(String attributBinding, String
value) {
       YRCFormatResponse response = null;
//validForDataType(String)mehtod 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;
}
```

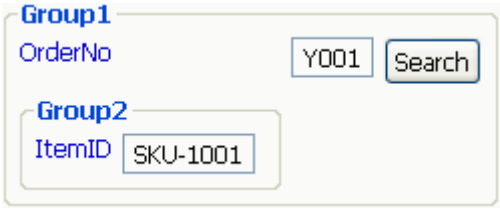
17. Override the inherited abstract getDeformattedValue() method. Write the deformatted value of the field you want to store in the XML and return the deformatted value. For example, the deformatting logic as explained in Example can be:

```
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;
}
```

## 17.17 Siblings

Siblings are the first level children of the parent. For example, let us consider the following scenario:

*Figure 17–1 Sample Siblings*




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).

# 17.18 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.

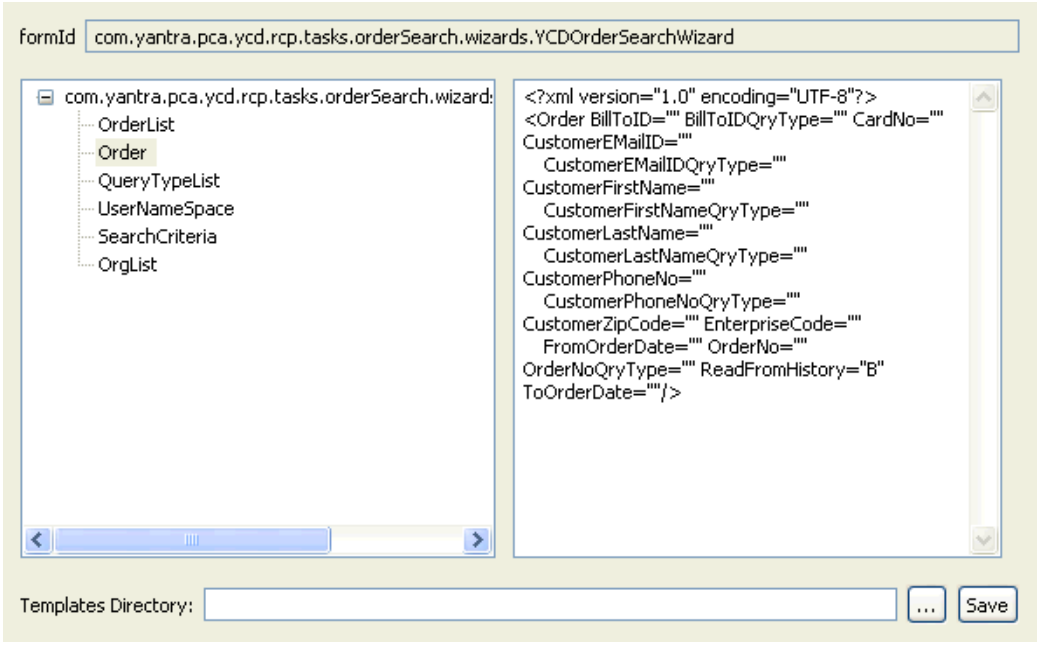
## 17.18.1 Viewing Screen Models

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:

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 as shown in Figure 0–2.

***Figure 0–2 The Screen Models Window***




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.

### 17.18.1.1 Saving Models as Templates

To save existing models as templates:

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.

Click Save. The system stores the models of each screen as templates in their respective folders.

# Index