

WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects

What are JavaServer Faces?

JavaServer Faces in practice

Service Data Objects in
practice



Ueli Wahli
Gabriel Cohen
Matthew Perrins
Luis Sanchez Acera
Mads Zandersen



International Technical Support Organization

**WebSphere Studio 5.1.2 JavaServer Faces and
Service Data Objects**

July 2004

Note: Before using this information and the product it supports, read the information in “Notices” on page xix.

First Edition (July 2004)

This edition applies to Version 5.1.2 of WebSphere Studio Application Developer and WebSphere Studio Site Developer, together with Version 5.1 of WebSphere Application Server.

© Copyright International Business Machines Corporation 2004. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xix
Trademarks	xx
Preface	xxi
The team that wrote this redbook	xxii
Acknowledgments	xxiii
Become a published author	xxiv
Comments welcome	xxiv
Part 1. Introduction to JavaServer Faces	1
Chapter 1. JavaServer Faces overview	3
Introduction	4
Model-view-controller architecture	4
Development user roles	6
Rapid application development	6
JavaServer Faces scenario	7
JSF specification	7
Concepts	8
JSF application	8
Container independence	10
Faces servlet	10
Restore Component Tree	11
Apply Request Values	11
Process Validations	11
Update Model Values	11
Invoke Application	12
Render Response	12
Common Event Processing	12
Example Faces Request	13
Faces configuration file	13
Navigation	14
Sample navigation scenario	15
Managed beans	17
UI components	18
Events	19
Binding	19
Converters and validators	20
JSF component libraries	20

UI component tree	21
JavaServer Faces page example	21
Summary	24
Chapter 2. JSF support in WebSphere Studio	25
JSF support in WebSphere Studio	26
Overview	26
Creating a JSF project	27
Project organization	28
Creating a JSF page	29
New Faces JSP File wizard	30
JSF page structure	31
Page code	31
UI components in page code	32
Managed beans in page code	32
PageCodeBase class	33
Page templates and fragments	33
Web perspective	34
Page Designer	34
Page Designer modes	35
Context menu	36
Palette	36
Faces components drawer	37
Data drawer	37
Drag and drop	37
Customization	38
Attributes view	39
Value binding	40
Action binding	40
Navigation tab	41
Format tab	41
Validation tab	42
All tab	43
Page Data view	43
Context menu	44
Data objects	45
JSP scripting	47
Action list	49
Drag and drop	49
Tree view	49
Client Data view	49
Quick Edit view	50
Java events	50
JavaScript events	51

Snippets	51
Page initialization	51
Testing, debugging, and deploying the JSF project	52
Server targeting.	52
Test environment	53
Considerations and guidelines	54
Creating a JSF page and managed beans	54
Renaming a JSF page	54
Navigation rules	54
Scope variables	55
Broken link warning	55
Deleting a JSF page	55
Moving a JSF page to another folder	55
Moving the page code to another package	56
Recreating a JSF page	56
Refactoring managed beans	56
Deleting components	57
Deleting managed beans	57
Renaming a project	58
Creating a welcome page	58
Summary	59
 Chapter 3. JSF calculator example	 61
Simple example that shows the basic facilities	62
Calculator	62
Preparation	62
Development steps	62
Creating a new Web project	62
Import the calculator bean	63
Creating a JSF page for the calculator	63
Let's examine what has been generated	63
Creating the base layout	64
Implementing component attributes and validation	67
Testing the validation	71
Binding the frontend to the calculator	71
Testing the binding	73
Invoking the business logic of the calculator	73
Implementing an error page	75
Implementing page navigation	76
Implementing a validator	78
Implementing a value change event	80
Implementing internationalization	82
Implementing page initialization	83

Complete solution.	84
Chapter 4. JSF components.	85
Overview	86
Component model	86
Renderers	87
JSF libraries	87
Standard JSF libraries	88
Using Faces in Studio	88
Studio operations	88
Component attributes and events	89
Common attributes	89
Binding	91
Value binding	91
Method binding	92
Attributes view	93
Common attribute editors.	93
Common events	94
Logic with Quick Edit.	95
Component libraries	95
Libraries included with WebSphere Studio	96
Core library.	97
HTML library.	101
Form: <h:form>	102
Output: <h:outputText>	106
Output - Formatted Text: <h:outputFormat>	106
Input: <h:inputText>	108
Input - Password: <h:inputSecret>	111
Input - Hidden: <h:inputHidden>	114
Input - Text Area: <h:inputTextArea>	115
Display Error: <h:message>	118
Display Errors: <h:messages>	120
Label: <h:outputLabel>	122
Command - Hyperlink: <h:commandLink>	124
Radio Button Group: <h:selectOneRadio>	127
Check Box: <h:selectBooleanCheckBox>.	130
Check Box Group: <h:selectManyCheckBox>	132
Combo Box: <h:selectOneMenu>	134
List Box - Single Select: <h:selectOneListBox>	136
List Box - Multiple Select: <h:selectManyListBox>	139
Data Table: <h:dataTable>	141
IBM Extension Library	146
Command - Button: <hx:commandExButton>.	147

Link: <hx:outputLinkEx>	150
Image: <hx:graphicImageEx>	153
File Upload: <hx:fileUpload>	155
Horizontal Rule: <hx:outputSeparator>	159
Panel Group Box - HTML Panel: <hx:jspPanel>	159
Panel Group Box - Snap to Border: <hx:panelLayout>	160
Panel Group Box - List: <hx:panelBox>	162
Panel Menu Bar: <hx:panelActionBar>	164
Panels - Tabbed: <odc:tabbedPannel>	167
Rich Text Area: <r:inputRichText>	174
Media - Generic A/V Player: <hx:playerGenericPlayer>	177
Media - Macromedia Flash Player: <hx:playerFlash>	178
Media - Macromedia Shockwave Player: <hx:playerShockwave>	180
Media - RealOne Player: <hx:playerRealPlayer>	182
Media - Windows Media Player: <hx:playerMediaPlayer>	184
Data Table component extensions	186
Complete solution.	187
Part 2. JSF application development	189
Chapter 5. JSF banking application	191
Banking application overview	192
JSF banking application architecture	192
Model	193
Bank	193
Customer	193
Account	193
TransRecord	193
Banking	193
Model implementation	194
Controller	194
Exceptions	194
View	195
Implementation details	195
EJBBANK database	196
Sample data	197
Preparation	197
Implementing the banking application with JSF	198
Define a utility project	198
Create an enterprise application	199
Create a Web project for the banking application	199
Preparing managed beans with customer and transaction data	201
Creating the home page	201
Navigation	204

Implementing the action	205
Testing the home page and customer retrieval	205
Implementing the error page	205
Implementing the list of accounts	206
Test the design	208
Implement navigation	208
Define the account managed bean	209
Defining a hyperlink	209
Test account selection	211
Implementing the account details page	211
Navigation logic	214
Test account management	216
Implementing the transfer result page	216
Test transfer	217
Implementing the transaction list	217
Test list transactions	220
Implementing paging and transaction details	220
Test the paging and transaction details	222
Implementing custom validation	222
Test the validation	223
Displaying the customer image	223
Running the JSF banking application	225
Complete solution	226
Chapter 6. JSF application design and common patterns	227
Roles	228
Page author	228
Application developer	228
Component developer	228
Application architect	228
Tools vendor	228
Application architecture	229
Goals and principles	229
N-tier architecture	230
Presentation tier	231
Business logic tier	231
Data access tier	232
Database tier	232
Design considerations	232
Web-tier and the MVC design pattern	232
Designing for portability	233
Other considerations	234
Session scope management	234
Data conversion	236

Data validation	236
Internationalization	236
Custom components	237
Custom renderers for other client platforms	237
What are patterns?	238
Create, read, update, delete (CRUD)	239
Master to detail	239
Form to form wizard	240
List to list	240
Example problem	240
Setting up you development environment for patterns	241
Create, read, update and delete (CRUD) pattern	242
Pattern design	242
Read	242
Create, update and delete	243
Creating the table of products	244
Creating CRUD pages	245
Read page with relational record list	245
Displaying the table	246
Apply style changes	247
Adding navigation to CRUD	249
Building the update/delete part of CRUD	250
Building the create part of CRUD	252
Testing the CRUD application	255
Implementing concurrency	255
Updating the update/delete function	255
Testing concurrency	256
Improving error messages	256
Summary	257
Form wizard pattern	257
Pattern design	257
Building the form wizard pattern	259
Creating the JavaBean	260
Creating the form pages	260
Defining the form address page	262
Adding navigation	263
Completing the other pages of the Wizard	265
Order page	265
Delivery address	265
Adding business logic	266
Completing the form wizard pattern	267
Testing of form wizard pattern	268
Master to detail pattern	268
Pattern design	268

Working pattern	269
Building the master to detail pattern	270
Creating the queries for the pattern	270
Laying out the page.	271
Adding event logic.	273
Testing	274
Complete solution.	274
Summary	274
Chapter 7. JSF and Web services	277
Web services concepts	278
Documentation	278
Discovering and importing a Web service	279
Client code generation	281
Using Web services with JSF pages	281
Example Web service application	282
Creating the project.	282
Finding the Web service	283
Invoking the Web service from JSF	285
Input page	286
Output page	287
Test the Web service invocation	288
Complete solution.	288
Best practices.	288
Chapter 8. JSF and Struts	289
Introduction to Struts	290
Comparing JSF and Struts	291
Choosing Struts and/or JSF	292
Combining JSF and Struts	293
Struts-Faces request.	294
Adding Faces support to a Struts application	294
Creating a new Struts-Faces page	295
Converting a Struts page into a Struts-Faces page	296
Summary	296
Part 3. Service Data Objects	297
Chapter 9. SDO concepts	299
Introduction to SDO	300
Data object.	300
Data graph	301
Data mediator	301
WebSphere Studio support for SDO	301

Relational record	301
Relational record list	302
Developing an SDO object	302
SDO wizard	302
Relational record and relational record list	302
Record properties	303
New connection	304
Column selection and other tasks	306
Generated and updated files	307
SDO metadata files	307
Connection information	308
Web deployment descriptor	309
Page code	309
Data source	310
Database connections and data sources	311
Development time connections	312
Creating a connection	312
Runtime connections	313
SDO components	314
Data retrieval	314
Displaying a relational record list	315
Execution	318
Retrieving a single record	318
Data update	320
Update and delete	320
Creating the form for update and delete	321
Action code	322
Create	323
Auto key generation	325
Deleting SDO objects	325
Using SDO in JSPs	326
Relational record list	326
Generated tags	326
Relational record	327
Generated tags	327
Running the JSP with SDO	328
SDO API	329
Data object	329
Important methods	329
Query syntax	330
Data graph	330
Important methods	331
Data mediator	331
Important methods	331

WDO implementation	332
Retrieving a record	333
Dynamic queries	333
Updating a record	334
Deleting a record	335
Inserting a record	335
Adding a new data graph	335
Adding to an existing data graph	336
Calculating a new primary key	336
Joined records	337
Single transaction commit	337
Debugging data graphs	338
Summary	339
Chapter 10. SDO banking application	341
SDO banking application overview	342
SDO banking application architecture	342
Model	343
Controller	343
View	343
Implementation details	343
Preparation	344
Implementing the banking application with SDO	345
Create an enterprise application	345
Creating the Web project	345
Creating the customer ID query page	346
Retrieving the customer name using a SDO	347
Creating the database connection	348
Creating the database query	349
Generated code	351
Action logic	352
Test the home page	352
Implementing the error page	353
Implementing the list of accounts	353
Retrieving the customer's accounts using a join	354
Adding the account list to the page	356
Test the list accounts page	359
Implementing the account details page	359
Creating the Page Data	359
Designing the account details page	361
Test the account details page	366
Implementing the list transactions page	367
Test the list transactions page	368
Implementing the transfer result page	368

Test the transfer result page	369
Implementing single transaction commit	369
Implementing a transaction details page	371
Test the transaction details page	372
Implementing a table maintenance application	372
Customer maintenance using row edit support	372
Account maintenance using row select support	374
Account listing using row category support	376
Complete solution	377
Part 4. JSF advanced topics	379
Chapter 11. JSF and portlets	381
Concepts	382
Portal Toolkit	383
Portlet API	383
Deciding which API to use in your applications	383
Future directions for WebSphere Portal JSR 168 support	384
JavaServer Faces support	384
Navigation	384
Actions of command buttons and hyperlinks	384
Portlet modes and JSF	384
JSP page model	385
Known problems	385
Unsupported components	386
SDO support	386
Examples	387
Example 1: JSF calculator in a portlet	387
Create the Portlet project	387
Import the CalculatorBean	389
Design of the calculator JSP	390
Bind the frontend to the calculator bean	391
Create an error page	392
Implementing the page navigation	393
Invoking the calculate method	394
Creating a portlet server for testing	394
Testing the calculator application	395
Example 2: Migration of the JSF banking example	396
Creating and configuring a server	396
Define a utility project	399
Creating the project and the portlet	400
Importing the JSF banking application	400
Fixing the JSF pages	403
Configuring the portlet	404

Testing the portlet banking application	404
Fixing the JSF application	405
Testing the complete application	406
Complete solution	407
Chapter 12. JSF and EGL	409
EGL concepts	410
JSF and EGL page handlers	410
Creating a sample EGL JSF application	411
Preparation	411
Workspace preferences	411
Creating an EGL Web project	412
EGL build descriptor	413
Creating a JSF page for EGL	414
Generated EGL file	415
Compiled Java file	415
Implementing the home page	416
Creating an SQL record for customer	416
Completing the welcome page	418
Adding the action logic	420
Testing the welcome page	421
Debugging EGL code	421
Displaying the customer information	422
Creating an SQL record for the list of accounts	424
Displaying the list of accounts	425
Testing the list of accounts	427
Selecting an account and passing the account number	428
Implementing account management	428
Creating an SQL record for the account	429
Displaying the account information	430
Completing the account details page	431
Implementing account operations	433
Implementing SQL functions for transaction records	434
Implementing the list of transaction records	436
Running the EGL banking application	437
Complete solution	437
EGL considerations	438
Tailoring data items	438
Chapter 13. Faces Client framework	439
Faces Client framework introduction	440
Capabilities	440
Faces Client concepts	440
Emitters	441

SDO for JavaScript	441
JavaScript Library	441
DiffGram	441
DiffHandlers	442
Faces Client tools	443
Faces Client page	443
Page Designer	444
Creating client data	444
Client Data view	444
EMF artifacts	446
SDO4JS emitters	446
ODC taglib and properties file	447
Configuring client data objects	447
Binding client data to controls	448
Editing client component events	450
Testing Faces Client applications	450
Faces Client components	450
Data Grid: <odc:dataGrid>	450
Graph: <odc:graphDraw>	453
Tree View: <odc:tree>	454
Web Service: <odc:webService>	456
Sample Faces Client application: Data Grid	457
Creating the user interface	457
Improving the functionality with update	460
Implementing server-side update of the client changes	461
Creating the difference handler for the department object	462
Creating the difference handler for the employee object	463
Triggering the update	464
Test the application	466
Sample Faces Client application: Graph	468
Creating the user interface	468
Configuring the graph	468
Running the graph example	470
Using the grouping functions	471
Other options	472
Sample Faces Client application: Tree View	472
Processing tree operations	473
Running the Tree View	474
Sample Faces Client application: Web Service	475
Adding the Web service project	475
Copying the JavaBeans required for Page Data	476
Creating the JSP to run the Web service	476
Adding the Web Service component	477

Invoking the Web service	478
Running the Web service client application	478
Complete solution.	478
Chapter 14. Extending the JSF framework	479
Custom components	480
Overview	480
Scenario	481
Designing the component	482
Web project structure	483
Development steps	485
JSF tag library	486
UI component model	488
Tag library component	490
User interface renderer	495
Faces configuration file	498
Initial testing	499
How to package JSF components	503
Ant build script.	503
Summary.	505
Custom converters	506
Converter interface	506
Registering the converter	506
Converter example	507
Studio support for custom components	509
Adding a custom component to the palette.	510
Create a plug-in project	510
Extending the Studio palette	512
Prepare early for packaged build.	513
Adding extensions	513
Testing the plug-in	516
Creating a deployable plug-in	517
Ant build script	518
Component visualizer and attributes.	519
Other extensions.	519
Testing custom components	519
Installing custom components in Studio	520
Using custom components in a Web application	520
Summary.	523
Part 5. Appendixes	525
Appendix A. Installation instructions	527
DB2 Universal Database Enterprise Edition 8.1.4	528

WebSphere Application Server	529
WebSphere Application Server cumulative fix 3 (or fix 4)	529
WebSphere Studio Application Developer	530
WebSphere Portal Toolkit	531
Setting up the environment for this redbook	533
Creating the EJBBANK database	533
Sample content	533
Setting up server targeting	534
Setting up the WebSphere test environment	535
Defining the JSFServer	535
Configuring the JSFServer	535
Deployment to WebSphere Application Server 5.1	538
Configure WebSphere Application Server	538
Authentication alias	538
JDBC driver path	538
JDBC driver	539
Data source	539
Save the configuration	541
Stop and start the server	541
Installing an enterprise application	541
Installation process: Multiple-steps	542
SDO runtime	542
Starting the enterprise application	543
Testing the enterprise application	543
Appendix B. Additional material	545
Locating the Web material	545
Using the Web material	546
System requirements for downloading the Web material	546
How to use the Web material	546
Workspace projects	547
Application Developer setup	548
Importing a solution enterprise application	548
Deploying enterprise applications	548
Abbreviations and acronyms	549
Related publications	551
IBM Redbooks	551
Other publications	552
Online resources	552
How to get IBM Redbooks	553
Help from IBM	553

Index 555

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

developerWorks®
ibm.com®
z/OS®
ClearCase®

CICS®
DB2 Universal Database™
DB2®
IBM®

Rational®
Redbooks™
Redbooks (logo) ™
WebSphere®

The following terms are trademarks of other companies:

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM Redbook describes the new JavaServer Faces (JSF) technology for building Web applications and the WebSphere® Studio Application Developer tools supporting it. JSF is based on the Java™ Specification Request (JSR) 127, of which the first release was finalized in March, 2004.

The goal of JSF technology is to provide a complete framework to develop and deploy Web applications easily and rapidly. The framework provides solutions for a variety of tasks found in Web application development, such as user interface design, application navigation and flow, session and object management, validation and error feedback, internationalization, and container portability.

This redbook covers all of the core features of JSF Web development and the tools that are available in WebSphere Studio Application Developer 5.1.2 to design, develop, deploy, and test JSF Web applications. These tools include:

- ▶ Visual tools for user interface design
- ▶ Wizards and editors to create and manage all the objects of an application
- ▶ Extensions to the framework to enrich the client user experience
- ▶ Test environment to test and debug JSF Web applications
- ▶ Integration of Service Data Objects with JSF applications

Service Data Objects (SDO) is a data programming architecture and API for the Java platform that unifies data programming across data source types, provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data.

This book is divided into four parts:

- ▶ Part 1 presents the core concepts of JavaServer Faces. It describes the framework, the IBM® WebSphere Studio JSF tooling and support, a simple example of developing a JSF Web application, and the set of JSF components.
- ▶ Part 2 describes how to develop JSF Web applications using an existing backend or a Web service, and how to integrate JSF with Struts.
- ▶ Part 3 introduces Service Data Objects and details how to use them in JSF Web application. A complete development example is included for reference.
- ▶ Part 4 discusses advanced topics, such as the development of portlets using JSF, integration with enterprise generation language (EGL), and the Faces Client Framework. It also shows how to extend the JSF framework to enrich your Web applications and to improve development productivity.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.



Ueli Wahli is a Consultant IT Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 19 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide on application development, object technology, WebSphere Application Server, and lately WebSphere Studio products. In his ITSO career, Ueli has produced over 30 IBM Redbooks. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology.

Gabriel Cohen is a Software Engineer in Research Triangle Park, North Carolina. He has three years of experience in modernizing legacy applications in the Enterprise Transformation area, working on products such as Host On-Demand and Host Access Transformation Services. He holds a bachelor's degree from Duke University. His areas of expertise include Java, J2EE, WebSphere Portal, and WebSphere Studio plug-in development.

Matthew Perrins is a Consultant IT Specialist working for the IBM Software Services for WebSphere organization as a Solution Architect driving the implementation of successful solutions across Europe. He works out of the Hursley development laboratory in the UK and has worked for IBM since 1989. He has spent the last seven years developing and designing applications using Web technologies using WebSphere Application Server and WebSphere Studio. His areas of expertise include Java, J2EE, WebSphere Studio tools, and he has extensive knowledge of .Net and Microsoft® development technologies.

Luis Sanchez Acera is an IT Specialist at the AMS Organization in Spain. He has four years of experience developing Web applications, specially using the Java programming language. He holds a bachelor's degree in Computer Science from the Politechnic University of Madrid (UPM). His areas of expertise include Java, the J2EE framework, application servers, and integration with other technologies, such as Visual Basic.

Mads Zandersen is a Senior IT Specialist in software technical sales in IBM Denmark. His areas of expertise cover application development practices within the application construction area. Mads has worked for IBM since 1997 and has 18 years of experience with software development, primarily within administrative systems development.

Acknowledgments

Thanks to the following people for their contributions to this project:

- ▶ Steven Stansel, IBM Rational, for resolving the project funding
- ▶ George DeCandio, IBM Raleigh, for organizing technical support
- ▶ Dan O'Connor, Alasdair McDowell, and Scott Paxton, for JSF support
- ▶ Thomas Mutdosch, Kevin Bauer, and Kevin Williams, for SDO support
- ▶ Laurent Hasson, Robert Zitelli, Robert Gallagher, Masato Noguchi, and Qun Zhou for JSF Client framework and Portlet support
- ▶ Sanjay Chandru, Kevin Sloop, and YC Lui, for JSF with EGL support
- ▶ Dave Artus and John Catlin from IBM Software Services for WebSphere
- ▶ Gabrielle Velez for editing support

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

ibm.com/redbooks

- Send your comments in an Internet note to:

redbook@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 80-E2
650 Harry Road
San Jose, California 95120-6099



Part 1

Introduction to JavaServer Faces

In Part 1 of the book we introduce the core concepts behind JavaServer Faces, or JSF. We describe the framework, introduce the IBM WebSphere Studio JSF tooling, guide you through the creation of a simple, yet fully-functional JSF application, and outline the set of existing JSF components.

- ▶ In Chapter 1 we introduce the JSF basic concepts.
- ▶ In Chapter 2 we discuss the support that WebSphere Studio Application Developer Version 5.1.2 provides to develop JSF Web applications.
- ▶ In Chapter 3 we develop a simple JSF Web application step-by-step.
- ▶ In Chapter 4 we describe all the JSF components—the components provided by the framework and the IBM custom components.



JavaServer Faces overview

This chapter lays a foundation for understanding and utilizing the JavaServer Faces framework. It outlines the features of the framework, its history, and its behavior and artifacts.

Introduction

JavaServer Faces (JSF) is a framework for developing Java Web applications. The JSF framework aims to unify techniques for solving a number of common problems in Web application design and development, such as:

- ▶ **User interface development**

JSF allows direct binding of user interface (UI) components to model data. It abstracts request processing into an event-driven model. Developers can use extensive libraries of prebuilt UI components that provide both basic and advanced Web functionality.

- ▶ **Navigation**

JSF introduces a layer of separation between business logic and the resulting UI pages; stand-alone flexible rules drive the flow of pages.

- ▶ **Session and object management**

JSF manages designated model data objects by handling their initialization, persistence over the request cycle, and cleanup.

- ▶ **Validation and error feedback**

JSF allows direct binding of reusable validators to UI components. The framework also provides a queue mechanism to simplify error and message feedback to the application user. These messages can be associated with specific UI components.

- ▶ **Internationalization**

JSF provides tools for internationalizing Web applications, supporting number, currency, time, and date formatting, and externalizing of UI strings.

JSF is easily extended in a variety of ways to suit the requirements of your particular application. You can develop custom components, renderers, validators, and other JSF objects and register them with the JSF runtime.

Model-view-controller architecture

Applications built with JavaServer Faces are intended to follow the **model-view-controller** (MVC) architectural pattern (Figure 1-1). According to the MVC pattern, a software component should separate its business logic along the following lines:

Model	Encapsulates the state and behavior of the application
View	Renders the model
Controller	Processes user events and drives model and view updates

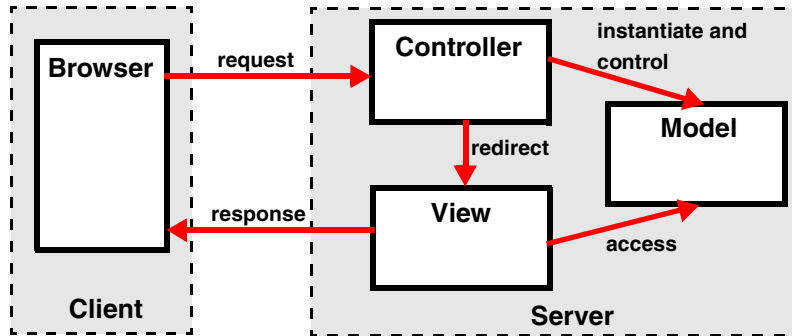


Figure 1-1 MVC Web application architecture

We can group the important components of JSF as they fall into these categories (Figure 1-2):

► **Model**

Managed beans make up the model of a JSF application. These Java beans typically interface with reusable business logic components or external systems, such as a mainframe or database.

► **View**

JSPs make up the view of a JSF Web application. These JSPs are created by combining model data with predefined and custom-made UI components.

► **Controller**

The FacesServlet, which drives navigation and object management, makes up most of a JSF application's controller. Event listeners also contribute to the controller logic.

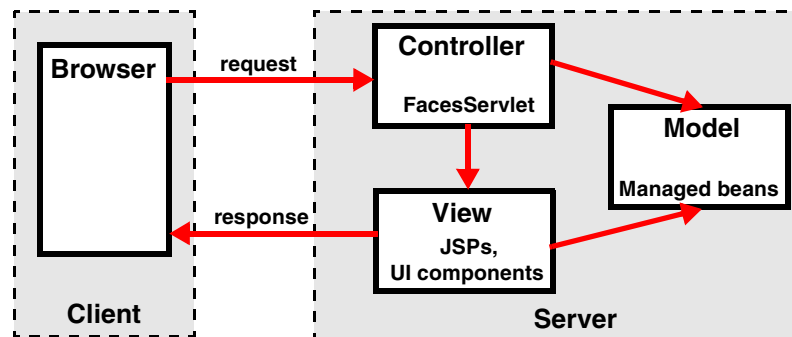


Figure 1-2 JSF's MVC breakdown

Development user roles

Because of its MVC architecture, and hence, the clear boundaries between the pieces of its business logic, JSF supports the creation of **user roles** in the development cycle.

A user role is the classification of a user as having a specific set of skills, and therefore, a specific set of tasks which he is expected to execute. This prevents the common problem of assigning tasks to a user which are outside of his typical domain, which is an inefficient use of his time, and decreases overall productivity.

In software development, user roles break down the development tasks into different groups associated with specific types of developers. This breakdown allows for parallel, distributed development of the application. The basic JSF development user roles are outlined here:

- ▶ **Application developer**—These users create the business logic, or model, of the application. They provide object-oriented access to backend data, manage persistence of the model, and expose interfaces for common backend tasks. They must have Java skills and knowledge of any backend integration.
- ▶ **Component developer**—These users create reusable JSF components, such as UI components, renderers, and validators. They are responsible for encoding the data from the application developer's interfaces, as well as decoding the request information to update the model. They require skills in J2EE and JSF APIs.
- ▶ **Page author**—These users are responsible for the look and feel, or view, of the application. They create Web pages using the component developer's components and the application developer's business logic and data objects. These users usually own the navigation and model life cycle configuration. They must be proficient in HTML, CSS, JavaScript, and other browser-based technologies.

Rapid application development

JSF is commonly viewed as a rapid application development (RAD) framework. RAD is the idea that the application development process can be made more efficient through:

- ▶ Rapid prototyping
- ▶ Early, iterative functional testing of designs
- ▶ Re-using standard software components
- ▶ Using visual design tools

The advantages of using a RAD framework such as JSF are that overall development costs are reduced, that a customer gets to see immediate results

and approve designs, and that the application does not become an anachronism due to changing business conditions during development.

JSF qualifies as a RAD framework due to the standardization it provides, the tooling built on its specification, and its extensive libraries of off-the-shelf components.

JavaServer Faces scenario

An application developer has written some Java classes which provide an object-oriented interface for a set of tables in a database. The developer packages the code into a JAR file and hands it off to a page author.

A page author creates a new JSF application and imports the supplied JAR file from the application developer. He creates a new JSF page for interfacing with the database. He adds a class from the JAR to the page as a managed bean, then drags it on to the page. A set of fields are created with which to update the record. He drags a command button onto the page from the Web design palette, and associates it with a supplied update method in the supplied business logic. He adds navigation rules to display other pages depending on the success of the update operation. Finally, he tests the JSF page by running it on the application server.

The end user accesses the Web application for the database application. He can now update the database directly through a familiar Web interface.

The application developer has no Web skills, the page author has no Java skills, and the end user does not even know he is updating a database. This scenario took under an hour.

JSF specification

JavaServer Faces was created by Java Specification Request (JSR) 127. IBM, Sun, and BEA, along with many other industry leading companies participated in the development of the specification. The final release of the specification was approved in March, 2004.

The current JavaServer Faces specification resides at:

<http://www.jcp.org/en/jsr/detail?id=127>

The JSR 127 Expert Group owns the living specification for JavaServer Faces. In addition, as dictated by the Java Community Process (JCP), anyone may contribute and provide feedback to the specification.

IBM provides its own implementation of the JSF specification, which was used in the development of this book. IBM also provides extensions to the framework along the lines described in the specification, such as custom UI controls.

Due to the open nature of the specification, tooling is being created for JSF across the industry which is implementation-agnostic. IBM has provided tools for JSF in WebSphere Studio Application Developer Version 5.1.1 since December 2003 (beta code), and we specifically discuss the tools as they exist in WebSphere Studio Application Developer Version 5.1.2.

Concepts

We briefly outline the artifacts and behavior which comprise a JavaServer Faces application. For further reading, these concepts are defined in more detail in the JSF specification itself.

JSF application

A JSF application is a J2EE Web application, so a JSF application meets all the basic requirements of a Web application as defined in the J2EE specification. Figure 1-3 shows the basic structure of a JSF application.

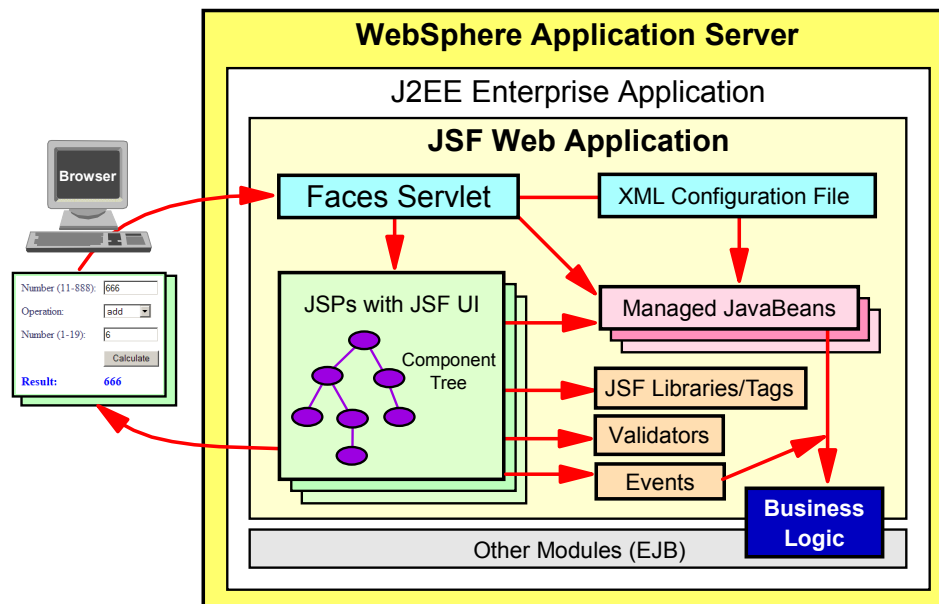


Figure 1-3 JSF application structure

The main pieces of a JSF application are:

- ▶ **JSF pages**—JSPs built from JSF components, where each component is represented by a server-side class.
- ▶ **Faces Servlet**—One servlet (FacesServlet) controls the execution flow.
- ▶ **Configuration file**—An XML file (faces-config.xml) that contains the navigation rules between the JSPs, validators, and managed beans.
- ▶ **Tag libraries**—The JSF components are implemented in tag libraries.
- ▶ **Validators**—Java classes to validate the content of JSF components, for example, to validate user input.
- ▶ **Managed beans**—JavaBeans defined in the configuration file to hold the data from JSF components. Managed beans represent the data model and are passed between business logic and user interface. JSF moves the data between managed beans and user interface components.
- ▶ **Events**—Java code executed in the server for events (for example, a push button). Event handling are used to pass managed beans to business logic.

The structure of a simple JSF application, as created in WebSphere Studio, is shown in Figure 1-4.

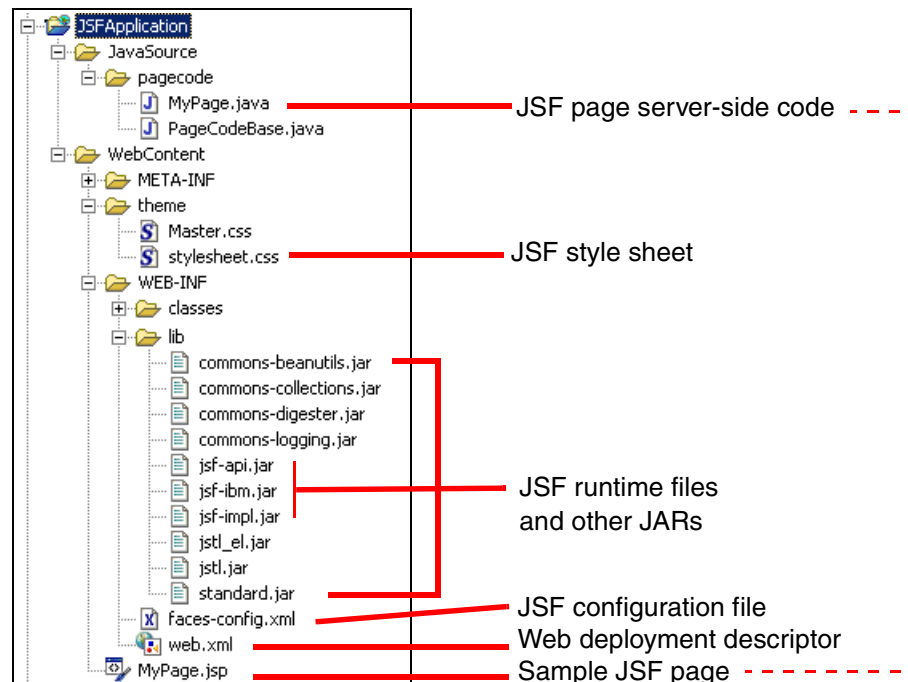


Figure 1-4 JSF application folder tree

Container independence

Though this book focuses on the use of JSF in the J2EE framework, JSF applications are not required to execute in a Web application server. A layer of abstraction, the ExternalContext, is used to access container-specific objects in a unified way. This way, an application's model can be written once and execute in both a servlet container and a portlet container.

Business logic is written to use Faces mechanisms, and not container mechanisms, for most tasks, so it is not likely you will even use the ExternalContext directly.

Faces servlet

The FacesServlet (`javax.faces.webapp.FacesServlet`), is the heart of a JSF application. It is responsible for processing and using the metadata in the `faces-config.xml` to handle the model life cycle and navigation. It receives UI events and maps them to the appropriate business logic.

The FacesServlet processes external input, also known as the request, in a set of phases, which eventually result in the production of new output, or a response (Figure 1-5).

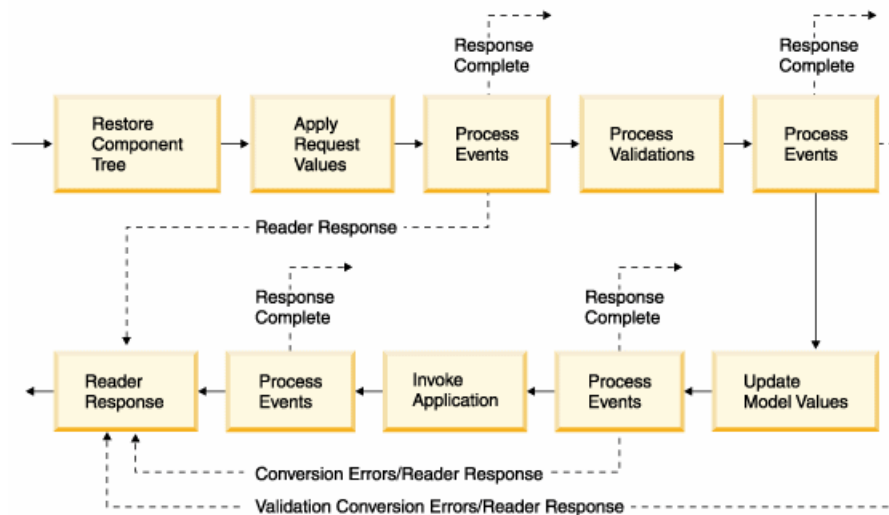


Figure 1-5 JavaServer Faces life cycle

It is important to understand the progression of these phases, because they dictate the flow of information in application and the sequencing of common events. We briefly outline the phases for you.

Restore Component Tree

This phase is responsible for reconstituting the server-side representation of the view contained in the page which was just submitted:

- ▶ First, the servlet must identify the original page if this was a request originating from a Faces page. This is done by analyzing the URI.
- ▶ The servlet then attempts to restore a saved view corresponding to that page. If there are no query parameters or post data in the request, the servlet invokes the render response phase immediately.
- ▶ At the end of this phase, if applicable, the working instance of the view consists of the previously saved view at the end of the last render response phase.

Apply Request Values

The purpose of this phase is to cause all of the components in the view to update themselves from the request data:

- ▶ Each component contains its own logic for analyzing the request parameters, headers, cookies, and so on to find the latest value corresponding to its control.
- ▶ Conversion problems or other types of events may be generated in this phase by invalid data in the request. Errors may also terminate the cycle and invoke the render response phase directly.
- ▶ At the end of this phase, the working instance of the view is updated as much as possible with the latest data from the request.

Process Validations

In this phase, any validators which have been registered on a component by the page author are invoked:

- ▶ Validation logic may create new events to be processed later, or they may terminate the cycle and go directly to the render response phase.
- ▶ At the end of this phase, all components in the view should have their current data validated.

Update Model Values

In this phase, all components update the model with their new component state data:

- ▶ When this phase is reached, it can be assumed that the incoming request contained only valid and correct data.
- ▶ Each component is responsible for identifying the model object to which it corresponds. Once identified, the component takes its own value, having

been updated from the request and validated, and replaces the model's current value. Again, this process may generate events for later processing, or escape to the render response phase.

- ▶ At the end of this phase, the model should be completely updated with all of the latest data from the view.

Invoke Application

During this phase, the JSF runtime handles application level events, such as a form submission or link, by invoking business logic written by the application developer:

- ▶ On a form submission, the runtime must evaluate the action result code from the submitting component and determine the subsequent page.
- ▶ The runtime either evaluates the business logic method bound to the component or uses the component's action attribute directly to obtain the result code. Then the runtime analyzes the current state and result code to determine the next page. Once the next page has been discovered, JSF sets the current view as corresponding to this new page.
- ▶ At the end of this phase, we have finished all of the control functions and are ready to begin creating the new view.

Render Response

In the final phase of Faces processing, the runtime must cause the response to render back to the client, as well as store the state of view to be used in future processing:

- ▶ This behavior can vary depending on the implementation, but in the servlet and JSP implementation, the response is rendered by dispatching a request to the new page.
- ▶ When the page is invoked, the tags in that page use components to display model data as Web controls. The tags also store the overall component structure which makes up the view for later use. This phase concludes JSF request processing.

Common Event Processing

In these phases, any enqueued events targeted for the preceding phase are dequeued and processed by their corresponding registered listeners. It is possible that event processing itself generates more events. These are generic phases for situational event and error handling.

Example Faces Request

Suppose an end user is currently viewing a Web page, `updateCustomer.jsp`, which is used to update a customer record. The page contains name and address fields, which are bound to a customer model object, and a submit button, which is bound to an update method. There is a navigation rule defined which forwards to the page `customerList.jsp` on a successful update. The user changes the customer's last name from *Smith* to *Smythe*, and clicks *Submit*.

Table 1-1 shows the processing steps.

Table 1-1 Example of Faces request processing phases

Phase	Actions
Restore Component Tree	JSF looks up the stored view information for <code>updateCustomer.jsp</code> , and successfully finds it.
Apply Request Values	The component bound to the customer object's last name property pulls the value <i>Smythe</i> from the request and stores it in itself.
Process Validations	The validator associated with the last name component determines that the new value is all letters, and hence, valid.
Update Model Values	The last name component updates the last name field in the customer JavaBean object.
Invoke Application	JSF invokes the update method, which writes the customer data from the customer object back into the database. JSF sees that the method returned success and determines that the next page is the <code>customerList.jsp</code> .
Render Response	JSF forwards to <code>customerList.jsp</code> , which is rendered to the client. While rendering, JSF stores the component structure from the page as the saved view.

Faces configuration file

The `faces-config.xml` is the repository for all the application-level configuration data in a JSF application:

- ▶ This file is where the list of managed data is maintained, navigation rules are encoded, custom JSF extensions are registered, and other application settings are stored.
- ▶ This file is used as input to the `FacesServlet` when it starts up.
- ▶ A sample `faces-config.xml` file is shown in Figure 1-6.

```

<faces-config>
  <lifecycle>
    <phase-listener>
      com.ibm.faces.webapp.ValueResourcePhaseListener
    </phase-listener>
  </lifecycle>
  <managed-bean>
    <managed-bean-name>pc_EntryPage</managed-bean-name>
    <managed-bean-class>pagecode.EntryPage</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
  <managed-bean>
    <managed-bean-name>pc_DataViewPage</managed-bean-name>
    <managed-bean-class>pagecode.DataViewPage</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
  <navigation-rule>
    <from-view-id>/EntryPage.jsp</from-view-id>
    <navigation-case>
      <from-outcome>viewdata</from-outcome>
      <to-view-id>/DataViewPage.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>

```

Figure 1-6 An example faces-config.xml file

Note: The phase-listener declared in the faces-config.xml example is automatically added by WebSphere Studio to new JSF applications. You do not have to define or use this object.

Navigation

Page authors are responsible for organizing the navigation through the pages in a JSF application by authoring and maintaining a set of *navigation rules*.

Navigation rules glue the various JSPs of a the application into a sequential flow of pages. During the Invoke Application phase of the life cycle, the JSF runtime uses the library of navigation rules associated with the application to determine which JSP to display next. These rules are stored in the faces-config.xml as <navigation-rule> elements. Each rule consists of one or more *navigation cases* (<navigation-case>), which combine a condition and a resulting page to display. JSF can use any combination of the following aspects of the current application state to test a navigation condition:

► Current page—<from-view-id>

This is the name of the JSP page containing the component which resulted in this navigation event. If the current page is omitted, the rule is a global rule that applies to all pages. This property may include the wildcard character *.

► Action name—<from-action>

If a component invokes business logic for its action, this is the method name of the implementation attached to an action (command button or hyperlink). If the action name is omitted, the rule applies regardless of the invoked method. This property may include the wildcard character *.

► Result string—<from-outcome>

If a component invokes business logic for its action, this is the return string from the invoked method. If the component does not invoke business logic, this is the action attribute value in the component's tag. This result string is the *symbolic name* of the target JSP.

► Target JSP—<to-view-id>

The FacesServlet redirects to the JSF page specified by the <to-view-id> element of the first case in the list, which has all conditions evaluated as true.

Sample navigation scenario

Here is a source code snippet from a JSF page, `updateCustomer.jsp`, which contains two navigation controls (Figure 1-7).

```
<hx:commandExButton type="submit" value="Submit"
    id="button1" action="#{pc_UpdateCustomer.doUpdateAction}"/>
<br>
<h:commandLink id="link1" action="details">
    <h:outputText id="text1" styleClass="outputText"
        value="Customer Details"/>
</h:commandLink>
```

Figure 1-7 JSF navigation example page source

The command button invokes the `doUpdateAction` method in the `pc_UpdateCustomer` managed bean, whereas the command link always uses the action result `details`.

The `doUpdateAction` method processes the request and returns the symbolic name of the navigation rule, for example:

```
return "success";
```

Here is the corresponding set of navigation rules clipped from the faces-config.xml (Figure 1-8).

```
<navigation-rule>
  <navigation-case>
    <from-outcome>error</from-outcome>
    <to-view-id>/error.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/updateCustomer.jsp</from-view-id>
  <navigation-case>
    <from-action>#{pc_UpdateCustomer.doUpdateAction}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/customerList.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>details</from-outcome>
    <to-view-id>/viewCustomer.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Figure 1-8 Sample navigation rules

- ▶ The first rule is global, which is to say, it is not limited to a specific JSP. The absence of the <from-view-id> element indicates that the navigation rule is global. The solitary case for this rule redirects to the error.jsp whenever the error result code is received.
- ▶ The second rule only applies while on updateCustomer.jsp. The first case only applies when the invoked action is doUpdateAction in the pc_UpdateCustomer managed bean. It redirects to customerList.jsp on the result code success. The second case is not limited to a specific action. It redirects to viewCustomer.jsp upon the result code details.

Using this system, JSF decouples the flow of the application from the business logic and the UI. This means that the page author can modify the sequencing of the pages without having to modify the business logic. For instance, the destination used upon receiving the success code can be changed to viewCustomer.jsp without modifying the JSP or the doUpdateAction method implementation.

In the event that the available conditions are not sufficient, navigation can be driven directly from business logic. Application Developer provides a convenience method to assist with manual navigation, which is described in Chapter 2, “JSF support in WebSphere Studio” on page 25.

Managed beans

Application developers must encapsulate and expose interfaces for the business logic of their applications in *managed beans*. Once registered in a JSF application's `faces-config.xml`, the runtime instantiates these objects as appropriate, initializes them if specified, and destroys them when they are no longer needed.

JSF bases its life cycle management of these objects on a property called *scope*. The concept of scope should be familiar to those with background in J2EE development. Scope comes in four varieties:

request	The bean is stored in the <code>HttpServletRequest</code> and is persisted between two pages or on a page reload.
session	The bean is stored in the <code>HttpSession</code> and exists for the life of a user's session.
application	The bean is stored in the <code>ServletContext</code> and exists for the life cycle of the JSF application, with one instance for all users.
none	The bean is not stored anywhere.

Each managed bean is declared in the `faces-config.xml` as a `<managed-bean>` XML element (Figure 1-9).

```
<managed-bean>
  <description>data record</description>
  <managed-bean-name>testData</managed-bean-name>
  <managed-bean-class>itso.jsf.TestData</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>property1</property-name>
    <value>test1</value>
  </managed-property>
  <managed-property>
    <property-name>property2</property-name>
    <value>test2</value>
  </managed-property>
</managed-bean>
```

Figure 1-9 Sample managed bean declaration

The `<managed-bean-name>` tag declares the name of the bean so it can be looked up at runtime. The `<managed-bean-class>` tag refers to the Java class defining this managed bean. The scope of a bean is stored in the `<managed-bean-scope>` tag.

You may optionally initialize specific bean properties with fixed values by declaring `<managed-property>` tags. These elements must be specified with a bean property name (`<property-name>`) and an initialization value (`<value>`). You may also initialize `java.util.List` or `java.util.Map` properties here; please refer to the JSF specification for details.

Note: In JSF applications built with WebSphere Studio, there are managed beans automatically generated and maintained which have names that begin with `pc_`. These correspond to the page code beans for a JSF page, and are discussed in Chapter 2, “JSF support in WebSphere Studio” on page 25.

UI components

JavaServer Faces *UI components* are the individual elements which make up the view of a JSF application. These components represent the state and behavior of a wide variety of functional UI elements, from the most basic, such as a button or input field, to the complex, such as a grid layout composite, made up of other UI components.

The JSF UI component architecture is extremely flexible because the rendering is decoupled from the actual component. This *renderer* is a separate object which can be plugged into the component depending on the requirements of the client platform. That is to say, the same UI components can be rendered in full HTML for a Web browser client, or in WML for a handheld device. Therefore, JSF views achieve device independence (Figure 1-10).

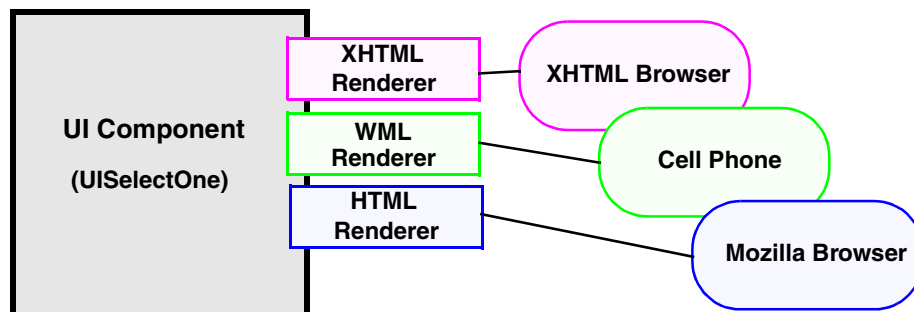


Figure 1-10 JSF rendering to multiple clients

Component developers are responsible for the construction of UI components and renderers. These extensions can then be packaged for use by third party JSF developers. See Chapter 14, “Extending the JSF framework” on page 479 for guidelines on creating and packaging JSF extensions.

Events

JSF provides an event-listener framework which allows server-side handlers to be tied directly to UI component events. When an event is generated, it stores a reference to the generating component. The phase in which the event is to be handled must also be specified. Event listeners are registered on the components which process the events at the end of their specified phases.

Two standard Faces events and listeners are provided for some of the standard UI components:

- ▶ **ActionEvent**—This event is automatically generated by a `UICommand` component, when a button or hyperlink is clicked. Events of this type are always handled in the Invoke Application phase by an `ActionListener` registered on their components, followed by the default application `ActionListener`.

Action event methods return a string, which is the symbolic name specified in a navigation rule. If the method returns `null` or an empty string, the same JSP is redisplayed reusing its component tree. If a non-empty string value is returned, the navigation cases are examined to determine the next JSF page. A new component tree is generated, even if the navigation rule goes back to the same JSP.

- ▶ **ValueChangeEvent**—This event is automatically generated by an `UIInput` component when validation of new local value succeeds in the Process Validations phase. Events of this type are handled immediately after the Process Validations phase by a `ValueChangeListener` registered on their components.

This framework dramatically simplifies the process of responding to client events in server code, negating the requirement for complicated HTTP request parsing.

Binding

UI components can be directly associated with model objects for various purposes. The mapping of a UI component to a model object is called a ***binding***.

Binding which defines a relationship for viewing and modifying data is called a ***value binding***. This value binding is used by a component to automatically update its internal state from the model state during the Render Response phase, and to update the model state with its internal state during the Update Model Values phase.

Binding which maps a UI component's action event to a business logic method is called a ***method binding***. A component with a specified method binding has its designated action method invoked during the Invoke Application phase.

We discuss binding specifics and syntax in more detail in Chapter 4, “JSF components” on page 85.

Converters and validators

When a component has defined a value binding, there is a representation of the data stored both in the component and in the model. These representations may not be identical. For instance, a model object may have a currency field, which a text input component treats as a string. The renderer is normally responsible for conversion between these two formats. However, this conversion function may be delegated to a **converter**. JSF comes with a set of predefined converters which are sufficient for basic conversion tasks. Component developers can implement and register new JSF converters (see “Custom converters” on page 506).

Predefine converters are the number converter (to integer and floating point) and the date converter (to different data formats).

JSF supports validating the local value of a UI component before updating the model. For instance, you may want to restrict the value of a numeric variable to only positive integers and not update the model when given a bad value. During the Process Validations phase, JSF invokes any **validators** that have been bound to a component. These validators check the new value using business logic, and generate errors and events if any problems are encountered. JSF comes with a number of predefined validators for common scenarios, and new validators can be created by component developers as the need arises.

Predefined validators are length validators and range validators (long integer and double floating point values).

JSF component libraries

Libraries of components are made available to JSF pages by including the libraries’ TLD files in page declarations. Component are then included in a JSF page by using the JSP tags declared in those TLDs.

There are a number of component libraries already in existence:

- **Core**—These are the universal components used in JSF views, even if the JSF application is not rendering to an HTML client. This library is supplied by Sun. The TLD URI is:
<http://java.sun.com/jsf/core>
- **HTML**—These components correspond to the set of controls defined by the HTML specification. This library is also provided by Sun. The TLD URI is:
<http://java.sun.com/jsf/html>

- **Extended HTML**—These components are for functions that go above and beyond basic HTML. Some components in this library are intended to replace those defined in the HTML library while adding more functionality. This library is provided by IBM. The TLD URI is:

http://www.ibm.com/jsf/html_extended

- **Browser framework**—These components are used to provide rich client functionality in the browser. These components make substantial use of JavaScript, and even use some browser plug-ins, to achieve their function. This library is also provided by IBM. The TLD URI is:

<http://www.ibm.com/jsf/BrowserFramework>

You may use any number of standard, in-house developed, and third-party ISV libraries when developing a JSF applications.

UI component tree

As discussed previously, JSF stores a representation of the current view on the server when rendering the response. This representation of the view is a `UIViewRoot` object, which is a container for all of the UI components used in the JSF page. This object organizes the components into a tree data structure. Composite components, such as a form or table, have sub-nodes for each component they contain, and so on. In this manner, the tree should contain all of the Faces objects on the page.

The stored view also has a property, `viewId`, which directly corresponds to the `from-view-id` and `to-view-id` properties in JSF navigation rules. In the standard servlet environment, this identifier is the relative path to the JSP file which corresponds to this view object.

The UI component tree is initially created during a Render Response phase by invoking the `createView` method on the application's `ViewHandler`. During the other application phases, the component tree is used as the working set of components which are to be acted upon.

JavaServer Faces page example

JSP pages, created and maintained by page authors, make up the highest-level view objects in the JSF framework. JSF JSPs are collections of UI components and referenced managed beans. Figure 1-11 shows a simple JSF page.

Update Customer

[Customer Details](#)

First Name:

Last Name:

Figure 1-11 An example of a JSF page

Let us walk through the source for the JSP page and take notice of all of the interesting artifacts:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%-- jsf:codeBehind language="java"
      location="/JavaSource/pagecode/UpdateCustomer.java" --%>
<%-- /jsf:codeBehind --%>
```

- Here we use the `jsf:codeBehind` comment tag. This tag is only used by WebSphere Studio to maintain the linkage between the JSP file and the source of the page code managed bean. This is not functionally important for the rest of the page.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://www.ibm.com/jsf/html_extended" prefix="hx"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1" %>
<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<TITLE>Update Customer</TITLE>
<LINK rel="stylesheet" type="text/css" href="theme/stylesheet.css"
      title="Style">
</HEAD>
```

- This is the page header. Here we include the all of the taglibs defining the tags for the UI components we want to use in this page. Note that we include both standard tag libraries from Sun and also an extended tag library from IBM. We also include a reference to the supplied style sheet defining CSS classes for the JSF UI components.

```
<f:view>
  <BODY>
    <hx:scriptCollector id="scriptCollector1">
      <h:form styleClass="form" id="form1">
```


- Here the `<f:view>` tag denotes the root of the UI component tree on this page. The `<hx:scriptCollector>` is an IBM extension, which you do not have to know about, because it is automatically maintained by the WebSphere Studio tools. The form begins, so we can start including interactive controls.

```
<h2><h:outputText styleClass="outputText" id="text5"
    value="Update Customer"></h:outputText></h2>
<br>
<h:commandLink id="link1" action="details">
    <h:outputText id="text6" styleClass="outputText"
        value="Customer Details"></h:outputText>
</h:commandLink>
```

- Here we have the header of the page displayed as a static string. Next we have a link that returns the action code details. Note how UI components can contain other components.

```
<table><tbody><tr>
    <td align="left">First Name:</td>
    <td style="width: 5px">&nbsp;</td><td>
        <h:inputText id="text1"
            value="#{customerData.firstName}"
            styleClass="inputText" required="true">
            <hx:validateConstraint regex="AlphabetOnly" />
        </h:inputText>
        <h:message styleClass="message" id="message1"
            for="text1"/>
    </td></tr><tr>
    <td align="left">Last Name:</td>
    <td style="width: 5px">&nbsp;</td><td>
        <h:inputText id="text2"
            value="#{customerData.lastName}"
            styleClass="inputText" required="true">
            <hx:validateConstraint regex="AlphabetOnly" />
        </h:inputText>
        <h:message styleClass="message" id="message2"
            for="text2"/>
    </td></tr></tbody></table>
```

- This is the table with the first and last name input fields. Notice how each input field has a value binding to a `customerData` managed bean property. These fields display the current value from the managed bean when the page loads, and update those properties when the page is submitted. Each component also uses nested validators to make sure the field contents are alphabetic.

Finally, there are `<h:message>` components that display any queued validation errors associated with the component (matched by its `id` attribute) specified in their `for` attribute.

```
<h:messages styleClass="messages" id="messages1"/><BR>
```

- This tag includes another message component, which is not bound to a specific component. This component displays all the error messages. In most applications you either work with individual `<h:message>` components or with one generic `<h:messages>` component.

```
        <hx:commandExButton id="button1" styleClass="commandExButton"
            type="submit" value="Submit"
            action="#{pc_UpdateCustomer.doUpdateAction}"/>
    </h:form>
</hx:scriptCollector>
</BODY>
</f:view>
</HTML>
```

- Finally, there is a Submit button that submits the form and invokes the `doUpdateAction` method in the page code managed bean in the Invoke Application phase. The rest of the remaining page tags are closed and the JSP source ends.

Summary

In this chapter we described the basic concepts of JSF, independent of a particular implementation. Understanding these concepts is important when we describe the implementation of JSF in Application Developer 5.1.2.



JSF support in WebSphere Studio

This chapter provides a description of the JavaServer Faces support in WebSphere Studio Application Developer.

The major pieces of the JSF support are:

- ▶ Web projects with JSF development environment
- ▶ Page code, server-side Java classes for event (action) logic
- ▶ Page Designer with Palette, Attributes view, Page Data view, and Quick Edit view

We also describe considerations and guidelines when renaming, moving, refactoring, and deleting JSF components.

JSF support in WebSphere Studio

WebSphere Studio Application Developer supports the creation, development, test, and deployment of JavaServer Faces applications. Application Developer's JSF support is tightly integrated into the existing set of Web tools, tailoring the maintenance of Web applications and the design of Web pages. The JSF-related tools improve the usability of the Faces framework by supplying intuitive interfaces for interacting with Faces artifacts. These tools ease the development of JSF code through intelligent Java code generation and maintenance. The tools also extend the functions of the JSF framework by providing support for a variety of new components.

Overview

Figure 2-1 gives an overview of the JSF support in Application Developer:

- ▶ The dark boxes (Faces Servlet, configuration file, one managed class per JSP, event logic) represent generated artifacts.
- ▶ The cross-hatched boxes represent Application Developer views that support the development of JSF pages.
- ▶ In addition custom components can be developed.

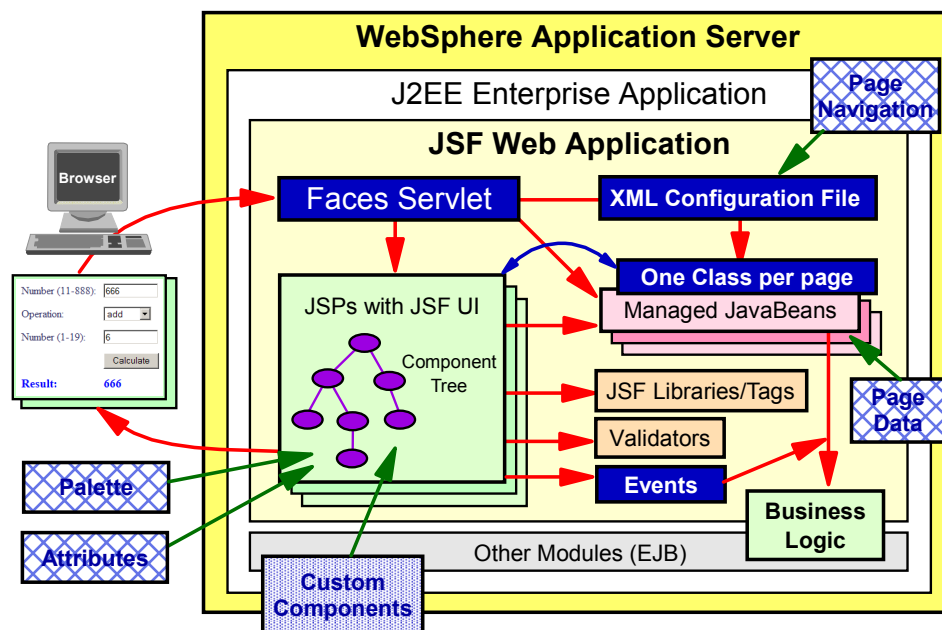


Figure 2-1 Application Developer support for JSF

Creating a JSF project

In WebSphere Studio, a JavaServer Faces project is a variation of the existing project type, *Dynamic Web Project*.

Therefore, the first step in creating a new JSF project is to create a new *Dynamic Web Project* (Figure 2-2):

- ▶ Select *File > New > Project* from the Application Developer menu.
- ▶ Select *Web* in the left panel, then select *Dynamic Web Project* in the right panel.

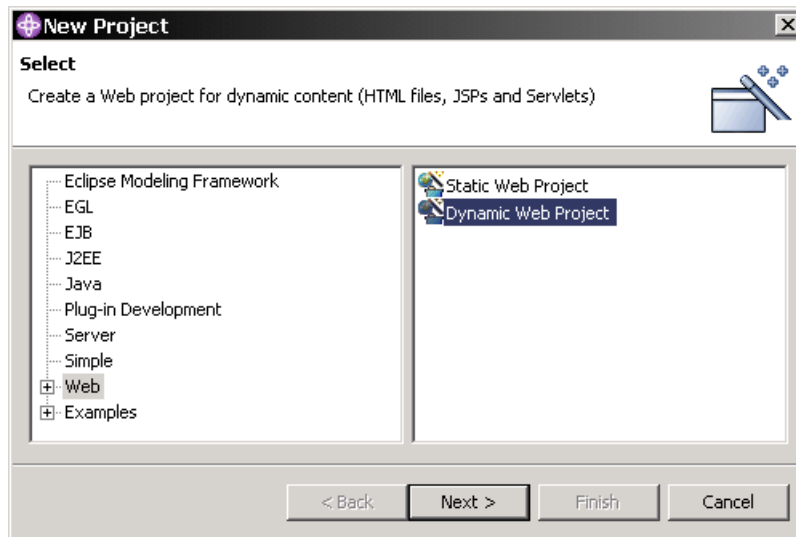


Figure 2-2 Creating a new Dynamic Web Project

- ▶ Click *Next*.
- ▶ Enter a project name in the appropriate field and click *Finish*.

The project is created. Open the Web perspective.

Note: You may select *New -> Dynamic Web Project* directly if you are already in the Web perspective.

The most convenient way to turn a Web project into a Faces project is to simply create a Faces JSP page in the project, as explained below. The creation of the Faces page trigger the necessary changes in the Web project.

Alternately, you can convert this project to a Faces project by modifying the Web project's features (Figure 2-3):

- ▶ Select the root of the project in the Project Navigator view and select *Properties* in the context menu.
- ▶ Select *Web Project Features* in the left pane and select *Add Faces Base Components*.
- ▶ Click *OK*.

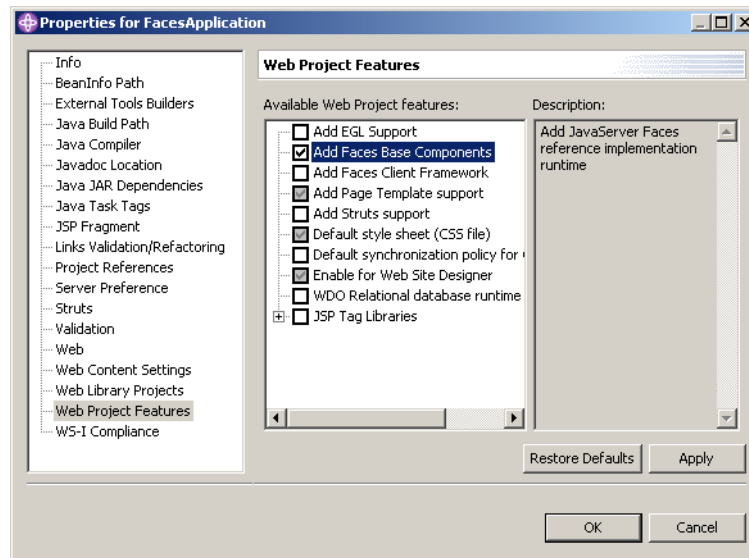


Figure 2-3 Adding Faces support to a project

At this point all of the Faces libraries and support files are added to the project.

Project organization

A JSF Web project is organized in the following manner on the file system:

- ▶ /
This is the root folder the project. There are no JSF specific files here.
- ▶ /JavaSource/
This is the default Java source folder of the project. You can perform Java development in this application, as it is also a Java development project. The classes built from the source folder reside in /WebContent/WEB-INF/classes. Note that this folder is labeled *Java Resources* in the Project Navigator view.
- ▶ /JavaSource/pagecode/

This is the Java package in which all the page code source for the JSF pages is located (see “Page code” on page 31).

► /WebContent/

This folder contains all of the Faces JSP files.

► /WebContent/META-INF/

This is a metadata folder. There are no JSF specific files here.

► /WebContent/WEB-INF/

This is the Web metadata folder. It contains the following relevant files:

- web.xml—This file which contains the servlet definition and mapping for the FacesServlet.
- faces-config.xml—This file contains the Faces project configuration. See “Faces configuration file” on page 13 for more details. This file is automatically maintained by Application Developer.

► /WebContent/WEB-INF/lib

This folder contains Java archives (JAR) used by the project. This folder contains a number of JSF JAR files defining APIs and component libraries.

► /WebContent/theme/

This is the theme folder in which all templates and style sheets are stored. It also contains:

- stylesheet.css—This file contains the definitions for all of the classes used by default by the JSF UI components. You can modify the look and feel of a JSF application using this style sheet.

Creating a JSF page

If a Web project is not yet a Faces project, creating a JSF page in it is the easiest way to convert that project to support Faces.

Unlike creating the Web project, you use a new wizard for creating a Faces page. The structure of the created page differs from a regular JSP created in Application Developer. The tools also generate and register a special managed bean that is associated with the new JSF page. This managed bean is referred as **page code** and is used to access user interface components and to execute event logic.

New Faces JSP File wizard

To create a new Faces page, observe the following steps:

- ▶ Select the WebContent folder (or a subfolder where you want to create the page) in the Project Navigator view and *New -> Faces JSP File* (context). This launches the New Faces JSP File wizard.
- ▶ On the first page of the wizard (Figure 2-4):
 - Select a destination folder. If you selected a folder before launching this wizard, the destination folder is automatically selected. Generally you want to put a JSP into the WebContent folder or a subfolder.
 - Enter the file name for the new page. You are not required to add the .jsp extension, as it is automatically added.
- ▶ You should not have to modify any of the other options available, so click *Finish* to create the page.

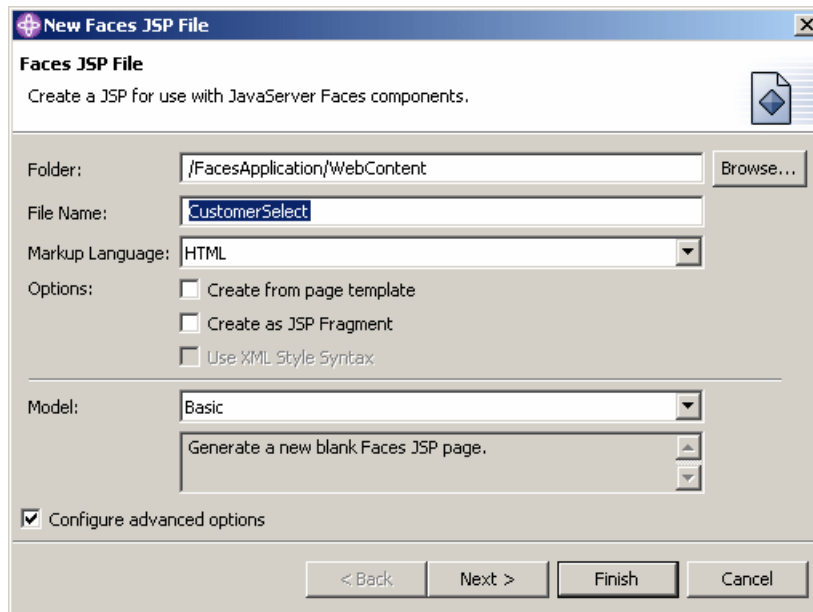


Figure 2-4 Creating a new Faces JSP file

Completing the wizard converts the project to support Faces if that support is not present. The wizard creates the JSP file and opens it in the Page Designer. Also, the wizard generates a page code managed bean.

JSF page structure

The structure of a newly created Faces JSP is as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%-- jsf:codeBehind language="java"
      location="/JavaSource/pagecode/CustomerSelect.java" --%>
<%-- /jsf:codeBehind --%>
<HTML>
<HEAD>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ page
      language="java"
      contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"
%>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM WebSphere Studio">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet"
      type="text/css">
<TITLE>CustomerSelect.jsp</TITLE>
</HEAD>
<f:view>
  <BODY>
    <P>Place content here.</P>
  </BODY>
</f:view>
</HTML>
```

For more detail on the elements making up a Faces JSP file, see “JavaServer Faces page example” on page 21.

Dragging an interactive Faces UI component into the page automatically generates the form and script collector tags, as well as import new tag libraries as necessary.

Page code

Upon creating a Faces JSP file, an empty Java class with the same name (minus extension) is generated in the pagecode package. For instance, the above example generates the class `pagecode.CustomerSelect`. This class is registered as a request scope managed bean in the Faces configuration file, with the name of the Java class prefixed with `pc_`, so the `CustomerSelect` class is registered as the managed bean `pc_CustomerSelect`. This managed bean is then used for the root of most of the value and method binding expressions in the Faces JSP.

Page code managed beans are repositories for convenience methods for accessing relevant data from the page, such as UI components, other managed beans, and action logic. These methods are usually coupled with private references to the data, which are lazy initialized by the accessor methods.

UI components in page code

Drag a Command - Button into a Faces page, change its ID (selectButton), and edit its command event (return "select";), so the JSP source is:

```
<hx:commandExButton type="submit" value="Submit"
    styleClass="commandExButton" id="selectButton"
    action="#{pc_CustomerSelect.doSelectButtonAction}">
</hx:commandExButton>
```

The page code managed bean is automatically updated with an instance of the UI component backing this tag, a convenience method for initializing and accessing that component, and the method implementing the command event:

```
protected HtmlCommandExButton selectButton;
protected HtmlCommandExButton getSelectButton() {
    if (selectButton == null) {
        selectButton = (HtmlCommandExButton)
            findComponentInRoot("selectButton");
    }
    return selectButton;
}
public String doSelectButtonAction() {
    // Type Java code to handle command event here
    // Note, this code must return an object of type String (or null)
    return "select";
}
```

Likewise, all components that are added to the page become accessible through methods in the page code. Value changed event implementations are also stored in the page code for the page in which the input component resides.

Managed beans in page code

Upon adding a managed bean reference to the page using the Page Data view, a field and accessor methods to access the managed bean are generated into the page code. If you add the managed bean `customerData` to the Faces JSP, the following code is generated:

```
protected CustomerData customerData;
/**
 * @author WebSphere Studio
 * @beanName customerData
 * @managed-bean true
 * @beanClass itso.jsf.CustomerData
```

```

    */
    public CustomerData getCustomerData() {
        if (customerData == null) {
            customerData = new CustomerData();
            customerData =
                (CustomerData) getFacesContext()
                    .getApplication().createValueBinding(
                        "#{customerData}").getValue(
                            getFacesContext());
        }
        return customerData;
    }
    public void setCustomerData(CustomerData customerData) {
        this.customerData = customerData;
    }
}

```

Note that if you subsequently bind a UI component to this managed bean, the bean is accessed as a property of the page code bean. Examine this input field, which is bound to a property of the `customerData` managed bean:

```

<h:inputText styleClass="inputText" id="text1"
    value="#{pc_CustomerSelect.customerData.firstName}"/>

```

PageCodeBase class

Upon creating the first Faces JSP file in the Web project, the parent class for all page code, `pagecode.PageCodeBase`, is generated into the project. This is a good location to add common functions that you want to use in all of your page code.

The parent for all page code comes with a number of convenience methods to be used in the action logic. These methods assist in finding components in the UI tree, manual navigation, and logging, among other functions. Please take a moment to examine the methods provided by the `PageCodeBase` class.

Page templates and fragments

You can apply Application Developer page templates to JSF pages. This eases the development of the look and feel of the JSF application. You can even embed Faces forms and controls in the template, which is useful for common navigation functions. Make certain the IDs of any elements in the template do not overlap those used in the Faces pages.

Faces JSP files may be created as page fragments. This is useful for creating reusable chunks of formatted user interface. Make certain when using fragments that you do not duplicate any UI component IDs. Also, take note that including a JSF page fragment in a JSF page does not generate convenience methods in the page code of the containing page for elements in the fragment.

Web perspective

The majority of JSF development is done in the Web perspective (Figure 2-5). Though the Web perspective is not unique to or specifically for JSF development, a number of the views and editors have new functions and different behavior when working with Faces applications.

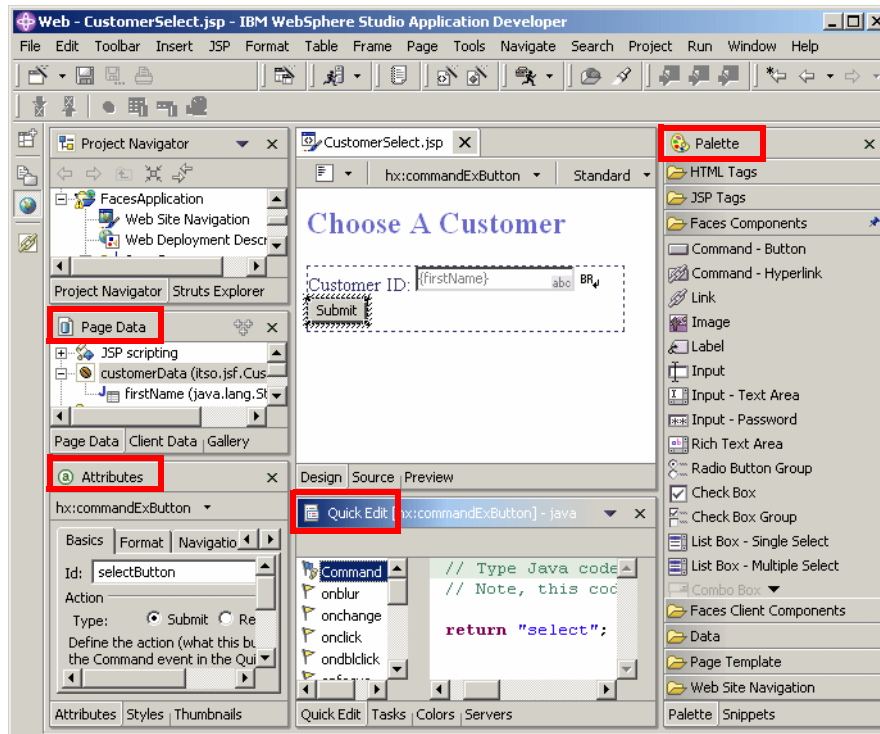


Figure 2-5 Web perspective with Palette, Page Data, Attributes, and Quick Edit

Page Designer

The Page Designer is the default editor for Faces JSP files (Figure 2-6). This editor is used for editing all JSP pages in Application Developer, but it supports some new behavior for JSF pages.

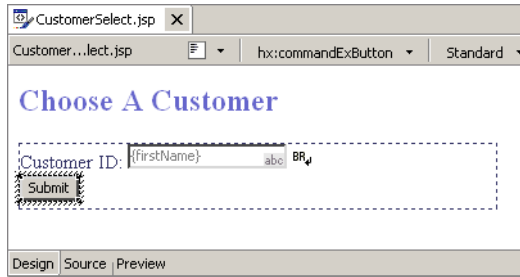


Figure 2-6 Page Designer editor

Page Designer modes

The page designer supports three views of the JSP file: Design, Source, and Preview, each accessible through a tab at the bottom of the editor:

- ▶ When the **Design** page is active, the layout and content of the JSP can be modified visually, providing a WYSIWYG interface for page design. This is a useful interface for page authors less familiar with the intricacies of HTML and JSP development.

JSP UI components are visualized in the design view by rendering them as they would appear in the browser, but decorating them with other useful data. Input and output components, in particular, are decorated with the name of the bound value in brackets, and an indicator on the right side of the field for the format type (Figure 2-7).



Figure 2-7 JSP component visualization in Page Designer

- ▶ When the **Source** page is active, the source of the JSP file is displayed with formatting and syntax highlighting. This is a good interface for those page authors with significant familiarities with HTML and JSP development. For complex pages, you might find it useful to select an object in the Design page, then switch to the Source page, and the element selection is preserved. Also, you can take advantage of context assist in source mode by pressing **Ctrl-Space** while editing the source.
- ▶ The **Preview** page is used to show a representation of the Web page as it appears at runtime. This representation is as accurate as possible, within the constraint of not having access to dynamic data supplied at runtime. Use this view to verify the page design and styles, as the artificial selection borders from the Design page are not used here.

Context menu

Some new items specific to JSF are available from the context menu in the Design and Source pages:

- ▶ *View Page Code*—This menu item can be used to open the source of this JSF page's page code managed bean for editing.
- ▶ *Edit Faces Value Changed Event*—This menu item is available when a JSF input component is selected. It can be used to open the implementation of an associated value change listener for editing within the page code.
- ▶ *Edit Faces Command Event*—This menu item is available when a JSF command component is selected. It can be used to open the implementation of an associated action listener for editing within the page code.

Palette

The Palette view is used to display the components that the developer can work within Page Designer (Figure 2-8).

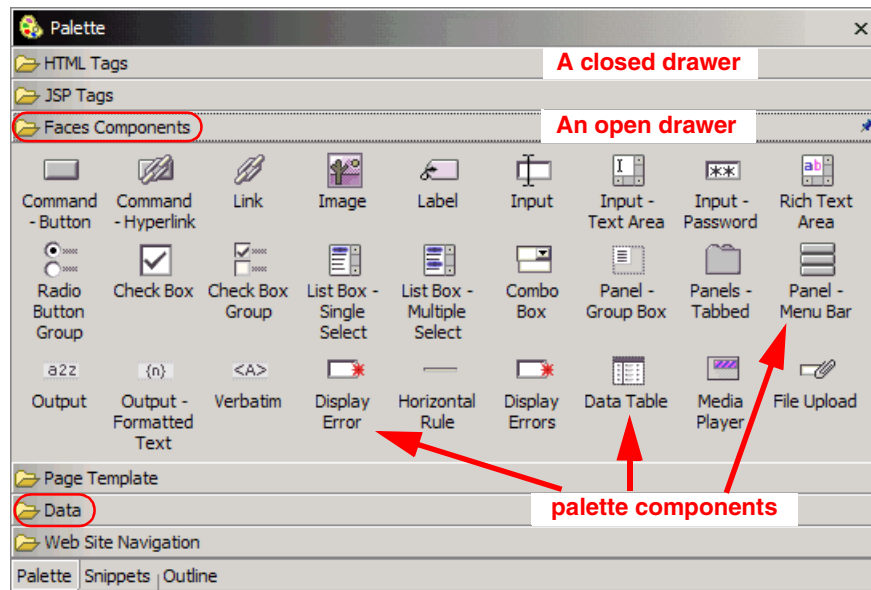


Figure 2-8 Palette view (customized to columns layout)

The palette is divided by groups of elements belonging to the same framework, called drawers. The palette is dynamic, allowing you to activate and expand a preferred drawer of components for the development task. JSF contributes some new drawers, *Faces Components* and *Data*, which should be visible when you

are working with a Faces JSP file in the Page Designer. The existing set of drawers and their components may still be used in Faces pages.

Components can be added to JSPs directly from the palette by double-clicking or drag and drop. Page code generation is triggered by adding components using the palette. Finally, the palette may also be customized directly by the developer.

Attention: Custom JSF tag libraries can be included within pages by just adding the syntax for the tag library and including the tags within the page. See “Extending the Studio palette” on page 512 on how to extend the palette to add you own custom components to WebSphere Studio.

Faces components drawer

The *Faces Components* drawer contains components which correspond to Faces UI component tags. Using these components in conjunction with the Page Designer in the design page allows you to visually design the user interface of your Faces application. Chapter 4, “JSF components” on page 85 covers the specifics of each available component in depth.

Data drawer

The *Data* drawer contains components which correspond to managed beans. Using these components allows you to generate new managed beans for Web services or database queries, or add references to existing Java beans. See the section “Data objects” on page 45 for more detail on these components.

Note: The *Faces Client Components* drawer (not shown in Figure 2-8) is covered in Chapter 13, “Faces Client framework” on page 439.

Drag and drop

You can easily drag and drop components into a JSP page:

- ▶ Select the component you want to use by selecting it with the mouse.
- ▶ Go to the JSF page (the mouse pointer changes) and click in the place where you want the new component to be inserted.

Alternately, select and hold a component in the palette, then drag it over the Page Designer and release the mouse button when in the proper position. You can also double-click a component in the palette to insert the item at the current selection in the Page Designer. Finally, you can select a component and *Insert* from the context menu to add the component at the current selection.

Upon adding a component from the palette, new code is added to the page code bean for the affected page.

Tip: A drag operation from the palette to the Page Designer automatically generates a unique ID for a given component. If you manually edit the ID you have to make sure that the ID value is unique.

Customization

You may customize the palette contents and behavior by selecting *Customize* or *Settings* from the palette's context menu:

- ▶ Use the Customize Palette dialog to arrange the drawers and components in the palette, hide elements, and modify component properties (Figure 2-9).

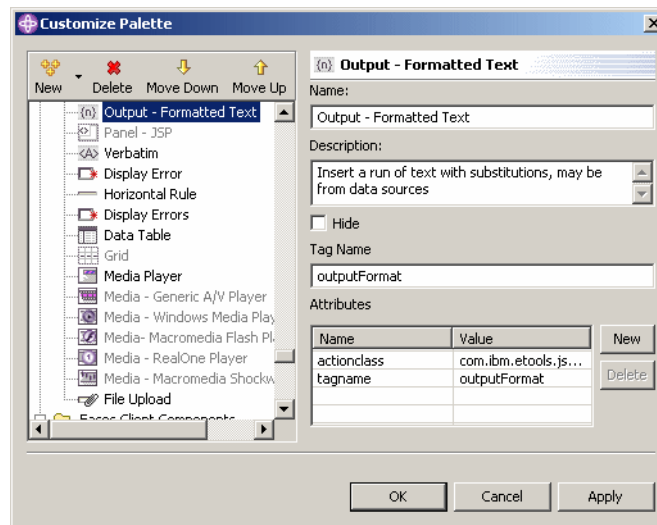


Figure 2-9 Customize Palette dialog

- ▶ Use the Palette Settings dialog to modify the palette's layout, style, and behavior (Figure 2-10).

You can also change the layout by selecting *Layout -> Columns* (or *Icons Only* or *Details* or *List*) from the context menu.

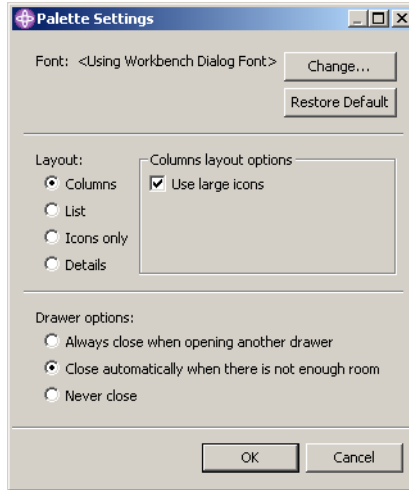


Figure 2-10 Palette Settings dialog

Attributes view

The Attributes view is used to modify attributes for a selected component in the Page Designer (Figure 2-11).

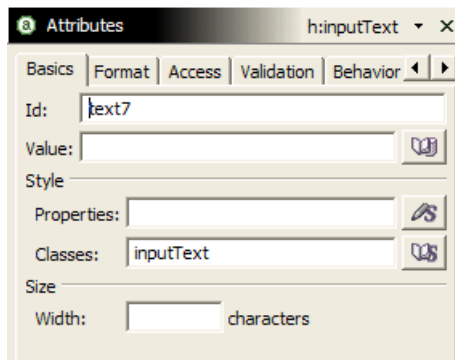


Figure 2-11 Example Attributes view for an Input component

You can select the tabs and move through all the attributes in turn for a given component. As you change these values, they are automatically reflected in the JSP source, and hence in the Page Designer. If you modify any attributes in the Page Designer, they are reflected the Attributes view.

Certain attribute fields which are common to multiple components have associated dialogs which can be used to pick an appropriate values. Some of these are not specific to JSF, such as those related to CSS references. JSF contributes new common dialogs for value binding and action binding.

Value binding

The value field is common to all JSF UI components that can be bound to model data. This field takes a value in the JSF value binding syntax, explained in detail in Chapter 4, “JSF components” on page 85.

Clicking the  icon next to the Value field launches the Select Page Data Object dialog (Figure 2-12), which can be used to visually select a model object or property from those available in the page.

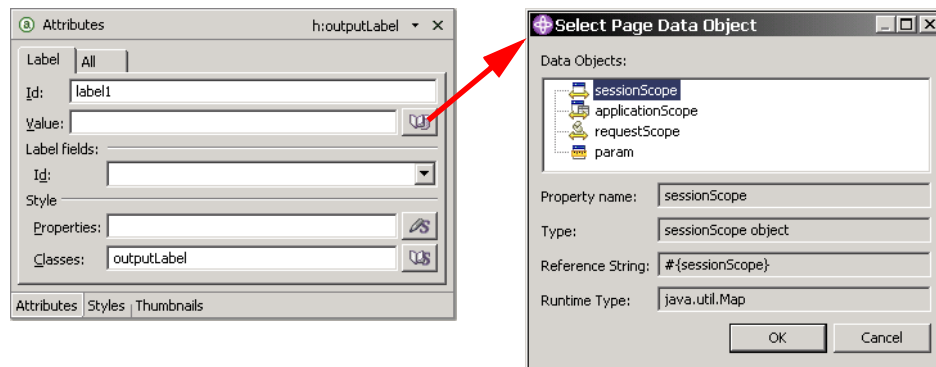


Figure 2-12 Using the value binding common dialog

Action binding

The action field is available on the All tab for all command components. This field is used to specify a method binding for the default action listener for the selected component. This field takes a value specified in JSF method binding syntax, explained in detail in Chapter 4, “JSF components” on page 85.

Click the  icon next to the Action property's value field to launch the Select Page Data Object dialog (Figure 2-13). In this context, the dialog is used to select an action from the list of actions available in the page code.

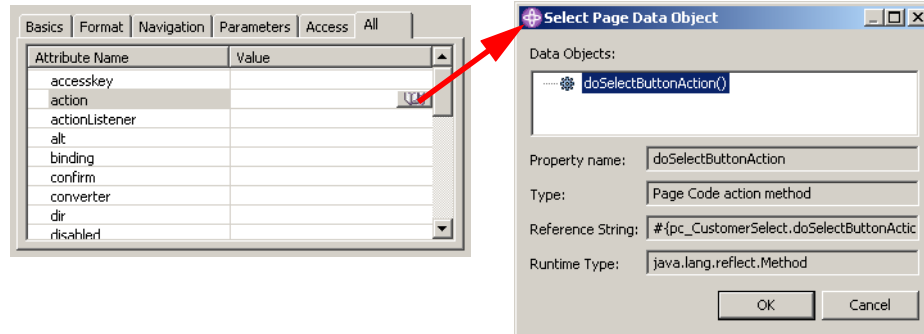


Figure 2-13 Using the method binding common dialog

Navigation tab

Command components all share a common tab in the Attributes view for working with JSF navigation rules. The Navigation tab contains a table listing all the navigation rules originating from the currently open JSF page in the Page Designer. Global rules are also displayed in the table. Rules can be added, edited, or removed from the table.

Clicking *Add* opens a dialog for specifying the properties of the new rule (Figure 2-14). All the fields may have their values selected from drop-down lists or entered by hand. Navigation rules are explained in detail in “Navigation” on page 14.

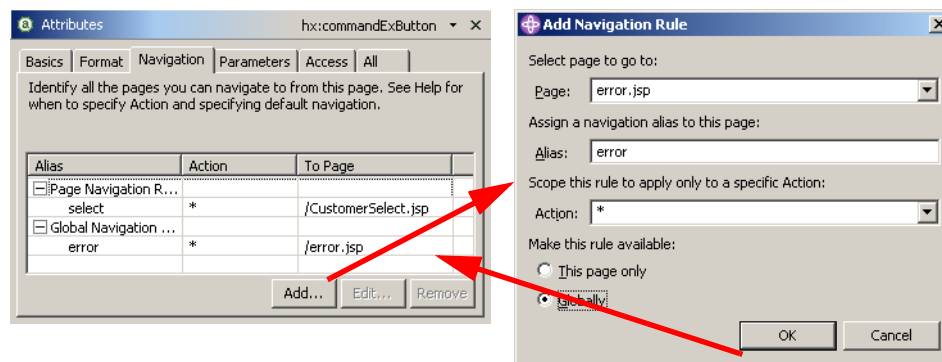


Figure 2-14 Constructing a new navigation rule using the Attributes view

Format tab

UI components that support data conversion have a Format tab available in the Attributes view (Figure 2-15).

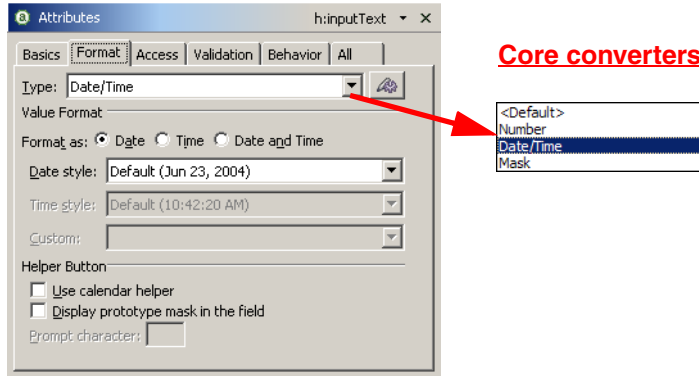


Figure 2-15 Attributes view Format tab

The Format tab allows you to choose from the set of JSF core converters and configure their attributes. Upon choosing a converter, a converter tag is added as a child of the selected component.

The *Mask* converter is an IBM extension to the core JSF converter set. Use this converter to nicely format social security numbers, credit card numbers, phone numbers, or the like.

You can also specify a Helper on the Format tab, which assists the user in entering the data in the desired format. For example, the Mask converter helper populates the field initially with the data format, and the Date converter helper provides a pop-up calendar control for selecting a date.

Validation tab

UI components that support data validation have a Validation tab available in the the Attributes view (Figure 2-16).

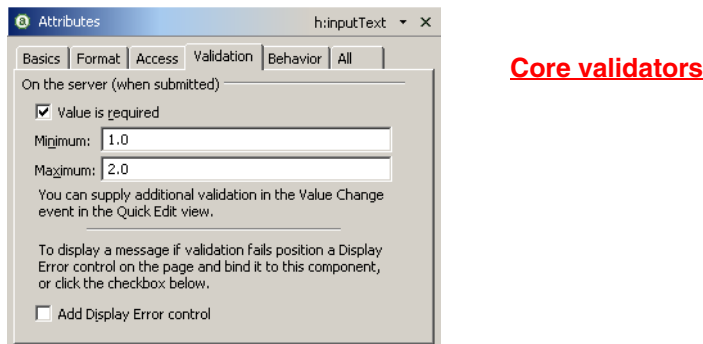


Figure 2-16 Attributes view Validation tab

The Validation tab allows you to choose one or more validators and configure their attributes. Upon selecting validators, their tags are added as children of the selected component. Note that the *Value is required* validator is actually registered through an attribute of the UI component tag, and not a child tag.

Select *Add display error control* to automatically generate an Error Message component that is bound to the selected component.

For more information about validation see “Data validation” on page 236.

All tab

If you want to view all attributes available for the selected component, select the *All* tab, and a table of all supported attributes is displayed (Figure 2-17). Click on an attribute’s value column field to see if there is a specialized dialog for supplying the attribute value.

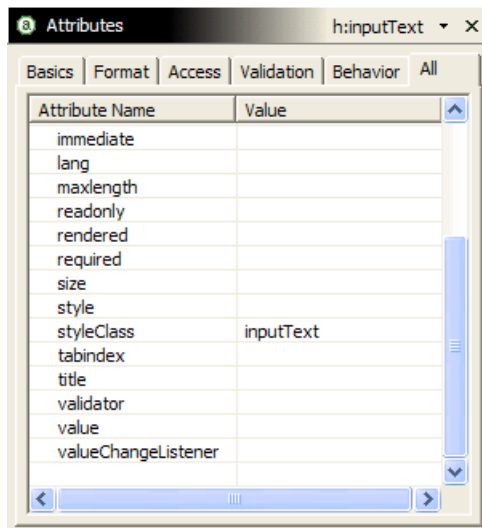


Figure 2-17 Attributes view All tab

Refer to Chapter 4, “JSF components” on page 85 for detailed information about the attributes of the UI components provided in Application Developer.

Page Data view

The Page Data view allows you to define and manipulate objects (JavaBeans and scope variables) that can be used in the currently active JSF page in the Page Designer. You can usually find the Page Data view in the left side of the workspace, between the Project Navigator view and the Attributes view.

The only objects that are predefined are the scope objects. You can expand JSP scripting to see them (Figure 2-18). The JSP scripting scopes are covered in “Managed beans” on page 17.

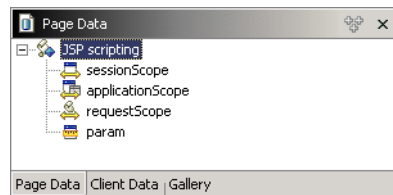


Figure 2-18 Page Data view

Context menu

The context menu allows you to create, delete, and configure all the objects you can define in the Page Data view. It also allow you to bind the Page Data objects to the JSF components in the Page Designer.

Adding a new object

To add a new object in the page data, select *New* in the context menu or click the  icon (Figure 2-19). The possible objects are:

- ▶ Web Service
- ▶ Relational Record List
- ▶ Relational Record
- ▶ JavaBean
- ▶ Scope variables and parameters. If you select the application, session or request scope, the context menu allows you to define variables for these scopes. If the *param* list is selected, you can define new parameters.

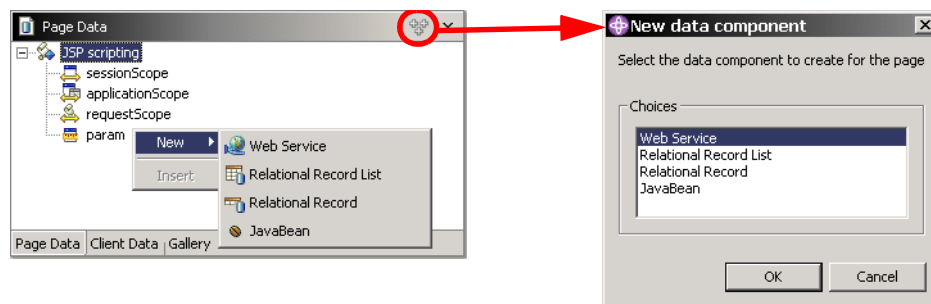


Figure 2-19 Adding data components to the Page Data view

Deleting an object

To delete an object from the Page Data view select the object and *Delete* in the context menu.

Configuring an object

To configure an object select *Configure* from the context menu. Depending on the type of object, you obtain a different editor for its configuration.

Binding an object to a component

When you are designing a new JSF page you can bind the components to the object you have in the Page Data view. You can do this binding from the Page Data view in two ways:

- ▶ Select the object you want to bind and drag in into the Page Designer. Drop it on the component you want to use for the binding. Information appears over the cursor when you hover over a valid component for binding.
- ▶ In the Page Designer select the visual component you want to bind to the object. Go to the Page Data view and right-click in the object you want to bind. A new option in the context menu appears (*Bind to <component_id>*). Select this option and the binding is made for you (Figure 2-20).

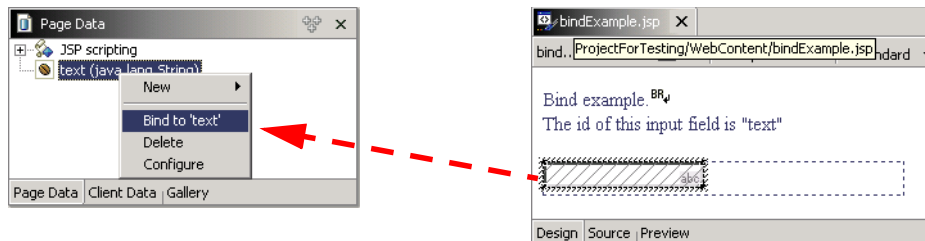


Figure 2-20 Example of component binding

Note: When binding a collection or a list of objects, a pop-up window appears to select the base type for the collection.

Data objects

You can define four different kind of data objects. You can use the Page Data view or the Data drawer of the palette (refer to “Palette” on page 36 for more details) to add them to a JSF page.

Web service

This data object allows you use a Web service client inside your JSF pages. The Web service object is covered in Chapter 7, “JSF and Web services” on page 277.

Relational Record and Relational Record List

These data objects allow you to include a record or a set of records from a relational database in your pages. These objects are covered in Chapter 9, “SDO concepts” on page 299.

JavaBean

This component allows to define managed beans for JSF pages. When you define or modify or delete a JavaBean, the Faces configuration file (WebContent\WEB-INF\faces-config.xml) is updated with a definition of the managed bean.

WebSphere provides a wizard to define and configure these objects. When you create a JavaBean object using the Page Data view, the Add JavaBean wizard opens (Figure 2-21).

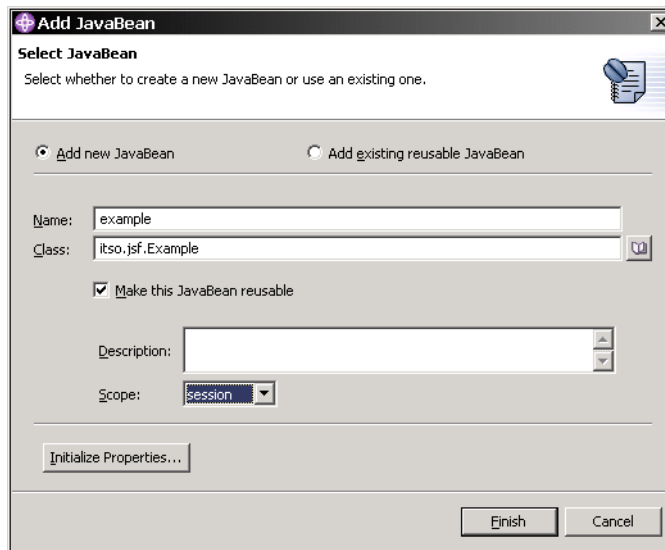


Figure 2-21 Add Bean wizard

If you select *Add a new JavaBean* (Figure 2-21), you have to provide a unique identifier for the object and the Java class implementing the object. Additionally you can define the JavaBean as reusable and add it to the configuration file. That means that you are able to select that bean later in other JSF pages.

If you select *Add an existing reusable JavaBean*, the wizard shows the list of reusable beans that are defined in the `faces-config.xml` file (Figure 2-22).

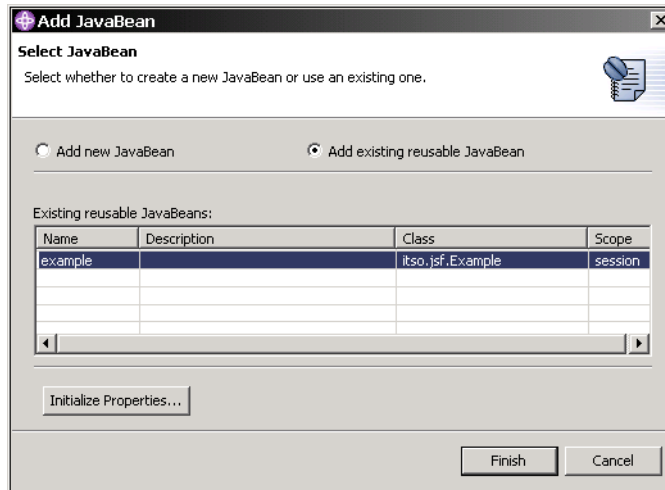


Figure 2-22 Add reusable JavaBean wizard

In any of the above wizards you can click the button *Initialize Properties* to select the initial values for the attributes of the objects. In the Initialize Properties dialog (Figure 2-23) you can add, delete and reorder the initialization of attributes.

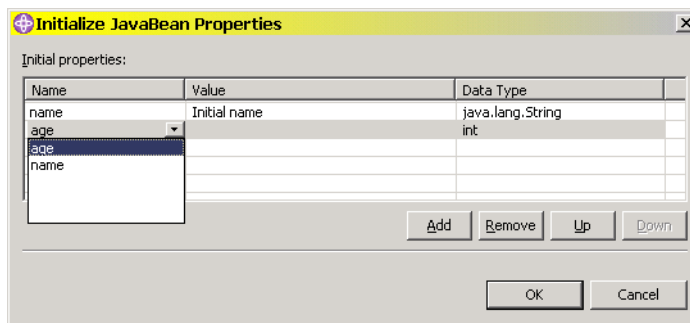


Figure 2-23 Initializing JavaBean properties

When you define the initialization for JavaBean attributes, the necessary updates are done in the `faces-config.xml` file.

JSP scripting

The JSP scripting part of the Page Data view controls the variables that are defined in the different scopes the page can access. Using the JSP scripting you can define the parameters you want to pass to a JSF page.

You can add, delete, configure, and bind the variables and the parameters. You can drag and drop them into the JSP page from here. You can also configure a specific scope or the complete list of parameters of the page.

To add a new variable to a scope, or to add a new parameter to the page, expand the JSP scripting, select the scope you want to use and select *Add scope variable* or *Add parameter* from the context menu. A pop-up window appears and asks you about the name of the variable or parameter and the type of the object. You can select any type from the workspace, but you are not allowed to use primitive types. You must use the classes representing the primitive types, for example, you use the type `java.lang.Integer` instead of the primitive type `int` (Figure 2-24).

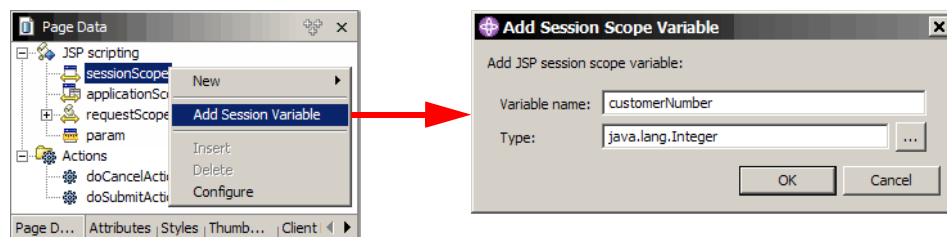


Figure 2-24 Adding a scope variable

To change a variable, parameter or scope, select it and *Configure* from the context menu. The appropriate editor opens. If you are configuring a scope or the list of parameters, a complete list of variables or parameters appears. From this editor you can add, delete, and configure the variables of that scope or the parameters for the page (Figure 2-25).

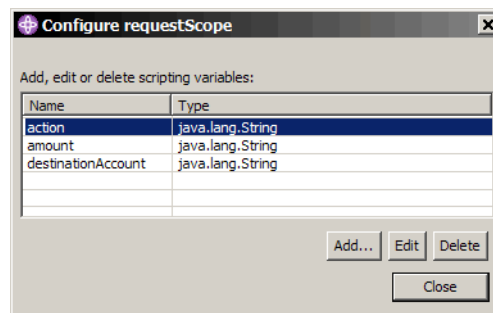


Figure 2-25 Configuring scope variables

To bind a variable or parameter to a component in your page, you can drag and drop it inside the selected component or use the *Bind to <component_id>* option in the context menu when the *<component_id>* component is selected in the Page Designer.

To include a variable or parameter inside the JSP page, you can drag and drop it into the page. You can also select the *Insert new controls for <variable_name>* option in the context menu. This option opens a configuration dialog for the controls that will be added to the page.

Action list

When you include server side code in the components of the page, the associated page code Java class is updated with new Java methods to perform the actions. All the action are displayed in the Page Data view under a new tree with the root labeled *Actions* (Figure 2-24). Any method added manually to the page code which returns a `java.lang.String` and takes no parameters appears in the action tree. Expand this tree to see all of the available actions in this page.

Moreover, you can drag and drop these actions onto command components in the page.

Drag and drop

The Page Data view allows you to use drag and drop to include elements from this view into your page. The drag and drop function has two main functions:

- ▶ If you drag and drop any element from the Page Data view to the JSF page, Studio attempts to create a new component in the page. The component is created according to the definition of the element of the Page Data view that is been dropped in the page.
- ▶ If you drag and drop any element from the Page Data view onto an existing component in the page, Studio tries to bind that component to the element you are dragging and dropping.

Tree view

The Page Data view shows its elements using a tree view. When you define a component in this view, a tree is shown for the component. You can expand this tree and inspect the contents of the component.

Client Data view

The Client Data view allows you to define, manage and destroy objects that are use by the pages in the client side. This view is in the same place as the Page Data view (Figure 2-26).

The Client Data view is part of the Faces Client component framework. This framework covered in Chapter 13, “Faces Client framework” on page 439.

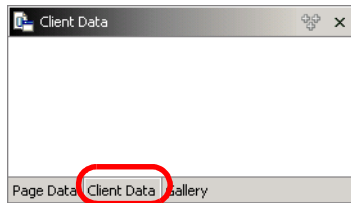


Figure 2-26 Client Data view

Quick Edit view

The Quick Edit view allows you to edit the code for the action logic, that is, the code that is executed when an event is fired. This view is at the bottom of the Studio workspace (Figure 2-27).

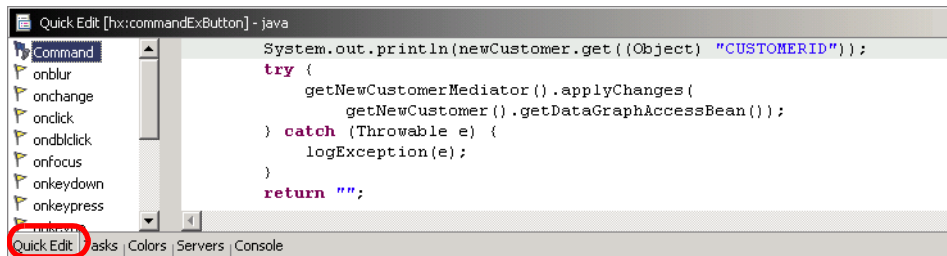


Figure 2-27 Quick Edit view

The Quick Edit view always refers to the component that is selected in the JSF page. Using this editor you can edit client-side events and server-side events. Note the rightmost text in the title bar for the view. This text, either *java* or *javascript* indicates whether the selected event handler is implemented on the server or the client, respectively.

Java events

Java events are server-side events. Components such as the Command Button have this kind of server-side event. The event handling code is part of the page code of the JSF page where the component is contained, and is executed during the corresponding Faces life cycle phase.

You can edit the event handling code by selecting the component and *Edit Faces Event* (context).

JavaScript events

If the component that is selected in the page allows client side events, these events are shown in the Quick Edit view and you can include client action logic. The programming language used for the client side events is JavaScript.

Snippets

A code snippet is an object which acts like a little library. WebSphere Studio Application Developer provides a set of snippets for common functionalities.

The snippets are provided for both the Java events and the JavaScript events. To select and insert a snippet, select *Insert Snippet* in context of the Quick Edit view for the selected event. A list of available snippets is shown. Select the snippet you want to insert and the code is inserted (Figure 2-28).

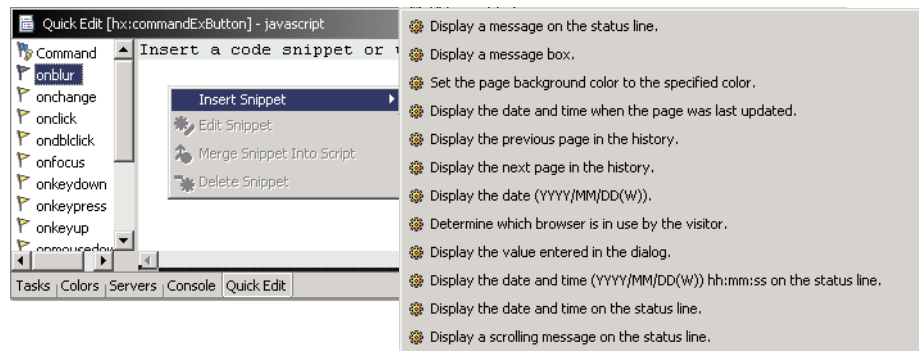


Figure 2-28 Inserting a snippet in a client side event

Page initialization

You can attach logic to the page load and post events. The page events are attached to the <hx:scriptCollector> tag of a JSF page. The events are:

- ▶ Page Load Begin—Before rendering (changes to managed beans take effect)
- ▶ Page Load End—After rendering (changes to managed beans have no effect)
- ▶ Page Post—When the page is submitted

The generated tag is:

```
<hx:scriptCollector id="scriptCollector1"
  preRender="#{pc_bean.onPageLoadBegin}"
  postRender="#{pc_bean.onPageLoadEnd}" decode="#{pc_bean.onPagePost}">
```

The onPageLoadBegin, onPageLoadEnd, and OnPagePost methods are generated into the page code class. Refer to “Implementing page initialization” on page 83 for an example.

Testing, debugging, and deploying the JSF project

WebSphere Studio Application Developer allows you to test and debug JSF Web application projects in the same way as other Web projects. The only restriction to use JSF technology is that the Web project has to be targeted for WebSphere Application Server Version 5.1.

Server targeting

To target the project for a server, you first have to enable the server targeting function for the whole workspace. To enable server targeting, select *Window -> Preferences*. Select *J2EE* and *Enable server targeting support* (Figure 2-29).

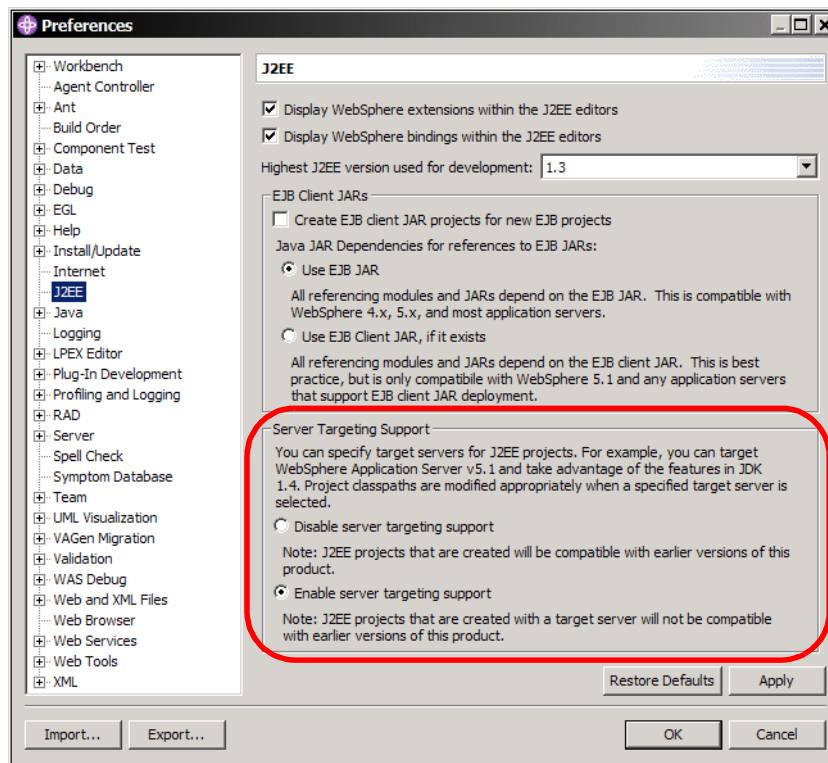


Figure 2-29 Enable targeting support

After enabling server targeting you can set the target server for the project. Select a Web project and *Target Server -> Modify*. In the pop-up window, select the target server.

Important: You must select *WebSphere Application Server v5.1* for Faces projects.

Test environment

To test JSF Web applications, you can use the WebSphere test environment server 5.1. To define such a server, open the Server perspective.

In the Server Configuration view (at the left bottom side in the workbench) select *New -> Server and Server configuration* from the context menu. In the pop-up window:

- ▶ Enter a name for the server.
- ▶ Expand the WebSphere application server 5.1 tree and select *Test Environment* (Figure 2-30).
- ▶ Click *Finish*.

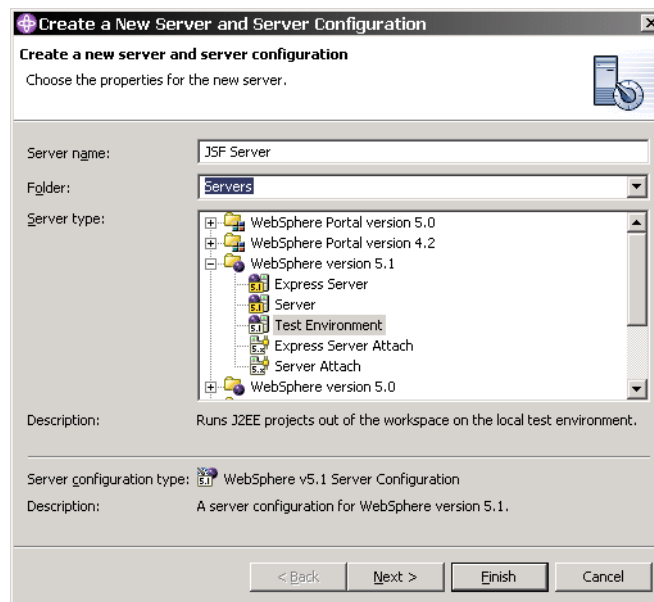


Figure 2-30 Defining the test environment server

Now you can configure this server for your application requirements. You find an example of how to configure a server for the examples of this redbook in “Setting up the WebSphere test environment” on page 535.

Considerations and guidelines

In this section we discuss about things that you should keep in mind when you develop JSF Web applications with WebSphere Studio Application Developer.

Creating a JSF page and managed beans

If you are going to create a JSF page, make sure you do not use the same name as any of the Java class you plan to use as a managed bean. The Page Data view may get confused as to which bean should be displayed, the page code bean or the managed bean. To avoid any conflict make sure the name is different from the JSF page name.

Renaming a JSF page

If you decide to rename a JSF page, the page code Java class is automatically renamed as well.

Navigation rules

If you rename a JSF page, you must take note of the navigation rules. When the renaming is done, the navigation rules are not changed and you must change them manually.

Example

We have two JSF pages `source.jsp` and `target.jsp` with the following navigation rule defined in the file `faces-config.xml`:

```
<navigation-rule>
  <from-view-id>/source.jsp</from-view-id>
  <navigation-case>
    <from-outcome>navigate</from-outcome>
    <to-view-id>/target.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Now we rename the target file to `newTarget.jsp` and we want to keep the navigation. We must redefine the navigation as follows:

```
<navigation-rule>
  <from-view-id>/source.jsp</from-view-id>
  <navigation-case>
    <from-outcome>navigate</from-outcome>
    <to-view-id>/newTarget.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```


You have to be careful with this because WebSphere Studio Application Developer does not fix the navigation rules automatically, nor does it warn you about broken links in the navigation rules.

Scope variables

When you rename a Web project with JSF pages you may lose the definitions of the scope variables. These definitions are stored in the `.jspPersistence` control file in the project folder (use the Navigator view to see these files).

A request scope variable may be defined as shown here:

```
<request id="/ItsoJSFBank/WebContent/error.jsp">
    <variable name="errorMessage" type="java.lang.String"/>
</request>
```

As you can see the JSP page is part of the definition. Renaming a page does not update these definitions and you have to manually correct them. Otherwise you do not find the scope variables in the Page Data view.

Broken link warning

Sometimes you get a broken link warning in the Tasks list about the reference of the page code in the `faces-config.xml` file. Select the project and *Project -> Rebuild Project* to resolve such an error.

Deleting a JSF page

If you want to delete any JSP files from a Web application, you have to do a couple of things to keep your application clean. Before deleting the JSP, you should note the name of the page code Java bean and the name of the JSP page.

To maintain the application do the following:

- ▶ Delete the page code Java class. You usually find this Java file under the Java Resources folder, inside the pagecode Java package with the same name as the JSP page you deleted.
- ▶ Delete all the navigation cases where the deleted JSP file was involved from the `faces-config.xml` file. You do not want to have unused navigations in the configuration file.

Moving a JSF page to another folder

If you want to move a JSP file from one folder to another in a Web application, the page code Java class is also moved accordingly.

Example

We have the JSP page `source.jsp` in the `WebContent` folder of our application and its page code Java class is `pagecode.Source.java` under the `Java Resources` folder. If you move the JSP page to the folder `newFolder/source.jsp` under the `WebContent` folder, WebSphere Studio creates a new package for the page code Java class and moves this class to the new package. The final JavaBean will be `pagecode.newFolder.Source.java`.

When you move a JSP page to another folder you have to fix the all the navigation rules in which that JSP page is involved.

Moving the page code to another package

If your company has the requirement that all Java package names must follow a strict naming convention, you can move the generated page code classes to another folder. For example:

- ▶ Create a tailored package, such as, `com.ibm.itso.jsf.pagecode`.
- ▶ Move the generated classed and the `PageCodeBase` class from `pagecode` to `com.ibm.itso.jsf.pagecode`. The references in the matching JSPs and in the configuration file are automatically updated.

Recreating a JSF page

To recreate a JSF page the easiest way is follow these three steps:

- ▶ Save the JSP code.
- ▶ Delete either the JSF page and the Java class file.
- ▶ Recreate the JSF page and restore the information you want to keep.

Refactoring managed beans

If you have to refactor a managed bean (rename the Java class or the Java package) you have to fix the `faces-config.xml` file manually. WebSphere Studio warns about the broken link in the Task view.

Example

We have defined the bean `calculator.CalculatorBean` to be reusable in the session scope. In the configuration file it appears as the following:

```
<managed-bean>
  <managed-bean-name>calc</managed-bean-name>
  <managed-bean-class>calculator.CalculatorBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

If we rename the bean Java class to `itso.jsf.CalculatorBean`, the configuration file remains the same, but we have to fix the definition to update the reference. The correct definition must be the following:

```
<managed-bean>
  <managed-bean-name>calc</managed-bean-name>
  <managed-bean-class>itso.jsf.CalculatorBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Deleting components

Sometimes you want to remove UI components from a JSP page. Be careful with the action components (Command Button, Command Link) because their associate actions are not deleted from the page code Java class.

These actions should be deleted manually to keep your code clean. You can see all the action function that WebSphere Studio has done for you in the Page Data view. Our advice is to delete those that are not used:

- Figure 2-31 shows an example of this. Only the `doButton1Action()` method is valid (belonging to the *Validate* button). Remove the other two methods.

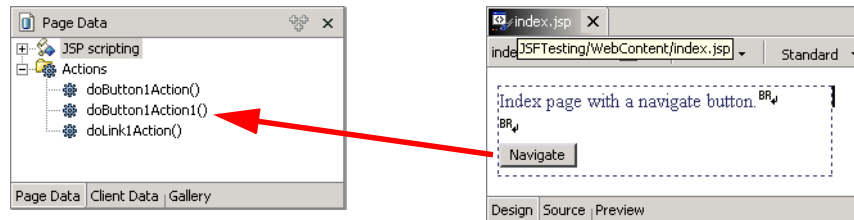


Figure 2-31 Example of actions that should be removed

Deleting managed beans

If you have to delete any of the managed beans you are using in JSF pages, the correct order is as follows:

- Delete the bean definition in the Page Data view of the JSP from which you want to delete the managed bean. If the bean is reusable (application, session or request scope) you are prompted if you want to keep the bean definition for others pages. Answer *No* to this question and the definition of the bean is deleted from the configuration file.
- Delete completely all the bindings the managed bean has with any of the UI JSF components used in your pages. If not, you get a server error when the page is rendered to the browser. The bindings are not deleted automatically.

- Delete the Java classes for the managed beans to keep you application clean.

Renaming a project

When you rename a Web project with JSF pages you may loose the definitions of the scope variables. These definitions are stored in the `.jspPersistence` control file in the project folder (use the Navigator view to see these files).

A request scope variable may be defined as shown here:

```
<request id="/ItsoJSFBank/WebContent/error.jsp">
    <variable name="errorMessage" type="java.lang.String"/>
</request>
```

As you can see the project name is part of the definition. Renaming a project does not update these definitions and you have to manually correct them. In addition open the Web deployment descriptor and on the Overview page change the Display name property to the new project name.

Creating a welcome page

The welcome page is the page that is automatically loaded in the browser when a Web application is invoked without an HTML or JSP name. As in any other Web applications, the welcome page in a JSF Web application is a list of files defined in the Web deployment descriptor file (`WebContent\WEB-INF\web.xml`).

To define a welcome page in a JSF Web application follow these steps:

- Create the JSF page you want to be the welcome file. For instance, suppose the name of this page is `welcomeJSF.jsp`.
- Create a regular JSP page, `index.jsp`, in the Web application. This page is a blank page. Insert a new Forward component from the JSP Tags palette and type `faces/welcomeJSF.jsp` in the page property. Click *OK*.
- The `index.jsp` code should be something like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1" %>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM WebSphere Studio">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet"
type="text/css">
<TITLE>index.jsp</TITLE>
</HEAD>
```

```
<BODY>
<jsp:forward page="faces/welcomeJSF.jsp"></jsp:forward>
</BODY>
</HTML>
```

Note: You can pass parameters to the welcome page using the `<jsp:forward>` tag.

- Now open the Web deployment descriptor. Select the Pages page and remove all the welcome pages, except `index.jsp` (Figure 2-32).

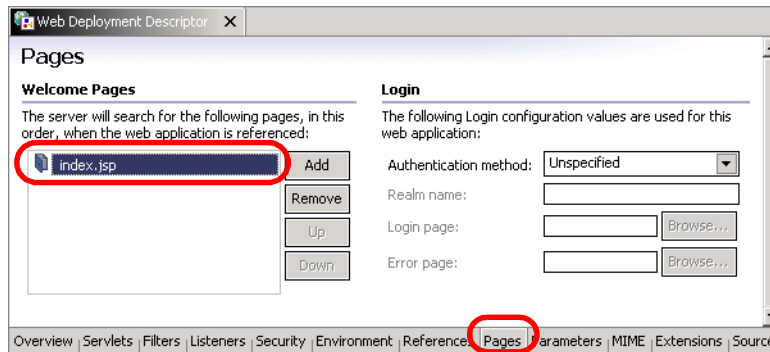


Figure 2-32 Defining the welcome page

- Save all the files. You can test the welcome page by running the Web application in the server.

Summary

In this chapter we outline the support that Application Developer provides for JSF. This support includes the enhancements to the Page Designer, the views (Palette, Page Data, Attributes, Quick Edit), and the special page code Java class.



JSF calculator example

In this chapter we describe how to use WebSphere Application Developer Studio to create a simple JSF example.

This chapter covers these topics:

- ▶ Create JSF pages using JSF components with Page Designer
- ▶ Binding UI components to a JavaBean
- ▶ Implementing basic component validation
- ▶ Implementing basic navigation
- ▶ Creating a custom validator
- ▶ Implementing a value change event
- ▶ Making an application work with several languages

Simple example that shows the basic facilities

In this part we create a JSF Web project and experiment with some of the functions and components of JSF. We use a simple calculator JavaBean with a JSF frontend.

Calculator

We provide a JavaBean named `CalculatorBean` that provides the basic mathematical operations on two integer (long) numbers.

This bean has five properties: two numbers, an operation, a result and an error message. The bean also has getter and setter methods for all its properties. The `calculate` method performs the operations **Add**, **Subtract**, **Multiply** and **Divide** and calculates the result. If a division does not result in an integer, an exception is thrown. The result is stored in the first number for subsequent operations. Finally, a `toString` method displays the numbers and the operation.

Together, we create a JSF frontend for the calculator business logic.

Preparation

Before implementing the calculator example we have to setup the environment as described in “Setting up the environment for this redbook” on page 533:

- ▶ Create and load the EJBANK database (for later chapters)
- ▶ Setup server targeting for the workspace
- ▶ Prepare the JSFServer for testing

Development steps

In this section we give detailed instructions for creating the complete calculator application.

Creating a new Web project

JSF applications are dynamic Web projects. In the Web perspective, create a Dynamic Web Project. Enter **ItsoJSFCalc** as name. Select *Configure Advanced options* and click *Next*.

Enter **ItsoJSF03Calc** as the EAR project. Leave the context root and J2EE level 1.3. Select *WebSphere Application Server v5.1* as the *Target Server*. You can go through the other pages (and accept the defaults) or just click *Finish*. Click *OK* to repair the server configuration.

In the Servers view, select the JSFServer and *Add and remove projects* (context). Select the ItsoJSF03Calc project and click *Add* to associate the enterprise application with the server. Click *Finish*.

Import the calculator bean

In the ItsoJSFCalc project create a Java package named `itso.jsf.calculator`. Select the package and Import. Select File System, locate and select the file `\\labscode\\dev-calculator\\CalculatorBean.java`.

Now we can use the functionality of the calculator bean in our Web application.

Creating a JSF page for the calculator

First of all we create a blank JSF page for the calculator by selecting *New -> Faces JSP File* in the Web perspective under the WebContent folder:

- ▶ Enter **calculate.jsp** for the name of the page. Select *Configure Advanced Settings*. Deselect create from page template. Click *Next*.
- ▶ The `jsf/core` tag library is defined by default. Because is the first JSF page we cannot add other tag libraries (they are not in the project yet). Application Developer adds other libraries when required by the code. Click *Next*.
- ▶ A style sheet, `Master.css`, is added by default. Application Developer creates automatically another CSS file for JSF code later. Click *Next*.
- ▶ Click *Finish*. Application Developer generates all the necessary information and opens the new page.

Let's examine what has been generated

Application Developer has generates Java classes and control files:

- ▶ Under Java Resources you find a new package named **pagecode**, with two files: **Calculate.java** and **PageCodeBase.java**:

Open `Calculate.java`. The class is empty for now. It is a subclass of `PageCodeBase`.

Open `PageCodeBase`. This class contains shared code for all JSF pages:

- Access to `FacesContext`, `requestScope`, `sessionScope`, and so forth
- `gotoPage`—to switch to another JSF page
- `findComponent`—find any JSF component

- findComponentInRoot—find a JSF component under the root component
- resolveExpression—resolves a JSF reference: #{xxxxx}
- log—simplified test output to System.out.println
- ▶ There is a new style sheet, **stylesheet.css**, in the theme folder. Open the file; it has many predefined styles.
- ▶ In the WEB-INF/lib folder you find the JSF runtime JAR files:
 - jsf-api and jsf-impl are the basic JSF JAR files from Sun
 - jsf-ibm are the IBM extensions
 - jstl and jstl_el are JSP standard tag libraries
 - commons-xxx are common utilities
- ▶ In WEB-INF you find the JSF configuration file, **faces-config.xml**. Open the file. It contains a life cycle listener and already one managed bean, pc_Calculate, which is the Java code for the JSF page.
- ▶ Open the deployment descriptor, web.xml. It contains two servlets. The Faces Servlet is the main servlet for all /faces/* calls. The JS Resource Servlet is an IBM extension for generated JavaScript functions.

Creating the base layout

Now we create the basic JSP layout in the Page Designer:

- ▶ Replace the generated content (Place content here.) with **JSF Calculator**. Select the text, JSF Calculator, and *Insert -> Paragraph -> Heading 1*.
- ▶ Select the *Output* component in the Faces Components palette. Point the cursor under the heading and drop the component (wait till it appears).

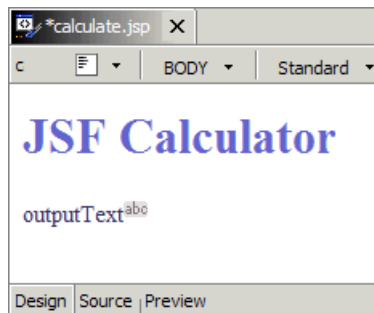


Figure 3-1 Creating the base layout for the calculator

- ▶ Select the outputText component. In the Attributes view, change the Id to **comment**; Enter the value: **Created using Application Developer**; Enter style properties: **font-size: 18; font-weight: bold**.

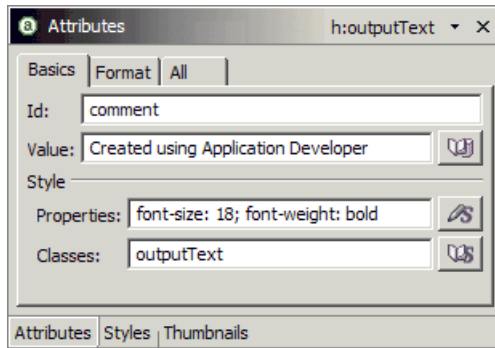


Figure 3-2 Setting attributes for the Output component

- Position the cursor behind the output text and press Enter twice to add
s. Select the Display Errors component and drop it under the breaks. This is for validation error messages.

Select the new component and notice in the Attributes view the messages style class.

- In the Styles view (bottom left) expand the stylesheet.css. Find the .messages class and open it (double-click). In the Set Style Properties window, select *Fonts* and set the color to Red. Click *OK* and the code is inserted into the stylesheet.css file:

```
.messages {
    color: red
}
```

Change the .message entry in the same way (this is used for an error message attached to one field). Save and close the stylesheet.css. The {Error Messages} component is now in red.

- Insert a table with five rows and two columns under the error messages field. Select *Insert -> Table*, change rows to 5, border width to 0, and padding to 3. Click *OK* (Figure 3-3).

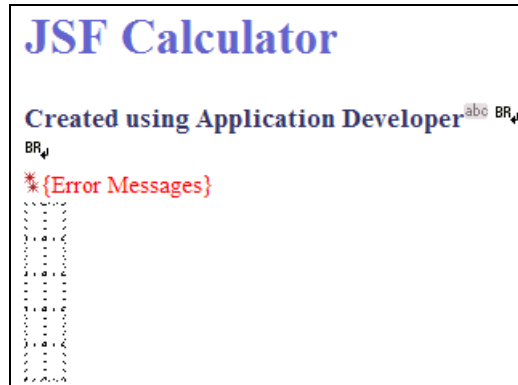


Figure 3-3 Inserting a table component

- Save the JSP page. Open the `Calculate.java` file by selecting *View Page Code* from the context menu inside the JSP in Page Designer (you can also open the Java file itself). Notice the code that has been generated, for example:

```
protected HtmlOutputText comment;
protected HtmlOutputText getComment() {
    if (comment == null) {
        comment = (HtmlOutputText) findComponentInRoot("comment");
    }
    return comment;
}
```

Note that each component is defined with a lazy getter method; therefore we never use the variable in our own methods (always use the get method).

- To fill the table (Figure 3-4):
 - Drop Output components into column 1, rows 1, 2, 3, and 5.
 - Drop Input components into column 2, rows 1 and 3.
 - Drop a Combo Box component into column 2 row 2.
 - Drop a Command - Button component into column 2 row 4.
 - Drop an Output component into column 2 row 5.



Figure 3-4 Inserting JSF UI components into the table

- ▶ Change the Output components in column 1. Select each component and in the Attributes view change the value field (Figure 3-5):
 - Column 1 row 1: **Number 1 (11-888):**
 - Column 1 row 2: **Operation:**
 - Column 1 row 3: **Number 2 (1-19, odd):**
 - Column 1 row 5: **Result:**

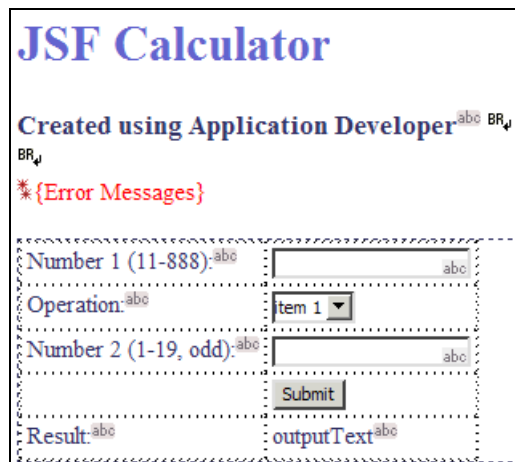


Figure 3-5 Changing the output components

Implementing component attributes and validation

- ▶ Select the first Input component. In the Attributes view Basics tab:
 - Change the id to: **number1**
 - Change the width to: **12**

On the Format tab:

- Select the type as: *Number*
- Number style: *Decimal*
- Select *Integer Only*

On the Validation tab (Figure 3-6):

- Select *Value is required*
- Set minimum and maximum to **11** and **888**
- Select *Add Display Error control*

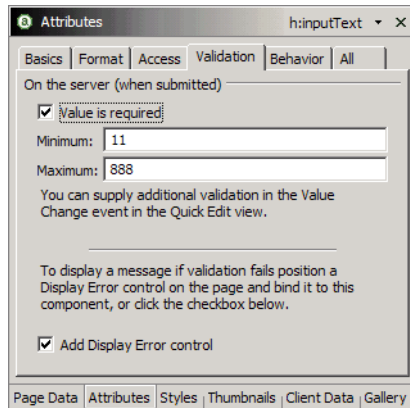


Figure 3-6 Input data validation

- An error message field is added to the right of the input field and is now selected (Figure 3-7).

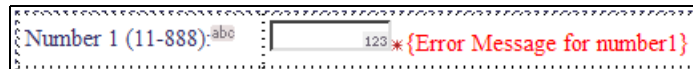


Figure 3-7 Error message for the input component

- Select the input field again. In the Attributes view, Behavior tab set maximum length to **3**.

On the All tab you can see all the values we entered. You could also enter the values there.

- Select the second Input component. In the Attributes view:
 - Basics: id=**number2**, width=**12**
 - Format: *Number, Decimal, Integer Only*
 - Validation: *required*, minimum=**1**, maximum=**19**
 - Behavior: maximum length=**2**

- ▶ Select the Combo Box component. In the Attributes view:
 - Basics: id=**operation**
 - Choices (Figure 3-8): click *Add Text Item* four times. Change the generated names to **add**, **subtract**, **multiply**, and **divide**. Change the generated values to **Add**, **Subtract**, **Multiply**, and **Divide**.

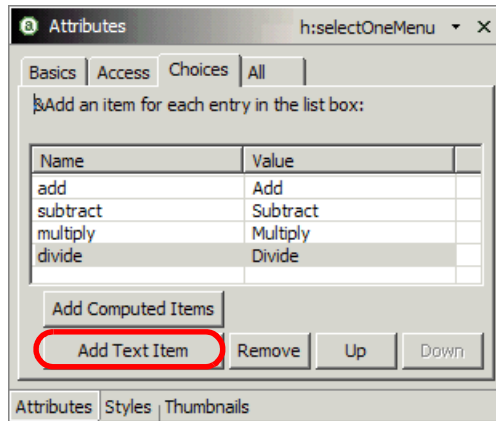


Figure 3-8 Setting choices for the combo box

- ▶ Select the Submit button. In the Attributes view:
 - Basics: id=**calc**
 - Format: label=**Calculate**
- ▶ Select the Result: Output component (column 1). In the Attributes view set the style properties to:


```
color: blue; font-size: 18; font-weight: bold
```
- ▶ Select the last Output component (column 2). In the Attributes view:
 - Set the id to: **result**
 - Set the style properties to:


```
color: blue; font-size: 18; font-weight: bold
```
- ▶ The basic design is shown in Figure 3-9.

Figure 3-9 Basic design for the calculator

Now we can save the JSF page and study some of the tags in the *Source* tab:

► Tag libraries:

```
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://www.ibm.com/jsf/html_extended" prefix="hx" %>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

► JSF style sheet:

```
<LINK rel="stylesheet" type="text/css"
href="theme/stylesheet.css" title="Style">
```

► The whole page is imbedded in a <f:view> tag. Around the table is a <h:form> tag.

► Input field with error message field:

```
<h:inputText styleClass="inputText" id="number1" size="12"
required="true" maxLength="3">
<f:convertNumber type="number" integerOnly="true"/>
<f:validateLongRange minimum="11" maximum="888"></f:validate...>
</h:inputText>
<h:message styleClass="message" id="message1" for="number1"></h:message>
```

► Combo box:

```
<h:selectOneMenu styleClass="selectOneMenu" id="operation">
<f:selectItem itemLabel="add" itemValue="Add" />
<f:selectItem itemLabel="subtract" itemValue="Subtract" />
<f:selectItem itemLabel="multiply" itemValue="Multiply" />
<f:selectItem itemLabel="divide" itemValue="Divide" />
</h:selectOneMenu>
```

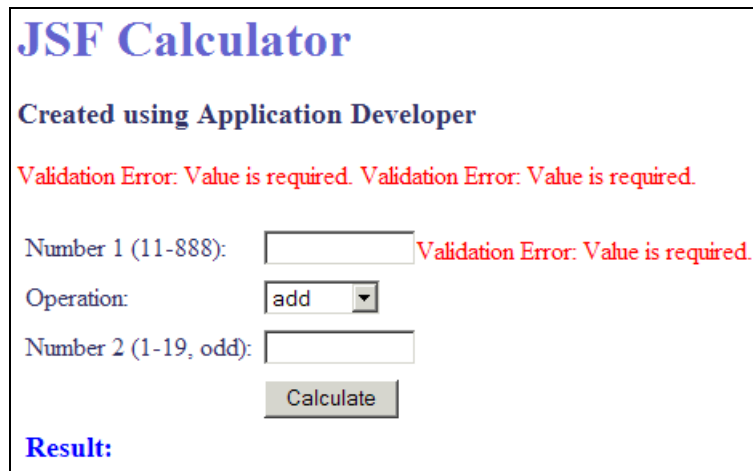

- Submit button (uses an IBM extension tag):

```
<hx:commandExButton type="submit" value="Calculate"
                    styleClass="commandExButton" id="calc"></hx:...>
```

Testing the validation

At this point we can test the validation. To do this, start the JSFServer in the Servers view. Once the server is ready, select the `calculate.jsp` and *Run on Server*. The page appears in the browser.

Leave the field empty and click *Calculate*. You get two error messages in the common error message field, and one message in the error message field of number 1 (Figure 3-10).



The screenshot shows a web page titled "JSF Calculator" with the subtitle "Created using Application Developer". Below the title, there are two red error messages: "Validation Error: Value is required." and "Validation Error: Value is required.". The form contains three input fields: "Number 1 (11-888):", "Operation:", and "Number 2 (1-19, odd):". The "Number 1" field is empty and has a red error message "Validation Error: Value is required." next to it. The "Operation:" field is a dropdown menu with "add" selected. The "Number 2" field is empty. Below the input fields is a "Calculate" button. At the bottom of the form, there is a "Result:" label.

Figure 3-10 Testing the validation

Enter 9 and 2 for the two numbers. You get a validation error for the first number. Enter an invalid value for the second number and you get an error message as well. Enter valid values and no error message are displayed.

Note that the second number is only validated for the range. Standard validation does not check for “odd” numbers. We have to implement that ourselves.

Binding the frontend to the calculator

With the GUI design complete we can hook it up to the `CalculatorBean` to perform the operations and display the result.

With the JSF page open in Page Designer, go to the Page Data view. In the context menu select *New -> JavaBean*. In the Add JavaBean dialog (Figure 3-11):

- ▶ Enter **calculator** as name.
- ▶ For the class click the *Browse* icon, then enter `Calcu` to locate the `CalculatorBean`. Click *OK*.
- ▶ Select *Make this JavaBean reusable* (it is registered in the configuration file). Enter a description (`Calculator Bean`) and select *session* for the scope.
- ▶ Click *Initialize Properties*. In the dialog click *Add*. In the name column select the *operation* property. In the value column enter `Subtract` and press *Enter*. Click *OK*. When we run the code, the subtract operation is preselected.
- ▶ Click *Finish* and the bean is added to the Page Data view. Save the JSP.

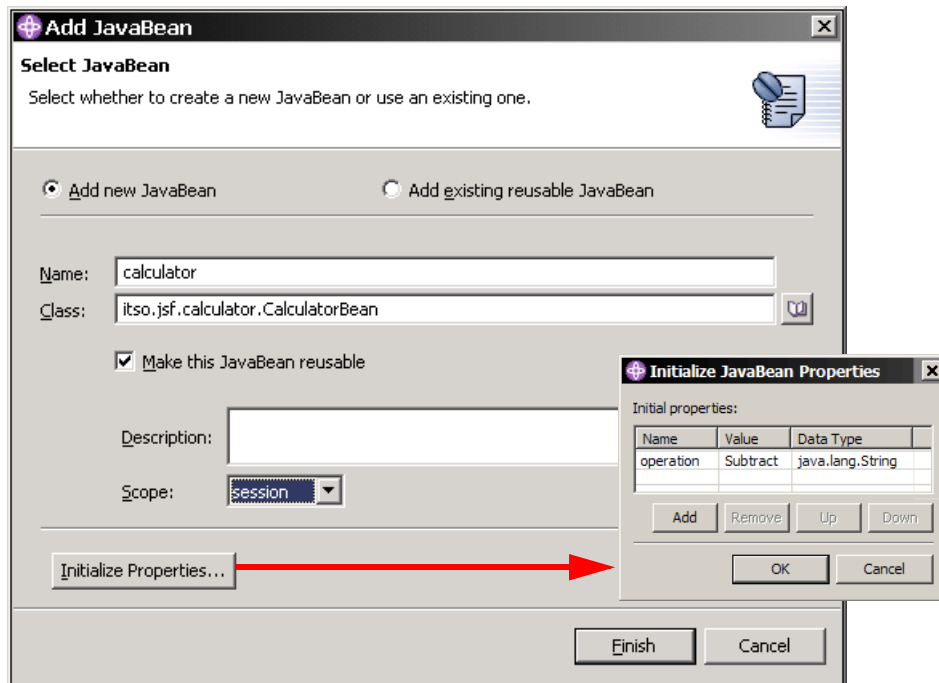


Figure 3-11 Defining the JavaBean

- ▶ In the `faces-config.xml` file you can see the new managed bean. The calculator bean is also added to the `Calculate.java` code with a `getCalculator` method.

- ▶ Select the first input field (number1). In the Attributes view, for the value field, click the  icon and select the number1 property of the calculator bean. Notice the generated binding: #{pc_Calculate.calculator.number1}.
- ▶ Repeat this for the number2 input field and bind it to the number2 property.
- ▶ Expand the calculator bean in the Page Data view. Select the operation property and drag/drop it onto the combo box. This has the same effect. Select the result property and drag/drop it onto the output field in column 2.
- ▶ The property names appear in the Design view for the input and output fields (Figure 3-12).

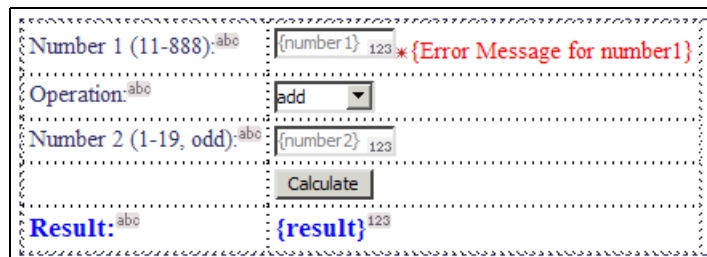


Figure 3-12 Binding properties to the JSF components

Testing the binding

The frontend is now coupled with the calculator bean and we can test the binding:

- ▶ In the Servers view select the JSFServer and *Restart Project* -> *ItsoJSF03Calc*. This is necessary when the configuration file changes.
- ▶ Run the calculate.jsp. Notice that no values are displayed in the number fields and subtract is displayed for the operation. These values come from the calculator bean. Enter some values and click *Calculate*. You can see that the values are assigned to the bean by watching the test output in the Console view.

Invoking the business logic of the calculator

To actually use the calculator we have to invoke the calculate method of the bean when *Calculate* is clicked:

- ▶ In the Page Designer select the *Calculate* button. In the Quick Edit view, select *Command* in the left pane. Click in the right pane and this sample code appears:

```
// Type Java code to handle command event here
// Note, this code must return an object of type String (or null)
return "";
```

- Replace the generated action code with this code from `\labscode\dev-calculator\Submit.txt`:

```
log("CalculateAction start");
try {
    getCalculator().calculate();
} catch (Exception e) {
    log("Calculator-Exception: "+e.getMessage());
    if (getCalculator().getErrorMessage() == null)
        getCalculator().setErrorMessage("Exception: "+e.getMessage());
} finally {
    log("CalculateAction end");
}
return null;
```

- Save the `calculate.jsp`. Open the `Calculate.java` file and notice that the action code has been added in the `doCalcAction` method. We can make further changes either in the Quick Edit view or in the Java class.
- Select the Calculate button. In the Attributes view, All tab, you can see that the action is set to: `#pc_Calculate.doCalcAction`. This is the link to the Java code.
- Rerun the `calculate.jsp` (Figure 3-13). Perform some calculations. Try the divide operation with values that result in a non-integer result. Notice the exception in the Console. The result is still calculated. We will implement an error page to display the error message in the next section.

JSF Calculator

Created using Application Developer

Number 1 (11-888):

Operation:

Number 2 (1-19, odd):

Result: 85

Figure 3-13 Using the calculator Web application

Implementing an error page

The exception error could be reported back on the same page, but to illustrate page switching we implement an error page:

- ▶ Under WebContent, create a new JSF Faces page named **error.jsp**. Replace the content with a heading 1, JSF Calculator Error.
- ▶ In the Page Data view, click *New -> JavaBean*. In the dialog, select *Add existing reusable JavaBean*. Select the calculator bean and click *Finish*.
- ▶ Expand the calculator bean. Select the `errorMessage` property and drag/drop it under the heading. In the Insert JavaBean dialog (Figure 3-14), select *Displaying fields*, change the label to **Error message:**, and click *Finish*.

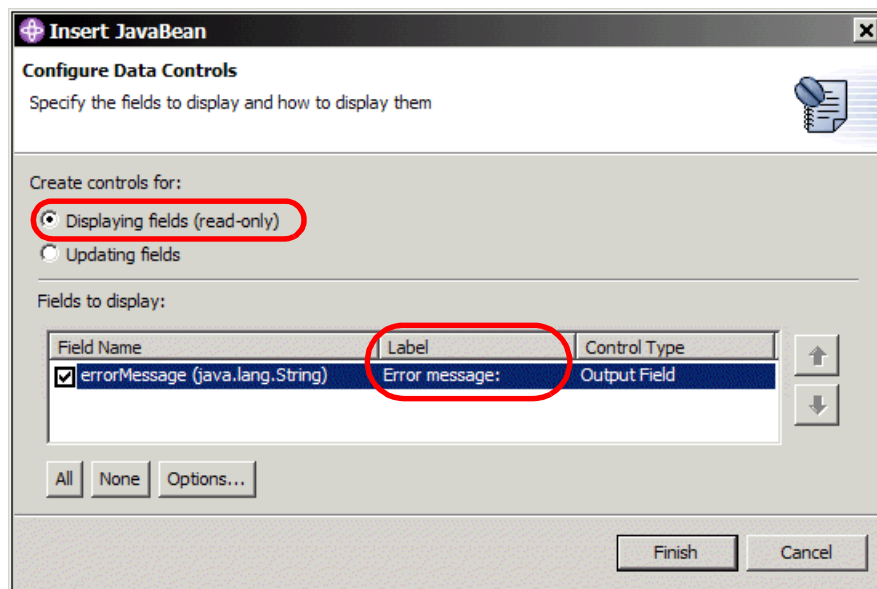


Figure 3-14 Binding the error message of the bean to the error JSP

This creates a table with an output text constant (Error message:) and an output text bound to the `errorMessage` property, plus a generic Error Messages component.

- ▶ Add a Command - Button at the bottom, and change the label in the Attributes view, Format tab to Back. Save the `error.jsp` (Figure 3-15). Notice the matching `Error.java` code in the `pagecode` package.

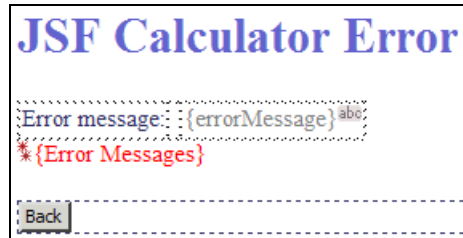


Figure 3-15 Error page design

Implementing page navigation

Next we implement the navigation between the `calculate.jsp` and the `error.jsp`. Navigation rules are stored in the configuration file. There are local rules (for one JSF page) and global rules.

- Select the Back button. In the Attributes view, Navigation tab, click *Add* to add a navigation rule. Complete the dialog by selecting the `calculate.jsp` page, entering the alias name as **back**, and ***** for the action (meaning any action). Click *OK*. This completes the navigation for the error page (Figure 3-16). Save the `error.jsp` file.

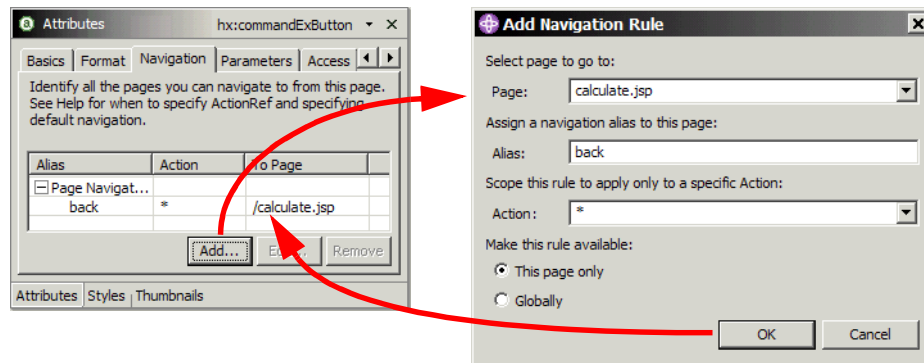


Figure 3-16 Adding a navigation rule

- Select the Back button and in the Attributes view, All tab, set the action attribute to **back**. The symbolic alias name of the action can be entered directly, or in the action logic (what we will do for the `calculate.jsp`). Save the `error.jsp`.
- In the `calculate.jsp` select the Calculate button. In the Attributes view, Navigation tab, add two rules: **calculate** (stay on the page), and **error** (go to the error page). Define the error rule as a global rule.

For our simple case a global rule is not really necessary, but it illustrates the concept of the global rules (Figure 3-17).

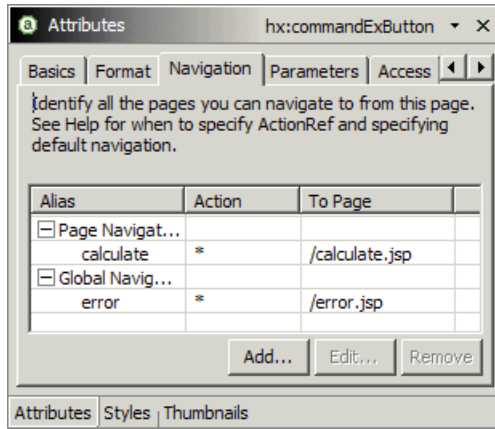


Figure 3-17 Navigation rules defined for the calculator

- Open the faces-config.xml file and locate the code that was added for the navigation rules:

```

<navigation-rule>
  <from-view-id>/error.jsp</from-view-id>           <=== local rule
  <navigation-case>
    <from-outcome>back</from-outcome>
    <to-view-id>/calculate.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/calculate.jsp</from-view-id>       <=== local rule
  <navigation-case>
    <from-outcome>calculate</from-outcome>
    <to-view-id>/calculate.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>                                   <=== global rule
  <navigation-case>
    <from-outcome>error</from-outcome>
    <to-view-id>/error.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

```

- Next we change the action code for the Calculate button. In case of an exception we go to the error page, otherwise we stay on the calculate page. We return one of the two symbolic names, calculate or error.

You can make the changes in the Quick Edit view or in the `Calculate.java` file (use the sample code in `\labscode\dev-calculator\SubmitNavigate.txt`):

```
log("CalculateAction start");
try {
    getCalculator().calculate();
    return "calculate"; // stay on the calculate page
} catch (Exception e) {
    log("Calculator-Exception: "+e.getMessage());
    if (getCalculator().getErrorMessage() == null)
        getCalculator().setErrorMessage("Exception: "+e.getMessage());
    return "error"; // go to the error page
} finally {
    log("CalculateAction end");
}
return null; // remove the existing return
```

To test the calculator with navigation, we have to restart the enterprise application (we made changes to the configuration file):

- ▶ Select the JSFServer and *Restart Project -> ItsoJSF03Calc*.
- ▶ Run the application (`calculate.jsp`), then try a division that produces the exception. The error page is displayed. Click *Back* to get back to the calculator.
- ▶ You may want to change the error message to red by adding **color: red** to the style properties (Attributes view).

Note: Instead of returning the "calculate" string, we can return `null` or an empty string and the `calculate.jsp` is redisplayed.

Implementing a validator

We want to restrict the second number to odd values. There is no standard validator for this. We have to write a validator, register it in the configuration file, and link it to the `number2` field:

- ▶ Select the `itso.jsf.calculator` package and *New -> Class*: enter **OddValidator** as name. Click *Add* for Interfaces, locate the `javax.faces.validator.Validator` interface and click *OK*. Select *Inherited abstract methods*. Click *Finish*.
- ▶ The skeleton class is generated with this method skeleton:

```
public void validate(FacesContext arg0, UIComponent arg1, Object arg2)
    throws ValidatorException { .....
```

`FacesContext arg0`—gives access to all components

- `UIComponent arg1`—is the component to be validated
- `Object arg2`—is the value (of the component) to be validated
- Replace the method body with the sample code below (use the code in `\labscode\dev-calculator\OddValidator.txt`):


```

System.out.println("OddValidator start");
UIInput field = (UIInput)arg1;
int value = ((Long)arg2).intValue();
System.out.println("Field="+field.getId()+" Value="+value);
if (value%2 == 1) {
    field.setValid(true);
    System.out.println("OddValidator end: valid");
} else {
    System.out.println("OddValidator end: invalid");
    FacesMessage errmsg = new FacesMessage
        (FacesMessage.SEVERITY_ERROR,
         "2nd number not odd.",
         "Second number must be odd.");
    throw new ValidatorException(errmsg);
}
      
```
- To resolve the classes, select *Source -> Organize Imports*.
- We access the `arg1` field to display the name. We can use `UIInput` or `HtmlInputText` classes.
- We get the value from `arg2` because we know its data type.
- In case of error we create an error message, a **FacesMessage**. The first message is the *summary* message (displayed in the global error message field), the second message is the *detailed* message (displayed in an error message field attached to the input field—we did not define one).
- Save and close the `OddValidator`.
- Open the `faces-config.xml` file and add these lines at the bottom (`\labscode\dev-calculator\OddValidatorRegister.txt`):


```

<validator>
    <description>
        Registers the OddValidator
    </description>
    <validator-id>oddValidator</validator-id>
    <validator-class>itso.jsf.calculator.OddValidator</validator-class>
</validator>
      
```
- In the `calculate.jsp`, select the `number2` field and go to the Source page in the Design view:
 - Place the cursor just before the end of the `number2` field:


```

.....</f:validateLongRange> PUT CURSOR HERE </h:inputText>
          
```

- In the menu bar, select *JSP -> Insert Custom*. In the Insert Custom Tag dialog select the “f” tag library (left) and *validator* (right) and click *Insert*. Then click *Close*. The custom tag is inserted.
- Enter **oddValidator** for the validatorId attribute, either directly in the Source, or in the Attributes view. This generates the tag as:

```
<f:validator validatorId="oddValidator"></f:validator>
```
- Save the calculator.jsp.
- ▶ Restart the enterprise application. Enter an even value and you should see the error message. Notice that the *summary* message is displayed. Notice the test output in the Console.

Implementing a value change event

With a value change event we can monitor changes to values. The event can be invoked immediately or when the next action (button) is performed. As an example we monitor the change of the operation combo box:

- ▶ Select an Output component and drop it on the right of the operation combo box (Figure 3-18). We place a text there when the operation has changed. In the Attributes view change the id to **opchanged**.

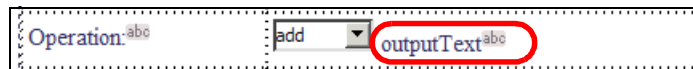


Figure 3-18 Creating an output text to monitor changes in the combo box values

- ▶ Select the operation combo box. In the Quick Edit view, select *Value Changed* (left), click in the code pane (right), and enter this code (`\labscode\dev-calculator\ValueChanged.txt`):

```
log("OperationValueChangeEvent start");
HtmlSelectOneMenu operation =
    (HtmlSelectOneMenu)valueChangedEvent.getSource();
String opnew = (String)valueChangedEvent.getNewValue();
String opold = (String)valueChangedEvent.getOldValue();
log("operation old="+opold+" new="+opnew);
getCalculator().setOperation(opnew); // is also done later by binding
getOpchanged().setValue("changed"); // change the output text field
getCalc().setValue(opnew);           // change the Calculate button
log("OperationValueChangeEvent end");
```

The method that is generated has one parameter: `valueChangedEvent`. From this parameter we can access the source component and the old and new values. We set the `opchanged` output field to: `changed`. We use the new value to change the label of the Calculate button.

Notice in the Attributes view, All tab, the `valueChangeListener` is set to `#{pc_Calculate.handleOperationValueChange}`.

- Save the `calculate.jsp`. Open the `Calculate.java` file and you can find the `handleOperationValueChange` method with your code:

```
public void handleOperationValueChange(ValueChangeEvent valueChangeEvent){  
    .....  
}
```

- Run the `calculate.jsp`. You can see in the Console that the event is invoked. However, no change is visible in the GUI. This is because the event is not immediate and the refresh of the GUI after the action loses the changes if a navigation rule is involved. (If our action logic returned null, the changes would be visible.)
- To make the event immediate, select the operation combo box. In the Quick Edit view select *onchange* (left), click in the code pane (right) and enter this code:

```
document.forms['form1'].submit();
```

This creates a JavaScript function that is invoked from the combo box:

```
<SCRIPT type="text/javascript">  
function func_1(thisObj, thisEvent) {  
    //use 'thisObj' to refer directly to this component instead ... 'this'  
    //use 'thisEvent' to refer to the event generated instead of ... 'event'  
    document.forms['form1'].submit();  
}</SCRIPT>
```

In the Source code of the combo box you find:

```
<h:selectOneMenu ..... onchange="return func_1(this, event);">
```

- Save and retest. When you change the operation, you see the immediate result (Figure 3-19).

The screenshot shows a web application titled "JSF Calculator" with the subtitle "Created using Application Developer". It contains two input fields: "Number 1 (11-888):" with the value "23" and "Number 2 (1-19, odd):" with the value "1". Between them is an "Operation:" dropdown menu currently set to "multiply". A red circle highlights the word "changed" next to the dropdown arrow. Below the inputs is a "Multiply" button, also circled in red. At the bottom, the "Result:" is displayed as "23".

Figure 3-19 Result of value changed event

Note that the button changes back to *Calculate* at the next operation because a navigation rule is involved. If you change the action logic to return `null`, the changes to the user interface are permanent.

Implementing internationalization

To create JSF pages that can run in multiple languages we use the Internationalization feature. We put the text constants and error messages into a properties file that is used by the JSF components and by the logic. We only implement a few constants to illustrate the concept:

- Select **Java Resources** and *New -> Other -> Simple -> File*. Enter **constants_en.properties** as file name and click *Finish*.

In the editor, enter these lines, then save and close the file:

```
page_comment=* Created using Application Developer *
text_operation=Calculator operation:
text_calculate=Perform Calculation
```

- In the `calculate.jsp` Source code, after the `jsf/core` tag library, enter:

```
<f:loadBundle var="constants" basename="constants"></f:loadBundle>
```

This tag provides access to the text constants for the selected language.

- In the Design view (Figure 3-20), select the comment field (Created using Application Developer). In the Attributes view, change the value attribute to:

```
#{constants.page_comment}
```

Similarly change the Operation: text to:

```
#{constants.text_operation}
```

Change the Calculate button label in the Format tab to:

```
#{constants.text_calculate}
```

- All JSF error messages come in multiple languages. For your own error messages generated in action code, you would have to load the constants bundle and retrieve the messages according to the language. We do not implement internationalization for error messages in this example.

Note: To implement multiple languages you have to create multiple `constants_xx.properties` files, where `xx` is the language. In addition you have to change the `faces-config.xml` file as described in “Internationalization” on page 236.

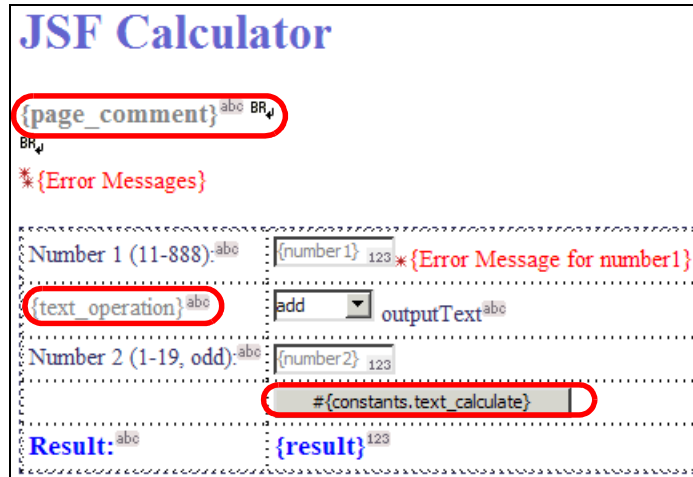


Figure 3-20 Changing components to run in multiple languages

Restart the project in the server, then run the `calculate.jsp`. The texts have changed to the values contained in the resource file (Figure 3-21).

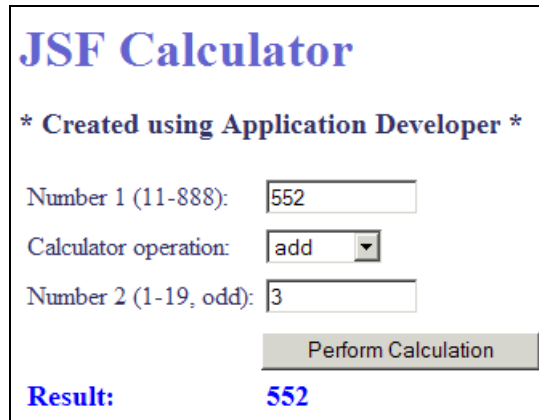


Figure 3-21 Calculator Web application

Implementing page initialization

To illustrate page initialization we implement the Page Load Begin event to put values into the number fields of the calculator bean:

- ▶ Select the `<hx:scriptCollector>` tag either in the Source page, or in the Design page in the menu bar of the JSP.

- In the Quick Edit view, select the Page Load Begin event and add this logic:

```
log("Calculator is loading");
if ( getCalculator().getNumber1() == null) {
    getCalculator().setNumber1( new Long(333) );
    getCalculator().setNumber2( new Long(3) );
}
```

- Save and open the page code. The logic is added into an onPageLoadBegin method. The binding is visible in the Attributes view All tab:

```
preRender: #{pc_Calculate.onPageLoadBegin}
```

Note: If the logic is added to the Page Load End event then changes to managed beans are not reflected in the user interface because rendering has already passed.

- Rerun the application and the number fields are initialized.

Complete solution

We provide the complete solution for the calculator in the additional material in the file `\labscode\solution\ItsoJSF03Calc.ear`. See “Importing a solution enterprise application” on page 548 for instructions.



JSF components

This chapter describes the JSF components that are included with WebSphere Studio Application Developer 5.1.2. These components offer a rich set of function that can be used to create dynamic J2EE Web applications.

We explain each of the components and how they can be included within a Faces pages. We discuss the key attributes that can be modified. We also include a simple example of each component and how it works within Studio.

Overview

WebSphere Studio Application Developer fully implements the JSR-127 specification (see “JSF specification” on page 7). This specification stipulates that all supporting development tools and runtimes are required to implement the standard tag libraries.

JavaServer Faces (JSF) has been designed to allow third party component developers to extend the framework. These component libraries can be easily configured within a JSF page. WebSphere Studio includes the default libraries defined in the JavaServer Faces specification as well as some developed by IBM.

Component model

JavaServer Faces technology provides a set of UI component classes and associated behavioral interfaces that specify all the UI component functionality, such as holding component state, maintaining a reference to objects, and driving event handling and rendering for a set of standard components (Figure 4-1).

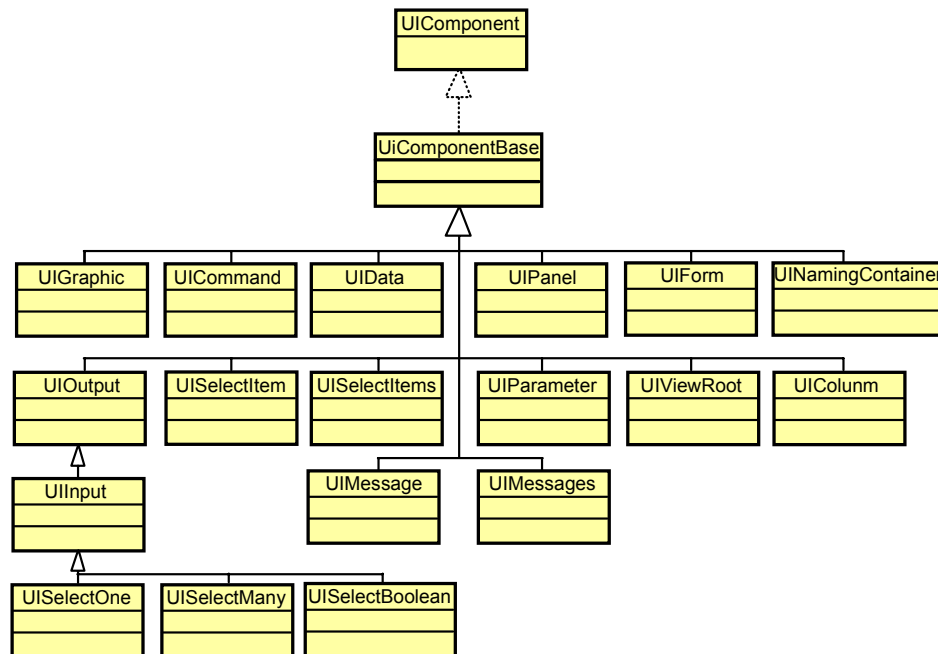


Figure 4-1 Model component hierarchy

Renderers

The JavaServer Faces component architecture is designed such that the functionality of the components is defined by the component classes. In the other hand, the component rendering can be defined by a separate renderer. This design has two key benefits:

- ▶ When you write a component you can define its behavior once and then create multiple renders for it. Each of the different renders defines a way to render the component for an specific client or for any different clients.
- ▶ You can easily change the appearance of a component by selecting the tag that represents the combination of your component and your render.

A *render kit* defines how component classes are mapped to component tags that are appropriate for a particular client.

The JavaServer Faces implementation includes a standard `RenderKit` class for rendering to an HTML client. For every UI component that a render kit supports, the render kit defines a set of `Renderer` objects. Each `Renderer` defines a different way to render the particular component to the output defined by the render kit.

For example, a `UISelectOne` component has three different renderers. One of them renders the component as a set of radio buttons. Another renders the component as a combo box. The third one renders the component as a list box. The tags used for the three renders are `<h:selectOneRadio>`, `<h:selectOneMenu>` and `<h:selectOneListBox>` respectively.

Each JSP custom tag defined in the standard `HTML RenderKit` class is composed of the component functionality (defined in the `UIComponent` class) and the rendering attributes (defined by the `Renderer`). The JavaServer Faces implementation provides a custom tag library for rendering components in HTML.

JSF libraries

A JSF library contains components that can be used to develop functionally rich Web applications. They contain components that can display text and react to events. They can be included with a JavaServer Page and can be declared using `<XML>`. All faces libraries are built using the JSP tag library specification and can be easily declared with their own namespace.

Standard JSF libraries

When a Faces page is created within WebSphere Studio the Core library is automatically added to the page. Other libraries are automatically added as required by components of the page. JSF out of the box consists of two libraries (Table 4-1).

Table 4-1 JSF standard libraries

Library	URI	Prefix
Core	http://java.sun.com/jsf/core	f
HTML	http://java.sun.com/jsf/html	h

In the calculator example (see Chapter 3, “JSF calculator example” on page 61) you have been through the process of creating a Web project and then created a Faces page. If you open one of these pages in Page Designer, you see the following tag library definition at the top of the page.

```
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://www.ibm.com/jsf/html_extended" prefix="hx"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

You can see the two standard libraries defined as well as the HTML extension library. The redbook includes a chapter that discusses how to develop custom components (see Chapter 14, “Extending the JSF framework” on page 479).

Using Faces in Studio

The JavaServer Faces developer spends most of the time using three views of the Web perspective. These views expose the key functionality for driving rapid application development with JSF components, and they enable easy construction of pages through drag and drop and editing of the component attributes.

For more details about the Faces support in WebSphere Studio Application Developer refer to Chapter 2, “JSF support in WebSphere Studio” on page 25.

Studio operations

When developing Web application using the components using WebSphere Studio Application Developer, there are four basic steps (Figure 4-2):

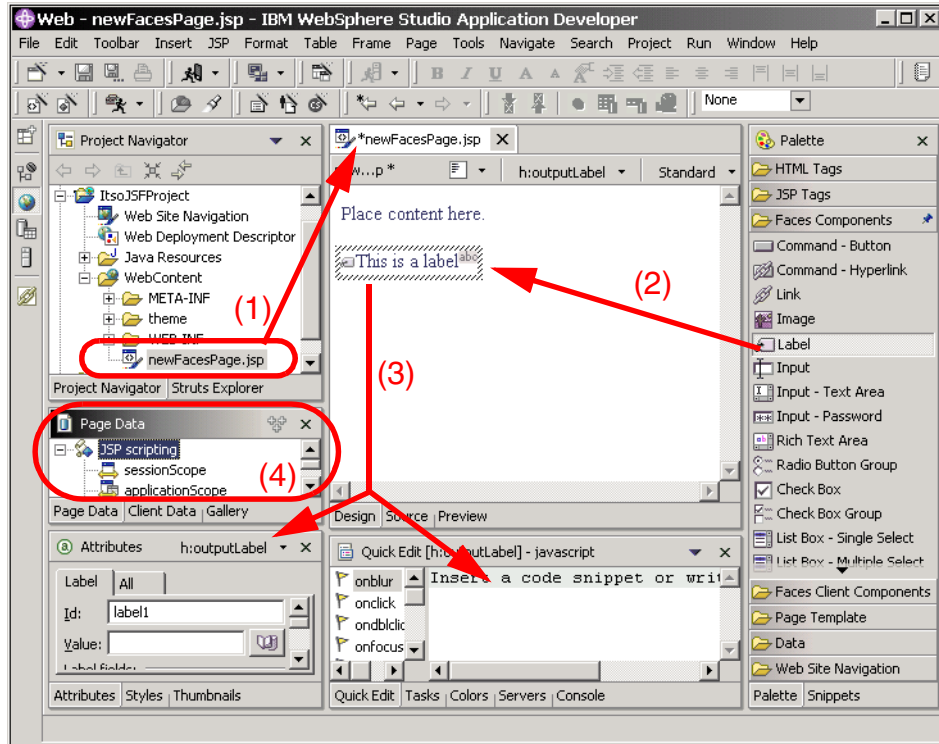


Figure 4-2 Example of JSF development using WebSphere Studio

1. Create a Faces page.
2. Drag a component (for instance, Label) from the palette onto the page.
3. Adjust the component attributes through the Attributes view. Change component behavior through the Quick Edit view.
4. Drag a property from the Page Data view to bind with the component.

Component attributes and events

In this section we discuss about attributes and events that all the JSF components have.

Common attributes

The HTML renderkit is based on HTML 4.01 and supports a range of common attributes that are shared across the other components within the render kit.

These attributes can be modified through the Attributes view. A proficient developer can also modify the attributes through the Source tab of the Page Designer.

Note: These modification are synchronized so as each property is changed the other views are updated

You can find the common attributes listed in Table 4-2.

Table 4-2 List of common HTML 4.01 attributes

Attributes	Description
id	The id attribute is a unique reference that is used to identify the component. It can be used by other components as a reference. A drag operation from the palette automatically generates a unique id for a component.
value	A JSF component can modify data within a managed bean through the <i>value</i> attribute. This attribute can contain a literal value ("101") or a data binding notation value (<code>{account.accountNum}</code> would reference the <code>getAccountNum</code> method of the <code>Account</code> bean).
rendered	Can be set to true or false: ▶ false—component is not rendered to the browser at runtime ▶ true—component is rendered
immediate	If true, skip straight to render phase of life cycle.
binding	The <code>UIComponent</code> that represents this component, usually in the backing file.
accesskey	Assigns an access key to the component, which should be a single keyboard character. At runtime pressing an access key assigned to a component gives focus to the component.
dir	Specifies the component's direction when rendered at runtime. The values can be either: - ltr—left to right - rtl—right to left
disabled	If disabled is set to true, the component is displayed at runtime but will not be functional. If left blank, the default value is false.

Attributes	Description
lang	Specifies the base language of the component's attribute values and text content. Takes ISO-standard language abbreviation codes (for example, "en" for English, "en-US" for American English, "fr" for French, "de" for German).
styleClass	Space-separated list of CSS style classes to be applied when this element is rendered. This value must be passed through as the class attribute on generated markup.
style	Specifies CSS style information for the component (for example, style="font-size : 8pt ; color : red").
target	Specifies the position of the component in the tabbing order in the JSP. This value must be a number between 0 and 32767.
title	Specifies the title text, shown in browser as a tooltip at runtime.

Note: Within the sections that follow these attributes are referred as *common*.

Tip: A drag operation from the palette to the Page Designer automatically generates a unique id for a given component. If you manually edit the id you have to make sure that the id value is unique.

Each attribute within a JSF application can support a concept of value binding (see “Value binding” on page 40). The section that follows explains how value binding works.

Binding

Binding in JSF is the language used to denote the mapping of a UI component to a model object.

Value binding

UI component values can be bound to a variety of model objects, such as primitive types, collections, and arrays. These value bindings are typically stored in the value attribute of a UI component tag. The binding syntax allows to traverse through model objects until you find the appropriate type of field for the component.

Note: The value binding language in JSF is syntactically identical to the Expression Language (EL) defined in the JavaServer Pages Specification 2.0. The only differences are that JSF uses the delimiter `#{ }` instead of the delimiter `${ }`, and that EL functions are not supported in JSF value bindings.

For example, this input component's value is dynamically produced from the name property of the `customerData` managed bean included in this page (Figure 4-3).

```
<h:inputText id="text1" value="#{customerData.name}"/>
```

Figure 4-3 Example of value binding

Value expressions can be more complex than simple bean property accessors (Table 4-3).

Table 4-3 Example JSF value binding expressions

Expression	Meaning
<code>#{customerData.name}</code>	The bean property name
<code>#{customerList[3]}</code>	The third item in the array <code>customerList</code>
<code>#{customerMap["John"]}</code>	The element identified by the key <code>John</code> in the <code>customerMap</code> collection
<code>#{customerList[3].name}</code>	The name property of the third item in the array <code>customerList</code>

JSF provides a number of objects which can be used in value binding which are available on every page of the application. For example, the following input field value is supplied by the `userid` property of the `HTTPSession` (Figure 4-4).

```
<h:inputText id="text1" value="#{sessionScope["userid"]}"/>
```

Figure 4-4 Example of value binding to the session map

Please refer to the JSF specification for an exhaustive list of implicit objects, and for more detail in general about value binding.

Method binding

JSF method binding expressions are a variant of JSF value binding expressions. To be valid for method binding, a Java method must have no parameters and return a `java.lang.String`. The method name is then used like a bean property in the expression.

A method binding expression is typically set in the `action` attribute of a UI component tag. For example, clicking a button invokes the `doUpdateAction` method in the `pc_UpdateCustomer` bean during the Invoke Application phase of request processing (Figure 4-5). Refer to the JSF specification for more details on the method binding language.

```
<hx:commandExButton type="submit" value="Submit" id="button1"
    action="#{pc_UpdateCustomer.doUpdateAction}"/>
```

Figure 4-5 Example of method binding

Attributes view

The Attributes view is used to modify attributes for a selected component in the Page Designer. You can see an example of the Attributes view in Figure 4-6.

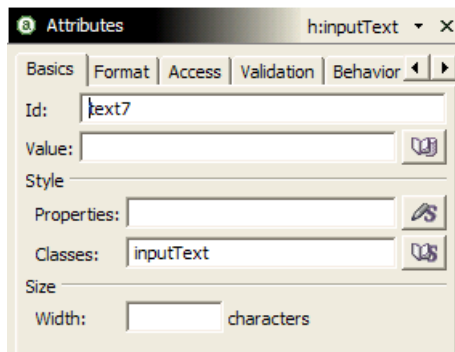


Figure 4-6 Attributes view for an Input component

The Attributes view is discussed in “Attributes view” on page 39.

Common attribute editors

The All tab in the Attributes view lists all the attributes that a component supports. There are a set of common editors that can be used to define these attributes.

In the example of Figure 4-7 you can see the common editors for the value, style properties, and style classes attributes, and the value editor opened. You can use either the input field or the icon to set these attributes.

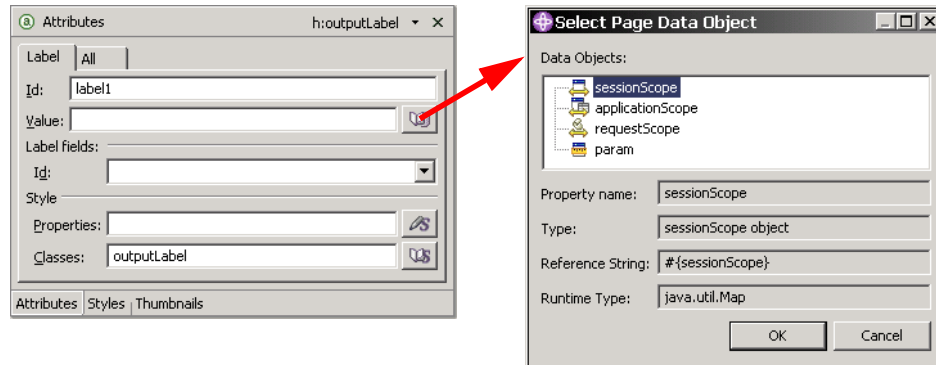


Figure 4-7 Common editor example

Common events

Most JSF components support a common set of events, which can be triggered by actions being performed on the component. To create application flow behavior the developer has to add logic to these events. As all the components are rendered using the HTML render kit and the HTML 4.0.1 events are supported for the components.

In WebSphere Studio JSF events have two categories:

- ▶ JavaScript events which are activated on the client in the Web browser. The logic for these events is developed in JavaScript (Table 4-4).
- ▶ Java events which are activated on the server in the page code class that is bound to each JSF page. The logic for these events is developed in Java.

Table 4-4 List of events supported by HTML render kit

Event	Type	Description
onblur	JavaScript	Code executed when this element loses focus.
onclick	JavaScript	Code executed when a pointer button is clicked over this element.
ondblclick	JavaScript	Code executed when a pointer button is double clicked over this element.
onfocus	JavaScript	Code executed when this element receives focus.
onkeydown	JavaScript	Code executed when a key is pressed.
onkeypress	JavaScript	Code executed when a key is pressed and released over this element.

Event	Type	Description
onkeyup	JavaScript	Code executed when a key is released.
onmousedown	JavaScript	Code executed when a pointer button is held down.
onmousemove	JavaScript	Code executed when a pointer button is moved within this element.
onmouseout	JavaScript	Code executed when a pointer button is moved away from this element.
onmouseover	JavaScript	Code executed when a pointer button is moved onto this element.
onmouseup	JavaScript	Code executed when a pointer button is released.

Logic with Quick Edit

The Quick Edit view (Figure 4-8) is designed to help the JSF developer adding action events without being exposed to the full source code editor of the JSP or its page code class. For example, a Web designer does not have to see the complete page code class to work with action events.

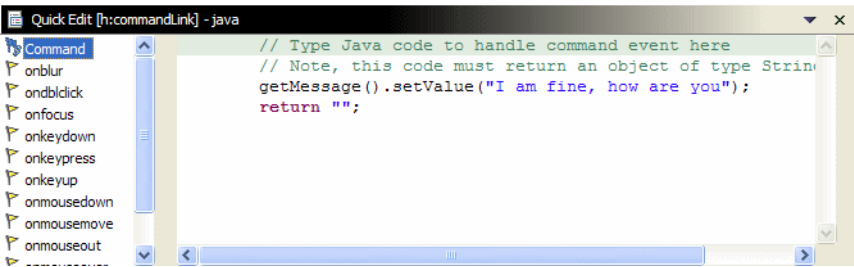


Figure 4-8 Quick Edit view with the Command event displayed

The other key benefit of using the Quick Edit view is that it displays both the client and server events in the same area, therefore enabling a complete view of the events of a selected component. You can enter JavaScript and Java code to react to the different event types.

Component libraries

We have already mentioned that JSF specification includes two common libraries: *Core* and *HTML*. WebSphere Studio includes a range of other libraries that extend the JSF tooling to allow even greater flexibility in creating J2EE applications. These libraries all use a combination of scripting, CSS, DHTML and

J2EE to allow them to be used with any J2EE compliant application server. The benefit to users of Studio is that this rich set of components does not have to be developed from scratch. As you work through more examples of this book you experience the productivity advantages of using JSF and how the Studio tools makes the use of these components easier.

The sections that follow provide details about the libraries and examples how each component can be used in Studio.

Libraries included with WebSphere Studio

WebSphere Studio includes a set of JSF libraries that can use used in your application. These libraries offer a range of components that can be used to construct powerful and easy to develop applications:

Core	The core components are used to manage events and the tags that are used to hold the JSF framework together.
HTML	Web applications are the most common of solutions built with J2EE and the JSF framework includes a set of HTML components that are used for HTML-based applications.
HTML Extensions	The base set of HTML components does not cover all areas of user interface interaction and IBM has created a set of HTML extensions that make these operations easier with JSF and Studio.
Tabbed Panel	This component allows you to have a pile of panels in the same area of the window. Only one panel shows at a time, but you can navigate between the panels by using tabs. Each tab has an associated panel that is displayed when you select the tab.
Rich Text Editor	This component displays an editor similar in design to WordPad in Windows®. HTML output. is generated.
Faces Client	The libraries described so far offer simple components that can be wired into more complex applications. To assist in the improved productivity of Studio and JSF, a set of four client components has been included: Graph, Data Grid, Tree View and Web Service. This library is discussed in Chapter 13, “Faces Client framework” on page 439.

These libraries can be added through the operation of dragging and dropping a control into a page. If the JSF page does not already include that library it is automatically added to the page.

If you import a JSF page into your project and the libraries are not used already in the Web project, you have to import the libraries manually:

- ▶ Select the Web project and *Properties* in the context menu.
- ▶ In the pop-up menu, select the *Web Project Features* option.
- ▶ In the right panel, select the features you want to include in the project (Figure 4-9).

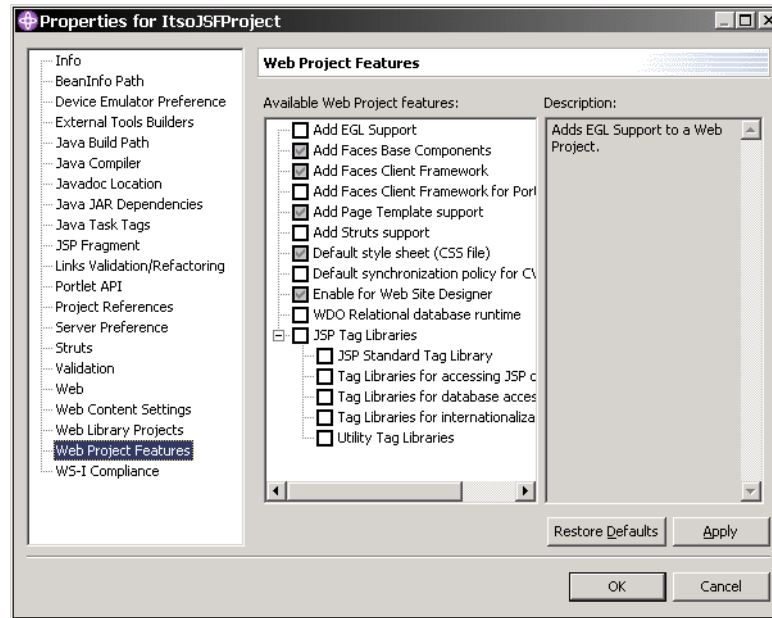


Figure 4-9 Adding features to a Web project

This operation adds to all the necessary libraries for the required functionality to the project class path.

The sections that follow describe in detail the behavior of each component and how you would use it in the Page Designer of WebSphere Studio.

Core library

The JSF core library is designed to have components that are independent of the rendering kits. This allows operations like conversion and validation to be kept separate from the user interface component library. This library also contains the key components for integrating JSF into a JSP. Table 4-5 lists the components that are supported.

This book is not designed to teach the reader every aspect of JSF; it is focused on showing how JSF can be used with Application Developer. Therefore we do not provide a full detailed description of each component. You can reference the JSR-127 specification for more details:

<http://www.jcp.org/en/jsr/detail?id=127>

You can see from this list of core components that they can be used to change the behavior of some of the key HTML render kit components.

Table 4-5 List the Core components supported by JSF

Tag Name	Description
<f:actionListener>	The tag creates an instance of a class that must subclass javax.faces.ActionListener interface. It can be use to customize which class events are sent to be handled. <pre><f:actionListener type="com.ibm.itso.EventListener" /></pre>
<f:attribute>	This tag sets a generic attribute for a parent component. It can be used to tailor custom values for a component. <pre><f:attribute name="age" value = "36"/></pre>
<f:convertDateTime>	This tag invokes a date converter and can be used to convert an input value into a specific date format or style. <pre><h:inputText value="#{order.date}"> <f:convertDateTime dateStyle="full"/> </h:inputText></pre>
<f:convertNumber>	This tag invokes a number converter and can be used to convert an input value into an integer, floating point, currency, or according to a mask. <pre><h:inputText value="#{order.ordernumber}"> <f:convertNumber pattern="###000"/> </h:inputText></pre>
<f:converter>	This tag invokes a custom converter and takes the id of a converter that is registered in the face-config.xml file. <pre><f:converter converterId="NumberToPONumber"/></pre>
<f:facet>	This tag allows a set of JSF components to be children of a facet parent with a specific name. This allows a define set of actions to be applied to all of the components. <pre><f:facet name="PageNumber"> <h:outputText value="#{page.num}"/> </f:facet></pre>

Tag Name	Description
<f:param>	This tag allows a parameter to be set on a parent JSF component. This can be a static value or bound at runtime. This tag is most often used to pass parameters along with a hyperlink or command. <pre> <h:inputText id="text1"> <f:param value="Hello"/> </h:inputText> </pre>
<f:selectItem>	This tag works in conjunction with a JSF components that allows items that can be selected through the user interface. It adds an itemValue and itemName to a component such as a <h:selectManyList>. <pre> <h:selectManyListbox value="#{compy.regions}"/> <f:selectItem itemName="North" itemValue="N" /> <f:selectItem itemName="East" itemValue="E" /> </h:selectManyListbox> </pre>
<f:selectItems>	This tag adds items to a selectable style component using a bind by value reference. The reference can be of type java.util.ArrayList and must contain SelectOne objects. <pre> <f:selectItems value="#{regions.listofRegions}" /> </pre>
<f:subView>	This is a container action. This action is used on a nested page through a <jsp:include> tag or any custom action that dynamically includes another page from the same Web application. <pre> <f:subview id="myNestedPage"> <jsp:include page="theNestedPage.jsp"/> </f:subview> </pre>
<f:validateDoubleRange>	This tag generates an error message if the input value is not within its range for a double value. <pre> <h:inputText value="#{order.value}"> <f:convertNumber type="currency"/> <f:validateDoubleRange minimum="0.0" maximum="999.99"/> </h:inputText> </pre>

Tag Name	Description
<f:validateLength>	This tag forces an input component to allow for a minimum and maximum number of characters. <pre> <h:inputText value="#{customer.age}"> <f:convertNumber type="number"/> <f:validateLength minimum="0" maximum="3"/> </h:inputText> </pre>
<f:validateLongRange>	This tag forces an input component to set a range (minimum and maximum) for a long value. <pre> <h:inputText value="#{house.value}"> <f:convertNumber type="number"/> <f:validateLongRange minimum="150000" maximum="500000"/> </h:inputText> </pre>
<f:validator>	This tag invokes a custom validator for an input component. The validator has to be registered in the faces-config.xml file. <pre> <h:inputText value="#{order.ordernum}"> <f:convertNumber type="number"/> <f:validator validatorId="maxorderable"/> </h:inputText> </pre>
<f:valueChangeListener>	This tag triggers an event when a value has changed in an input component. The custom class has to implement javax.faces.event.ValueChangeListener. <pre> <h:selectBooleanCheckBox value="Accept" immediate="true"> <f:valueChangeListener type="com.ibm.itso.AcceptAgreement" </h:selectBooleanCheckBox> </pre>
<f:verbatim>	This tag can be used to display HTML or XML style markup within a browser. It replaces the <> brackets with &lt; type characters. <pre> <f:verbatim escape="true"> Google </f:verbatim> </pre>
<f:view>	This tag creates a parent component that can hold children JSF components. It can have its locale changed and all children are asked to render using another language. <pre> <f:view locale="#{login.locale}"> ... </f:view> </pre>

HTML library

JavaServer Faces contains a set of components that support the standard HTML components used to develop Web applications. The most common form of a J2EE application is the Web application using servlets, JSPs, and tag libraries. The HTML library provides common HTML components with server-side interactions that are easily managed with the JSF framework. When you combine these controls with page data and managed beans you get a powerful environment that allows complex UI interactions to be created in easy steps.

The Palette view of the Page Designer displays all the components that are available with WebSphere Studio. Table 4-6 lists the components of the core HTML library (`<h:>`). We separated the IBM extensions into a different section to emphasize their added value above the standard set defined in JSR-127.

Table 4-6 Standard HTML components on the Palette

Palette Name	Tag Name	Base Type
 Form	<code><h:form></code>	UICommand
 Label	<code><h:outputLabel></code>	UIOutput
 Command - Hyperlink	<code><h:commandLink></code>	UICommand
 Output	<code><h:outputText></code>	UIOutput
 Output - Formatted Text	<code><h:outputFormat></code>	UIOutput
 Input	<code><h:inputText></code>	UIInput
 Input - Text Area	<code><h:inputTextArea></code>	UIInput
 Input - Password	<code><h:inputSecret></code>	UIInput
 Input - Hidden	<code><h:inputHidden></code>	UIInput
 Display Error	<code><h:messages></code>	UIMessage
 Display Errors	<code><h:messages></code>	UIMessages
 Radio Button Group	<code><h:selectOneRadio></code>	UISelectOne
 Check Box	<code><h:selectBooleanCheckBox></code>	UISelectBoolean
 Check Box Group	<code><h:selectManyCheckBox></code>	UISelectMany
 Combo Box	<code><h:selectOneMenu></code>	UISelectOne
 List Box - Single Select	<code><h:selectOneListbox></code>	UISelectOne
 List Box - Multiple Select	<code><h:selectManyListbox></code>	UISelectMany

Palette Name	Tag Name	Base Type
 Data Table	<h:dataTable>	UIData

Note: Two standard components are missing from the Table 4-6: **Command - Button** and **Image**. These two components are listed under the IBM extension library, where they exist with additional functionality.

Most of the components listed in Table 4-6 relate to an equivalent HTML component. A user interface designer will soon be able to understand how to use these components in a page. The two new components that are not related to HTML are the Display Error(s) components. These components are used often in the detailed examples contained in this redbook.

Form: <h:form>

The `h:form` component is used to act as a container for all form based JSF components. It renders itself as a `<FORM>` element with the action attribute set to the Faces servlet, and all the form data is passed to the server. Only JSF components that are children of the form are included in the form processing.

Attributes

Table 4-7 shows the attributes of the `<h:form>` component.

Table 4-7 Command Link attributes

Attributes	Description
<i>common</i>	id, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, target, title
accept	List of content types that a server processing this form will handle correctly
acceptcharset	List of character encodings for input data that are accepted by the server processing this form.
enctype	Content type used to submit the form to the server. If not specified, the default value is "application/x-www-form-urlencoded".

Events

Table 4-8 shows the events of the `<h:form>` component.

Table 4-8 Command Link Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Usage

This example shows how a form is incorporated into a page:

- Create a Faces JSP file called **form.jsp**.
- From the palette drag an Input component and a Command - Button (Figure 4-10). Change the text to say **Sample Form Faces Page**.

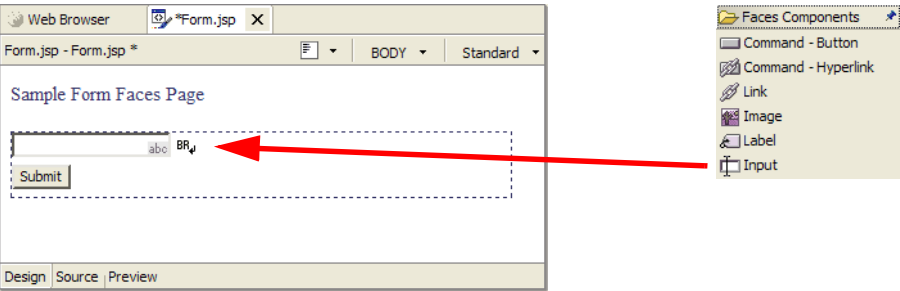


Figure 4-10 Example of a drag operation

- By using the drag operations the user sees that `<f:form>` has been automatically added to the page and placed around the JSP components that participate in the form processing. Example 4-1 shows all the code that is generated by creating a Faces page and adding an Input component and a Command - Button.

Example 4-1 Example form page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%-- jsf:codeBehind language="java" location="/JavaSource/pagecode/Form.java"
--%><%-- /jsf:codeBehind --%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://www.ibm.com/jsf/html_extended" prefix="hx"%>
<HTML>
<HEAD>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ page language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
%>
```

```

<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM WebSphere Studio">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet"
      type="text/css">
<TITLE>form.jsp</TITLE>
<LINK rel="stylesheet" type="text/css" href="theme/stylesheet.css"
      title="Style">
</HEAD>
<f:view>
  <BODY><hx:scriptCollector id="scriptCollector1">
    <P>Sample Form Faces Page</P>
    <h:form styleClass="form" id="form1">
      <h:inputText styleClass="inputText" id="text1"></h:inputText>
      <BR>
      <hx:commandExButton type="submit" value="Submit"
        styleClass="commandExButton" id="button1"></hx:commandExButton>
    </h:form>
  </hx:scriptCollector></BODY>
</f:view>
</HTML>

```

- ▶ If you run this page you notice that the data entered into the input field is accessible by the page code class.

How it works

For the more experienced developers we examine what is generated when the JSF components are added to the page.

- ▶ When you drag a component that requires a `<h:form>` as a parent into a JSF page, the `<h:form>` element is automatically added to the page. All other components that are dragged onto the page design are placed inside the `<h:form>` `</h:form>` tags.
- ▶ If you want to create a second form in a page, you first have to make the Form component visible in the Palette. Select *Customize* in the context menu of the Faces component palette section (Figure 4-11).

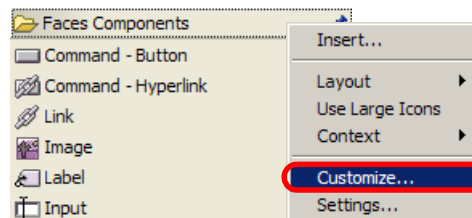


Figure 4-11 How to open the customize dialog for the palette section

- Once the customize dialog appears (Figure 4-12), scroll down until you see the Faces components sections. The Form component is grayed out. Deselect *Hide* to make the Form component visible in the palette.

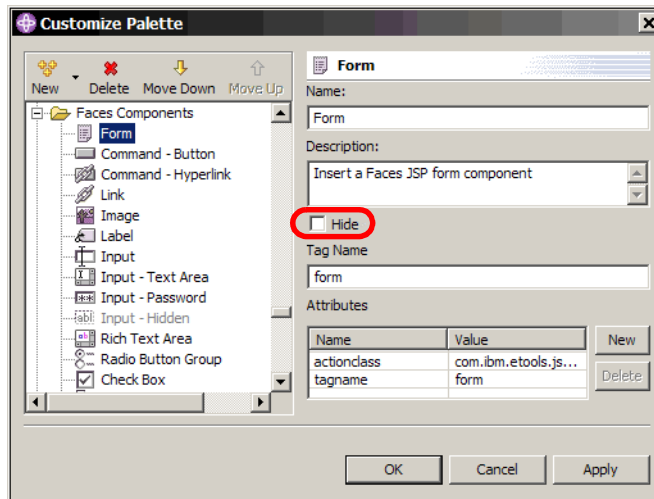


Figure 4-12 Customize Palette

- Now you can drag the Form element into a JSP and an area is displayed. You can then drag other components into this area (Figure 4-13). The first form was automatically created by dragging the first component that requires a form into the page. The second involved dragging a Form component into the page and then adding components to it.

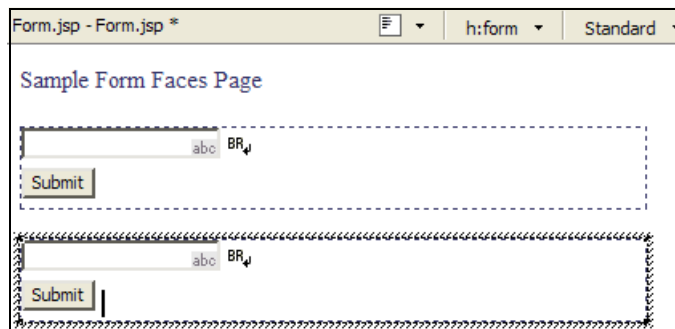


Figure 4-13 Creating two forms in the same page

Although the Form component is invisible to most page designers it is an important component that holds a form page together. WebSphere Studio can automatically control the insertion, but as you have see this component can be

added more than once to a page to allow different form handling events to be triggered.

Output: <h:outputText>

This component allows displays a text or a value.

Component type

UIOutput The UIOutput component displays data that cannot be modified by the user. The most common example for this type of component is a label.

Attributes

Table 4-9 shows the attributes of the <h:outputText> component.

Table 4-9 Attributes for the Output component

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, binding, title
converter	Converter instance registered with this component.
escape	This flag indicates if the characters that are sensitive in HTML and XML markup must be escaped. This flag is set to true by default.

Usage

The usage of the component is basic. To use it, you only have to drag and drop the Output component from the palette.

The value of this component can be static or can be bound to a defined variable or bean in one of the visible scopes for the page containing the *Output* component. It also can be bound to a property value through a resource bundle. This feature is very important in environments where multiple languages are supported in a Web application.

How it works

When the page is rendered to the client, the HTML code for this component is generated from the value.

Output - Formatted Text: <h:outputFormat>

This component allows to display a set of concatenated messages using a formatted MessageFormat pattern, as described in the Java API documentation for the class `java.text.MessageFormat`.

Component type

UIOutput The UIOutput component displays data that cannot be modified. The most common example for this type of component is a label.

Attributes

Table 4-10 shows the attributes for the `<h:outputFormat>` component.

Table 4-10 Attributes for the Output - Formatted component

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, binding, title
converter	Converter instance registered with this component.
escape	This flag indicates if the characters that are sensitive in HTML and XML markup must be escaped. This flag is set to true by default.

Usage

Here is an example of how to use the Output - Formatted Text component to display a simple formatted concatenated message:

- ▶ Create a JSF page called **formattedOutput.jsp** and change the text to **Sample of Output-Formatted Text Usage**. Drag an Output - Formatted Text component from the palette onto the page under this text.
- ▶ Select the new component and in the Attributes view change the `id` property to **format** and the `value` property to **My name is {0} and my age is {1}**.
- ▶ Select the Parameters tab and add two parameters. Enter **name** and **age** for their names. Enter **John Smith** and **23** as their values (Figure 4-14).

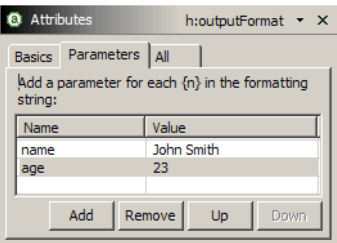


Figure 4-14 Defining parameters for the Output - Formatted Text component

- ▶ Save and test the page. You can see the formatted message in your page.

How it works

You can see the code that is generated for this example:

```
<h:outputFormat styleClass="outputFormat" id="format"
    value="My name is {0} and my age is {1}">
    <f:param id="param1" name="name" value="John Smith"></f:param>
    <f:param id="param2" name="age" value="23"></f:param>
</h:outputFormat>
```

When the page is rendered to the client, the numbers in braces are substituted by the values of the parameters in the order that they were defined. That is, number {0} is substituted by the first parameter we defined, param1, and number {1} is substituted by the value of param2.

You can bind the values of the parameter to any defined variable or bean in any of the scopes that are visible from the JSP. You can also bind the values of the parameters to the values of properties defined in a resource bundle.

Because the message is formatted using the MessageFormat pattern, you can use all the functionality this class is providing to enrich the end user Web experience. For more information refer to the Java API documentation for the class `java.text.MessageFormat`.

Input: <h:inputText>

The Input component allows you to create an input field for text input. The base implementation from JSR-127 only support String or simple text. WebSphere Studio has extended the support to these data types:

- ▶ Number—a number field for which you can set the style and format
- ▶ Date/Time—a date and time field for which you can set the format.
- ▶ Mask—a filter that selectively includes or excludes certain values.

When you add an input component to a page, choose one of the these input types on the Format tab of the Attributes view.

Note: These extra data types are extensions to the base HTML component from JSR-127. IBM has added these types because they are common data input types required by Web applications.

Any Web application that requires the collection of information through user input uses this component. Using the input types enables tailoring the input form design and have a degree of validation to be completed. This removes the requirement for separate validation and conversion code to be written.

Component type

UIInput

The UIInput component displays data that can be modified by the user. The most common example for this type of component is an input field in a form.

Attributes

Table 4-11 shows the attributes for the `<h:input>` component.

Table 4-11 Attributes for the Input component

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
alt	Alternate textual description of the element rendered by this component.
converter	Specifies the data type to convert the value to. The JavaServer Faces implementation provides a set of Converter implementations that you can use to convert your component data to a type not supported by its renderer.
maxLength	Specifies the maximum length of characters the input component can allow.
size	Specifies the initial width of the component. The width value refers to the number of characters that are displayed.
readonly	Specifies that the value of the component cannot be changed by the user.
required	This flag indicates that the user is required to provide a value for this input component. This is verified by a specialized validator.
tabindex	Position of this element in the tabbing order for the current document.
validator	Class name of validator to instantiate.
valueChangeListener	The value binding name of a method to handle events when a value has changed within the input field.

Events

Table 4-12 shows the events of the `<h:input>` component.

Table 4-12 Input Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
Value Changed	Adding code to this event causes a method to be generated in the page code class and a binding reference placed in the valueChangeListener attribute for this component.
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when the input field is selected.

Usage

This example shows how a valueChangeListener can display the old and new values for a field:

- ▶ Create a JSF page called **input.jsp** and change the text to **Sample Input Page**. Drag an Input component from the palette onto the page below this text. To help display the old value drag an Output component below the input.
- ▶ Through the Attributes view change the id of the Input component to **name** and the id of the Output component to **text**.
- ▶ Select the Input component and open the Quick Edit view.
- ▶ Select the *Value Change* event and enter the following code into the snippet area:

```
Object obj = valueChangedEvent.getOldValue();
getText().setValue(obj);
```
- ▶ This code takes the old value from the input field and sets the value of the output text.
- ▶ The last step is to trigger an event. Drag a Command - Button and place it below the Input and Output components. We do not add any logic for the button. When it is clicked it automatically forces all the valueChangeListener to be execute in the page code class.
- ▶ Save the page and run it using the WebSphere test environment.
- ▶ If you enter the first name and click *Submit* nothing is displayed. If you then enter the second name the output text changes to the first name.

How it works

For the more experienced developers we can examine how these events are controlled by the JSF framework:

- In the Source tab of the Page Designer we can see the tag of the Input component:

```
<h:inputText styleClass="inputText" id="name"
  valueChangeListener="#{pc_Input.handleNameValueChange}">
</h:inputText>
```

- The valueChangeListener event is mapped to the handleNameValueChange method in the page code.
- Select *View Page Source*. Scroll down the event handler code:

```
public void handleNameValueChange(ValueChangeEvent valueChangedEvent) {
    // Type Java code to handle value changed event here
    // Note, valueChangeEvent contains new and old values
    Object obj = valueChangedEvent.getOldValue();
    getText().setValue(obj);
}
```

The input component is widely used in JSF applications and can be combined with validators and converters to add powerful form processing with JSF.

Input - Password: <h:inputSecret>

This component is rendered as an HTML input component with the type attribute set to password. It allows a secret value to be entered and display stars as the user types the password.

Component type

UIInput

This class type represents an input component. Values can be entered by the user and when the values change a valueChangeEvent is triggered.

Attributes

Table 4-13 shows the attributes for the <h:inputSecret> component.

Table 4-13 Input - Password attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
valueChangeListener	The value binding name of a method to handle events when a value has changed within the input field.
validator	Class name of validator to instantiate.
size	Specifies the initial width of the component. The width value refers to the number of characters that are displayed.

maxLength	Specifies the maximum length of characters the input component can allow.
converter	Specifies the data type to convert the value to. The JavaServer Faces implementation provides a set of Converter implementations that you can use to convert your component data to a type not supported by its renderer.
readonly	Specifies that the value of the component cannot be changed from its initial value.
alt	Alternate description for the component.
redisplay	This flag indicates that the value in this field should be rendered when the form is created. Because of the potential security risk, password fields are not displayed by default.
required	This flag indicates that the user is required to provide a submitted value for this input component.
size	The number of characters used to determine the width of the field.
tabindex	The component position in the tab order of the page.

Events

Table 4-14 shows the events of the `<h:inputSecret>` component.

Table 4-14 Input Secret Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
Value Change	Adding code to this event causes a method to be generated in the page code class and a binding reference placed in the valueChangeListener attribute for this component.
onselect	JavaScript code executed when the input field is selected
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.

Usage

The `<h:inputSecret>` tag can be used for representing a password field in the application. To create a simple example follow these steps:

- ▶ Create a Faces JSP file and name it **inputSecret.jsp**. Replace the default text by **Example of Input Secret**.
- ▶ Insert an HTML table with two rows and two columns. Enter **Enter your user id:** in the first column of the first row. Enter **Enter your password:** in the first column of the second row.
- ▶ Drag and drop an Input component into the second column of the first row. Drag and drop an Input - Password component in the second column of the second row.
- ▶ Drag and drop a Command - Button component under the table.
- ▶ Add a new line and enter **User Id:**. Add an Output component.
- ▶ Add a new line and enter **Password:**. Add an Output component.
- ▶ Select the Input component and, using the Attributes view change its id property to **usr**. Change the width property to **10**.
- ▶ Select the Input - Password component and change its id property to **pass** and its width property to **10**.
- ▶ Select the first Output component and set its name to **typedUserId**.
- ▶ Select the second Output component and set its name to **typedPassword**.
- ▶ Using the Page Data view, define two new variables in the session scope:
 - **userid** of type `java.lang.String`.
 - **password** of type `java.lang.String`.
- ▶ Bind the `userid` session variable to the Input component and to the first Output component.
- ▶ Bind the `password` session variable to the Input - Password component and to the second Output component (Figure 4-15).
- ▶ Save and test the page.

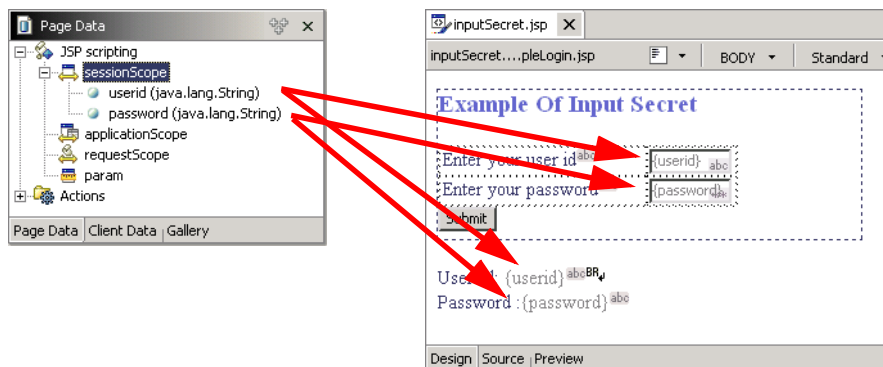


Figure 4-15 Design of the Input - Password component example

How it works

This component works the same way as the Input component but using the asterisks to hide the value that you are typing.

Another important feature of the Input - Password component is the `redisplay` property (you can find it in the All tab of the Attributes view). If this attribute is `false` (the default), the value is not displayed in the field when the server renders the response to the client.

You can test this `redisplay` property by setting it to `true` and see what happens. When you submit the values to the server, when the page is rendered again, you can see that the value you entered is still in the field (represented as asterisks).

Input - Hidden: <h:inputHidden>

The Input - Hidden component is used to pass data to the next JSF page using a hidden field in an HTML form.

Component type

UIInput

The `UIInput` component displays data that can be modified by the user.

Attributes

Table 4-15 shows the attributes for the `<h:input>` component.

Table 4-15 Attributes for the Input -Hidden component

Attributes	Description
<i>common</i>	<code>id</code> , <code>value</code> , <code>rendered</code> , <code>immediate</code> , <code>binding</code>
<code>converter</code>	Specifies the data type to convert the value to. The JavaServer Faces implementation provides a set of Converter implementations that you can use to convert your component data to a type not supported by its renderer.
<code>required</code>	This flag indicates that the user is required to provide a value for this input component. This is verified by a specialized validator.
<code>validator</code>	Class name of validator to instantiate.
<code>valueChangeListener</code>	The value binding name of a method to handle events when a value has changed within the input field.

Events

Table 4-16 shows the events of the `<h:input>` component.

Table 4-16 Input - Hidden Quick Edit events

Event	Description
Value Changed	Adding code to this event causes a method to be generated in the page code class and a binding reference placed in the <code>valueChangeListener</code> attribute for this component.

Usage

The Input - Hidden component is typically used to pass a string value from one JSF page to another. This is one way to pass a request block value of one page to the next page.

Input - Text Area: `<h:inputTextArea>`

This component is rendered as Text Area HTML component. It allows multiple lines of text to be entered.

Component type

UIInput This class type represents an input component. Values can be entered by the user and when the values change a `valueChangeEvent` is triggered.

Attributes

Table 4-17 shows the attributes of the `<h:inputTextArea>` component.

Table 4-17 Input - Text Area attributes

Attributes	Description
<i>common</i>	<code>id</code> , <code>value</code> , <code>style</code> , <code>styleClass</code> , <code>rendered</code> , <code>immediate</code> , <code>binding</code> , <code>accesskey</code> , <code>dir</code> , <code>disabled</code> , <code>lang</code> , <code>title</code>
<code>cols</code>	The number of columns to be displayed.
<code>converter</code>	Specifies the data type to convert the value to. The JavaServer Faces implementation provides a set of Converter implementations that you can use to convert your component data to a type not supported by its renderer.
<code>maxLength</code>	Specifies the maximum length of characters the input component can allow.
<code>readonly</code>	Specifies that the value of the component cannot be changed from its initial value.
<code>required</code>	This flag indicates that the user is required to provide a value for this input component.

rows	The number of rows to be displayed.
size	Specifies the initial width of the component. The width value refers to the number of characters that are displayed.
validator	Class name of validator to instantiate.
valueChangeListener	The value binding name of a method to handle events when a value has changed within the input field.

Events

Table 4-18 shows the attributes for the <h:inputTextArea>.

Table 4-18 Input - Text Area Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
Value Change	Adding code to this event causes a method to be generated in the page code class and a binding reference placed in the valueChangeListener attribute for this component.
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when the input field is selected

Usage

The <h:inputTextArea> tag can be used for representing Text Area input fields in the application. Here are the instructions for an example:

- ▶ Create a JSF page and name it **textArea.jsp**. Change the text to **Usage of Text Area Sample**. Drag an Input - Text Area component into the JSP.
- ▶ Drag an Output component below the Text Area component. Finally drag a Command - Button next to the Text Area component. Using the attribute view change the id of the Text Area to **textarea** and the id of the Output Text to **output**.
- ▶ Select the Submit button and in the Quick Edit view select the Command event and add this code into the snippet area:

```

getOutput().setValue(getTextarea().getValue());
return "";

```
- ▶ Save the page and run it within the WebSphere test environment (Figure 4-16). You notice that when you enter some text into the text area and then click *Submit* the text is displayed in the output component.

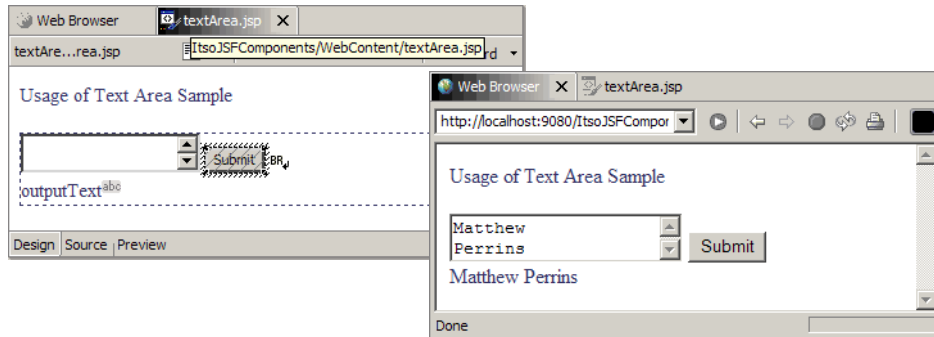


Figure 4-16 Example of Text Area usage

How it works

For the more experienced developers we can examine how these events are controlled by the JSF framework:

- In the Source view you can see the tag as it is created for the component:

```
<h:inputTextarea
    styleClass="inputTextarea"
    id="textarea">
</h:inputTextarea>
```

- To open the page code select *View Page Code*. Look for the action logic:

```
public String doButton1Action() {
    // Type Java code to handle command event here
    // Note, this code must return an object of type String (or null)
    getOutput().setValue(getTextarea().getValue());
    return "";
}
```

- This code simply gets a handle to the Output JSF component and sets its output value using the value from the Text Area component.

The Input - Text Area component is very useful input control for handling data that is more than one line in length. You can also apply validators to this component.

Display Error: <h:message>

The Display Error component is used to show the error associated with another component in the page.

Component type

UIMessage

The UIMessage component is used to display error messages to the user. This message is bound to an UIComponent to display the error messages associated with that component.

Attributes

Table 4-19 shows the attributes of the <h:message> component.

Table 4-19 Attributes for the Display Message component

Attributes	Description
common	id, value, style, styleClass, rendered, immediate, binding, title
for	Client identifier of the component for which to display messages.
showDetail	This flag indicates whether the detail portion of displayed messages should be included. Default value is true.
showSummary	This flag indicates whether the summary portion of displayed messages should be included. Default is false.
errorClass	CSS style class to apply to messages with severity ERROR.
errorStyle	CSS style(s) to apply to messages with severity ERROR.
fatalClass	CSS style class to apply to messages with severity FATAL.
fatalStyle	CSS style(s) to apply to messages with severity FATAL.
infoClass	CSS style class to apply to messages with severity INFO.
infoStyle	CSS style(s) to apply to messages with severity INFO.
tooltip	Flag indicating whether the detail portion of the message should be displayed as a tooltip.
warnClass	CSS style class to apply to messages with severity WARN.
warnStyle	CSS style(s) to apply to messages with severity WARN.

Usage

This example shows how to use the Display Message component to display the error messages for an Input component:

- ▶ Create a JSF page called **errorMessage.jsp** and change the text to **Sample Error Page**. Drag an Input component and a Command - Button component from the palette onto the page below this text.
- ▶ Through the Attributes view change the **id** property of the Input to **text**. Change the **id** property of the Submit button to **submit**.
- ▶ Select the Input component and select the Validation tab in the Attributes view. Select *Value is required* and *Add Display Error Control*. An Error Message component appears near the Input component.
- ▶ Save and test the page. When the page appears, click *Submit* without entering any value in the Input field and an error message appears.

Now we add a second Error Message component for the same Input component:

- ▶ Return to the design of the **errorMessage.jsp** page. Drag and drop a second Display Error component from the palette under the Submit button (Figure 4-17). You get an error because the message tag is incomplete.
- ▶ Select the last Display Error component and bind it to the Input field (in the Attributes view, select *text* in the For validation errors on: combo box).
- ▶ Save the page and test. You get the same error message twice, one time near the Input field and another under the Submit button.

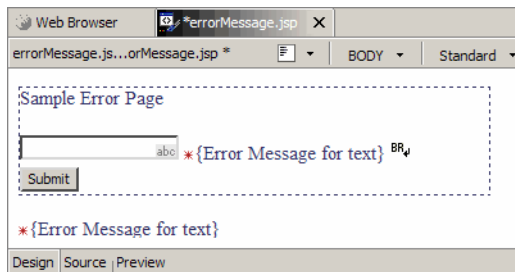


Figure 4-17 Error Message components

How it works

The validation errors are processed in the validation phase of the JSF standard request-response life cycle (see Figure 1-5 on page 10). When the validation error for the Input field occur, JSF renders the response to the client.

The **showSummary** and **showDetail** properties of the **UIMessage** component indicate which message of the error message (**FacesMessage**) to display. The severity style of the message is used to apply CSS styles or classes.

Display Errors: <h:messages>

The Display Errors component is the same as the Display Error, but it displays all the messages of the page.

Component type

UIMessages

The UIMessages component is used to display all the error messages to the user.

Attributes

Table 4-20 shows the attributes for the <h:messages> component.

Table 4-20 Attributes for the Display Messages component

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, title
globalOnly	Flag indicating that only global messages (that is, messages not associated with any client identifier) are to be displayed. Default value is false.
showDetail	This flag indicates whether the detail portion of displayed messages should be included. Default value is false.
showSummary	This flag indicates whether the summary portion of displayed messages should be included. Default is true.
errorClass	CSS style class to apply to messages with severity ERROR.
errorStyle	CSS style(s) to apply to messages with severity ERROR.
fatalClass	CSS style class to apply to messages with severity FATAL.
fatalStyle	CSS style(s) to apply to messages with severity FATAL.
infoClass	CSS style class to apply to messages with severity INFO.
infoStyle	CSS style(s) to apply to messages with severity INFO.
layout	The type of layout markup to use when rendering error messages. Valid values are <i>table</i> (an HTML table) and <i>list</i> (an HTML list). If not specified, the default value is <i>list</i> .
tooltip	Flag indicating whether the detail portion of the message should be displayed as a tooltip.
warnClass	CSS style class to apply to messages with severity WARN.
warnStyle	CSS style(s) to apply to messages with severity WARN.

Usage

This example shows how to use the Display Messages component to display several error messages at the same time:

- Create a JSF page called **errorMessages.jsp** and change the text to **Sample Errors Page**. Drag two Input components and a Command - Button component from the palette onto the page below this text (Figure 4-18).

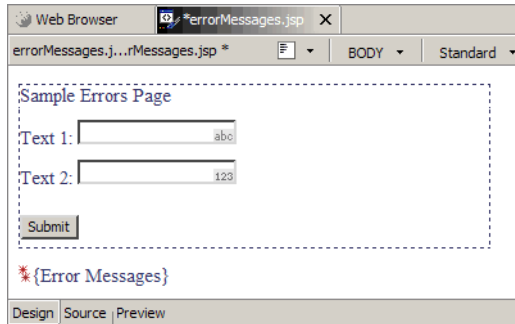


Figure 4-18 Error Messages component

- Through the Attributes view change the `id` property of the first Input component to **text**. In the Validation tab, select *Value is required*, set the minimum length to **1** and the maximum length to **2**. In the constraint combo box, select *Alphabetical values only*.
- Select the second Input component and the `id` property **text2**. In the Format tab, select *Number* in the Type combo box. In the validation tab, select *Value is required*, set the minimum to **10** and the maximum **100**.
- Change the `id` property of the Submit button to **submit**.
- Now drag a Display Messages component below the Submit button. Select it and in the Attributes view select the All tab. Change the layout property to **table** (Figure 4-19).
- Save and test the page. Type valid and invalid values into the two Input components and see what errors are displayed in the Display Messages component.

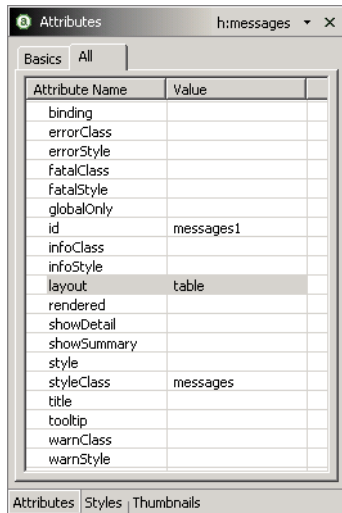


Figure 4-19 Changing the Display Messages layout property

How does it work

This component is the same as the Display Error component, but it displays all the messages of all the components that are having errors. If the value of the layout attribute is table, it renders a nested table, tr, and td elements in that order. Otherwise, it does not render messages in a table.

If the globalOnly property is true, the error messages for individual components are not shown.

Label: <h:outputLabel>

This component renders an HTML Label element. Find the component specified by the value of the for attribute, and render its client id as the value of the for attribute.

Component type

UIOutput

The UIOutput component displays data that cannot be modified by the user. The most common example for this type of component is a label.

Attributes

Table 4-21 shows the attributes of the <h:outputLabel> component.

Table 4-21 Attributes for the Label component

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
converter	Specifies the data type to convert the value to. The JavaServer Faces implementation provides a set of converter implementations that you can use to convert your component data to a type not supported by its renderer.
for	Client identifier of the component for which this element is a label.
tabindex	Position of this element in the tabbing order for the current document.

Events

Table 4-22 shows the event of the `<h:outputLabel>` component.

Table 4-22 Input Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Usage

The `<h:outputLabel>` tag is used to attach a label to a specified input field for accessibility purposes.

The `for` attribute of the `<h:outputLabel>` tag maps to the `id` of the component to which the label is attached. The `outputText` tag nested inside the `outputLabel` tag represents the actual label component. The `value` attribute of the `outputText` tag is the label that is displayed.

Example:

```
<h:outputLabel styleClass="outputLabel" id="label1" for="text2">
  <h:outputText id="text3" value="Amount"></h:outputText>
</h:outputLabel>
<h:inputText styleClass="inputText" id="text2"></h:inputText>
```

Command - Hyperlink: <h:commandLink>

The Command - Hyperlink component acts like a hyperlink in HTML. Clicking the link triggers an event that can be captured in a command action method. You can then create a response to this event. This component is typically used to assist with the navigation of an application.

Component type

UICommand

The UICommand component allows an event to be triggered. Example of this control are the Command - Button and Command - Hyperlink components.

Attributes

Table 4-23 shows the attributes of the <h:commandLink> component.

Table 4-23 Command - Hyperlink attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
actionListener	The name of the class that responds to actions. This is only relevant for components that can have actions, such as buttons and links.
action	Specifies the bound method that is executed when this component is selected. The page code class contain a method and is referred to in the following notation: #{pc_SampleClass.doLink1Action}. The first part refers to the page code class and the second part is the method that is executed.
target	Specifies the title text, shown in browser as a tooltip at runtime.
charset	The character encoding of the resource designated by this hyperlink.
coord	The position and shape of the hot spot on the screen (for use in client-side image maps).
hreflang	The language code of the resource designated by this hyperlink.
lang	Specifies the base language of the component's attribute values and text content. Takes ISO-standard language abbreviation codes (for example, "en" for English, "en-US" for American English, "fr" for French, "de" for German).

Attributes	Description
rel	The relationship from the current document to the anchor specified by this hyperlink. The value of this attribute is a space-separated list of link types.
rev	A reverse link from the anchor specified by this hyperlink to the current document. The value of this attribute is a space-separated list of link types.
type	Specifies the button type, either <i>Submit</i> or <i>Reset</i> .

Events

Table 4-24 shows the events of the `<h:commandLink>` component.

Table 4-24 Command - Hyperlink Quick Edit events

Event	Description
<i>common</i>	onblur, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
Command	Adding code to this event causes a method to be generated on the page code class and a binding reference placed in the action attribute for this component.

Usage

This example shows how a Command - Hyperlink component can be used to change the text in an Output component:

- Create a Faces JSP file called **commandLink.jsp**.
- You can place a Command - Hyperlink component onto a JSF page by dragging it from the Palette into the page (Figure 4-20).

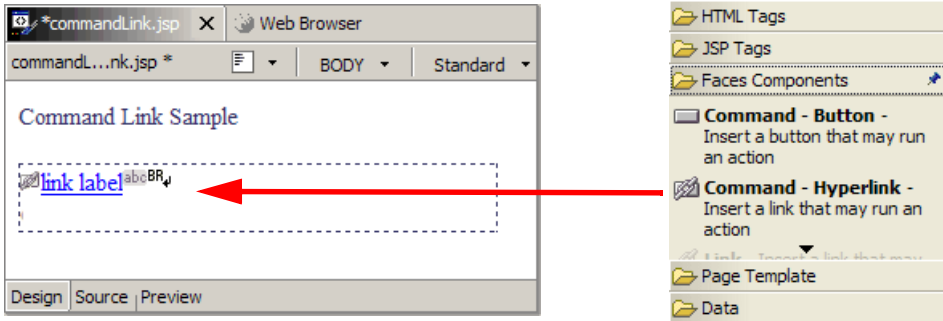


Figure 4-20 Example of a drag operation

- ▶ The control is represented by two visual items on the page. The first icon relates to the link control and the `link_label` is defined with an Output (`<h:outputText>`) control. You have to select them individually to modify their respective attributes.
- ▶ Select the `link_label` text and it becomes highlighted.
- ▶ Within the attributes view change the value field to **How are you?**
- ▶ Drag an Output control onto the page below the Command - Hyperlink. Using the Attributes view change the `id` to **message**.
- ▶ Select the  icon in the page and select *Quick Edit* (context). The Quick Edit view is displayed and you can see a list of actions and JavaScript events that can be added to the control (Figure 4-21).

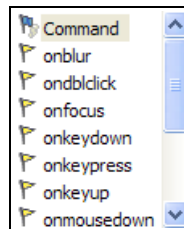


Figure 4-21 List of events for a control in the Quick Edit view

- ▶ Select the *Command* action, then click in the right pane. Studio automatically adds an action event and leaves the editor ready to enter a small amount of code:

```
getMessage().setValue("I am fine, how are you?");
return "";
```

This code snippet sets the value attribute of the Output component.

- ▶ Save the page and test. When you click on the link, the text changes.

How it works

For the more experienced developers we can examine how these events are controlled by the JSF framework.

- ▶ Select the Command - Hyperlink control. We can examine the JSF tags that have been automatically generated after we dropped the control and how their values have changed through the Attribute view.

```
<h:commandLink styleClass="commandLink" id="link1"
    action="#{pc_CommandLink.doLink1Action}">
    <h:outputText id="text1" styleClass="outputText"
        value="How are you?"></h:outputText>
</h:commandLink>
<h:outputText styleClass="outputText" id="message"></h:outputText>
```


- ▶ In the Page Data view you can see a new action method called `doLink1Action` has been defined. You can reuse these events and drag them onto other controls, if you have a common piece of code that has to be invoked from multiple places on the page. The drag operation defines the action attribute for the control the action has been dropped on.
- ▶ In the Attributes view **All** tab you can see that the action attribute has been bound to the action event method:

```
# {pc_CommandLink.doLink1Action}
```

- ▶ Open the page code by selecting *View Page Code* (context). This displays the Java code that is controlling all the events of the page. You can see the code we added above:

```
public String doLink1Action() {
    // Type Java code to handle command event here
    // Note, this code must return an object of type String (or null)
    getMessage().setValue("I am fine, how are you?");
    return "";
}
```

- ▶ The `getMessage` method is a simple getter that has been generated to access the message `Output` component:

```
protected HtmlOutputText getMessage() {
    if (message == null) {
        message = (HtmlOutputText) findComponentInRoot("message");
    }
    return message;
}
```

Note: In most cases you want to pass data to the action method invoked by the hyperlink. This is done by defining parameters in the Attributes view Parameters tab of the hyperlink.

The Command - Hyperlink control offers useful features for managing the interaction with the JSF controller. You will see in the banking example (Chapter 5, “JSF banking application” on page 191) how this control can be used in a table generated from data to link to a new JSF page.

Radio Button Group: <h:selectOneRadio>

This component represents a radio button group. You have a list of possible selections and you can select only one.

Component type

UISelectOne

The `UISelectOne` component allows you to select one value from a set of values.

Attributes

Table 4-25 shows the attributes of the `<h:selectBooleanCheckBox>` component.

Table 4-25 Radio Button Group attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, target, title
border	This attribute represents the width (in pixels) of the border to be drawn around the table containing the options list.
converter	Converter instance registered with this component.
layout	Orientation of the options list to be created. Valid values are <code>pageDirection</code> (list is laid out vertically), or <code>lineDirection</code> (list is laid out horizontally). If not specified, the default value is <code>lineDirection</code> .
required	This flag indicates that the user is required to provide a value for this input component.
readonly	Flag indicating that this component prohibits changes by the user. The element may receive focus unless it has also been disabled.
tabindex	Position of this element in the tabbing order for the current document. This value must be an integer between 0 and 32767.

Events

Table 4-26 shows the events for the `<h:selectBooleanCheckBox>` component.

Table 4-26 Radio Button Group Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when text within this element is selected by the user.

Usage

The `<h:selectOneRadio>` tag displays a list of choices, where you can only select one of the choices. This example shows how to use the `<h:selectOneRadio>` tag:

- ▶ Create a JSF JSP page and name it **radioButtonGroup.jsp**. Replace the default text and enter **Example of Radio Button Group**. Enter **Select your preference:**.
- ▶ Drag and drop a Radio Button Group component from the palette.
- ▶ Drag and drop a Command - Button from the palette.
- ▶ Add a couple of new lines. Enter **Your preference was:** and add an Output component.
- ▶ Select the Radio Button Group component. In the Basics tab of the Attributes view, select *Vertical* in the combobox of the Direction property. Using the Choices tab, add three text items (click three times *Add Text Item*). Select the first item and enter **High** for the label and **very much** for the value. In the second choice, enter **Medium** for label and **medium** for the value. For the third item, enter **Low** for the label and **very low** for the value.
- ▶ Using the Page Data view, define a new requestScope variable with the name **choice** and the type `java.lang.String`.
- ▶ Now bind the new request variable to the Radio Button Group component and also bind it to the Output component. This variable takes the value of the Radio Button Group selection and displays it in the Output text.
- ▶ Save and test the page. Select one of the choices and click *Submit* (Figure 4-22).

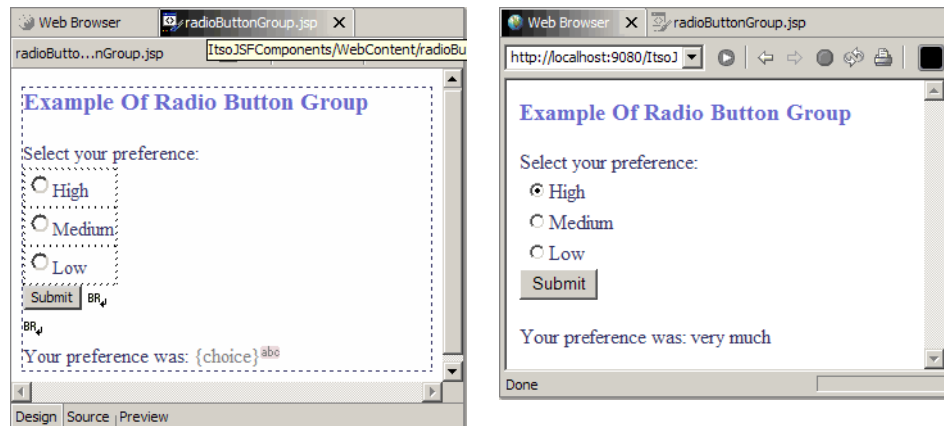


Figure 4-22 Radio Button Group example design and execution

How it works

This component renders an HTML table with a set of Input components of type radio. All the radio button components have the same name, so only the value of the selected radio button is sent to the server. The value is sent to the server using a hidden field.

Check Box: <h:selectBooleanCheckBox>

This component is a check box with two states: checked and unchecked. It represents a boolean value inside a Web page.

Component type

UISelectBoolean The `UISelectBoolean` class defines components that have a boolean value.

Attributes

Table 4-27 shows the attributes of the `<h:selectBooleanCheckBox>` component.

Table 4-27 Check Box attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, target, title
converter	Converter instance registered with this component.
required	This flag indicates that the user is required to provide a submitted value for this input component.
validator	Method binding representing a validator method that is called during Process Validations to perform correctness checks on the value of this component.
valueChangeListener	Method binding representing a value change listener method that is notified when a new value has been set for this input component.
readonly	Flag indicating that this component prohibits changes by the user. The element may receive focus unless it has also been disabled.
tabindex	Position of this element in the tabbing order for the current document. This value must be an integer between 0 and 32767.

Events

Table 4-28 shows the event of the `<h:selectBooleanCheckBox>` component.

Table 4-28 Check Box Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when text within this element is selected by the user.

Usage

The `<h:selectBooleanCheckbox>` tag is the only tag that JavaServer Faces technology provides for representing boolean state. This component is used to check a boolean condition. Here is an example:

- ▶ Create a Faces JSP page and name it **checkBox.jsp**. Change the text to **Usage of Check Box Sample**. Drag a Check Box component. Insert a text describing it, if you want. Add two more check boxes.
- ▶ In the Page Data view define a requestScope variable named **first** of type `java.lang.Boolean`. Bind the variable to the first check box.
- ▶ Add a Command - Button component. In the Quick Edit view add the action logic:

```
log("Checkbox1 request="+getRequestScope().get("first"));
log("Checkbox2="+getCheckbox2().getValue());
log("Checkbox3="+getCheckbox3().getValue());
return "";
```
- ▶ Now you can save and test the page. In the Console either true or false is displayed.

How it works

When you use this component, the generated code is:

```
<h:selectBooleanCheckbox styleClass="selectBooleanCheckbox"
id="checkbox1" value="#{requestScope.first}"></h:selectBooleanCheckbox>
first checkbox
```

This component renders an HTML input element of type check box. If you want to bind a check to a variable or property, use a `java.lang.Boolean` type.

Check Box Group: <h:selectManyCheckBox>

This component creates a group of possible selections. Each selection is represented by a check box that can be checked or unchecked. Using this component you can select zero or more items from the group.

Component type

UISelectMany The UISelectMany class defines components that allow the user to select zero or more values from a set of values.

Attributes

Table 4-29 shows the attributes of the <h:selectBooleanCheckBox> component.

Table 4-29 Check Box Group attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
converter	Converter instance registered with this component.
required	This flag indicates that the user is required to provide a submitted value for this input component.
validator	Method binding representing a validator method that is called during Process Validations to perform correctness checks on the value of this component.
valueChangeListener	Method binding representing a value change listener method that is notified when a new value has been set for this input component.
readonly	Flag indicating that this component prohibits changes by the user. The element may receive focus unless it has also been disabled.
tabindex	Position of this element in the tabbing order for the current document. This value must be an integer between 0 and 32767.
border	Width (in pixels) of the border to be drawn around the table containing the options list.
disabledClass	CSS style class to apply to the rendered label on disabled options.
enabledClass	CSS style class to apply to the rendered label on enabled options.

Attributes	Description
layout	Orientation of the options list to be created. Valid values are <code>pageDirection</code> (list is laid out vertically), or <code>lineDirection</code> (list is laid out horizontally). If not specified, the default value is <code>lineDirection</code> .

Events

Table 4-30 shows the events of the `<h:selectBooleanCheckBox>` component.

Table 4-30 Check Box Quick Edit events

Event	Description
<i>common</i>	<code>onblur</code> , <code>onclick</code> , <code>ondblclick</code> , <code>onfocus</code> , <code>onkeydown</code> , <code>onkeypress</code> , <code>onkeyup</code> , <code>onmousedown</code> , <code>onmousemove</code> , <code>onmouseout</code> , <code>onmouseover</code> , <code>onmouseup</code>
<code>onchange</code>	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
<code>onselect</code>	JavaScript code executed when text within this element is selected by the user.

Usage

The `<h:selectManyCheckbox>` tag renders a set of check boxes, with each check box representing one value that can be selected. To use this component follow these steps:

- ▶ Create a Faces JSP page and name it **checkBoxGroup.jsp**. Change the text to **Usage of Check Box Group Sample**. Drag a Check Box Group component. Insert a text describing it, if you want.
- ▶ In the Attributes view set **checkbox** as the `id` property (Basics tab). Select *Vertical* in the Direction combo box. In the Choices tab, add three new choices (click *Add Text Item* three times). Enter **First Choice**, **Second Choice** and **Third Choice** for their names. Enter **choice1**, **choice2** and **choice3** for their values. In the All tab, select **2** for the border property.
- ▶ Create a JavaBean `itso.jsf.Choices`, with one property with getter/setter:

```
private String[] choices = null;
```
- ▶ Add the Choices bean to Page Data with the name **myChoices** (no scope). Then bind the check box group to `myChoices.choices`.
- ▶ Add a Command - Button component with the action logic:

```
log("Choices were:");
String[] choices = getMyChoices().getChoices();
for (int i=0; i<choices.length; i++) {
```

```

        log("choice="+choices[i]);
    }
    return "";
}

```

- Save the page and test it. Select some of the check boxes and watch the Console output. The selected choice values are listed.

How it works

The JSP code generated is:

```

<h:selectManyCheckbox styleClass="selectManyCheckbox" id="checkbox1"
    layout="pageDirection" border="2"
    value="#{pc_CheckBoxGroup.myChoices.choices}">
    <f:selectItem itemLabel="First Choice" itemValue="choice1" />
    <f:selectItem itemLabel="Second Choice" itemValue="choice2" />
    <f:selectItem itemLabel="Third Choice" itemValue="choice3" />
</h:selectManyCheckbox>

```

The rendered output for this tag is an HTML option list. You can select one or more values. All item values of the selected items are sent to the server in the request.

You can bind the check box group to a String array. This array will contain the values of the selected items.

Combo Box: <h:selectOneMenu>

This component represent a list box. The user can select one of the possible values displayed in the list.

Component type

UISelectOne The UISelectOne component allows the user to select a value from a set of values. This component can be rendered as a list box, a radio button o a menu.

Attributes

Table 4-31 shows the attributes of the <h:selectOneMenu> component.

Table 4-31 Combo Box attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, target, title
converter	Converter instance registered with this component.

Attributes	Description
required	This flag indicates that the user is required to provide a submitted value for this input component.
validator	Method binding representing a validator method that is called during Process Validation to perform correctness checks on the value of this component.
valueChangeListener	Method binding representing a value change listener method that is notified when a new value has been set for this input component.
readonly	Flag indicating that this component prohibits changes by the user. The element may receive focus unless it has also been disabled.
tabindex	Position of this element in the tabbing order for the current document. This value must be an integer between 0 and 32767.

Events

Table 4-32 shows the events of the `<h:selectOneMenu>` component.

Table 4-32 Combo Box Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when text within this element is selected by the user.

Usage

This is an example of how to use the Combo Box component to display a simple formatted concatenated message. To do the example follow these steps:

- ▶ Create a JSF page called **comboBox.jsp** and change the text to **Sample of Combo Box Usage**. Drag a Combo Box component from the palette onto the page below this text. Insert a description before it if you want.
- ▶ Using the Attributes view do the following:
 - Set its name to **combo1**.

- In the Choices tab, add 3 new (*Add Text Items*) with names **Choice 1**, **Choice 2**, and **Choice 3**. Let their values be **choice1**, **choice2**, and **choice3** respectively.
- Save the page and test it.

How it works

The source generated in the example is this:

```
<h:selectOneMenu styleClass="selectOneMenu"
    id="combo1" title="First example of combo">
    <f:selectItem itemLabel="Choice 1" itemValue="choice1" />
    <f:selectItem itemLabel="Choice 2" itemValue="choice2" />
    <f:selectItem itemLabel="Choice 3" itemValue="choice3" />
</h:selectOneMenu>
```

The tool generates a `<h:selectOneMenu>` tag for the combo and for each choice generates a `<f:selectItem>` tag. This tag is one of the JSF core tag (see “Core library” on page 97 for more details). These tags are used because you can bind the `itemLabel` property and/or the `itemValue` property to a bean, variable or a property in a resource bundle. You can bind the combo box to a String variable.

If the `styleClass` attribute is specified, it renders a `span` element and render the value of the `styleClass` attribute as the value of the `class` attribute on the `span` element. Then a HTML `select` element is rendered. After that, the items are computed and rendered inside the `select` element.

List Box - Single Select: `<h:selectOneListBox>`

This component is a list of items where only one of them can be selected at one time.

Component type

UISelectOne

The `UISelectOne` component allows the user to select one value from a set of values.

Attributes

Table 4-33 shows the attributes of the `<h:selectOneListBox>` component.

Table 4-33 List Box - Single Select attributes

Attributes	Description
<i>common</i>	<code>id</code> , <code>value</code> , <code>style</code> , <code>styleClass</code> , <code>rendered</code> , <code>immediate</code> , <code>binding</code> , <code>accesskey</code> , <code>dir</code> , <code>disabled</code> , <code>lang</code> , <code>target</code> , <code>title</code>
<code>converter</code>	Converter instance registered with this component.

Attributes	Description
required	This flag indicates that the user is required to provide a submitted value for this input component.
validator	Method binding representing a validator method that is called during Process Validation to perform correctness checks on the value of this component.
valueChangeListener	Method binding representing a value change listener method that is notified when a new value has been set for this input component.
readonly	Flag indicating that this component prohibits changes by the user. The element may receive focus unless it has also been disabled.
tabindex	Position of this element in the tabbing order for the current document. This value must be an integer between 0 and 32767.
size	Number of available options to be shown at all times. If not specified, all available options are shown.

Events

Table 4-34 shows the events for the <h:selectOneListBox> component.

Table 4-34 List Box - Single Select Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when text within this element is selected by the user.

Usage

In this example we show the usage of the List Box - Single Select component. You use this component when you want to select only one item from a list. Follow these steps:

- Under Java Resources create a new plain file and name it **continents_en.properties**. Enter these lines into the file:

```
amshort=america
```

```

amlong=American Continent
eushort=europe
eulong=European Continent
afshort=africa
aflong=African Continent
aushort=australia
aulong=Australian Continent
asshort=asia
aslong=Asian Continent

```

- Create a Faces JSP page and name it **listSingleBox.jsp**. Change the text to **Sample List Box - Single Select Usage**.
- To use the resource file in the JSP page add this line after the closing head tag (</HEAD>) :

```
<f:loadBundle var="continents" basename="continents" />
```

- Drag and Drop a List Box - Single Select component into the page.
- In the Attributes view Basics tab, set the id to **listbox** and set height to **3**. In the Choices tab, add five text items. Name the first **#{continents.amlong}** with a value of **#{continents.amshort}** and fill the other as shown in Figure 4-23.

Note. The name is displayed in the list box at execution time, and the value of the selected item can be passed to action logic by binding the components value attribute to a variable (Basics tab).

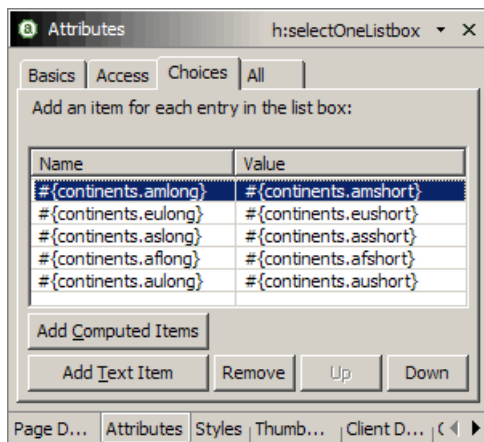


Figure 4-23 List Box - Single Select choices

- Save and test the page. You see a list of five continents and you can select one of them. The name and values are obtained from a resource bundle.

How it works

The code generated in the example is shown here:

```
<h:selectOneListbox styleClass="selectOneListbox" id="listbox" size="3">
  <f:selectItem itemLabel="#{continents.amlong}" itemValue="#{continents.amshort}" />
  <f:selectItem itemLabel="#{continents.eulong}" itemValue="#{continents.eushort}" />
  <f:selectItem itemLabel="#{continents.aslong}" itemValue="#{continents.asshort}" />
  <f:selectItem itemLabel="#{continents.aflong}" itemValue="#{continents.afshort}" />
  <f:selectItem itemLabel="#{continents.aulong}" itemValue="#{continents.aushort}" />
</h:selectOneListbox>
```

For each item defined, a `<f:selectItem>` tag is used. We have bound a resource bundle property to the `itemLabel` and `itemValue` attributes. Other beans or scope variables can be bound instead.

This component renders an HTML select tag where you only can select one element at a time.

List Box - Multiple Select: `<h:selectManyListbox>`

This component is similar to the List Box - Single Select, but you can select as many items from the list as you want.

Component type

UISelectMany The `UISelectMany` class defines components that allow the user to select zero or more values from a set of values.

Attributes

Table 4-35 shows the attributes for the `<h:selectManyListbox>` component.

Table 4-35 List Box - Select Many attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, target, title
converter	Converter instance registered with this component.
required	This flag indicates that the user is required to provide a submitted value for this input component.
validator	Method binding representing a validator method that is called during Process Validations to perform correctness checks on the value of this component.

Attributes	Description
valueChangeListener	Method binding representing a value change listener method that is notified when a new value has been set for this input component.
readonly	Flag indicating that this component prohibits changes by the user. The element may receive focus unless it has also been disabled.
tabindex	Position of this element in the tabbing order for the current document. This value must be an integer between 0 and 32767.
size	Number of available options to be shown at all times. If not specified, all available options are shown.

Events

Table 4-36 shows the events of the `<h:selectManyListBox>` component.

Table 4-36 List Box - Select Many Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when text within this element is selected by the user.

Usage

You can follow the instructions of the List Box - Single Select component example to create a List Box - Multiple Select example, or you can reuse the `listSingleBox.jsp` page:

- Drag and Drop a List Box - Multiple Select component into the page.

Note: We are reusing the resource bundle made in the previous example. This resource is already included in the `listBoxSingle.jsp`.

- In the Attributes view:
 - In the Basics tab, set the `id` to `listboxmultiple`. Set height to **3**.

- In the Choices tab, add five text items. Use the same names and values as in the single list box (Figure 4-23 on page 138).
- Note.** You can copy the `<f:selectItem>` tags in the Source tab.
- In the All tab, select the size property and set it to **3**.
- Save and test the page. Now you have two list boxes (Figure 4-24). The second one allows you to select more than one element.

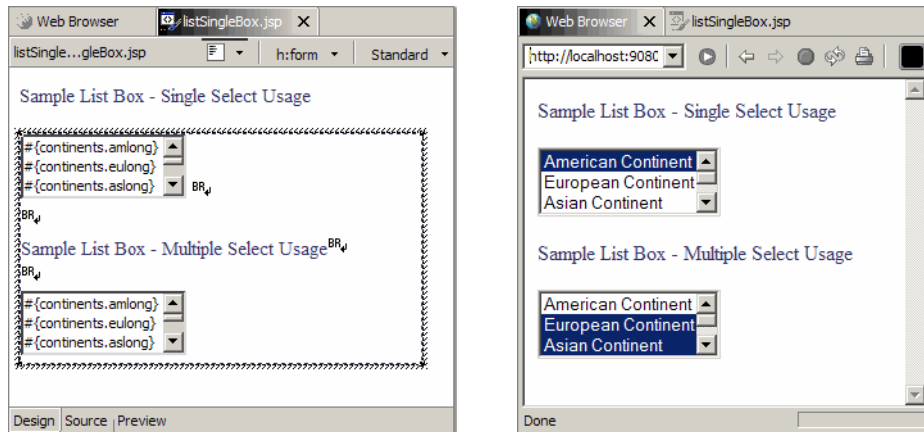


Figure 4-24 List Box - Single and Multiple Select in the same page

How it works

The behavior of this component is the same as the List Box - Single Select component. The only difference is that you can select zero or more elements from the list.

Data Table: `<h:dataTable>`

The Data Table component acts like a HTML table. You use this component when you want to show a set of results in a table. These result may include data from a database or data source, a list of values from a bean, a scope variable, and so forth. It is very useful when you do not know the total number of items of the list because this component lets you iterate over the data.

Component type

UIData

The UIIData component supports data binding to a collection of data objects. It does the work of iterating over each record in the data source. The standard table renderer displays the data as an HTML table.

Attributes

Table 4-37 shows the attributes of the `<h:dataTable>` component.

Table 4-37 Attributes for the Data Table component

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
bgcolor	Name or code of the background color for this table.
border	Width (in pixels) of the table border.
cellpadding	Definition of how much space should be between the border of each cell and its contents.
cellspading	Definition of how much space should be between the left side of the table and the leftmost column, the top of the table and the top of the top side of the topmost row, and so on for the right and bottom of the table. It also specifies the amount of space to leave between cells.
columnClasses	This attribute describes a list of CSS style classes (separated with a comma) that are applied to the columns of this table. For any individual column, a list of CSS styles separated by space may also be specified. If the number of elements in this list is less than the number of columns specified in the <code>columns</code> attribute, no <code>class</code> attribute is output for each column greater than the number of elements in the list. If the number of elements in the list is greater than the number of columns specified in the <code>columns</code> attribute, the additional elements are ignored.
dir	Direction indication for text that does not inherit directionality. Valid values are LTR (left-to-right) and RTL (right-to-left).
first	Number of the first row to be displayed. If this property is set to zero, rendering begins with the first row of the underlying data.
footerClass	List of CSS style class(es) separated by space that is applied to any footer generated for this table.
rules	This property specifies which rules appear between cells within this table. Valid values are: none (no rules, default value); groups (between row groups); rows (between rows only); cols (between columns only); and all (between all rows and columns).

Attributes	Description
summary	Summary of this table's purpose and structure, for render it using non-visual media such as speech and Braille.
var	Name of a request-scope attribute under which the model data for the row selected by the current value of the rowIndex property (and also the current value of the rowData property) is exposed.
width	Width of the entire table.

Events

Table 4-38 shows the event of the <h:dataTable> component.

Table 4-38 Data Table Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
Command	Adding code to this event causes a method to be generated in the page code class and a binding reference placed in the action attribute for this component.

Usage

You use a Data Table component when you want to display (or display and update) a collection of data and you do not know how many elements are in the set.

We introduce now a simple example of the usage of this component. We use a simple bean that contains a String array. The array is initialized with a hundred of elements and those element are shown (and they have the possibility of being updated) in the Data Table component. Here are the instructions:

- ▶ Under the Java Resources folder create a new package and name it **itso.jsf**.
- ▶ Import the Java file \labscode\dev-component\JSFArray.java into the package.
- ▶ In the WebContent folder, create a new faces JSP file and name it **dataTable.jsp**. Change the default text for **Data Table sample**.
- ▶ In the Page Data view right-click and select *New -> Add a new JavaBean*. In the pop-up window select **array** for the name of the variable and **itso.jsf.JSFArray** for the type of the variable. Select *Make this bean reusable* and select *session* from the scope combo box (Figure 4-25). Click *Finish*.

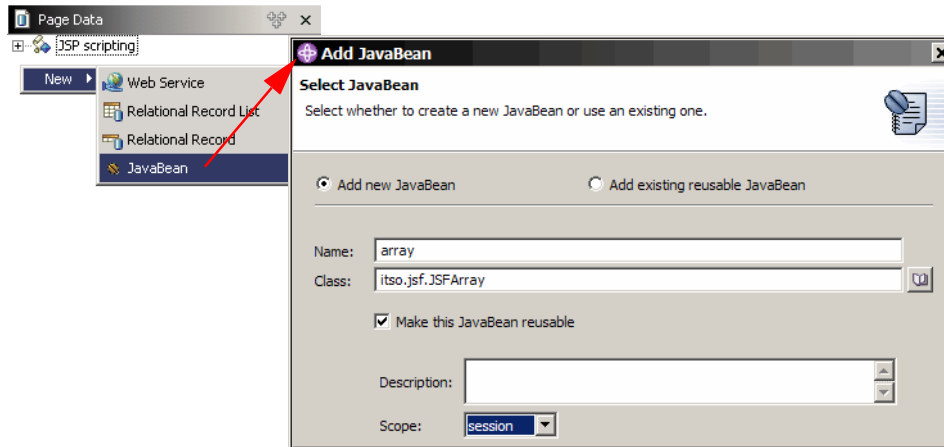


Figure 4-25 Defining the bean for the data table

- Drag a Data Table component from the palette. A little square appears. In the Attributes view do the following enter **dataTable** for the id of the component. Leave the data table selected in the Page Designer.
- In the Page Data view, expand the array bean. Select Contained Type under elements and *Change Contained Type* (context). Change the type to `java.lang.String`.
- Select the elements array and *bind to 'dataTable'*. In the pop-up window edit the label and enter **Array Elements** (Figure 4-26). Click *Finish*.

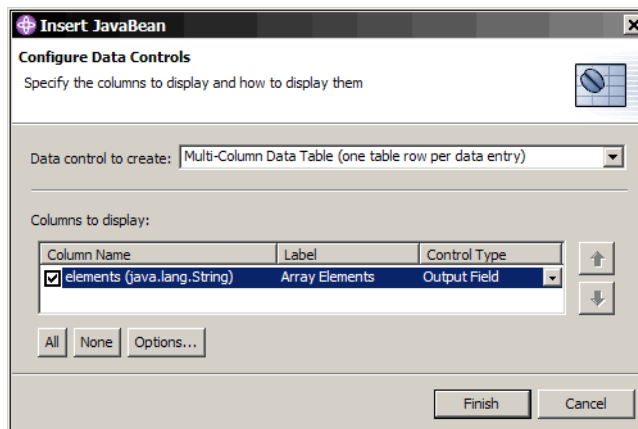


Figure 4-26 Binding the array to the Data Table component.

- Select the heading and in the Attributes view enter **color: teal; font-size: 14pt** for the Properties property in the Style section.

- ▶ Select the Data Table component. In the Attributes view, Paging tab, enter **8** in the Items/Page property. Click *Add a deluxe pager*. We are paging the data by eight elements, with a deluxe pager at the footer of the table.

Sometimes the pager does not appear in the design page. If this is the case, go to the Source tab and move the footer tag after the end of the column tag:

```
.....
</h:column>
<f:facet name="footer">
    <hx:panelBox styleClass="panelBox" id="box1">
        <hx:pagerDeluxe styleClass="pagerDeluxe" id="deluxe1" />
    </hx:panelBox>
</f:facet>
```

- ▶ Select the Data Table component. In the All pane, enter **2** for the value of the border property.
- ▶ Now save and test the JSP. We have a data table with 100 elements, displaying eight at a time.

How does it work

The value attribute of Data Table component references the data to be included in the table. This data can take the form of:

- ▶ A list of beans
- ▶ An array of beans
- ▶ A single bean
- ▶ A `javax.faces.model.DataModel`
- ▶ A `java.sql.ResultSet`
- ▶ A `javax.servlet.jsp.jstl.sql.ResultSet`
- ▶ A `javax.sql.RowSet`

The `UIData` component also has the ability to display only a subset of the underlying data. To do this you use the optional `first` and `rows` attributes. You specify the first row and the number of rows to display.

The rendering of the table is performed as follows:

- ▶ If the table has a header, this header is rendered using `tr` and `th` elements. It applies the styles for the tag in the case it is necessary.
- ▶ If the table has footer, the same process as for the header is applied to it.
- ▶ The body is rendered using a `tbody` element. It keeps track of the result of the `rows` property. It also keeps track of the number of rows that have been rendered so far. Set the `rowIndex` property to the `UIData` component as we iterate through the rows. Output a `tr` element to start with the rows.
- ▶ Output the value of the `rowClasses` according to the attribute description. For each `UIColumn` child, output a `td` element, attaching the value of the

columnClasses attribute of the UIData component. Recursively encode each child of each UIColumn child. Close out the td element. When done with the row, close out the tr element. When done with all the rows, close out the tbody element.

- When the rendering of all the rows is done, set the rowIndex property of the UIData to -1, and close out the table element.

Tip: The Data Table has been extended with additional function through IBM extensions. See “Data Table component extensions” on page 186 for details.

IBM Extension Library

One of the major benefits of JSF is its ability to be extended with useful components to support application requirements. The JSF specification has included a base set of components that map to HTML controls. The WebSphere Studio product now includes a powerful set of extension components that improve productivity and add richness to the existing base components (see Table 4-6 on page 101).

The JSF community will eventually include a large set of open source and product supplied components that customers from small businesses to enterprises can use in application development. Table 4-39 lists the extension components that are shipped with WebSphere Studio 5.1.2.

Table 4-39 List the IBM extension components on the Palette

Palette Name	Tag Name	Type
 Command - Button	<hx:commandExButton>	UICommand
 Link	<hx:outputLinkEx>	UICommand
 Image	<hx:graphicImageEx>	UIGraphic
 File Upload	<hx:fileUpload>	
 Horizontal Rule	<hx:outputSeparator>	UIOutput
 Panel Group Box - HTML Panel	<hx:jspPanel>	UIPanel
 Panel Group Box - Snap to border	<hx:panelLayout>	UIPanel
 Panel Group Box - List	<hx:panelBox>	UIPanel
 Panel Menu Bar	<hx:panelActionBar>	UIPanel
 Panels - Tabbed	<odc:tabbedPanel>	

Palette Name	Tag Name	Type
 Rich Text Area	<r:inputRichtext>	
 Generic A/V Player	<hx:playerGenericPlayer>	
 Macromedia Flash Player	<hx:playerFlash>	
 Macromedia Shockwave Player	<hx:playerShockwave>	
 RealOne Player	<hx:playerRealPlayer>	
 Windows MediaPlayer	<hx:playerMediaPlayer>	

Note: Two of the IBM extensions replace standard components (**Command - Button** and **Image**); the other components are additional components.

These components can be found in the Palette view of the Web perspective. They can be easily dragged into the Page Designer and their attributes modified with the Attributes view and the Source tab.

The sections that follow explain each component in detail.

Command - Button: <hx:commandExButton>

The Command - Button behaves like a HTML form button. When you click the button an event is generated. You can write action logic that is executed for this event and you return a symbolic name that is used in navigation rules to decide on the next step of the application.

Attributes

Table 4-40 shows the attributes of the <hx:commandExButton> component.

Table 4-40 Command - Button attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
action	Value returned when the component is clicked. This value is used in the navigation rules.
actionListener	The name of the class that respond to actions.
target	The position of the component in the tabbing order.
alt	Alternate text for the component.

Attributes	Description
converter	Specifies the data type to convert the value to.
image	The path for the image of the component.
imagekey	Used in conjunction with the bundle attribute, the imageKey attribute relates to a key within the set of key/value pairs specified in the bundle, where the value is a URL/URI to an image file.
readonly	The value of the component cannot be changed from its initial value.
size	Specifies the initial width for the component.
tabindex	The tab order of the component.
type	Whether the component is <i>Submit</i> or <i>Reset</i> .

Events

Table 4-41 shows the events of the `<hx:commandExButton>` component.

Table 4-41 Command - Button Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when text within this element is selected by the user.
Command	Adding code to this event causes a method to be generated in the page code class and a binding reference placed in the <code>valueChangeListener</code> attribute for this component.

Usage

This example shows how a Command - Button can be used to link to another page using navigation rules:

- Create a Faces JSP file called **commandButton.jsp**.
- You can place a Command - Button component into a JSF page by dragging it from the palette to a destination within the page (Figure 4-20 on page 125).
- The control is displayed as a button.

- ▶ Within the Attributes view on the Format tab change the Label property to **Jump Now!**.
- ▶ We have to create a page to jump to, using the familiar process. Create a page called **anotherPage.jsp**. Change the default text to **Thank you for jumping here**. Save and go back to the `commandButton.jsp` page.
- ▶ We now define the navigation rule to display another page when clicking the button. In the Attributes view select the Navigation tab and click *Add*. Select **anotherPage.jsp** and specify **jump** for the alias and ***** for the action. Click *OK* (Figure 4-27).

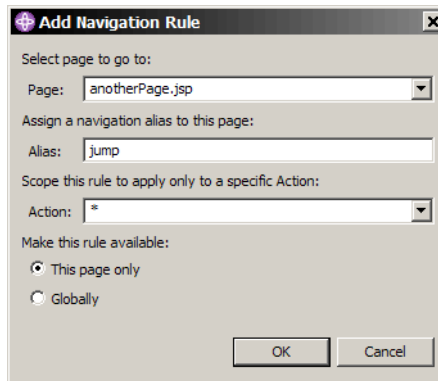


Figure 4-27 Navigation rule for a Command - Button

- ▶ You can see that the rule has been added to the list.
- ▶ Now we have to react to the click event. Select the button and in the Quick Edit view select *Command*. Change the return string to **jump**:

```
return "jump";
```
- ▶ Save the page. You are now ready to test it. When you click the button the navigation rule forces the display of the `anotherPage.jsp` page.

How it works

For the more experienced developers we can examine how the Command - Button controls these events:

- ▶ Select the Command - Button control. We can examine the JSF tags that have been automatically generated after we dropped the control and how the values have changed through the Attribute view:

```
<hx:commandExButton type="submit" value="Jump Now!"
  styleClass="commandExButton" id="button1"
  action="#{pc_CommandButton.doButton1Action}">
</hx:commandExButton>
```

- ▶ In the Page Data view you can see that a new action method called `doButton1Action` has been defined. You can reuse these events and drag them onto other controls, for example, if you have a common piece of code that has to be invoked from multiple places on the page. The action attribute of the target control points to the action logic.
- ▶ In the Attributes view All tab you can see that the action attribute has been bound to the action event method:

```
# {pc_CommandLink.doButton1Action}
```

- ▶ We can now examine the page code by selecting *View Page Code*. This displays the Java code that is controlling all the events of the page. Look for the action logic:

```
public String doButton1Action() {
    // Type Java code to handle command event here
    // Note, this code must return an object of type String (or null)

    return "jump";
}
```

- ▶ In this example we did not use any Java code. All we changed was the return value for the method. This value matches the alias for the navigation rule. To manage a more complicated navigation we could create more pages and navigation rules and use `if` statements to control the logic flow and change the resulting return value.

The Command - Button control is widely used in JSF application as a key component to control page navigation. It offers support for image buttons and mouse overs so it can be fully customized to work within you User Interface guidelines. You can see this control used in the JSF banking application.

Link: <hx:outputLinkEx>

This component allows you to create an HTML link in a page. It creates a link and the text to be displayed for the link. As additional resources you can create parameters to be used in the link.

Attributes

Table 4-42 shows the attributes for the `<hx:outputLinkEx>` component.

Table 4-42 Link attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title

Attributes	Description
charset	The character encoding of the resource that this component is linking to.
coords	The position and shape of the hot spot on the screen (for use in client-side image maps).
hreflang	The language code of the resource designated by this hyperlink
mimeType	Used for declaring the media MIME type when src is bound to binary data
rel	The relationship from the current document to the anchor specified by this hyperlink. The value of this attribute is a space-separated list of link types.
rev	A reverse link from the anchor specified by this hyperlink to the current document. The value of this attribute is a space-separated list of link types.
shape	Sets the shape of the link.
tabindex	Specifies the position of the component in the tabbing order in the JSP.
target	Assigns the target frame within a frameset that a page should be loaded into when the hyperlink is clicked.
type	Specifies the type <i>Submit</i> or <i>Reset</i> .

Events

Table 4-43 shows the events of the `<hx:outputLinkEx>` component.

Table 4-43 Link Quick Edit Events

Event	Description
<i>common</i>	onblur, onfocus

Usage

In this example we show how to use a Link component:

- Create a faces JSP files and named **linkExample.jsp**. Add a suitable title.
- Drag and drop a Link component in the **linkExample1.jsp** page. In the pop-up dialog, enter **anotherPage.jsp** for the URL and **Go to another page** for the text to be displayed. A Link component and an OutputText component are created (Figure 4-28).

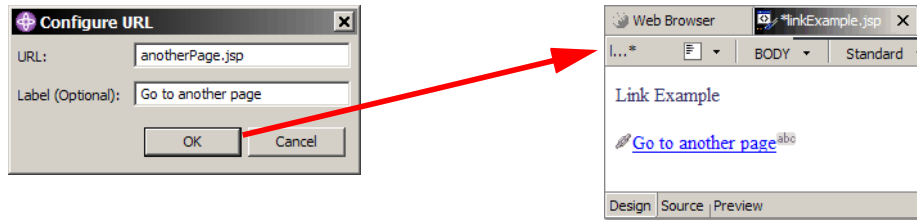


Figure 4-28 Dragging a Link component

Now we add parameters to the link to pass them to the second page for display:

- ▶ Select the Link component in `linkExample.jsp` page and go to the Attributes view. Select the Parameters tab and click *Add* twice to add two parameters. Enter **first** and **second** for their names and **First Parameter** and **Second Parameter** for their values, respectively. Save the page.

Note: Note that the value of the parameters can be bound to a variable or bean defined in the application.

- ▶ Open the `anotherPage.jsp` and expand JSP scripting in the Page Data view. Select `param` and *Add parameter* (context). Enter **first** for the name. Repeat the process again to define a second parameter and enter **second** for its name.
- ▶ Select both parameters together and drop them into the JSP. In the pop-up window change the label to **Parameter 1:** and **Parameter 2:** respectively (Figure 4-29).

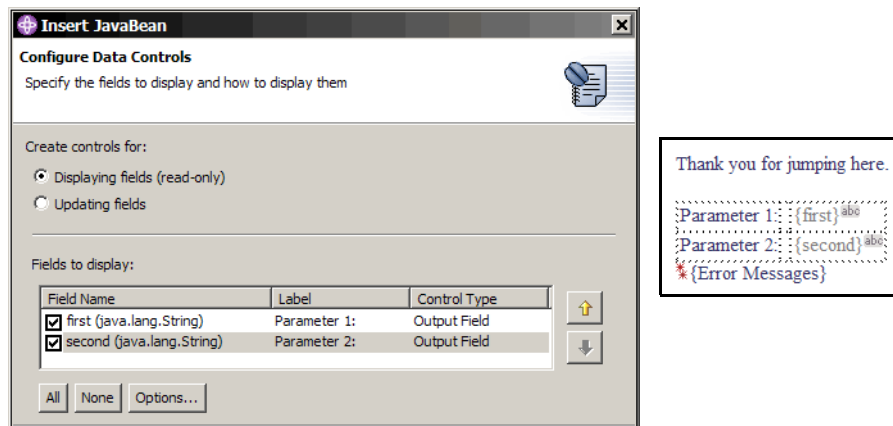


Figure 4-29 Dragging the parameters into the page

- Test `linkExample.jsp`. When you select the link, `anotherPage.jsp` is loaded and the two parameters are displayed.

How it works

When the JSF framework renders this component, it creates a new HTML anchor component. This anchor component has the URL to get to the target specified in the `value` attribute of the `Link` component. The text of the anchor component is the `value` property of the `Output` component.

The `Link` component in the example is rendered as follows:

```
<a id="linkEx1"
  href="anotherPage.jsp?param1=First%20Parameter&param2=Second%20Parameter"
  class="outputLinkEx">
  <span id="text1" class="outputText">Go to another page</span>
</a>
```

Notice in the code how the parameters are passed to the target page.

Image: <hx:graphicImageEx>

This component allows you to display an image on the page. You can display an image from a data source or even compute the source for the image programmatically.

Attributes

Table 4-44 shows the attributes of the `<hx:graphicImageEx>` component.

Table 4-44 Image attributes

Attributes	Description
<i>common</i>	<code>id</code> , <code>value</code> , <code>style</code> , <code>styleClass</code> , <code>rendered</code> , <code>immediate</code> , <code>binding</code> , <code>accesskey</code> , <code>dir</code> , <code>disabled</code> , <code>lang</code> , <code>title</code> .
<code>align</code>	Specifies the alignment of the component. There are nine choices for this attribute.
<code>alt</code>	Specifies the alternate text for the component.
<code>border</code>	Set the border of the component.
<code>height</code>	Set the component's height.
<code>hspace</code>	Specifies the amount of white space in pixels to be inserted to the left and right of the component.

Attributes	Description
ismap	If set to true, the image component is set to use a server side image map. Note that the graphic component must be contained within a hyperlink component.
longdesc	Specifies a link to the long description for the image. This attribute should supplement the alt attribute.
mimeType	Used for declaring the media MIME type when src is bound to binary data
tabindex / target	Specifies the tab index for the component.
usemap	Specifies an image map, defined by a MAP element, to be used with the component. This attribute must match the name of the MAP element.
vspace	Specifies the amount of white space in pixels to be inserted to the top and bottom of the component.
width	Specifies the width of the image.

Events

Table 4-45 shows the events of the <hx:graphicImageEx> component.

Table 4-45 Image Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.

Usage

In this example we show how to use an image component. Import the images used in the banking application as examples:

- ▶ Create a folder named **images** under the WebContent folder.
- ▶ Import the images from \labscode\images folder.
- ▶ Create a Faces JSP file and name it **graphicImage.jsp**.
- ▶ Drag and drop an Image component from the palette. In the Attributes view locate the file using *Browse* and select the **redbooks.gif** from the images folder (Figure 4-30).

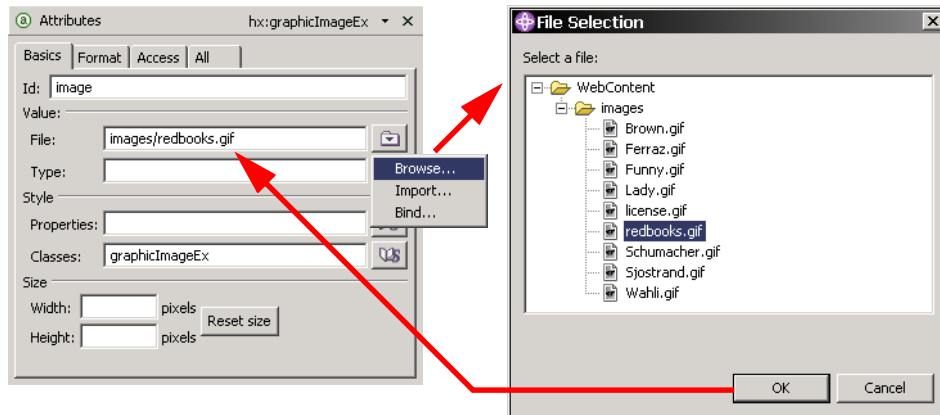


Figure 4-30 Defining the resource for the Image component

- In the page design you can see the GIF file. Save and test the file and the image is displayed in the JSP.

How it works

When rendered, this component send an image HTML tag to the client. The src for the image is computed and added to the tag to display the image correctly.

If the image is mapped, the map attribute is rendered to the client to display the image with all the map sections.

File Upload: <hx:fileUpload>

This component displays an input field and associated *Browse* button for uploading a file.

Attributes

Table 4-46 shows the attributes of the <hx:fileUpload> component.

Table 4-46 File Upload attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
accept	Specifies a list of content types that the server handles correctly. This list is a comma-separated list.
alt	Alternate text for the component.

Attributes	Description
exclude	Mime types to be excluded.
maxLength	The maximum number of characters that you can enter in the field.
name	Name for the component. It should be unique in the JSP.
size	Initial width for the component.
tabindex / target	The position of the component in the JSP tab order.

Events

Table 4-47 shows the events of the `<hx:fileUpload>` component.

Table 4-47 File Upload Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup
onchange	JavaScript code executed when this element loses focus and its value has been modified since gaining focus.
onselect	JavaScript code executed when text within this element is selected by the user.
valueChanged	JavaScript code executed when the component changes its value. The <code>valueChangedEvent</code> is passed and you can get either the old and the new values.

Usage

Now we create a simple example that shows how to use the File Upload component. We upload an image file into a backend bean:

- ▶ Create a package named **itso.jsf** under the Java Resources folder. Import the file `\labscode\dev-component\BinaryFile.java` into the new package. We use this bean to hold a binary file using a byte array.
- ▶ Create a Faces JSP page and name it **fileUpload.jsp**. Delete the default text and enter **File Upload Example**. Under this text enter **Select an image file to send to the server:**.
- ▶ Create a Faces JSP page and name it **fileUploadResult.jsp**. Close it for now.

- ▶ In the `fileUpload.jsp` page, drag and drop a File Upload component into the page. Add two new lines (*Enter* key twice). Drag and drop a Command - Button component into the page.
- ▶ In the Page Data view, define a new JavaBean. In the pop-up window, enter **binaryfile** for its name and select the Java class **itso.jsf.BinaryFile**. Make the bean reusable in the session scope. Click *Finish*.
- ▶ Expand the bean and bind its `file` attribute (that is a byte array) to the File Upload component.
- ▶ Select the Submit button and define a navigation rule from `fileUpload.jsp` page to `fileUploadResult.jsp` page with the **go** alias.
- ▶ Using the Quick View editor, put the following code in the Command event:


```
return "go";
```
- ▶ Save the page (Figure 4-31).

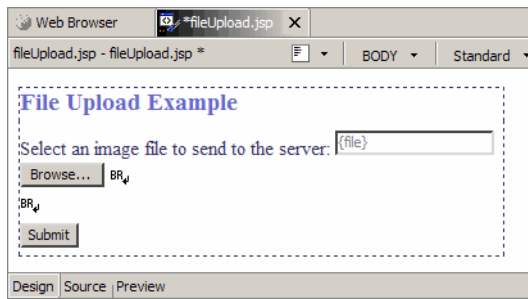


Figure 4-31 Design of the file upload page

- ▶ Open the `fileUploadResult.jsp` page. This page is used to display the image that is sent to the server. Change the default text and enter **File Upload Example**. Add a new line of text and enter **This is the image file you sent**, add a new line and drag an Image Component after that.
- ▶ In the Page Data view define a new JavaBean. Select *Add existing reusable JavaBean* and select `binaryfile` from the list. Click *Finish*.
- ▶ Select the Image component. In the Basic tab of the Attributes view, select the `File` attribute and bind it to the `file` attribute of the `binaryfile` JavaBean. You can use the wizard to bind or enter this reference:

```
#{pc_FileUploadResult.binaryfile.file}
```

Because the source of the image is a byte array, we must set the MIME type for it. In the All tab of the Attributes view, select the `mimeType` attribute and set it to **image/gif**.

- ▶ Add a Command - Button component and enter **Back** for its label property (Format tab in the Attributes view).
- ▶ In the Page Data view define a new JavaBean. Select *Add existing reusable JavaBean* and select `binaryfile` from the list below. Click *Finish*.
- ▶ Select the Submit button and in the Quick Edit view enter the action logic:


```
return "back";
```
- ▶ Define a new navigation rule from `fileUploadResult.jsp` to `fileUpload.jsp` using the alias **back**.
- ▶ Save the page (Figure 4-32).



Figure 4-32 Design of the file upload result page

- ▶ Run the `fileUpload.jsp` and test it. When the page is in the browser, you select an image file by browsing the hard disk and the Web application sends its contents to the server. The `BinaryFile` JavaBean stores the file in the byte array and the image is displayed in the `fileUploadResult.jsp` page.

How it works

The code that is generated for the File Upload component is shown here:

```
<hx:fileupload styleClass="fileupload" id="fileupload"
  value="#{pc_FileUpload.binaryfile.file}">
  <hx:fileProp name="fileName" />
  <hx:fileProp name="contentType" />
</hx:fileupload>
```

The component value is bound to the JavaBean in the server and the data is stored there. Then two properties are created (`<hx:fileProp>` tags) with the file name and the content type. This data is use for rendering all the necessary code when the page is sent to the client.

When the response is rendered, the form is created to use the POST HTTP method (send a large amount of data). An input field of type file is rendered to allow you to browse the hard disk and find the file you want to upload. A new input field of type hidden is create to send all the form to the server. The send of the data is done by using a JavaScript library.

Horizontal Rule: <hx:outputSeparator>

This component creates a horizontal line to visually separate information on the page.

Attributes

Table 4-48 shows the attributes of the <hx:outputSeparator> component.

Table 4-48 Horizontal Rule attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment of the component.
converter	Specifies the data type to convert the value to.
height	Specifies the height of the component.
noshade	Specifies whether display the component in the normal way or using solid color.
width	The width of the component.

Usage

You can use the Horizontal Rule component to separate content inside the page.

How it works

When rendered, the Horizontal Rule component renders an <HR> HTML tag to show the separator in the page.

Panel Group Box - HTML Panel: <hx:jspPanel>

The HTML Panel component is the simplest container for other JSF components.

Attributes

Table 4-49 shows the attributes of the <hx:jspPanel> component.

Table 4-49 Panel Group Box - HTML Panel attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accessKey, dir, disabled, lang, title

Usage

This component is used as a simple container for other JSF components. It is used to hold other components inside more complex structures.

How it works

When this component is rendered, it creates SPAN HTML tags to hold the contained components.

Panel Group Box - Snap to Border: <hx:panelLayout>

This component creates a container in which you can group other components. This panel organizes the components along five areas: the top, the bottom, two sides, and the center. This layout is known as the *border* layout.

Attributes

Table 4-50 shows the attributes of the <hx:panelLayout> component.

Table 4-50 Panel Group Box - Snap to Border attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment of the component.
bgcolor	Specifies the background color.
border	Specifies the border of the panel.
frame	Specifies which sides of the frame surrounding a table are visible.
height	Specifies the height of the panel.
summary	Specifies a summary details string for user agents rendering to non-visual media.
width	Specifies the width of the panel.

Usage

This example shows how to use the Snap to Border component:

- ▶ Create a Faces JSP file and name it `snapToBorder.jsp`. Change the default text and enter **Example of Snap To Border Panel**.
- ▶ Drag and drop a Panel - Group Box component from the palette. In the pop-up window, select *Snap To Border* and click *OK* (Figure 4-33).

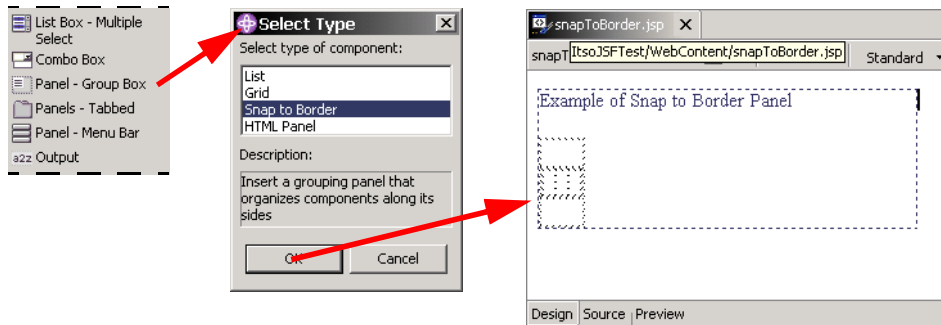


Figure 4-33 Creating a Snap To Border panel

Notice the new component. It has five different areas where you can drag new components. You cannot write anything directly in any of those areas. You only are allowed to drag and drop new components.

There is a limit with these components: Only one component can be dragged in the top, bottom, right, left, or center area. If you need more components in any of the areas, you should drag another panel (such as the HTML Panel component) and use it as the container for all the components in that area. We do this in the example:

- ▶ Select the component and using the Format tab of the Attributes view change its horizontal alignment to **Center** and the background color to **Silver**. In the All tab, set its border to **1**.
- ▶ Drag and drop an Output component in each of the spaces near to the border. Enter **Top side of the panel** for the Value property of the Output component in the top area. Enter **Bottom side of the panel** for the Value property of the Output component in the bottom area. Enter **Left side of the panel** for the Value property of the Output component in the left area. Enter **Right side of the panel** for the Value property of the Output component in the right area.
- ▶ As we only can drop an UI component in each area, drop a Panel - Group Box component into the center area of the panel in the page. Select *HTML Panel* in the pop-up window and click *OK*. A new panel appears in the center of the parent panel. Now you can drop items inside that panel.

- ▶ Drag a new Image component inside the HTML Panel component. Browse the project and select `WebContent\images\redbook.gif` or import it from `\labscode\images\redbook.gif`.
- ▶ Drag a new Output component inside the HTML Panel component. Enter **Center of the panel, with an Image** for the value property of the Output component. Set the cursor at the end of the Output component and press *Enter* to insert a line break.
- ▶ Save and test the page.

Important: When using several components inside a panel that is inside one of the areas of the Snap To Border panel you have to ensure that *all* the components are *inside* the inner panel. There can only be one control inside a facet tag, otherwise you get an error at runtime.

How it works

When you create a new Snap To Border panel, five facet components are created for you. Each facet component represents one of the five areas where you can add components. The source code for a new panel is shown here:

```
<hx:panelLayout styleClass="panelLayout" id="layoutId">
  <f:facet name="body"></f:facet>
  <f:facet name="left"></f:facet>
  <f:facet name="right"></f:facet>
  <f:facet name="bottom"></f:facet>
  <f:facet name="top"></f:facet>
</hx:panelLayout>
```

Each facet tag in the group has the name of the area it represents. The limitation is that you can only put one component as child for each facet. However, you can use a panel inside the facet component and place as many components as you want into the panel.

When this panel is rendered as an HTML is created with all the properties according to the definition of the panel. The contained components are rendered inside the table.

Panel Group Box - List: <hx:panelBox>

This component creates a container in which you can group other components. This panel has a list layout, that is, it organizes all the components in a list.

You can drop as many components as you want into this panel. The panel maintains its list layout no matter how many components you drop in it. A horizontal scroll bar is displayed as needed.

Attributes

Table 4-51 shows the attributes of the <hx:panelBox> component.

Table 4-51 Panel Group Box - List attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment for the component.
bgcolor	Specifies the background color for the component.
border	Specifies the width of the border.
cellpadding	Specifies the amount of space between the border of the cell and its contents. If the value of this attribute is a pixel length, all four margins should be this distance from the contents.
cellspading	Specifies how much space the browser should leave between the left side of the table and the left-hand side of the leftmost column, the top of the table and the top side of the topmost row, and so on for the right and bottom of the table.
height	Specifies the height of the component.
layout	Specifies the layout as horizontal or vertical.
valign	Specifies the vertical position of data within a cell. Values include top, middle, bottom.
width	Specifies the width of the component.

Usage

This is a simple example of using the List panel component:

- ▶ Create a JSF page and name it **panelBox.jsp**. Change the default text and enter **Example of Group Box - List component**.
- ▶ Drag a Panel - Group Box component from the palette. In the pop-up window, select *List* and click *OK*.
- ▶ A new component appears and you can now drag components into it. Drag two Image components and two Output components. Enter **Redbook logo** in the Value attribute of the first Output component. Enter **California license plate** in the value attribute of the second Output component. Select (or import from \labscode\images) the file **redbook.gif** for the first Image component. Select the **license.gif** for the second Image component.

- ▶ Select the Panel Box - List component and using the Attributes view do the following:
 - In the Format tab, select **Center** for horizontal alignment. Select **Silver** for the background color. Select **5** pixels for padding and spacing.
 - In the items tab, order the components: text1, imageEx1, text2, and imageEx2. See what happens in the Design view with the panel.
- ▶ Save and test the page.

How it works

This component is a panel that can contain other components. It uses a list layout to display the components. You can order all the items in the list by using the Attributes view.

This component is rendered as an HTML table with only one row. The table is rendered according to the properties the panel has, such as cell padding and alignment.

Panel Menu Bar: <hx:panelActionBar>

This component inserts a panel that places commands into a menu bar. You can drag and drop buttons, hyperlinks, and horizontal rules into the panel or add them in the Attributes view of the component. You can also add a sub-menu bar within a menu bar.

Attributes

Table 4-52 shows the attributes of the <hx:panelActionBar> component.

Table 4-52 Panel Menu Bar attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
autoseparate	Specifies whether a separator is automatically inserted between each contained components.
contentClass	Alternative CSS class names associated with the component. The situation in which these are used can be determined from the names.
height	Specifies the height of the component.
layout	Specifies the layout as horizontal or vertical.

Attributes	Description
nestedExpanded	Expand or collapse the child panelActionBar. Applies only if one panelActionBar is nested inside another.
nestedImagePosition	Places the image before or after the label of the child panelActionBar. Applies only if one panelActionBar is nested inside another.
nestedIndent	Amount by which the submenu of the child panelActionBar is indented. Applies only if one panelActionBar is nested inside another.
nestedTitleImage	Image to be displayed in the title of the child panelActionBar. Applies only if one panelActionBar is nested inside another.
nestedTitleText	Label of child panelActionBar. Applies only if one panelActionBar is nested inside another.
separatorSize	Specifies the pixel width of the rendered table cell containing the separators.
separatorStyle	Assigns a defined CSS style sheet style for the separator. The CSS style sheet file must be linked into the component's JSP before it can be used.
width	Specifies the width of the component.

Usage

Here is a simple example of how to use a Panel Menu Bar component to create a menu:

- ▶ Create a JSF page and name it **panelBar.jsp**. Change the default text and enter **Example of Panel - Menu Bar**.
- ▶ Drag and drop a Panel - Menu Bar component from the palette. A new area appears on the screen.
- ▶ Select the Items tab in the Attributes view. You can use the buttons to add new components to the menu bar, or you drag components from the palette into the menu bar (in which case they do not appear in the items list).
Add two Command - Hyperlink components, a separator and a sub-panel to the panel. Change the labels for the hyperlinks, the button, and the sub-panel (by clicking in the Label column in each component). Enter **First hyperlink**, **Second hyperlink**, and **Sub-Panel** respectively.
- ▶ Select the sub-panel and in the Attributes view select the Items tab. Add two hyperlinks and change their labels to **First sub-hyperlink** and **Second sub-hyperlink** respectively (Figure 4-34 left).

- Create a JSF page and name it **targetMenuBar.jsp**. This page is the target for all the links and the button in the menu that we create. Change the default text to **Example of Panel - Menu Bar**. In the Page Data view, define a parameter named **source**. Drag and drop the parameter into the page. Delete the Display Errors component that is automatically created. Add a Command - Button, name it **Back**, and add a navigation rule with alias **back** to the `panelBar.jsp`, and add the Quick Edit code `return "back";` (Figure 4-34 right).

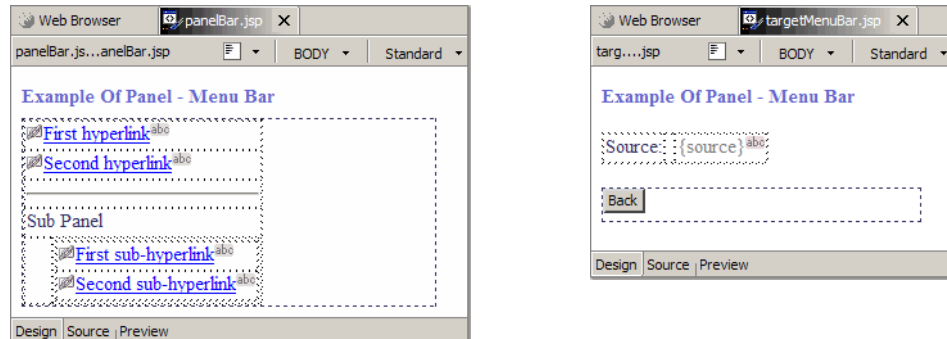


Figure 4-34 Example of a Panel - Menu Bar component

- Select the `panelBar.jsp` page again. Select the first Command-Hyperlink component and, using the Attributes view, define a new navigation rule to `targetMenuBar.jsp` page with the **go** alias.
- For each Command-Hyperlink component:
 - Insert the line **return "go";** in the action code in the Quick Edit view.
 - Using the Attributes view, Parameters tab, define a parameter and name it **source**. Put the label of the hyperlink as the value for the parameter.
- Save and test the page.

We have a menu with two links. This menu has a submenu (it can be expanded by clicking on its label). We have defined all the links to go to the other JSP page and we are passing a parameter to this page.

How it works

At design time a `<hx:panelActionBar>` tag is created. All the components you add using Attributes view are included inside this tag. Only a subsets of tags are allowed to be included in this tag.

When rendered, an HTML table representing the menu is created. The necessary JavaScript code is generated to get the expand and collapse behavior.

All the values for the components in the menu are accessed using JavaScript functions.

Panels - Tabbed: <odc:tabbedPanel>

This component takes advantage of the possibility of extending the JSF framework to make more complex components. The Tabbed Panel component is one of the IBM extension components that allow you to create richer applications.

This component uses the same tag library as the JSF Client components. Refer to Chapter 13, “Faces Client framework” on page 439 for more details.

To use this component, the correct namespace must be include. This is done for you when dragging and dropping a new Tabbed Panel component into a JSF page. The namespace is:

```
<%@taglib uri="http://www.ibm.com/jsf/BrowserFramework" prefix="odc"%>
```

The Tabbed Panel component is represents a set of overlapping panels in the same area of the page. To access the content of each panel you have tabs and *Next* and *Back* buttons. Either the tabs and the buttons can be hidden. If you hide the buttons, you have a pure Tabbed Panel. If you hide the tabs, you have the functionality of a wizard.

This component should be used when you have to show or retrieve a big amount of information in a limited space.

As this component is a container, you can use any of the other components shown in this chapter to show or get information from a user of the Web application.

Attributes

Table 4-53 shows the attributes of the <odc:tabbedPanel> component.

Table 4-53 Tabbed Panel attributes

Attributes	Description
common	id, binding, rendered, styleClass
height	The height of the component
numOfTabs	This attribute specifies the number of tabs that must be shown at the same time.
showBackNextButton	If the Next and Back buttons are shown
showTabs	If the tabs are shown.

Attributes	Description
slantActiveLeft	Set the left bevel length for the active tab. The default value is 0.
slantActiveRight	Set the right bevel length for the active tab. The default value is 0.
slantInactiveLeft	Set the left bevel length for all the inactive tabs. The default value is 0.
slantInactiveRight	Set the right bevel length for all the inactive tabs. The default value is 0.
variableTabLengh	If <code>false</code> , all the tabs have the same length. If <code>true</code> , each tab can have a different length depending on the displaying text.
width	The width of the component.

Events

Table 4-54 shows the events of the `<odc:tabbedPanel>` component.

Table 4-54 Tabbed Panel Quick Edit events

Event	Description
oncancel	This event is executed when the <i>Cancel</i> button is clicked.
onfinish	This event is executed when the <i>Finish</i> button is clicked. The DOM object corresponding to the form is passed as parameter. This method must return a boolean result and depending on this value, the form is submitted or not (the submission is done automatically).
onpanelenter	This event is executed when any panel in the Tabbed Panel component is entered. This is useful to add business logic or verify information input by the user. The <code>id</code> of the panel is passed as parameter.
onpanelexit	This event is similar as the previous one, but is executed when a panel is exited. This event is very useful when you are using the Tabbed Panel component as a wizard. The <code>id</code> of the panel is passed as parameter. This method can return <code>null</code> (the transition to the next panel is forbidden) or a panel <code>id</code> (to transit to).

If you click inside any of the panel the Tabbed Panel component holds, you find the following events for the single panel (Table 4-55).

Table 4-55 Individual panel Quick Edit events

Event	Description
onenter	This event is called when the selected panel is entered. This event is like the 'global' <i>onpanelenter</i> event, except that it is specific for a panel.
onexit	This event is called when the selected panel is exited. This event is like the 'global' <i>onpanelexit</i> event, except that it is specific for a panel.

Usage

Now we show an example of this component:

- ▶ Select the Java Resources folder in the Web application. Create a Java package and name it **itso.jsf**.
- ▶ Import the file `\\labscode\dev-component\Employee.java`. This file represents a simple example of employee information that we use in our example.
- ▶ Create a JSF JSP page and name it **tabbedPanel1.jsp**. Studio may prompt you if you want to copy the new resources into the project. Click **Yes**.
- ▶ Replace the default text by **Example of Tabbed Panel**.
- ▶ Create a JSF JSP page and name it **tabbedPanel2.jsp**. Close the file for now.
- ▶ Select the file `tabbedPanel1.jsp` and drag and drop a Panels - Tabbed component from the palette. A new Tabbed Panel is created with two tabs (representing two panels) for you. Now we use the Attributes view to change its appearance:
 - In the Basic tab, set the width property to **275** pixels and the height property to **200** pixels.
 - Select *Use the Finish button for submission*. We use this feature to send all the data to the server.
 - You can notice in the Basic tab that you can hide the tabs and set the number of them that would be displayed.
 - Select the Panel List tab. Set the label of the first tab (named `bfpanel1`) to **Employee info**. and the label for the second tab (named `bfpanel2`) to **Job info**. For the `bfpanel2` tab, select `true` as the value for Show finish/cancel (Figure 4-35).

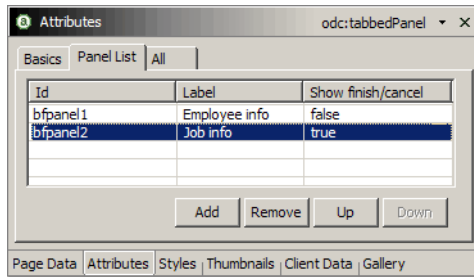


Figure 4-35 Setting properties for the tabs of the Tabbed Panel component

- In the Page Designer select the Employee info tab of the Tabbed Panel Component. Insert an HTML table with four rows and two columns in the panel. Set the table border to 0. Drag and drop the following components into the table:
 - Row 1, column 1: Output component. Set its value to **Name:**.
 - Row 1, column 2: Input component. Set its name property to **name**. Set its width property to 20 characters. In the Validation tab, select *Value is required*.
 - Row 2, column 1: Output component. Set its value to **Surname:**.
 - Row 2, column 2: Input component. Set its name to **surname**. Set its width to 20 characters. In the Validation tab, select *Value is required*.
 - Row 3, column 1: Output component. Set its value to **Age:**.
 - Row 3, column 2: Input component. Set its name to **age**. Set its width property to 2 characters. In the Format tab, select **Number** in the Enter combobox and more properties appear. Select *Integer only*. In the Validation tab, select *Value is required*. In the Behavior tab, enter 2 for the Maximum length property.
 - Row 4, column 1: Output component. Set its value to **Gender:**.
 - Row 4, column 4: Combobox component. Set its name to **gender**. In the Choices tab, add two new text items (click twice in the *Add Text Item* button). Enter **male** and **female** for their labels and **Male** and **Female** for their values, respectively.
- Select the Job info. tab of the Tabbed Panel Component. Insert an HTML table with two rows and two columns in the panel. Set the table border to 0. Drag and drop the following components into the table:
 - Row 1, column 1: Output component. Set its value to **Position:**.
 - Row 1, column 2: Input component. Set its name property to **position**. Set its width property to 20 characters.

- Row 2, column 1: Output component. Set its value to **Details:**.
- Row 2, column 2: Input-Text Area component. Set its name to **details**. Set its width to **25** characters and its height to **4** lines.
- ▶ Drag a Display Errors component from the palette to the bottom of the Tabbed Panel component. This component shows all the validation or conversion errors.

You can see the final design for the two tabs in Figure 4-36. Notice that you have *Next* and *Back* buttons to navigate through the panels and, in the second panel, you have the *Finish* and *Cancel* button to send the information to the server or cancel the operation.

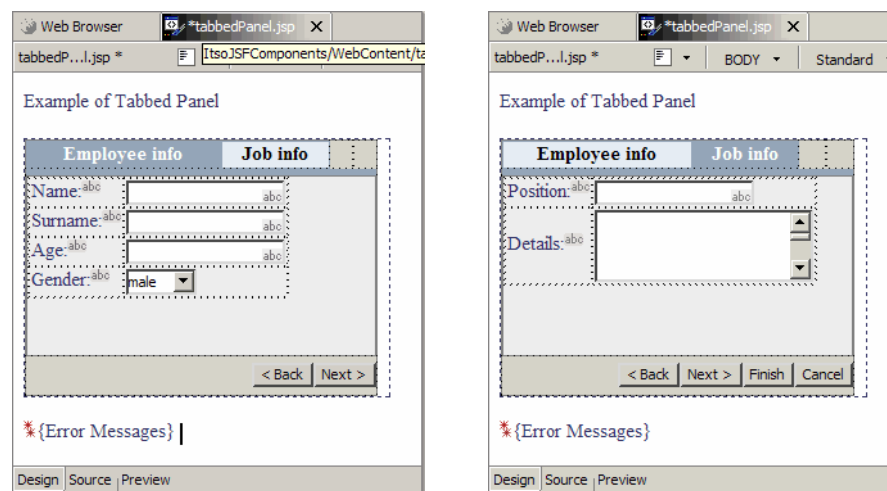


Figure 4-36 Design of the Tabbed Panel component

Now we include the Java class to get all the information from the panel, and show this information in a second page:

- ▶ Using the Page Data view, define a new JavaBean. Enter **employee** for its name and select the Java class **itso.jsf.Employee** for its class. Select *Make this JavaBean reusable*. In the Scope combo box, select *request*. Click *OK*.
- ▶ Now bind the JavaBean properties to the information in the panels in the Tabbed Panel component:
 - Bind the name property to the name Input component.
 - Bind the surname property to the surname Input component.
 - Bind the age property to the age Input component.
 - Bind the jobRole property to the position Input component.
 - Bind the jobDetails property to the details Input - Text Area component.
- ▶ Select the *Finish* button and enter the action code `return "showInfo";`.

- ▶ Using the Navigation tab in the Attributes view, define a navigation rule to the **tabbedPanel12.jsp** page using the alias **showInfo**.
- ▶ Save the file.
- ▶ Open the page **tabbedPanel12.jsp** page. Replace the default text by typing **Example of Tabbed Panel**. Insert an Output component with the text **This is the information you entered:**. Insert a new line.
- ▶ Using the Page Data view, add a JavaBean. In the pop-up window, select *Add existing reusable JavaBean* and select the **employee** JavaBean we defined earlier. Click **OK**.
- ▶ Drag and drop the JavaBean in the **tabbedPanel12.jsp** page. In the pop-up window, select **Displaying fields (read-only)** option, reorder the information and enter the values for the labels as follows (Figure 4-37):
 - For the name property, enter **Name:**.
 - For the surname property, enter **Surname:**.
 - For the age property, enter **Age:**.
 - For the gender property, enter **Gender:**.
 - For the jobRole property, enter **Job position:**.
 - For the jobDetails property, enter **Job details:**.
 - Click **OK**.

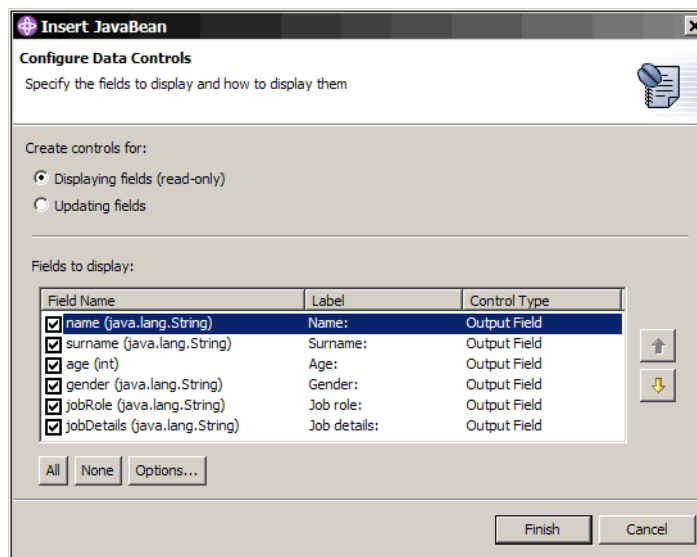


Figure 4-37 Designing the employee result display

- ▶ Drag and drop a Command-Button component below the table. Select it and in the format tab of the Attributes view enter **Back** for its label. In the Quick Edit

view enter return "back"; and define a navigation rule named **back** to the `tabbedPanel.jsp` (Figure 4-38).

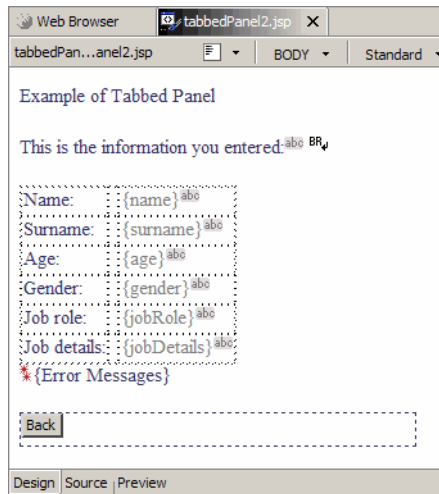


Figure 4-38 Design of the `tabbedPanel2.jsp`

Now you can test the `tabbedPanel.jsp` page. Enter values for the attributes of the first panel, click the *Next* button and enter values in the fields of the second panel. Click *Finish*. The page with the results should be displayed.

How it works

When you create a Tabbed Panel component, Studio add a new `<odc:tabbedPanel>` tag is added. This tag is the main container for all the panel components.

For each panel that is added to the tabbed panel, a `<odc:bfPanel>` tag is added. This tag represents each panel accessible using its tab.

After the panels, four buttons are added to the panel. Each button is added inside a `<f:facet>` tag from the JSF Core library. These buttons are the *Back*, *Next*, *Cancel* and *Finish* buttons. By default, the *Cancel* and *Finish* buttons are hidden in every panel.

The code generated when you include a new Tabbed Panel is the following:

```
<odc:tabbedPanel id="tabbedPanel1" width="300" slantActiveRight="4"
  styleClass="tabbedPanel" showTabs="true" slantInactiveRight="4"
  variableTabLength="false" showBackNextButton="true" height="300">
  <odc:bfPanel id="bfpanel1" name="Tab1"
    showFinishCancelButton="false"></odc:bfPanel>
  <odc:bfPanel id="bfpanel2" name="Tab2"
```

```

        showFinishCancelButton="false"></odc:bfPanel>
    <f:facet name="back">
        <hx:commandExButton type="submit" value="&lt; Back"
            id="tabbedPanel1_back" style="display:none"></hx:commandExButton>
    </f:facet>
    <f:facet name="next">
        <hx:commandExButton type="submit" value="Next &gt;"
            id="tabbedPanel1_next" style="display:none"></hx:commandExButton>
    </f:facet>
    <f:facet name="finish">
        <hx:commandExButton type="submit" value="Finish"
            id="tabbedPanel1_finish"
style="display:none"></hx:commandExButton>
    </f:facet>
    <f:facet name="cancel">
        <hx:commandExButton type="submit" value="Cancel"
            id="tabbedPanel1_cancel"
style="display:none"></hx:commandExButton>
    </f:facet>
</odc:tabbedPanel>

```

Notice the buttons inside the `<f:facet>` tags and the `<odc:bfPanel>` tags.

When the component is rendered to the browser, all the necessary `<div>` tags are created to display the component as a Tabbed Panel. In addition to the HTML code necessary to display the panels correctly, all the necessary JavaScript code is added to handle the events, the navigation, and the access through tabs.

Rich Text Area: `<r:inputRichText>`

The Rich Text Area component allows you to have a text editor in JSF Web pages. This editor is not a plain text editor because it has additional features:

- ▶ You can format the text using different fonts and sizes.
- ▶ You can include tables in the texts.
- ▶ You can include links in the texts.
- ▶ You can include bulleted and numbered list.

All the formatting is represented using HTML code. Because of this, you can use the editor as an HTML editor.

When you include the Rich Text Area component in JSF pages, WebSphere Studio include all the necessary libraries for you. A new tag library is include in the page:

```
<%@taglib uri="http://www.ibm.com/jsf/rte" prefix="r"%>
```

Now you can use the tag for the rich text editor:


```
<r:inputRichText height="350" width="700" id="richTextEditor1">
</r:inputRichText>
```

Attributes

Table 4-56 shows the attributes of the `<r:inputRichText>` component.

Table 4-56 Rich Text Area attributes

Attributes	Description
<i>common</i>	id, binding, value, rendered
height	The height of the component
readOnly	This attribute specifies if text can be edited using this component.
width	The width of the component.

Usage

This component is as easy to use as powerful. We now use it in a simple example:

- ▶ Create a Faces JSP page and name it **richTextEditor.jsp**. Replace the default text by **Example of Rich Text Editor**.
- ▶ Drag and drop a Rich Text Area component from the palette. Studio may prompt you if you want to copy the new resources into the project. Click **Yes**.
- ▶ A new Rich Text Area component is added to the page. Using the Attributes view, change its name to **richText**. Change its width property to **400** pixels and its height property to **200** pixels.

We use a request scope variable to store all the text entered in the rich text area. Then we can see the result text by dragging the variable back to the page.

- ▶ Drag and drop a Command - Button component under the rich text area. This component is used to send the data to the server.
- ▶ Using the Page Data view, define a new request scope variable. In the pop-up window, enter **text** for the name of the variable and `java.lang.String` for its type. Click **OK**.
- ▶ Bind this new variable to the Rich Text Area component (drag and drop it over the Rich Text Area component).
- ▶ Drag and drop the new variable into the page. In the pop-up window, change its label to **Text entered in the rich text area:**. Click **OK**.
- ▶ In the Attributes view set the height to **160**.
- ▶ Save and test the page (Figure 4-39).

How it works

When you use the Rich Text Area component, all the necessary JavaScript code is generated and included in the page to render the editor. If the editor is bound to an object in the page, a JavaScript is used to put the correct data inside the form to send it to the server. This component holds HTML text and send it to the server as is.

Media - Generic A/V Player: <hx:playerGenericPlayer>

This component creates a media player in the Web page and plays a file using a plug-in. This plug-in is determined by the mime type of the file the component has to play.

Attributes

Table 4-57 shows the attributes of the <hx:playerGenericPlayer> component.

Table 4-57 Generic A/V Player attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment for the component.
alt	Alternate text for the component.
archive	This is a space-separated list of URIs for archives containing resources relevant to the object.
border	Set the border for the component.
codebase	Specifies the base path used to resolve relative URIs specified by other attributes in the component.
height	Set the height of the component.
hidden	Specifies whether or not to hide the player on a web page.
hspace	Specifies the amount of white space in pixels to be inserted to the left and right of the component.
contentType	Used for declaring the media MIME type when src is bound to binary data.
playerAttributes	For adding a list of additional player attributes/parameters, the list should be a comma separated list of name=value pairs.

Attributes	Description
pluginsPage	Specifies URL containing player code for download, in the case the player is not already installed.
pluginUrl	Specifies location of Java Archive (JAR) file containing player code for download, in the case the player is not already installed.
src	Media file to be played.
standby	Specifies a message that a browser may render while loading the component's implementation and data.
tabindex / target	Specifies the position of the component in the tabbing order in the JSP.
vspace	Specifies the amount of white space in pixels to be inserted to the top and bottom of the component.
width	Specifies the width of the component.

Events

Table 4-58 shows the events of the `<hx:playerGenericPlayer>` component.

Table 4-58 Generic A/V Player Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Usage

This component should be use in the case you want to play media files in the Web application. We use an example of a flash player in the next section.

Media - Macromedia Flash Player: `<hx:playerFlash>`

This component represent the Macromedia Flash player and it is designed to play Flash files.

Attributes

Table 4-59 shows the attributes of the `<hx:playerFlash>` component.

Table 4-59 Macromedia Flash Player attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment for the component.
alt	Alternate text for the component.
archive	An a space-separated list of URIs for archives containing resources relevant to the object.
autostart	Whether or not start playing automatically.
border	Set the border for the component.
codebase	Specifies the base path used to resolve relative URIs specified by other attributes in the component.
height	Set the height of the component.
hidden	Specifies whether or not to hide the player on a web page.
hspace	Specifies the amount of white space in pixels to be inserted to the left and right of the component.
loop	Specifies whether the movie repeats indefinitely or stops when it reaches the last frame. Default is true.
menu	Specifies the menu type. If true, it displays the full menu; if false, it displays a shorter menu.
mimeType	Used for declaring the media MIME type when src is bound to binary data.
playerAttributes	For adding a list of additional player attributes/parameters, the list should be a comma separated list of name=value pairs.
quality	Specifies the player quality: low, autolow, autohigh, medium, high and best.
salign	Specifies alignment, values can be l, r, t, b.
scale	Specifies scale: showall, No Border, Exact Fit.
src	Media file to be played.
standby	Specifies a message that a browser may render while loading the component's implementation and data.

Attributes	Description
tabindex / target	Specifies the position of the component in the tabbing order in the JSP.
vspace	Specifies the amount of white space in pixels to be inserted to the top and bottom of the component.
width	Specifies the width of the component.
wmode	Specifies the window mode: window, opaque, transparent.

Events

Table 4-60 shows the events of the `<hx:playerFlash>` component.

Table 4-60 Macromedia Flash Player Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Usage

This example shows how to use the Flash Player component:

- ▶ Create a folder named **resources** under WebContent. Select the folder and import the `\labscode\dev-component\redbooks.swf` file into it.
- ▶ Create a Faces JSP file and name it **flashPlayer.jsp**. Replace the default text with **Flash Player Example**.
- ▶ Drag and drop a Media Player component from the palette. In the pop-up window, select *Macromedia Flash Player* and click *OK*.
- ▶ Select the Flash Player component and, using the Basic tab of the Attributes view set the following:
 - In the File attribute, browse and find the file `resources\redbooks.swf`.
 - Set the width to 70%.
 - Set the height to 70%.
- ▶ Save and test the page to watch the Flash Player animate the Redbooks logo.

Media - Macromedia Shockwave Player: `<hx:playerShockwave>`

This component is used to play Macromedia Shockwave files in a Web application. This component should be used as the same way as the Flash Player component.

Attributes

Table 4-61 shows the attributes of the <hx:playerShockwave> component.

Table 4-61 Macromedia Shockwave Player attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment for the component.
alt	Alternate text for the component.
archive	An a space-separated list of URIs for archives containing resources relevant to the object.
bgcolor	Specifies the color of the movie rectangle before the movie itself appears.
border	Set the border for the component.
codebase	Specifies the base path used to resolve relative URIs specified by other attributes in the component.
height	Set the height of the component.
hspace	Specifies the amount of white space in pixels to be inserted to the left and right of the component.
contentType	Used for declaring the media MIME type when src is bound to binary data.
playerAttributes	For adding a list of additional player attributes/parameters, the list should be a comma separated list of name/value pairs.
src	Media file to be played.
standby	Specifies a message that a browser may render while loading the component's implementation and data.
swRemote	Controls all Shockmachine functionality and has separate subparameters.
swStretchAlign	Specifies horizontal alignment when swStretchStyle is set to meet or stage. Value can be Left, Center or Right.
swStretchStyle	Specifies the stretch style to use for the Shockwave media, values are: none—no stretching, meet—preserve proportions, fill—stretch to fill, stage—expand stage size

Attributes	Description
swStrechVAlign	Specifies vertical alignment when swStretchStyle is set to meet or stage. Value can be Top, Center, or Bottom.
tabindex / target	Specifies the position of the component in the tabbing order in the JSP.
vspace	Specifies the amount of white space in pixels to be inserted to the top and bottom of the component.
width	Specifies the width of the component.

Events

Table 4-62 shows the events of the <hx:playerShockwave> component.

Table 4-62 Macromedia Shockwave Player Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Usage

You use this component to display Macromedia Shockwave files in a Web applications. You can customize the way the player shows the files by using all the attributes of the component.

Its usage is similar to the usage of the Flash Player (see “Media - Macromedia Flash Player: <hx:playerFlash>” on page 178).

Media - RealOne Player: <hx:playerRealPlayer>

This component allows you to include Real Media content in a Web application. You can use either audio files and video files.

Attributes

Table 4-63 shows the attributes of the <hx:playerRealPlayer> component.

Table 4-63 RealOne Player attributes

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment for the component.

Attributes	Description
alt	The alternate text for the component.
archive	Specifies a space-separated list of URIs for archives containing resources relevant to the object.
autostart	Specifies whether start the clip automatically.
backGroundColor	Sets the background color for the image window. When a clip includes transparent regions, the background color also shows through these areas.
border	Set the border of the component.
codebase	Specifies the base path used to resolve relative URIs specified by other attributes in the component.
console	When the same console name is specified for multiple control on a Web page, this parameter can be used to enable these controls to manage playback of a single embedded presentation.
controls	Embeds the specified RealPlayer control on the Web page.
height	Specifies the height of the component.
hspace	Specifies the amount of white space in pixels to be inserted to the left and right of the component.
loop	Specifies whether playback of the clip should continue or loop, indefinitely.
mimeType	Used for declaring the media MIME type when src is bound to binary data.
playerAttributes	A list of comma separated list of name/value pairs for the player.
shuffle	Specifies whether all unplayed clips in a presentation, should be played back in a random order.
src	Media file to be played.
standby	Specifies a message that a browser may render while loading the component's implementation and data.
tabindex / target	Specifies the tab order of the component in the page.
vspace	Specifies the amount of white space in pixels to be inserted to the top and bottom of the component.
width	Specifies the width of the component.

Events

Table 4-64 shows the events of the `<hx:playerRealPlayer>` component.

Table 4-64 *RealOne Player Quick Edit events*

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Usage

Its usage is similar to the usage of the Flash Player (see “Media - Macromedia Flash Player: `<hx:playerFlash>`” on page 178).

Media - Windows Media Player: `<hx:playerMediaPlayer>`

This component is used to include Windows Media files in JSF Web applications.

Attributes

Table 4-65 shows the attributes of the `<hx:playerMediaPlayer>` component.

Table 4-65 *Windows Media Player attributes*

Attributes	Description
<i>common</i>	id, value, style, styleClass, rendered, immediate, binding, accesskey, dir, disabled, lang, title
align	Specifies the alignment for the component.
alt	Alternate text for the component.
archive	This is a space-separated list of URIs for archives containing resources relevant to the object.
autostart	Specifies if the component must start playing automatically.
baseUrl	Specifies the base URL in which the media player will be operating.
border	Set the border for the component.
codebase	Specifies the base path used to resolve relative URIs specified by other attributes in the component.
enableContextMenu	Specifies whether the context menu appears when the user clicks the right mouse button. Default is <code>true</code> .
height	Set the height of the component.

Attributes	Description
hidden	Specifies whether or not to hide the player on a web page.
hspace	Specifies the amount of white space in pixels to be inserted to the left and right of the component.
contentType	Used for declaring the media MIME type when src is bound to binary data.
playCount	Specifies the number of times a clip plays.
playerAttributes	For adding a list of additional player attributes/parameters, the list should be a comma separated list of name=value pairs.
src	Media file to be played.
standby	Specifies a message that a browser may render while loading the component's implementation and data.
stretchToFit	Specifies whether maintain the aspect ratio for the clip.
tabindex / target	Specifies the position of the component in the tabbing order in the JSP.
uiMode	Specifies the appearance of the embedded Windows Media Player. Values can be none, mini, full or invisible.
volume	Volume, in hundredths of decibels ranging from -10,000 to 0. Default is -600.
vspace	Specifies the amount of white space in pixels to be inserted to the top and bottom of the component.
width	Specifies the width of the component.

Events

Table 4-66 shows the events of the <hx:playerMediaPlayer> component.

Table 4-66 Windows Media Player Quick Edit events

Event	Description
<i>common</i>	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Usage

Its usage is similar to the usage of the Flash Player (see “Media - Macromedia Flash Player: <hx:playerFlash>” on page 178).

Data Table component extensions

IBM provides a number of extensions for the Data Table component to make it more useful for end user interaction:

- ▶ **Table header**—A table header area can be added to the data table. This header area can be populated with other components, for example an Output component holding the heading of the table.
- ▶ **Table footer**—A table footer area can be added. This footer area is most often used for paging of a data table with many rows, using paging controls.
- ▶ **Row action support**—Row action support makes the row active so that the user can click anywhere in the row to invoke server-side action logic. A column with a hyperlink icon is added to the table. You can attach action logic to the hyperlink and pass parameters holding some of the data of the selected row to the action logic. The extra column is not displayed at runtime.

For an example of row action support refer to “Implementing paging and transaction details” on page 220.

- ▶ **Row select support**—Row select support adds a column with check boxes to the table. The user can select rows and pass the selected row indices to the server for processing of those rows.

For an example of row edit support refer to “Account maintenance using row select support” on page 374.

- ▶ **Row edit support**—Row edit support adds a column into which you drag and drop other column values for editing. At runtime a column with an ***edit*** hyperlink is added to each row of the data table. When the edit hyperlink is clicked, a panel with selected row values is opened for editing. A Submit button is usually added to commit the changed data to the server.

For an example of row edit support refer to “Customer maintenance using row edit support” on page 372.

- ▶ **Row category support**—Row category support enables the sorting of the rows by one column and present the list of rows with collapsed or expanded sections attached to the sort column value. Expanding and collapsing of sections is done at the client through JavaScript. A column with an arrow is added to the data table to bind to the sort value.

For an example of row edit support refer to “Account listing using row category support” on page 376.

- ▶ **Paging**—A number of paging controls using different styles are provided to page through large data tables (Figure 4-41):



Figure 4-41 Paging controls

The paging controls in Figure 4-41 are (from left to right):

- **Simple** pager with a drop-down menu providing *Next* and *Previous*.
- **Deluxe** pager with arrows for forward, backward, end, and start actions.
- **Goto** pager with a drop-down menu providing page numbers
- **Web style** pager with a list of page numbers that are hyperlinks
- **Page statistics** displaying the number of pages total and visited

For paging you specify the number of items per page. Adding a paging control automatically creates a footer area. For an example of paging support refer to “Implementing paging and transaction details” on page 220.

Usage

The extended functions of the Data Table component are selected in the Attributes view (Figure 4-42).

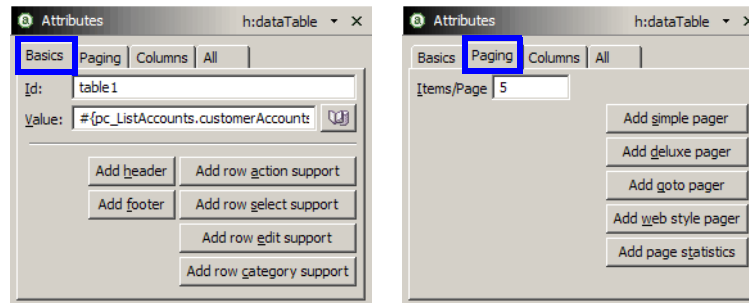


Figure 4-42 Extended functions of the data table

Note that for one data table you can only select one of the row functions, but you can select multiple paging functions.

Complete solution

We provide the complete solution for these examples in the additional material in the file `\\abscode\\solution\\ItsoJSF04Comp.ear`. See “Importing a solution enterprise application” on page 548 for instructions.



Part 2

JSF application development

In Part 2 we describe how to develop JSF Web applications and how to integrate JSF with others frameworks such as Jakarta Struts.

Chapter 5 provides a step-by-step example of designing and implementing a JSF application with database support.

Chapter 6 covers the design principles and some common JSF patterns in Web development.

Chapter 7 describes how to utilize Web Services in the JSF framework and provides an example application.

Chapter 8 compares and contrasts JSF and Struts, providing guidelines for development utilizing these two Web application frameworks.



JSF banking application

This chapter describes how to implement a simple banking application using a JSF frontend with an existing backend database implementation.

Banking application overview

The sample banking application has been described in two previous Redbooks:

- ▶ *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- ▶ *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957

For this redbook we reuse parts of the previous implementations.

JSF banking application architecture

The architecture of the application is based on model-view-controller (MVC) principles and is shown in Figure 5-1.

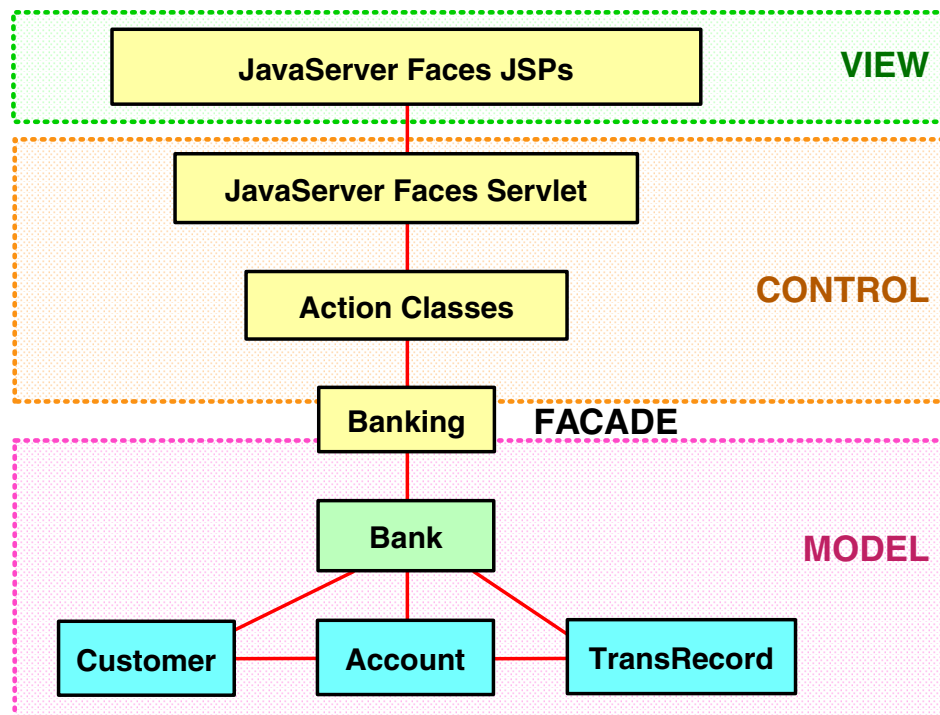


Figure 5-1 Banking application architecture

Model

The model is represented by three model classes, Customer, Account, and TransRecord, and a Bank interface that provides access to the model classes.

Bank

The Bank interface provides these functions to access the model classes:

- ▶ `getCustomer`—return a Customer object by key.
- ▶ `getAccount`—return an Account object by key.
- ▶ `getAccounts`—return an array of Account objects for a customer.
- ▶ `getTransactions`—return an array of TransRecord objects for an account.
- ▶ `deposit`—deposit an amount into an Account. This updates the Account balance and create a TransRecord.
- ▶ `withdraw`—withdraw an amount from an Account. This updates the Account balance and create a TransRecord.
- ▶ `transfer`—transfer an amount from one Account into another Account. This updates the Account balances and create two TransRecord.

The Bank interface can be implemented through a real class, for example, BankEJB (a session EJB) or BankJDBC (direct JDBC access).

Customer

The Customer class has these properties: `id`—identification key (101, 102, and so forth), `title` (Mr, Mrs, Ms), `firstName`, and `lastName`.

Account

The Account class has these properties: `id`—identification key (101-1001, 102-2001, and so forth), `type` (CHECKING, SAVINGS), and `balance`.

TransRecord

The TransRecord class has these properties: `timeStamp`—identification key, `transType` (C=credit, D=debit), and `transAmt` (amount of transaction).

Banking

The Banking class is a facade to make the controller completely independent of the model implementation. The facade provides the same methods as the Bank itself, and invokes the methods of the Bank interface.

Model implementation

The three model classes are mapped directly to tables in a relational database named EJBBANK.

For the purpose of the JSF implementation of the frontend (the view) it does not matter how the model is implemented:

- ▶ The model could be implemented using EJBs, as described in the previous Redbooks. The Bank class is a session bean (BankEJB) and the three model classes are container-managed entity beans that map to the database tables.
- ▶ For this redbook we implemented the Bank class as **BankJDBC** using direct JDBC access to the database. The three model classes are implemented as data transfer objects (DTO). In Chapter 10, “SDO banking application” on page 341, we replace the backend with direct access to the database using Service Data Objects (SDO).

Controller

The controller is implemented using the JSF servlet and action classes. Each action class provides the access to the backend for the JSF JSPs. The action classes are:

- ▶ **ListAccountsAction**—retrieve a customer by key and retrieve the accounts of the customer. Return a JavaBean with a Customer object and an array of Account objects.
- ▶ **AccountDetailsAction**—retrieve the account details after an account is selected.
- ▶ **RefreshAccountAction**—refresh the account details after a banking transaction (deposit, withdraw, transfer).
- ▶ **PerformTransactionAction**—execute a banking transaction (deposit, withdraw, transfer).
- ▶ **ListTransactionAction**—retrieve an array of TransRecord objects for an account.

The action classes are invoked from the action logic associated with JSF command buttons and hyperlinks. The action classes acquire the Banking facade to execute the backend methods.

Exceptions

The action classes handle a number of predefined exceptions, such as CustomerDoesNotExistException, AccountDoesNotExistException, InsufficientFundsException (balance not enough for withdrawal), and BankException (database access problems).

View

The view is implemented using JSF JSPs:

- ▶ **index.jsp**—welcome page to enter a customer key.
- ▶ **listAccounts.jsp**—list the accounts of a customer for selection
- ▶ **accountDetails.jsp**—display account details and perform transactions (list transactions, deposit, withdraw, transfer).
- ▶ **transferResult.jsp**—display account details after a transfer.
- ▶ **listTransactions.jsp**—display the list of transaction records for an account.
- ▶ **transactionDetails.jsp**—display the details of one transaction record (not shown in Figure 5-2).

Implementation details

The structure of the JSF banking application is shown in Figure 5-2.

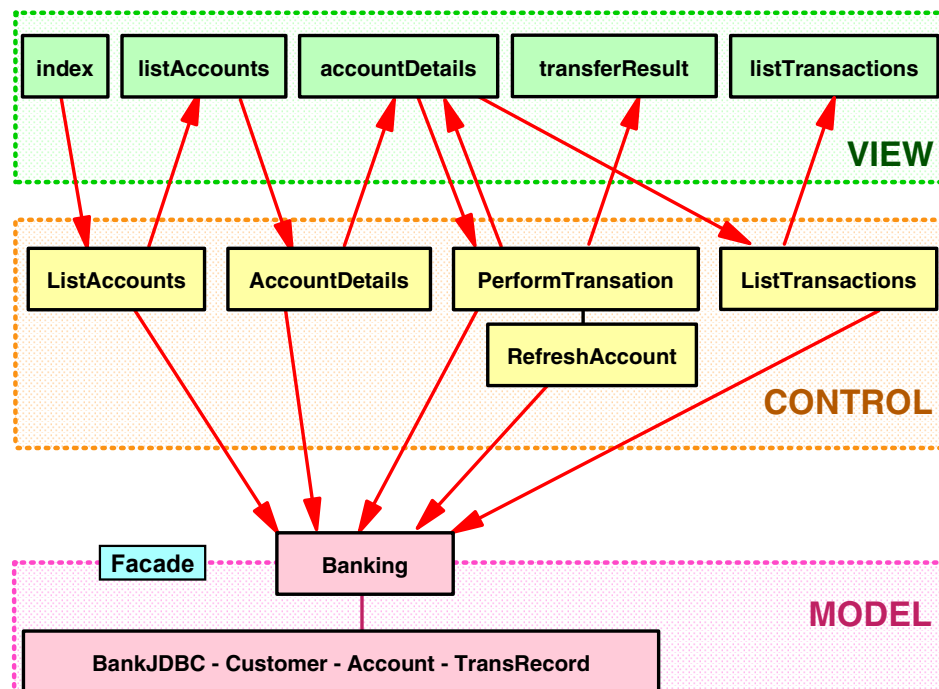


Figure 5-2 JSF banking application

EJBBANK database

The EJBBANK database is composed of eight tables (Figure 5-3).

For the JSF banking application we only use CUSTOMER, ACCOUNT, TRANSRECORD, and CUSTACCT (the relationship between CUSTOMER and ACCOUNT).

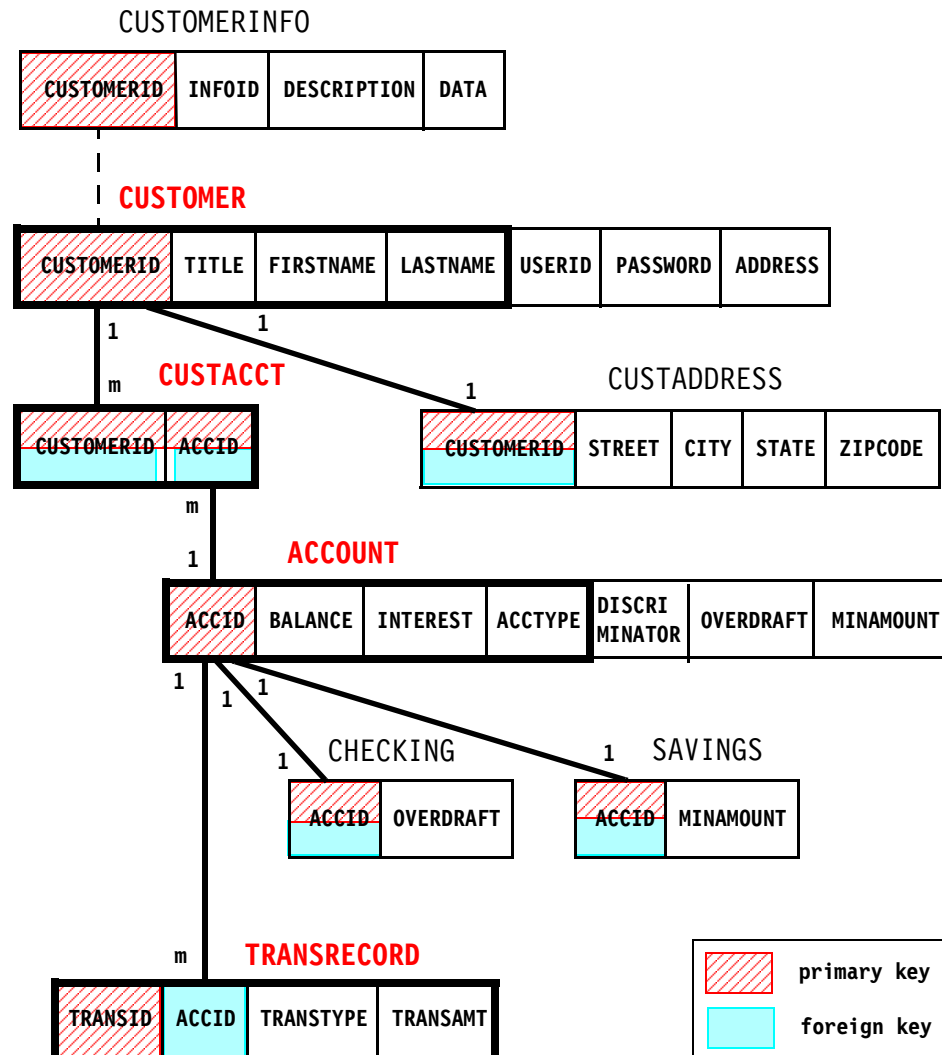


Figure 5-3 EJBBank database and tables

The CUSTOMER and ACCOUNT tables hold more data than we are using in the JSF banking application. In the CUSTOMER table we only use CUSTOMERID, TITLE, FIRSTNAME, and LASTNAME. In the ACCOUNT table we only use ACCID, BALANCE, and ACCTYPE.

Sample data

When testing the code you have to enter valid keys.

- ▶ Customer keys:

- 101 - Schumacher
- 102 - Ferraz
- 103 - Sjostrand
- 104 - Brown
- 105 - Lady
- 106 - Wahli

- ▶ Account keys:

- 101-1001 - checking
- 101-1002 - savings
- 102-2001 - savings
- 102-2002 - checking
- 103-3001 - savings
- 104-4001 - savings
- 104-4002 - checking
- 105-5001 - checking
- 106-6001 - checking
- 106-6002 - savings
- 106-6003 - savings

Preparation

Before implementing the JSF banking application we have to setup the environment as described in “Setting up the environment for this redbook” on page 533:

- ▶ Create and load the EJBBANK database
- ▶ Setup server targeting for the workspace
- ▶ Prepare the JSFServer for testing

Implementing the banking application with JSF

In this section we provide detailed instructions to implement the JSF banking application. We implement the banking application with a JSF frontend and a JDBC backend. We use action classes (a Struts concept) as the link between the frontend and the backend.

Define a utility project

We want to use the same banking model for different implementations in enterprise applications. The best way to handle this is by having the model in a utility project that is part of the enterprise application:

- ▶ Define a new project (*File -> New -> Project -> Java -> Java Project*) with the name **ItsoBankModel**. Click *Next*.

For Java Settings, go to the Libraries page and click *Add Library*. Select *JRE System Library* and click *Next*. Select *WebSphere v5.1 JRE* and click *Finish*. This extra library is required for JDBC support (javax.sql).

Click *Finish*. At the prompt, click *Yes* to switch to the Java perspective.

- ▶ Import the banking model by selecting the **ItsoBankModel** project and *Import* from the context menu (right mouse button). Select *Zip file* and click *Next*. Click *Browse* to locate `\labscode\dev-java\BankingModel.jar`. Click *Open*.

Make sure the destination folder is the **ItsoBankModel** project. Click *Finish*.

- ▶ Study the code of the model.

The **itso.bank.model** package contains the model classes.

- The **Customer** class contains `id`, `title`, `firstName`, and `lastName`.
- The **Account** class contains `id`, `type` (CHECKING or SAVINGS), and `balance`. It provides a `deposit` and a `withdraw` method.
- The **TransRecord** class contains `timeStamp`, `transType` (C for credit or D for debit), and `transAmt` (amount).

- ▶ The **itso.bank.facade** package contains the **Bank** interface, two implementations, **BankMemory** and **BankJDBCTest**, and **BankingTest**, a facade between applications and the implementation of the model.

- **Bank** interface defines the business methods: `getAccount`, `getCustomer`, `getAccounts` (for one customer), `getTransactions` (for one account), `deposit`, `withdraw`, and `transfer`. Every operation on funds creates a transaction record.
- **BankMemory** is an in-memory implementation of the interface. It holds a number of customers, their accounts, and one transaction record per

account for the initial deposit, in memory. It provides the business functions.

- **BankJDBCTest** is an implementation using JDBC to access the EJBBANK database. This is the persistent implementation of the bank interface that we will use for the JSF application.
- **BankingTest** is the facade between applications and the implementation of the model. It instantiates one of the bank implementations, to which it passes all requests. Initially it is setup to use the memory bank.

The **itso.bank.exceptions** package contains a number of application exceptions, such as customer not found, or not enough funds for a withdrawal.

The **itso.bank.main** package contains the **BankMain** test program that executes a few functions of the model using the **BankingTest** facade.

- ▶ Close the Java perspective. Select the Web perspective.

Create an enterprise application

We define an enterprise application named **ItsoJSF05Bank** to hold the JSF banking application:

- ▶ Select *New -> Project -> J2EE -> Enterprise Application Project*. Select J2EE 1.3 and click *Next*. Enter **ItsoJSF05Bank** as name. For Target Server verify *WebSphere Application Server V5.1*. Click *Finish*. Do not switch perspectives.
- ▶ Open the EAR deployment descriptor (expand the project). On the Module page, under Project Utility JARs, click *Add* and select the **ItsoBankModel** project. This project will be available to all modules.
- ▶ Save and close the EAR deployment descriptor.
- ▶ In the Servers view, select the JSFServer and *Add and remove projects* from the context menu. Select the **ItsoJSF05Bank** project and click *Add* to associate the enterprise application with this server. You can remove all other projects for faster starting of the server. Click *Finish*.

Create a Web project for the banking application

We start with a Web project that has access to the banking model (**ItsoBankModel**) and import the facade and JDBC backend:

- ▶ Create a Web project (*New -> Dynamic Web Project*). Enter **ItsoJSFBank** as name, and select *Configure advanced options*. Select **ItsoJSF05Bank** as EAR project and select *Use the same target server as the EAR project*. Click *Finish*.

Click *OK* to repair the server configuration.

- ▶ Select the `ItsoJSFBank` project and *Properties* (context). Select *Java JAR Dependencies* and select the `ItsoBankModel.jar` file. Click *OK*. The Web project now has access to the data model.
- ▶ Open the deployment descriptor, `web.xml`. On the *References* page, *Resource* tab, click *Add* and define the data source reference for the `EJBBANK` database. Enter **`jdbc/mybank`** as name. Locate the `javax.sql.DataSource` class and enter the JNDI name as **`jdbc/ejbbank`** (Figure 5-4).

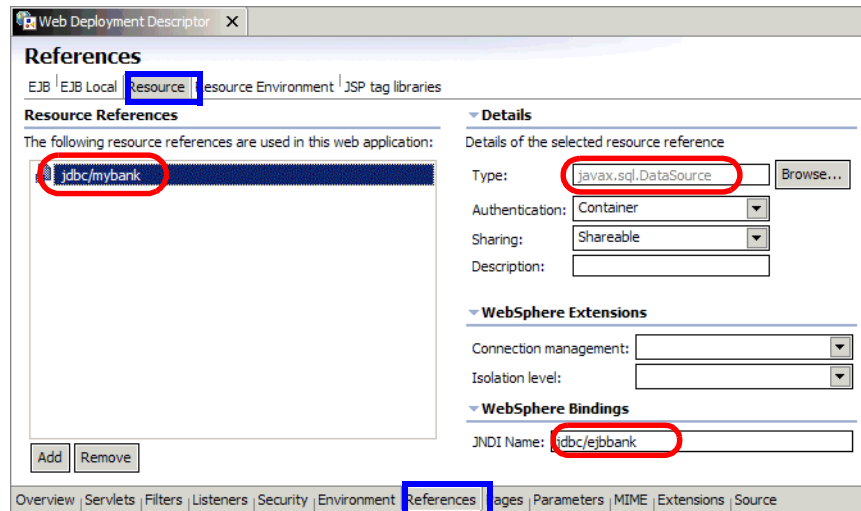


Figure 5-4 Define a JDBC reference for the `EJBBANK` database

- ▶ Under *Java Resources*, create a package named **`itso.bank.facade`**. Select the package and *Import*. Select *File system*, then locate `\labscode\dev-jsfbank\backend` and import **`Banking.java`** and **`BankJDBC.java`**.
 - `Banking`—facade between business logic and database backend.
 - `BankJDBC`—simple JDBC access to the `EJBBANK` database using the **`jdbc/mybank`** data source.

The backend could easily be switched to an EJB implementation. However, this is not important to learn about JSF.

- ▶ Create two more packages for helper classes; **`itso.jsf.action`** (action classes), and **`itso.jsf.data`** (managed beans).

Preparing managed beans with customer and transaction data

While a customer is working with the banking application we want to keep important customer information (ID, name) and the accessible accounts (array of account numbers) in the session. We define a JavaBean to hold this information. When displaying transaction records we keep an array of TransRecord beans on the session:

- In the `itso.jsf.data` package create a class named **CustomerData**. Enter this code (from `\labscode\dev-jsfbank\beans\customerdata.txt`):

```
private Customer customer = new Customer();
private String[] accountNumbers = null;

public String[] getAccountNumbers() { return accountNumbers; }
public Customer getCustomer()      { return customer; }

public void setAccountNumbers(String[] strings) {
    accountNumbers = strings; }
public void setCustomer(Customer customer)     {
    this.customer = customer; }
```

- In the `itso.jsf.data` package create a class named **TransRecordData**. Enter this code (from `\labscode\dev-jsfbank\beans\trdata.txt`):

```
private TransRecord[] transactions = null;
private int           numberRecords = 0;

public int           getNumberRecords() { return numberRecords; }
public TransRecord[] getTransactions()  { return transactions; }

public void setTransactions(TransRecord[] records) {
    transactions = records;
    numberRecords = transactions.length;
}
```

- Save and close the classes.

Creating the home page

We start with the home page where a customer enters the customer ID:

- Under WebContent, create a JSF page (*New -> Faces JSP File*). Enter **index.jsp** as name and click *Finish*.
- In the Page Data view, select *New -> JavaBean*. Select *Add new JavaBean*, enter **customerData** as name, locate the CustomerData class, select *Make this JavaBean reusable to other pages*, and select *session* for the scope. Click *Finish*.

- Expand the customerData bean and the customer bean inside. Select the id property and drag/drop it into the index.jsp Design view under Place content here.
 - Select *Updating fields*. The control changes to *Input Field*.
 - Change the label to: Please enter your customer ID:
 - Click *Finish* (Figure 5-5).

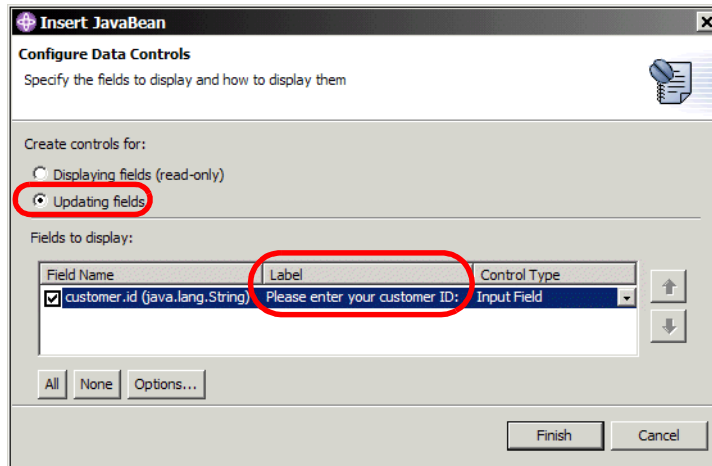


Figure 5-5 Creating an input field for the customer number

- This adds a label, an input field (bound to customerData.customer.id), an error message field, and a Submit button.
- Select the {Error Messages} field and in the Attributes view enter color: red for the style properties. Error messages will be displayed in red.
- To validate the customer ID, select the input field and in the Attributes view, Validation tab, select *Value is required*, minimum 3, maximum 3, and select *Digits only* for constraints (Figure 5-6).

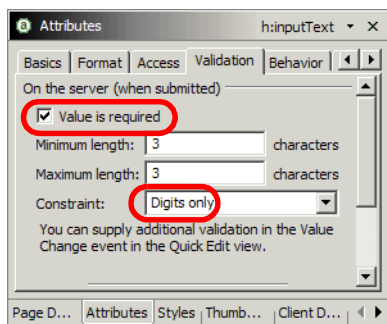


Figure 5-6 Field validation

- Under the Submit button and form (`</h:form>`), add this text in the Source view:

```
<P>For more information on ITS0 and RedBooks, please visit our
<A href="http://www.redbooks.ibm.com">Internet Site</A></P>
```

Alternative: You can use Ctrl-o (insert paragraph) and Ctrl-q (insert link) in the Design view and enter the URL and text.

- Select WebContent and *Import*. Locate the \labscode folder. In the Import dialog expand labscode and select the **images** folder (Figure 5-7). Click *Finish*.

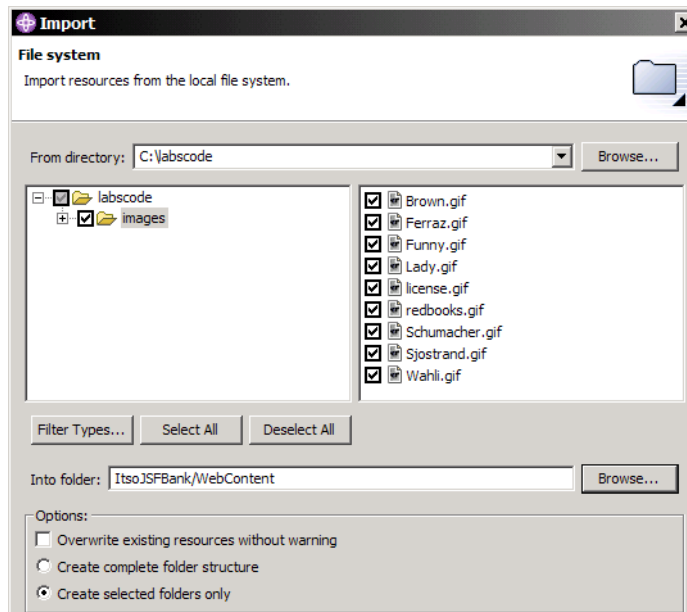


Figure 5-7 Import images

- Replace the default heading text Place content here with the text Welcome to the JSF Bank, and make it a heading 2 (select the text and *Insert -> Paragraph -> Heading 2*).
- Expand WebContent/images. Select the redbooks.gif file and drop it on the top of the page (Design view) to get the final design (Figure 5-8). Save the index.jsp.

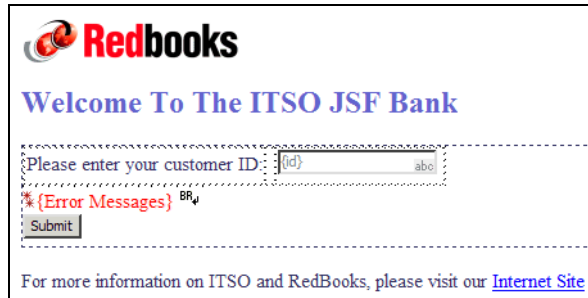


Figure 5-8 Design view of the index.jsp

Navigation

When the Submit button is clicked we retrieve the customer from the database and list the accounts in a new page, `listAccounts.jsp`. If an error occurs (for example, customer not found), we display a generic error page, `error.jsp`:

- ▶ Create two JSF pages, **`listAccounts.jsp`** and **`error.jsp`**. Add `LISTACCOUNTS` and `ERROR` to the default text so you know which page is displayed. Close the files for now, we will implement them later.
- ▶ Select the Submit button and go to the Attributes view. Change the `id` to `submit`. On the Navigation tab (Figure 5-9), click *Add*.
 - Select the `listAccounts.jsp`, enter an alias of **`success`** and `ActionRef *`.
 - Create another rule: `error.jsp`, alias=**`error`**, `*`, and select *Globally*.
 - Save the `index.jsp`.

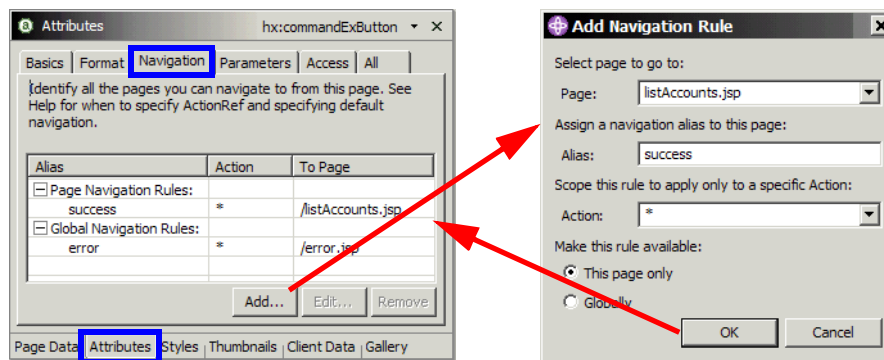


Figure 5-9 Navigation for the index.jsp

Implementing the action

The action logic, `ListAccountsAction`, retrieves the customer and the accounts from the database and fills the `CustomerData` bean. We invoke the action logic from the Submit button and set the navigation result depending on the return value of the `ListAccountsAction`:

- ▶ Select the `itso.jsf.action` package and *Import*. Locate `\labscode\dev-jsfbank\actions` and import `ListAccountsAction.java`. Open the class and study the code. The class returns either "success" or an error message.
- ▶ Select the Submit button and in the Quick Edit view, select *Command* and enter this code (`\labscode\dev-jsfbank\logic\indexAction.txt`):

```
log("Customer login");
String result = (new itso.jsf.action.ListAccountsAction())
                .execute(getCustomerData());

if ( result.equals("success") )
    return "success";
else {
    log("Error: "+result);
    getRequestScope().put("errorMessage", result);
    return "error";
}
```

Notice that we pass the error message of the action logic in the request scope variable `errorMessage`. Save the `index.jsp`.

Testing the home page and customer retrieval

We are ready to test the home page and the retrieval of the customer information:

- ▶ Start the JSFServer. If publishing fails, remove the `ItsoJFS06` application from the server, then add it again.
- ▶ Select the `index.jsp` and *Run on Server*. Set the JSFServer as default server and click *Finish*.

Try a good customer ID (101) and bad ones (empty, 1234, abc, 199) to see the validation errors. Also watch the Console view for messages.

Implementing the error page

We use the `error.jsp` as a generic error page for the whole application. The error message to be displayed is passed as a request variable, `errorMessage`.

Open the **error.jsp**:

- ▶ In the Page Data view, expand *JSP scripting* -> *requestScope* and *Add Request Scope Variable*.
- ▶ Enter **errorMessage** as name and `java.lang.String` as type.
- ▶ Click *OK*.

Edit the **error.jsp** (Figure 5-10):

- ▶ Change the page text to: **ITSO Bank Error Message** (as heading 2)
- ▶ Drag the **redbooks.gif** image to the top.
- ▶ Select the *requestScope* -> *errorMessage* and drag/drop it under the heading. Select *Displaying fields (read-only)* and change the label to: An error has occurred:
- ▶ Select the `{ErrorMessages}` output field and set the style properties to `color: red`.
- ▶ Add a Command - Button component at the bottom, change the id to **back** and the label to **Back**. For Navigation add a rule: `index.jsp, home, *`
- ▶ In the Quick Edit view (for the Back button), enter: `return "home";`



Figure 5-10 Design view of the **error.jsp**

- ▶ Test the application with an invalid customer ID (123) to see the custom error message.

Implementing the list of accounts

When retrieval of the customer and account numbers is successful we display the customer name and the account numbers (for selection).

Open the **listAccounts.jsp**:

- ▶ In the Page Data view select *New* -> *JavaBean*. Select *Add existing reusable JavaBean*, select the `customerData` bean, and click *Finish*.

- ▶ Drag/drop the `redbooks.gif` to the top.
- ▶ Replace the generated text with a heading 2: `Accounts of:`.
- ▶ Add two `OutputText` components to the heading 2, with one space between the components (Figure 5-11).



Figure 5-11 Initial design to list the accounts

- ▶ In the Page Data view expand `customerData`. Drag/drop the `firstName` and `lastName` properties onto the two `OutputText` components in the heading.
- ▶ Drag/drop the `customerData` bean below the heading. Select *Displaying fields*. Only select the `accountNumbers` property and delete its label text. Leave the control type as *Multi-Column Data Table*. Click *Finish* and a data table is inserted into the page (Figure 5-12).

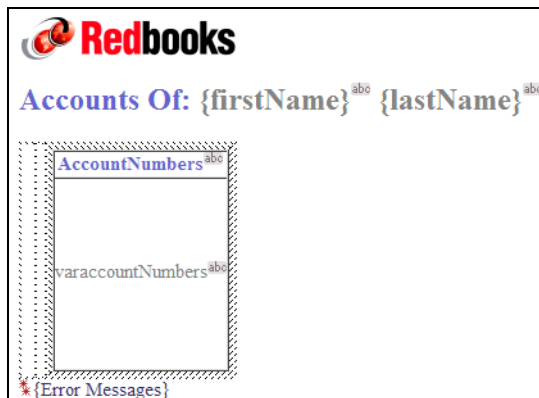


Figure 5-12 Adding a data table for the account numbers

- ▶ Select the column heading and in the Attributes view change the text to: `Select an Account number:`
- ▶ Place a `Command - Button` component at the bottom. Change the id to `cancel`, and set the label to `Cancel` (Figure 5-13).



Figure 5-13 Design view of the listAccounts.jsp

- Save the listAccounts.jsp.

Test the design

We can now test the listAccounts.jsp. Restart the enterprise application in the JSFServer. Run the application with a valid customer ID (101) and the list of accounts is displayed (Figure 5-14).

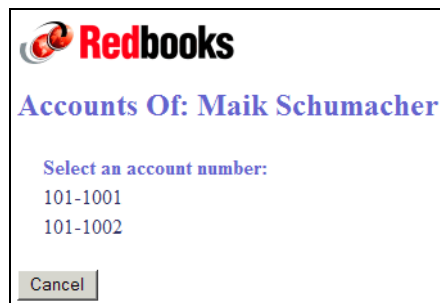


Figure 5-14 Testing the list accounts functionality

Implement navigation

We have to implement account selection in the output table. When an account is selected, the account details are displayed in the next JSP. When the Cancel button is clicked, we go back to the home page:

- Import the other action classes from \labscode\dev-jsfbank\actions into the itso.jsf.action package (**AccountDetailsAction**, **ListTransactionAction**, **PerformTransactionAction**, **RefreshAccountAction**).
- Create a new JSF page named **accountDetails.jsp**. Close the file for now.

- In the Design view of the `listAccounts.jsp`, select the Cancel button and define three navigation rules (Figure 5-15):
 - `accountDetails.jsp`, **details**, *
 - `index.jsp`, **home**, *
 - `listAccounts.jsp`, **stay**, *

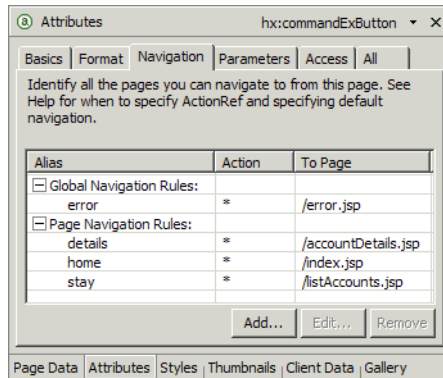


Figure 5-15 Navigation for the `listAccounts.jsp`

- Select the Cancel button and in the Quick Edit view select *Command* and enter the action logic (use the file `(\labscode\dev-jsfbank\logic\listAccountsBack.txt)`):

```
log("Account selection: back");
getSessionScope().remove("customerData");
return "home";
```

Notice that we clean up the session data.

Define the account managed bean

After selecting an account we retrieve the account from the database and put the Account JavaBean into the session scope:

- In the Page Data view, select *New -> JavaBean*. Select *Add new JavaBean*, enter **account** as name, and select the `itso.bank.model.Account` class. Select *Make this JavaBean reusable to other pages*, and select *session scope*.

Defining a hyperlink

To select an account number we define a hyperlink in the account list table:

- In the Design view, select a Command - Hyperlink component and drop it onto the `varaccountNumbers` field. A link icon appears on the left of the field (Figure 5-16). If the link icon appears somewhere else, delete it and try again.


```

        return "error";
    } catch (Exception e) {
        getRequestScope().put("errorMessage",
                             "Exception: "+e.getMessage());
        return "error";
    }
    return "details"; // go to the account details page

```

- Save the `listAccounts.jsp`.

Test account selection

We can now test the selection of an account. Restart the enterprise application in the JSFServer. Run the application and test selecting an account. Also test the Cancel button.

Implementing the account details page

After selecting an account we display the account details (type, balance). Then the user can perform the actions: deposit, withdraw, transfer, and list the transactions records.

Open the `accountDetails.jsp`:

- Add the `redbooks.gif` on the top. Change the default text to a heading 2: Account Information and Transactions.
- In the Page Data view select *Add -> JavaBean* and select the existing account bean. Repeat this step to define the existing customerData bean.
- In the Page Data view, select *New -> JavaBean*. Select *Add new JavaBean*, enter `trData` as name, and select the `itso.jsf.data.TransRecordData` class. Select *Make this JavaBean reusable to other pages*, and select *session* scope.
- In the Page Data view add three *requestScope* variables named **action**, **amount**, and **destinationAccount**, all `java.lang.String` (Figure 5-18).

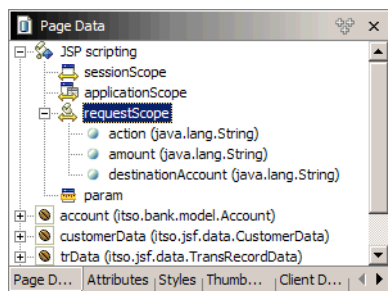


Figure 5-18 Defining request scope variables

- ▶ Drop the account bean under the heading. Select *Displaying fields*. Change the id field label to Account number:.
- ▶ Move the type field above the balance field using the arrow on the right. Click *Finish*.
- ▶ Change the errorMessages field to color: red (Figure 5-19).

Figure 5-19 Designing the accountDetails.jsp

- ▶ At the bottom, after two BR (breaks), add an Output component with the text: Select the transaction you would like to perform:.
- ▶ Add a table (*Insert -> Table*) with one row, two columns, border 1, padding 8.
- ▶ Drop a Radio Button Group component into the first column. In the Attributes view:
 - Set the id to actionRadio.
 - Set the value to #{requestScope.action} (use the icon to select).
 - Set the direction to *Vertical*.
 - On the Choices tab add 4 text items with the names: List Transactions, Deposit, Withdraw, Transfer. Set the values of the choices to listTransactions, deposit, withdraw, transfer (Figure 5-20).

Name	Value
List Transactions	listTransactions
Deposit	deposit
Withdraw	withdraw
Transfer	transfer

Figure 5-20 Defining a radio button group

- ▶ Insert a table into the second column: two rows, two columns, border 0, padding 5:
 - Into the first column drop two Output components with text: Amount: and Destination account:.
 - Into the second column drop two **Input** components. Set the ids to **amountInput** and **destinationAccountInput**, and set the width to 12.
 - The amountInput field is used for deposit, withdraw, and transfer. The destinationAccountInput field is only used for transfer.
 - Bind the two input fields to the matching two request variables (drag/drop the request variables to the fields, or use the Attributes view value field).
 - Select the amountInput field. On the Validation tab select *Add Display Error control*.
 - Select the destinationAccountInput field. On the Validation tab set minimum and maximum length to 8 and select *Add Display Error control*.
 - Select the field again and on the Behavior tab set maximum length to 8 (you cannot enter more characters). Select *Enable validation in the browser*. Set style class on failure to message, and action on failure to *Select field*.
 - Set color: red for both error message fields (Attributes view style).
- ▶ Add two Command - Button components at the bottom. Set the ids to submit and cancel and the labels to Submit and Cancel (Figure 5-21).

The image shows a web page design for 'Redbooks Account Management'. It features a form with the following elements:

- Account Information:** Labels for 'Account number:', 'Type:', and 'Balance:' followed by input fields with IDs {id}, {type}, and {balance} respectively.
- Error Messages:** A red text label '{Error Messages}'.
- Transaction Selection:** A label 'Select the transaction you would like to perform:' followed by a list of radio buttons: 'List Transactions', 'Deposit', 'Withdraw', and 'Transfer'.
- Transaction Form:** A table with two rows:

Amount:	{amount}	{Error Message for amountInput}
Destination account:	{destinationAccount}	{Error Message for destinationAccountInput}
- Buttons:** 'Submit' and 'Cancel' buttons at the bottom.

Figure 5-21 Design view of the accountDetails.jsp

Navigation logic

From the account details page we can go back to the list accounts page; we can perform transactions (deposit, withdraw) and stay on the details page; we can perform a transfer and display both accounts; or we can list the transaction records on the list transactions page:

- ▶ Create two JSF pages named **listTransactions.jsp** and **transferResult.jsp**. Close the two pages.
- ▶ In the Design view of `accountDetails.jsp`, define four navigation rules for the Submit button:

- `accountDetails.jsp`, **perform**, *
- `transferResult.jsp`, **transfer**, *
- `listTransactions.jsp`, **listTransactions**, *
- `listAccounts.jsp`, **back**, *

- ▶ Select the Cancel button and add the logic in the Quick Edit view (`\labscode\dev-jsfbank\logic\accountDetailsCancel.txt`):

```
log("AccountDetails back");
getSessionScope().remove("account");
return "back";
```

- ▶ Select the Submit button and add the logic in the Quick Edit view (`\labscode\dev-jsfbank\logic\accountDetailsSubmit.txt`):

```
log("Account operations");
String action = (String)getRequestScope().get("action");
String amount = (String)getRequestScope().get("amount");
String destination =
    (String)getRequestScope().get("destinationAccount");
String result = null;
log("Action: "+action+" Amount: "+amount);
if (action == null) {
    javax.faces.application.FacesMessage errmsg =
        new javax.faces.application.FacesMessage
            (javax.faces.application.FacesMessage.SEVERITY_ERROR,
             "Select an action.", "Select an action.");
    getFacesContext().addMessage(null, errmsg);
    return "perform";
}
if ( action.equals("listTransactions") ) {
    log("listTransactions: "+getAccount().getId());
    try {
        Account account2 = getAccount();
        itso.bank.model.TransRecord[] tr =
            (new itso.jsf.action.ListTransactionAction())
                .execute(account2);
        getTrData().setTransactions(tr);
        log("transactions: "+getTrData().getNumberRecords());
    }
}
```



```

        return "listTransactions";
    } catch (Exception e) {
        getRequestScope().put("errorMessage",e.getMessage());
        return "error";
    }
}
if (amount.equals("")) {
    javax.faces.application.FacesMessage errmsg =
        new javax.faces.application.FacesMessage
            (javax.faces.application.FacesMessage.SEVERITY_ERROR,
             "Amount required.", "Amount is required.");
    getFacesContext().addMessage("amountInput",errmsg);
    return "perform";
}
if (action.equals("transfer") && destination.equals("")) {
    javax.faces.application.FacesMessage errmsg =
        new javax.faces.application.FacesMessage
            (javax.faces.application.FacesMessage.SEVERITY_ERROR,
             "Destination account required.",
             "Destination account is required.");
    getFacesContext().addMessage("destinationInput",errmsg);
    return "perform";
}
try {
    result = (new itso.jsf.action.PerformTransactionAction()).execute
        (action, getAccount(), new java.math.BigDecimal(amount),
         destination, getCustomerData().getCustomer().getId());
} catch (Exception e) {
    getRequestScope().put("errorMessage",e.getMessage());
    return "error";
}
log("Result: "+result);
if (result.equals("success")) {
    getRequestScope().put("amount","");
    getRequestScope().put("destinationAccount","");
    if (action.equals("transfer")) {
        Account destacct = new Account();
        destacct.setId(destination);
        (new itso.jsf.action.RefreshAccountAction()).execute(destacct);
        getRequestScope().put("destAccount",destacct);
        log("Transfer: dest="+destacct);
        return "transfer";
    }
    return "perform";
} else {
    getRequestScope().put("errorMessage",result);
    return "error";
}
}

```

- ▶ Save the `accountDetails.jsp`.

Test account management

We can now test deposit and withdraw in the account details page. Restart the enterprise application in the JSFServer. Run the application and test deposit and withdraw operations.

Implementing the transfer result page

The transfer result page displays both accounts involved in the transfer operation.

Open the `transferResult.jsp`:

- ▶ In the Page Data view, define a *requestScope* variable named **destAccount** of type `itso.bank.model.Account`. This variable is passed from the account details page.
- ▶ In the Page Data view, define the existing account `JavaBean`.
- ▶ Place the `redbooks.gif` at the top.
- ▶ Define a heading 2: `Transfer Result`.
- ▶ Define two heading 3: `From Account:` and `To Account:`.
- ▶ Drop the account bean under the from account heading. Select *Displaying fields*. Click *Finish*.
- ▶ Drop the `destAccount` variable under the to account heading. Select *Displaying fields*. Click *Finish*.
- ▶ Delete the two error messages fields.
- ▶ Add a Command - Button labelled `Back` at the bottom. Set the id to `back` (Figure 5-22).

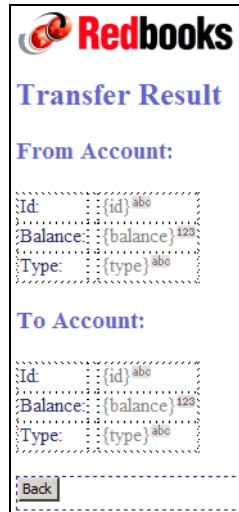


Figure 5-22 Design view of the `transferResult.jsp`

- ▶ Select the Back button and define one navigation rule:
`accountDetails.jsp, back, *`
Enter the navigation logic in the Quick Edit view: `return "back";`
- ▶ Save the `transferResult.jsp`.

Test transfer

We can now test a transfer of funds. Restart the enterprise application in the JSFServer. Run the application and test the transfer operation with an amount and a destination account.

Implementing the transaction list

The transaction list page displays all the transactions of the current account.

Open the `listTransactions.jsp`:

- ▶ On the Page Data view, define the existing account and `trData` managed beans.
- ▶ Put the `redbooks.gif` at the top.
- ▶ Delete the generated text.
- ▶ Drop an Output - Formatted Text component under the `redbooks` image. In the Attributes view, change the value to: `Account: {0} {1}`. The two parameters are the account number and type:

- In the Attributes view, Parameters tab, click *Add* twice. Change the names to number and type. Set the values to the id and type properties of the account bean, using the reference icon. The values become:

```
#{pc_ListTransactions.account.id}
#{pc_ListTransactions.account.type}
```

Note: You can also drag and drop the account properties onto the formatted output component to add the two parameters, then change the generated names from msgx to id and type.

- Select the formatted output component and make it a heading 2 (*Insert -> Paragraph -> Heading 2*).
- Drop two Output components under the heading. Bind the first component to `trData.numberRecords`. Change the value of the second component to: transactions, sorted in order of occurrence (Figure 5-23).

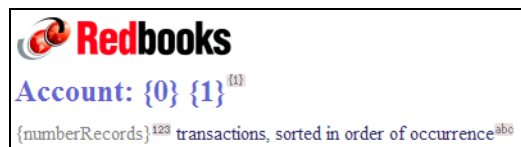


Figure 5-23 Designing the `listTransactions.jsp`

- In the Page Data view expand `trData` and drop the transactions property into the JSP (Figure 5-24).

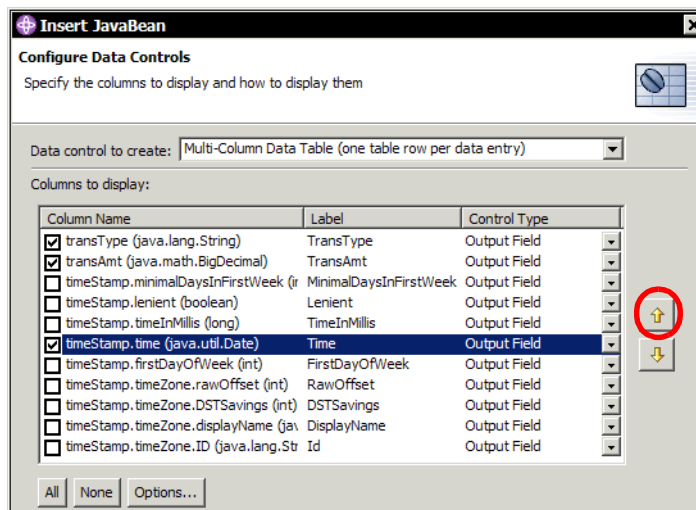


Figure 5-24 Creating a data table for the list of transactions

- Select only `transType`, `transAmt`, and `timeStamp.time`.
- Move `timeStamp.time` to the top (using the arrow on the right).
- Click *Finish* and the data table is inserted (Figure 5-25).

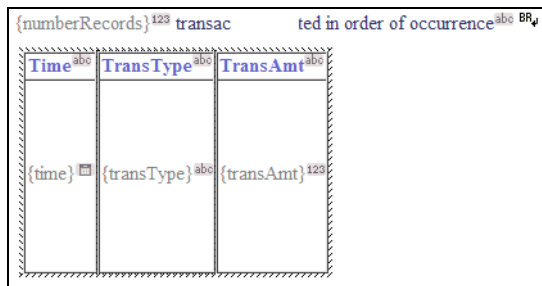


Figure 5-25 Design view of the `listTransactions.jsp`

- Select each column heading and change the text to `Timestamp`, `Type`, and `Amount`.
- Select the data table (`<h:dataTable>`) and in the `Attributes` view, `All` tab, set border to **1**. (You can select a column in the table and then in the `Attributes` view, in the title pull-down, select the `h:dataTable`.)
- Select the `{time}` output component and in the `Attributes` view, `Format` tab, change `Format` as to *Date and Time*. Leave the default styles for date and time (Figure 5-26).

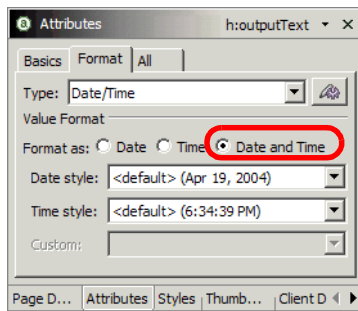


Figure 5-26 Changing the format of the timestamp

- Select the `{transType}` output and set the style to `text-align: center`.
- Select the `{transAmt}` output and set the style to `text-align: right`.
- ▶ Drop a `Command - Button` at the bottom, with label `Back` and id `back`.
- ▶ Make the error message field red.
- ▶ For navigation, select the `Back` button and add one rule:
`accountDetails.jsp, back, *`

- ▶ Select the Back button and add the logic in the Quick Edit view:

```
getSessionScope().remove("trData");
return "back";
```

- ▶ Save the `listTransactions.jsp`.

Test list transactions

We can now test the list of transactions for an account. Restart the enterprise application in the JSFServer. Run the application and test the list transactions operation.

Implementing paging and transaction details

We can setup the data table with paging so that only a certain number of records are displayed. We can also setup a hot link in the transaction list to display one transaction record. The data table provides row action support.

Create a JSF page named `transactionDetails.jsp`. Close the page for now.

Open the `listTransactions.jsp`:

- ▶ Select the inner `h:dataTable`. In the Attributes view, click *Add row action support*. An extra column is added.
- ▶ In the Attributes view, Paging tab, click *Add deluxe pager* and set Items/Page to 4 (Figure 5-27).

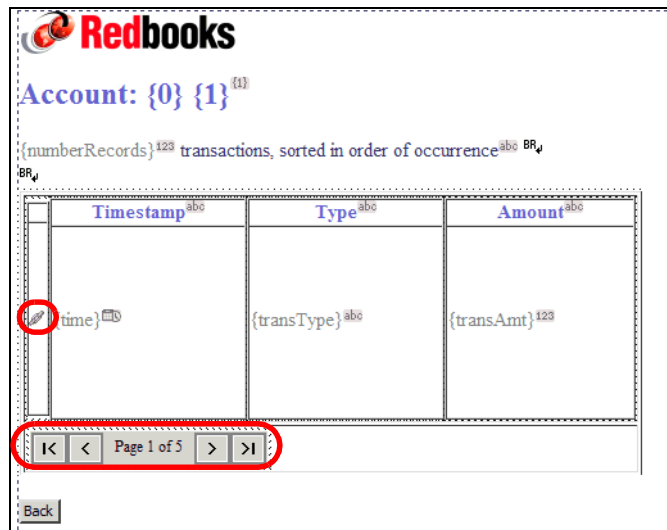


Figure 5-27 Implementing paging for the list of transactions

- ▶ Select the row action link icon and in the Attributes view, Navigation tab, add a rule: `transactionDetails.jsp, details, *`.
- ▶ Select the row action link icon and in the Quick Edit view enter the logic (`\labscode\dev-jsfbank\logic\listTransactionsLink.txt`):

```
log("ListTransaction -> transactionDetails:
    "+getRowAction1().getRowIndex());
try {
    getRequestScope().put("transDetail",
        getTrData().getTransactions()[getRowAction1().getRowIndex()]);
    return "details";
} catch (Exception e) {
    getRequestScope().put("errorMessage", e.getMessage());
    return "error";
}
```

The `rowIndex` is the selected entry. We extract the `TransRecord` from the array and put it into the request scope variable `transDetail`.

Open the **transactionDetails.jsp**:

- ▶ Place the `redbooks.gif` at the top and a heading 2: Transaction Details.
- ▶ In the Page Data view, add a *requestScope* variable named **transDetail** of type `itso.bank.model.TransRecord`.
- ▶ Add the existing managed beans `customerData` and `account`.
- ▶ Insert a table with three rows, two columns, border 1, padding 6.
- ▶ In the left column place three heading 3: Customer, Account, Transaction.
- ▶ In column two first cell, place an Output component and bind it to `customerData.customer.lastName`.
- ▶ Drag the `account` bean into the second cell. Select *Displaying fields*.
- ▶ Drag the `requestScope.transDetail` variable into the third cell. Select *Displaying fields*. Only select `timestamp.time` (place on top), `transType`, and `transAmt`. Change labels to `timestamp`, `type`, and `amount`. Click *Finish*.
- ▶ Select the `{time}` output and set the format to *Date and Time*.
- ▶ Delete the error messages fields.
- ▶ Add a Command - Button component and set the label to Back (Figure 5-28).

Transaction Details							
Customer	{lastName} abc						
Account	<table border="1"> <tr> <td>id</td> <td>{id} abc</td> </tr> <tr> <td>balance</td> <td>{balance} 123</td> </tr> <tr> <td>type</td> <td>{type} abc</td> </tr> </table>	id	{id} abc	balance	{balance} 123	type	{type} abc
id	{id} abc						
balance	{balance} 123						
type	{type} abc						
Transaction	<table border="1"> <tr> <td>timestamp</td> <td>{time} </td> </tr> <tr> <td>type</td> <td>{transType} abc</td> </tr> <tr> <td>amount</td> <td>{transAmt} 123</td> </tr> </table>	timestamp	{time}	type	{transType} abc	amount	{transAmt} 123
timestamp	{time}						
type	{transType} abc						
amount	{transAmt} 123						
Back							

Figure 5-28 Design view of the transactionDetails.jsp

- ▶ Select the Back button:
 - Enter a navigation rule: listTransactions.jsp, **back**, *.
 - Enter the logic: return "back";
- ▶ Save the transactionDetails.jsp.

Test the paging and transaction details

We can now test the paging and the transaction details page. Restart the enterprise application in the JSFServer. Run the application and test the paging in the transaction list and the transactions details (click anywhere in the row).

Implementing custom validation

The amount and destination account on the accountDetails.jsp can be validated using custom validators:

- ▶ Create a package named itso.jsf.validate. Import the two validators (Java classes) from \labscode\dev-jsfbank\validation.

The **AmountValidator** verifies that the amount entered can be converted into a BigDecimal. The **AccountValidator** verifies that the destination account entered follows the pattern 999-9999.

- ▶ Open the faces-config.xml file and register the validators at the end of the source file (\labscode\dev-jsfbank\validation\register.txt):


```

<faces-config>
.....
<validator>
  <description>Registers the AmountValidator</description>
  <validator-id>amountValidator</validator-id>
  <validator-class>itso.jsf.validate.AmountValidator</validator-class>
</validator>
<validator>
  <description>Registers the AccountValidator</description>
  <validator-id>accountValidator</validator-id>
  <validator-class>itso.jsf.validate.AccountValidator</validator-class>
</validator>
</faces-config>

```

- In the Source view of the `accountDetails.jsp`, add the **<f:validator>** tag to the two input fields (the tag must go before `</h:inputText>`):

```

<TD><h:inputText styleClass="inputText" id="amountInput" size="12"
  value="#{requestScope.amount}">
  <b><f:validator validatorId="amountValidator"></f:validator>
</h:inputText><h:message .....></h:message></TD>
.....
.....
<TD><h:inputText styleClass="inputText" id="destinationAccountInput"
  size="12" value="#{requestScope.destinationAccount}">
  <f:validateLength minimum="8" maximum="8"></f:validateLength>
  <b><f:validator validatorId="accountValidator"></f:validator>
</h:inputText><h:message .....></h:message></TD>

```

- Save the `accountDetails.jsp`.

Test the validation

We can now test the custom validators. Restart the enterprise application in the JSFServer. Run the application and test the custom validation. Enter characters into the amount field, and enter invalid account numbers (abc-1234, 123.4567) into the destination account field.

Displaying the customer image

In the `listAccounts.jsp` we can display the image of the customer:

- Open the `listAccounts.jsp`. Drop an Image component after the customer name (Figure 5-29).



Figure 5-29 Displaying an image

- ▶ Select the image component and in the Attributes view, All tab, bind the `url` attribute to the customer last name (use the icon on the right to locate the property):

```
{pc_ListAccounts.customerData.customer.lastName}
```

Change the generated text in the `url` attribute to access the GIF image in the images folder:

```
From: #{sc_ListAccounts.jsp.customerData.customer.lastName}
```

```
To: images/#{sc_ListAccounts.jsp.customerData.customer.lastName}.gif
```

- ▶ Save the `listAccounts.jsp`. Test the application (Figure 5-30).



Figure 5-30 Displaying a customer image

Running the JSF banking application

The JSF banking application is now complete. A sample run of the application is shown in Figure 5-31.

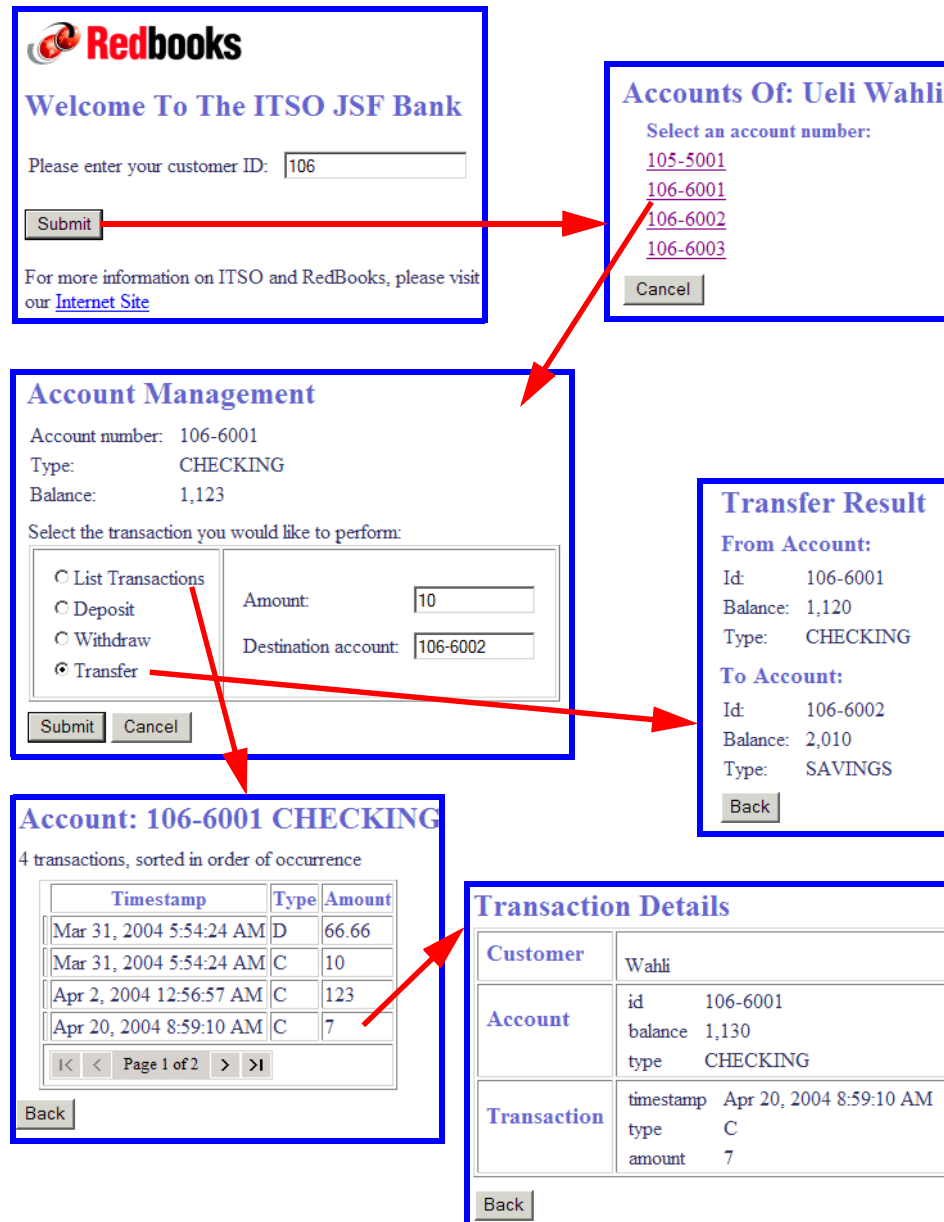


Figure 5-31 Sample run of the JSF banking application

Complete solution

We provide the complete solution for this application in the additional material in the file `\labscode\solution\ItsoJSF05Bank.ear`. See “Importing a solution enterprise application” on page 548 for instructions.



JSF application design and common patterns

In this chapter we describe Web application architecture issues and provide high-level guidelines about how to design Web applications using JavaServer Faces technology.

We cover the following topics:

- ▶ Roles in JSF Web applications development
- ▶ Application architecture issues
- ▶ Application design considerations

In this chapter we also describe a set of common user interface design patterns. These patterns are designed to help developers understand some of the most common User Interface designs that can be developed with JSF. When a developer starts to learn a new API or framework, it always helps to have a set of examples that can be used to help explain these concepts. The patterns start with a simple design overview and then the reader can work through the example in detail.

The following patterns are covered in this chapter:

- ▶ Create, read, update and delete (CRUD)
- ▶ Master to detail
- ▶ Form to form wizard

Roles

In this section we describe the different roles that can be considered in the development of Web applications using JSF technology. These roles should be used as a guide, as they represent the typical roles that can be found in a development team.

More than one role can be involved in a concrete step of the application development and each component of the development team could play one or more of these roles during the life cycle of the application.

Page author

Page authors are the primary users of all the tag library included with JSF. They are also the primary users of the custom components (custom tag library) specially developed for a concrete application.

They usually use markup language such as HTML to design and implement pages for Web applications. Page authors usually have a lot of experience with graphic design and with graphics editors.

Application developer

Application developers are responsible for the implementation of all the objects in the application, such as common objects, event handlers, converters and validators. They can also provide all the extra classes required in the Web application.

Component developer

Component developers are responsible for the creation of new custom UI components. They usually have user interface programming experience. These new custom UI components can be created directly from the UI component classes or they can extend others UI components (either the standard or the custom component library).

Application architect

They are responsible for the design of the complete Web application, ensuring reusability and scalability. They define page navigation, configure beans and register objects with the application.

Tools vendor

Tools vendors provide the development tools (such as WebSphere Studio Application Developer 5.1.2) to ease the tasks of development JSF-based Web applications.

Application architecture

In this section we talk about Web application architecture. We first cover the goals and principles that an architecture must try to achieve and then we cover the 4-tier architecture.

Goals and principles

When designing a JSF Web application and selecting an architecture, we have to keep in mind that every JSF Web application is a J2EE application. For this reason, the Web application must follow the same principles as any J2EE application. In fact, these principles are applicable to any Web-based business applications today.

These principles are:

- ▶ **Robustness**—The software inside a Web application must be reliable and bug-free. We must ensure this and take advantage of the technology to write quality code.
- ▶ **Performance and scalability**—Any Web application must meet the performance expectations of their users. It must also exhibit enough scalability, that is the potential for an application to support increased load, given appropriate hardware. Scalability is particularly important for Web applications because is difficult to predict the number of users and their behavior. We have to keep in mind the possibility of using a cluster for a Web application.
- ▶ **Take advantage of object-oriented (OO) design**—OO design principles offer proven benefits for complex systems. Good OO design practice is promoted by the use of proven design patterns, which are recurring solutions to common problems. These patterns are not technology-specific or language-specific. Java is an object-oriented language, so we have to keep in mind the possible advantages we could get in our applications when using OO patterns.
- ▶ **Low complexity**—We must do things simple. Complexity adds costs throughout the software life cycle and therefore can be a serious problem. On the other hand, the analysis must ensure that we do not have a naive and simplistic view of requirements.
- ▶ **Maintainability and extensibility**—Maintenance is by far the most expensive phase of the software life cycle. It is particularly important to consider maintainability when designing a Web application. A Web application can be a key application for an organization for years, so it must be able to accommodate new business requirements. Maintainability and extensibility depend largely on clean design (low-coupled modules and objects).

- ▶ **On time delivery**—Productivity is a key aspect when we intend to publish a new Web application.
- ▶ **Ease of testing**—Testing is a vital part of the software life cycle, and sometimes testing Web applications is a hard task.
- ▶ **Reusability**—We must try to minimize the duplication of code. Code reuse usually results from good OO design practice.
- ▶ **Support for multiple client types**—We have to consider that the Web application could be used through others channels than the Web.
- ▶ **Portability**—We may want to ensure the portability of the Web application in the case of we change the database provider, or if we decide to run it in another application server, or even run in two different application servers at the same time. Portability may be a business requirement of the Web application.

N-tier architecture

As other J2EE Web applications, a JSF Web application should have an n-tier architecture. In most of cases, the n stands for 3: Web tier, middle tier and backend tier. We have divided the middle tier into two tiers, so we have a 4-tier architecture as shown in Figure 6-1.

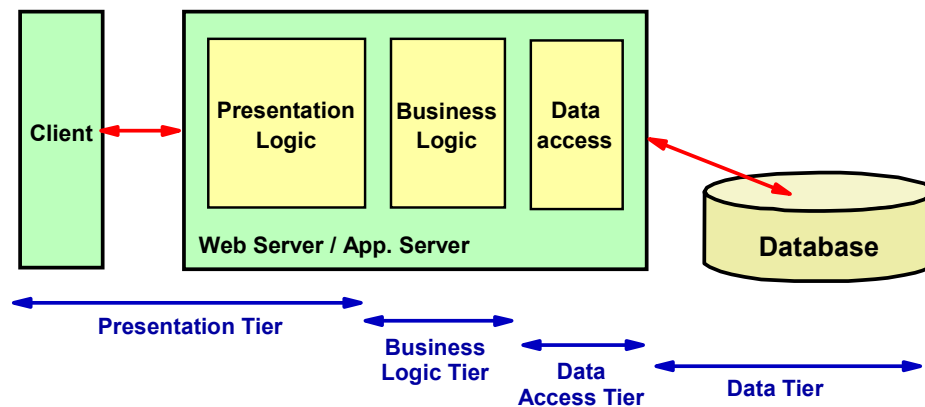


Figure 6-1 Web application architecture

Systems with three or more tiers have proven more scalable and flexible than client-server systems, in which there is no middle tier. In a well-designed multi-tier system, each tier should depend only on the tier beneath it. For example, changes to the database (database tier) should not demand changes to the Web interface (presentation tier).

The tiers considered are the following:

- ▶ Presentation tier—This tier handles the client presentation. It has the client and server logic to create all the user interface.
- ▶ Business logic tier—This tier has all the necessary logic for data manipulation. It is responsible for the data transformation and session control.
- ▶ Data access tier—This tier normally handles interfaces to access the backend data. It hides all data access from the business logic tier.
- ▶ Data tier—This tier controls data storage and access to backend systems, such as database management systems and legacy mainframe applications.

Now we can explore each tier in more detail.

Presentation tier

This is first tier we find in any Web application. It involves all the interface, either the server and the client parts. It exposes the business data to clients.

JSF technology is focused in this tier. JSF provides a set of extensible components to develop either the client and the server side. Those components are explained in Chapter 4, “JSF components” on page 85.

As this tier is the entrance point of our Web application, all the roles explained in “Roles” on page 228 are involved in its development.

Business logic tier

This layer is responsible for the data treatment. It receives all the data from the presentation layer. This data should contain the client identity, the operations the client wants to perform, and the necessary data for the operations.

In this tier are the business objects and all the business rules necessary to perform the client operations. This tier can be implemented using a lot of technologies, such as servlets, Enterprise JavaBeans, tailored beans, and so forth. In addition to these technologies, this tier could be implemented using Web services, or even invoking a third-party Web service.

Within JSF the communication between the presentation tier (in which this tool is more powerful) and this business logic tier can be done in several ways:

- ▶ Using managed beans
- ▶ Defining service data objects (see Part 3, “Service Data Objects” on page 297) to map the data sent/received to/from the presentation layer
- ▶ Discovering and invoking Web services

The roles working in this tier are the application architect and the application developer. The page author must know the interface between this tier and the presentation tier in order to publish all the business data correctly to the client.

Data access tier

The responsibility of this tier is to provide a way of communication between the business logic tier and the database tier. This interface hides all data access from the business logic tier.

If the database tier changes in the future, we can redevelop this tier without changing the business logic tier.

Usually entity beans or Web services are used in this tier to perform the communication with the business logic tier.

In this tier, only the application architect and the application developer should work.

Database tier

This tier contains the backend system. In this tier we can find database management systems, legacy mainframe applications, midsize server applications, and so forth.

This tier is typically implemented by the application architect and the application developer.

Design considerations

In this section we discuss design considerations for the development of Web applications using JSF technology. We start with a review of the features that a Web tier must accomplish and the MVC model. Then we discuss briefly portability. Finally, we include additional design considerations related to JSF.

Web-tier and the MVC design pattern

A good Web design must be extensible and maintainable. To achieve this, the two following things are necessary:

- **Cleanliness**—We say that a Web tier is clean when the generation of the markup language, the control flow, and the invocation to the business control are separated clearly.

- ▶ **Thinness**—We say that a Web tier is thin when it does not contain more Java code than is necessary to initiate business processing from user actions, and displaying the results. It means that the Web tier should contain only Web-specific control logic (such as the choice of the layout of the data) and not business logic (such as data retrieval).

Apart from these two goals, the Web tier has to be separate from the business logic tier. A proven way of achieving separation between presentation and logic is to apply the MVC architectural pattern to Web applications. We discussed this topic in “Model-view-controller architecture” on page 4. This architectural pattern also helps us to implement clean and thin Web tiers.

JSF technology is built using the MVC architectural pattern:

- ▶ **Model**—JSF provides managed beans for the build of the model in the Web tier. These managed beans are low-coupled with the UI components.
- ▶ **View**—JSF provides an extensible set of UI components that let you build rich user interfaces.
- ▶ **Controller**—JSF provides the mechanism to control the navigation over all the pages of the Web application, with a single entry point for all the requests of the clients of the Web application.

The extensive use of the MVC pattern ensures a high degree of cleanliness and thinness in all the Web tiers you develop using JSF.

Designing for portability

We are designing a Web application to run in a J2EE application server. In this kind of environment, portability is a key requirement. We have to ensure that we are able to retain as much portability as possible even if we choose or are forced to leverage the particular capabilities of the initial target platform. This can be achieved in three ways:

- ▶ Write code that complies with the J2EE specifications.
- ▶ Know the capabilities of the standard J2EE infrastructure and avoid using platform-specific solutions when a satisfactory standard alternative exists.
- ▶ Use loose coupling to isolate the use of platform-specific features, to ensure that the application design (if not all its code) remains portable.

Using JSP technology assures portability in the Web tier. JSF is a technology 100% written in Java and is designed to run on J2EE application servers. WebSphere Application Server 5.1 supports the JSF specification. This means that you can deploy a JSF Web application in all the platforms where WebSphere Application Server 5.1 is available.

Other considerations

In this section we present other considerations you have to keep in mind when developing Web applications using JSF technology.

Session scope management

Because HTTP is a stateless protocol, sometimes we have to maintain certain information about the client while he or she is navigating through the pages of a Web application. Other times we have to share information for *all* the users of a Web application.

In J2EE applications there is a set of *scopes* in which you can store the application data:

- ▶ **Application**—Data stored in this scope is shared among the entire Web application.
- ▶ **Session**—Data stored in this scope is valid only for a single user while interacting with the Web application.
- ▶ **Request**—Data stored in this scope is valid only for the page or pages associated with a request.
- ▶ **Page**—Data stored in this scope is valid only for the page where it has been defined. In JSF, this scope is referred as *param* scope.

In J2EE Web applications, session data can be managed in either the client or the server. Doing it in the client (browser) has a lot of limitations and security problems, so generally session management is left to the J2EE Web container, that is, the management is done in the server.

JSF technology does the management of these scopes automatically for you. When you define a managed bean to use in a Web application, you can make the bean reusable and select any of the scopes (application, session, request and page) where you want to share the bean (Figure 6-2).

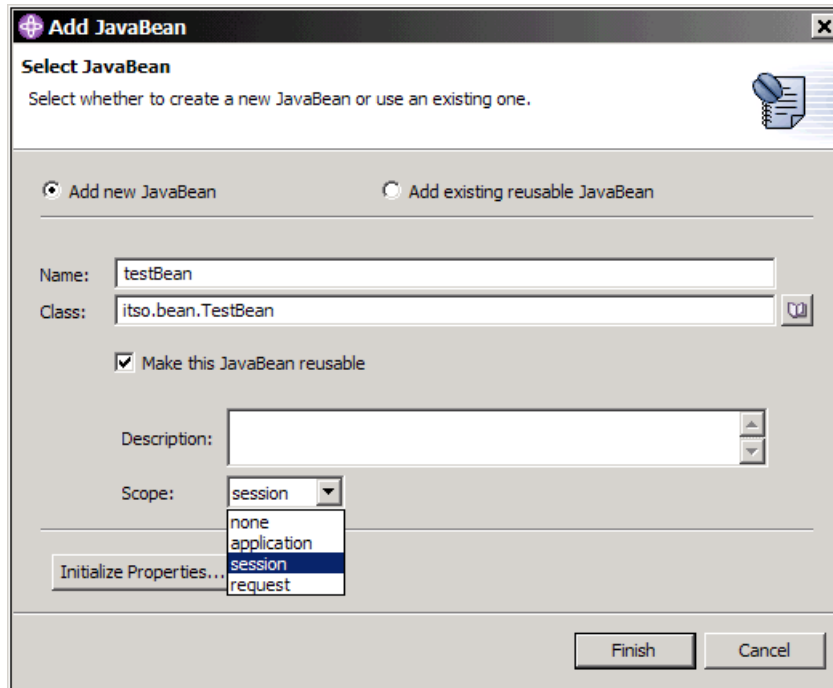


Figure 6-2 Selecting session scope for a new managed bean

You can also define variables or more complex objects for each scope, as shown in Figure 6-3.

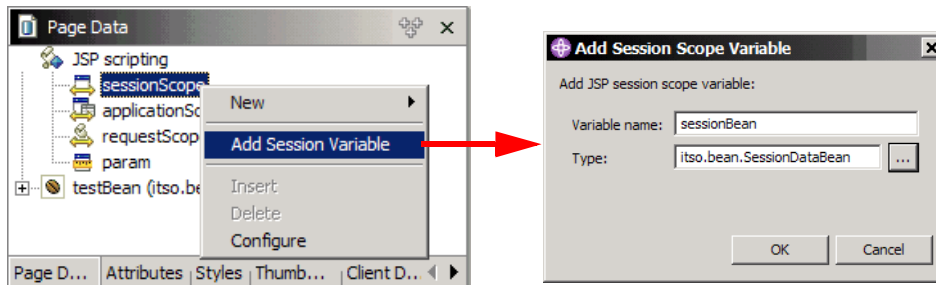


Figure 6-3 Defining objects or variables for the session scope

Once you have the variables or objects defined for each scope, you can use them in JSF pages by binding them to UI components. The UI binding is explained in “Binding an object to a component” on page 45.

Data conversion

Data conversion occurs mainly between the user interface and the model. Values entered in input fields as a number of characters must be converted into the data types of the model. Typical conversions are from input characters into numbers (Integer, Long, Float, Double, BigDecimal) and dates (in multiple formats).

Data conversions are defined in Studio in the Attributes view Format tab (see “Format tab” on page 41). Custom converters can be implemented as described in “Custom converters” on page 506.

Data validation

Data validation is very important when designing Web applications. We must ensure that the data sent to the server is as much accurate as possible. Data validation can be done either in the client or in the server; or in both:

- ▶ **Client-side validation** is a good first filter for the data. We can perform simple validation in the client and avoid unnecessary sending to the server. For each JSF UI component we have a set of JavaScript functions to perform such early validation.

Example: a good client side validation is to ensure all the obligatory fields have data to send to the server.

- ▶ **Server-side validation** is the common validation designed within JSF technology. Each UI component has its own way of validation and an error message can be associated with the validation to display the error. You can see how to define validation for a component in the JSF calculator example (Figure 3-6 on page 68).

The default validation in the JSF UI components has its limitations. As we saw in the JSF Calculator example, sometimes you have to design your own validators (see “Implementing a validator” on page 78).

Common rules for validations are:

- ▶ Perform client-side validation to avoid unnecessary requests to the server.
- ▶ Perform all the validation in the JSF UI components (server-side) to pass the most accurate data to the business objects. Remember that anybody could try to send a request to the pages programmatically without using a Web client.

Internationalization

JSF technology allows internationalization. This is done by using resource bundles. We have one properties text file for each language supported in an application.

To implement internationalization, you must declare all the supported languages in the `faces-config.xml` file. For instance, if you want the Web application to support the Spanish language, you should declare it the configuration file:

```
<faces-config>
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>es</supported-locale>
    </locale-config>
  </application>
  ....rest of config file
</faces config>
```

Then, in each JSP file source code we include the resource bundle using this JSF directive:

```
<f:loadBundle var="filename" basename="basename"></f:loadBundle>
```

The `basename` stands for the resource bundle, that is, there is a file called **basename_xx.properties** where `xx` is the language, and the request scope variable `filename` is used to access each property from the JSF UI components by using a binding of `{filename.property_name}`.

An example of the use of internationalization can be found in the JSF calculator example in “Implementing internationalization” on page 82.

You should know at design time if the Web application has the requirement of multiple languages. In that case, you have to keep in mind the design of a resource bundle that covers all the languages required for the Web application.

Custom components

If you are planning to develop new components or adapt the existent ones to your requirements, refer to Chapter 14, “Extending the JSF framework” on page 479. In this chapter you find all the information about how to develop custom components with WebSphere Studio Application Developer 5.1.2 and using them in JSF Web applications.

Custom renderers for other client platforms

If you are planning to use a JSF application with other clients, such as mobile phones and PDAs, you can change the output of the JSF application. You can do this by using a custom renderer for all the JSF UI components.

You can generate other languages than HTML. You may want to generate XML, WML, or other markup languages. Moreover, the development of a custom render allows you to generate your own language if it fits the requirements.

What are patterns?

This section is designed to help the Studio developer understand and develop JSF application using common design patterns. The patterns that are covered here have been identified as being of value to the Web application developer. First, we explain what a pattern is and define some of the characterization and classification that make up a thin client business application.

Patterns, in very simple terms, are common programming problems solved and documented for reuse. Over the years that J2EE has been available a wide variety of patterns have been designed and document to assist the developer in building applications. Patterns are not just reusable code but the design principle behind how that code is constructed.

Most J2EE developers are familiar with the model-view-controller pattern. This is a common principle for separating the data model, logic controller and the presentation layer into separate parts that can be easily developed and joined together to deliver a clean and maintainable Web application. JSF is built using this pattern and it works on top of J2EE by following the MVC paradigm.

So thinking about the characteristics of thin client business applications helps us classify these characteristics into common application models and patterns. One of the challenges of making Web application development easier is to try and create common user interface tasks. These have to be very simple to develop and allow an environment of productivity to be created. J2EE has struggled to achieve this. Although it contains the detail necessary to create these style of applications it has often been very hard to achieve the productivity. JSF is one of the first steps by the J2EE community to address these issues.

Therefore, to help us understand some of these characteristics let's define some of the common tasks that a Web application may be expected to follow:

- ▶ The requirement to manage and edit data
- ▶ The ability to view detail forms based on a search
- ▶ The ability to complete and validated a form through a wizard

If any of these application characteristics were to be developed, they would have to follow some underlying principles in application design:

- ▶ Be easy to use and understand
- ▶ Be rendered using simple HTML and JavaScript
- ▶ Map to a simple underlying data model
- ▶ Be common across application designs
- ▶ Be easily mapped to real world application use cases
- ▶ Provide good performance and not be server intensive

Based on this information we can describe some common patterns that we will document:

- ▶ Create, read, update, delete (CRUD)
- ▶ Master to detail
- ▶ Form to form wizard
- ▶ List to list

Each of these application classifications can be rendered quickly and efficiently using JSF. We now describe each pattern in more detail.

Create, read, update, delete (CRUD)

The CRUD pattern is very common in Web application designs. If an application has to maintain persistent data (for example a list of customers or list of products), a mechanism for allowing these items to be read from the database and displayed on the screen is required. We can then take these items and either update their information or delete them. Finally, we can create new items.

If you are an experienced Web developer with JSPs and servlets, you probably have developed an application that contains this pattern. You would also have found some of the complexity in doing this. With JSF the ability to map user interface components to an underlying data model and allowing a CRUD pattern to be developed is very easy. You can also include validation and converters to forms to allow even more control to be passed to JSF and away from having to write complex servlet code to manage this. This pattern could be used to maintain a list of products and manage the life cycle of creating, updating, deleting, and reading these items.

Master to detail

The master to detail pattern is often used within both thick and thin client applications. It has been used in applications designs going back to the early days of business software. The requirement to retrieve a record of data and then display the detail associated with it is a programming challenge for even today's Web applications. This pattern can be easily mapped to a many-to-one relational schema used within a database design. Let's take the example of having to find and display a specific product item from a database. We can enter a search criteria in a small form and use this criteria to construct an SQL statement that returns a result set. This result set is displayed in a table. The user then selects a product in the table to view the details of that product.

Form to form wizard

Any application that has to collect information in the style of a form could use the form to form wizard pattern. This pattern can be used to help manage the way in which a subsection of a data model is collected through a form page. This data is then validated. After successfully completing the initial form page data, the user steps forward to the next page. Finally, the form is completed and the data is submitted for processing. The key to the design is that each section of the data model is broken down into an individual page. As the user steps through these pages the data must be completed according to validation rules.

This pattern is common in online Web applications where a large amount of information is required from a user, for example, ordering a product or applying for an insurance quote. With the form data being broken into pages, the user entering the data has a better experience of the application as each stage of the form processing is managed and controlled before reaching the end.

List to list

The requirement to move data from one collection to another—where both are shown in a list box—is used widely in application designs. This pattern is often referred to as a *slush bucket* pattern or the *twin box* design. This design can be used in thick and thin application designs. You can find more detail on the UI design guidelines site for Eclipse:

<http://www.eclipse.org/articles/Article-UI-Guidelines/v200202/Contents.html#Dialogs>

This pattern is used in Web application in the configuration or selection of items that are mapped to a master data element. For example, if you are defining a sales person and you may want to add the regions that person is covering. This can be achieved with a multiple selection list box, which can be hard to manage and operate by a user. The list to list pattern gives greater flexibility and is easier to understand by the application user. They can see the items that are available and using the arrow buttons to move the selected items into their own selection.

Note: We did not implement the list to list pattern.

Example problem

To help understand these patterns further, let's explain the design behind each pattern and give an example of where it can be used. Throughout this chapter we use a scenario of a small business selling products within a sales territory.

This business has asked its Web development team to build an application that allows the display of sales data and products to the sales management team.

The team would have normally built the application with JSP, servlets and Struts, but has decided to use JSF for this Web application. Part of the reasoning for doing this is that they feel that the combination of WebSphere Studio and JSF will give them a productive platform to build modern style Web applications.

They have the identified four use cases that have to be built for the first release. They are:

- ▶ Manage the product database
- ▶ Search for specific customers and see the detail.
- ▶ Complete order entry form
- ▶ Allocate regions to a sales person

Based on our knowledge of the four key pattern described earlier can now map each use case to a particular JSF pattern (Table 6-1).

Table 6-1 Mapping use cases to patterns

Use Case	Pattern
Manage the product database	Create, read, update, delete
Search for specific customer	Master to detail
Complete Order Entry form	Form to form wizard
Allocate regions to a sales person	List to list

The rest of this chapter describes each use case in detail and shows how WebSphere Studio can be used to implement the use case using the underlying pattern.

Setting up you development environment for patterns

The pattern development examples can all be built using WebSphere Studio Application Developer 5.1.2 and JavaServer Faces. We also use Service Data Objects (SDO) to demonstrate how the patterns can link into JDBC databases.

Important: You should read Chapter 9, “SDO concepts” on page 299 before implementing these patterns on your own machine.

You have to setup the working environment for JSF development as described in “Setting up the environment for this redbook” on page 533.

To setup the pattern projects let's create an enterprise application and a Web project:

- ▶ Select *File -> New -> Project -> J2EE -> Enterprise Application Project*.
- ▶ Select *Create J2EE 1.3 Enterprise Application project*.
- ▶ Give the project the name **ItsoJSF06Pattern** and click *Finish*.
- ▶ Create a dynamic Web project named **ItsoJSFPatterns** as part of the ItsoJSF06Pattern enterprise application. We will develop the pattern examples in this project.

The pattern scenarios require database tables. We place the tables into the existing EJBBANK database for simplicity, although the pattern application has nothing to do with banking. The sample tables are described in the create, read, update and delete pattern.

Create, read, update and delete (CRUD) pattern

The CRUD pattern can be built very easily with WebSphere Studio and JSF. It shows the real power of this technology. When combined with Studio the user can complete this with a few lines of Java code and some navigation logic.

Pattern design

The CRUD pattern design works with a single relational table:

- ▶ Initially we present a list of the contents of this relational table to the user. The user can then navigate forward and backward through the table.
- ▶ Once a row has been identified for update, the user can click on the key column and view an update page.
- ▶ In the update page the user can change the values of the row and click *Update* to replace the record in the database table. The user can also click *Delete* to remove the record from the table. The user will be prompted for a confirmation to delete the row.
- ▶ The user can create a new row and insert it into the table.

Read

To help you understand how this pattern can be built, let's take you through a finished version. When you run the Web application you start with a *Read* page showing a list of records. You can navigate forward and backward using the paging control at the bottom of the product list (Figure 6-4).

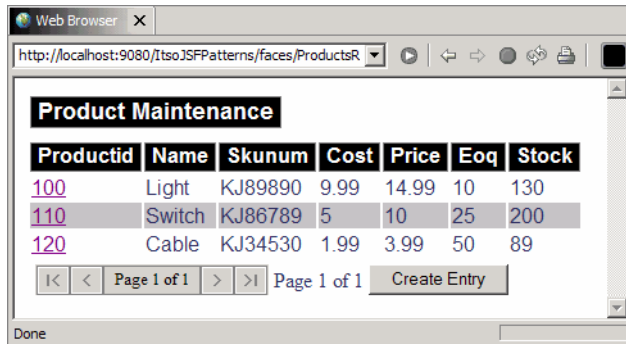


Figure 6-4 CRUD pattern: Read page

Create, update and delete

You can create a row by clicking *Create Entry* in the footer of the data list. This presents you with a blank form for entering a new record (Figure 6-5 left). The *Productid* is auto-generated, saving you to enter a value, and maintaining data integrity. Click *Create* to insert the record.

You can update a row by select a product in the *Productid* column. This action presents an update form allowing the data to be changed (Figure 6-5 right). Validation can be performed using the JSF validation tags. Click *Update* to update the data, or click *Delete* to remove the record from the table.

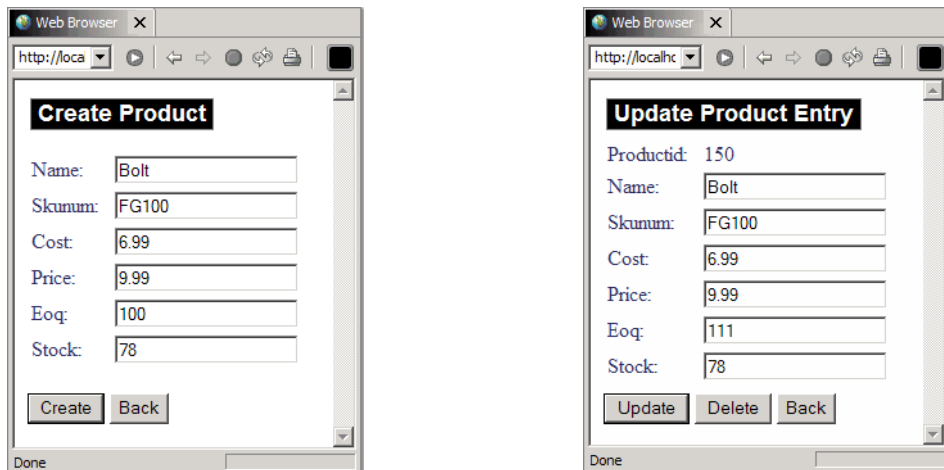


Figure 6-5 CRUD pattern: Create, update, and delete products

The sections that follow take you through a step-by-step guide on how to build this pattern.

Creating the table of products

First we define the PRODUCTS table and load sample data:

- ▶ Select *File -> New -> Other -> Data -> SQL/DDL Script*. In the wizard enter `ItsoJSFPatterns/WebContent/sql` for the parent folder. Name the SQL file `product.sql` and click *Finish*.
- ▶ You are presented with a sample script. Replace the contents of this script with these SQL statements (`\labscode\dev-patterns\product.sql`):

```
-- Drop tables --
DROP TABLE ITSO.PRODUCTS;
DROP TABLE ITSO.KEYVALUE;

--- Create PRODUCTS table ---
CREATE TABLE ITSO.PRODUCTS (
    productid  INTEGER  NOT NULL,
    name       CHAR(30) NOT NULL,
    skunum     CHAR(10),
    cost       DECIMAL(8,2),
    price      DECIMAL(8,2),
    eoq        INTEGER,
    stock      INTEGER,
    update_key INTEGER  NOT NULL,
    PRIMARY KEY (productid)
);

-- Add a few products
INSERT INTO ITSO.PRODUCTS
    VALUES(100,'Light','KJ89890',9.99,14.99,10,130,0);
INSERT INTO ITSO.PRODUCTS
    VALUES(110,'Switch','KJ86789',5.00,10.00,25,200,0);
INSERT INTO ITSO.PRODUCTS
    VALUES(120,'Cable','KJ34530',1.99,3.99,50,89,0);

-- Create an Auto Key Generating Table for the Products
CREATE TABLE ITSO.KEYVALUE
(
    appkey      INTEGER NOT NULL,
    keycounter  INTEGER NOT NULL
);

-- Add a starting key and increment
INSERT INTO ITSO.KEYVALUE VALUES(1,150);
```

- ▶ Save the `product.sql` file and execute it by selecting *Run On Database Server* (context). Run it against the EJBBANK database, using a connection named EJBBANK and the IBM DB2 APP DRIVER of DB2® Version 8.1.

- ▶ You now have a PRODUCTS table and some data to start building the sample pattern.

Note: The UPDATE_KEY column will be used for optimistic concurrency. See “Implementing concurrency” on page 255 for more details.

Creating CRUD pages

Next we create the JSF pages for the pattern. The very cool thing about JSF is that this pattern can be built around three pages. If you have been through the other examples in this book you are now very familiar with the steps for creating JSF pages in Studio. Create the following JSF pages:

```
ProductsRead.jsp  
ProductsUpdateDelete.jsp  
ProductsCreate.jsp
```

Read page with relational record list

We start to define the *Read* part of the CRUD pattern:

- ▶ Open the ProductsRead.jsp and change the text to Product Maintenance. You can apply a style later, once you have included a few JSF components.
- ▶ We have to create a link to the PRODUCTS table. In the Page Data view select *New -> Relational Record List*. Enter **productread** as name and click *Next*.
- ▶ SDO maintains connections that have been established. Because we have not established any SDO connections, we create one:
 - Click *New*, then select *New connection* and change the name to **ItsoJSFPatterns**. Click *Next*.
 - Connect to the EJBBANK database using the IBM DB2 APP DRIVER. Click *Finish*.

Tip: If you find that you have multiple connections with different names, follow the process in “Deleting SDO objects” on page 325.

- ▶ Select the ITS0.PRODUCTS table (Figure 6-6) and click *Next*.
- ▶ The final page of the dialog can be used to select columns, add relationships, and filter and order results. We do not use these facilities here. Click *Finish*.
- ▶ The relational record list appears in the Page Data view as productread (PRODUCTS).

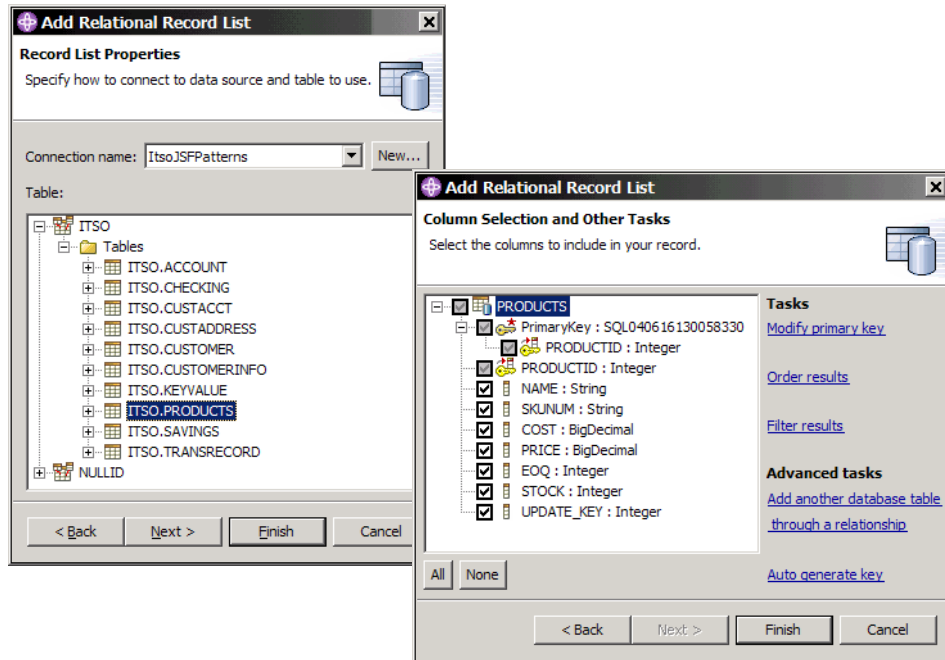


Figure 6-6 Creating a relational list

Displaying the table

We use the relational record list to display the table data:

- ▶ From the Page Data view drag the productread object onto the JSP page underneath the title.
- ▶ You are presented with a dialog allowing you to choose the columns to include in the table. Deselected the UPDATE_KEY column. Click *Finish* and the table is added to the JSP as a Data Table component (Figure 6-7).

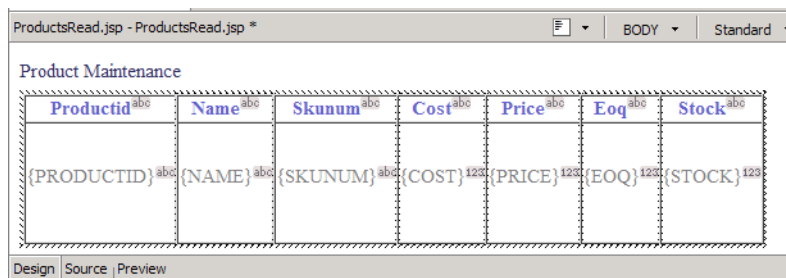


Figure 6-7 Displaying the products in a table

Now we add the features to allow navigation and row selection. Select the data table, `<h:datatable>`, and in the Attributes view:

- ▶ Change the id to `readtable`.
- ▶ On the Paging tab click *Add page statistics* and *Add deluxe pager* and adjust the Items/Page to **5**.
- ▶ Drag a Command - Button from the palette and position it behind the *Statistics* in the footer area of the table. In the Attributes view change the id to `productCreate` and its value to `Create Entry`.
- ▶ Drag a Command - Hyperlink from the palette and drop it on the first column value `{PRODUCTID}` in the table.

Select the link icon in the column and in the Attributes view Parameters tab add a parameter called `productid` and in the value field click the reference icon to select the `PRODUCTID` (Figure 6-8).

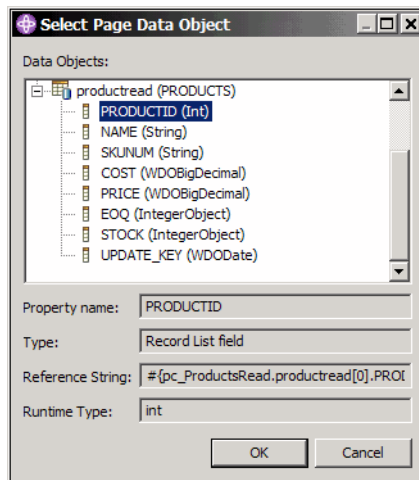


Figure 6-8 Select the product ID as parameter value

The parameter value is set to `#{varproductread.PRODUCTID}`.

Tip: This technique for referencing a row in a table has been used in the banking application. This is a small pattern that can be applied to many table selections.

Apply style changes

The out of the box style of the data table component `<h:datatable>` often needs improvements to meet the style guidelines for an organization. JSF allows easy modification of styles through CSS (cascading style sheets):

- ▶ In the Styles view open the `theme/stylesheet.css` style sheet.
- ▶ First let's allow the table to display a grey row on every odd item. This makes the table easier to read. Scroll down to the styles `.rowClass1` and `.rowClass2`:
 - Select `.rowClass1` and *Edit* (context). For fonts select the Arial font and click *Add* and *OK*.
 - Select `.rowClass2` and *Edit* (context). For fonts select the Arial font and click *Add* and *Apply*. For background select color *Silver*. Click *OK*.
 - Change the `.headerClass` to use Arial font, white foreground font color, bold, and black background color.
 - Save the style sheet.
- ▶ Select the data table in the JSP page and in the Attributes view on the All tab:
 - Scroll down to the `rowClasses` attribute and select the style editing wizard. Then expand the `theme/stylesheet.css` and select the `.rowClass1` and `.rowClass2` (Figure 6-9). Click *OK*.

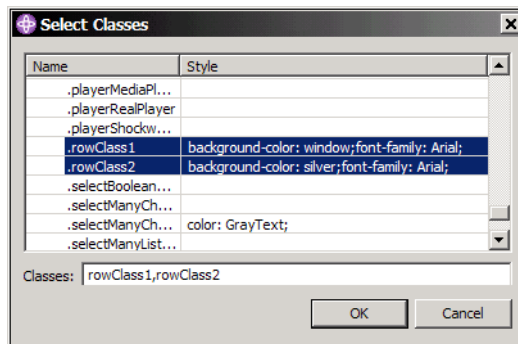


Figure 6-9 Changing style properties

- This applies the `rowClass1` style to the first row and `rowClass2` to the second row and alternates for the rest of the table.
- Using the same process change the `headerClass` to `.headerClass`.
 - ▶ We can also change the page heading (Product Maintenance) to the `.headerClass` style. Select the text and *Style -> Apply Class -> headerClass* (context).
- Also change the page heading text to an HTML heading 3 (`<h3>`). Select the text and select *Insert -> Paragraph -> Heading 3* (context).

Adding navigation to CRUD

Navigation is important to all application and within the CRUD pattern we can navigate from the `ProductsRead.jsp` to the `ProductsUpdateDelete.jsp` and `ProductsCreate.jsp`:

- ▶ Select the hyperlink icon or the *Create Entry* button. In the Navigation tab of the Attributes view add three navigation rules (Figure 6-10):
 - `ProductsRead.jsp`, alias: **read**, scope:*
 - `ProductsCreate.jsp`, alias: **create**, scope:*
 - `ProductsUpdateDelete.jsp`, alias: **update**, scope:*

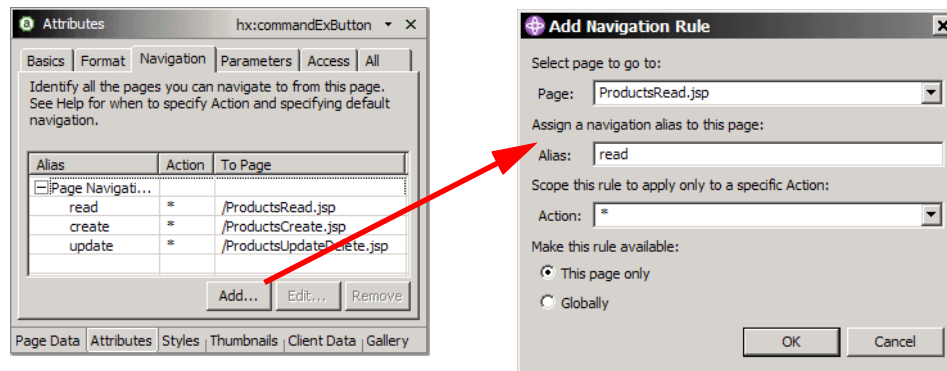


Figure 6-10 Adding navigation rules

- ▶ Activate the create navigation rule by selecting the Create Entry button and in the Quick Edit view select *Command* and enter the code:

```
return "create";
```

- ▶ Activate the update navigation rule by selecting the hyperlink and in the Quick Edit view select *Command* and enter this code (`\\labscode\\dev-patterns\\productHyperLink.txt`):

```
// Type Java code to handle command event here
// Note, this code must return an object of type String (or null)

String id = ((Integer)getParam1().getValue()).toString();
// Check if we have a valid row
if (id == null)
    return "read";
// Place the Id in the Session
getSessionScope().put("productid",id);
// Navigate the Update Page
return "update";
```

Note: So far this is the only Java code that has to be added.

- Finally we define the session variable for the productid. In the Page data view select *sessionScope* and *Add Session Variable*. Enter productid as variable name and java.lang.String as type.

The rest you can design to your own preferences for styling or extra features you may want to display in the table. We now have a table which supports paging and allows you to select a row and navigate to the ProductsUpdate.jsp page.

Building the update/delete part of CRUD

The key element for *Update/Delete* part of the CRUD pattern is to select an individual row from the table based on the productid to trigger the navigation.

Open the ProductsUpdateDelete.jsp:

- In the Page Data view create a Relational Record and call it productupdate. Click *Next*.
- Reuse the ItsoJSFPatterns connection and select the ITS0.PRODUCT table. Click *Finish*.
- We have to modify the default filter condition (an SQL WHERE clause). We use the productid session attribute to select an individual row.

In the Page Data view select the productupdate bean and *Configure* (context). On the Conditions tab select the generated condition and click the *Edit* icon to change the condition. Change the value of the condition to `# {sessionScope.productid}` (Figure 6-11). Click *Close*.

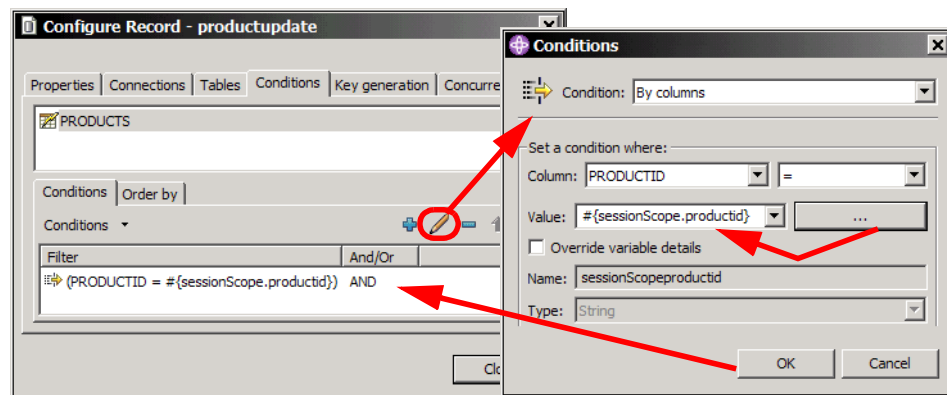


Figure 6-11 Adding the condition to the Record List

To display the record for updating, you could manually create all the labels and fields within an HTML table. Application Developer makes it much simpler. Drag the productupdate record into the JSP:

- In the dialog select *Update an existing record*.
- For the PRODUCTID select *Output Field* (the key cannot be modified).
- Then click *Options*, select both *Submit* and *Delete* buttons, and change the Submit button text to Update. You could also change the labels.
- Click *Finish* (Figure 6-12).

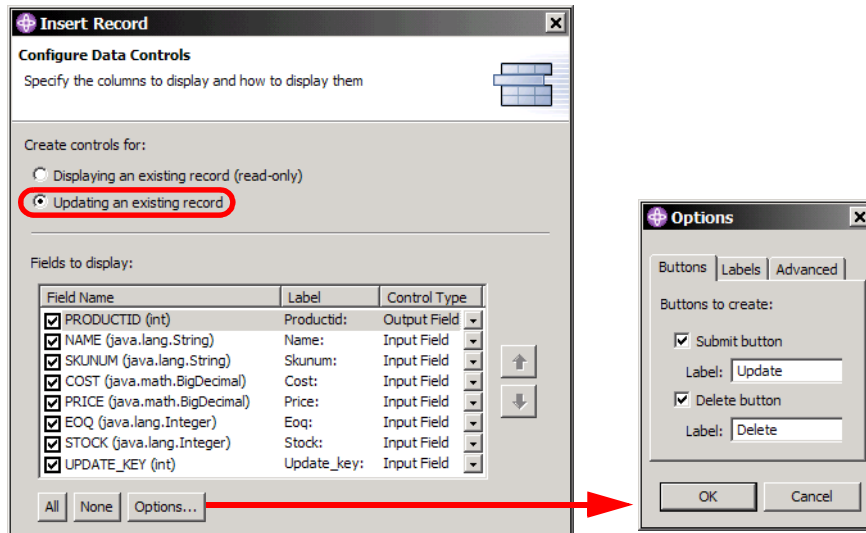


Figure 6-12 Tailoring the update/delete JSP

The form to update a record is automatically generated for you. It is that easy. Let's clean up the page:

- Select the {UPDATE_KEY} input field and in the Attributes view Behavior Tab select *Control is read-only* (this field is for concurrency control).
- To make the errors stand out select the {Error Messages} JSF component and change the color to red (Attributes view, Properties: color: red).
- To prevent a record from being deleted without a user prompt to confirm the deletion, select the *Delete* button and add a JavaScript:

In the Quick Edit view select *onclick* and add the following JavaScript code (`\labscode\dev-patterns\productDeleteScript.txt`):

```
var agree = confirm("Are You sure you want to delete this entry ?");
if ( agree)
    return true;
else
    return false;
```

- To help navigation drag a Command - Button onto the page next to the Delete button and call it Back. In the Quick Edit view set the code to return "read"; and save.

Select the Update or Delete button and study the Quick Edit view for the Command event. The code to apply the database updates has been generated.

- Add a navigation rule read to go back to the ProductsRead.jsp. Change the return statement in both action code methods to return "read";.

Update:

```
try {
    getProductupdateMediator().applyChanges(
        getProductupdate().getDataGraphAccessBean());
} catch (Throwable e) {
    logException(e);
}
return "read";
```

Delete:

```
try {
    getProductupdate().getDataGraphAccessBean().deleteDataObject();
    getProductupdateMediator().applyChanges(
        getProductupdate().getDataGraphAccessBean());
} catch (Throwable e) {
    logException(e);
}
return "read";
```

- Finally let's change the default heading text to Update Product Entry, apply the .headerClass, and make it an <h3> paragraph.

Make sure everything is saved. You have now completed the creation of the update/delete part of the CRUD pattern.

Building the create part of CRUD

The final part for the pattern to be built is the *Create* part that is used to insert a product. This is the simplest operation, the combination of the relational record and JSF combined with the intuitive Studio integration makes this very easy to complete.

Open the ProductsCreate.jsp:

- Replace the default text with Create Product. Notice that you cannot apply any styles from the StyleSheet.css file. The reason is that the style sheet has not been added to the page. This automatically gets done when a JSF

component is dropped on the page. Otherwise you can do it manually. We will format the heading later.

- ▶ In the Page Data view create a Relational Record called `productcreate`. Select *Create an empty record in order to create a new row in the database* and click *Next*.
- ▶ Reuse the `ItsoJSFPatterns` connection and select the `ITSO.PRODUCTS` table. Click *Next*.
- ▶ The creation operation of a table can be made easier by automatically generating the key value for the primary key column:
 - Select *Auto generate key*.
 - Select *Use Auto Key Generation* and select the `ITSO.KEYVALUE` table.

This table has been created to store the application key counters. For the identity column, select `APPKEY`, and for the incrementor column, select `KEYCOUNTER`. Change the Keys fetched at once to **10** (Figure 6-13).

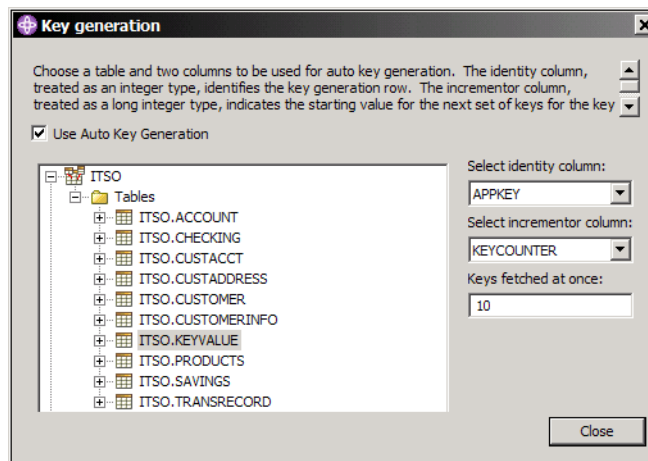


Figure 6-13 Setting up auto key generation

- ▶ Click *Finish* to add the relational record to the Page Data view.
- ▶ Now we create the form on the page. Drag the `productcreate` object into the JSP underneath the header text. A dialog appears allowing some final settings to be changed before the form is generated (Figure 6-14).

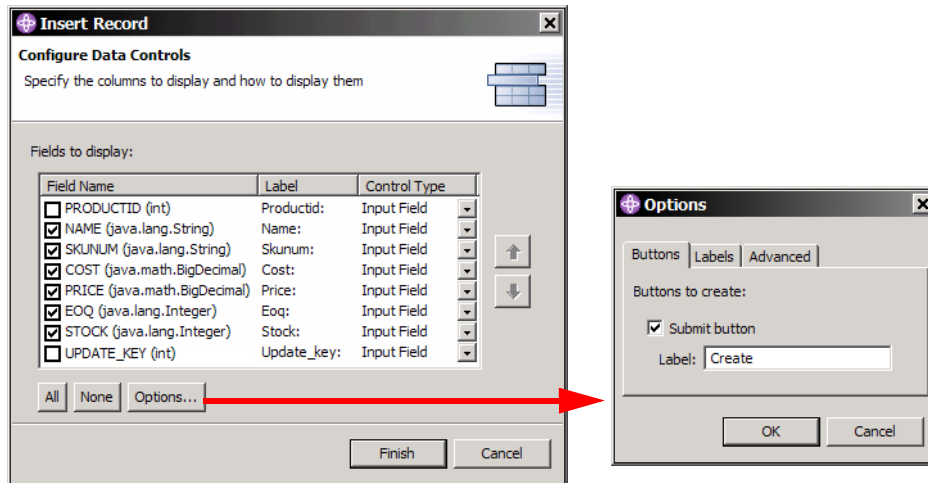


Figure 6-14 Tailoring the create product table

- Deselect the PRODUCTID and UPDATE_KEY columns (both are generated columns) and in the *Options* dialog change the Submit button to **Create**. Click *OK* to save the changes and then click *Finish*.
- The form is generated on the page. Apply the same style settings as for the update/delete page (heading 3, {Error Messages} in red).
- Select the Create button and study the Quick Edit view code. The key is generated and the record is inserted:

```
try {
    getProductcreateMediator().autoGenerateKey(getProductcreate());
    getProductcreateMediator().applyChanges(
        getProductcreate().getDataGraphAccessBean());
} catch (Throwable e) {
    logException(e);
}
return "read";
```

Add a navigation rule read to go back to the ProductsRead.jsp. Change the return statement in the action logic to return "read";.

- You also want to allow the user to navigate back to this page even if a create is not complete. Drag a Command - Button next to the Create button, set its label to Back, and set the action code to return "read";.
- Save the page. You can now start looking at testing.

Testing the CRUD application

To test this pattern select the `ProductsRead.jsp` and *Run on Server*. Select the JSFServer. The data source is added to the server and the read page is displayed. You can select a row to update its values or create a new entry in the PRODUCTS table (see Figure 6-4 and Figure 6-5 on page 243).

Note: If you do not change any data and click *Update*, no SQL UPDATE statement is issued. SDO compares the data from the GUI to the data object.

Implementing concurrency

Application Developer's Service Data Objects provide an optimistic concurrency function. A table column can be designated for optimistic concurrency with the effect that it is incremented automatically on each update.

In the PRODUCTS table we setup the **UPDATE_KEY** column for this purpose. Such a column must be defined as `INTEGER NOT NULL`.

The concurrency implementation of SDO checks that the `UPDATE_KEY` column has not been modified between displaying the data and performing the update.

Updating the update/delete function

In the update/delete function we have to setup optimistic concurrency so that SDO can check that the `UPDATE_KEY` column value has not been modified:

- ▶ Open the `ProductsUpdateDelete.jsp`.
- ▶ In the Page Data view, select the `productupdate` object and *Configure*. On the Concurrency tab select the PRODUCTS table and select the `UPDATE_KEY` column as the collision column (Figure 6-15). Save the `ProductsUpdateDelete.jsp`.

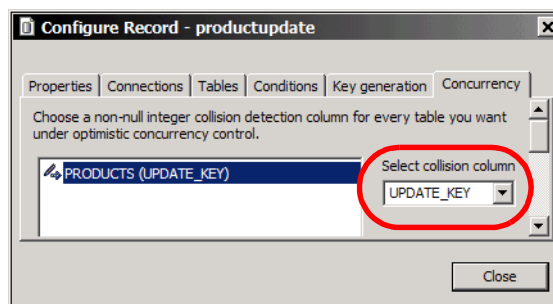


Figure 6-15 Defining the concurrency column

Testing concurrency

To test the concurrency implementation, run the `ProductsRead.jsp`, then select a product. Change some data and click *Update*. Select the same product again and you can see that the `UPDATE_KEY` value has been incremented by 1.

Click *Update* without changing any data. Select the same product again and you can see that the `UPDATE_KEY` has not been modified.

Next change some data, but before clicking *Update*, change the `UPDATE_KEY` column value in the database using a DB2 Command window, for example:

```
db2 connect to ejbbank
db2 update itso.products set update_key=9, stock=140 where productid=100
```

Alternatively open another browser window, run the same application, and change the same product:

```
http://localhost:9080/ItsoJSFPatterns/faces/ProductsRead.jsp
```

You can see an error message trace in the Console and the message:

```
com.ibm.websphere.wdo.mediator.rdb.exception.OCCException:
The update attempt collided
```

Improving error messages

The application user does not get any feedback about the concurrency error (or any other exceptions). We can improve the design by giving the user an error message and retrieve the latest data:

- In the `ProductsUpdateDelete.jsp` change the action code of the Update button (`\labscode\dev-patterns\productUpdateButton.txt`):

```
try {
    getProductUpdateMediator().applyChanges(
        getProductUpdate().getDataGraphAccessBean());
} catch (OCCException e) {
    FacesMessage msg = new FacesMessage
        ("Data was updated by another user - new values are displayed");
    getFacesContext().addMessage("",msg);
    productupdate = null; // force a reread of the data
    getProductUpdate();
    return "";
} catch (Throwable e) {
    FacesMessage msg = new FacesMessage
        ("Update failed: "+e.getMessage());
    getFacesContext().addMessage("",msg);
    productupdate = null; // force a reread of the data
    getProductUpdate();
}
```

```
        logException(e);  
        return "";  
    }  
    return "read";
```

Select *Source* -> *Organize Imports* to resolve the missing classes.

- A nice error message can also be added to the Delete button if the delete operation fails and in the ProductsCreate.jsp for the Create button.

Retest the concurrency by changing the UPDATE_KEY value and you receive the nice error message that the data was changed by another user.

Note: The user should not see the UPDATE_KEY column in the update/delete JSP. This field is for internal concurrency control. In a real application the field should be made a *hidden* field instead of an input field.

Summary

As you have worked through these steps to create this example of the CRUD pattern you should have noticed that very little Java code was required and how integrated JSF and relational database access is in the WebSphere Studio workbench. This combination of tooling and rich client framework allows these type of patterns to be made more reusable and component-oriented.

Form wizard pattern

The form wizard pattern can be used to collect data in separate logical pages. These pages can then be individually validated before passing processing onto the next page in the form. This style of forms are often used when a large amount of data has to be entered. Having the data broken into separate pages allows the validation to be perform in stages rather than all together at the end. If you have ever applied for a loan, car insurance, or a financial product you have probably been through one of these types of online form.

With the powerful validation and navigation features of JSF this type of pattern is easy to implement.

Pattern design

The form wizard pattern design is based on the requirement for a product order to be entered into the application. The order is broken into three main parts:

- Entering the name and address of the company or person placing the order
- Entering the actual order of the product

- Entering the delivery address

Once all these pages have been completed and validated, the order is submitted to the server for processing.

In this example we are not using server-side persistence but store all the data of the form in a JavaBean. This demonstrates how easy it is to use a single model class and map its data items to validating forms on multiple pages.

The pattern solution is built with four Faces pages. Each page forwards to the next page using JSF navigation. Each page has validation so that the fields of data are validated before the navigation occurs. The final page demonstrates how to pull data from other parts of the model to populate the fields of the page.

So let's take the example of ordering a product through a Web interface. We have a data model that holds all the items of data required to process the product order, implemented as a JavaBean. Three JSF pages are used to collect the data and a fourth page displays all the data before submitting the order:

- Enter the name and address information. The name and city are validated.
- Enter the order details. The product name and order number are validated (Figure 6-16).

The figure illustrates the first two steps of a form wizard pattern for a product order. The left panel, 'Product Order - Enter Address Details', shows a form with fields for Name, Address (three lines), City, and Postcode, all filled with sample data. A 'Next >' button is at the bottom. A red arrow points from this button to the right panel. The right panel, 'Product Order - Enter Order Details', shows a form with fields for Product Name, Order Number, Price, and Value, also filled with sample data. It has '< Back' and 'Next >' buttons at the bottom. A red arrow points from the 'Next >' button to the right, indicating the flow to the next step.

Figure 6-16 Form wizard pattern example (1)

- Enter the delivery address. Is it very common for these types of Web user interface to have a check box that asks if the delivery address is the same as the business or home address. In this page we add the business logic to pull the values from the data model.
- Verify the order before submitting (Figure 6-17).

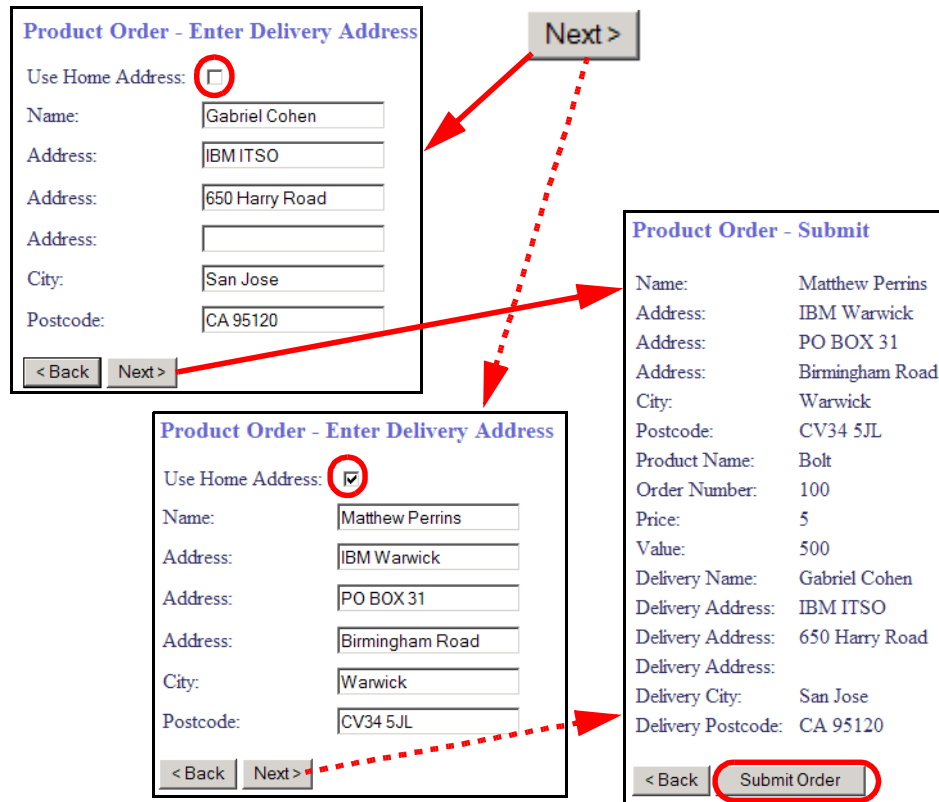


Figure 6-17 Form wizard pattern example (2)

Having a pattern described in JSF allows developers to understand the steps necessary to build these style of Web applications.

Building the form wizard pattern

In the sections that follow we describe the steps required to build this sample pattern in WebSphere Studio and JSF. Very little code is required, and where it is necessary it is for business logic.

This pattern could be extended to support the persistence of the form data after completing each page and a more detailed set of validation rules. Once you have this working you can then extend the sample to meet the requirements of the application you are developing.

Creating the JavaBean

We use the same Web project, *ItsoJSFPatterns*, for this example:

- ▶ Under *JavaSource*, create a package named **itso.formwizard**. In the package create a class named **ProductOrder**.
- ▶ Add the following properties to the class (*\labscode\dev-patterns\ProductOrder.txt*):

```
// Customer Details
String name;
String address_1;
String address_2;
String address_3;
String city;
String postcode;

// Product Details
String productname;
int ordernum;
int price;
int value;

// Delivery Address
boolean usedeliveryaddress;
String delName;
String delAddress_1;
String delAddress_2;
String delAddress_3;
String delCity;
String delPostcode;
```

- ▶ Select *Source -> Generate Getter and Setter* and in the dialog select all the fields and method and click *OK*. The data model class is now complete and ready to be used within the JSF form pages.
- ▶ Save and close the JavaBean.

Creating the form pages

The form wizard is built from a set of individual pages that can be navigated forward and backwards. Create the following JSF pages:

```
FormAddress.jsp
FormOrder.jsp
FormDeliveryAddress.jsp
FormSubmit.jsp
```

- Open the `FormAddress.jsp` page and in the Page Data view define the JavaBean (*New -> JavaBean*) with a name of `productorder` and make it reusable in the session scope (Figure 6-18).

Add JavaBean

Select JavaBean
Select whether to create a new JavaBean or use an existing one.

☒ Add new JavaBean ☐ Add existing reusable JavaBean

Name:

Class:

☒ Make this JavaBean reusable

Description:

Scope:

Figure 6-18 Adding the `ProductOrder` bean to the session

- In the other three pages you add the `ProductOrder` JavaBean by selecting *Add existing reusable JavaBean* (Figure 6-19).

Add JavaBean

Select JavaBean
Select whether to create a new JavaBean or use an existing one.

☐ Add new JavaBean ☒ Add existing reusable JavaBean

Existing reusable JavaBeans:

Name	Description	Class	Scope
productorder		itso.formwizard.ProductO...	session

Figure 6-19 Add Existing managed bean to other pages

- Change the titles for each page as a heading 3 (select the text and *Insert -> Paragraph -> Heading 3*).

FormAddress.jsp	Product Order - Enter Address Details
FormOrders.jsp	Product Order - Enter Order Details
FormDeliveryAddress.jsp	Product Order - Enter Delivery Address
FormSubmits.jsp	Product Order - Submit

- Now that the pages are all prepared let's start to define the forms. One of the great features in Studio is its ability to allow JavaBeans and other components to be dragged and dropped onto the page and the fields to be automatically added. Even more powerful is Studio's ability to only select the fields you want to use from a given bean and then drag these selected fields to the page.

Defining the form address page

To complete the FormAddress.jsp:

- Select the required fields from the productorder bean and drop them into the JSP (Figure 6-20).

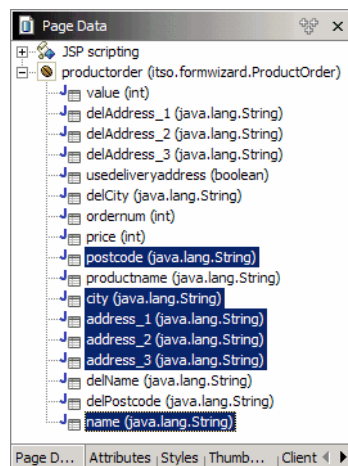


Figure 6-20 Selecting the address fields from the bean

- Change the order of the fields using the up and down arrow buttons, and change the labels (Address). Select *Updating fields* and click *Options* to change the Create button to **Next >**. Click *Finish* (Figure 6-21).

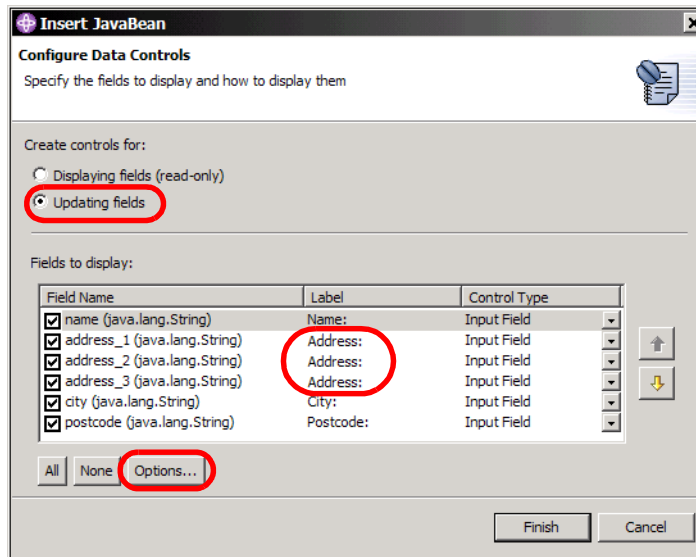


Figure 6-21 Ordering the selected fields

- ▶ The forms in this wizard do not use a general error message area. Instead we will modify the validation to add an error message field for each validated field. So select the {Error Message} component and delete it.
- ▶ In the Attributes view change the id of the input fields to correspond to the data fields they are bound to (name, address1, address2, address3, city, postcode).
- ▶ We want to validate the name and city fields. This is enough to prove the validation mechanism for this pattern:
 - Select the name field and in the attributes view select the *Validation* tab. Select *Value is required* and *Add Display Error control*. An {Error Message for name} field appears next to the input field.
 - Repeat this for the city field.
 - Note that there could be more validation rules such as minimum and maximum length of fields.
- ▶ To set all error messages to red color, modify the .message style. In the Styles view expand theme/stylesheet.css and modify the .message to have a red foreground color.

Adding navigation

To make the form wizard behave like a normal wizard we use **Next** > and/or < **Back** buttons on all pages. We also define global navigation rules.

Note: Global navigation rules apply to all pages. In our Web application we implement multiple patterns and the global rules apply to all patterns, even where they do not make sense. In a real application you would only have one pattern.

- ▶ Select the *Next >* button and define the global (select *Globally*) navigation rules (Figure 6-22):
 - Alias pageOne to FormAddress.jsp
 - Alias pageTwo to FormOrder.jsp
 - Alias pageThree to FormDeliveryAddress.jsp
 - Alias complete to FormSubmit.jsp

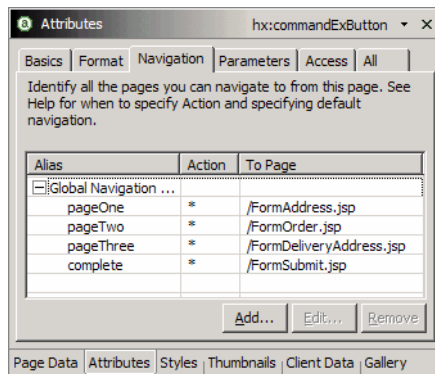


Figure 6-22 Global Navigation rules for the form wizard pattern

- ▶ To make the *Next >* button move to the correct page, enter the return statement in the Quick Edit view:


```
return "pageTwo";
```

This tells JSF to move to the page defined with the navigation rule pageTwo.
- ▶ Finally to make sure the buttons are displayed in the same place on each page. Select the table of the form input fields and change the height of the table to **250** pixels.

Note: This is not a recommended technique for setting fixed sized tables. It is only being used in the sample for ease of display. We would recommend that an experienced page designer lays out the pages ready for JSF controls to be added to them.

Completing the other pages of the Wizard

Now you have been through the process of creating the first page. Complete the order, delivery address, and submit pages of the form wizard in the same way.

Order page

To complete the order page perform these steps:

- ▶ Open the `FormOrder.jsp` page and drag the fields from the `productorder` bean to the page and order the fields:

Field	Label
productname	Product Name:
ordernum	Order Number:
price	Price:
value	Value:

- ▶ Select *Options* and make a < **Back** button.
- ▶ Set the field id to the field names.
- ▶ Delete the {Error Message} component.
- ▶ Add validation for the `productname` and `ordernum` fields. Select *Value is required* and *Add Display Error control*.
- ▶ Change table height to 250 pixels.
- ▶ Add a **Next** > button.
- ▶ Set the action code for the < Back button to return "pageOne" and for the Next > button to return "pageThree".
- ▶ Select the < Back button and in the Attributes view All tab set the immediate attribute to True. This enables going back even if there are validation errors.

Delivery address

To complete the delivery address page perform these steps:

- ▶ Open the `FormDeliveryAddress.jsp` page and drag the fields from the `productorder` bean to the page and order the fields:

Field	Label	ID
usedeliveryaddress	Use Home Address:	usehomeaddress
delName	Order Number:	delName
delAddress_1	Address:	delAddress1
delAddress_2	Address:	delAddress2
delAddress_3	Address:	delAddress3
delCity	City:	delCity
delPostcode	Postcode:	delPostcode

Note: `usedeliveryaddress` should really be named `usehomeaddress`.

- ▶ Select *Options* and make a < **Back** button.
- ▶ Set the field id to the values indicated above.
- ▶ Do **not** delete the {Error Message} component. We will validate that the delivery address is either copied or entered by hand.
- ▶ Change table height to 250 pixels.
- ▶ Add a **Next** > button.
- ▶ Set the action code to return "pageTwo" and return "complete".

Adding business logic

To make this page a little more interactive, we are pulling the address details from the underlying model if the usehomeaddress check box is selected.

- ▶ Select the check box and in the Quick Edit view add the following code to the *onclick* event. This JavaScript code forces the selection event to trigger a submit event with the effect that we can add business logic to the *valueChange* event of the component.

```
form1.submit();
```

- ▶ Select the *Value Changed* event and add the logic to set the values of the delivery address fields based on the customer home address details:

```
// Retrieve the value of the check box
Boolean checked = (Boolean)valueChangedEvent.getNewValue();

// Get a handle to the ProductOrder model
ProductOrder po = getProductorder();

// Check the state of the check box.
// If checked then display the address otherwise reset the values.
if (checked.booleanValue()) {
    getName().setValue(getProductorder().getName());
    getAddress1().setValue(getProductorder().getAddress_1());
    getAddress2().setValue(getProductorder().getAddress_2());
    getAddress3().setValue(getProductorder().getAddress_3());
    getCity().setValue(getProductorder().getCity());
    getPostcode().setValue(getProductorder().getPostcode());
} else {
    getName().setValue("");
    getAddress1().setValue("");
    getAddress2().setValue("");
    getAddress3().setValue("");
    getCity().setValue("");
    getPostcode().setValue("");
}
```

- Select the Next > button and modify the action logic to verify that a delivery address was entered:

```
Boolean checked = (Boolean)getUsehomeaddress().getValue();
if (getDelName().getValue().equals("") ||
    getDelAddress1().getValue().equals("") ||
    getDelCity().getValue().equals("") ||
    getDelPostcode().getValue().equals("") ) {
    FacesMessage msg = new FacesMessage
        ("Delivery address missing");
    getFacesContext().addMessage("", msg);
    return ""; // stay on the page

    return "complete";
}
```

- Save the page.

Note: Setting *Value is required* for the entry fields makes the check box ineffective—the copied values are not displayed.

Completing the form wizard pattern

Now that the main form pages are complete update the `FormSubmit.jsp` page to display the values that have been entered into the form. This page is very typical in form applications. It allows the user to see what has been entered. If the user is happy with the details the values can be submitted to the application for processing.

- Open the page `FormSubmit.jsp` and drag the `productorder` bean from the Page Data view to the JSP. Select *Displaying fields*. Deselect the `usedeliveryaddress` field. Order the fields and change the labels.

The order of the fields is:

```
Name, Address (3), City, Postcode
Product Name, Order Number, Price, Value
Delivery Name, Delivery Address (3), Delivery City, Delivery Postcode
```

- Delete the {Error Message} component.
- Change table height to 250 pixels.
- Add a < **Back** and a **Submit Order** button.
- Set the action code of the < Back button to return "pageThree".
- Set the action code of the Submit Order button to:

```
log("Processing the order");
// add the business logic here
return "";
```

- Save the page. You are now ready to test the form wizard pattern.

Testing of form wizard pattern

Having completed the pattern you can see that implementing Web forms with JSF is very easy. The validation rules can be mostly handled by the JSF validation tags and the fields for each form page can be easily bound to a JavaBean that contains all the data for the form. Once the form has been completed this JavaBean is passed to the server for processing.

Test the form wizard pattern by selecting the `FormAddress.jsp` and *Run on Server*. Progress through the dialog and test each step. Leave fields empty to test validation. Once a page has been successfully validated, navigate to the next page in the form wizard. Notice that the delivery address is populated from the initial address when you click the check box.

Master to detail pattern

The master to detail pattern is used widely in thick client applications. Using JSF and its event model it is now very easy to implement this pattern with very limited Java programming. This pattern can be used to help the user see detailed information after performing a query.

Pattern design

The master to detail pattern design works around a single relational database. For this example we work with the `PRODUCTS` table used in the CRUD pattern. The key differences in this pattern to CRUD is that you can perform a search and view the results and the details all on the same page. The pattern is built around a single JSF page and involves the interaction of three sections of the same page to deliver the solution to the user.

Let's explain how the pattern works. First, the user enters a search criteria into an input field and click *Search*. The results of the search are displayed in a table with only the key field and the product name. This is enough information for the user to make a decision to then view the details for any given item in the table.

To view the details the user clicks on a row in the table and the details are displayed in a read-only form that is positioned on the side the results table. This enables the user to quickly select other items on the result set and view them.

All the records of the data table are displayed if the user clicks *Search* without a search criteria. If the number of rows returned from the database is greater than a set limit the data table paging navigator is displayed.

Working pattern

Before you start constructing this pattern let walk you through a working version:

- ▶ When starting the application nothing is retrieved (Figure 6-23).

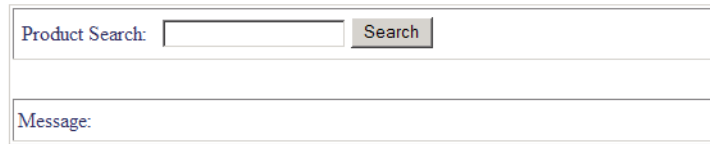


Figure 6-23 Master to detail completed pattern view

- ▶ Click *Search* with or without a search criteria to retrieve matching products. A % sign for the LIKE clause is added automatically. The deluxe pager is displayed if there are too many rows (Figure 6-24).



Figure 6-24 Product list after search

- ▶ Select an ID to display the details (Figure 6-25).

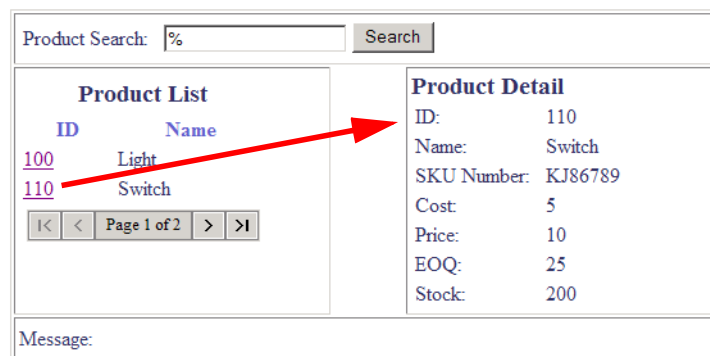


Figure 6-25 Product list with details

This pattern can be used to help a user see a wide variety of information in a single page using an intuitive user interface design. This prevents the user from having to continually navigate back and forth between pages.

Building the master to detail pattern

In the sections that follow we describe the steps required to build this sample pattern in WebSphere Studio and JSF. The master to detail pattern is all built around a single JSF page:

- ▶ Create a JSF page called **ProductView.jsp**.
- ▶ Change the default text to Product View and change the style to a heading 3.
- ▶ Save the page.

Creating the queries for the pattern

The master to detail pattern is based around PRODUCTS table that was previously defined in the CRUD pattern. We use two SDO objects to access the table. The first object retrieves a list of products, while the second returns a single row.

- ▶ In the Page Data view define a requestScope variable named productName of type java.lang.String. This will be the search value.
- ▶ In the Page Data view create a Relational Record List object and call it productsearch. Reuse the existing database connection and select the ITS0.PRODUCTS table. Select only the PRODUCTID and NAME columns (Figure 6-26).
- ▶ Click *Filter results* and define a filter that performs a match of the NAME with an SQL LIKE clause, using the productName request scope variable.

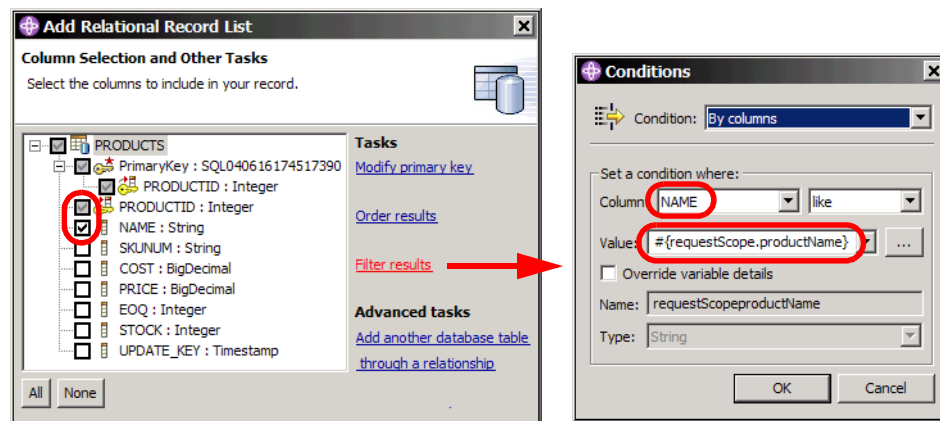


Figure 6-26 Product search query with filter on name

- ▶ Click *OK* to return to the Filters dialog. Select *Custom* in the Conditions drop-down menu to see the SQL condition:


```
(NAME like ?) <=> requestScopeproductName = #{requestScope.productName}%
```
- ▶ Close this dialog and click *Finish* to define the object in the Page Data view.
- ▶ Now we define an object that retrieves one record. Add a Relational Record named `productdetail`, using the `ITS0.PRODUCTS` table. Deselect the `UPDATE_KEY` column. A default filter is created with the condition:


```
(PRODUCTID = #{param.PRODUCTID})
```

We pass a parameter named `PRODUCTID` to this query.
- ▶ Click *Finish* and you have two data objects in the Page Data view.

Laying out the page

The master to detail pattern can be built into one JSF page. To help with the layout we use an HTML table to hold the sections of the pattern:

- ▶ Drag a table from the HTML Tags palette into the JSP. Select three rows and three columns, border 1 and cellpadding 3.
- ▶ Select the cells in the top row and *Table -> Join Selected Cells*. Do the same for the bottom row. Select the borders to make the table bigger (Figure 6-27).

Product View



Figure 6-27 Laying out the Master to detail pattern

- ▶ Drag the `productName` request scope variable from the Page Data view into the top row of the table. When prompted select *Updating fields*, change the label to `Product Search:`. Click *Options* and change the button label to `Search`.
- ▶ Delete the `{Error Messages}` component and move the `Search` button after the input field.
- ▶ Drag the `productsearch` object from the Page Data view into the left cell of the middle row of the table. When prompted change the label of the `PRODUCTID` field to `ID`. Click *Finish* to add the data table inside the JSP table.
- ▶ Select the data table and in *Attributes view* *Page* tab change the *Items/Page* to **2** and click *Add deluxe pager*.

Note: The reason for setting the pager size so low is to demonstrate that the pager works in this pattern example.

- ▶ Select the layout table and modify the cell attributes so that all cell contents are aligned horizontally to the left and vertically to the top.
- ▶ To complete the layout in the third column of the second row drag a JSF Panel - Group Box component into the cell. When prompted select *HTML Panel*. Next drag a JSF Output component into the panel, then change the value of the control to `Product Detail` and change its font size to 14 and bold (in Properties).
- ▶ Drag the `productdetail` object from the Page Data view underneath the text into the panel. When prompted select *Displaying an existing record* and change the labels to suitable text. Click *Finish*.
- ▶ Drag the generated `{Error Messages}` component into the bottom row of the layout table and change its style to `color: red`. Drag an Output component in front of the `{Error Messages}` component and set the value to `Message:`.
- ▶ Select the three middle cells (one at a time) and change the width to **45%**, **10%**, and **45%**. Select the layout table and change the width to **100%**. The table adjusts now to the screen size.
- ▶ The final layout is shown in Figure 6-28.

Product View

Product Search: {productName} Search

ID	Name
{PRODUCTID}	{NAME}

Product Detail

ID:	{PRODUCTID}
Name:	{NAME}
SKU Number:	{SKUNUM}
Cost:	{COST}
Price:	{PRICE}
EOQ:	{EOQ}
Stock:	{STOCK}

Message: *{Error Messages}

Figure 6-28 Final layout of the master detail pattern

- ▶ Select the deluxe pager and in the Attributes view All tab set `rendered` to `False`. By default we do not display the pager. We set it to `true` in the action logic.

- ▶ Select the data table in the left column and in the Attributes view All tab set rendered to False. We set it to true when the Search button is clicked.
- ▶ Select the panel in the right column (select the Product Detail text and in the Attributes view select hx:jspxPanel in the pull-down on top). Set the id to productPanel and set rendered to false (All tab). We set it to true when a product is selected in the product list.

You can now start to add the event logic that manipulates the record list and record objects.

Adding event logic

The key to the master to detail pattern is that it manipulates the relational record list (product list) and relational record (product detail) from its page code Java class. There are two steps for doing this:

1. Take the input search criteria and apply it to the productsearch object.
 2. Take a row that is selected in the results data table and display its value in the product detail area of the page.
- ▶ Select the Search button and within the Command event code section in the Quick Edit view add the following code
(\labscode\dev-patterns\ProductSearch.txt):

```
// retrieve the search argument
String search = (String)getRequestScope().get("productName");
if (search.length() == 0 || !search.substring( search.length()-1 )
                                     .equals("%"))
    search += "%";
log("Search on: "+search);
getRequestScope().put("productName",search);
productsearch = null;           // force new retrieve
getProductsearch();

// setup pager if more rows than limit
if (getProductsearch().size() >= getTable1().getRows())
    getDeluxe1().setRendered(true);
else
    getDeluxe1().setRendered(false);

// display the product list but not the details
getTable1().setRendered(true);
getProductPanel().setRendered(false);

return "";
```

This code adds a % sign to the parameter (if none is there). Then a retrieve is forced by setting the productsearch variable to null. If there are more rows than the limit, the deluxe pager is displayed.

- Select the hyperlink and in the Quick Edit view add the code to force a retrieve of the product details:

```
log("Hyperlink="+getRequestParam().get("PRODUCTID"));  
// retrieve and display the details  
getProductdetail();  
getProductPanel().setRendered(true);  
return "";
```

- Save the page.

You are now ready to test the pattern and check that it behaves as designed.

Testing

To test the master to detail pattern, select the `ProductView.jsp` and *Run on Server*. Wait for the page to be displayed on the screen. You are now in a position to test some of the logic we have built into the pattern.

This pattern demonstrates some key concepts on how to manipulate data, refine search criteria, and redisplay the same page with different elements selected. This pattern could be extended and modified to offer a wide range of features to support the requirements of your J2EE application.

Complete solution

We provide the complete solution for this examples in the additional material in the file `\labscode\solution\ItsoJSF06Pattern.ear`. See “Importing a solution enterprise application” on page 548 for instructions.

Summary

As you can see by these pattern examples, JavaServer Faces is very powerful in helping a Web developer create interactive applications with little or no knowledge of the detailed J2EE specifications or programming APIs. These three patterns only start to scratch the surface of what can be developed using JSF. Using a variety of the common themes that have been demonstrated in the chapter enables developers to start to build applications using the combined power of WebSphere Studio and JavaServer Faces.

In the process of creating these patterns the following key concepts for JSF have been covered:

- ▶ How to integrate SDO into JSF pages.
- ▶ How to use JavaScript to trigger events.
- ▶ How to use validation.
- ▶ How to use page navigation.
- ▶ How to integrate SDO into JSF pages, with retrieve, update, delete, and insert operations.

With a wider set of JSF components being developed by tooling vendors and the open source community, more complete and complex patterns can be developed to help the development community understand how to best to use this technology.

The patterns in this chapter were chosen to help developers with little or no knowledge of J2EE to understand how to use WebSphere Studio to build a functional component of a wider Web application. JSF takes away the need for the developer to worry about managing data state and navigation—these were always areas of Web development with J2EE that became complex and hard to maintain. JSF allows the developer to worry about the requirements of the application rather than having to spend valuable time building the integration points between data and display.



JSF and Web services

This chapter explains the WebSphere Studio tooling for using a Web service within a JSF application. The chapter also guides you through creating a JSF application which can invoke a Web service and display the output.

Web services concepts

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. Web services perform encapsulated business functions, ranging from simple request-reply transactions to full business process interactions. These services can be new applications or just wrapped around existing legacy systems to make them network-enabled. Services can rely on other services to achieve their goals.

A Web service, once published through a UDDI registry or another mechanism, can be consumed by anyone with HTTP access to the Web service's location. One such Internet-based registry of services is located at:

<http://www.xmethods.net>

Web services description language (**WSDL**) is an XML-based interface and implementation description language. A Web service provider uses a WSDL document in order to specify the operations a Web service provides, as well as the parameters and data types of these operations. A WSDL document also contains the service access information.

A WSDL document contains a set of **operations**, which are abstract descriptions of actions supported by the service. Each operation defines the input and output message for its action, as well as an optional fault message.

A **port** is the combination of a network address, a set of operations and their implementation details. Later in this chapter, we select individual ports when discovering Web services.

IBM supports Web services in JSF applications by analyzing a Web service descriptor (WSDL document), then generating all the code necessary to treat the service as a set of managed Java beans.

Documentation

This chapter assumes that the reader is familiar with Web services. If you want to learn more about Web services and the support in IBM products, refer to the IBM Redbook *WebSphere Version 5 Web Services Handbook*, SG24-6891.

Discovering and importing a Web service

The first step in utilizing a Web service in a JSF application is to locate the service to be consumed. WebSphere Studio supports a number of mechanisms for discovering Web services. The preferred mechanism for importing a Web service is using the Page Data view.

You can select the *New* icon, , and select *Web Service*. You may also right click and select *New -> Web Service*. This launches the Add Web Service dialog (Figure 7-1).

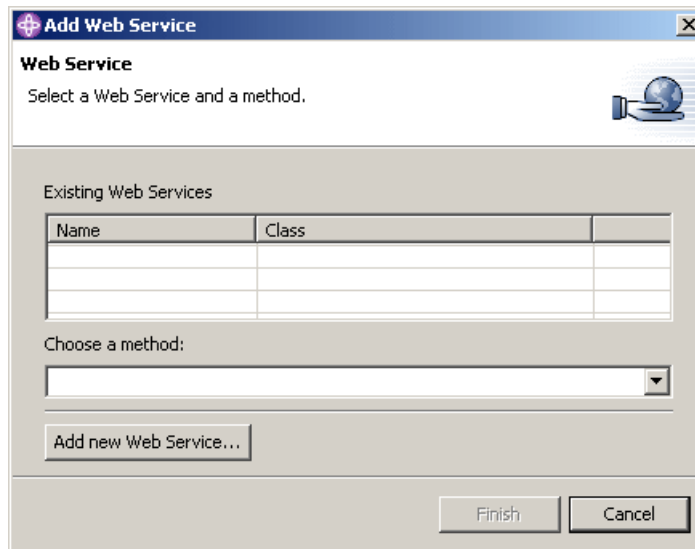


Figure 7-1 Adding a Web Service to a JSF page

From this dialog, click *Add new Web Service*. You are now presented with the Web Service Discovery Dialog (Figure 7-2).



Figure 7-2 Web Service Discovery dialog

The three options allow you to search for Web services in a variety of locations.

► **Web Services from a UDDI Registry**

UDDI registries, as discussed previously, are repositories for many Web services, which are searchable. When you select this link, follow the instructions for searching for a particular web service, and select the corresponding WSDL link to continue.

► **Web Services from a known URL**

Enter an Internet, intranet, or local file system URL to a WSDL file describing the desired Web Service. Once the URL is entered, click *Go* and continue.

► **Web Services from the workspace**

Selecting the final link searches the workspace for any deployed and running Web services. This process does **not** find any Web service definitions you may have previously imported into the workspace, if they are not deployed and active. Select the desired service and continue.

On the final page of the Web Service Discovery Dialog, you must expand the selected service and select the desired port. Finally, click *Add to Project*.

Note: You may optionally click *Details* to launch the Web Services Explorer with which you can test the referenced Web service.

Client code generation

When the Web Service Discovery Dialog adds the selected Web service port to the project, it generates a number of new artifacts.

- ▶ **Java code**

WebSphere Studio generates all the necessary Java code to access the targeted Web service through a regular function call. The code that is generated acts as a proxy for the Web service, accepting a simple call, and invoking the SOAP runtime to call the Web service.

- ▶ **WSDL**

The WSDL file describing the targeted Web service is cached locally in the project in `WEB-INF/wsdl`.

- ▶ **XML and XMI deployment descriptors**

A few XML and XMI files are created which you do not have to use in the scope of this book.

- `WEB-INF/webservicesclient.xml`
- `WEB-INF/ibm-webservicesclient-bnd.xmi`
- `WEB-INF/ibm-webservicesclient-ext.xmi`
- `WEB-INF/serviceName_mapping.xml`

Once the Web Service Discovery Dialog is finished, you are back in the Add Web Service Dialog. You should see the new service in the Existing Web Services table. Select it, then select the desired method, and click *Finish*.

More Java classes are generated into the project. The `ParamBean` represents the input parameters to the Web service method. The `ResultBean` holds the output from the method invocation. Finally, a new method is added to the page code to invoke the Web service proxy. The new beans and new action method are all added to the Page Data view.

Using Web services with JSF pages

The parameter and result bean are not managed beans, in the sense that they are in the page code of the page to which you added the Web service, and they are not referenced in the `faces-config.xml`. This means that the beans are not available to any other pages, by default.

You must invoke the Web service action on the same page where you populated the parameter bean. To display the results, you must bind the result bean to controls on the same page in which you invoked the service method. In essence, the input and output for the Web service must be tied to a single page.

This single-page restriction is easily circumvented. You may recreate the input bean as a session-scope managed bean, so that you can gather the inputs over multiple pages, and invoke the service later. You may store the output data in another context, such as the request block, so that you may display the output from the Web service on a different page. We provide an example of these changes later in the chapter.

Example Web service application

In this part we create a set of JSF pages to invoke a Web service interface of a banking application. The Web service retrieves a customer and the associated accounts for a given customer ID.

Creating the project

We use a dynamic Web project to invoke the Web service and we import an existing Web service into another dynamic Web project:

- ▶ Create a dynamic Web project named **ItsoJSFWebService**. Select *Configure advanced options* and click *Next*.
- ▶ Click *New* and create an EAR project. Enter **ItsoJSF07WebServ** for the project name and click *Finish*. Click *Finish* again to exit the wizard.
- ▶ Create another dynamic Web project named **ItsoBankingWebService**. Select *Configure advanced options* and click *Next*. Set the EAR project to **ItsoJSF07WebServ**. Click *Finish* to exit the wizard.
- ▶ Import an existing Web service. Select the **ItsoBankingWebService** project and *Import*, then select *WAR file*. Click *Browse* to locate the WAR file `\labscode\dev-jsfwebserv\ItsoBankingWebService.war`.

The target project should be set to **ItsoBankingWebService**. Select *Overwrite existing resources without warning* and click *Finish*.

- ▶ Associate the enterprise application (**ItsoJSF07WebServ**) with a server (**JSFServer**).
- ▶ Start or restart the server so that the Web service is running

Note: The **ItsoBankingWebService** project contains the **AccountListingSQL** Web service with the operations **getCustomer** (one Customer), **getAccounts** (array of Account), and **retrieveAccounts** (**AccountListingResult** JavaBean with one Customer and an array of Account).

Finding the Web service

We use two JSF pages, one page to invoke the Web service and one page to display the results:

- ▶ Create two JSF pages named **webserviceInput.jsp** and **webserviceOutput.jsp** in the **ItsoJSFWebService** project. Close the **webserviceOutput.jsp** for now.
- ▶ In the Page Data view select *New -> Web Service*. In the Add Web Service dialog click *Add new Web Service*. The Web Service Discovery Dialog opens.
- ▶ Click *Web Services from your workspace* to locate services that are running. The **AccountListingSQL** Web service is found (Figure 7-3).

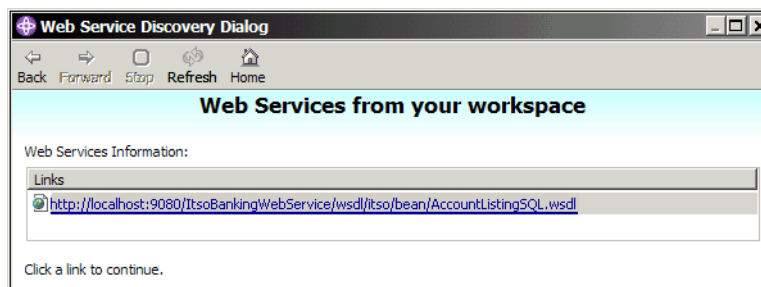


Figure 7-3 Locating the running Web service

- ▶ Click the link and the Web service is displayed (Figure 7-4).

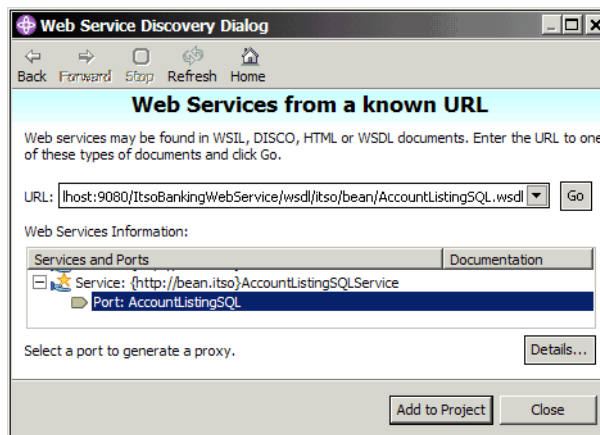


Figure 7-4 Selecting the running Web service

- ▶ Click *Add to Project*.

This generates the client proxy classes in the `itso.bean` package and the WSDL file and deployment information in the `WEB-INF` folder.

The Web service is added to the list of Web services (Figure 7-5).

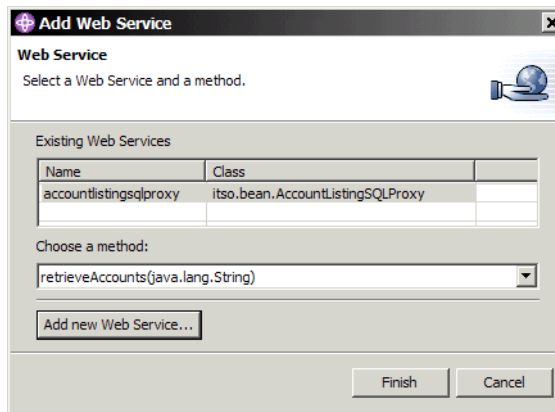


Figure 7-5 Available Web services

- ▶ Select the `retrieveAccounts` method of the Web service. This Web service also has `getCustomer` and `getAccounts` methods.
- ▶ Click *Finish* and two beans are added to the Page Data view (Figure 7-6):
 - `retrieveAccountsParamBean` contains the parameters to invoke the Web service
 - `retrieveAccountsResultBean` contains the results of the Web service

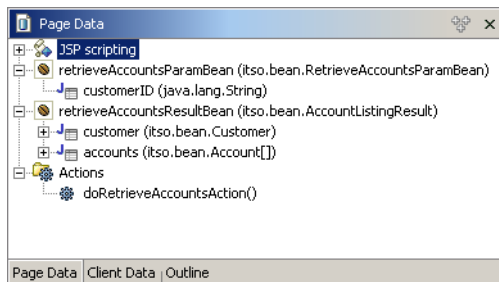


Figure 7-6 Page Data view containing generated beans and actions

- ▶ A `doRetrieveAccountsAction` method to invoke the Web service is added to the page code class. We will use this method in the action logic.

```
public String doRetrieveAccountsAction() {
    if (accountListingSQLProxy == null) {
        accountListingSQLProxy = new AccountListingSQLProxy();
    }
}
```

```

    }
    try {
        retrieveAccountsResultBean =
            accountListingSQLProxy.retrieveAccounts(
                getRetrieveAccountsParamBean().getCustomerID());
    } catch (RemoteException e) {
        logException(e);
    }
    return "";
}

```

- Save the JSP.

Invoking the Web service from JSF

After discovering the Web service discovery and generating the code, we must bind the service to JSPs in the application. We manipulate the scope of the input and output data so we can perform the invocation of the service over multiple pages.

Optional

You may recreate the input bean as a managed bean in the session scope. This enables you to place the controls bound to this bean on any page in the application. This is just an example of how you would do this, though we do not take advantage of this change in this application.

- Edit the **webserviceInput.jsp**.
- Select the `retrieveAccountsParamBean` and *Delete* (context).
- Create a new `JavaBean` in the `Page Data` view as shown in Figure 7-7.

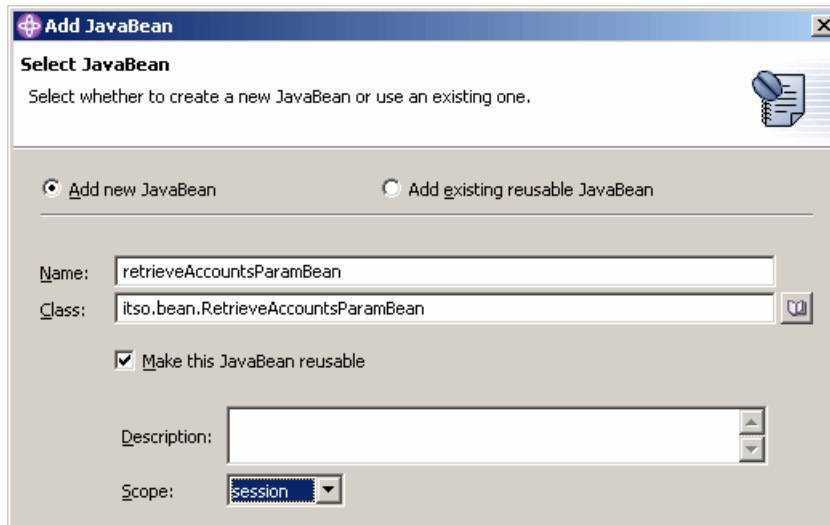


Figure 7-7 Recreating the input bean as a managed bean

Input page

Edit the `webserviceInput.jsp`:

- ▶ Add a heading 2: **Web Service Input**.
- ▶ Drop the `retrieveAccountsParamBean` into the JSP:
 - Select *Updating fields* to get an input field.
 - Change the label to: **Enter a customer ID:**
 - Click *Finish*.
- ▶ View the resulting page (Figure 7-8).



Figure 7-8 Web service input page

- ▶ Select the Submit button and add a navigation rule with alias **success** to the `webserviceOutput.jsp`.
- ▶ Enter the action logic for the Submit button
(\labscode\dev-jsfwebserv\webserviceSubmit.txt):


```
System.out.println("Web Service Start");
```



```
// invoke the generated method to call the Web Service
doRetrieveAccountsAction();
System.out.println("Web Service End: "+retrieveAccountsResultBean);
getRequestScope().put("retrieveAccountsResultBean",
                      retrieveAccountsResultBean);

return "success";
```

This function invokes the generated doRetrieveAccountsAction method for driving the Web service, and stores the result in the request scope, so that we can access the results on the next page.

Output page

Edit the `webserviceOutput.jsp`:

- ▶ In the Page Data view add a request scope variable named **retrieveAccountsResultBean** of type `itso.bean.AccountListingResult`.
- ▶ Add a heading 2: **Web Service Output**.
- ▶ Drop the `retrieveAccountsResultBean` into the JSP:
 - Select *Displaying fields*.
 - Move accounts to the end of the list.
 - Click the ... icon for accounts to see the sub-fields.
 - Change the labels (Firstname, Lastname).
 - Click *Finish*.
- ▶ Set the inner accounts table border to **1** (Attributes view, All tab).
- ▶ Add a Back button with navigation to the input JSP.
- ▶ View the resulting page (Figure 7-9).

Web Service Output

Id: {id} abc

Title: {title} abc

Firstname: {firstName} abc

Lastname: {lastName} abc

Accounts:

Id	Balance	Type
{id} abc	{balance} 123	{type} abc

* {Error Messages}

Back

Figure 7-9 Web service output page

Test the Web service invocation

Test the Web service application by running it on the server:

- ▶ Start the ItsoJSF08 project in the JSFServer. Run the `webserviceInput.jsp`. Supply a customer ID and click *Submit* (Figure 7-10).

Web Service Output		
Id:	102	
Title:	Mr	
FirstName:	Fabio	
LastName:	Ferraz	
Accounts:	Id	Balance
	Type	
	102-2001	9,370
	102-2002	100
	SAVINGS	CHECKING

Figure 7-10 Running the JSF Web service application

Complete solution

We provide the complete solution for this exercise in the additional material in the file `\labscode\solution\ItsoJSF07WebServ.ear`. See “Importing a solution enterprise application” on page 548 for instructions.

Best practices

There are a number of advantages for structuring an application into a Web service tier and a JSF client tier. A multi-tier application forces further separation, and hence purity, of the application developer’s and page author’s roles. The workload can be distributed, because Web services often reside in a location separate from the JSF client application.

Finally, and perhaps most importantly, the application is well-positioned to fit into a service-oriented architecture (SOA). For example, when you move into a process choreography scenario using business process execution language (BPEL), no new development is required. A possible downside to breaking out these two tiers is the introduction of another layer of network communication. This could result in individually slower transactions, but the advantage gained through distributed processing may offset this.



JSF and Struts

This chapter discusses the relationship of the Struts framework to the JavaServer Faces framework. It compares and contrasts the two frameworks, then discusses how you might use them in conjunction.

This chapter is not intended to be a thorough treatment of the Struts architecture or Struts development.

Introduction to Struts

Struts is an open source framework for building Web applications. It is part of the Jakarta project, sponsored by the Apache Software Foundation. Struts, like JSF, is an attempt to apply the model-view-controller (MVC) architectural pattern to Web application development.

The Struts framework includes several pieces which make up the controller component of its applications, such as a central servlet (the `ActionServlet`) and developer-defined request handlers. Struts comes with a JSP UI tag library which supports the view component of its applications. The view of a Struts application may also make use of JSTL, XSLT, and Tiles, among other technologies. The Struts application model is made up of Java beans, created by an application developer, which can make use of a variety of technologies, such as JDBC or EJBs.

Struts introduces the concept of *actions*, which are used as request handlers when a special URI is used for form submission. These actions are responsible for processing form data (beans), possibly creating new form beans, and redirecting to a new JSP page. These actions act as an developer-defined intermediary between the Struts `ActionServlet` and the application model.

The Struts architecture can be seen in Figure 8-1.

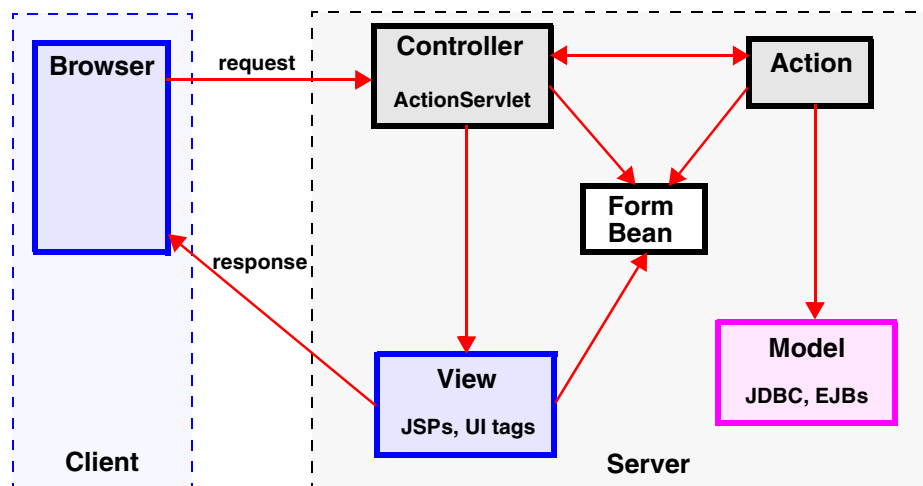


Figure 8-1 Struts architecture

Comparing JSF and Struts

Struts and JSF contain many of the same features, and both have a very similar overall structure. However, depending on the requirements of the application, it is useful to examine the various discrepancies between the two frameworks (Table 8-1).

Table 8-1 *JavaServer Faces and Struts feature comparison*

	JavaServer Faces	Struts
Components	▶ Rich data bound UI components with events provided ▶ Custom components	▶ Struts-specific tag library ▶ Only very basic, form bean bound components provided
Device independence	▶ Reader kits that provide device independence	▶ None
Error handling and validation	▶ Validation framework ▶ Many predefined validators	▶ Validation driven by an XML descriptor (<code>validation.xml</code>)
Scripting	▶ Scripts can be attached to events ▶ All components accessible from scripts	▶ Scripts written in Java action classes ▶ Form data, but not components, accessible
Page flow	▶ Simple navigation file (<code>faces-config.xml</code>) ▶ XML-based	▶ Sophisticated, flexible framework ▶ XML-based
Session and object management	▶ Automatic	▶ Manual

Apart from the features of the frameworks, it is important to consider the strength of their relative tools, maturity, and future directions (Table 8-2).

Table 8-2 *JavaServer Faces and Struts considerations*

	JavaServer Faces	Struts
Tooling	IBM and Sun IDE support are relatively new, many tools soon to follow	Extensive, mature IDE support
Maturity	Relatively young, though at least one version of the specification has been ratified	Quite mature, stable, not subject to significant changes

	JavaServer Faces	Struts
Future plans	Lots of development vitality, future implementations planned, more extensive component libraries to be developed, and new contexts to be supported	Relatively static

Choosing Struts and/or JSF

JSF originated as a follow-on to the Struts framework, largely driven by the original creator of Struts, Craig McClanahan. However, JSF is not intended to be a complete replacement for Struts. In fact, when deciding which framework to use when developing your applications, the choice between JSF and Struts is not mutually exclusive. They are actually intended to be used effectively in conjunction in a variety of ways.

A Struts-Faces integration library is currently being developed by the Apache Jakarta project, which enables cooperation between JSF and Struts, with the intention of utilizing the strongest points of both frameworks. The integration library is based on JavaServer Faces 1.0 and Struts 1.1, both of which are available in Application Developer. There are no fixed dates for Struts-Faces Version 1.0 availability, but as of the time of writing, the daily builds available from the URL below are roughly Version 0.5. Find more information at:

<http://cvs.apache.org/builds/jakarta-struts/nightly/struts-faces/>

Given the current features, maturity, and future plans of JSF and Struts, we provide the some recommendations for future development as dictated by the nature of the projects (Table 8-3).

Table 8-3 Struts/Faces project development recommendations

Project Type	Recommendation
Developing a new project from scratch	Use pure Faces, or possibly Struts-Faces if you have existing Struts expertise or reusable components
Existing Struts application that require minor development	Continue using pure Struts
Existing Struts application that require significant UI work	Start using Struts-Faces

Combining JSF and Struts

The Struts-Faces integration library is targeted to allowing developers to use Faces UI components in the frontend of their applications, while still allowing the familiar controller function and flexibility of a Struts application. This gives the developer the ease of the Faces event listener programming model, the richness of the Faces UI component libraries, the flexibility and control of Struts navigation control, and the power of the Struts validation framework.

The Struts-Faces library consists of the following components:

- `FacesRequestProcessor`

This object extends the base Struts `RequestProcessor`, which is normally responsible for interpreting form action URLs, and mapping to actions. Faces requests are intercepted by this controller and processed accordingly. Non-Faces requests are handled by the parent class.

- `ActionListenerImpl`

This object extends the base Faces `ActionListener`, which handles `ActionEvents` generated by Faces components. The new implementation delegates the action handling back to the `FacesRequestProcessor` so a component action can be treated like a form action.

- A new form component

A new Faces UI component, renderer, and tag are provided for the form element, to take the place of the existing form element. This new component is integrated with the Struts life cycle.

- New output components

These are replacements for Faces components used solely for output, along with new renderers and tags.

- Struts-Faces tag library

A new tag library defining tags for the new UI components is available at:

<http://jakarta.apache.org/struts/tags-faces>

This file is for reference only; a copy is included in the `struts-faces.jar`.

- `faces-config.xml`

This configuration file registers the `ActionListenerImpl` and Struts-Faces UI components with the Faces runtime. This file is for reference only; a copy is included in the `struts-faces.jar`.

Struts-Faces request

All Struts-Faces page submissions go through the `FacesRequestProcessor`. If the request is determined to be a Faces request, the process must go through both the Faces controller, for processing the Faces life cycle, and then the Struts controller, for driving the page navigation (Figure 8-2).

Non-Faces requests go directly to the Struts controller. The resulting page forward may go through the `FacesServlet` if the `.faces` extension is used on the targeted page name. If not, a normal page include is used.

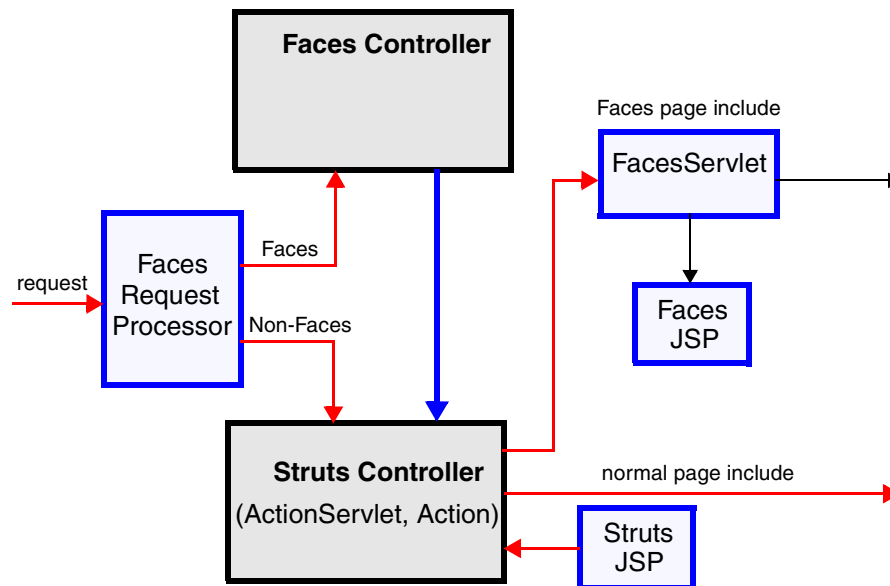


Figure 8-2 Struts-Faces hybrid request processing

Adding Faces support to a Struts application

To add Faces support to a Struts application, follow these steps:

- ▶ Obtain the most current version of the Struts-Faces integration library.
- ▶ Add the `struts-faces.jar` from the integration library into the `WEB-INF/lib` directory of the Struts project.
- ▶ Add the JSF JAR files, `jsf-api.jar` and `ri.jar`, into the `WEB-INF/lib` folder. These files can be found in the `com.ibm.etools.jsf.runtime.api_5.1.2` and `com.ibm.etools.jsf.ri_5.1.2` plug-in runtime folders.
- ▶ Edit the project's Web features and select the *JSP Standard Tag Library* option. This makes the JSTL tags accessible in the project.

- ▶ Add an entry for the FacesServlet to the Web deployment descriptor. Make sure it is loaded on startup before the Struts controller servlet.
- ▶ Add a servlet mapping for the JSF servlet to the Web deployment descriptor. Use the URL pattern `*.faces`.
- ▶ Set the controller in the `struts-config.xml` to the class `org.apache.struts.faces.application.FacesRequestProcessor`.

Follow the README and documentation accompanying the Struts-Faces integration library for complete details.

Once you have completed adding Faces support, you can treat the Web application as a Struts-Faces hybrid.

Creating a new Struts-Faces page

The following steps can be used to create a new Struts page which makes use of Faces UI components:

- ▶ Create a new JSP file.
- ▶ Use the following tag library include statements:


```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@ taglib prefix="s" uri="http://jakarta.apache.org/struts/tags-faces" %>
```
- ▶ Create a Struts form bean and action to process this page.
- ▶ Use the normal Faces UI component tags to generate the view.
 - Use JSF value binding to associate the components with form bean or managed bean properties.
- ▶ For the HTML form, use the `<s:form>` tag instead of `<h:form>`.

This tag hooks the form submission into the Struts framework for processing by the `FacesRequestProcessor` controller which has been registered. Configure this tag to point to the action you defined to process this page. The action name does not have the `.do` extension, because the request is processed by the `FacesRequestProcessor`.
- ▶ If you require a Struts-style Cancel button, use a `<h:commandButton>` that has its `id` attribute set to **cancel**.

A Faces button with this `id` is recognized by Struts as a Cancel button.

- ▶ Modify any Struts actions which forward to this page to go through the `FacesServlet`.

Modify the corresponding forward statements in the `struts-config.xml`. The changes are dependent on how you established the servlet mapping.

- Struts forward statement ==> `<forward name="update" path="/Update.jsp"/>`
- Faces forward statement ==> `<forward name="update" path="/Update.faces"/>`

Because navigation is handled by Struts, action attribute method binding on UI command components do not drive Faces navigation rules. Any bound methods may, however, manually drive forwards or redirects. You are also free to bind server side action listeners, which do not affect navigation, to command components.

Converting a Struts page into a Struts-Faces page

You must follow the guidelines above to convert an existing page to Struts-Faces. You should not have to modify the form beans or actions at all.

- ▶ Update the tag library includes as described previously.
- ▶ Convert Struts UI tags to the equivalent JSF UI component tags, converting the binding syntax to JSF value binding syntax for any components that use the form bean.
- ▶ Replace any Cancel buttons as described above.
- ▶ Convert the form tag to the Struts-Faces form tag, preserving the attributes. Remove the `.do` extension from the action name in the page and in the `struts-config.xml`.
- ▶ Change any forwards to this page as described previously.

Summary

Integration of Faces and Struts is being developed and will appear in future versions of the IBM tools.



Part 3

Service Data Objects

In Part 3 we provide a description of Service Data Objects and how to use them in a JSF Web applications.

In Chapter 9 we introduce the Service Data Objects framework and tools.

In Chapter 10 we provide a JSF Web application example that uses Service Data Objects.



SDO concepts

This chapter describes Service Data Objects (SDO) and how to use them in JSF Web applications using WebSphere Studio Application Developer as the development tool.

This chapter discusses the following topics:

- ▶ Introduction to SDO
- ▶ WebSphere Studio Application Developer support for SDO
- ▶ SDO components and their uses
- ▶ SDO API

Introduction to SDO

Service Data Objects (SDO) is a data programming architecture and API for the Java platform that unifies data programming across data source types, provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data.

The Java specification request is JSR-235 and can be found here:

<http://www.jcp.org/en/jsr/detail?id=235>

SDO are designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and enterprise information systems. The SDO architecture consists of three major components: the **data object**, the **data graph**, and the **data mediator** (Figure 9-1).

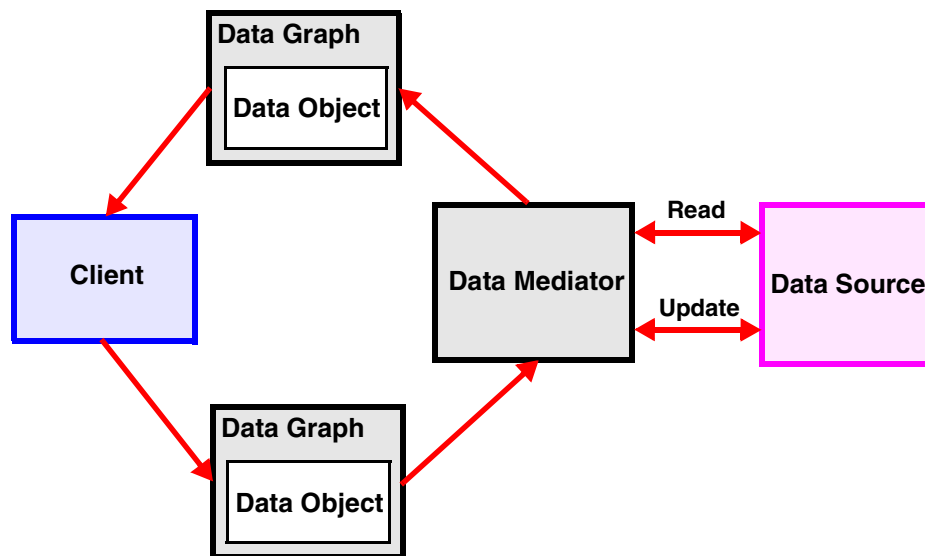


Figure 9-1 SDO architecture

Data object

The **data object** is designed to be an easy way for a Java programmer to access, traverse, and update structured data. Data objects have a rich variety of strongly and loosely-typed interfaces for querying and updating properties. This enables a simple programming model without sacrificing the dynamic model required by tools and frameworks. A data object may also be a composite of other data objects.

Data graph

SDO is based on the concept of disconnected **data graphs**. A data graph is a collection of tree-structured or graph-structured data objects. Under the disconnected data graphs architecture, a client retrieves a data graph from a data source, mutates the data graph, and can then apply the data graph changes to the data source. The data graph also contains some metadata about the data object including change summary and metadata information. The metadata API allows applications, tools, and frameworks to introspect the data model for a data graph, enabling applications to handle data from heterogeneous data sources in a uniform way.

Data mediator

The task of connecting applications to data sources is performed by a **data mediator**. Client applications query a data mediator and get a data graph in response. Client applications send an updated data graph to a data mediator to have the updates applied to the original data source. This architecture allows applications to deal principally with data graphs and data objects, providing a layer of abstraction between the business data and the data source.

WebSphere Studio support for SDO

WebSphere Studio does not support the complete specification of SDO. The implementation provided by WebSphere Studio is based on WebSphere Data Objects (WDO) and only supplies a data mediator for JDBC database access.

WDO represents a precursor to the more broadly accepted SDO architecture. These architectures, however, are largely the same, so we may use the terms WDO and SDO interchangeable in this book.

Relational record

WebSphere Studio Application Developer 5.1.2 provides a special structure, called a **relational record**, to access the content of a single record in a relational database. This structure allows you to display, create, and update single records from JSF pages. A relational record can be a join over multiple tables.

When you create a relational record, Application Developer guides you through creating a database query. The tools then automatically generates a JDBC mediator, which performs that query. The structure of the data graph supplied by that mediator is also stored, so that you can treat the data object as a strongly typed managed bean. You can bind UI components in JSF pages to the properties of this bean, thereby using the data object in a Web application.

Relational record list

Application Developer 5.1.2 allows you to define an structure to access the content of several records of a relational database. This structure is called **relational record list**. A relational record list can be a join over multiple tables.

When you create a relational list, you define a query that returns multiple database records. The tools then act in much the same way as defining a relational record. The query and record structure are stored for you so that you can treat the results as a managed bean. You can bind the properties of this bean to the UI components of the JSF pages in a Web application.

Note that you can easily convert a relational record list into a relational record by using a query that only returns one record of the table.

Developing an SDO object

If you are planning to use SDO for accessing database data, you must use WebSphere Application Server 5.1 as the target server for dynamic Web projects.

In this section we explain how to develop an SDO object in a JSP page (JSF or regular JSP page) using WebSphere Studio Application Developer 5.1.2.

SDO wizard

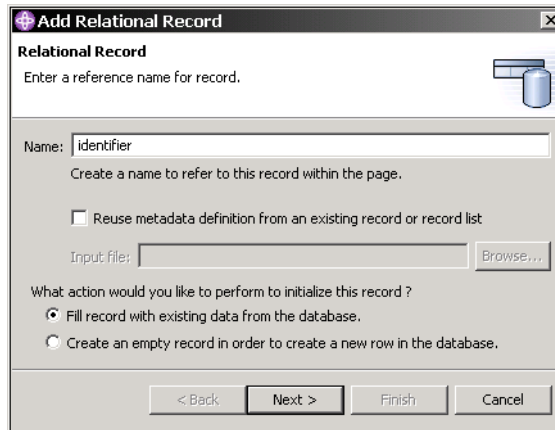
You can start the SDO wizard by selecting *New -> Relational Record List* or *New -> Relational Record* in the context menu in the Page Data view.

The same wizard is used for the two objects and is composed of several pages. We now explain each page.

Relational record and relational record list

In the first page of the wizard (Figure 9-2) you can set the following properties:

- ▶ The name of the object. This name is used in the page to access the data.
- ▶ You can reuse an existing definition of another relational data record or relational record list for the new object. Metadata definitions are XML files that represent the relational record or list.
- ▶ If you are defining a relational record, you can decide if the data object is initialized for retrieve or insert. This option is not available for the relational record list.



Add Relational Record

Relational Record
Enter a reference name for record.

Name:

Create a name to refer to this record within the page.

☐ Reuse metadata definition from an existing record or record list

Input file:

What action would you like to perform to initialize this record ?

☒ Fill record with existing data from the database.

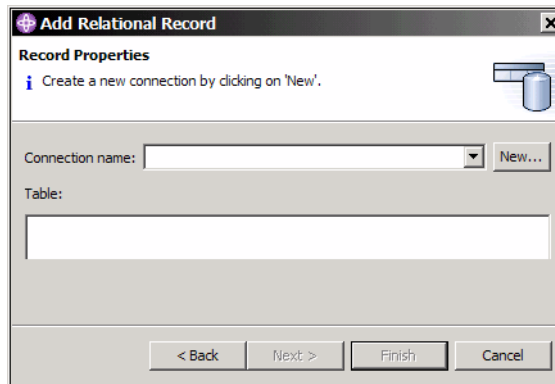
☐ Create an empty record in order to create a new row in the database.

< Back **Next >** Finish Cancel

Figure 9-2 Defining a new relational record

Record properties

In this page you must select the connection to the database and afterwards the table in the database (Figure 9-3).



Add Relational Record

Record Properties
i Create a new connection by clicking on 'New'.

Connection name:

Table:

< Back **Next >** Finish Cancel

Figure 9-3 SDO record properties

Initially no SDO connections are defined, so click *New*. The New Connection wizard allows you to define new connections or reuse existing connections that have been created in the Data perspective (Figure 9-4).

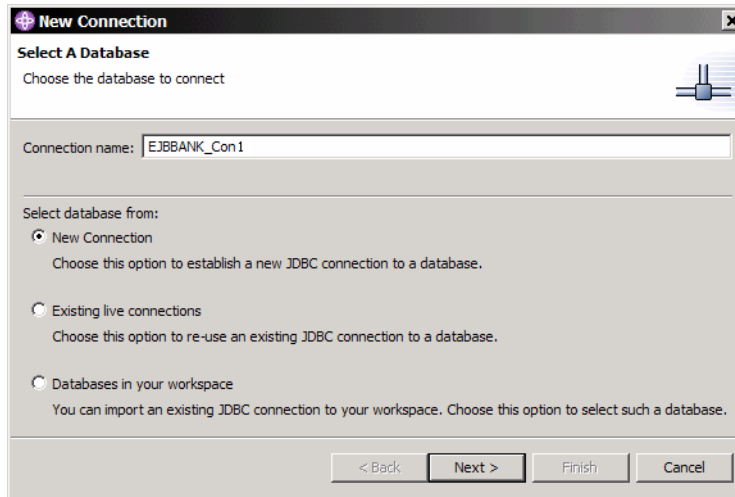


Figure 9-4 Select a database dialog

The three options are:

- ▶ *New Connection*—to define a connection. Note that the name becomes the data source JNDI name.
- ▶ *Existing live connection*—to select a connection that is defined and live (connected) in the DB Servers view,
- ▶ *Database in your workspace*—to work with a database that was imported from the DB Servers view.

Tip: Even if you have created database connections previously in the project they do not appear in the list of SDO connections. You have to click *New* to select an existing connection.

New connection

If you select *New Connection* you enter the database information as shown in Figure 9-5.

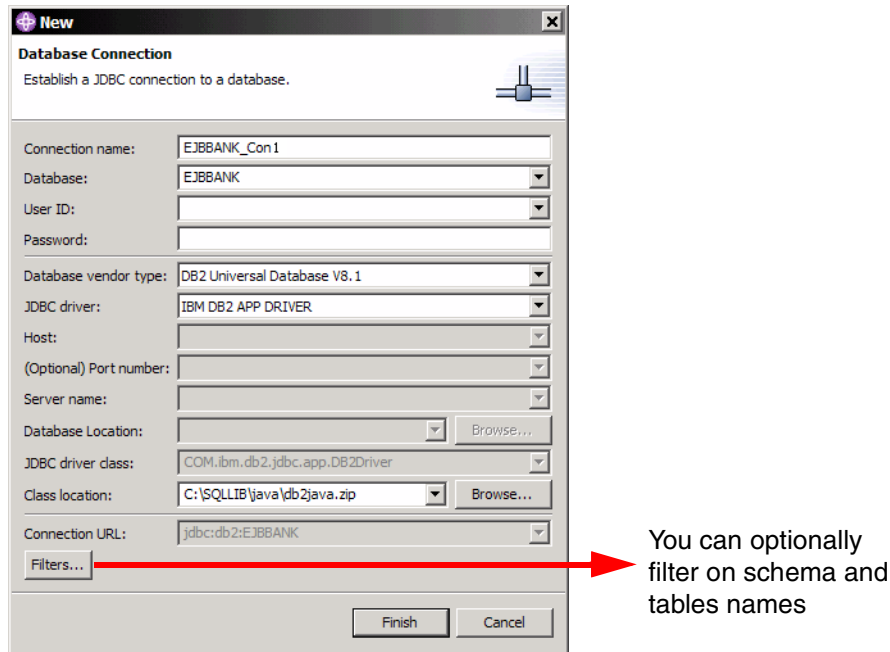


Figure 9-5 Database connection dialog

You can select the database vendor and JDBC driver from here and supply connection details, such as the user ID and password for the database.

Tip: Before you create a database connection always make sure the local development database product is running or that you have a valid live database connection open to a remote development time database. If you do not have a live connection you will have to close the dialog and start again.

Once the connection has been created, a live connection is maintained inside Studio and the metadata for the database is loaded. This increases the performance of the tooling because it prevents it from having to retrieve table metadata on each request.

More details about database connections and the resulting data sources are provided in “Database connections and data sources” on page 311.

The Record Properties panel is now filled with the tables retrieved from the database (Figure 9-6).

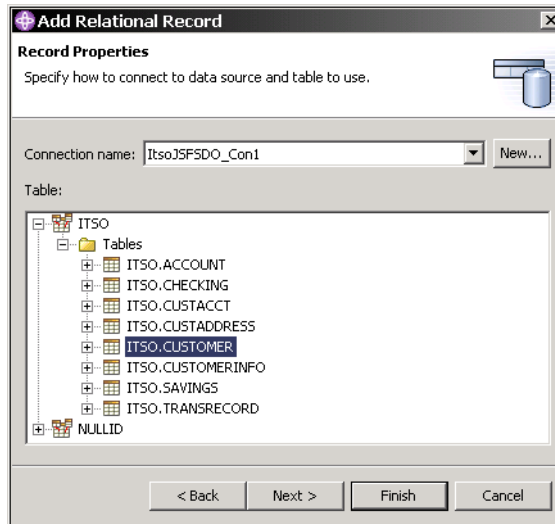


Figure 9-6 Connection and table selection page

Column selection and other tasks

In this page you select the fields of the table you want to use (Figure 9-7).

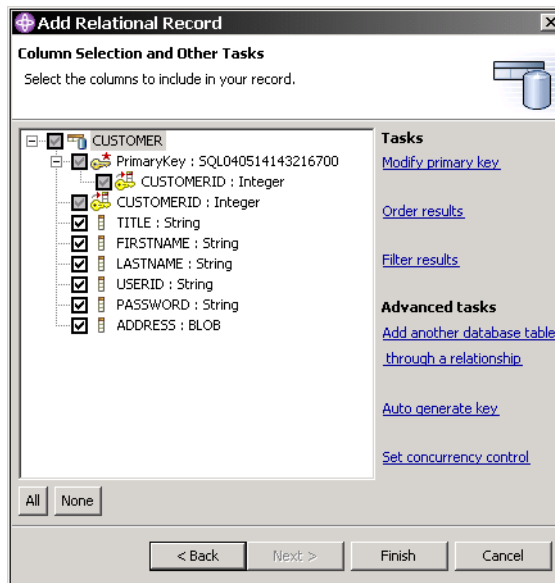


Figure 9-7 Selecting the columns and performing additional tasks

You can do the following basic tasks in this page:

- ▶ **Modify primary key**—Edit the primary key of the table. You can add or remove columns to the definition of the primary key.
- ▶ **Order results**—Set or modify the columns that the query uses for ordering the results.
- ▶ **Filter results**—Add or remove conditions to the query. You can define complex conditions and you can use the page data objects (data from application, session or request scope variables and data defined as parameters) to define the conditions.

In addition to the basic tasks, you can do several advanced task:

- ▶ **Relationship** (*Add another database table through a relationship*)—You can add more data to the set by performing a join. If you select this, all existing relationships are shown and you can choose one of them. You can also define additional table relationship that are not predefined through foreign keys.
- ▶ **Auto generate key**—This options allows you to use a utility table to generate the next value of the key automatically when you insert data in the table.
- ▶ **Set concurrency control**—If you want to have collision control for the table, you have to select a non-null integer column of the table. This column value is incremented by one on each update. It can be used to detect collisions when the table row has been updated by other users.

These advance tasks should be done by database administrators working with the application architect. The database design must match the application requirements and the requirements may lead into database restrictions.

Generated and updated files

Generated and updated files include the SDO metadata files, connection information, Web deployment descriptor, page code, and data source definitions.

SDO metadata files

When you define a relational data object, its metadata is stored in an XML file in a folder called **wdo**. This folder is placed under `WebContent\WEB-INF`.

These XML files store all the necessary information about the table (or tables with relationships), the selected columns, the filter conditions, and the ordering.

A sample metadata XML file with a join is shown in Figure 9-8.

```

<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.websphere.wdo mediator.rdb.metadata:Metadata xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.websphere.wdo.mediator
  .rdb.metadata="http://com.ibm/websphere/wdo/mediator/rdb/metadata.ecore"
  rootTable="//@tables.0">
  <tables name="CUSTACCT" schemaName="ITS0">
    <primaryKey columns="//@tables.0/@columns.0 //@tables.0/@columns.1"/>
    <foreignKeys columns="//@tables.0/@columns.1"/>
    <columns name="CUSTOMERID"/>
    <columns name="ACCID" type="4"/>
    <filter predicate="( CUSTOMERID = ?)">
      <filterArguments name="sessionScopecustomerID" type="4"/>
    </filter>
  </tables>
  <tables name="ACCOUNT" schemaName="ITS0">
    <primaryKey columns="//@tables.1/@columns.0"/>
    <columns name="ACCID" type="4"/>
    <columns name="ACCTYPE" type="4"/>
    <columns name="BALANCE" type="9"/>
  </tables>
  <relationships exclusive="false" childKey="//@tables.0/@foreignKeys.0"
    parentKey="//@tables.1/@primaryKey"/>
</com.ibm.websphere.wdo.mediator.rdb.metadata:Metadata>

```

Figure 9-8 Sample SDO metadata file

Connection information

The connection information is stored in a control file named `.wdo-connections` in the project root folder. A sample file is shown here:

```

<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.websphere.wdo.connections:connections ... >
  <connection id="EJBBANK_Con1">
    <development
      xsi:type="com.ibm.websphere.wdo.connections:rsc-live-connection"
      id="EJBBANK_Con1_dev" name="EJBBANK_Con1"/>
    <runtime
      xsi:type="com.ibm.websphere.wdo.connections:datasource-connection"
      auto-deploy="true" class-location="C:\SQLLIB\java\db2java.zip"
      id="EJBBANK_Con1_runtime"
      classname="COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource"
      database-name="EJBBANK" jndi-name="jdbc/EJBBANK_Con1"
      password="{xor}" resource-reference-name="EJBBANK_Con1"
      server-name="" sql-vendor-type="23" userid=""/>
    </connection>
  </com.ibm.websphere.wdo.connections:connections>

```

Note that a development and a runtime connection are defined. The runtime connection refers to a data source that must be defined in the server.

Note: Use the Server or Resource perspective to see the project control files.

Web deployment descriptor

The Web deployment descriptor is updated with a database resource reference. This reference can be found in the References page when you open the Web deployment descriptor (Figure 9-9).

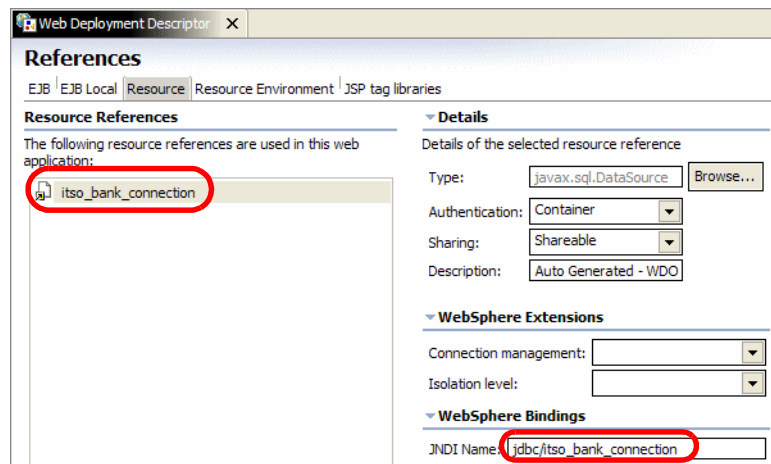


Figure 9-9 Resource reference created for the connection

Page code

The page code Java class contains variables and methods for the data object:

- Data source name:

```
private static final String customer_connectionName = "EJBBANK_Con1";
```

- Filter arguments and values (default filter on key):

```
protected static final String[] customerArgNames = {"paramCUSTOMERID" };  
protected static final String[] customerArgValues =  
    { "#{param.CUSTOMERID}" };
```

- Method to retrieve the data object:

```
public DataObjectAccessBean getCustomer() {  
    if (customer == null) {  
        try {  
            resolveParams(  
                getCustomerMediator().getParams(),  
                customerArgNames,  
                customerArgValues,  
                customer_connectionName,  
                customer_connectionName);  
        } catch (Exception e) {  
            // ...  
        }  
    }  
    return customer;  
}
```

```

        customerArgValues,
        "customer_params_cache");
    DataGraphAccessBean graph = getCustomerMediator().fetchGraph();
    customer = graph.getDataObjectAccessBean();
} catch (Throwable e) {
    logException(e);
}
}
}
return customer;
}

```

- Method to retrieve the JDBC mediator:

```

protected JDBCMediatorAccessBean getCustomerMediator() {
    if (customerMediator == null) {
        try {
            customerMediator =
                new JDBCMediatorAccessBeanImpl(
                    getRealPath(customer_metadataFileName),
                    customer_connectionName);
        } catch (Throwable e) {
            logException(e);
        }
    }
    return customerMediator;
}

```

For more information about the SDO API see “SDO API” on page 329.

Data source

When you run the application using the WebSphere Application Server 5.1 Test Environment, the definition of this server is updated with an authentication alias and a data source:

- A new JAAS entry is defined in the Security page (Figure 9-10).

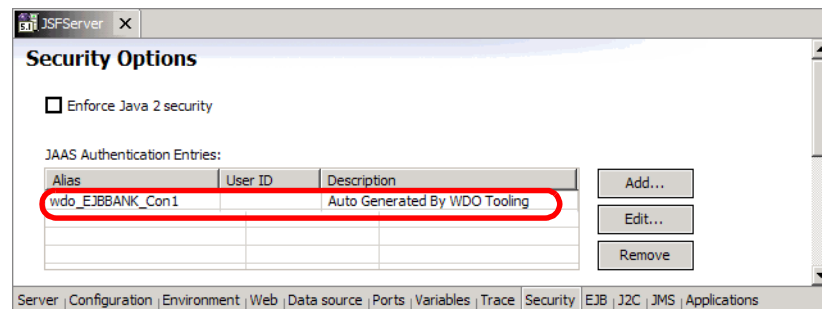


Figure 9-10 Security entry created in the server

- ▶ A new data source is created in the Data Source page (Figure 9-11). This data source has the same JNDI name as the resource reference defined in the Web deployment descriptor.

This makes testing of SDO-based JSF applications very easy because no manual server configuration is required.

Note: You have to define the data sources manually when deploying SDO applications to a real WebSphere server.

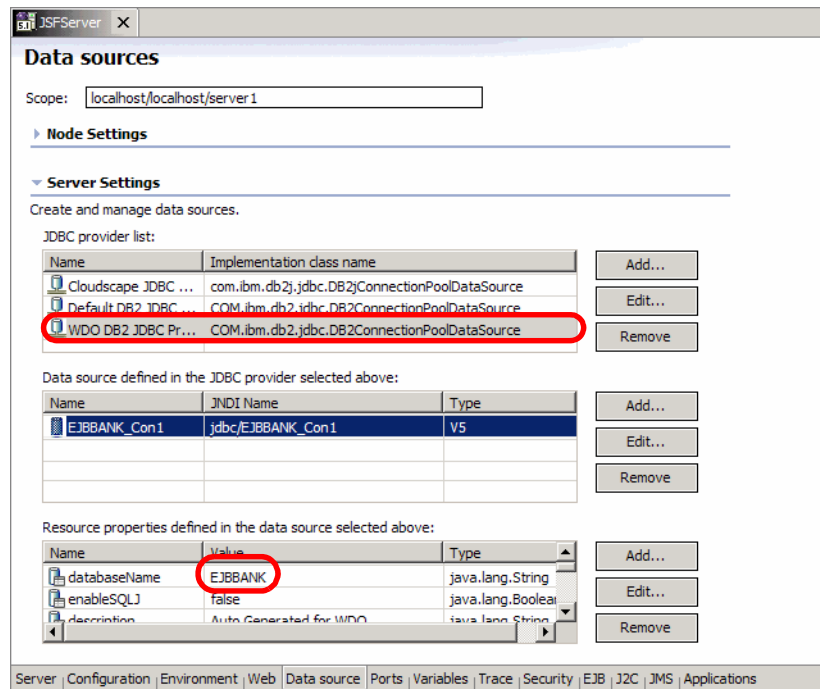


Figure 9-11 Data source definition

Database connections and data sources

There are two main concepts for data connections within WebSphere Studio Application Developer:

Development time These connections are used at development time and allow the Studio tool to have a live connection to a database to retrieve metadata. This metadata is then used in the tooling wizards to create SDO objects.

Run time

These connections are maintained in the WebSphere runtime configuration. Studio creates data source references. When the application is deployed the data sources must be configured in the production Application Server.

In the following section we discuss these data connections and explain how they are defined and used within both WebSphere Studio and WebSphere Application Server.

Development time connections

WebSphere Studio has built in support for JDBC compliant databases. The tool has a Data perspective that allows easy integration with databases and allows DDL and SQL queries to be developed and executed. This perspective also contains a view that allows the definition of database connections. The view is called *DB Servers*. This view can be easily added to the Web perspective and allows the developer the ability to manage and maintain the database connections that are used during the development of the JSF application:

- ▶ To add the DB Servers view to the Web perspective select the *Windows -> Show View -> Other*. When prompted select the *Data -> DB Servers* view.
- ▶ You can use the IDE to drag and drop the view to a suitable development area of the Web perspective (Figure 9-12).

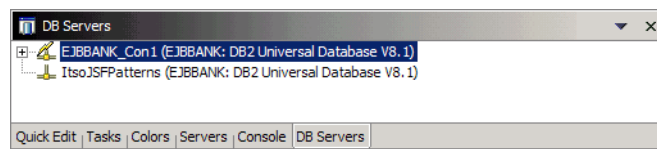


Figure 9-12 DB Services view

This view allows connection to be created, deleted, or reconnected. The state of a connection can depend on what activity you have performed within the workbench.

Creating a connection

To create a connection select *New Connection* from the context menu. In the connection dialog enter the required information (see Figure 9-5 on page 305).

After creating a connection you can navigate through the metadata (Figure 9-13) or perform these operations:

- ▶ **Import to Folder**—Import the metadata from a connection to a folder within a project. This can be used for offline development.

- ▶ **Refresh**—Refresh the metadata if new tables or database changes have been made to the database.
- ▶ **Disconnect**—Disconnect from the connection.
- ▶ **Reconnect**—Reconnect to a database and refresh the metadata. This is necessary when you restart the workbench.
- ▶ **Delete**—Delete the connection from the DB Servers view.

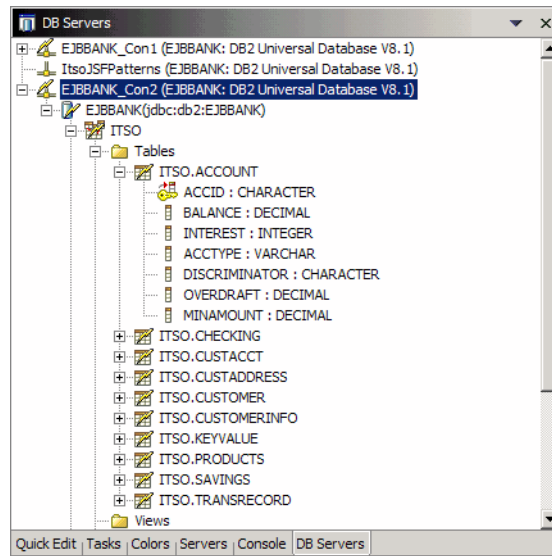


Figure 9-13 Database connection metadata

Development time connections allow the developer easy access to information that is required by the development tooling for the navigation through SDO wizards. The other key point is that when a connection is created it also has a corresponding data source created in the test environment.

Runtime connections

Underlying the SDO framework is its ability to connect to JDBC data sources. These data sources manage the connection pooling and interface with the JDBC database. If a JSF page is directly using an SDO object, the connection name stored in the page code. Here is an example of the variable being automatically generated by Studio when a SDO object is placed within a page:

```
/**
 * @generated by WebSphere Studio
 */
private static final String customerName_connectionName = "EJBBANK_Con1";
```

This name directly relates to the data source name that must be defined within a WebSphere Application Server. If you are planning to deploy this Web application within a stand alone WebSphere configuration you have to manually define the data source with this name.

Within the Studio test environment the creation of these data source is performed automatically when you run the JSF Web application on the test server.

Tip: If you change the value of the connection variable in the page code you have to manually edit the data source configuration name to match the name you have entered.

SDO components

The first release of SDO for WebSphere Studio Application Developer 5.1.2 contains a set of Studio wizards that integrate the SDO mediators for JDBC into the development environment. The tooling in studio includes two components a *Relational Record List* and a *Relational Record*. These two components can be used in a JSF page or JSP page.

When you create a JSF page you can add these components to the Page Data view of the page. This allows the developer the ability to manage these objects in a common area of the development environment.

In this section we describe how to use these components in JSF pages. As an example we use the `ITS0.CUSTOMER` table in the `EJBBANK` database.

Data retrieval

First we describe how to display customer information in table format. Using the SDO wizard we create a relational record list called `customers` (Figure 9-14):

- ▶ Select *New -> Relational Record List* in the Page Data view.
- ▶ Enter `customers` as the name.
- ▶ Defines or reuse a connection to the `EJBBANK` database.
- ▶ Select the `ITS0.CUSTOMER` table.
- ▶ Select the desired columns.
- ▶ Click *Finish*.

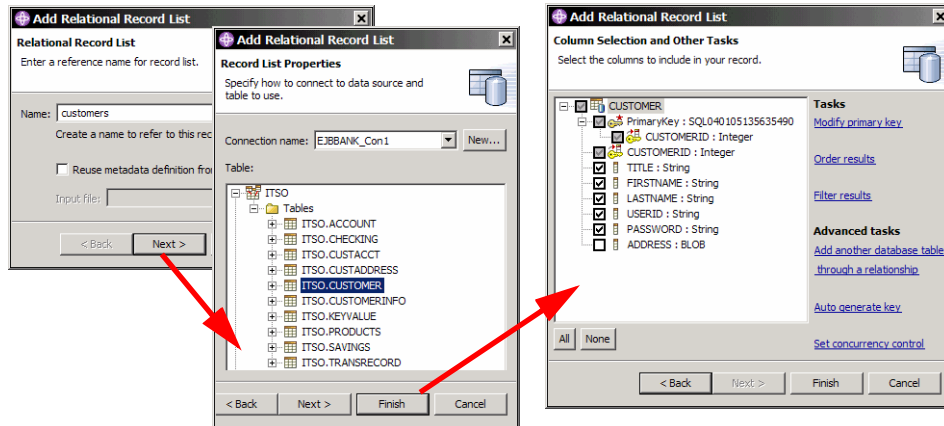


Figure 9-14 Steps for creating a relational record list

Once you have created the Relational Record List it is added the Page Data view of the Web perspective (Figure 9-15).

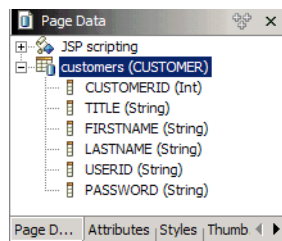


Figure 9-15 Page Data view with relational record list

Displaying a relational record list

To include this component in the JSF page you can either drag the whole list or individual columns (by selecting multiple columns under the table) into a JSF page.

A dialog opens to allow you to customize the creation of the list in a Data Table component (Figure 9-16).

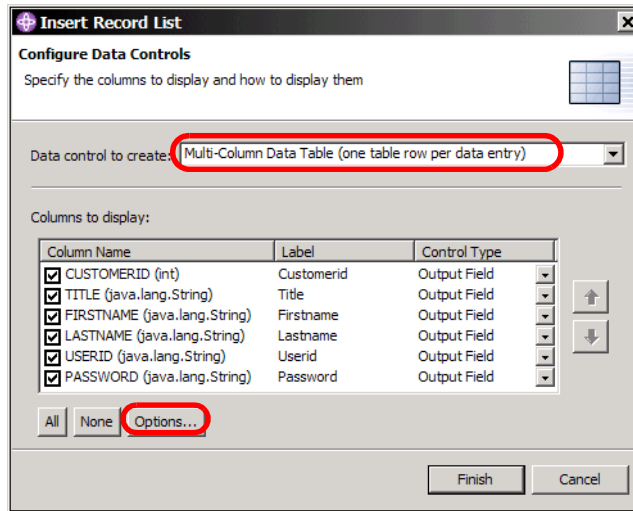


Figure 9-16 Insert Record dialog that is display after a drop on a JSF page

Using this dialog you can change the selection of the columns that are added to the JSP. You can also change the order using the arrow keys on the right hand side.

Click *Options* to configure to configure how the labels in the header are displayed, if the titles are capitalized and have colons appended. You can also modify the row depth using the Advanced tab (Figure 9-17).

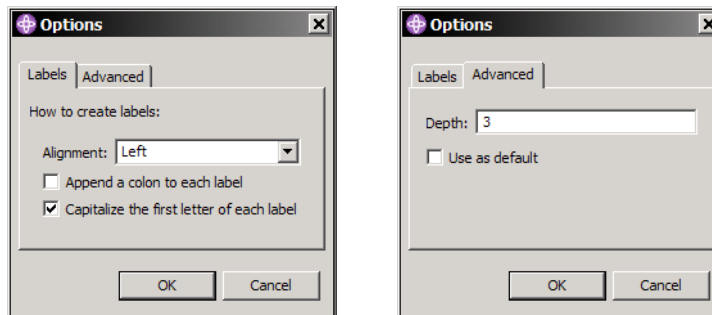


Figure 9-17 Configuring the options

For now we display all fields as output fields. We describe input fields or combo boxes when you explore updating of data.

Multi-column or single-column

The other important setting on the Insert Record List dialog is deciding which data control is created (Figure 9-18).

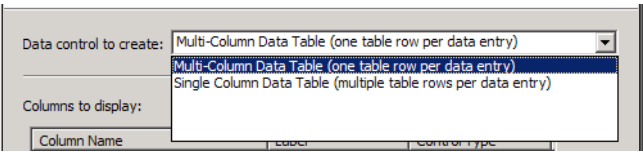


Figure 9-18 Select the type of Data list control to be created on the page.

The default of *Multi-Column Data Table* creates one row for each record in the relational record list (Figure 9-19).

Customerid ^{abc}	Title ^{abc}	Firstname ^{abc}	Lastname ^{abc}	Userid ^{abc}	Password ^{abc}
{CUSTOMERID} ^{abc}	{TITLE} ^{abc}	{FIRSTNAME} ^{abc}	{LASTNAME} ^{abc}	{USERID} ^{abc}	{PASSWORD} ^{abc}

Figure 9-19 Data list as multi-column data table

If you change the data list control to a Single Column Data Table you get a layout where all the table elements are shown in one cell (Figure 9-20).

Customers ^{abc}	
Customerid:	{CUSTOMERID} ^{abc}
Title	{TITLE} ^{abc}
Firstname	{FIRSTNAME} ^{abc}
Lastname	{LASTNAME} ^{abc}
Userid	{USERID} ^{abc}
Password	{PASSWORD} ^{abc}

Figure 9-20 Data list as single-column data table

Every record retrieved is shown with the labels and all the fields.

By default the data table has no border. You can select the data table and in the Attributes view All tab set the border attribute to 1 (and possibly other table attributes (Figure 9-21).

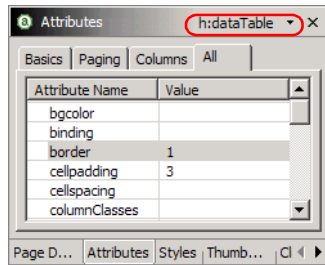


Figure 9-21 Setting table attributes

Execution

Run the page in the server and the query is executed and the list is displayed (Figure 9-22).

Multi-Column Table

Customerid	Title	Firstname	Lastname	Userid	Password
101	Mr	Maik	Schumacher	cust101	MS
102	Mr	Fabio	Ferraz	cust102	FF
103	Mr	Henrik	Sjostrand	cust103	HS
104	Mr	Ian	Brown	cust104	IB
105	Ms	Unknown	Lady		
106	Mr	Ueli	Wahli	cust106	UW

Single-Column Table

Customers	
Customerid	101
Title	Mr
Firstname	Maik
Lastname	Schumacher
Userid	cust101
Password	MS
Customerid	102
.....	

Figure 9-22 Results of running the relational record list

When the page is displayed the `getCustomers` method in the page code is executed to retrieve the data. The data is then displayed in the chose format.

Retrieving a single record

The same process can be used to display a single row of data. The key differences are that when you create the record you are prompted to decide if the object is filled with an existing record or used to create a new record (see Figure 9-2 on page 303). For now we retrieve data.

The default filter is based on a parameter that is passed to the JSF page. In many cases you want to retrieve a row based on another Page Data variable, for example a session variable. Click on *Filter results* when defining the record and define the condition. You can either edit the existing condition or delete it and define a new condition (Figure 9-23).

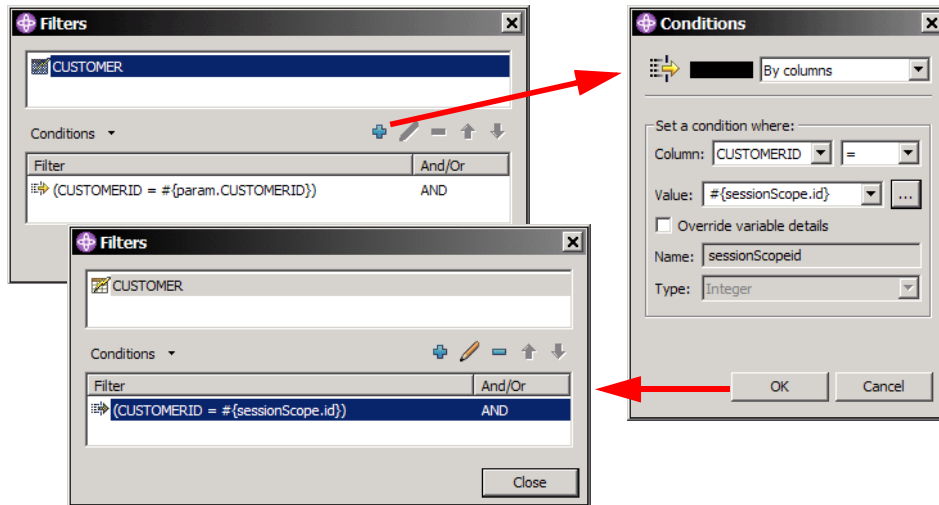


Figure 9-23 Defining a filter

When you come to drop the relational record on the JSF page you get a similar dialog to the relational record list, but this time you have the ability to use the record for updating (Figure 9-24).

If you select *Updating an existing record*, all control types change to Input Field.

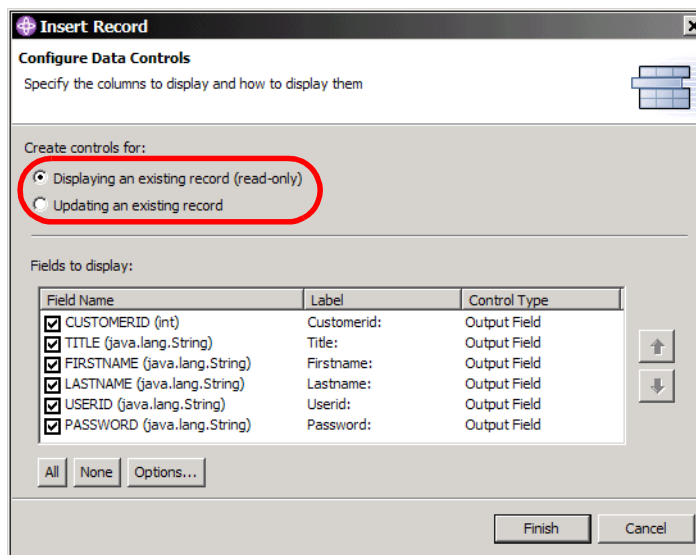


Figure 9-24 Displaying a single record

Once you complete this step an HTML table is built to display the record (Figure 9-25). You can set attributes of the table in the Attributes view.



Customer	
Customerid:	{CUSTOMERID} abc
Title:	{TITLE} abc
Firstname:	{FIRSTNAME} abc
Lastname:	{LASTNAME} abc
Userid:	{USERID} abc
Password:	{PASSWORD} abc
* {Error Messages}	

Figure 9-25 HTML table to display a single record

When you run this page the data is displayed as shown in Figure 9-22 on page 318 (right side).

With the relational record list and relational record objects you can start to deliver very powerful applications with little or no code development. This shows how SDO and JSF, when used together, allow for very productive development.

Data update

Performing a data update operation with SDO is very simple when used in conjunction with the Studio tooling. Using the drag and drop operation allows the SDO updates to be added to pages. The detail behind how data update is performed at an API level is covered in “SDO API” on page 329.

This section of the chapter explains how to use the Studio tooling to add data update operations to you applications. There is a very detailed example in “Create, read, update and delete (CRUD) pattern” on page 242.

The default update operations on a relational record object are dependent on the option selected when creating the record (Figure 9-2 on page 303). When you create a relational record component the default path through the wizard is to create an object that reads a record from a relational database. Such a record can be used for update and delete operations.

Update and delete

Update and delete operations are performed on a record that was used to retrieve the data. When such a record is dropped into a JSP you select that input fields are creates so that you can update the data. In addition the dialog can generate Update and Delete buttons with associated action code for you.

Let's build a simple customer record for updating. In the Page Data view select *New -> Relational Record*, connect to the EJBBANK database, select the CUSTOMER table, and click *Finish*.

For this simple example we change the filter to a fixed value (in general the key would be supplied from an input field or another page):

- Select the customer object and *Configure* (context). On the Conditions tab change the default condition to the fixed value of **101** (Figure 9-26).

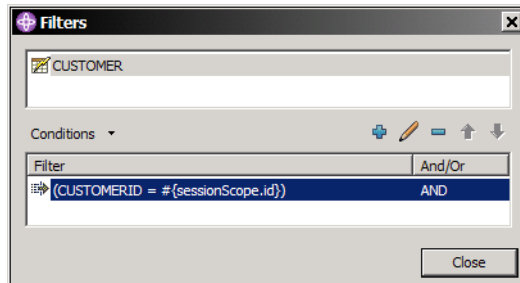


Figure 9-26 Setting a fixed value filter

Creating the form for update and delete

Drop the customer object into a JSF page. In the dialog (Figure 9-27):

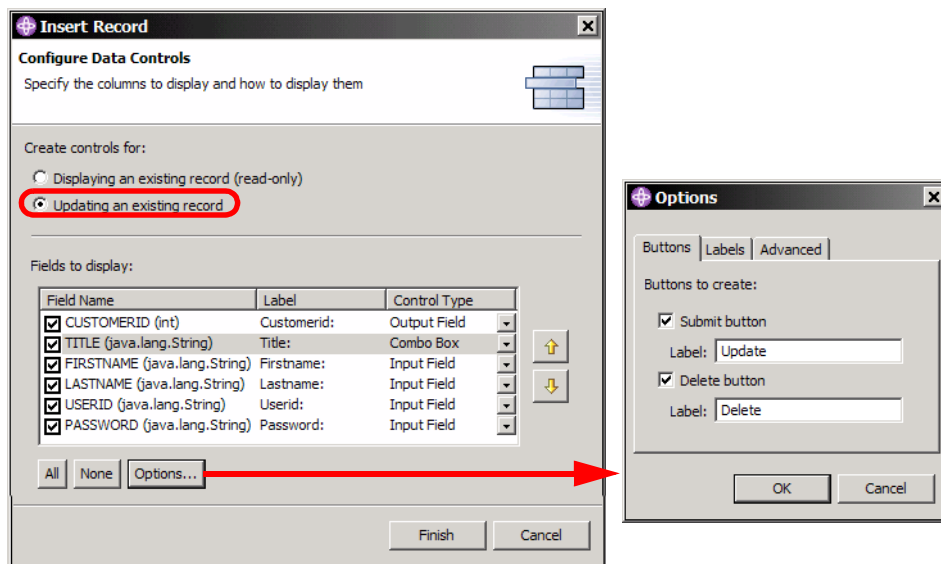


Figure 9-27 Selecting the options to create a form for update

- ▶ Select *Updating an existing record*.
- ▶ Change CUSTOMERID to an output field (the key cannot be updated).
- ▶ Change TITLE to a combo box (there are only three values allowed).
- ▶ You could tailor the generated labels.
- ▶ Click *Options* to select Update and Delete buttons.
- ▶ Click *Finish* to insert the table into the JSP.

When you have completed this dialog the form fields are created on the page (Figure 9-28). You can also see that the {Error Messages} component is created to display any errors in the update fields before the update action is executed.

The drop operation creates all the underlying JSF components and the action code required to perform the update of this relational record. This is a good example of how combining the JSF with Studio can create real productivity.

Figure 9-28 Form to update and delete a relational record

Select the title combo box and in the Attributes view Choices tab add three text items with values Mr, Mrs, and Ms.

Action code

Select the Update or Delete button and study the action code in the Quick Edit view (Example 9-1).

Example 9-1 Page code methods for the Update and Delete buttons

```
public String doButton1Action() { <===== UPDATE
    try {
        getCustomerMediator().applyChanges(
            getCustomer().getDataGraphAccessBean());
    } catch (Throwable e) {
        logException(e);
    }
    return "";
}
```

```

public String doButton2Action() { <===== DELETE
    try {
        getCustomer().getDataGraphAccessBean().deleteDataObject();
        getCustomerMediator().applyChanges(
            getCustomer().getDataGraphAccessBean());
    } catch (Throwable e) {
        logException(e);
    }
    return "";
}

```

This code gets the changes from the form and passes the data object to the `applyChanges` method of the mediator. The changes are then committed to the database.

As you can see performing an update or delete of a record is very straight forward with SDO and the relational record component within Studio. The combination of JSF, SDO, and Studio adds real productivity to these types of operations.

Tip: The default behavior of the Delete button is to delete the record once the button is clicked. A useful tip is to put a piece of JavaScript into the `onClick` event so that the user can confirm the delete operation before the record is actually deleted:

```

var agree = confirm("Are you sure you want to delete this entry ?");
if (agree)
    return true;
else
    return false;

```

This JavaScript prompts you before the event is send back to the server for the delete operation to occur.

Create

We can use the relational record component to perform an SQL INSERT operation through a JSF page. It is very similar to the steps that were used to perform the update process. The main difference is that you create the component as a blank record and you do not have to define a filter.

When you create the relational record component you specify that it is going to be used for a create (Figure 9-29).

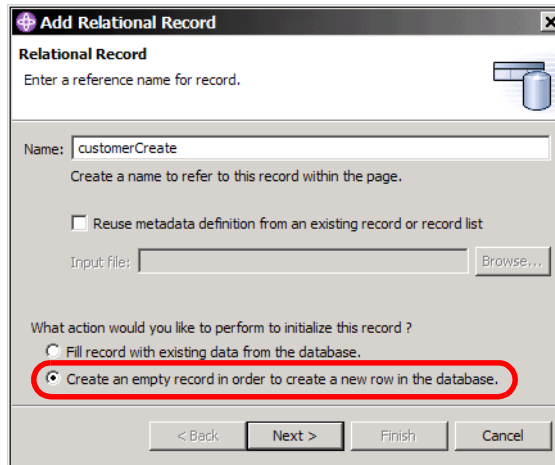


Figure 9-29 Creating a relational record component for insert

When you drag the relational record component from the Page Data view to the JSF page you see a familiar dialog. Studio knows that this record is being used for create and limits the configuration of the dialog (Figure 9-30).

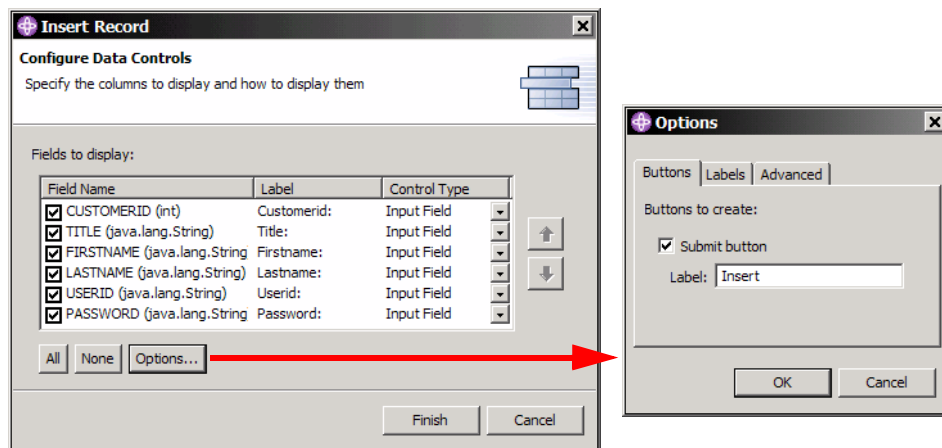


Figure 9-30 Creating the form for insert

Once the drag has been completed the form is created. At this point you can customize it for use within you application. They key elements have been created, such as the form layout, the {Error Message} component, and the *Insert* button for performing the create (Figure 9-31).

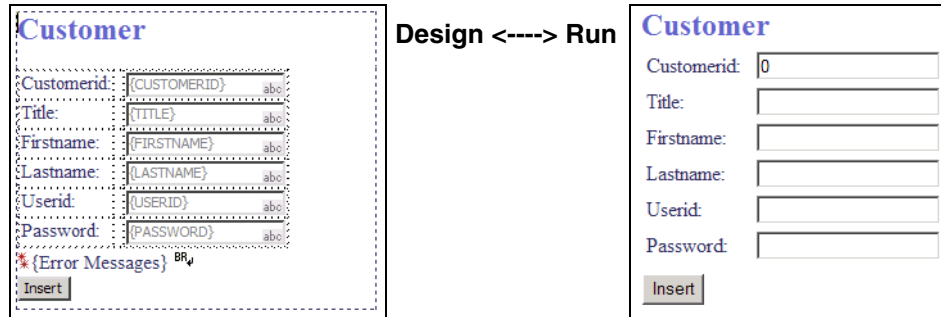


Figure 9-31 Form for inserting a relational record

The action code of the Insert button is the same code that is generated by the update operation. The difference is that the relational record is being used for create so the underlying datagraph is prepared for an insert operation on the relational database.

Auto key generation

Automatic key generation can be used by a relational record to allow a unique key to be generated by the create operation. An example is described in “Building the create part of CRUD” on page 252.

Deleting SDO objects

Deleting unused SDO objects involves a number of steps:

- ▶ Delete the object in the Page Data view.
- ▶ Delete the XML file in the WEB-INF\wdo folder.
- ▶ Delete the variables and methods in the page code Java class (if the object was once used in the associated JSP).

Deleting unused connections involves a number of steps:

- ▶ Delete the resource reference in the web.xml deployment descriptor.
- ▶ Delete the connection information from the .wdo-connections file in the project.
- ▶ Delete the data source from the test server.

Using SDO in JSPs

SDO is not limited to JSF pages. You can use SDO objects in basic JSP pages as well.

Relational record list

We do not explore this option in detail in this book, but here is a small example:

- ▶ Create a regular JSP page named **nonJSF.jsp**. Change the default title.
- ▶ In the Page Data view select *New -> Relational Record List* and name it **goldAccounts**.
- ▶ Select the ACCOUNTS table, then select the ACCID, BALANCE, INTEREST, and ACCTYPE fields.
- ▶ Select *Filter results* and define the condition BALANCE > 999.99.
- ▶ Select *Order results* and sort on BALANCE, descending.
- ▶ Click *Finish*.
- ▶ Drag and drop the goldAccounts object into the JSP. Order the fields and change the labels. Click *Finish*.

Generated tags

All the code is in JSP tags. Here is an extract of the tags:

```
<%@taglib uri="http://www.ibm.com/websphere/wdo/core" prefix="wdo"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
.....
<%-- wdo:useDataList id="goldAccounts" mediator="goldAccountsMediator"
      input="/WEB-INF/wdo/goldAccounts.xml" datatype="RDB" action="FILL" --%>
<%-- /wdo:useDataList --%>

<wdo:useMediator id="goldAccountsMediator"
      input="/WEB-INF/wdo/goldAccounts.xml" connection="EJBBANK_Con1"
      datatype="RDB">
</wdo:useMediator>
<wdo:execute mediator="${goldAccountsMediator}" id="goldAccounts" />
.....
<table>
  <thead><tr>
    <th>Accid</th><th>Type</th><th>Balance</th><th>Interest</th>
  </tr></thead>
  <tbody>
    <c:forEach var="varGoldAccounts" items="${goldAccounts}">
      <tr>
        <td><c:out value="${varGoldAccounts.ACCID}" /></td>
```



```

        <td><c:out value="\${varGoldAccounts.ACCTYPE}" /></td>
        <td><c:out value="\${varGoldAccounts.BALANCE}" /></td>
        <td><c:out value="\${varGoldAccounts.INTEREST}" /></td>
    </tr>
</c:forEach>
</tbody>
</table>

```

Relational record

Next we create a relational record for updating:

- ▶ Create a relational record named account in the Page Data view.
- ▶ Use the ACCOUNT table and select the first four columns.
- ▶ Change the default condition to ACCID='101-1001'.
- ▶ Drop the account record into the JSP. Select *Updating an existing record*. Change the control type for ACCID and ACCTYPE to an output field.
- ▶ Click *Options* and select Update and Delete buttons.

Generated tags

The tags for updating are somewhat more complex:

```

<%-- wdo:useDataObject id="account" mediator="accountMediator"
      input="/WEB-INF/wdo/account1.xml" datatype="RDB" action="FILL" --%>
<%-- /wdo:useDataObject --%>

<wdo:useMediator id="accountMediator" input="/WEB-INF/wdo/account1.xml"
      connection="EJBBANK_Con1" datatype="RDB">
</wdo:useMediator>
<wdo:find mediator="\${accountMediator}" id="account"></wdo:find>

<c:choose>
  <c:when test="\${param.submit != null}">                                     <=== UPDATE

    <c:set var="formField" value="\${param.ACCID}" />
    <c:set property="ACCID" target="\${account}" value="\${formField}" />

    <c:set var="formField" value="\${param.BALANCE}" />
    <c:set property="BALANCE" target="\${account}" value="\${formField}" />
    .....
    <wdo:commit mediator="\${accountMediator}" dataObject="\${account}" />

    <%--Forward off to a new page when the action is complete
      <jsp:forward page="index.jsp"></jsp:forward>
    --%>
  </c:when>

```

```

<c:when test="${param.delete != null}">                                     <=== DELETE
    <wdo:remove mediator="${accountMediator}" dataObject="${account}"
        commit="true" />
    <%--Forward off to a new page when the action is complete
    <jsp:forward page="index.jsp"></jsp:forward>
    --%>
</c:when>
</c:choose>

<form action="nonJSF.jsp" method="post">
<table border="1">
..... table with input fields .....

```

Notice the `<jsp:forward>` tags in comments. They can be changed to navigate to the next JSP after an update or delete.

Running the JSP with SDO

When running the `nonJSF.jsp` the `<wdo>` tags execute the database requests and the data is displayed in HTML tables (Figure 9-32).



Figure 9-32 Executing a regular JSP with SDO

SDO API

To exploit the full flexibility of the SDO framework, it is important to understand how to programmatically use the artifacts and functions generated by the Studio when you create relational records and relational lists. Please refer to this chapter's introduction for a basic discussion of the SDO architecture.

Data object

The data object (`DataObject`) is designed to be an easy way for a Java programmer to access, traverse, and update structured data. Data objects have a rich variety of strongly and loosely typed interfaces for querying and updating properties. The data object provides a way to access its container object and its containing data graph. It also provides methods to delete the object or to create a new data object. Finally, it provides the ability to introspect the data object's type.

Important methods

The important methods of the data object are:

- ▶ `Object get(String path)`
This method accepts a path statement, which is explained in detail later (see “Query syntax” on page 330), to find a particular property of this data object. There is also a version of this method that takes an index (`int`) parameter and there are strongly typed versions of this function as well.
- ▶ `void set(String path, Object value)`
This method stores a new value at the location specified by the given path statement. The index parameter version of this method also exists, as well as the strongly typed versions.
- ▶ `DataObject getContainer()`
This method returns the containing data object for this object, or `null` if none exists. This is useful for traversing upwards in the container hierarchy.
- ▶ `DataGraph getDataGraph()`
This method returns the containing data graph for this data object.
- ▶ `DataObject createDataObject(String path)`
This method creates a new data object at the specified path and returns it.
- ▶ `void delete()`
The method removes this data object from its container and its data graph.

Query syntax

The get and set methods' path parameter syntax is based on the XML Path Language (XPath), which was designed as a syntax for addressing parts of an XML document. The XPath specification can be found at:

<http://www.w3.org/TR/xpath>

Paths are constructed from a given data object as though a contained object is a sub-path of that object. Objects may also be addressed by index in a path statement. Finally, objects can be referenced by whether or not their properties match a specific value. Sample path statements are shown in Table 9-1.

Table 9-1 Example path statements

Path from Given Data Object	Reference
"name"	The name property of this data object
"address/zipCode"	The zipCode property of the address property of this data object
"accounts[1]/balance"	The balance property of the 1st (0-indexed) object in the accounts property of this data object.
"accounts.1/balance"	Same as previous example
"accounts[userName=Luis]"	The first object in the accounts property of this data object with an userName property of Luis
".."	The containing data object for this data object

These examples should be sufficient for most tasks, but refer to the XPath specification for a complete language reference.

Data graph

The data graph (DataGraphAccessBean) is a container for a single root data object, as well as any of its contained data objects. The data graph also contains some metadata about its root data object, including change summary information and type data.

Data graphs can be serialized to and deserialized from XML by the client. Once the data graph has been obtained, the client is expected to manipulate the contained data objects in place.

Important methods

The important methods of the data graph are:

- ▶ `DataObject getRoot()`

This method returns the root data object of this data graph.

- ▶ `DataObjectAccessBean createDataObject()`

This method creates a new data object and sets it as the root of the data graph. The new data object can be accessed using the `getRoot` method.

Data mediator

The data mediator (`MediatorAccessBean`) is the interface for obtaining data graphs from and sending updated data graphs to a data source. The mediator supplies functions to create completely new data on the data source as well. The supplied mediator implementation (`JDBCMediatorAccessBean`) is reusable for all JDBC database access, so it must be configured with properties describing the specific data source. Hereafter, we specifically address working with the JDBC mediator.

Important methods

The important methods of the data mediator are:

- ▶ `DataGraphAccessBean fetchGraph()`

This method invokes the JDBC query and constructs a data graph from the results.

- ▶ `void applyChanges(DataGraphAccessBean graph)`

This method executes any changes in the given data graph as database update statements.

- ▶ `DataGraphAccessBean createEmptyGraph()`

This method creates an empty data graph, which is used when creating new records. This method allows you to create a new record the database without retrieving any data first.

- ▶ `Map getParams()`

This method retrieves the parameter map, which is used to supply any dynamic properties used in the database query. You can update this map directly if you want to programmatically modify the query.

WDO implementation

In Application Developer, we are supplied with slightly different objects. If you create a new relational record on a page called *myObject*, several methods and fields are generated for you in the page code:

► **private DataObjectAccessBean myObject**

This is a local copy of the most recently retrieved data. The `DataObjectAccessBean`, or `DataListAccessBean`, which correspond to a relational record or a relational list, are constructs that wrap the various objects in the SDO architecture. We recommend you work with the underlying SDO objects directly, as they are more flexible and more functional. These objects can be retrieved directly from either type of wrapper:

To retrieve the data graph: `myObject.getDataGraphAccessBean()`

To retrieve the data object: `myObject.getDataGraphAccessBean().getRoot()`

► **public DataObjectAccessBean getMyObject()**

If the `myObject` `DataObjectAccessBean` is not null, this method returns it. If it is null, this method invokes the mediator to fetch new data, set the `myObject` variable, and return it. Therefore, any access to other objects should use the `getMyObject` method:

```
getMyObject().getDataGraphAccessBean()  
getMyObject().getDataGraphAccessBean().getRoot()
```

► **private static myObject_connectionName**

This is the connection name as specified in the SDO wizard.

► **private static myObject_metadataFileName**

This is a reference to the file path of the XML metadata for this SDO.

► **protected JDBCMediatorAccessBean getMyObjectMediator()**

The `getMyObjectMediator` method returns the data graph's JDBC mediator. The mediator works by executing the query stored in the XML metadata for the SDO. Any filter parameters for the SDO are looked up in the mediator's parameter hash table when executing a graph fetch. This method instantiates the mediator if necessary using the `myObject_connectionName` and `myObject_metadataFileName` values.

► **protected static String[] myObjectArgNames**

These are the filter parameter names for the SDO's query.

► **protected static String[] myObjectArgValues**

These are the filter parameter values for the SDO query. Dynamic JSF value binding strings, which will be resolved at runtime, may be used here.

Retrieving a record

To manually retrieve query results, you update the mediator parameters and fetch a new data graph. Then, you retrieve the data object from that graph and you have the resulting data object (Figure 9-33):

```
// update any dynamic mediator parameter values
resolveParams(getCustomerRecordMediator().getParams(),
    customerRecordArgNames,
    customerRecordArgValues,
    "customerRecord_params_cache");
DataGraphAccessBean graph = null;
// invoke the DB operation and retrieve the data graph
try {
    graph = getCustomerRecordMediator().fetchGraph();
} catch (MediatorException e) {
    e.printStackTrace();
    return;
} catch (IOException e) {
    e.printStackTrace();
    return;
}
// update the page code's copy of the data wrapper
customerRecord = graph.getDataObjectAccessBean();
// pull out the real data object
DataObject results = graph.getRoot();
```

Figure 9-33 Example of manually retrieving data

In the example we use a convenience method, `resolveParams`, which uses the supplied parameters to refresh the mediator's parameters. This method is provided by the `PageCodeBase` class. We also update the local copy of the wrapper bean, in case it is bound to any UI components.

Note: This code is generated into the `getMyObject` method that is generated by Application Developer (except for the last line (`getRoot`)).

Dynamic queries

You may want to modify the way an SDO query works without creating a new object altogether. You are limited to modifying SDO condition filter parameters and result ordering.

Modifying a filter parameter may be useful depending on your application. To do this, simply update the corresponding mediator parameter as follows:

```
getCustomerRecordMediator().getParams().put("customerID", "103");
```

Then, execute the query and grab the new data graph.

```
DataGraphAccessBean graph = getCustomerRecordMediator().fetchGraph();
```

Remember that it is possible to perform filtering on the already retrieved results. For instance, the record found by the above query modification can be found in the existing result set, using an XPath statement:

```
getCustomerRecord().getDataGraphAccessBean()  
    .getRoot().get("CUSTOMER[CUSTOMERID=103]");
```

Updating a record

To update a record, you must fetch the data graph containing the record from the mediator, make the changes in the appropriate data object, and apply it back to the mediator.

When you make the changes, you can retrieve the underlying data object as described previously. You may only update data object properties that correspond to columns you selected when creating the data object. Here is an example of updating a property in a data object:

```
getCustomerRecord().getDataGraphAccessBean()  
    .getRoot().set("CUSTOMER.0/FIRSTNAME", "Susy");
```

When you drop a relational record into a JSF page for updating, Application Developer can automatically create a Submit button with attached action logic to apply the data back to the mediator:

```
try {  
    getCustomerRecordMediator().applyChanges(  
        getCustomerRecord().getDataGraphAccessBean());  
} catch (Throwable e) {  
    logException(e);  
}
```

This code is sufficient for use anywhere when you want to update the database with the current contents of the data graph. If you want to embed the update call in more complex business logic, feel free to relocate it.

Note: The default behavior of the JDBC mediators is to immediately update the database upon an `applyChanges` call. This is called auto-commit. If you have multiple mediators with changes, you may want to perform all updates at once. Refer to “Single transaction commit” on page 337 for details.

Deleting a record

To delete a record, you must fetch the data graph containing the record from the mediator, call the delete function on the appropriate contained data object, and send the data graph back to the mediator.

When you drop a relational record into a JSF page for updating, you can optionally create a Delete button that is bound to this generated action code:

```
try {
    getCustomerRecord().getDataGraphAccessBean().deleteDataObject();
    getCustomerRecordMediator().applyChanges(
        getCustomerRecord().getDataGraphAccessBean());
} catch (Throwable e) {
    logException(e);
}
```

Note that it is not obvious from the above example which data object in the graph is being deleted. The previous code deletes the 0-indexed data object in the data graph's root. It is possible, however, to delete a specific data object as specified by a path in the data graph root. Substitute a call to `DataObject.delete()`, as seen in this code fragment:

```
try {
    //getCustomerRecord().getDataGraphAccessBean().deleteDataObject();
    getCustomerRecord().getDataGraphAccessBean().getRoot().
        getDataObject("CUSTOMER[CUSTOMERID=103]").delete();
    getCustomerRecordMediator().applyChanges(
        getCustomerRecord().getDataGraphAccessBean());
} catch (Throwable e) {
    logException(e);
}
```

Note: There is no way in the SDO architecture to delete a data object without first retrieving it from the mediator.

Inserting a record

To create a new record, you must first access an existing data graph or create a new data graph. Then you create a new data object in the data graph and apply changes back to the mediator.

Adding a new data graph

If you do not have an existing data graph to work within, you must generate a new data graph from the mediator.

If you drop a relational record for inserting in Application Developer, the page code is generated such that retrieving the `DataObjectAccessBean` for the first time creates a new data graph and adds an empty data object to it. Here is the code that creates and populates the new data graph:

```
DataGraphAccessBean graph =
    getCustomerRecordMediator().createEmptyGraph();
customerRecord = graph.createDataObject();
```

Once you are done setting the properties on the empty data object in the graph, apply changes to the mediator as normal.

Adding to an existing data graph

You can create new objects within an existing data graph underneath the root data object (or any contained data object). You must specify the type of the new object when creating it. The type corresponds to the database table name. Next you must populate any required columns in the data object, most likely the primary key. Here is a code fragment creating a data object directly:

```
//instantiate blank data object of type CUSTOMER in the data graph
DataObject newCustomer =
    getCustomerRecord().getDataGraphAccessBean().
    getRoot().createDataObject("CUSTOMER");
//configure the blank data object
newCustomer.set("FIRSTNAME", "Ron");
newCustomer.set("CUSTOMERID", new Integer(123));
```

You then apply the data graph to the mediator as normal.

Calculating a new primary key

Generally, database records require a unique value for the primary key, which cannot be computed directly without accessing the database. You may want to use a specialized SDO to retrieve the highest primary key in the database, so you can calculate the new key correctly.

If you are working with a data graph with a single table level data object, you may invoke a convenience method in the mediator which essentially does this operation for you:

```
getCustomerRecordMediator().autoGenerateKey(getCustomerRecord());
```

Note: Automatic key generation must be setup in the database as described in the example “Building the create part of CRUD” on page 252.

Joined records

If you have created a relational record that has relationships defined from its root table, the path statements you create to traverse those relationships are specialized.

Suppose you have created a relationship from the CUSTOMER table to the CUSTACCT table, then create another relationship from CUSTACCT to ACCOUNT. Now you would like to access an account from the data graph root. You can access an account in two different ways:

- Accessing an account directly can be accomplished because the account is itself a data object. Use this path to access the account balance:

```
"ACCOUNT[ACCID=103-3001]/BALANCE"
```

- If you want to access an account as a property of the customer, you must use sub-paths that represent the parent table name concatenated (with an underscore) with the child table name. For example, you might generate a path to the balance of the first account of customer 103 as followings:

```
"CUSTOMER[CUSTOMERID=103]/CUSTOMER_CUSTACCT.0/CUSTACCT_ACCOUNT/BALANCE"
```

Note, if you serialize the data graph to XML, you will see these table name concatenations used as property names for the parent object. Also, these name concatenations are used in the Page Data view as the labels for trees which represent relationships.

Single transaction commit

If you modify the database using multiple mediators, each `applyChanges` call on a mediator immediately commits its database transaction. In some cases, the changes you wish to make may occur across multiple mediators, but must occur in the same transaction. For rollback purposes, you do not want to commit any change without all changes successfully going through.

To fix this problem, you must construct the mediators in such a way that they all share the same underlying connection to the database, and that all of the actions from the various mediators occur in the same transaction on that connection.

You can change the generate code which instantiates the mediators to accomplish this. Create a new passive connection wrapper, using the connection name stored in the page code:

```
conn = ConnectionManager.createPassiveConnectionWrapper  
      (customerRecord_connectionName);
```

A passive connection wrapper does not commit the database transaction when the mediators that use it have their `applyChanges` method called. The commit of the transaction must be invoked directly.

Modify the generated code which constructs the various mediators to use the connection wrapper, instead of each generating its own wrapper:

```
customerRecordMediator = new JDBCMediatorAccessBeanImpl (
    getRealPath(account_metadataFileName),
    //customerRecord_connectionName);
    conn);
```

Now, when performing the various data modifications, call `applyChanges` to the mediators as normal. When you have finished all of the data modification, commit the transaction and close the connection:

```
conn.getConnection().commit();
conn.getConnection().close();
```

Note: Refer to “Implementing single transaction commit” on page 369 for a detailed example of implementing these changes.

Debugging data graphs

Data graphs, as disconnected objects, can be easily serialized to XML. Viewing the active data graph at various points in the application can be extremely useful. You can view the raw database query results, as well as verify that any data modifications happen as expected.

You can use an object output stream, writing to a byte array, to get a string containing the XML representation of the data graph. Here is a sample function that converts a data graph into an XML string (`\labscode\dev-sdoapi\serializeGraph.txt`):

```
public String serializeGraph(DataGraphAccessBean graph) {
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    ObjectOutputStream out = null;
    try {
        out = new ObjectOutputStream(stream);
        out.writeObject(graph);
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    return stream.toString();
}
```

The XML representation of a data graph has some useful features. Note the `<models>` portion of the data graph XML. This contains the EMF definitions for the data objects in this graph. The `<datagraph:root>` section contains all of the data objects stored in the datagraph's root. The `<events>` section contains change summary information, including updates, additions, and deletions.

Summary

SDO is a powerful architecture that brings access to data closer to the Web application designer. SDO can be integrated with JSF pages and data access is generated into the page code Java classes. SDO can also be integrated with regular JSPs and access is performed through the tag library provided by SDO. Finally, SDO can be used programmatically in any server-side code.

SDO objects are defined using XML metadata files that are used by the SDO runtime.

Application Developer makes the use of SDO easy by providing a palette of SDO objects and an SDO wizard to generate the XML metadata definition using live database connections. In addition Application Developer generates the Java code to retrieve, update, and delete SDO objects.



SDO banking application

This chapter details how you can use Service Data Objects (SDO) to implement the banking application described in Chapter 5, “JSF banking application” on page 191.

This chapter describes the application architecture and a step-by-step guide to building the first SDO-based JSF application.

SDO banking application overview

We now build an end-to-end banking application using the JSF framework and the WebSphere Studio tooling for SDO. The advantage of using SDO is that you do not require interface code between the database and the JSF application. You use SDO to access the relational database (EJBBANK) without writing any JDBC or SQL statements. By using the SDO tooling, you are able to rapidly engineer the application interface to the database.

For a general description of the banking application see “Banking application overview” on page 192.

SDO banking application architecture

The architecture of the application is shown in Figure 10-1.

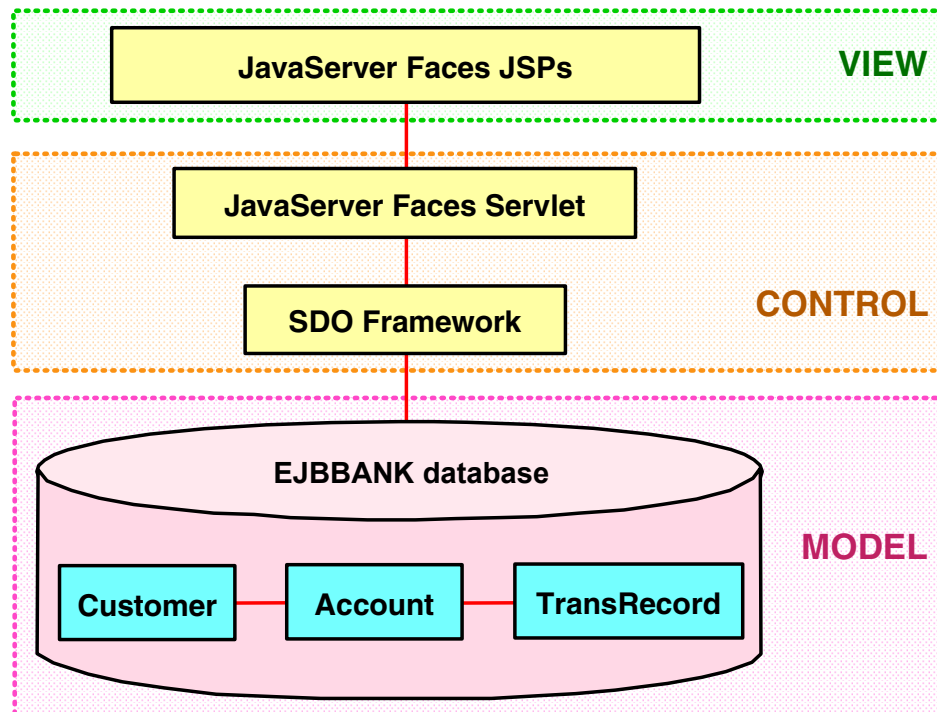


Figure 10-1 SDO banking application architecture

Model

The model is the EJBBANK database with tables for customers, accounts, and transaction records. There are no Java classes predefined for the model. The SDO framework objects for relational record and relational record list are used as data objects in the view.

The EJBBANK is described in “EJBBANK database” on page 196.

Controller

The controller is implemented using the JSF servlet and the SDO framework. We define a number of SDO objects to access the EJBBANK database:

- ▶ **customerName**—retrieve a customer name (firstname and lastname) by key.
- ▶ **customerAccounts**—retrieve the list of accounts for a customer.
- ▶ **account**—retrieve the selected account.
- ▶ **fromAccount**—retrieve and update the account for banking operations.
- ▶ **destAccount**—retrieve and update the destination account for a transfer operation.
- ▶ **trans**—insert a transaction record for banking operations.
- ▶ **listTransactions**—retrieve the transaction records for an account.
- ▶ **transDetail**—retrieve a selected transaction record.

View

The view is implemented using JSF JSPs:

- ▶ **index.jsp**—welcome page to enter a customer key.
- ▶ **listAccounts.jsp**—list the accounts of a customer for account selection.
- ▶ **accountDetails.jsp**—display account details and perform transactions (list transactions, deposit, withdraw, transfer).
- ▶ **transferResult.jsp**—display account details after a transfer.
- ▶ **listTransactions.jsp**—display the list of transaction records for an account.
- ▶ **transactionDetails.jsp**—display the details of one transaction record (not shown in Figure 10-2).

Implementation details

The complete structure of the JSF banking application is shown in Figure 10-2.

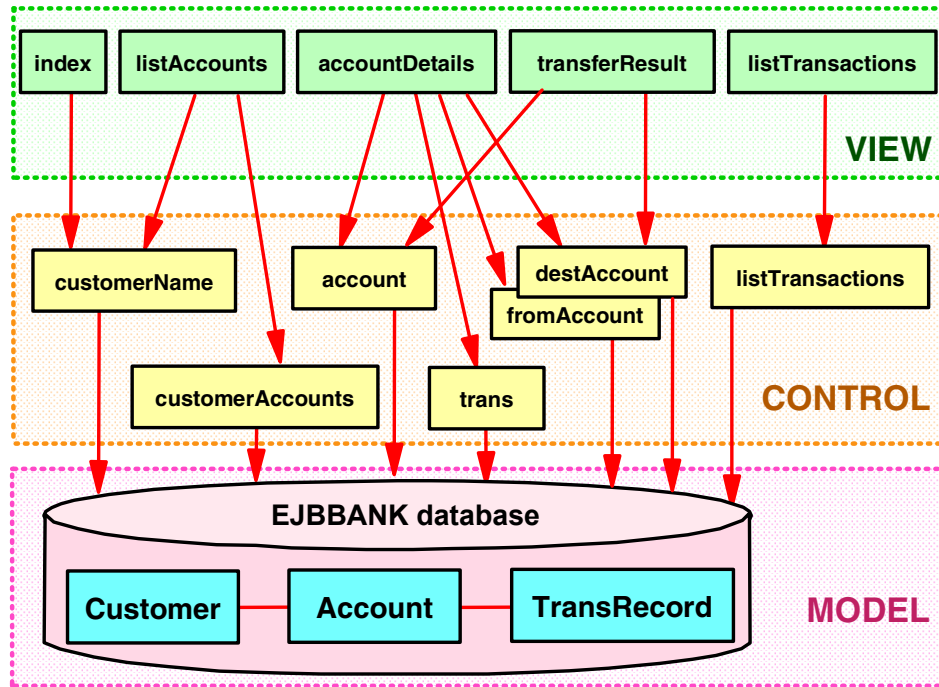


Figure 10-2 JSF banking application

Note that the `transactionDetails.jsp` is not shown to simplify the drawing.

Preparation

Before implementing the SDO banking application we have to setup the environment as described in “Setting up the environment for this redbook” on page 533:

- ▶ Create and load the EJBBANK database
- ▶ Setup server targeting for the workspace
- ▶ Prepare the JSFServer for testing

Implementing the banking application with SDO

In this section we provide detailed instructions to implement the SDO banking application. We implement the banking application with a JSF frontend and SDO for database access.

Create an enterprise application

We define an enterprise application to hold the SDO banking application:

- ▶ Select *New -> Project -> J2EE -> Enterprise Application Project*. Select J2EE 1.3 and click *Next*. Enter **ItsoJSF10SD0** as name. For Target Server verify *WebSphere Application Server v5.1*. Click *Finish*. Do not switch perspectives.
- ▶ In the Servers view, select the JSFServer and *Add and remove projects* from the context menu. Select the ItsoJSF10SD0 project and click *Add >* to associate the enterprise application with this server. You can remove all other projects for faster starting of the server. Click *Finish*.

Creating the Web project

- ▶ Create a Web Project (*New -> Dynamic Web Project*):
 - Enter **ItsoJSFSD0** as the name, and select *Configure advanced options*.
 - Select ItsoJSF10SD0 as the EAR project and select *Use the same target server as the EAR project*.
 - Select *WDO Relational database runtime* in the Web project features, as seen in *Figure 10-3*.

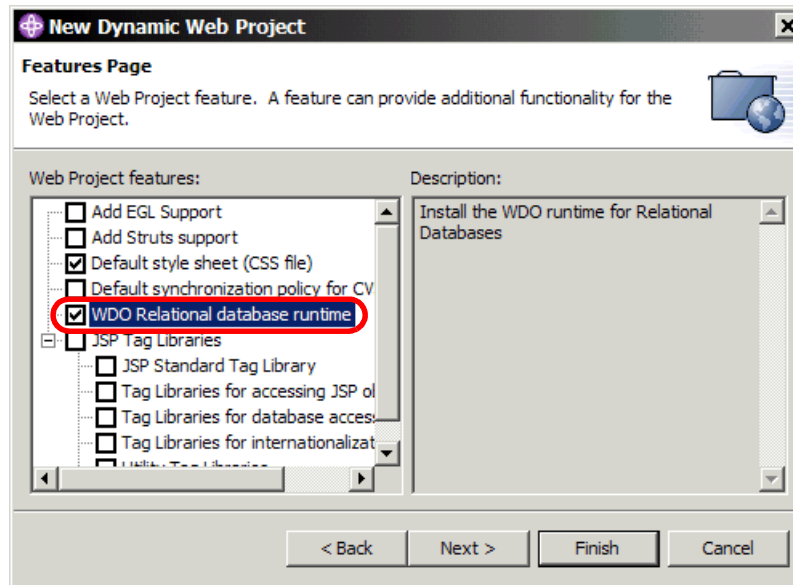


Figure 10-3 Web project features: WDO runtime

- Click *Finish*.
- Click *OK* to repair the server configuration.

You now have a Dynamic Web Project with WDO runtime support.

- ▶ Import the images folder under WebContent:
 - Import from `\labscode\images`.
 - Or, copy and paste the images folder from the previously created `ItsoJSFBank` project.

Creating the customer ID query page

The first page we create contains the form to enter the customer ID. This form tests for the existence of the identified customer, and if successful, redirects to the customer account listing page:

- ▶ Under WebContent, create a JSF page (*New -> Faces JSP File*). Enter **index.jsp** as the name and click *Finish*.
- ▶ Design the `index.jsp` with a heading, output component, input field with error message field, and a Submit button (Figure 10-4).



Figure 10-4 Design view of index.jsp

- Set the id of the input field to **customerIDinput** before generating the error message field.
- Set length to **12**, and validation to *required*. The minimum and maximum values should both be **3**, and the field should be set to *Digits only*.
- Set the id of the Submit button to **submit**.
- ▶ In the Page Data view, create two session scope variables named **accountID** and **customerID** (java.lang.String). We will have these variables available as session data for the complete user interaction.
 - Bind the customerIDinput field to the customerID session variable using drag and drop or through the Attributes view.
- ▶ Create two empty JSF pages, **listAccounts.jsp** and **error.jsp**. Close the files for now.
- ▶ Select the Submit button (in index.jsp) and set up two navigation rules:
 - Global: **error** to error.jsp
 - Local: **success** to listAccounts.jsp
- ▶ Save the index.jsp.
- ▶ Open the theme\stylesheet.css file. Add **color: red;** to .message and .messages to display all errors in red.

```
.message { color: red;
}
.messages { color: red;
}
```

Retrieving the customer name using a SDO

The first use of SDO is to retrieve the customer name. We set up a data object (relational record) that is invoked in the Submit button action logic:

- ▶ In the Page Data view select *New -> Relational Record* (Figure 10-5):
 - Enter **customerName** as name.

- Select *Fill record with existing data from the database*.

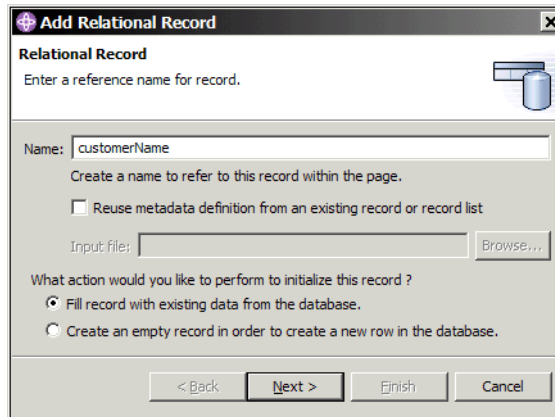


Figure 10-5 Add Relational Record Dialog

- Click *Next*.

Creating the database connection

We must define a new database connection for the relational record:

- ▶ Click *New* to create a connection (Figure 10-6).
- ▶ Change the connection name to **EJBBANK_Con1** and select *New Connection*.

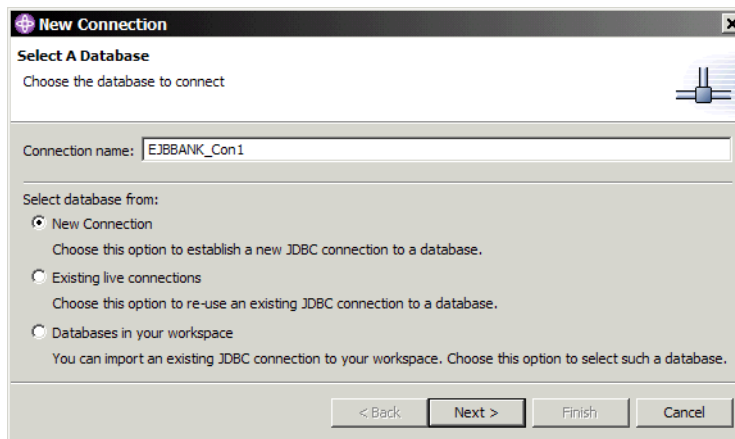


Figure 10-6 Creating a database connection

- ▶ Click *Next*.

- Define the database connection parameters (Figure 10-7):
 - Database name: EJBBANK
 - Vendor: IBM DB2 Universal Database v8.1
 - Driver: IBM DB2 APP DRIVER

New Connection

Database Connection
Establish a JDBC connection to a database.

Connection name: EJBBANK_Con1

Database: EJBBANK

User ID:

Password:

Database vendor type: DB2 Universal Database V8.1

JDBC driver: IBM DB2 APP DRIVER

Host:

(Optional) Port number:

Server name:

Database Location: Browse...

JDBC driver class: COM.ibm.db2.jdbc.app.DB2Driver

Class location: C:\SQLLIB\java\db2java.zip Browse...

Connection URL: jdbc:db2:EJBBANK

Filters...

< Back Next > Finish Cancel

Figure 10-7 Defining the database connection parameters

- Click *Finish*. The connection is established and the tables are retrieved.

Creating the database query

Next we identify the record(s) to retrieve from that database:

- The connection we created is now selected.
- The tables in the database are listed (Figure 10-8).

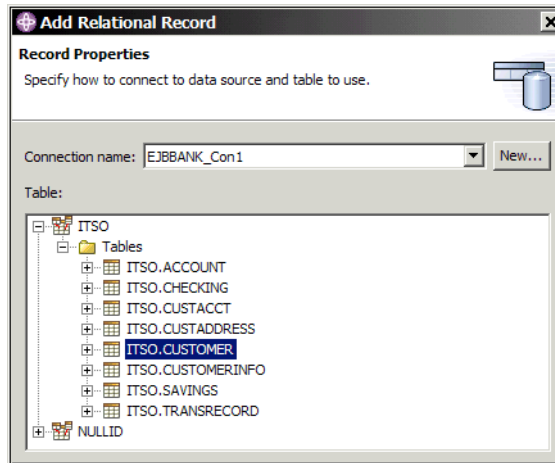


Figure 10-8 Database table listing

- ▶ Select the ITSO.CUSTOMER table. Click *Next*.
- ▶ Select only the FIRSTNAME and LASTNAME columns.
- ▶ Click *Filter results* to setup the WHERE condition (Figure 10-9):
 - Delete the generated condition that uses a parameter (select the line and click the  icon).
 - Add a condition by clicking the  icon. Select the CUSTOMERID column and for the value click the  icon to select the session scope variable customerID. Click *OK*.
 - Click *Close* in the Filters dialog.

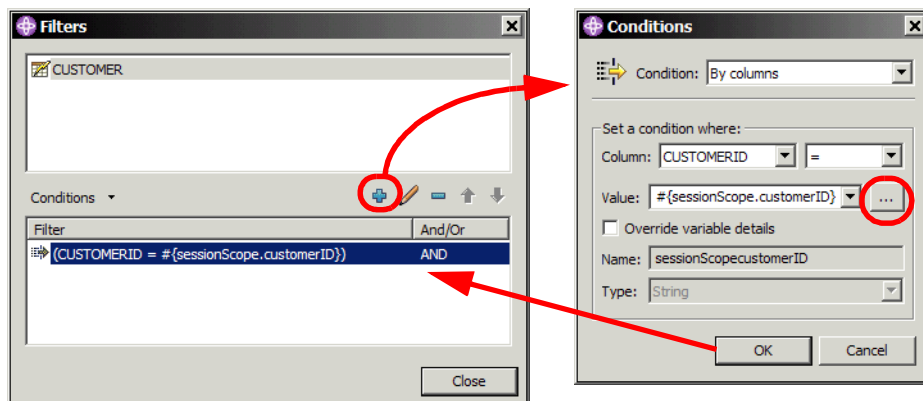


Figure 10-9 Filtering the contents of the retrieved table

- ▶ Click *Finish* and the `customerName` object with three properties appears in the Page Data view.
- ▶ Save the `index.jsp`.

Generated code

Open the `Index.java` file (in the `pagecode` package):

- ▶ Notice the connection name and the parameters:

```
private static final String customerName_connectionName= "EJBBANK_Con1";
protected static final String[] customerNameArgNames =
    { "sessionScopecustomerID" };
protected static final String[] customerNameArgValues =
    { "#{sessionScope.customerID}" };
```

- ▶ Notice the mediator code:

```
protected JDBCMediatorAccessBean getCustomerNameMediator() {
    if (customerNameMediator == null) {
        try {
            customerNameMediator =
                new JDBCMediatorAccessBeanImpl(
                    getRealPath(customerName_metadataFileName),
                    customerName_connectionName);
        } catch (Throwable e) {
            logException(e);
        }
    }
    return customerNameMediator;
}
```

- ▶ Notice the data access bean:

```
public DataObjectAccessBean getCustomerName() {
    if (customerName == null) {
        try {
            resolveParams(
                getCustomerNameMediator().getParams(),
                customerNameArgNames,
                customerNameArgValues,
                "customerName_params_cache");
            DataGraphAccessBean graph =
                getCustomerNameMediator().fetchGraph();
            customerName = graph.getDataObjectAccessBean();
        } catch (Throwable e) {
            logException(e);
        }
    }
    return customerName;
}
```

- ▶ Close the file.
- ▶ In WEB-INF you find a **wdo** folder with the `customerName.xml` file. The definition of the relational record or list is stored in this XML file.
- ▶ In the Server perspective you can find the **.wdo-connections** file in the `ItsoJSFSD0` project. The system keeps track of connections in this file. When testing the page a data source for each connection is automatically added to the server configuration. Go back to the Web perspective.

Action logic

We invoke the `getCustomerName` method when clicking *Submit* to check if the customer exists. This function logs the login attempt in the Console and tests for the existence of the requested customer:

- ▶ Select the Submit button in the Design view, and in the Quick Edit view enter the action logic (`\labscode\dev-sdobank\indexSubmit.txt`):

```
log("Customer login");
String lastname = (String)getCustomerName().get("LASTNAME");
log("lastname="+lastname);
if (lastname == null) {
    getFacesContext().addMessage(null,
        new FacesMessage("Customer not found: "
            +getSessionScope().get("customerID")));
    return "error";
} else {
    return "success";
}
```

- ▶ Save the `index.jsp` and resolve the error message `FacesMessage` cannot be resolved.
 - Select the Task view item and *Quick Fix* (context).
 - Select Import 'javax.faces.application.FacesMessage'.
 - Click *OK*.
 - Save the `Index.java` class.

Test the home page

We can now test the home page:

- ▶ Start or restart the JSFServer. Notice in the Console that a `jdbc/EJBBANK_Con1` data source is allocated. You can find a new JDBC provider (`WDO_DB2 JDBC Provider`) and this data source in the server configuration.
- ▶ Select the `index.jsp` and *Run on Server*. Set the server as the default server. Test with different customer IDs.

Implementing the error page

The error page is basically the same as for the JSF banking application, only now we take advantage of the FacesMessage queue in the application.

- ▶ Open the **error.jsp**.
- ▶ Add a heading 2, ITS0 Bank Error Message.
- ▶ Add an Output component from the Palette. Change the label to:
An error has occurred: <=== with a blank at the end
- ▶ Add a Display Errors (not Display Error) component from the Palette. In the Attributes view on the All tab:
 - Change the layout attribute to table so that multiple error message are displayed in a nice format.
 - Change the showDetail attribute to **true**.
- ▶ Add a Back button. Define a **home** navigation rule to return to the index.jsp and enter the action logic: return "home";
- ▶ Save and close the error.jsp.

Implementing the list of accounts

Next we implement the list of accounts. We create a relational record for the customer information, and we use a relational data list to retrieve the accounts of the customer.

Open the **listAccounts.jsp**:

- ▶ In the Page Data view select *New -> Relational Record*:
 - Enter **customerName** as name.
 - Select *Reuse metadata definition from an existing record*.
 - Click *Browse* and select the customerName.xml file.
 - Select *Fill record with existing data from the database*.
 - The connection and table are preselected.
 - The filter is predefined (from the index.jsp).
 - Click *Finish*.
- ▶ Verify that only CUSTOMERID, FIRSTNAME, and LASTNAME properties are in the data object in the Page Data view.
- ▶ Add an Output - Formatted Text component as a heading 2, then change the value to **Accounts of: {0} {1}**
 - On the Parameters page define two parameters, first and last. Set the values to customerName.FIRSTNAME and customerName.LASTNAME. Use the icons to select the variables.

Note that you can also drag/drop the FIRSTNAME and LASTNAME properties onto the output component, then add the Accounts of: text.

- Save the listAccounts.jsp for now, but do not close it.

Retrieving the customer's accounts using a join

To retrieve the accounts of the customer we have to build a data object (relational record list) with a join between the CUSTACCT and ACCOUNT tables.

- In the Page Data view select *New -> Relational Record List*.
 - Enter **customerAccounts** as name. Click *Next*.
 - Use the same connection and select the ITS0.CUSTACCT table. Click *Next*.
 - Select *Add another database table through a relationship*.
 - The From table is defined as ITS0.CUSTACCT. Select *Create an existing relationship from the database*. There are two possible relationships; to ACCOUNT and to CUSTOMER.
 - Select the ITS0.CUSTACCT->ITS0.ACCOUNT relationship (Figure 10-10). Click *Next*.

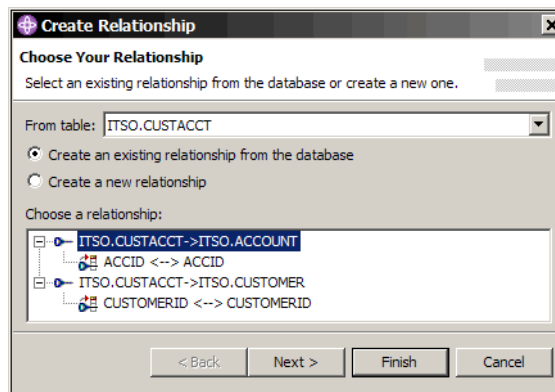


Figure 10-10 Selecting an existing relationship from the database

- The relationship information is established (Figure 10-11).

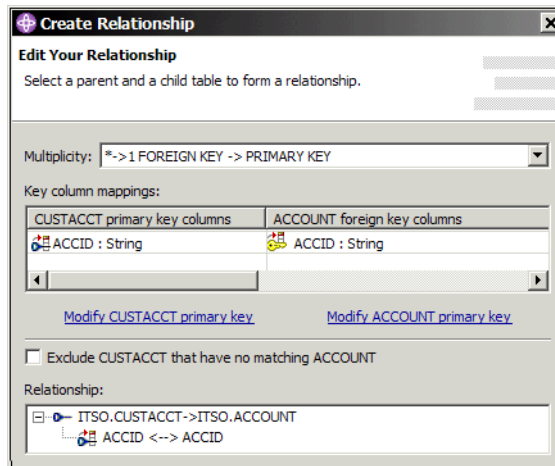


Figure 10-11 Configuring the table relationship

- Click *Finish* and the relationship is added to the table.
- Expand the relationship and the ACCOUNT table to select the BALANCE and ACCTYPE columns (Figure 10-12).

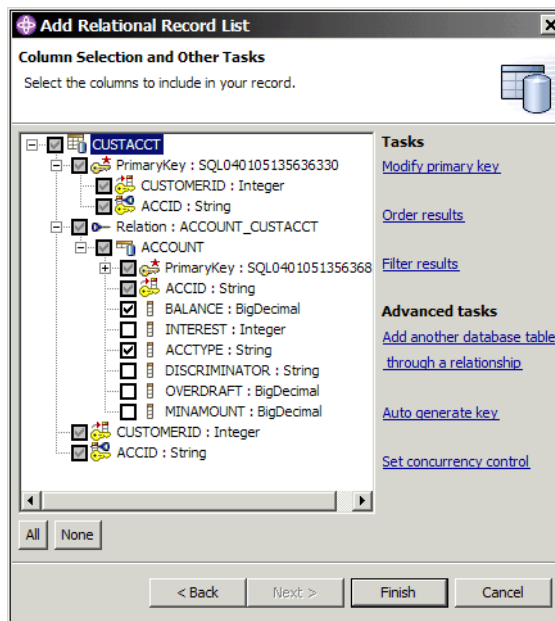


Figure 10-12 Selecting the desired columns to extract

- Click *Filter results*. Select the CUSTACCT table and click  to create a condition. Select the CUSTOMERID column and bind the value to the session variable customerID (Figure 10-13). Click *Close*.

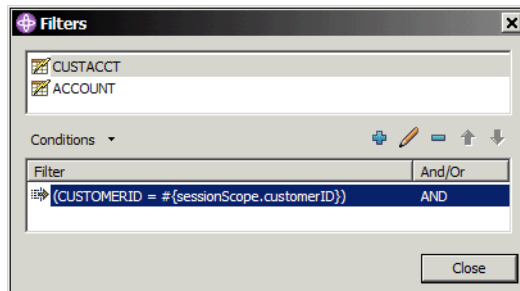


Figure 10-13 Filtering the accounts for one customer

- Click *Finish* and the customerAccounts record list is added to the Page Data view. Expand the data object to see its properties (Figure 10-14).

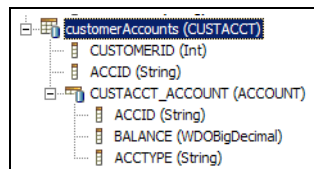


Figure 10-14 Page Data view with relational record list

Adding the account list to the page

Next we add the account listing (relational record list) to the JSF page:

- Select the CUSTACCT_ACCOUNT (ACCOUNT) property and drop it into the accountList.jsp. You are prompted with the Insert Record List Dialog (Figure 10-15):
 - Select *Multi-column Data Table (one table row per entry)*.
 - Move the BALANCE field to the end of the list.
 - Change the labels to Account, Type, and Balance.

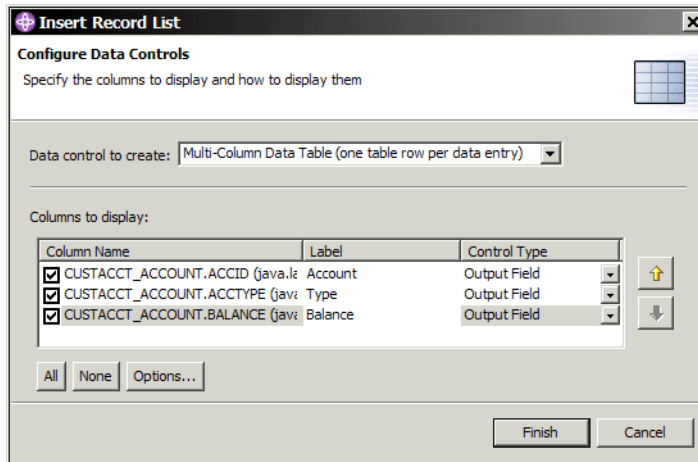


Figure 10-15 Inserting a record into a JSF page

- Click *Finish* and a table of the accounts is inserted (Figure 10-16).

Account ^{abc}	Type ^{abc}	Balance ^{abc}
{ACCID} ^{abc}	{ACCTYPE} ^{abc}	{BALANCE} ¹²³

Figure 10-16 Design view of the relational record list

A relational record list is displayed as a data table. We could use all the facilities provided by the data table and the IBM extensions (row action support, paging, and so forth). Because the list of accounts is usually a short list we do not add any paging support.

- Complete the design (Figure 10-17):
 - Select the `<h:dataTable>` and in the Attributes view, All tab, set border 1.
 - Add a prompt (Output component) above the table, with the text Select the account you want to work with:.
 - Add a Display Errors component under the table.
 - Add a Command - Button labelled Back at the bottom.

Redbooks

Accounts Of: {0} {1}

Select the account you want to work with:

Account	Type	Balance
{ACCID}	{ACCTYPE}	{BALANCE}

{Error Messages}

Back

Figure 10-17 Design view of listAccounts.jsp

- ▶ Create an empty JSF page named **accountDetails.jsp**. Close it.
- ▶ Select the Back button and define two navigation rules:
 - Local: **home** to index.jsp
 - Local: **details** to accountDetails.jsp

In the Quick Edit view enter the action logic to clean up the session data and return to the home page:

```
getSessionScope().remove("customerID");
return "home";
```

- ▶ Select a Command - Hyperlink component and drop it onto the {ACCID} field in the first column (you should release the mouse button when the cursor directly precedes the {ACCID} text):
 - Select the hyperlink and in the Attributes view, Parameters tab, define a parameter:
 - Name: selectedAccount
 - Value: #{varcustomerAccounts.ACCID}
 - (use the icon to locate the customerAccounts.ACCID property)
 - Select the hyperlink and in the Quick Edit view enter the logic (\labscode\dev-sdobank\listAccountsLink.txt):

```
log("Account selection");
String accountNumber =
    (String)getRequestParam().get("selectedAccount");
if (accountNumber == null) return "home";
getSessionScope().put("accountID",accountNumber);
return "details";
```


We pass the selected account to the next step.

- Save the `accountList.jsp`. If you get a warning, go to the Source tab. Sometimes dropping a hyperlink generates an extra `<h:outputText>` inside the existing `<h:outputText>`. Delete the extra tag:

```
<h:commandLink styleClass="commandLink" id="link1"
    action="#{sc_ListAccounts.jsp.doLink1Action}">
    <h:outputText id="text2"
        value="#{varcustomerAccounts.CUSTACCT_ACCOUNT.ACCID}"
        styleClass="outputText">
        <h:outputText id="text8" styleClass="outputText"
            value="link label"></h:outputText>
    </h:outputText>
    <f:param id="param3" name="selectedAccount"
        value="#{varcustomerAccounts.ACCID}"></f:param>
</h:commandLink>
```

Test the list accounts page

Restart the `ItsoJSF10SD0` project in the JSFServer, then run the `index.jsp`.

Implementing the account details page

The next step is the account details page where we display the account information. We can also execute account operations, such as deposit, withdraw, transfer, and list transactions, from this page.

Creating the Page Data

First we define all of the necessary Page Data objects:

- ▶ Open the `accountDetails.jsp`. On the Page Data view, define three request scope variables: **action**, **amount**, **destinationAccount** (`java.lang.String`).
- ▶ For account operations we require several data objects:
 - `account`—currently selected account
 - `fromAccount`—source account for transfer
 - `destAccount`—destination account for transfer
 - `trans`—transaction record to be inserted (deposit, withdraw, transfer)
 - `accountNumbers`—list of possible destination accounts
- ▶ Create an **account** data object to display the selected account:
 - Select *New -> Relational Record*.
 - Enter account as name and select *Fill record*.
 - Use the same connection and select the `ITS0.ACCOUNT` table.
 - Select the `BALANCE`, `INTEREST`, and `ACCTYPE` columns.

- Click *Filter results*. Update the generated condition (use the  icon):
ACCID = #{sessionScope.accountID}
- Click *Finish*.
- ▶ Create a **fromAccount** data object to update the selected account for deposit, withdraw, and transfer operations:
 - Select *New -> Relational Record*.
 - Enter fromAccount as name and select *Fill record*.
 - Use the same connection and select the ITS0.ACCOUNT table.
 - Select the BALANCE and ACCTYPE columns.
 - Click *Filter results*. Update the generated condition:
ACCID = #{sessionScope.accountID}
 - Click *Finish*.
- ▶ Create a **destAccount** data object to update the destination account for transfer operations:
 - Select *New -> Relational Record*.
 - Enter destAccount as name and select *Fill record*.
 - Use the same connection and select the ITS0.ACCOUNT table.
 - Select the BALANCE and ACCTYPE columns.
 - Click *Filter results*. Update the generated condition:
ACCID = #{requestScope.destinationAccount}
 - Click *Finish*.
- ▶ Create a **trans** data object for inserting transaction records:
 - Select *New -> Relational Record*.
 - Enter trans as name and select *Create an empty record in order to create a new row in the database*.
 - Use the same connection and select the ITS0.TRANSRECORD table.
 - Select all the columns.
 - Click *Finish*.
- ▶ Create an **accountNumbers** data list to display the account numbers for a transfer operation in a drop-down list:
 - Select *New -> Relational Record List*.
 - Enter accountNumbers as name and select *Reuse metadata definition*. Locate the customerAccounts.xml file. Click *OK* and *Finish*.
- ▶ The Page Data view is now complete (Figure 10-18).

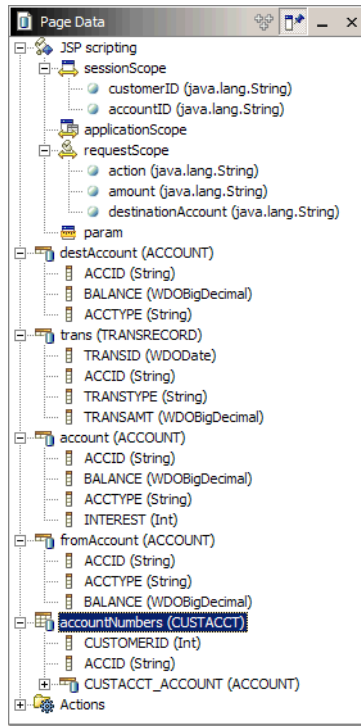


Figure 10-18 Page Data view for accountDetails.jsp

Designing the account details page

Edit the `accountDetails.jsp` and complete the page as shown in Figure 10-19:

- ▶ Create a heading 2 with: Account : and an Output component. Bind the output component to the session scope variable `accountID`.
- ▶ Drop the account data object into the JSF page. Select *Display an existing record*. Deselect the `ACCID` column. Change the labels to Balance, Interest, and Type. Move `ACCTYPE` to the top. Click *Options* and select *Append a colon to each label*. Click *Finish*.
 - Select the generated table and set the border to 1 and the padding to 5.
 - Select the middle empty column (approach the table from the top until a down arrow appears - click). When the column is selected press `Delete`.
- ▶ Add an output component before the generated {Error Messages} field with the text: Select the transaction you would like to perform:.

Redbooks

Account: {accountID} ^{abc}

Type:	{ACCTYPE} ^{abc}
Balance:	{BALANCE} ¹²³
Interest:	{INTEREST} ¹²³

BR
BR

Select the transaction you would like to perform: ^{abc} *{Error Messages}

Figure 10-19 Designing the accountDetails.jsp

- Create a table with one row and two columns (Figure 10-20):

☐ List Transactions ☐ Deposit ☐ Withdraw ☐ Transfer	<table border="1"> <tr> <td>Amount:</td> <td>{amount} ^{abc} *{Error Message for amountInput}</td> </tr> <tr> <td>Destination account:</td> <td>{selectItems.pc_AccountDetails.accountNumbers.ACCID.ACCID.toArray} ▼</td> </tr> </table>	Amount:	{amount} ^{abc} *{Error Message for amountInput}	Destination account:	{selectItems.pc_AccountDetails.accountNumbers.ACCID.ACCID.toArray} ▼
Amount:	{amount} ^{abc} *{Error Message for amountInput}				
Destination account:	{selectItems.pc_AccountDetails.accountNumbers.ACCID.ACCID.toArray} ▼				

Figure 10-20 Designing the actions table

- In the first column place a Radio Button Group component; *vertical*, with names List Transactions, Deposit, Withdraw, and Transfer and values listTransactions, deposit, withdraw, and transfer.
- Bind the component to `#{requestScope.action}`.
- In the second column place a table with two rows and two columns. Put Output components in the first column, with values Amount: and Destination account:. Put an Input component and a Combo Box component in the second column.
- Name the input field amountInput and create an error message field (using the Validation tab). Bind the component to `#{requestScope.amount}`.
- Name the combo box destinationAccountInput. Bind the component to `#{requestScope.destinationAccount}`.
- On the Choices tab, click *Add Computed Items*. This adds an `<items>` line. Bind the value to the `accountNumbers.ACCID` property (the list of account numbers).
- Add Submit and Back buttons at the bottom (Figure 10-21).

Figure 10-21 Design view of the accountDetails.jsp

- ▶ Create two empty JSF pages named **listTransactions.jsp** and **transferResult.jsp**. Close the files.
- ▶ Select the Submit button and enter navigation rules:
 - **perform** to accountDetails.jsp
 - **transfer** to transferResult.jsp
 - **listTransactions** to listTransactions.jsp
 - **back** to listAccounts.jsp
- ▶ Select the Back button and enter the logic to remove the account from the session and return to the list of accounts:


```
getSessionScope().remove("accountID");
return "back";
```
- ▶ Select the Submit button and enter the logic for all the banking operations (Example 10-1, use \labscode\dev-sdobank\accountDetailsSubmit.txt):

Example 10-1 Account details logic

```
log("Account operation");
String source = (String)getSessionScope().get("accountID");
String action = (String)getRequestScope().get("action");
String amount = (String)getRequestScope().get("amount");
String destination = (String)getRequestScope().get("destinationAccount");
```

```

log("Action: "+action+" Amount: "+amount+ " Destination: "+destination);
if (action == null) {
    FacesMessage errmsg =
        new FacesMessage(FacesMessage.SEVERITY_ERROR,
            "Select an action.", "Select an action.");
    getFacesContext().addMessage(null, errmsg);
    return "perform";
}
if (action.equals("listTransactions")) {
    return action;
}
if (amount.equals("")) {
    FacesMessage errmsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
        "Amount required.", "Amount is required.");
    getFacesContext().addMessage("amountInput", errmsg);
    return "perform";
}
try {
    BigDecimal bal = (BigDecimal)getAccount().get("BALANCE");
    BigDecimal amt = new BigDecimal(amount);
    log("Balance old: "+bal+" Amount: "+amt);

    if (action.equals("deposit")) {
        bal = bal.add(amt);
        log("Balance new: "+bal);
        getFromAccount().put("BALANCE", bal); // update the balance
        // update the account
        getFromAccountMediator().applyChanges
            ( getFromAccount().getDataGraphAccessBean() );
        createTransaction(source, "C", amt); // create a transaction rec
    }
    else if (action.equals("withdraw")) {
        bal = bal.subtract(amt);
        log("Balance new: "+bal);
        getFromAccount().put("BALANCE", bal); // update the balance
        // update the account
        getFromAccountMediator().applyChanges
            ( getFromAccount().getDataGraphAccessBean() );
        createTransaction(source, "D", amt); // create a transaction rec
    }
    else if (action.equals("transfer")) {
        if (destination.equals("")) {
            FacesMessage errmsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
                "Destination account required.",
                "Destination account is required.");
            getFacesContext().addMessage("destinationInput", errmsg);
            return "perform";
        }
        // update the destination account
    }
}

```

```

        BigDecimal bal2 = (BigDecimal)getDestAccount().get("BALANCE");
        log("Balance dest old: "+bal2);
        bal2 = bal2.add(amt);
        log("Balance dest new: "+bal2);
        getDestAccount().put("BALANCE", bal2); // update -to- balance
        // update the from account
        bal = bal.subtract(amt);
        log("Balance new: "+bal);
        getFromAccount().put("BALANCE", bal); // update -from- balance
        // apply the changes to the accounts
        getDestAccountMediator().applyChanges
            ( getDestAccount().getDataGraphAccessBean() );
        getFromAccountMediator().applyChanges
            ( getFromAccount().getDataGraphAccessBean() );
        // create transaction records -to- and -from-
        createTransaction(destination,"C",amt);
        createTransaction(source,"D",amt);
        // save the destination account for the transferResult page
        getRequestScope().put("destinationAccount",destination);
        log("Transfer result destAccount="+destination);
    }
    else {
        FacesMessage errmsg =
            new FacesMessage(FacesMessage.SEVERITY_ERROR,
                "Invalid action: "+action,"Invalid action: "+action);
        getFacesContext().addMessage(null,errmsg);
        return "perform";
    }

    // single transaction commit -later-
    // log("end of operation, commit.");
    // getConnectionWrapper().getConnection().commit();
    // getConnectionWrapper().getConnection().close();

    if (action.equals("transfer"))
        return "transfer"; // display transfer result page
    else {
        getAccount().put("BALANCE", bal); // to display correct balance
        return "perform"; // stay on the same page
    }
} catch (Exception e) {
    log("Exception: "+e.getMessage());
    FacesMessage errmsg = new FacesMessage("Exception: "+e.getMessage());
    getFacesContext().addMessage(null,errmsg);
    return "error";
}

```

- Save the code with the errors.

- ▶ Study the logic:
 - Depending on the action, the required input fields are verified.
 - For deposit and withdraw, only one account is involved and updated.
 - For transfer, two accounts are involved and are updated.
 - For each account update, a transaction record is created in the createTransaction method (to be implemented).
- ▶ Open the AccountDetails.java class:
 - Select *Source* -> *Organize Imports*. Select java.math.BigDecimal when prompted.
 - Add the createTransaction method that inserts a row into TransRecord (\labscode\dev-sdobank\accountDetailsCreateTrans.txt):

```
private void createTransaction
(String accid, String type, BigDecimal amount) throws Exception {
    // create a transaction record and fill the properties
    getTrans().put("TRANSID", new java.sql.Timestamp
        ( System.currentTimeMillis() ));
    log("create transrecord: "+getTrans().get("TRANSID"));
    getTrans().put("ACCID", accid);
    getTrans().put("TRANSTYPE", type);
    getTrans().put("TRANSAMT", amount);
    // apply the changes
    getTransMediator().applyChanges
        ( getTrans().getDataGraphAccessBean() );
    // reset the variable for the next insert
    trans = null;
}
```

- Save and close the file. No errors should remain.

Test the account details page

Restart the ItsoJSF10SD0 project in the JSFServer, then run the index.jsp. You can test deposit, withdraw, and transfer (without the result page).

- ▶ Notice the message in the Console:

```
[4/21/04 12:58:18:201 PDT] 79302313 SharedPool I J2CA0086W: Shareable
connection MCWrapper id 7c08e31a Managed connection
com.ibm.ws.rsadapter.spi.WSRdbManagedConnectionImpl@4601e31a
State:STATE_TRAN_WRAPPER_INUSE from resource jdbc/EJBBANK_Con1 was
used within a local transaction containment boundary.
```

We are performing multiple updates and by default every update is committed. Later we implement a single transaction to commit all updates together.

Implementing the list transactions page

The list transactions page lists the transaction records of one account. We create one data object (relational record list).

Open the `listTransactions.jsp`:

- ▶ In the Page Data view select *New -> Relational Record List*.
 - Enter `listTransactions` as name.
 - Select the `ITS0.TRANSRECORD` table.
 - Select all columns.
 - Click *Filter results*. Add a condition:
`ACCID = #{sessionScope.accountID}`
 - Click *Order results*. Select the `TRANSID` column and click `>`.
 - Click *Finish*.
- ▶ Set up a heading 2: List Transactions of Account: `{accountID}` using a text and an Output component, which is bound to `sessionScope.accountID`.
- ▶ Drop the `listTransactions` data object into the JSP:
 - Deselect the `ACCID` column (it has always the same value).
 - Change the labels to Timestamp, Credit/Debit, and Amount.
 - Change the table border to 1 and cell padding to 3 (Attributes view All tab).
 - Select the `{TRANSID}` output text and change the format to *Date and Time* (Attributes view Format tab).
- ▶ Add a Back button.
- ▶ Add a navigation rule: **back** to `accountDetails.jsp`, and enter the action code:
`return "back";`
- ▶ Figure 10-22 shows the completed design.

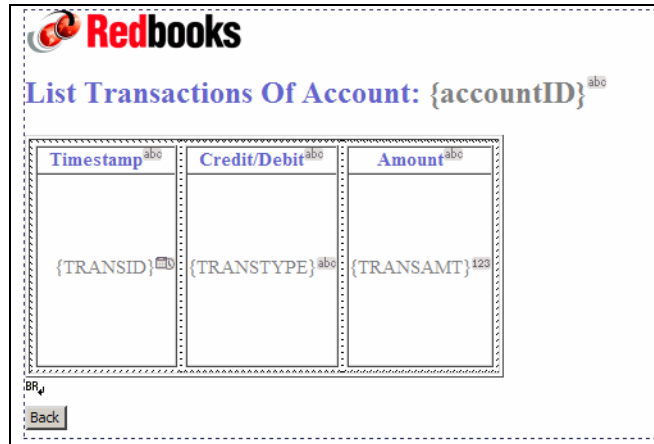


Figure 10-22 Design view of the listTransactions.jsp

Test the list transactions page

Restart the ItsoJSF10SD0 project in the JSFServer, then run the index.jsp and progress to list the transaction records for an account.

Implementing the transfer result page

The transfer result page displays the two accounts involved in the transfer.

Open the **transferResult.jsp**:

- ▶ In the Page Data view define a request scope variable named **destinationAccount** (java.lang.String). This value is passed from the accountDetails page.
- ▶ Select *New -> Relational Record*. Enter **account** as name. Select *Reuse metadata definition*. Click *Browse* and select the account.xml file. Click *Finish*. Verify that only ACCTYPE, BALANCE and INTEREST properties are in the Page Data view.
- ▶ Repeat this sequence and define the **destAccount** record by using the destAccount.xml file. Select columns BALANCE, and ACCTYPE. Make sure the filter condition is still correct.
- ▶ Create a heading 2: Transfer Result.
- ▶ Insert a table with two rows and two columns. In the first row enter each a heading 3: **From Account** and **To Account**.
- ▶ Drop the account record into the first cell of the second row. Deselect the INTEREST column. Display the fields with labels ID:, Type:, Balance:.

- ▶ Drop the destAccount record into the second cell of the second row. Display the fields with labels ID:, Type:, Balance:.
- ▶ Delete the generated error message fields.
- ▶ Delete the middle columns from the two account tables.
- ▶ Add a Back button.
- ▶ Add navigation rule **back** to accountDetails.jsp and the action code:


```
return "back";
```
- ▶ The complete design is shown in Figure 10-23.

Figure 10-23 Design view of the transferResult.jsp

Test the transfer result page

Restart the ItsoJSF10SD0 project in the JSFServer, then run the index.jsp and perform a transfer.

Implementing single transaction commit

The account operations perform multiple updates. We have to run all updates in one transaction. This is currently quite complicated and will get better in a future version.

Open the AccountDetails.java class:

- ▶ Look up one of the connection names:


```
private static final String account_connectionName = "EJBBANK_Con1";
```
- ▶ Create an instance variable for a JDBC connection:


```
import com.ibm.websphere.wdo.mediator.rdb.ConnectionWrapper;

private ConnectionWrapper conn;
```

- Create a new method `getConnectionWrapper` that creates a passive connection manager that does not automatically commit at each update (`\labscode\dev-sdobank\connectionWrapper.txt`):

```
private ConnectionWrapper getConnectionWrapper() {
    if ( conn == null ) {
        try {
            // conn = ConnectionManager.createConnectionWrapper
            //         (account_connectionName);
            conn = ConnectionManager.createPassiveConnectionWrapper
            //         (account_connectionName);
        } catch (MediatorException e) {
            e.printStackTrace();
        }
    }
    return conn;
}
```

Use *Source -> Organize Imports* to fix the errors. Select these classes:

```
import com.ibm.websphere.wdo.access.connections.ConnectionManager;
import com.ibm.websphere.wdo.mediator.exception.MediatorException;
```

- Find the `getFromAccountMediator`, `getDestAccountMediator`, and `getTransMediator` methods and change the code to call the `getConnectionManager` method:

```
protected JDBCMediatorAccessBean getFromAccountMediator() {
    if (fromAccountMediator == null) {
        try {
            fromAccountMediator =
                new JDBCMediatorAccessBeanImpl(
                    getRealPath(account_metadataFileName),
                    //fromAccount_connectionName);      <=====
                    getConnectionWrapper());           <=====
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
    return fromAccountMediator;
}
```

Important: do not change the `getAccountMediator` method.

- Activate single transaction commit in the Submit button action logic:

```
// single transaction commit -later-
log("end of operation, commit.");
getConnectionWrapper().getConnection().commit();
getConnectionWrapper().getConnection().close();
```

- Save the class and test the application with single transaction commit. There should be no warning message in the Console.

Implementing a transaction details page

Implement a hyperlink on the list transactions page so that you can select a transactions and display the details in a new details page.

Open the `listTransactions.jsp`:

- ▶ Drop a Command - Hyperlink onto the `{TRANSID}` output text.
- ▶ For the hyperlink define a parameter named `TRANSID` and bind it to the `listTransactions.transid` column.
- ▶ Add a navigation rule **details** to `transactionDetails.jsp` with the code `return "details";`

Create a **transactionDetails.jsp**:

- ▶ In the Page Data view create a relational record named **transDetail** for one transaction record (select the `ITS0.TRANSRECORD` table).
- ▶ Click *Filter results*. Edit the generated default condition, it reads:

`TRANSID = param.TRANSID`

This matches the parameter we defined for the hyperlink. However, we have to change the data type to `String`. Select *Override variable details* and change the type to `String` (Figure 10-24).

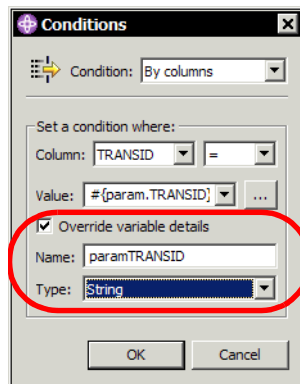


Figure 10-24 Filtering on the transaction ID

- ▶ Drop the `transDetail` into the JSP. Change the labels.
- ▶ Add a border to the table. Set the format of `TRANSID` to *Date and Time*.
- ▶ Add a Back button. Define a navigation rule (**back**) to the transaction list page.
- ▶ The complete design is shown in Figure 10-25.



Transaction Details

Timestamp:	{TRANSID} ¹²³
Account:	{ACCID} ^{abc}
Credit/Debit:	{TRANSTYPE} ^{abc}
Amount:	{TRANSAMT} ^{\$23}

Figure 10-25 Design view of the transactoinDetails.jsp

Test the transaction details page

Restart the ItsoJSF10SD0 project in the JSFServer, then run the index.jsp.

Implementing a table maintenance application

In this section we write small Web pages to maintain customer and account information. We illustrate the use of the data table row edit and row select facilities.

Customer maintenance using row edit support

We can modify the customer table in the database using one JSF page. We add a relational record list of the customers as a data table, then create an edit column that can be used to change an individual record.

Create and open a **customerUpdate.jsp**:

- ▶ Create an allCustomers relational record list, using the ITS0.CUSTOMER table. Do not apply any filters.
- ▶ Drop the allCustomers relational record list into the JSF page (Figure 10-26). Deselect the columns USERID, PASSWORD, and ADDRESS. Clean up the column headings.



Customer Maintenance

Customer ID ^{abc}	Title ^{abc}	First Name ^{abc}	Last Name ^{abc}
{CUSTOMERID} ^{abc}	{TITLE} ^{abc}	{FIRSTNAME} ^{abc}	{LASTNAME} ^{abc}

Figure 10-26 Output only controls for the customer list

- ▶ In the Attributes view of the data table, click *Add row edit support*.
- ▶ In the Page Data view, select TITLE, FIRSTNAME, LASTNAME and drop them into the new row edit column as input fields (Figure 10-27).



Customer Maintenance

	Customer ID ^{abc}	Title ^{abc}	First Name ^{abc}	Last Name ^{abc}
Title: {TITLE} ^{abc}	{CUSTOMERID} ^{abc}	{TITLE} ^{abc}	{FIRSTNAME} ^{abc}	{LASTNAME} ^{abc}
First name: {FIRSTNAME} ^{abc}				
Last name: {LASTNAME} ^{abc}				

Figure 10-27 Customer record list with a new edit column

To make the updates using row edit support, we have to commit the relational record list when we submit row edit changes.

- ▶ Select the TITLE input field, then in the Attributes view use the pull-down to select the `hx:commandExRowEdit` element. This is the command to save the update. In the Quick Edit view enter this code to commit the update of the selected row (`\labscode\dev-sdobank\accountDetailsCreateTrans.txt`):

```
log("row edit action");
try {
    getAllCustomersMediator().applyChanges(
        getAllCustomers().getDataGraphAccessBean() );
}
```

```

    } catch (Exception e) {
        e.printStackTrace();
    }
    return "";
}

```

- Save the JSP and run it on server (Figure 10-28).

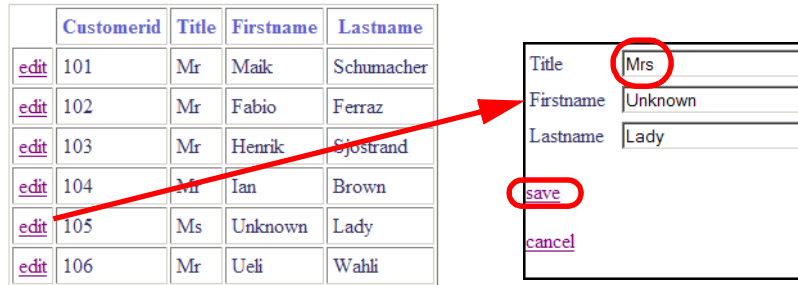


Figure 10-28 Updating a customer record

- Click *Edit*, then update the row data, and click *Save*. The update is committed on the server.
- You should see the customer record updated in the reloaded table.

Note: An alternative implementation can be achieved. When dropping the `allCustomers` object into the JSF page, select input fields for `TITLE`, `FIRSTNAME`, `LASTNAME`. Then add an `Update` command button where the same action code is executed. This action updates all changed rows at the same time. (See `customerUpdate2.jsp` in the sample code.)

Account maintenance using row select support

We can create a similar JSF page to select accounts for increasing the interest rate. We add a relational record list of the accounts as a data table, then create a row selection column that can be used to select multiple accounts.

Create and open an **accountSelection.jsp**:

- Create an `allAccounts` relational record list, using the `ITS0.ACCOUNT` table. Only select `BALANCE`, `ACCTYPE`, and `INTEREST` columns. Do not apply any filters.
- Drop the `allAccounts` relational record list into the JSF page. Clean up the column headings. Set the table border to 1.
- In the Attributes view, click *Add row select support*. This adds a column of check boxes.
- Add a `Display Errors` component and a `Submit` button (Figure 10-29).

Account Selection For Update

	Account ^{abc}	Balance ^{abc}	Type ^{abc}	Interest ^{abc}
☐	{ACCID} ^{abc}	{BALANCE} ¹²³	{ACCTYPE} ^{abc}	{INTEREST} ^{abc}

% {Error Messages} BRv

Figure 10-29 Account selection using row select support

Selected rows are passed in a managed bean to the action logic of the Submit button. The row selection column can be bound to a Boolean array or an Integer array. A Boolean array must be big enough to hold all the marks (selected and unselected). An Integer array only holds the indexes of selected rows.

We create a managed bean holding an Integer array:

- Create a JavaBean named `itso.jsf.data.SelectedRows` with this code:

```
package itso.jsf.data;
public class SelectedRows {
    private Integer[] rows = new Integer[0];
    public Integer[] getRows() { return rows; }
    public void setRows(Integer[] integers) { rows = integers; }
}
```

- Add the `SelectedRows` bean to the Page Data view using the name `selectedRows` without using a scope.
- Bind the `rows` property of `selectedRows` to the check box in the row select support column. Selected rows will be added to the Integer array.

The final step is the action logic of the Submit button:

```
log("Selected accounts:");
Integer[] rows = getSelectedRows().getRows();
for (int i=0; i<rows.length; i++) {
    int row = rows[i].intValue();           // selected row
    // get the account data object from the list
    DataObjectAccessBean account =
        getAllAccounts().getDataObjectAccessBean(row);
    String accountId = (String)account.get("ACCID");
    Integer interest = (Integer)account.get("INTEREST");
    log("Account="+accountId+" Interest="+interest);
    // increase the interest rate and store in the account
    interest = new Integer(interest.intValue()+1);
    account.put("INTEREST", interest);
}
```

```

getSelectedRows().setRows( new Integer[0] );
// commit the changes
try {
    getAllAccountsMediator().applyChanges(
        getAllAccounts().getDataGraphAccessBean());
} catch (Exception e) {
    log("Exception: "+e.getMessage());
    FacesMessage errmsg = new FacesMessage("Exception: "+e.getMessage());
    getFacesContext().addMessage(null,errmsg);
}
return "";

```

Run the accountSelection.jsp on the server. Select a few rows and watch the increase of the interest rate (Figure 10-30).

	Account	Balance	Type	Interest
☐	101-1001	80	CHECKING	4
☒	101-1002	375.26	SAVINGS	5
☐	102-2001	9,375.26	SAVINGS	5
☒	102-2002	75.5	CHECKING	3
☐	106-6003	3,000	SAVINGS	6

Submit

	Account	Balance	Type	Interest
☐	101-1001	80	CHECKING	4
☐	101-1002	375.26	SAVINGS	6
☐	102-2001	9,375.26	SAVINGS	5
☐	102-2002	75.5	CHECKING	4
☐	106-6003	3,000	SAVINGS	6

Submit

Figure 10-30 Updating the interest rate of selected accounts

Account listing using row category support

To illustrate the row category support of the data table we list the accounts sorted by customer, with collapsed/expanded sections by customer.

Create and open an **accountsOfCustomer.jsp**:

- ▶ Create an accountsByCustomer relational record list, starting with the CUSTACCT table and following the relationship to the ACCOUNT table. Only select BALANCE, ACCTYPE, and INTEREST columns of the ACCOUNT table. Do not apply any filters.
- ▶ Drop the accountsByCustomer relational record list into the JSF page. Clean up the column headings. Set the table border to 1.
- ▶ In the Attributes view, click *Add row category support*. This adds a column with an arrow for expanding and collapsing (Figure 10-31).

	Customer ^{abc}	Account ^{abc}	Type ^{abc}	Balance ^{abc}	Interest ^{abc}
	{CUSTOMERID} ^{abc}	{ACCID} ^{abc}	{ACCTYPE} ^{abc}	{BALANCE} ¹²³	{INTEREST} ^{abc}

Figure 10-31 Account listing using row category support

- Select the category arrow and in the Attributes view bind the arrow to the CUSTOMERID column for sorting by customer. You can also set the initial state, collapsed (default) or expanded.

Run the JSF page and the account listing is displayed sorted by customer. Expand and collapse the sections (Figure 10-32).

Accounts Sorted By Customer

Customer	Account	Type	Balance	Interest
► 101				
▼ 102				
102	102-2001	SAVINGS	9,375.26	5
102	102-2002	CHECKING	75.5	4
► 103				
► 104				
► 105				
▼ 106				
106	106-6001	CHECKING	1,000	3
106	106-6002	SAVINGS	2,000	4
106	106-6003	SAVINGS	3,000	6
106	105-5001	CHECKING	0	2

Figure 10-32 Listing accounts by customer

Complete solution

We provide the complete solution for this application in the additional material in the file `\labscode\solution\ItsoJSF10SD0.ear`. See “Importing a solution enterprise application” on page 548 for instructions.



Part 4

JSF advanced topics

In Part 4 we introduce some of the advanced combinations for JSF Web applications as well as possible custom extensions.

Chapter 11 discusses JSF integration with portal technology and the development of a JSF portlet application.

Chapter 12 covers EGL integration with JSF Web applications.

Chapter 13 discusses the Faces Client Framework.

Chapter 14 shows you how to extend the JSF framework using WebSphere Studio Application Developer 5.1.2.



JSF and portlets

This chapter provides information about the JavaServer Faces support included with Portal Toolkit Version 5.0.2 and includes two examples of JSF portlet applications.

Refer to “Related publications” on page 551 for more information about portlets and the Portal Toolkit.

Concepts

In this section we introduce the basic concepts to understand the technology of portal servers and portlets:

- Portal** A portal is a Web-based application that provides personalization, single sign-on, and content aggregation from different sources. A portal hosts the presentation layer of information systems. Aggregation is the process of integrating content from different sources within a Web page. A portal may have sophisticated personalization features to provide customized content to users. Portal pages may have different sets of portlets creating content for different users.
- Portlet** Portlets are Java-based Web components that process requests and generate dynamic content. A portlet's content normally aggregates with the content of other portlets to form the portal page. In the simplest terms, a portlet is a Java servlet that operates inside a portal.

The relationship between those terms is that, functionally, a portal server serves the portlets to the user. The portlets can be grouped in portlets applications (Figure 11-1). In addition, a portal server provides common services such as application connectivity, integration, administration, and presentation that is required across all portlet applications.

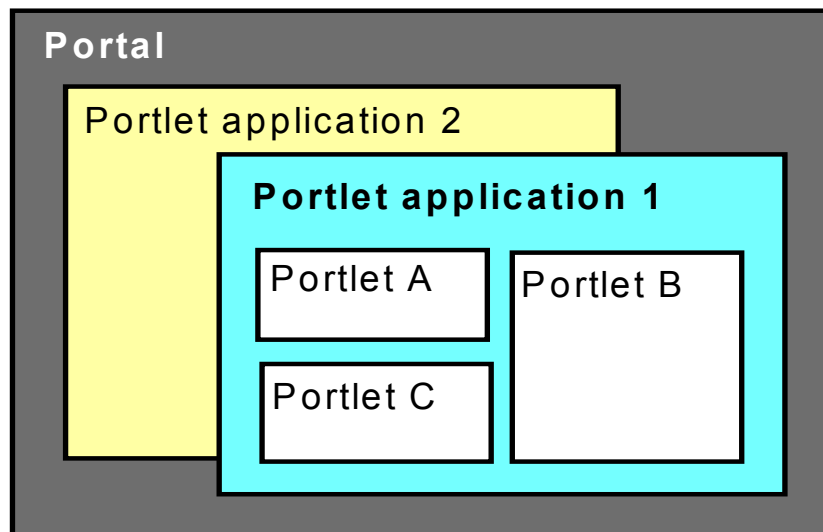


Figure 11-1 Portal and portlet relationship

Portal Toolkit

The WebSphere Portal Toolkit is set of visual portlet development tools and a WebSphere Portal unit test environment, which lets developers build and test portlets just as they would any other type of Web project. New portlets are created using wizards which generate a portlet project structure that conforms to both J2EE and also a portlet specification. The wizards can create portlets that comply with the IBM Portlet API or JSR 168, the industry standard portlet specification.

Developers can lay out the interface for portlets using Page Designer, and create portlets using the Struts framework and the Web diagram editor to visualize the structure and event flows. In addition, developers can combine portlets with JavaServer Faces to develop their portlets visually, using the Faces components. WebSphere Studio also automates the creation of the deployment descriptors and EAR files.

The version we used for this redbook is WebSphere Portal Toolkit 5.0.2.2.

Portlet API

The Portal Toolkit supports the development of portlets using either the IBM portlet API or the JSR 168 portlet API. This section provides you with some information to decide which API to use when you develop portlets. WebSphere Portal Server 5.0 supports both types of portlets:

- ▶ **IBM portlet API**—The IBM portlet API was initially supported for WebSphere Portal V4.x and is supported in subsequent versions.
- ▶ **JSR 168**—JSR 168 is a specification from the Java Community Process that addresses the requirements of aggregation, personalization, presentation, and security for portlets running in a portal environment. The purpose of the JSR 168 specification is to solve the problem of portlet compatibility between portal servers offered by different vendors.

Deciding which API to use in your applications

As the IBM portlet API continues to be supported in future versions of WebSphere Portal, you do not have to migrate existing portlets, unless a portlet is required to interoperate with other portals that support JSR 168.

For new portlets, consider using JSR 168 when the functionality it provides is sufficient for the portlet's requirements.

Future directions for WebSphere Portal JSR 168 support

IBM provides further enhancements to the JSR in follow-on versions to make the JSR 168 portlet API as useful as the current IBM counterpart. IBM is committed to wider adoption of open standards in WebSphere Portal, including JSR 168, JSP Standard Tag Library (JSTL), WSRP, Struts and JavaServer Faces.

JavaServer Faces support

You can develop portlet applications using JSF in much the same way as you develop Web applications. The Faces support in Portal Toolkit 5.0.2.2 simplifies the process of writing Faces portlet applications and eliminates the need to manage many of the underlying requirements of portlet applications.

WebSphere Portal Toolkit 5.0.2.2 provides a set of wizards that help you create Faces portal-related artifacts. In many cases, these wizards are identical to the wizards used to create standard Faces artifacts.

Navigation

The navigation model is exactly the same model as we have in JSF Web applications. This model is a set of rules written in the `faces-config.xml` file, located in the `WebContent\WEB-INF` subfolder of the portlet application.

Actions of command buttons and hyperlinks

Actions in JSF pages are managed the same way as in other Web applications. The code for the action is added to the page code Java class and the code is executed when you click the button or the hyperlink.

Portlet modes and JSF

In JSF portlets, the following modes are available:

View	The view mode must be implemented. The expected functionality in this mode is to generate markup language (HTML in a Web application) to show the state of the portlet.
Edit	The edit mode should provide content and logic to let the user customize the behavior of the portlet for that user.
Help	When a portlet is in this mode, it should provide help information about itself.
Configuration	The configuration mode displays one or more configuration views that let administrators configure portlet settings that are valid for all users.

When developing a JSF Web application you usually implement the View mode and the Help mode.

Note: Each portlet mode has a first page defined that is shown when the portlet switches to that mode. If you navigate or link to that page directly, this does not mean that the portlet changes its mode.

The portlet mode should be changed only with the buttons available for this operation. You should take care to avoid navigation that could lead into an incorrect application state.

JSP page model

When you create a JSP page inside a portlet development project, you have four possible choices. These choices are shown in the *New Faces JSP file wizard*. These choices refers to the desired page structure. The following models are available:

- ▶ **Basic**—When using this model, a complete HTML document structure is created in the JSP page, with all the HTML, HEAD and BODY tags.
- ▶ **Portlet**—This model creates an HTML fragment. The HTML, HEAD and BODY tags are not used in the new JSP file. Instead of these tags, a special `<portletAPI: init>` tag is included in the page to tell the portal server that the page is an HTML fragment.
- ▶ **Basic with client side data caching**—This model is the same as Basic model, but JSF Client support is added to the page.
- ▶ **Portlet with client side data caching**—In this model, the JSF Client support is added to the HTML fragment that is created.

Known problems

There are two known problems with WebSphere Portal Toolkit 5.0.2.2 and JavaServer Faces portlet development:

- ▶ The JSF standard validation is not working currently The way to implement validation is to write your own validation into the action code of the command components and, in case of errors, navigate to an error page to inform the user about the validation errors.
- ▶ JSF development using portlet projects based on JSR 168 is not working currently. Until this problem is fixed, you can develop portlet projects using the IBM Portlet API.

Unsupported components

The Faces support is not complete. There are components that are currently not supported:

- ▶ File Upload component (hx:fileupload)
- ▶ Components that support the download of binary data (although they are available in a non-portal Faces environment):
 - Image (hx:graphicImageEx), when bound to data, as in:

```
<hx:graphicImageEx value="#{myBean.photo}"></hx:graphicImageEx>
```
 - Link (hx:outputLinkEx), when bound to data, as in

```
<hx:outputLinkEx value="#{myBean.resume}"></hx:outputLinkEx>
```
 - Media Player (hx:playerGenericPlayer, hx:playerFlash, hx:playerMediaPlayer, hx:playerRealPlayer, hx:playerShockwave), when bound to data, as in:

```
<hx:playerGenericPlayer  
  value="#{myBean.movie}"></hx:playerGenericPlayer>
```
- ▶ When adding an Image component to a Faces portlet page, you must specify the URL relative to the document WebContent root folder, rather than relative to the project root folder:

Wrong:

```
<hx:graphicImageEx  
  value="/.YourPortlet/theme/yourimage.gif"></hx:graphicImageEx>
```

Correct:

```
<hx:graphicImageEx value="theme/yourimage.gif"></hx:graphicImageEx>
```

SDO support

Support for SDO in WebSphere Portal is limited to prototyping.

Because SDO is not fully supported for WebSphere Portal 5.0.x, applications that rely on SDO components should not be used in a production environment. However, the current tooling does supply this capability for prototyping purposes. When you attempt to add an SDO component (a relational record or a relational record list) to a Faces JSP file, a warning about this limitation is supplied.

Refer to Chapter 9, “SDO concepts” on page 299 for more information about SDO.

Examples

In the sections that follow, we provide two examples of using JSF inside a portlet project. The first example is a simple application built from scratch. In the second example we migrate an entire JSF Web application into a portlet project.

Before doing these examples, you must install the Portal Toolkit 5.0.2.2 to get the portal development framework into WebSphere Studio. Refer to “WebSphere Portal Toolkit” on page 531 for installation instructions.

Example 1: JSF calculator in a portlet

In this example we explain how to develop a simple JSF portlet application. We use the JSF calculator example, which was covered in Chapter 3, “JSF calculator example” on page 61.

The goal of this example is to have a simple JSF portlet application running in a portlet using the WebSphere Portal Test Environment.

Create the Portlet project

We first create a Portlet Development project and then we implement the first portlet. Follow these steps:

- Create a project and select *Portlet Development* -> *Portlet Project*. Do not select *Portlet Project (JSR 168)*. Click *Next* (Figure 11-2).

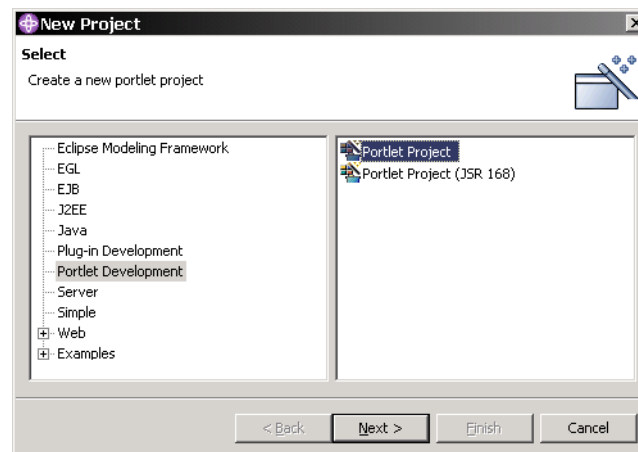


Figure 11-2 Creating a Portlet project

- Name the project **ItsoJSFPortletCalc**. Select *Faces portlet* as the type of portlet, check *Configure advanced options* and click *Next* (Figure 11-3).

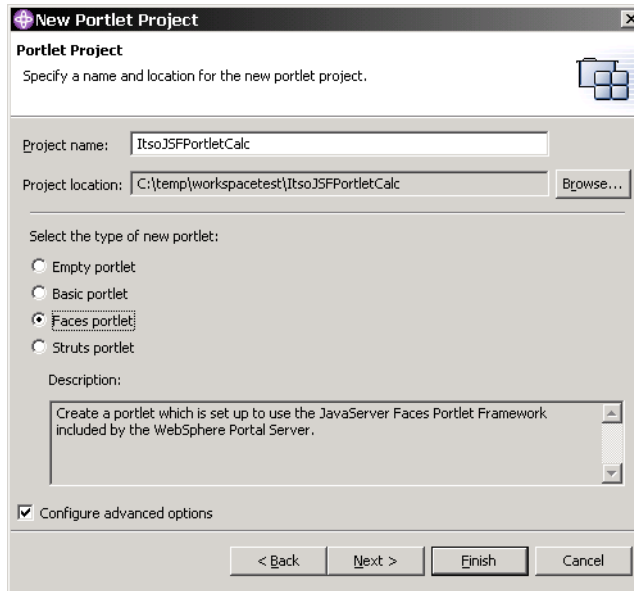


Figure 11-3 Selecting faces support for the portlet project

- On the next page, enter the EAR project name as **ItsoJSF11Portlet**. The *J2EE level 1.3 / WebSphere Portal 5.0 support* should be selected for you. If not, select it. Click *Next* (Figure 11-4).

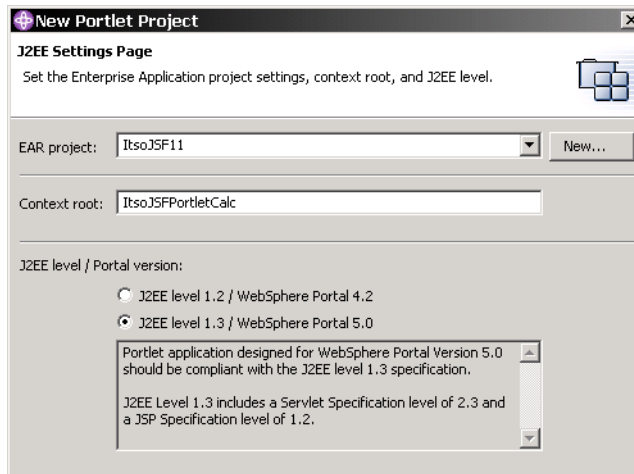


Figure 11-4 Selecting the J2EE level for the project

- ▶ Click *Next* in the Features page.
- ▶ Click *Next* in the Portlet settings page.
- ▶ In the Miscellaneous page, select only the *View Mode* for the portlet. Enter **calculate.jsp** as the name of the initial page (Figure 11-5).

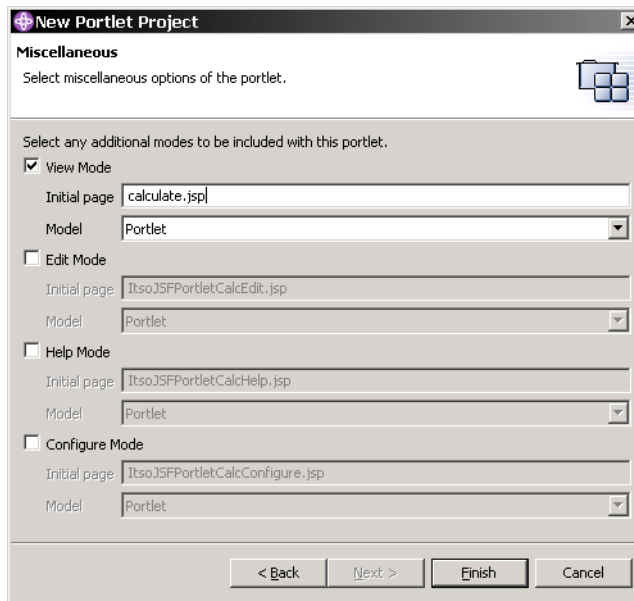


Figure 11-5 Selection of the portlet modes

Once the project is created, Application Developer prompts to change to the Portlet perspective. Click *OK*. The *Portlet Deployment Descriptor* and the JSF page **calculate.jsp** are opened for you.

Import the CalculatorBean

We have to import the `CalculatorBean` class to use it in the project:

- ▶ Select the `Java Resources` folder in the `ItsoJSFPortletCalc` project.
- ▶ Create a new Java package called **itso.jsf.calculator**.
- ▶ Select the new package and click *File* -> *Import*. In the pop-up window select *File System* and click *Next*.
- ▶ Select the folder `\labscode\dev-portlet` and then select the file `CalculatorBean.java`. Click *Finish*.

Design of the calculator JSP

To create the application we perform steps similar to “Development steps” on page 62:

- ▶ Open the file `calculate.jsp`. Replace the default text with a heading 2 paragraph and enter **JSF Portlet Calculator Example**. Enter **Created using WebSphere Studio Application Developer** in a normal paragraph.
- ▶ Insert a table with 5 rows and 2 columns. Insert the following components:
 - Drop Output components into column 1, cells 1, 2, 3 and 5.
 - Drop Input components into column 2, cells 1 and 3.
 - Drop a Combo Box component into column 2 cell 2.
 - Drop a Command - Button component into column 2 cell 4.
 - Drop an Output component into column 2 cell 5.
- ▶ Change the components using the Attributes view:
 - Row 1: enter **Number 1:** as the value for the Output component. For the Input component, change its width to **12**. Using the Format tab select the *Number* type, *Decimal* style and *Integer Only*.
 - Row 2, column 1: enter **Operation:** in the value field for the Output component.
 - Row 3: enter **Number 2:** in the value field of the Output component. For the Input component, change its width to **12**. Using the Format tab select the *Number* type, *Decimal* style and *Integer Only*.
 - Row 5, column 1: enter **Result:** in the value field for the Output Component. Set style properties to **color: blue; font-weight: bold**.
- ▶ Select the Combo Box component and in the Attributes view:
 - Set the id to **operation**.
 - Add four text items for the choices: Set their names to **add**, **subtract**, **multiply** and **divide**. Change the generated values to **Add**, **Subtract**, **Multiply** and **Divide**.
- ▶ Select the Submit button and set **calc** as the id and **Calculate** as its label.
- ▶ Select the Output component in row 5, column 2. Set **result** as its ID and set the style properties to **color: blue; font-weight: bold**.
- ▶ Save the page. The design is shown in Figure 11-6.

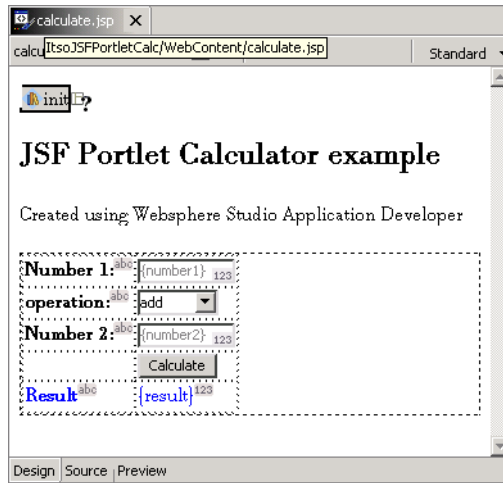


Figure 11-6 JSF Portlet calculator example design

Bind the frontend to the calculator bean

Now we bind the UI components in the `calculate.jsp` to the `CalculatorBean` to use its operations:

- ▶ Select the `calculate.jsp` page. Go to the Page Data view. In the context menu select *New -> JavaBean*. In the Add JavaBean dialog:
 - Enter **calculator** as name.
 - For the class click *Browse* icon and find the `CalculatorBean` class. Click *OK*.
 - Select *Make this JavaBean reusable* and select *session* for the scope.
 - Click *Initialize Properties*. In the dialog click *Add*. Select the **operation** property in the name column and enter **Subtract** in the value column. Then press Enter.
 - Click *Finish* and the bean is added to the Page Data view.
- ▶ Select the first input field (number1). In the Attributes view, for the value field, click the  icon and select the `number1` property of the `calculator` bean. Notice the generated binding:


```
#{pc_Calculate.calculator.number1}
```
- ▶ Repeat the operation for the `number2` input field and bind it to the `number2` property.

- ▶ Select the operation combo box. In the Page Data view, expand the CalculatorBean, select the *operation* property and *Bind to 'operation'* (Figure 11-7).

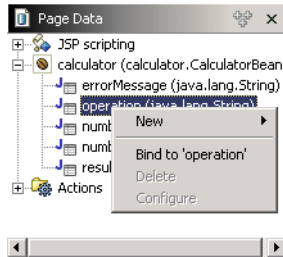


Figure 11-7 Bind the operation property to the operation component

- ▶ Bind the result property of the CalculatorBean to the result component in the page.

Create an error page

Now we create a new error page for the calculator example:

- ▶ Select the WebContent folder and *New -> Faces JSP File* in the context menu.
- ▶ Enter **error.jsp** for the new page.
- ▶ In the Model combo box, *Portlet* should be preselected. If not, select *Portlet* for the model for the new page. Click *Finish* (Figure 11-8).

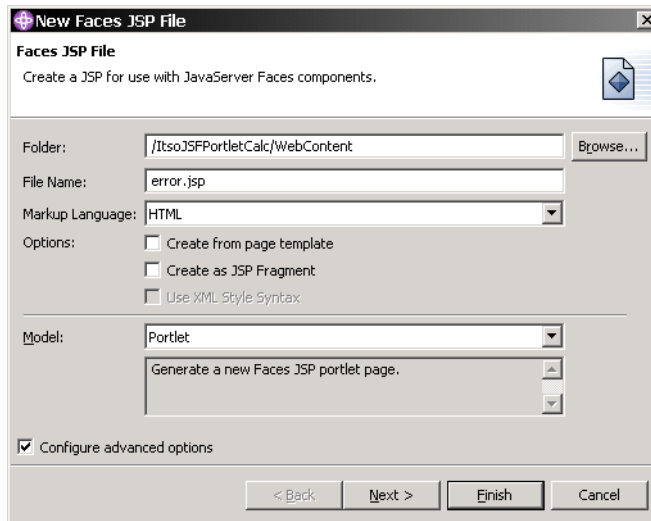


Figure 11-8 Create the Faces error page

Note: The Portlet model allows you to create the JSP file as an HTML fragment. This model is more correct because the page is sharing a Web page with other portlets and does not have to be a complete document.

The Basic model allows you to create the JSP file as a complete HTML document (with all the HTML, HEAD and BODY tags)

- ▶ Change the default text for a heading 2, **JSF Portlet Calculator Error Page**.
- ▶ In the Page Data view, select *New -> JavaBean*. In the dialog, select *Add existing reusable JavaBean*. Select the calculator bean and click *Finish*.
- ▶ Expand the calculator bean. Select the `errorMessage` property and drag/drop it under the heading. In the Insert JavaBean dialog, select *Displaying fields*, change the label to **Error message:** and click *Finish*.
- ▶ Remove the middle column from the table that is created.
- ▶ Add a Command - Button at the bottom. Change its label to **Back**.

Implementing the page navigation

To navigate between the two pages we have to declare the navigation in both pages:

- ▶ Select the page `calculate.jsp` and select the Calculate button. In the Attributes view define a new local page navigation:
 - Page: **error.jsp**.
 - Alias: **error**.
 - ActionRef: *****.
 - Click *OK*.
- ▶ Add another local navigation for this page:
 - Page: **calculate.jsp**.
 - Alias: **calculate**.
 - ActionRef: *****.
 - Click *OK*.
- ▶ Select the `error.jsp` and for the Back button define a new navigation rule:
 - Page: **calculate.jsp**.
 - Alias: **back**.
 - ActionRef: *****.
 - Click *OK*.

Invoking the calculate method

We have defined the binding of the user interface with the `CalculatorBean` class. We also have defined an error page and the navigations between the two pages.

We have to invoke the `calculate` method to perform the calculations:

- ▶ Select the Calculate button in the `calculate.jsp`.
- ▶ In the Quick Edit view, select *Command* in the left pane. Click in the right pane and replace the sample code with the code in `labcode\dev-portlet\CalculateButton.txt`:

```
log("CalculateAction start");
try {
    getCalculator().calculate();
} catch (Exception e) {
    log("Calculator-Exception: "+e.getMessage());
    if (getCalculator().getErrorMessage() == null)
        getCalculator().setErrorMessage("Exception: "+e.getMessage());
    return "error"; // go to the error page
} finally {
    log("CalculateAction end");
}
return "calculate"; // stay on the calculate page}
```

- ▶ To implement the navigation from the error page to the main page, select the `error.jsp`. Select the Back button and, using the Quick Edit view, include the following line for the Command event:

```
return "back";
```

Creating a portlet server for testing

To test the application we must define a server for publishing and testing of the portlet. To define this server follow these steps:

- ▶ Open the Server perspective. In the Server Configuration view (bottom left), select *New -> Server and Server Configuration*.
- ▶ Enter **JSFPortalServer** as the name of the server. Expand *WebSphere Portal 5.0* and select *Test Environment*. Click *Finish* (Figure 11-9).

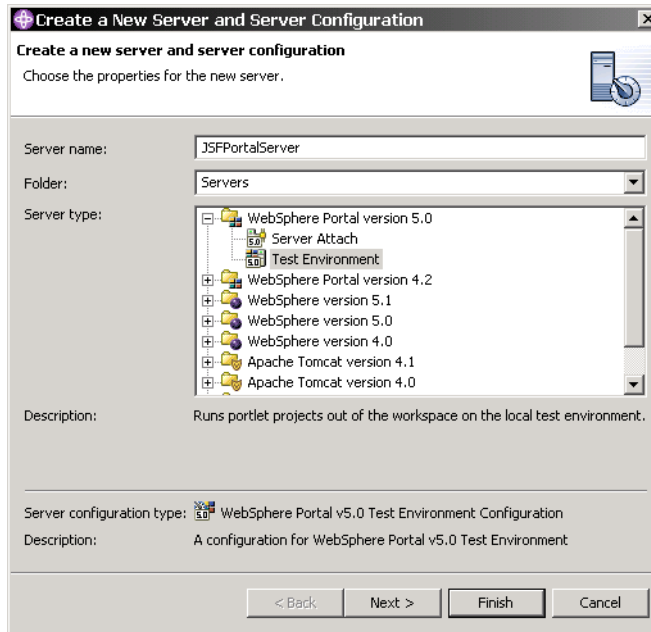


Figure 11-9 Portlet server definition

- The server now is created. In the Server Configuration view select the JSFPortalServer server and *Add Remove Projects* (context). Add the ItsoJSF11Portlet enterprise application. and click *Finish*.

Testing the calculator application

To test the portlet, follow these steps:

- Open the Portlet perspective if it is not opened. Select the ItsoJSFPortletCalc project and *Run on Server*.
- In the pop-up window the JSFPortletServer server is selected for you. Select *Set server as project default (do not prompt)* and click *Finish*.
- The server starts (this may take a few minutes) and the calculator application is loaded into a portlet (Figure 11-10).

Note: Because validation is not supported for JSF in portlets we did not implement validation in the calculator example.

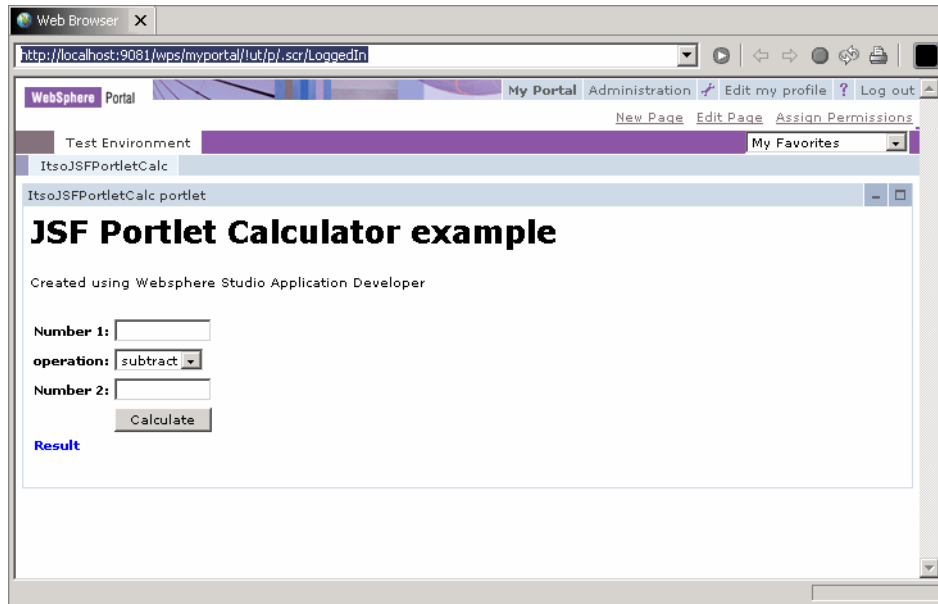


Figure 11-10 JSF calculator example running inside a portlet

Example 2: Migration of the JSF banking example

In this example we explain how to migrate an existing JSF application into a portlet. By doing this, we can leverage the existing code inside a Portal-based solution.

We have selected the application developed in Chapter 5, “JSF banking application” on page 191. This application is a simple application that uses backend beans to represent the bank data model. The data is stored in a database. The application uses JSF pages and components to build the user interface.

Creating and configuring a server

We require a server to test the applications. If you did not setup a portlet server yet, follow the instructions in “Creating a portlet server for testing” on page 394.

We have to configure the server with a data source for the banking application:

- In the Server Configuration view open the JSFPortletServer server configuration editor.

- Select the Security page. Click *Add* to add a new JAAS authentication entry. In the pop-up window, enter **DB2user** for the Alias field, and a valid DB2 user ID and password. Enter **DB2 user for portlet jsf applications** in the description field. Click *OK* (Figure 11-11).

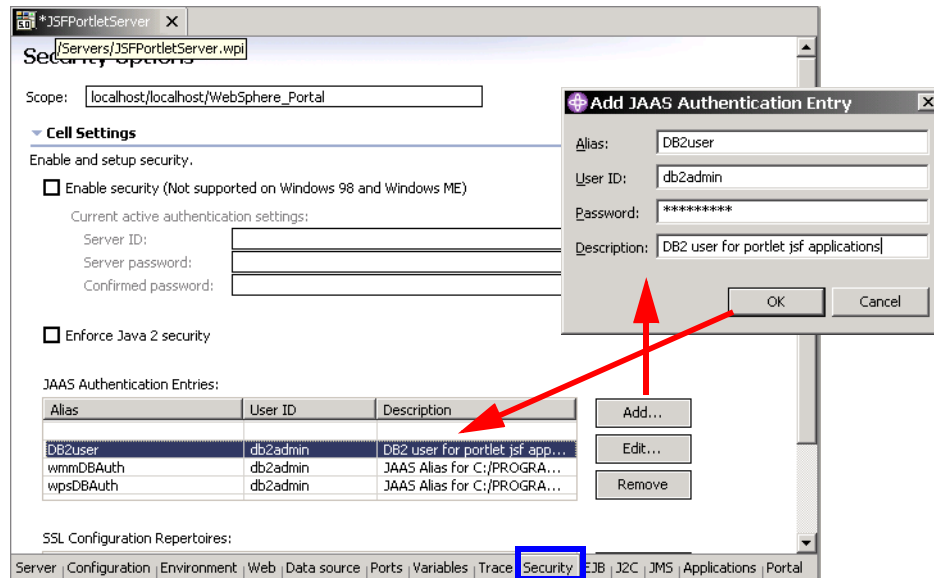


Figure 11-11 Defining a new JAAS authentication entry

- In the *Data Source* tab click *Add* in the JDBC Provider List. In the pop-up window select **IBM DB2** as the Database and **DB2 Legacy CLI-based Type 2 JDBC Driver** as JDBC provider type (Figure 11-12).

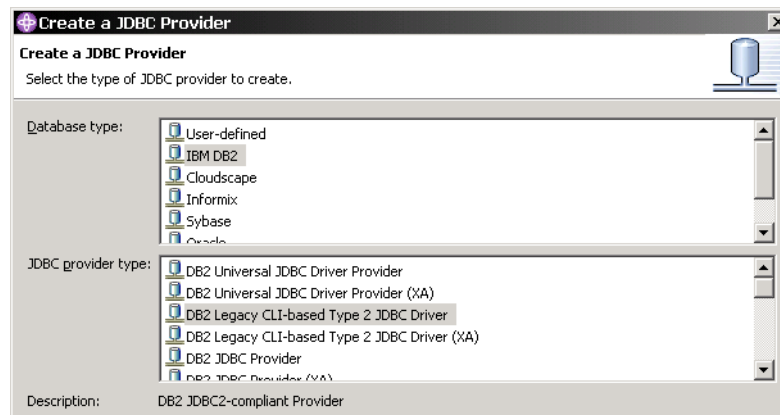


Figure 11-12 Defining a new JDBC database provider

- ▶ Click *Next* and enter **IBM DB2 Provider** in the Name field. Click *Finish*.
- ▶ In the Data Source page, select the *IBM DB2 Provider* in the JDBC provider list. In the Data source list click *Add* (Figure 11-13).

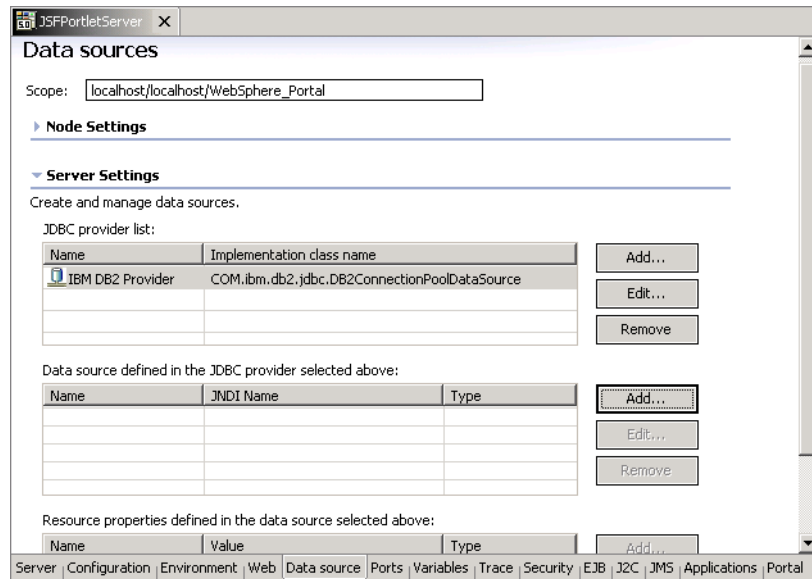


Figure 11-13 Adding a new data source for the JDBC provider

- ▶ In the pop-up window, select the *DB2 Legacy CLI-based Type 2 JDBC Driver and Version 5.0 data source*. Click *Next*.
- ▶ Enter the following values (Figure 11-14):
 - In the Name field, enter **EJBBANK**.
 - In the JNDI name field, enter **jdbc/ejbbank**.
 - Select *DB2user* in the Component-management authentication alias combo box and in the Container-management authentication alias combo box.
 - Deselect *Use this data source in container managed persistence (CMP)*.
- ▶ Click *Next*.
- ▶ In the Value field for the `databaseName` enter **EJBBANK**. Click *Finish*.
- ▶ Close and save the server configuration file.

Figure 11-14 Creating a data source

Now the server is configured to run the example.

Define a utility project

We have to define a Java utility project where the banking model is implemented. Create the **ItsoJSF11Portlet** enterprise application if you did not implement the calculator example.

The way to define this utility project is shown in “Define a utility project” on page 198:

- ▶ You only have to perform the first two steps: Create the **ItsoBankModel** project and import the **BankingModel.jar** file. If you implemented the JSF banking application, then you already have the **ItsoBankModel** project defined and you can skip these step.
- ▶ Open the EAR deployment descriptor of the **ItsoJSF11Portlet** project. On the Module page, under Project Utility JARs, click **Add** and select the **ItsoBankModel** project. This project will be available to all modules.

You may still have compilation errors. If this is the case, you have to enable server targeting for the enterprise application (**ItsoJSF11Portlet**) and select **WebSphere Application Server v5.1** as the target server.

Creating the project and the portlet

As in the previous example, we create a Portlet project with the first portlet configured:

- ▶ Create a new project and select *Portlet Development -> Portlet Project* (Figure 11-2 on page 387). Click *Next*.
- ▶ Name the project **ItsoJSFPortletBank**. Then select *Faces portlet* as the type of portlet, check *Configure advanced options* and click *Next* (Figure 11-3 on page 388).
- ▶ In the next page select *ItsoJSF11Portlet* as the EAR project and click *Next*. The *J2EE level 1.3 / WebSphere Portal 5.0 support* should be selected for you. Click *Next* (Figure 11-4 on page 388).
- ▶ Click *Finish*. The View mode page is created for the portlet and the name for the initial JSP is selected for you.

Importing the JSF banking application

Before importing the existing Web application we have to save a couple of files that should not be overwritten, because they are configuration files that are necessary for the project:

- ▶ Expand the WebContent folder and the WEB-INF subfolder. Copy and paste the **faces-config.xml** and **web.xml** files and rename the files as **faces-config.xml.portlet** and **web.xml.portlet** (Figure 11-15).

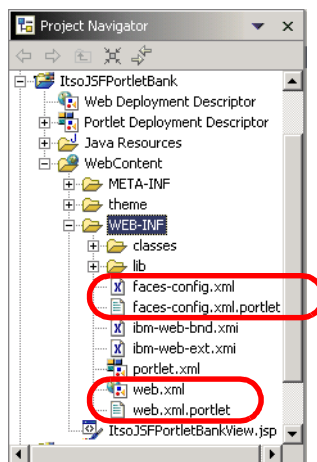


Figure 11-15 Copying the conflicting files

Next we import the JSF banking application into the portlet project. The objective is to have the entire JSF banking application running inside the portlet:

- ▶ Select the `ItsoJSFPortletBank` project and *Import*.
- ▶ Select *WAR File* and click *Next*.
- ▶ Select the file `\labscode\jsf-portlet\ItsoJSFBank.war` and click *Finish*.

The project shows compilation errors that we have to resolve:

- ▶ Open the file `faces-config.xml` file and the `faces-config.xml.portlet` file. We now merge these files by keeping the configuration for the Faces support of the portal application and adding the application configuration of the JSF banking application.
- ▶ Copy and paste all the managed beans, the navigation, and the validation configuration from `faces-config.xml` into `faces-config.xml.portlet` (Figure 11-16).

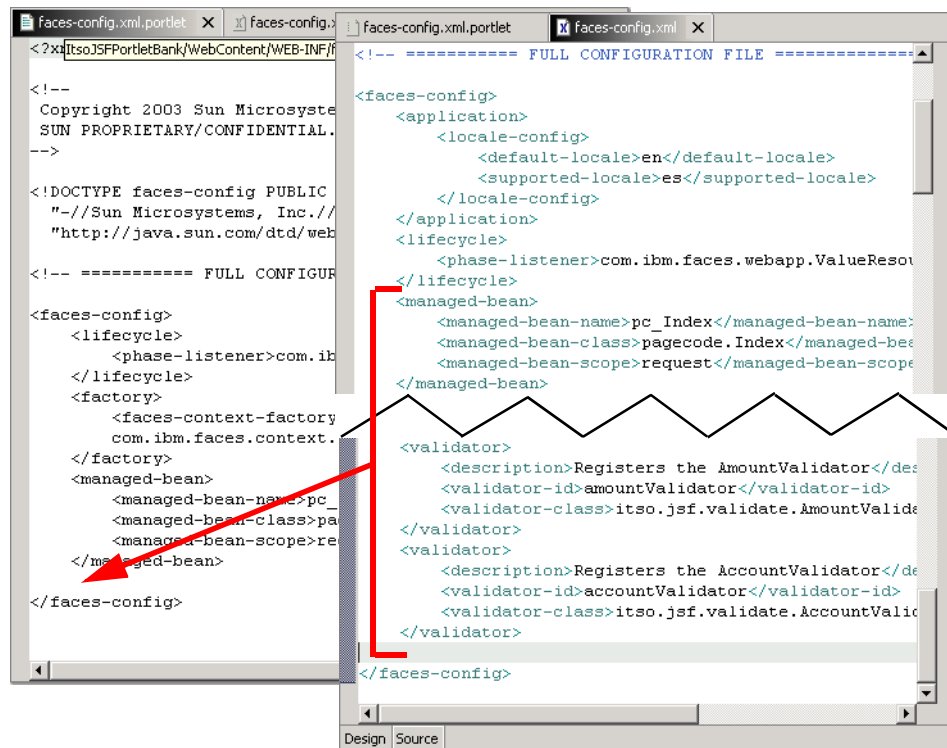


Figure 11-16 Copying the configuration file

- ▶ Save the files.

- ▶ Rename the file `faces-config.xml` to `faces-config.old`.
- ▶ Rename the file `faces-config.xml.portlet` to `faces-config.xml`.
- ▶ Compare the result file with `\labscode\jsf-portlet\faces-config.xml`. You can also import the file from there.
- ▶ Delete the imported `web.xml` file. Rename the file `web.xml.portlet` to `web.xml`. Select **Yes** when you are asked to fix the references.
- ▶ We have to define a database resource in the portlet project:
 - Open the file `web.xml`. On the references page, select the **Resource** tab.
 - Click **Add** and define the data source reference for the **EJBBANK** database. Enter **jdbc/mybank** as name. Locate the `javax.sql.DataSource` class and enter the JNDI name as **jdbc/ejbbank** (Figure 11-17).

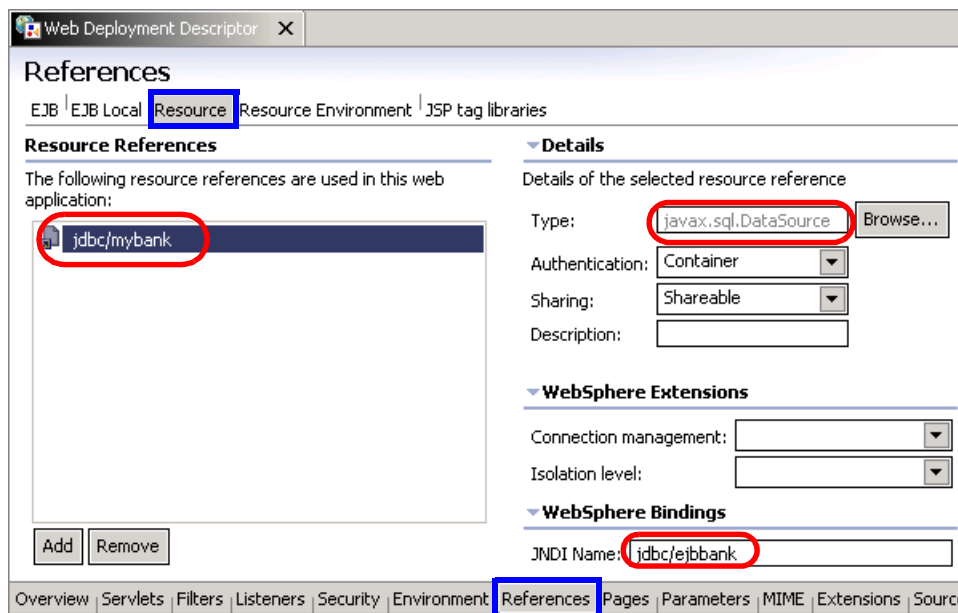


Figure 11-17 Defining the database resource in the deployment descriptor

- ▶ Open the `ItsoJSFPortletBank` project and verify the JAR Dependencies. Only the `ItsoBankModel` project should be selected. Also verify the Java Build Path and the `ItsoBankModel` project should be selected. If you make any changes, click **OK**.

The project should not have any compilation errors now.

Fixing the JSF pages

The JSF pages of the banking application are complete HTML documents with HTML, HEAD and BODY tags. In a portal environment each portlet is sharing the HTML page in the browser with other portlets, so it does not require all the tags.

To transform the JSP pages into correct portlet fragments perform these steps:

- ▶ Open each JSF page and delete the HTML, HEAD, TITLE, and BODY open and close tags.
- ▶ Include the portlet tag library:

```
<%@taglib uri="/WEB-INF/tld/portlet.tld" prefix="portletAPI"%>
```

- ▶ Insert the `<portletAPI:init />` tag before the `<f:view>` tag.

As an example we use the `index.jsp` file. The differences between the original file and the result after the transformation are shown in Figure 11-18.

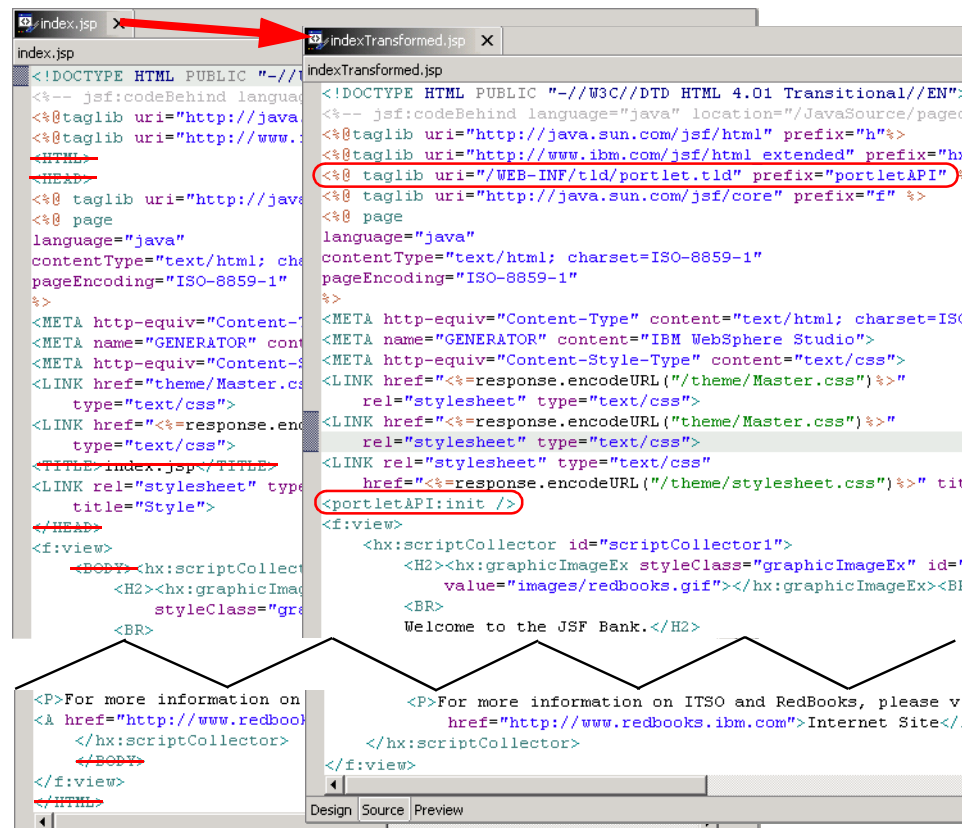


Figure 11-18 Transforming `index.jsp` document into a portlet fragment

Configuring the portlet

We have a Portlet project with one portlet and the complete JSF banking application imported in the project. We want the application to be shown inside the portlet.

One way to do this is to define the first Faces page (`index.jsp`) to be the page that is shown in the portlet view mode. We can do this by configuring the portlet:

- ▶ Under the `ItsoJSFPortletBank` project, open the *Portlet Deployment Descriptor* file.
- ▶ In the left panel, expand *Concrete Portlet Application* and select the portlet `itsojsfportletbank.ItsoJSFPortletBankPortlet`. The configuration information about this portlet appears in the right side.
- ▶ Scroll down and find *Settings Parameters*. Select the value for the parameter `com.ibm.faces.portlet.page.view` and replace the default page `/ItsoJSFPortletBankView.jsp` with the page `/index.jsp` (Figure 11-19).

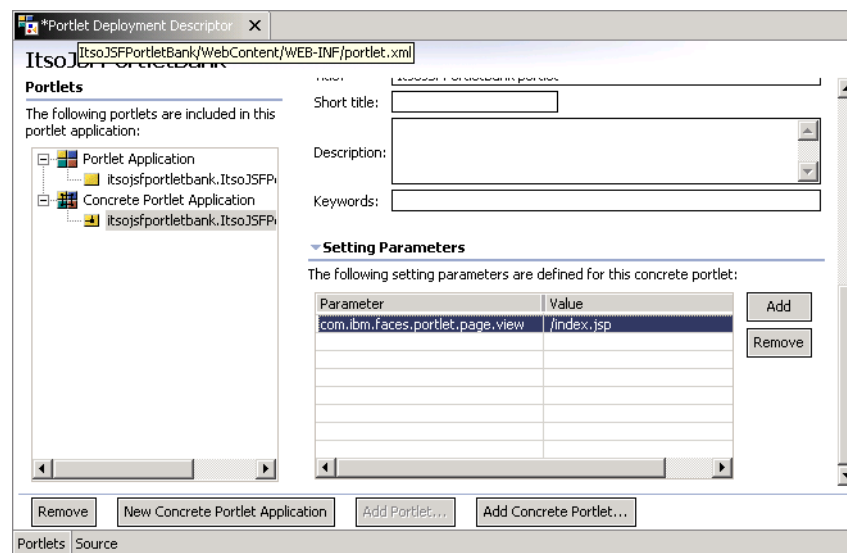


Figure 11-19 Changing the view page for the portlet

- ▶ Close and save the file.

Testing the portlet banking application

Now we can test the project in the WebSphere Portal Test Environment:

- ▶ Start the `JSFPortletServer`.

- ▶ The server may have to be republished. Click *OK* when publishing finishes. The server then starts, taking a few minutes.
- ▶ In the Portlet perspective select the `ItsoJSFPortletBank` project and *Run on Server*. The `JSFPortletServer` server is chosen. Select *Set server as project default (do not prompt)* and click *Finish*.

The Web browser opens and the JSF banking application is inside a portlet in a portal server. The name of this portlet is `ItsoJSFPortletBank`.

You can test the application using a valid user ID (for example, 101).

Note: Validation is not working in this version of WebSphere Portal Test Environment. When you enter a bad customer ID, the same page is displayed without any error message.

You may notice some differences between the portlet application and the original JSF banking application (Figure 5-31 on page 225). The main differences are:

- ▶ The images are not displayed well. As the application is running inside a portlet, we cannot use relative URLs.
- ▶ The style sheet is different. In this case the problem with the absolute URLs appears again.

We repair these problems in next section.

Fixing the JSF application

The image URLs must be encoded to be built correctly. This is done by using the function `encodeURL("<url as string>")` of the response object in the JSP pages:

- ▶ For example, you have to replace the expression:

```
old: <IMG src="images/redbooks.gif">
new: <IMG src="<%=response.encodeURL("images/redbooks.gif")%>">
```

There is a better way to display the images correctly. Use the JSF Image component to show all the images in the application.

- ▶ The URLs for the style sheet must be encoded as well. The solution is to include the style sheet using the same expression as for the images, for example:

```
old: <LINK href="theme/Master.css" rel="stylesheet" type="text/css">
new: <LINK href="<%=response.encodeURL("theme/Master.css")%>"
      rel="stylesheet" type="text/css">
```

To fix the project, follow these steps:

- Open the `index.jsp`.
- Delete the Redbooks image. Drag/drop an Image component from the palette in the place where the other image was. Using the Attributes view, select **images\redbooks.gif** as the File property or browse the resources using the browse button (Figure 11-20).

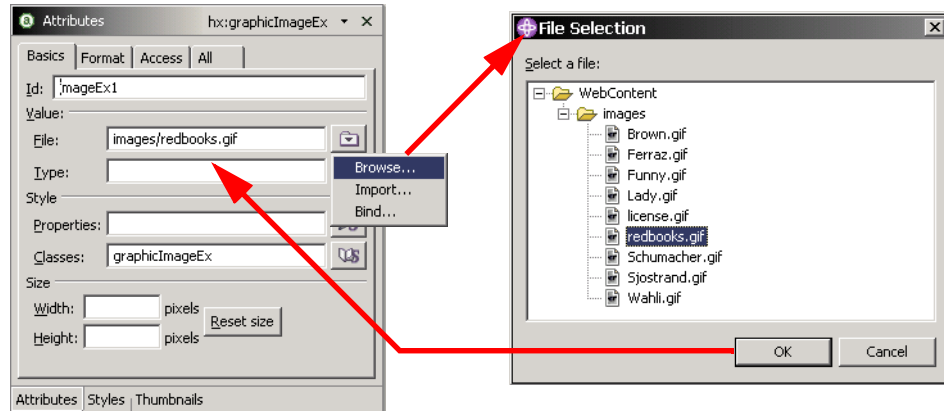


Figure 11-20 Setting the file for the Image component

- In the Source page, find the link to the Master.css style sheet. Duplicate the line. In the second line replace the URL with the `response.encodeURL()` function:

```
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<LINK href="<%=response.encodeURL("theme/Master.css")%>"
        rel="stylesheet" type="text/css">
```

The first reference resolves correctly in the page designer, and the second at runtime.

- Save and close the page.

Change all the other JSF pages in the same way (`accountDetails.jsp`, `error.jsp`, `listAccounts.jsp`, `listTransactions.jsp`, `transactionDetails.jsp` and `transferResult.jsp`).

Testing the complete application

To test the complete application restart the `JSFPortletServer` server. Go back to the Portlet perspective and rerun the `ItsoJSFPortletBank` project.

The Web browser appears and you have the complete JSF application running in the portal.

Complete solution

We provide the complete solution for the two applications in the additional material in the file `\labscode\solution\ItsoJSF11Portlet.ear`. See “Importing a solution enterprise application” on page 548 for instructions.



JSF and EGL

This chapter provides a short overview of how to implement and code a Web application in the JSF framework using Enterprise Generation Language (EGL) for the JSF data and event logic.

This chapter does not focus on the usage of EGL in general. EGL's capabilities range from advanced business logic and transaction handling to text-based user dialogs.

In this chapter we run all EGL functions on the development platform. It would also be possible to develop server-side EGL programs for the business logic and invoke those programs from the JSF EGL action logic. The business logic could be deployed on multiple platforms, including z/OS®. Note that COBOL generation for z/OS requires WebSphere Studio Enterprise Developer.

Note: Developing JSF application using EGL is only meant for developers that are familiar with EGL already. This chapter does not teach EGL concepts and the language.

Application Developer provides several examples for EGL. For example, to learn about basic operations, load the EGL Web Exercises project by selecting *File -> New -> Examples -> EGL -> EGL Web Exercises*.

EGL concepts

Enterprise Generation Language (EGL) is, as the name implies, a high-level programming language which generates code. During the generation process code in other programming languages is generated. The generation takes into account the target platforms capabilities as well as the supported programming language. Currently, EGL generates either Java or COBOL code, depending on the chosen target runtime platform.

Restriction: WebSphere Studio Application Developer does not support generation of any other code than Java. WebSphere Studio Enterprise Developer is capable of generating for other platforms, including COBOL for CICS® on z/OS.

Using a high-level abstraction language for development increases productivity dramatically. Having the same source target several native runtime environments allows for greater flexibility and performance.

At the same time a high-level language gives productivity to a wider range of developers, such as procedural VisualBasic developers or traditional COBOL developers.

In EGL, the server-side managed bean that handles the event and data for a JSF page is called a page handler.

JSF and EGL page handlers

In essence, EGL is another way of coding the various server-side services for a JSF application. The structure of the server code is very similar to the Java backend. Some commonalities are:

- ▶ There is a page handler per JSF page, similar to the page code file generated for JSF pages without EGL.
- ▶ The Faces configuration file is automatically maintained.
- ▶ The server-side event logic can be easily accessed in the page handler.

A special perspective, EGL Web, is used to make use of EGL in cooperation with JSF within Application Developer.

A special type of project, EGL Web Project, is used for EGL Web development.

Creating a sample EGL JSF application

To illustrate the EGL and JSF interoperation we create a subset of the banking application using EGL. Refer to “Banking application overview” on page 192 for an overview.

Preparation

Before implementing the banking application with EGL, we have to setup the environment as described in “Setting up the environment for this redbook” on page 533:

- ▶ Create and load the EJBBANK database.
- ▶ Setup server targeting for the workspace.
- ▶ Prepare the JSFServer for testing.

Workspace preferences

Select *Window -> Preferences* and go to the EGL section. Set the SQL Database Connection to point to the EJBBANK database (Figure 12-1).

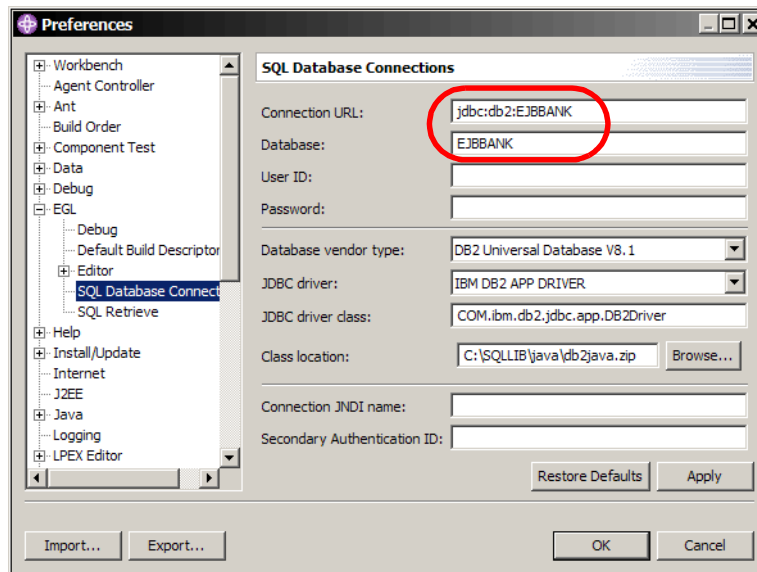


Figure 12-1 EGL preferences

Creating an EGL Web project

The Web project for EGL development is created in the same way as other projects by selecting *File -> New -> Project -> EGL -> EGL Web Project*. Enter **ItsoJSFEGL** as project name.

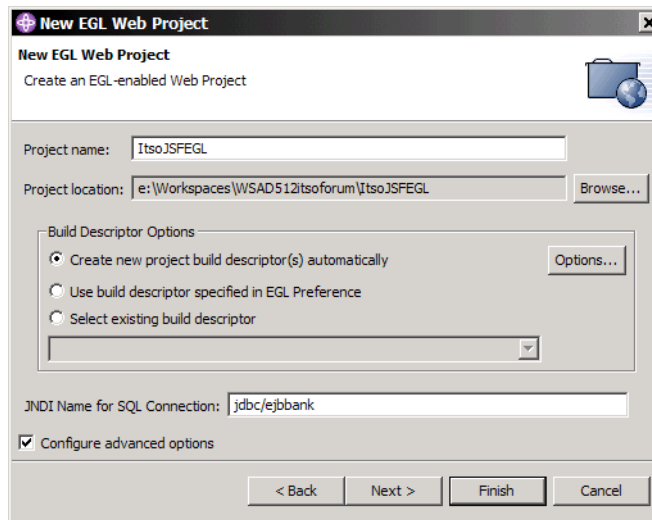


Figure 12-2 Creating a Web project for EGL development

In the New EGL Web Project dialog (Figure 12-2), pay attention to two areas:

- ▶ **Build Descriptor Options**—EGL build descriptors controls how the generation of the EGL into Java is done, including any necessary database connection information. The only option that can be modified in this dialog is the default SQL connection (click *Options* and verify that the database name is EJBBANK).
- ▶ **JNDI Name for SQL Connection**—set the JNDI name defined in the application server, for example, jdbc/ejbbank.

Select *Configure advanced options* and click *Next*. Enter the name of the EAR project as **ItsoJSF12EGL**. Select the target server as *WebSphere Application Server v5.1*, otherwise you are prompted later.

Click *Next*. In the Features Page, select *Add EGL Support* and *JSP Standard Tag Library* (Figure 12-3).

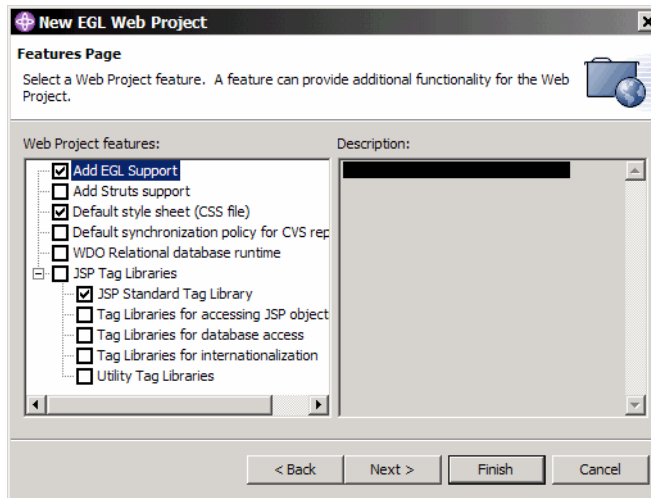


Figure 12-3 EGL Web project features

Click *Finish* to create the enterprise application and the Web project. Click *Yes* when prompted to switch to the EGL Web perspective.

EGL build descriptor

Let's examine the build descriptor for the EGL project. The build descriptor is located in the folder named `EGLSource`, and its name is `ItsoJSFEGLEglbld`. You can have several build descriptors in a project and customize them for different environments and targets.

Open the build descriptor (Figure 12-4). There are many options. By selecting *Show only specified options* you can reduce the number of displayed values.

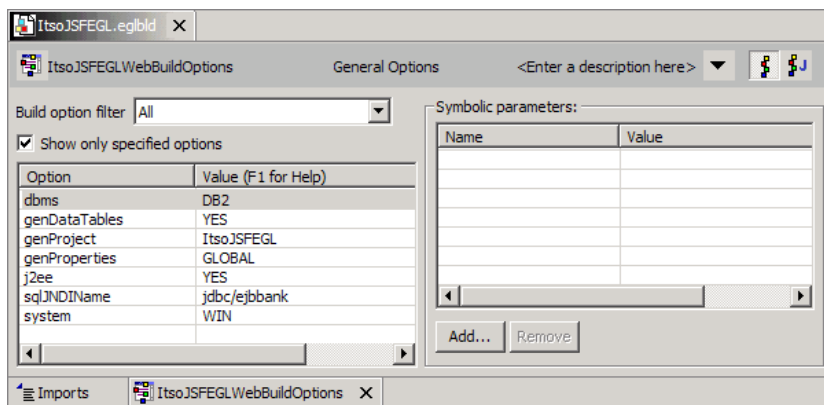


Figure 12-4 EGL build options for a Web project

Notice the JDBC JNDI name that we specified when creating the Web project.

The Web project also includes two style sheets (in the theme folder), the Faces configuration file, and the Faces servlets configured in the Web deployment descriptor.

Open the web.xml file to see the JSF servlets and a reference to jdbc/ejbbank on the References page Resource tab.

Select the ItsoJSFEGl project and *Properties*. In the properties dialog select *EGL Default Build Descriptors* and set the target system build descriptor to ItsoJSFEGlWebBuildOptions <ItsoJSFEGl/EGLSource/ItsoJSFEGl.eglbid>.

Creating a JSF page for EGL

Select WebContent and *New -> Faces JSP File*. Enter index.jsp as file name. Click *Finish* or go through the dialog pages using *Next*.

Upon creation of the first JSF page an EGL source package called pagehandlers is created (Figure 12-5). This package contains the page handlers for the JSF pages. A page handler EGL source file is created for each JSF Web page.

The EGL source file, index.egl, is compiled into an index.java file, stored in the pagehandlers Java package (under Java Resources).

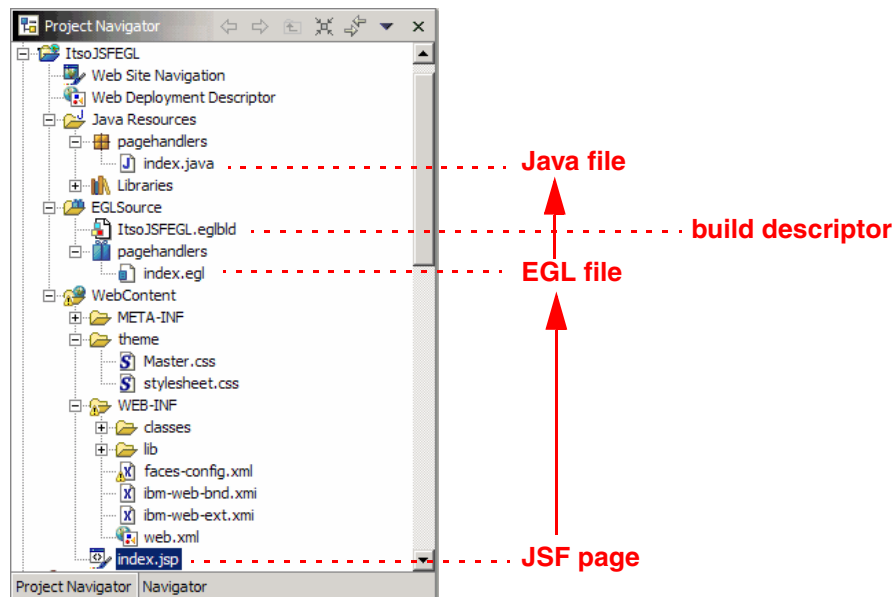


Figure 12-5 Layout of an EGL project

Generated EGL file

The page handler `index.egl` file contains this code:

```
package pagehandlers;

PageHandler index {view="index.jsp", onPageLoad=onPageLoad}

    Function onPageLoad()
    End
End
```

The page handler declaration links the source with the corresponding JSP file through the `view` attribute. The `onPageLoad` attribute points to the function that is executed before the page is shown for the first time.

Compiled Java file

The EGL source file is compiled into Java source code, `index.java`, shown here in compressed format:

```
package pagehandlers;
import com.ibm.etools.egl.format.*;
import com.ibm.vgj.server.*;
import com.ibm.vgj.wgs.*;
import java.util.ArrayList;
import java.util.HashMap;
import javax.faces.model.SelectItem;

public class index extends EGLPageBean
{
    public index() throws VGJException, Exception
    {
        this( "index" );
    }

    public index( String name$ ) throws VGJException, Exception
    {
        this( name$, new Integer( 1 ) );
    }

    public index ( String name$, Integer numRecords$ )
        throws VGJException, Exception
    {
        this( null, name$, numRecords$ );
    }

    public index ( EGLPageBean bean$, String name$, Integer numRecords$ )
        throws VGJException, Exception
    {

```

```

        super( bean$, name$, numRecords$.intValue() );
        VGJUtil.classLoader = index.class.getClassLoader();
        setInitialValues();
        setPage( "index.jsp" );
        setupEdits();
        setSubmitValueItem( null );
        eglOnPageLoad();
    }

    private void setupEdits() {}

    public void onPageLoad() throws VGJException
    {
        if ( EGLDebugSupport.DEBUG_MODE )
        {
            EGLDebugSupport.debug( this, "EGLSource/pagehandlers/index.egl",
                                   "ItsoJSFEGL", "onPageLoad", true );
        }
        else
        {
            $funconPageLoad();
        }
    }

    public void $funconPageLoad() throws VGJException
    {
        funcPush( "onPageLoad" );
        funcPop();
        return;
    }

    protected void processUserEdits() throws VGJException {}

    public void setInitialValues() throws VGJException {}
}

```

If you see a warning in the Tasks view, select the ItsoJSFEGL project and *Rebuild Project* to rebuild the project and the warning disappears.

Implementing the home page

To complete the home page we define an SQL record to retrieve the customer information, then complete the page design, and add the action logic.

Creating an SQL record for customer

Select the EGLSource folder and *New -> EGL Source File*. In the dialog enter itso.egl.data as package and CustomerRecord as file name. Click *Finish*.

The EGL editor opens on the CustomerRecord.egl file:

- In the source enter **r** and press Ctrl-space (content assist). Select *record - SQL record with table names* to generate a skeleton entry.
- Complete the skeleton with a name and a reference to the CUSTOMER table and delete the rest of the skeleton:

```
package itso.egl.data;

// SQL record with table names
//
record Customer type sqlRecord
    {tableNames=(ITSO.CUSTOMER)}

end
```

- Place the cursor on the Customer record and select *SQL Record -> Retrieve SQL* (context). The record declaration is completed by accessing the EJBBANK database. Remove the USERID and PASSWORD items (we do not use those):

```
record Customer type sqlRecord
    {tableNames=(ITSO.CUSTOMER), keyItems=(CUSTOMERID)}

    CUSTOMERID int    {column="CUSTOMERID", isReadOnly=no, isNullable=no, sqlVar=no};
    TITLE char(3)     {column="TITLE", isReadOnly=no, isNullable=no, sqlVar=no};
    FIRSTNAME char(30){column="FIRSTNAME", isReadOnly=no, isNullable=no, sqlVar=yes};
    LASTNAME char(30) {column="LASTNAME", isReadOnly=no, isNullable=no, sqlVar=yes};
    USERID char(8)   {column="USERID", isReadOnly=no, isNullable=yes, sqlVar=no};
    PASSWORD char(8) {column="PASSWORD", isReadOnly=no, isNullable=yes, sqlVar=no};
end
```

- Add a function to the CustomerRecord to retrieve the customer (\labscode\dev-jsfegl\CustomerRecord.txt):

```
package itso.egl.data;

Library CustomerRecord
    // retrieve the customer using the customerId
    function getCustomerByKey(customer Customer) returns(int)
        status int;
        status = 0;
        try
            get customer;
        onException
            status = sqlCode;
        end
        commit();
        return(status);
    end
end
record Customer type sqlRecord
.....
```

The `getCustomerByKey` function gets a customer record with the ID set as parameter, retrieves the complete customer record from the database, and sets a status field as return code. In case of errors the SQL code is returned.

- Save and close the file. Select the `CustomerRecord.egl` file and *Generate* (context) to generate the matching Java code.

Important: Whenever you change an EGL file be sure to execute the *Generate* option. Only page handler files are compiled into Java automatically.

Completing the welcome page

We now complete the welcome page, `index.jsp`, with a heading and a prompt for the customer ID. In the action code we retrieve the customer record:

- Change the generated code to a heading 1 with the text `ITS0 EGL Bank`.
- In the Palette view, expand EGL. Select *Record* and drop it into the JSP.
- In the Select a Record dialog, select the Customer record. Leave the name as customer and click *OK* (Figure 12-6).

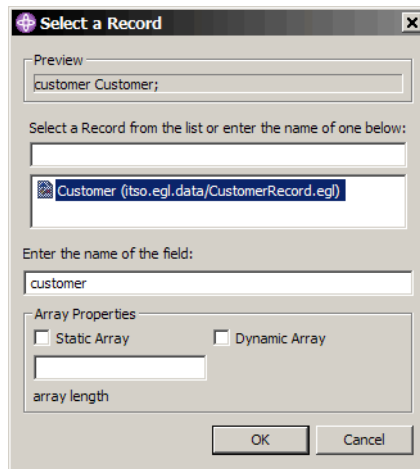


Figure 12-6 Selecting a record

- The customer record appears in the Page Data view (Figure 12-7).

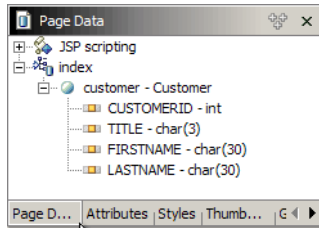


Figure 12-7 Page Data view with customer record

- Open the `index.egl` file. An import statement and a variable for the customer record has been added:

```
import itso.egl.data.*;
PageHandler index {view="index.jsp", onPageLoad=onPageLoad}

    customer Customer;
    .....

```

- Select the `CUSTOMERID` field in the Page Data view and drop it into the JSP. Select *Updating an existing record* and change the label prompt. Click *Options* and only generate a Submit button. Click *Finish* (Figure 12-8).

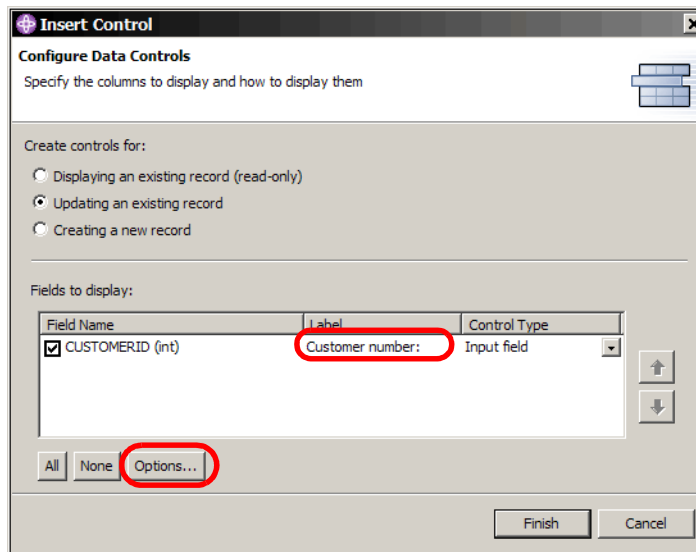


Figure 12-8 Adding a prompt for the customer ID

- The welcome page is now complete (Figure 12-9). Note that the input field is bound to the `CUSTOMERID` variable. Save the `index.jsp`.



Figure 12-9 ITS0 EGL Bank welcome page

Adding the action logic

When the Submit button is clicked, we have to retrieve the customer record from the database and display the next page, `listAccounts.jsp`:

- ▶ Create a JSF page named `listAccounts.jsp`. Close the file for now.
- ▶ In the `index.jsp`, select the Submit button. In the Attributes view change the ID to enter.
- ▶ In the Navigation tab, click *Add* to define a navigation rule. Select the `listAccounts.jsp` as target page, enter `accounts` as alias, and enter `*` for the action. Save the `index.jsp`.
- ▶ In the Page Data view, expand JSP scripting, select *sessionScope* and *Add Session Variable*. Enter `customerKey` as name and `java.lang.Integer` as type (Figure 12-10).

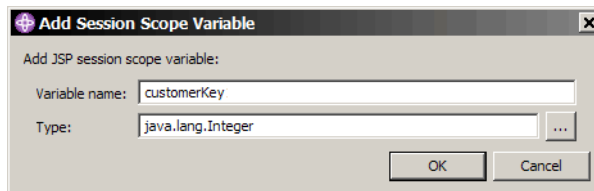


Figure 12-10 Add a session variable

- ▶ Select the Submit button and in the Quick Edit view, click into the code field. A function named `enterAction` is added. Add the action logic (`\labscode\dev-jsfeg1\IndexSubmit.txt`):

```
function enterAction()
    status int;
    // retrieve customer record using the EGL function
    status = CustomerRecord.getCustomerByKey(customer);
    if (status = 0)
        setSessionAttr("customerKey", customer.CUSTOMERID);
        forward to label "accounts";
    else
        if (status = 100)
```

```

        setError("Customer does not exist");
    else
        setError("Customer retrieval failed - database error");
    end
end
end
end

```

We retrieve the customer record using the EGL function we added to the EGL record. The customer ID is already in the record and is used as the key. We save the customer ID in the session variable `customerKey` for the rest of the user interaction. Then we set the next JSF page to be displayed. In case of an error we display an error message in the {Error Messages} field of the welcome page.

Testing the welcome page

To setup the test environment we add the enterprise application (`ItsoJSF12EGL`) to the JSFServer. For better performance remove any other applications.

Start the JSFServer, then select the `index.jsp` and *Run on Server*. When the welcome page is displayed, enter sample customer numbers:

- ▶ Enter 101 and the skeleton `listAccounts.jsp` is displayed.
- ▶ Enter 1aa and a data type error is displayed (only integers are allowed).
- ▶ Enter 123 and a customer not found message is displayed.

Debugging EGL code

To debug EGL code the EGL debug runtime must be added to the server:

- ▶ Stop the JSFServer.
- ▶ Select the JSFServer in the Servers view and select *Add EGL Debugger JAR Files* (context).
- ▶ Start the server in debug mode.
- ▶ Set breakpoints in the `index.egl` file. You can also set breakpoints in the `index.java` file.

Note. You cannot set breakpoint on declarations, only on executable code.

- ▶ Run the `index.jsp`. When prompted, select *Skip* and *Disable step-by-step mode*.
- ▶ The debugger stops at the end of the `onPageLoad` function. Click the *Resume* icon in the Debug view toolbar and the welcome page is displayed.
- ▶ Enter a customer number (101) and click *Submit*.
- ▶ Execution stops at a breakpoint (Figure 12-11).

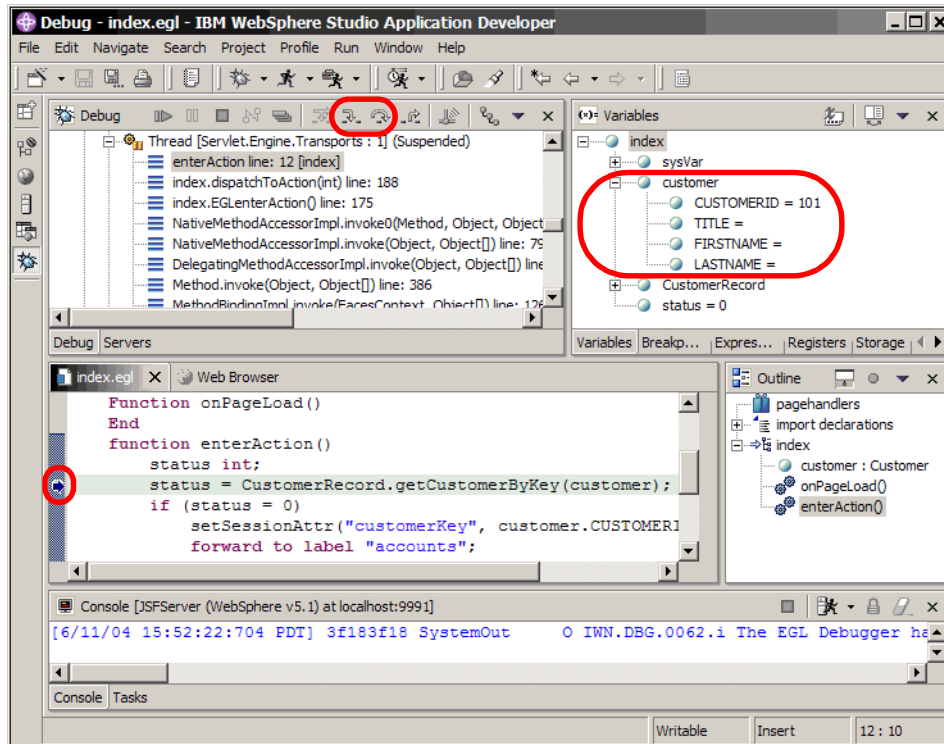


Figure 12-11 Debugging EGL code

- ▶ You can now step through the code using the *Step Into* and *Step Over* icons in the Debug view toolbar.
- ▶ Watch the variables in the Variables view.
- ▶ Stop the JSFServer when done.

Displaying the customer information

Next we display the customer information in the `listAccounts.jsp`:

- ▶ Open the `listAccounts.jsp`.
- ▶ Define a heading 1 as `Accounts of Customer`.
- ▶ In the Palette view, expand EGL, select *Record* and drop it into the JSP. When prompted select the Customer record. The record is added to the Page Data view.

- Select the customer record in the Page Data view and drop it into the JSP. When prompted, select *Displaying an existing record* and change the labels to suitable text constants (Figure 12-12).

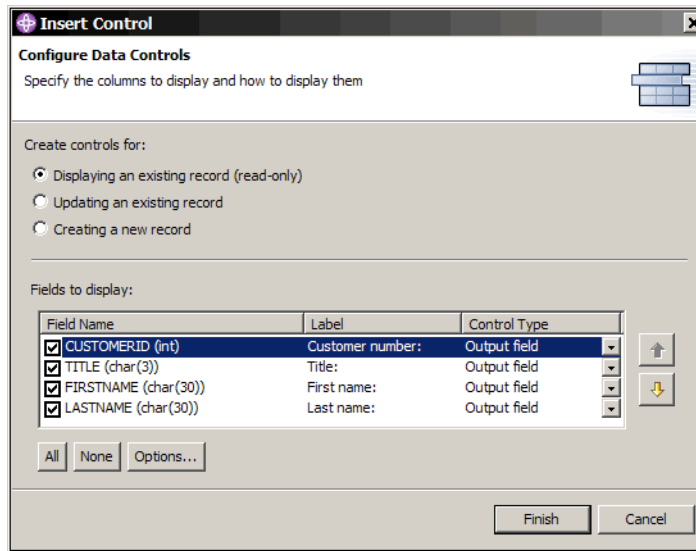


Figure 12-12 Displaying customer information dialog

- Click *Finish* and an HTML table with the four customer fields is added to the JSP (Figure 12-13).

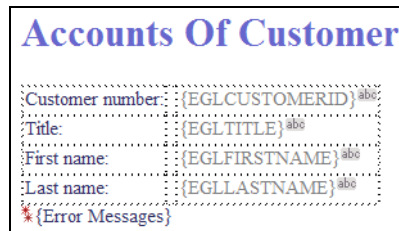


Figure 12-13 Displaying customer information in HTML table

- To display the customer information we have to retrieve it before the JSP is displayed. Open the `listAccounts.egl` file by selecting *View Page Code* in the JSP context.
- Complete the `onPageLoad` function with this code (`\labscode\dev-jsfegl\ListAccountsPageLoad.txt`):

```
Function onPageLoad()
    status int;
    // retrieve the customer using the key in the session
```

```

getSessionAttr("customerKey", customer.CUSTOMERID);
status = CustomerRecord.getCustomerByKey(customer);
if (status = 0)
    // retrieve accounts of customer - to be completed
else
    setError(Error retrieving customer");
end
End

```

- Save the JSP and EGL file. Test the application in the server and the customer information is displayed.

Creating an SQL record for the list of accounts

To display the list of accounts of a customer we have to define another SQL record:

- In the `itso.egl.data` package, create an EGL file named `CustomerAccounts`.
- Create a record skeleton:

```

record AccountsOfCustomer type sqlRecord
{tableNames=(ITS0.ACCOUNT A, ITS0.CUSTACCT C),
 defaultSelectCondition = #sql{A.ACCID=C.ACCID}}
end

```

- Select the record and *SQL Record -> Retrieve SQL*. The columns are retrieved.

Add the `CUSTOMERID` to the `defaultSelectCondition` (this is the search key). Delete the fields that we do not use.

```

record AccountsOfCustomer type sqlRecord
{tableNames=(ITS0.ACCOUNT A, ITS0.CUSTACCT C),
 defaultSelectCondition = #sql{A.ACCID=C.ACCID and
                                C.CUSTOMERID=:customerId},
 keyItems=(ACCID, CUSTOMERID, ACCID)}

ACCID char(8)          {column="A.ACCID", isReadOnly=yes,
                        isNullable=no, sqlVar=no};
BALANCE decimal(8,2)  {column="A.BALANCE", isReadOnly=yes,
                        isNullable=no, sqlVar=no};
INTEREST int          {column="A.INTEREST", ...};
ACCTYPE char(8)       {column="A.ACCTYPE", isReadOnly=yes,
                        isNullable=no, sqlVar=yes};
DISCRIMINATOR char(1) {column="A.DISCRIMINATOR", ...};
OVERDRAFT decimal(8,2){column="A.OVERDRAFT", ...};
MINAMOUNT decimal(8,2){column="A.MINAMOUNT", ...};
CUSTOMERID int        {column="C.CUSTOMERID", isReadOnly=yes,
                        isNullable=no, sqlVar=no};
ACCID char(8)         {column="C.ACCID", ...};
end

```

- Add the `getAccountsOfCustomer` function to the EGL file (`\labscode\dev-jsfegl\CustomerAccounts.txt`):

```
package itso.egl.data;
```

Library CustomerAccounts

```
// retrieve the accounts of a customer using the customerId
function getAccountsOfCustomer(customerId int,
                                accountsOfCustomer AccountsOfCustomer[]) returns(int)
    status int;
    status = 0;
    try
        get accountsOfCustomer
            usingKeys customerId;
    onException
        status = sqlCode;
    end
    commit();
    return(status);
end
end

// SQL record with table names
//
record AccountsOfCustomer type sqlRecord
.....
```

Notice that the function has two parameters, the `customerId` and an array `accountsOfCustomer`, which is filled by the SQL access. The SQL code is returned in the `status` variable.

- Save the `CustomerAccounts.egl` file. Select the file and *Generate* (context) to create the matching Java code.

Displaying the list of accounts

We now add the list of accounts to the `listAccounts.jsp`:

- In the Palette view, expand EGL, select *Record* and drop it into the JSP. When prompted select the `AccountsOfCustomer` record. Leave the name as `accountsOfCustomer`, and select *Dynamic Array* (we are retrieving multiple records). The record is added to the Page Data view.
- Select the `AccountsOfCustomer` record in the Page Data view and drop it into the JSP. When prompted, deselect `CUSTOMERID`, move `ACCTYPE` up, and change the labels to suitable text constants (Figure 12-14). Click *Finish*.

The data table is added to the JSP. Select the data table and in the Attributes view, All tab, change the border to 1 and cellpadding to 3.

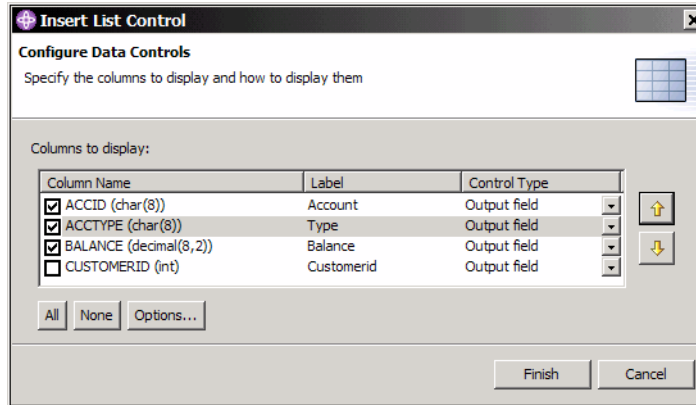


Figure 12-14 Tailoring the table of the list of accounts

- Complete the onPageLoad function in listAccounts.egl (\labscode\dev-jslgl\ListAccountsPageLoad2.txt):

```
Function onPageLoad()
    status int;
    // retrieve the customer using the key in the session
    getSessionAttr("customerKey", customer.CUSTOMERID);
    status = CustomerRecord.getCustomerByKey(customer);
    if (status = 0)
        // retrieve accounts of customer
        status = CustomerAccounts.getAccountsOfCustomer
            (customer.CUSTOMERID, accountsOfCustomer);
        case (status)
            when (0)
                ;
            when (100)
                ;
            otherwise
                setError("Error retrieving accounts of customer");
            end
        else
            setError("Error retrieving customer");
        end
    End
```

- Add a Command - Button component, change the id to cancel and the label to Cancel. Add a navigation rule with alias home that points to the index.jsp. Add a second navigation rule with alias details that points to accountDetails.jsp (this will be used for selecting an account). You have to enter accountDetails.jsp because it does not exist yet.

- In the Quick Edit view, add the action logic to clear the session data and return to the welcome page:

```
function cancelAction()
    clearSessionAttr("customerKey");
    forward to "home";
end
```

- The design is now complete (Figure 12-15). Save the `listAccounts.jsp`.

Accounts Of Customer

Customer number: {EGLCUSTOMERID} abc
 Title: {EGLTITLE} abc
 First name: {EGLFIRSTNAME} abc
 Last name: {EGLLASTNAME} abc

BR.

Account abc	Type abc	Balance abc
{EGLACCID} abc	{EGLACCTYPE} abc	{EGLBALANCE} abc

{Error Messages}

Cancel

Figure 12-15 Design view of `listAccounts.jsp`

Testing the list of accounts

Restart the enterprise application in the server, then run the `index.jsp`. Enter a customer number and the customer information and list of accounts is displayed (Figure 12-16).

Accounts Of Customer

Customer number: 106
 Title: Mr
 First name: Ueli
 Last name: Wahli

Account	Type	Balance
105-5001	CHECKING	
106-6001	CHECKING	993.00
106-6002	SAVINGS	2020.00
106-6003	SAVINGS	3000.00

Cancel

Figure 12-16 Testing the list of accounts

Selecting an account and passing the account number

The next step is to select an account and invoke the next JSP, passing the account number as parameter.

Note: EGL has no function to access the parameters in the request block in the action logic. However, the parameters are available in the `onPageLoad` function of the next JSP.

We could not perform the selection of an account using the row action support of the data table—the parameters are not passed.

Account selection is possible using the **Command - Hyperlink** or the **Link** component. The Command - Hyperlink uses action logic to set the next JSF page. The Link component uses the URL of the next JSF page. In both cases the passed parameters are accessed by name in the `onPageLoad` function of the next JSF page.

- ▶ Create a new JSF page named `accountDetails.jsp`. Close the page for now.
- ▶ Open the `listAccounts.jsp`. In the Page Data view select *sessionScope* and define a new session variable with the name `accountKey` and type `java.lang.String`. We keep the selected account as session data.
- ▶ Select a Command - Hyperlink component in the palette and drop it onto the account number `{EGLACCID}` in the data table.
- ▶ Select the link icon in the first column. In the Attributes view the id is `link1`.
- ▶ On the Parameters tab, click *Add* to define a parameter. Overtyping the name with `accountparm`. Select the value and click the bind icon . In the dialog expand *listAccounts* -> *accountsOfCustomer* and select `ACCID`. We pass the account number of the selected row.
- ▶ Select the link icon and in the Quick Edit view add the action logic:

```
function link1Action()  
    forward to label "details";  
end
```

Implementing account management

Next we implement the account details page with the typical banking operations of deposit, withdraw, transfer, and list of transactions.

Creating an SQL record for the account

Next we retrieve the selected account:

- In the `itso.egl.data` package, create an EGL file named `AccountRecord`.

- Create a record skeleton:

```
record Account type sqlRecord
    {tableNames=(ITSO.ACCOUNT)}
```

end

- Select the record and *SQL Record -> Retrieve SQL*. The columns are retrieved. Delete the columns that we do not use:

```
record Account type sqlRecord
    {tableNames=(ITSO.ACCOUNT),
    keyItems=(ACCID)}

    ACCID char(8)          {column="ACCID", isReadOnly=no,
                           isNullable=no, sqlVar=no};
    BALANCE decimal(8,2)  {column="BALANCE", isReadOnly=no,
                           isNullable=no, sqlVar=no};
    INTEREST int          {column="INTEREST", isReadOnly=no,
                           isNullable=no, sqlVar=no};
    ACCTYPE char(8)        {column="ACCTYPE", isReadOnly=no,
                           isNullable=no, sqlVar=yes};
DISCRIMINATOR char(1) {column="DISCRIMINATOR", ...};
OVERDRAFT decimal(8,2){column="OVERDRAFT", ...};
MINAMOUNT decimal(8,2){column="MINAMOUNT", ...};
end
```

- Add three data items and the `getAccountByKey` function to the EGL file:

```
package itso.egl.data;
```

```
DataItem AccountNumber char(8) end
```

```
DataItem TransactionAmount decimal(8,2) end
```

```
DataItem ActionString char(20) end
```

```
Library AccountRecord
```

```
// retrieve the account using the accountId
function getAccountByKey(account Account) returns(int)
    status int;
    status = 0;
    try
        get account;
    onException
        status = sqlCode;
    end
    commit();
    return(status);
```

```

        end
    end

    // SQL record with table names
    //
    record Account type sqlRecord
    .....

```

Notice that the function has one parameter, the account, with the account number set. The result of the function is the complete account record.

- ▶ Save the AccountRecord.egl file. Select the file and *Generate* (context) to create the matching Java code.

Displaying the account information

Open the accountDetails.jsp to display the selected account:

- ▶ Create a heading as Account Maintenance.
- ▶ Drop a Record from the palette into the JSP. Select the Account record, accept the defaults.
- ▶ Drop the account record from the Page Data view into the JSP. In the dialog move the ACCTYPE field into second position and change the labels (Figure 12-17).

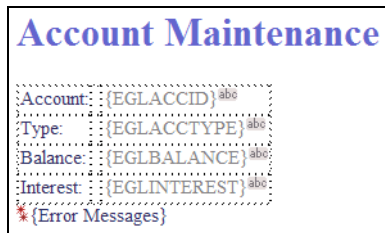


Figure 12-17 Displaying account information

- ▶ Open the accountDetails.egl file and complete the onPageLoad method (\labscode\dev-jsfeg1\AccountDetailsPageLoad.txt):

```

Function onPageLoad(accountparm AccountNumber)
    status int;
    accountid char(8);
    accountidlg int;
    errorMessage char(100);

    // retrieve the account using the key in the session
    getSessionAttr("accountKey", accountid);
    accountidlg = strLib.strLen(accountId);
    if (accountidlg = 0)

```



```

        // accountid passed from listAccounts
        accountid = accountparm;
        setSessionAttr("accountKey", accountid);
    end

    account.ACCID = accountid;
    status = AccountRecord.getAccountByKey(account);
    if (status != 0)
        errorMessage = "Error retrieving account: " + account.ACCID;
        setError(errorMessage);
        account.ACCID = "notFound";
    end

    // initialize the radio button
    // action = "listTransactions";    // to be used later
End

```

Note the parameter of the onPageLoad function. This is the parameter that we defined in the accountDetails.jsp for the hyperlink. The variable name (accountparm) must correspond to the parameter name and the type must be a data item that matches the parameter value.

- Save the file and run the application. Select an account and the details are displayed.

Completing the account details page

For the selected account we want to perform deposit, withdraw, and transfer operations. In addition we can list the transaction records:

- Add a table with a selection and two input fields to the accountDetails.jsp. Then add two command buttons (Figure 12-18).

Select the transaction:	
☐ List Transactions	
☐ Deposit	Amount:
☐ Withdraw	Destination account:
☐ Transfer	

Submit Cancel

Figure 12-18 Account operations design

- The radio button values are set to listTransactions, deposit, withdraw, and transfer.

- The AccountRecord.egl file defines the data items that we bind to the fields of the JSP:

```
DataItem AccountNumber char(8) end
DataItem TransactionAmount decimal(8,2) end
DataItem ActionString char(20) end
```

- In the accountDetails.jsp:
 - Select a DataItem component from the EGL palette and drop it into the JSP. Select the AccountNumber from the list and change the variable name to destinationAccountId.
 - Repeat this and select the TransactionAmount and set the variable name as amount.
 - Repeat this and select the ActionString and set the variable name as action.

The three variables are added to the accountDetails.egl file and are visible in the Page Data view of the JSP.

- Select a variable in the Page Data view and drop it onto the matching input field to create the binding (Figure 12-19):
 - Drop the amount field onto the first input field.
 - Drop the destinationAccountId variable onto the second input field.
 - Drop the action field onto the radio buttons.
 - Save the JSP.

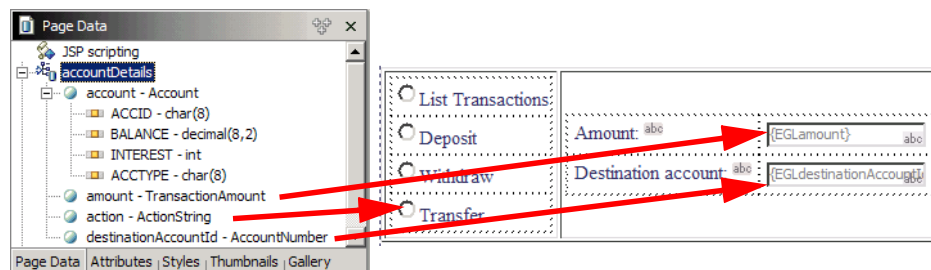


Figure 12-19 Binding EGL variables to input fields

- Select the Submit button and define three navigation rules:
 - stay—accountDetails.jsp
 - transactions—listTransactions.jsp (does not exist yet)
 - accounts—listAccounts.jsp
- Open the accountDetails.egl file and activate the code in the onPageLoad function to initialize the radio button:

```
// initialize the radio button
action = "listTransactions";
```

Implementing account operations

First we define the operations on the account record:

- Open the `AccountRecord.egl` file and add the `deposit` function to the `AccountRecord` library (`\labscode\dev-jsfeg1\AccountRecordDeposit.txt`):

```
function deposit (accountId AccountNumber, amount TransactionAmount)
    returns(int)

    status int;
    status = 0;
    account Account;
    account.ACCID = accountId;
    try
        get account forUpdate;
    onException
        status = sqlCode ;
    end
    if (status = 0)
        account.BALANCE = account.BALANCE + amount;
        try
            replace account;
        onException
            status = sqlCode ;
        end
        if (status = 0)
            // status = TransactionRecord.create(accountId, amount, "C");
        end
    end
    if (status = 0)
        commit();
    else
        rollback();
    end
    return(status);
end
```

The `deposit` function retrieves the account for update, changes the balance, and replaces the account. Then a transaction record is created (this is a comment for now). If both operations complete a `commit` is done.

- Add the `withdraw` and `transferfunds` functions in the same way:

```
\labscode\dev-jsfeg1\AccountRecordWithdraw.txt
\labscode\dev-jsfeg1\AccountRecordTransferFunds.txt
```

The `transferfunds` function has both the `withdraw` and `deposit` logic.

- Select the EGL file and *Generate* (context).
- Select the Submit button and in the Quick Edit view add the action logic (`\labscode\dev-jsfeg1\AccountDetailsSubmit.txt`):

```
function submitAction()
    errorMessage char(100);
    status int;
```

```

destinationAccount Account;
fromAccountId char(8);
status = 0;
fromAccountId = account.ACCID;
case (action)
    when ("listTransactions")
        forward to "transactions";
    when ("deposit")
        status = AccountRecord.deposit(fromAccountId, amount);
        if (status = 0)
            forward to "stay";
        else
            errorMessage = "Deposit failed";
        end
    when ("withdraw")
        status = AccountRecord.withdraw(fromAccountId, amount);
        if (status = 0)
            forward to "stay";
        else
            errorMessage = "Withdraw failed";
        end
    when ("transfer")
        status = AccountRecord.transferfunds(fromAccountId,
                                              destinationAccountId, amount);
        if (status = 0)
            forward to "stay";
        else
            errorMessage = "Transfer failed";
        end
    when ("")
        errorMessage = "Select the action you would like to perform";
    otherwise
        errorMessage = "Action not recognized: " + action;
end
if (errorMessage != " ")
    setError(errorMessage);
end
end

```

- Select the Cancel button and in the Quick Edit view add the action logic:

```

function cancelAction()
    clearSessionAttr("accountKey");
    forward to label "accounts";
end

```

Implementing SQL functions for transaction records

To insert a transaction record and to list the transactions records we create an SQL record and two functions:

- In the `itso.egl.data` package, create an EGL file named `TransactionRecord`.

- Create a record skeleton:

```
record TransRecord type sqlRecord
    {tableNames=(ITS0.TRANSRECORD)}

end
```

- Select the record and *SQL Record -> Retrieve SQL*. The columns are retrieved. Add the selection condition to retrieve records by account ID:

```
record TransRecord type sqlRecord
    {tableNames=(ITS0.TRANSRECORD),
    defaultSelectCondition = #sql{ACCID=:accountId},
    keyItems=(TRANSID)}

    TRANSID char(26)      {column="TRANSID", ...};
    ACCID char(8)         {column="ACCID", ...};
    TRANSTYPE char(1)     {column="TRANSTYPE", ...};
    TRANSAMT decimal(8,2){column="TRANSAMT", ...};

end
```

- Add the getTransForAccount and create functions to the file (\labscode\dev-jsfegl\TransactionRecord.txt):

```
DataItem TransAmount decimal(8,2) end

Library TransactionRecord
    // retrieve the transaction records of an account
    function getTransForAccount(accountId char(8), transRecord TransRecord[])
        returns(int)

        status int;
        status = 0;
        try
            get transRecord
                using accountId;
        onException
            status = sqlCode;
        end
        commit();
        return(status);
    end

    // insert a transaction record
    function create(accountId char(8), amount TransAmount, creditdebit char(1))
        returns(int)

        status int;
        status = 0;
        transRecord TransRecord;
        try
            add transRecord with
                #sql{
                    insert into ITS0.TRANSRECORD (TRANSID, ACCID, TRANSTYPE, TRANSAMT)
                    values (current timestamp, :accountId, :creditdebit, :amount)
                };
        end
```

```

        onException
            status = sqlCode;
        end
        return(status);
    end
end

```

- ▶ Open the AccountRecord.egl file. Activate the creation of transaction records in the deposit, withdraw, and transferFunds methods (there are four instances):

```
status = TransactionRecord.create(accountId, amount, "C");
```

Be sure to select the EGL file and *Generate*.

Implementing the list of transaction records

The account is available in the session data and we can retrieve the transaction records:

- ▶ Create the listTransactions.jsp.
- ▶ Add a heading Transaction Records of Account:.
- ▶ Drop an Output component after the heading and bind it to the *sessionScope* -> *accountKey* variable.
- ▶ Select *Record* from the EGL palette and drop it into the JSP. Select the TransRecord, leave the name as transRecord, and select *Dynamic Array*.
- ▶ Drop the transRecord from the Page Data view into the JSP. Deselect the ACCID field and change the labels to suitable column headings.
- ▶ Drop a command button under the list and name it Back.
- ▶ Select the Back button and add a navigation rule with alias back and pointing to the accountDetails.jsp. In the Quick Edit view enter the action logic:

```

function backAction()
    forward to label "back";
end

```

- ▶ This completes the design of the listTransactions.jsp (Figure 12-20).

Transaction Records Of Account: {accountKey}

Timestamp	Credit/Debit	Amount
{EGLTRANSID}	{EGLTRANSTYPE}	{EGLTRANSAMT}

Back

Figure 12-20 Design of the listTransactions.jsp

- Open the page code and complete the onPageLoad function (\labscode\dev-jsfeg1\ListTransactionsPageLoad.txt):

```

Function onPageLoad()
    status int;
    accountid char(8);

    // retrieve the transaction using the account key in the session
    getSessionAttr("accountKey", accountId);
    status = TransactionRecord.getTransForAccount(accountId,
                                                    transRecord);

    if (status != 0)
        setError(this,"","Error retrieving transactions: ",accountId,
                  " code=",status);
    end
End

```

Running the EGL banking application

The application is now complete and you can run through all the steps. Perform some banking operations and verify that the transactions records are created.

Complete solution

We provide the complete solution for this application in the additional material in the file \labscode\solution\ItsoJSF12EGL.ear. See “Importing a solution enterprise application” on page 548 for instructions.

EGL considerations

EGL is similar to how you would develop JSF applications with Java. That is, you can change the attributes on the various components in the Page Designer, add action logic to the commands, and so forth.

Note though that a wide range of validations can be built directly onto the data definitions in EGL, allowing for very sophisticated validation that can be reused across platforms and applications.

A general recommendation is that if a very rich JSF UI is expected, then do not use server-side JSF EGL. Instead use the EGL capability of wrapping EGL code into ordinary Java code and use Java for the JSF server-side code and for accessing the EGL business logic. This may be easier to implement for a Web developer and EGL skills would only be needed for business logic.

Tailoring data items

You can assign display names to data items so that good label names are generated when you drop a record into a JSF page. For example, in the `AccountRecord.egl`, change the code of the `ACCTYPE` item:

```
ACCTYPE char(8) {column="ACCTYPE", isReadOnly=no, isNullable=no,  
                  sqlVar=yes, displayname="Type"};
```

When you drop the account record into a JSP, the label is set to Type instead of Acctype.

Is is worthwhile studying the Help documentation *Developing -> EGL Applications*.



Faces Client framework

This chapter describes the Faces Client framework, which is a system for working with model data directly in the client. We discuss the concepts behind the Faces Client framework, the Application Developer support for the framework, and provide some example applications.

The Faces Client framework is an IBM extension of JavaServer Faces.

Faces Client framework introduction

The Faces Client framework is a system for providing rich client data interaction in the browser within the confines of the J2EE architecture. The framework establishes an entire MVC environment in the browser. In this client-based MVC environment, the model is provided by delivering the server data model into the browser, providing its interface through JavaScript. The view is provided by HTML controls that can update the client model. Client-data bound controls are also notified of changes to the model. Controller function is written as JavaScript logic against the model objects and HTML controllers.

The Faces Client framework is also known as the *On Demand Client* framework, or ODC framework.

Capabilities

The capabilities gained from using the Faces Client framework are impressive:

- ▶ **Operation batching**—Using this system, an application user can perform multiple operations on the data without having to submit the changes to the server until a transaction commit is required.
- ▶ **Rapid paging**—The entire model is resident in the browser so that you can page through a large set of data without having to interact with the server.
- ▶ **Control synchronization**—Controls which are bound to the same client model object are kept synchronized and they respond to model changes. This means you can choreograph multiple views of the same data.
- ▶ **On-the-fly data binding**—The model objects to which controls are bound can be changed through JavaScript calls, facilitating a master-detail pattern that does not access the server to change the objects.
- ▶ **Submitting only data changes**—Because the server model can be reconstituted by enacting all the operations in the client model, when submitting the changes to the server, only the delta between the original and current client model data is sent. This reduces network traffic.

Faces Client concepts

The Faces Client framework has a complicated flow of data from the server to the browser. Each step in this flow utilizes some new concepts.

The first step ODC is responsible for is supplying the server data model in a form that can be driven through JavaScript in the client.

Emitters

Emitters are responsible for taking Java objects from the server and converting them to HTML and JavaScript, which, when processed by the browser, become the client data model. Specialized emitters can be generated by the ODC tools for each type of object in the server model. Each emitter then takes a current model Java object, and serializes it to JavaScript logic that represents that object.

Once the model data is delivered to the browser, it is important to understand how that data is represented.

SDO for JavaScript

In the browser, model objects take the form of Service Data Objects for JavaScript (SDO4JS). SDO4JS basically implements the SDO framework using a partial JavaScript implementation of the Eclipse Modeling Framework (EMF). These client model objects can be introspected and accessed through both strongly and loosely typed methods. Any changes to client data are stored in the change logs of the corresponding SDO data graphs. These objects are the core model objects, against which the view and controller act.

The ODC framework allows for a variety of functions to be performed against the client data objects. Those functions are delivered in the form of a number of JavaScript include statements.

JavaScript Library

The JavaScript Library (JSL) is the large repository of reusable client function which must be resident in the browser to implement the SDO4JS framework, the event-listener model for the Web controls, and advanced Web control functionality, such as the data grid. This code is responsible for serializing the changes from the client objects for submission to the server.

When the appropriate JavaScript method is invoked, JSL calculates the model delta information and stores it in a form element. This delta is submitted to the server as a request parameter.

DiffGram

Model deltas are serialized into HTML forms as DiffGrams. DiffGrams are a hierarchical view of any model objects that have been created, updated, or deleted while the application user was using the browser to manipulate the data.

Only those objects which have been affected are present in the DiffGram. If you had one hundred records in the model, and you changed three before submitting to the server, the DiffGram only contains information about the three modified records.

The server code must inspect the DiffGram and update the server model objects accordingly. The business logic responsible for synchronizing the client and server model using DiffGrams must currently be implemented by the application developer.

DiffHandlers

Each type of model object requires a DiffHandler, which contains the necessary function to take a model object of that type, the corresponding section of a DiffGram, and update the model object accordingly. The DiffGram can be represented using EMF on the server, so EMF APIs are used to analyze the DiffGram.

Once all the server model objects have been updated by the DiffHandlers, the data is ready to be sent back to the browser for further modification.

The Faces Client model data round trip can be visualized as shown in Figure 13-1.

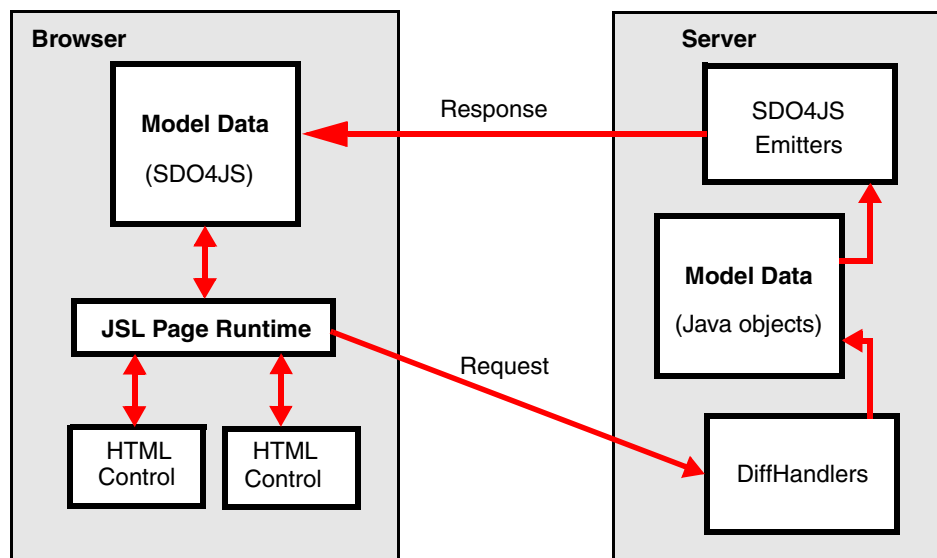


Figure 13-1 Faces Client model data round trip

Faces Client tools

WebSphere Studio Application Developer provides tools for developing JavaServer Faces Client applications. Specifically, Studio provides tools for creating client data objects and their associated artifacts, and for designing pages with client data bound controls. Studio provides new versions of some of the standard Faces UI components with Faces Client functions, as well as some altogether new components.

Faces Client page

To create a Faces Client page in a JSF page, launch the New Faces JSP File wizard. Select *Basic with client side model caching* for the page model (Figure 13-2).

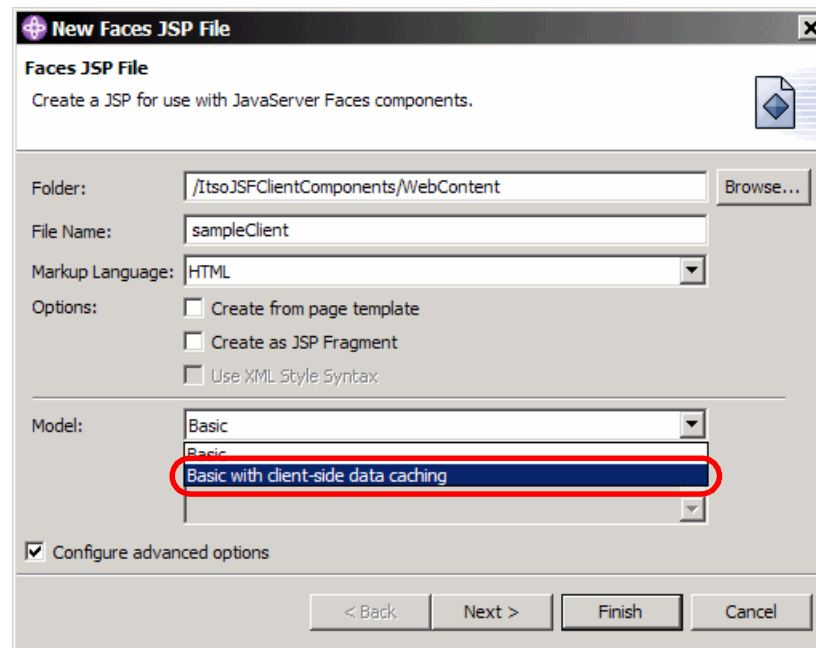


Figure 13-2 Selecting the Faces Client model for a JSF page

You may be prompted to target a WebSphere Application Server v5.1 instance if the project does not already target such a server.

When the wizard finishes, it adds the necessary Faces Client JARs to the project. It also creates a properties file in the Java Resources folder, called `OdysseyBrowserFramework.properties`.

This file contains various configuration properties for the framework. The name Odyssey comes from the acronym for On Demand Client, ODC. You should not have to modify this file.

Page Designer

Open a Faces Client JSP with the Page Designer. A new drawer is available in the palette, Faces Client Components. This contains the components unique to the JSF Client framework:

- ▶ Data Grid
- ▶ Graph
- ▶ Tree View
- ▶ Web Service

See “Faces Client components” on page 450 for more details on these components.

Some of the standard JSF UI components are modified to support client-side data binding:

- ▶ Image
- ▶ Input
- ▶ Input - Text Area
- ▶ Input - Password
- ▶ Radio Button Group
- ▶ Check Box
- ▶ Check Box Group
- ▶ List Box - Single Select
- ▶ List Box - Multiple Select
- ▶ Combo Box
- ▶ Output

See Chapter 4, “JSF components” on page 85 for more details on these components.

Creating client data

For this section we assume that a Customer JavaBean with properties `age`, `firstName`, and `lastName` has been defined as a managed bean called `customer` in the Page Data view.

Client Data view

The Faces Client framework is supported by a view named *Client Data* (Figure 13-3).

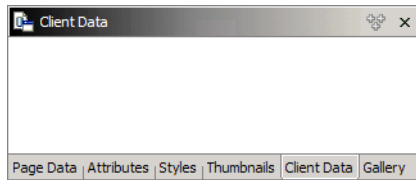


Figure 13-3 Client Data view (empty)

This view contains references to all client model objects that are included in this page. Any objects you have defined in the Page Data view can be imported as client data objects using this view.

Click the  icon to create a new client data object. This icon is only shown if a Faces JSP is open. In the New data component dialog, you only have one option, *Client Data*. Select it and click *OK*. You are presented with the Add Client Data dialog (Figure 13-4).

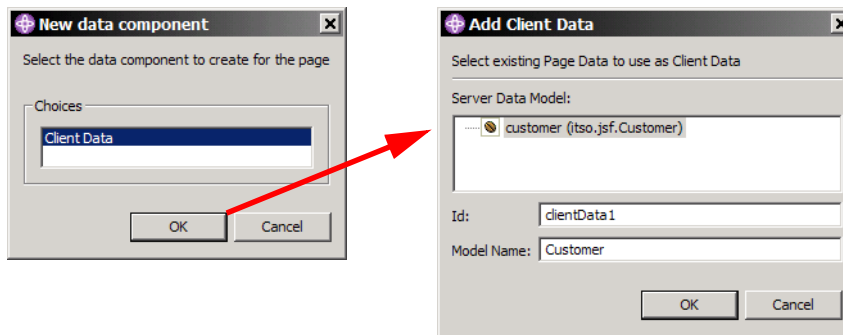


Figure 13-4 Selecting an existing page data object to use as client data

Here you can select the existing page data object to use as client data. You are prompted to specify an Id and a Model Name. The Id is used to distinguish between multiple client data objects in the JavaScript namespace.

Note: You do not have to remember or use the model name you enter here.

When you finish creating a client data object, you can see the object added the client data view. Note the decorators, which indicate the EMF object type of each property of the client data object (Figure 13-5). The types dictate to which components the objects can be bound to.

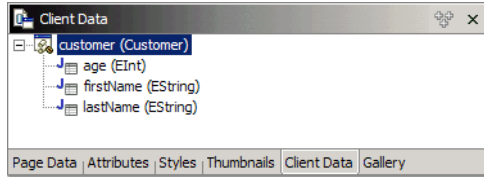


Figure 13-5 Client Data view (populated)

Two packages with files are created when a client data object is added.

EMF artifacts

EMF artifacts contain the EMF model and map information of the object, and are generated into the `com.ibm.dynwdo4jsmediators` package.

- `customer.ecore`—EMF model object with its own editor (Figure 13-6).

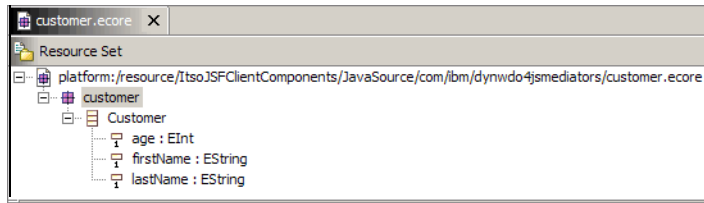


Figure 13-6 EMF model object

- `customer.emap`—XML file with EMF model mapping:

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings>
  <EPackage name="customer">
    <EClassMap app_class="itso.jsf.Customer" eclass_name="Customer"
      export="customer_Customer">
      <EFeatureMap efeature_name="age" id="true" get="getAge()" />
      <EFeatureMap efeature_name="firstName" id="true" get="getFirstName()" />
      <EFeatureMap efeature_name="lastName" id="true" get="getLastName()" />
    </EClassMap>
  </EPackage>
</mappings>
```

SDO4JS emitters

The SDO4JS emitters and an EMF model for the client data object are placed into the `com.ibm.dynwdo4jsmediators.customer` package:

```
customer_customer_Customer_wdo4js_js.java
customer_customer_Customer_wdo4js_xmi.java
customer_client.ecore
```


ODC taglib and properties file

The ODC taglib reference is added to the JSP, along with a tag in the body:

```
<%@ taglib uri="http://www.ibm.com/jsf/BrowserFramework" prefix="odc" %>
.....
<odc:clientData modelName="Customer"
    value="#{pc_SampleClient.customer}" id="clientData1"/>
```

This tag drives the SDO4JS emitters to write their output. The tag is bound to the managed bean we imported as a client data object for the purpose of encoding the model data. The JSP page code managed bean is updated to add a reference of type `HtmlClientData` and an accessor method (`getClientData1`) to the component used in the `<odc:clientData>` tag.

The `OdysseyBrowserFramework.properties` file is also updated to point to the new EMF artifacts:

```
EMAP_FILES=\ com\\ibm\\dynwdo4jsmediators/customer.emap
ECORE_FILES=\ com\\ibm\\dynwdo4jsmediators/customer.ecore
```

Configuring client data objects

Select the object in the client data view and *Configure*. This launches the Configure Client Data dialog (Figure 13-7), which is used to manipulate the properties of the client data object.

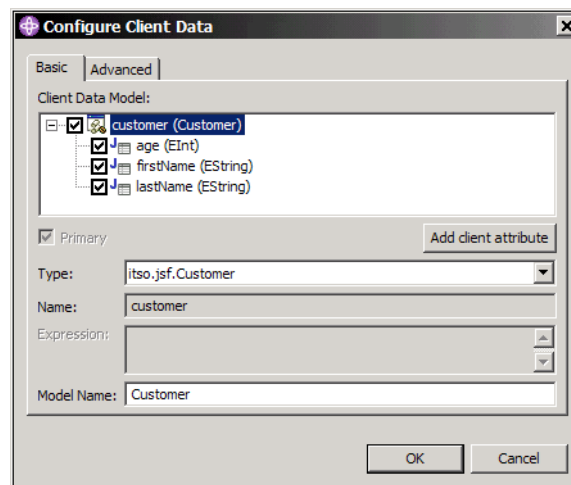


Figure 13-7 Configuring basic client data properties

This view is used to primarily to prune properties that do not have to be exposed as client data for the purposes of rich browser interaction. You may also create new properties that are calculated from JavaScript expressions. Press F1 and refer to the online help for more details on this panel.

Select the Advanced tab (Figure 13-8). This tab is for advanced manipulation of the client data object.

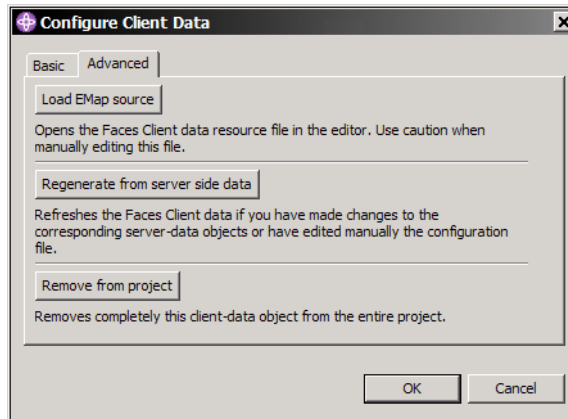


Figure 13-8 Advanced configuration of client data

The most useful feature on this panel is the *Remove from project* button, which removes all of the artifacts of the client data object from the project. This is more effective than selecting an object in the Client Data view and *Delete*.

The *Regenerate from server side data* button is also useful to keep the client data object in sync with any changes you make to the original page data object.

Binding client data to controls

Client data objects can be bound to client-enabled UI component using drag and drop. It is very important to realize that client data binding and server data binding are mutually exclusive. Binding to one eliminates any binding to the other. Client data bound component and server data bound component can be used side by side, however.

Attempting to access and grab the value from a client-bound component from server-side page code does not work. The component's value is stored in SDO4JS in the response, in the browser during client processing, and in a DiffGram on the request.

When you bind a client data object to a UI component, the binding is stored in a nested `<odc:clientBinder>` tag, and not in the value attribute. Examine the following input component that has been bound to a client data property:

```
<h:inputText styleClass="inputText" id="text1">
  <odc:clientBinder value="#{pc_SampleClient.customer.firstName}"/>
</h:inputText>
```

The tools must distinguish between client or server binding in the Attributes view. When the value is displayed in the tag attributes, @ is used instead of # in the value binding syntax. Therefore this example binding is displayed in the Attributes view as:

```
@{pc_SampleClient.customer.firstName}
```

On the All tab of the Attributes view, however, the value attribute is empty, so you have to edit the binding in the Basics tab.

If you click the  icon in the Basics tab, you can see that the Select Page Data Object dialog now includes an option to select from either server data, client data, or both (Figure 13-9).

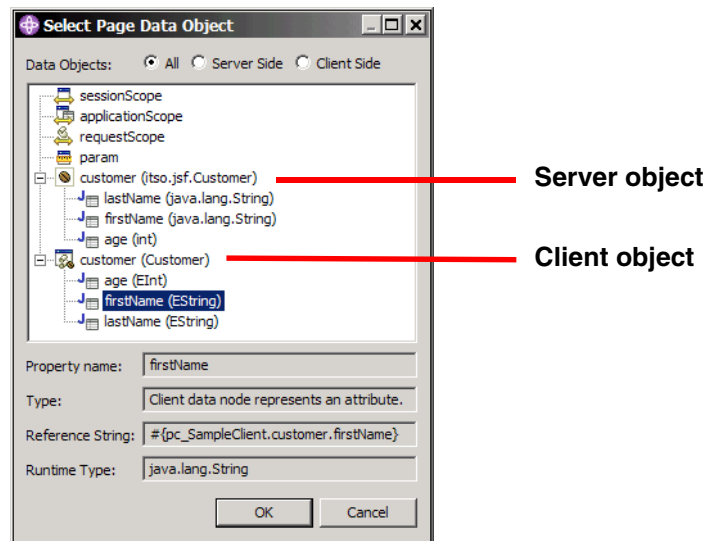


Figure 13-9 Selecting client data for value binding

Note: You only get the choice between server and client objects if the component is a server-side component. For pure JSF Client components you can only bind to client data.

Editing client component events

The Quick Edit view displays some information in comments about available objects for JavaScript events. There are also Faces Client specific code snippets available if you right-click in the edit panel. We do not cover advanced Faces Client JavaScript programming in this book.

Testing Faces Client applications

When testing Faces Client JSPs in the test environment, we received a class path error message in several instances:

```
java.lang.NoClassDefFoundError: com/ibm/etools/wdo/datagraph/DataGraph
```

We found that changing the class loader policy for the enterprise application in the server configuration editor, Applications page, from PARENT_FIRST to PARENT_LAST resolved the problem.

Faces Client components

In this section we provide a description of the Faces Client components provided by WebSphere Studio Application Developer 5.1.2.

Data Grid: <odc:dataGrid>

This component is used to represent data in a two-dimensional array. In this sense it is similar to the JSF Data Table component. However, it allows you to sort, filter, page through, and generally manipulate data on the client without causing server-side processing.

In the other hand, this component is fully driven by metadata and renders the content of its cells entirely in the browser. This means that you are limited as to what you can show in a cell. Whereas the JSF Data Table component allows you to easily nest a Data Table inside another, the Data Grid component is incapable of doing this.

If you want to customize the Data Grid component, you can edit the data grid style sheet in WebContent\theme\datagrid.css file. It contains a set of styles with the prefix datagrid.

Attributes

Table 13-1 shows the attributes of the <odc:dataGrid> component.

Table 13-1 Data Grid attributes

Attributes	Description
common	id, binding, value, rendered, styleClass.
allowRowAddAndDelete	Whether or not add edit buttons in the rows (delete row, add a new row).
multiSelection	Whether or not to allow multiple row selection.
navBarPosition	Sets the place where the navigation bar will be shown. The possible values are: -1 (none), 0 (bottom), 1 (top), and 2 (both).
pageSize	The number of items shown in each page of the table.
rowFilter	This attribute is a piece of code returning an boolean value. If the compute value is false, the row will not be shown in the table.
selectionColAlignment	If the showSelectionColumn attribute is true, this attribute sets the alignment for the check boxes or radio buttons in this column.
selectionColName	If the showSelectionColumn attribute is true, this attribute sets the name for the selection column.
selectionColWidth	If the showSelectionColumn attribute is true, this attribute sets the width of the selection column.
showRowIndex	If true, a new column with the row indexes is shown in the table.
showSelectionColumn	If true, a new column is shown. This column contains check boxes or radio buttons to select multiple or single columns, respectively.

Apart from these, there are more attributes for each of the columns in the table. To edit either the columns and their attributes, you use the Column List tab, the Column tab, and the Column Format tab in the Attributes view:

- **Column List**— Add, remove, reorder and modify the columns in the table.
- **Column**—Edit the column selected in the Column List tab and set these attributes:
 - Id for the column
 - Header of the column
 - Alignment
 - Width
 - If the column is read-only
 - If the column is displayed underlined

- **Column Format**—Format the data shown in the column that is selected in the Column List tab. You can format the values as string, numbers, dates, and mask fields.

You can see an example of these tabs in Figure 13-10.

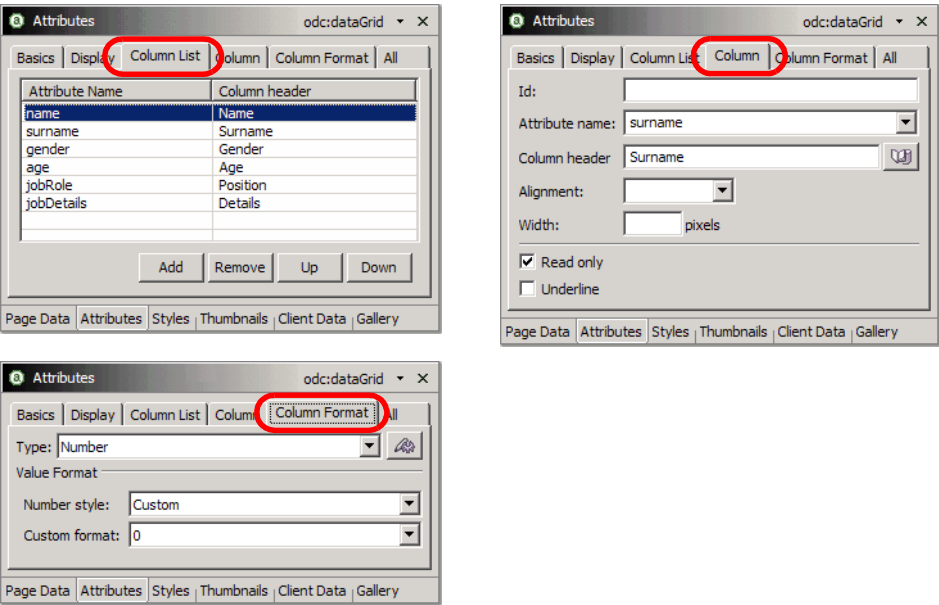


Figure 13-10 Attributes view tabs for column editing

Events

Table 13-2 shows the events of the <odc:dataGrid> component.

Table 13-2 Data Grid Quick Edit events

Event	Description
onhighlight	This event executes a JavaScript code when a row in the table is clicked and highlighted.
onselect	This JavaScript code will be executed when a row is selected. Row selection must be enabled for this event to get fired.
onunselect	This event executes a JavaScript code when a selected row is unselected. Row selection must be enabled for this event to get fired.

Graph: <odc:graphDraw>

This component is a rich component that uses Macromedia Flash technology instead of JavaScript or DHTML technology. It can be bound to data the same way as other Faces Client components.

This control allows you to set up many data series, choose between pie, bar, and line charts, group the values of a series using a function, and so forth.

Attributes

Table 13-3 shows the attributes of the <odc:graphDraw> component.

Table 13-3 Graph attributes

Attributes	Description
common	id, binding, value, rendered.
bar	If true, the bar graph will be shown. This is the default type of graph.
firstChart	This attribute can take the following values: bar, pie, and line. Specifies the default char that will be shown in run-time. If nothing is selected for this attribute, the bar graph will be the default one.
height	This attribute specifies the height of the component.
line	If true, the line graph will be shown.
pie	If true, the pie graph will be shown.
showLabel	This attribute specifies if the value for the data series must be shown.
showLegend	If true, a legend will be shown. This legend explain the color code for the data series and the labels for each data series.
splitYAxis	Selects whether or not plot y-axes on either side of the x-axis. This is done for bar and line graphs.
title	The title of the graph.
width	The width of the component
xaxisTitle	The title for the x axis.
yaxisDivision	This attribute specifies the scale of the y axis. This value is useful to normalize the graph when there are abnormal highest or lowest values.
yaxisTitle	The title for the y axis.

Apart from these, there are more attributes for each of data series that can be represented using this component. To edit their properties, you use the Data

Labels tab, the Data Series tab, the Label Format tab and the Data Format tab in the Attributes view:

- ▶ **Data Series**—Allows you to define the data that is displayed in the graph. You must add series to this dialog to use them in the graph. When adding data series to the graph, you can group these data series by selecting *Group values*. When you use the grouping, the group operation selector appears in the third column of the dialog. There you can choose the operation you want to use to group the data. This operation can be: SUM, AVG, COUNT, MIN, MAX, FIRST and LAST.
- ▶ **Data Labels**—Allows you to label the data series that is selected in the Data Series tab.
- ▶ **Label Format**—Allows you to set the format of the labels for the series. This format include string, number, date, and mask format.
- ▶ **Data Format**—Allows you to set the format of the data. This format include string, number, date and mask.

Events

The Graph component has no events.

Tree View: <odc:tree>

This component allows you to display hierarchical data using a tree shape.

Attributes

Table 13-4 shows the attributes of the <odc:tree> component.

Table 13-4 Tree View attributes

Attributes	Description
common	id, binding, value, rendered, styleClass.
enableSelect	If true, the selection of a particular node is enabled.
height	The height of the component.
rootVisibleFlag	This attribute specifies if the root node will be shown or not.
systemIconStyle	The style of the default icon witch is representing the nodes. This can be overridden for each node.
width	The width of the component.

When you bind data to this component, the Tree Node List tab is populated with the data and you have these functions:

- You can select which nodes you want to display in the tree, by editing the data (Figure 13-11).
- You can select the icons that are shown when you expand or collapse the nodes in the tree. When you select these icons, the files are imported to the project into the WebContent\ibmjsfres\bf\icons folder.

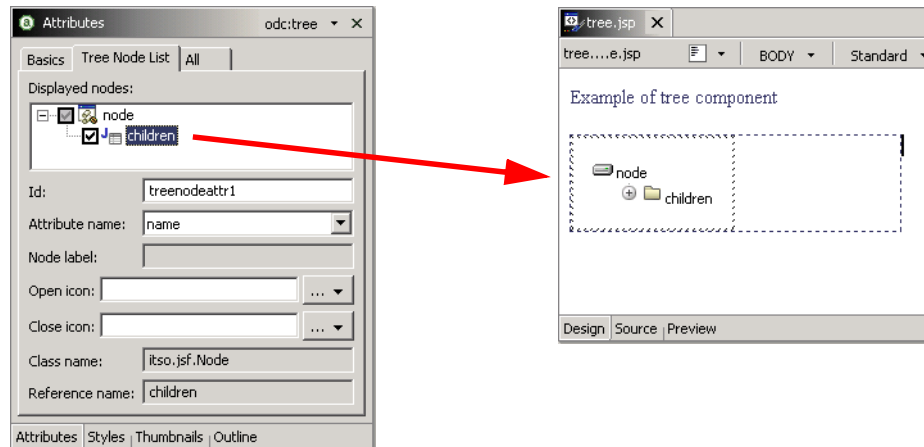


Figure 13-11 Configuring the tree

Events

Table 13-5 shows the events of the `<odc:tree>` component.

Table 13-5 Tree View Quick Edit events

Event	Description
onnodehighlight	This event is called when any node is highlighted by the user.
onnodeselect	This event is called when any node is selected. Node selection must be enabled for this event to get fired.
onnodeunselect	This event is called when any node is unselected. Node selection must be enabled for this event to get fired.

If you select only one node of the tree, you can modify the events for that single component (Table 13-6).

Table 13-6 Tree View node Quick Edit events

Event	Description
onhighlight	This event is called when the specified node is highlighted by the user.

Event	Description
onselect	This event is called when the specified node is selected. Node selection must be enabled.
onunselect	This event is called when the specified node is deselected. Node selection must be enabled.

Web Service: <odc:webService>

This Faces Client component drives a Web service from JavaScript in the browser, allowing you to retrieve data on the fly from a variety of sources without having to send a request to the server.

Attributes

Table 13-7 shows the attributes of the <odc:webService> component.

Table 13-7 Web Service attributes

Attributes	Description
common	id, binding, rendered.
webServiceOperation	The name of the operation in the WSDL that will be executed.
wsAlias	Short name of the Web service.
wsdlURL	The URL of the Web service WSDL file. When typed into a browser the WSDL file should be displayed. For example: http://host/webapp/services/NameOfService/wsdl/Service.wsdl
Input and output parameters	Input and output messages of the Web service must be bound to client data.

Events

The Web Service component has no events.

Usage

To see an example of a Web service, you can import the sample application using *File -> New -> Example -> Web -> Faces -> Portfolio*. Study the Attributes of the Web service (WSDL URL and input/output parameters). Also notice the regular button that is used to invoke the Web service using a JavaScript.

Sample Faces Client application: Data Grid

In this example we use the Data Grid component to create an example of using the Faces Client framework. We bind this component to a simple model to display and update data in the browser and then send the changes to the server.

Creating the user interface

To create the user interface follow these steps:

- ▶ Create a new dynamic Web project called **ItsoJSFClientComponents** using a new EAR project **ItsoJSF13Client**.
- ▶ Under the Java Resources folder create a Java package named **itso.jsf**.
- ▶ Import the files `\labscode\dev-jsfclient\Employee.java` and `\labscode\dev-jsfclient\Department.java` into the package. These files represent a simple example of a department with a few employees. We use these objects as the model for the Data Grid example.
- ▶ Create a Faces JSP file and name it **dataGrid.jsp**. Remember to select *Basic with client-side data caching* in the Model combo box. Click *OK*. Studio prompts you if you want to use WebSphere Application Server 5.1 as the target server for the project. Click *Yes*. Replace the default text in the JSP with **Example of Client Data Grid** and make it a heading 3.
- ▶ The Faces Client Component drawer should now be visible in the palette. Expand the drawer, then drag and drop a **Data Grid** component into the page (Figure 13-12). The Data Grid component has nothing but a text indicating that no data has been bound to it.

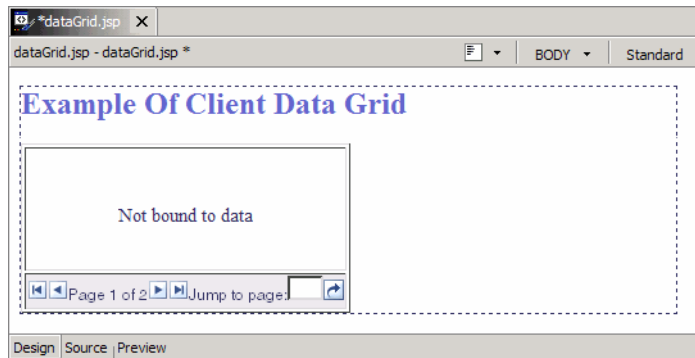


Figure 13-12 Drag and drop a Data Grid component into the page

To bind data to this client component, we have to declare data in the server side and in the client side. The client-side data model follows the server-side data model. In this example, we use the same model in the client as in the server.

- ▶ Using the Page Data view, add a new JavaBean to the page. Name it department and select the `itso.jsf.Department` class for its type. Select *Make this JavaBean reusable* and select *session* for the scope of the bean. Click *Finish*.
- ▶ In the Page Data view, expand department and verify that `employee` is an array of `itso.jsf.Employee`.
- ▶ In the Client Data view select *New -> Client Data*. In the dialog, select the server-side department JavaBean. Enter `clientData` for the id property and `dpt` as the model name (Figure 13-13). Click OK.

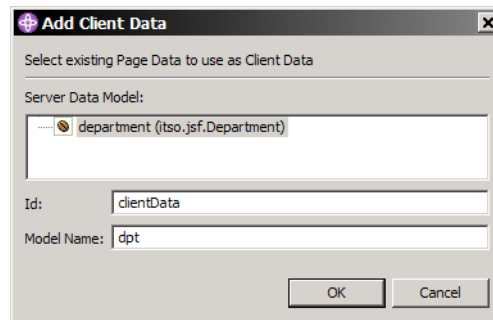


Figure 13-13 Defining the client data

The department model is now represented in the Client Data view and we can bind the object to the Data Grid component:

- ▶ Expand the department object and bind the `employee` array to the Data Grid component (drag and drop). The employee information appears in the Data Grid component.
- ▶ Select the Data Grid component and, using the Attributes view, make these changes:
 - In the Basic tab, enter `dataGrid` as the id of the component. Enter `5` in the items per page property.
 - In the Display tab use the default settings. You can add the selection column and the row number column using this tab.
 - In the Column List tab, reorder the columns into name, surname, gender, age, jobRole, and jobDetails. Change the columns headers to suitable names (Figure 13-14).

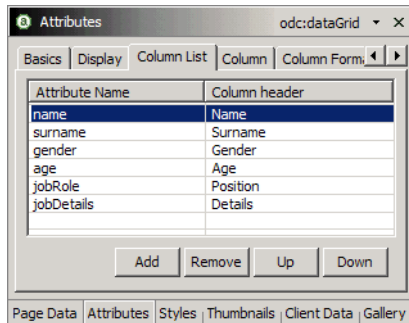


Figure 13-14 Reordering the columns

- In the Column List tab, select the age column. Then, in the Column Format tab select *Number* for the type. Set the number style as *Custom* and the custom format as **0** (integer).
- Save the page (Figure 13-15).

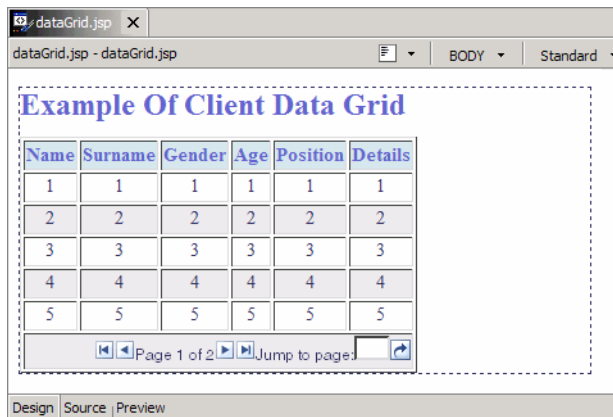


Figure 13-15 Data grid usage example

You can test the page now (Figure 13-16). You get a rich client table with all the data in the example department. You can access all the data and operate over it without round trips to the server. This component also allows you to:

- Highlight the lines by clicking on them.
- Order the data by any of the columns, by clicking in the name of the column.
- Change between the multiple pages of data.

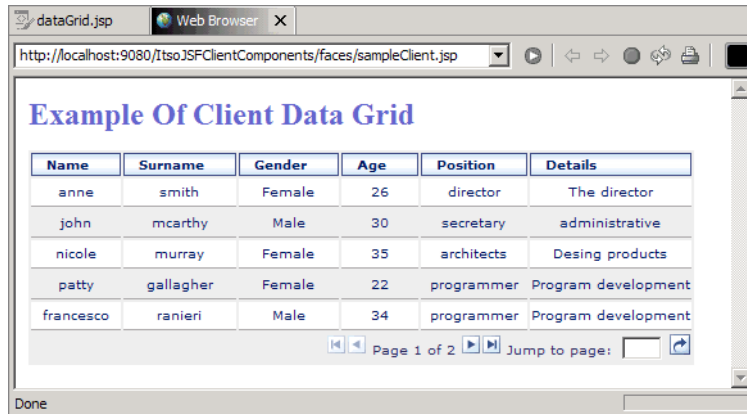


Figure 13-16 Executing the data grid example

Improving the functionality with update

Now we can include more capabilities to the table. We have decided we want to edit the job position and the job details. We also want to include new rows in the table or delete rows from the table.

To add these capabilities to the table, follow these steps:

- ▶ Select the table, and using the Attributes view do the following:
 - In the Basics tab, select *Editable*. A new row and a new column are added to the table. The new row has an *Add New Row* link to add a new row and the new column has a *Delete* link for each row.
 - In the Column List tab, select the `jobRole` column. In the Column tab for the `jobRole` column deselect *Read only*. The column is now represented using input fields and can be edited. Do the same for the `jobDetails` column.

Next we add controls to display and change the name of the department, and send the data to the server for updating of the server model (Figure 13-17).

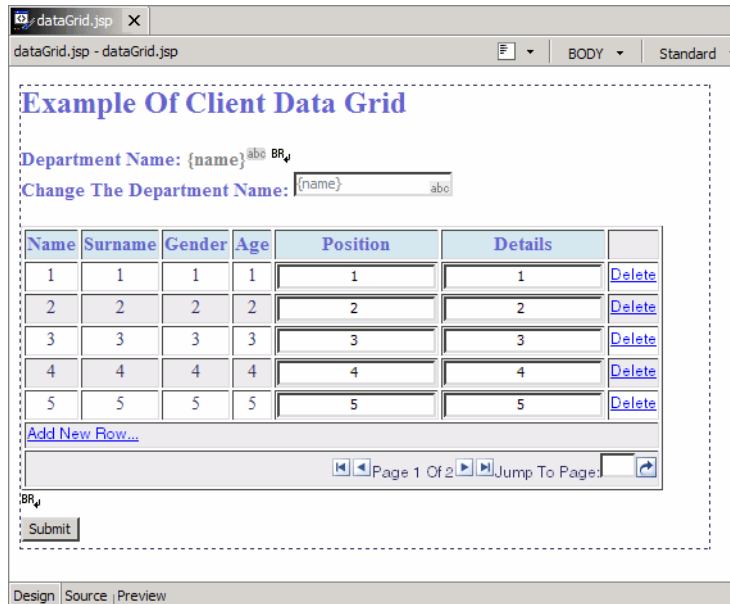


Figure 13-17 Design of the data grid example using the updating options

- ▶ At the top of the table, insert the text **Department name:** and then drag and drop an Output component. Name it **dptname** and bind it to the name attribute of the department object in the client data.
- ▶ Below this insert the text **Change the department name:** and drag and drop an Input component. Name it **departmentName** and bind it to the name property of the department object in the client data as well.
- ▶ Select this two lines (including the components) and, using the context menu select *Insert -> Paragraph -> Heading 4*.
- ▶ Insert a Command Button below the table and set the label to **Submit**.

Implementing server-side update of the client changes

Having completed the design of the page, we have to add all the logic to make updating work. As we mentioned before, the interaction with the user is done without round trips to the server. This means that you can do a large number of operations in the Data Grid component and then update the model in the server in one step.

We use the Submit button to send the changed data to the server and trigger the model update. Only the changed objects are sent to the server. We must process the delta changes in server logic and make the changes in the server model.

To process these changes we have to write a *processor* for each object in the model. This class is responsible for the changes in the model. The JavaBean we have to write must implement the **DiffHandler** interface. This interface has only one method with the signature:

```
public void handle(DiffInfo diff) throws Exception
```

The `DiffInfo` parameter holds all the information about the changes.

Creating the difference handler for the department object

To create a new handler for the department object, follow these steps:

- ▶ Select the `itso.jsf` package and select *New -> Class*.
- ▶ Enter **DepartmentDiffHandler** in the Name field. Add a new interface to implement. In the pop-up window, enter **diff** to find the `DiffHandler` interface and click *Add*. Click *OK*. Select *Inherited abstract methods* and click *Finish*.
- ▶ Add an attribute named **dpt** of type `Department`. The attribute represents the department object of the model. Generate setters and getters.
- ▶ Add a constructor for the class to populate the `dpt` attribute:

```
public DepartmentDiffHandler(Department d) {  
    dpt = d;  
}
```

- ▶ Implement the `handle` method to update the department object. Add imports as appropriate. The only property we have to update is the name of the department (`\labscode\dev-jsfclient\DepartmentDiffHandler.java`):

```
public void handle(DiffInfo diff) throws Exception {  
    if (diff.getCrud() != 'U') {  
        System.out.println("Only updates are allowed!!");  
        return;  
    }  
    //The only thing we can update is the name of the department.  
    EObject eo = diff.getCurrent();  
    String name = (String) EMFHelper.getEFeatureValue(eo, "name");  
    dpt.setName(name);  
}
```

Using the `DiffInfo` object we can retrieve the type of change: update, create or delete. Then we retrieve the current value of the name attribute and apply it to the department object. We have updated the server model with the value in the `DiffInfo` object.

Creating the difference handler for the employee object

The difference handler for the employee object is similar, but we must update more data. And we have to check if we are updating, deleting, or creating employees. To create this handler follow these steps:

- ▶ Create an **EmployeeDiffHandler** class in the `itso.jsf` package. Add the `DiffHandler` interface, select *Inherited abstract methods*, and click *Finish*.
- ▶ Add an attribute named `dpt` of type `Department`. This attribute represent the department object of the model, and through it we are able to access all the employees. Generate setters and getters.
- ▶ Add a constructor for the class to populate the `dpt` attribute:

```
public EmployeeDiffHandler(Department d) {  
    dpt = d;  
}
```

- ▶ Implement the `handle` method to update the employees in the department object. We get the information from the `DiffInfo` object and depending on the type of operation, we perform the update in the model. The `handle` method is shown here (`\labscode\dev-client\EmployeeDiffHandler.java`):

```
public void handle(DiffInfo diff) throws Exception{  
    //Take all the employee information  
    EObject eo = diff.getCurrent();  
    String name = (String)EMFHelper.getEFeatureValue(eo,"name");  
    String surname = (String)EMFHelper.getEFeatureValue(eo,"surname");  
    String gender = (String)EMFHelper.getEFeatureValue(eo,"gender");  
    Integer age = (Integer)EMFHelper.getEFeatureValue(eo,"age");  
    String jobRole = (String)EMFHelper.getEFeatureValue(eo,"jobRole");  
    String jobDetails = (String)EMFHelper.getEFeatureValue(eo,"jobDetails");  
  
    if (diff.getCrud() == 'U'){  
        //Update the model. We must find the employee to update.  
        boolean found = false;  
        Employee toUpdate = null;  
        int index = 0;  
        while (!found){  
            toUpdate = (Employee) dpt.getEmployee()[index];  
            if (toUpdate.getName().equals(name)){  
                found = true;  
            }  
            index++;  
        }  
        if (found){  
            ((Employee)dpt.getEmployee()[index-1]).setJobRole(jobRole);  
            ((Employee)dpt.getEmployee()[index-1]).setJobDetails(jobDetails);  
        } else {  
            System.out.println("Error. Cannot update an inexistent employee.");  
        }  
    }  
    else if (diff.getCrud() == 'C'){
```

```

        //Create a new employee and attach it to the departemnt.
        Employee newEmployee = new Employee(name,surname,gender,
            age.intValue(),jobRole,jobDetails);
        Employee [] newDepartment = new Employee[dpt.getEmployee().length + 1];
        System.arraycopy(dpt.getEmployee(),0,
            newDepartment,0,dpt.getEmployee().length);
        newDepartment[dpt.getEmployee().length] = newEmployee;
        dpt.setEmployee(newDepartment);
    }
    else if (diff.getCrud() == 'D'){
        //Delete the employee. We must find and delete the employee
        boolean found = false;
        Employee toUpdate = null;
        int index = 0;
        while (!found){
            toUpdate = (Employee) dpt.getEmployee()[index];
            if (toUpdate.getName().equals(name)){
                found = true;
            }
            index++;
        }
        if (found){
            //Remove the element
            Employee [] newDepartment = new Employee[dpt.getEmployee().length-1];
            System.arraycopy(dpt.getEmployee(),0,newDepartment,0, index-1);
            System.arraycopy(dpt.getEmployee(),index,newDepartment,
                index,(dpt.getEmployee().length - index));
            dpt.setEmployee(newDepartment);
        } else {
            System.out.println("Error. Cannot delete an inexistent employee.");
        }
    }
}
}

```

Triggering the update

The first thing we have to do is send the delta data to the server, using the JavaScript library. In the server, we have to process all the differences between the client model and the server model. This is done using the two difference handlers that we implemented, and using a class called **DiffTraverser**. To trigger the process, the handlers must be added to this object and the traverse function must be executed.

To implement the server-side processing of the changes follow these steps:

- Select the Submit button in the dataGrid.jsp. In the Quick Edit view, select the `onClick` event and add this code:

```
ODCRegistry.saveAllToForm(thisObj);
```

This code takes all the data from the table, format it, and store it in the form where the Submit button is located. The data is sent as a request parameter.

- Select the `itso.jsp` package and create a new class named **DiffProcessor**.
- Create a static method named `synch` to trigger the updates in the server (`\labscode\dev-jsfclient\DiffProcessor.java`):

```
public static boolean synch(Department d, String diffString,
    String dptName, String empName){
    try {
        DiffTraverser dt = new DiffTraverser(dptName,diffString);
        dt.addHandler(dptName, new DepartmentDiffHandler(d));
        dt.addHandler(empName, new EmployeeDiffHandler(d));
        dt.traverse();
        return true;
    } catch(Exception ex) {
        ex.printStackTrace();
    }
    return false;
}
```

The parameters for this function are the department model, a string representing the differences (the changes in the client side), the type of the department object in the client data, and the type of the employee object in the client data. These values are used to create the handlers for the specified types and add them to the `DiffTraverser` object.

In this method a new `DiffTraverser` object is created with the name of the parent object and the string representing the differences. We add the two handlers (one for the department changes and one for the employee changes) and execute the `traverse` method.

- This method is called in the server as the action logic of the Submit button. Select the Submit button in the `dataGrid.jsp` and in the Quick Edit view enter the action logic (`\labscode\dev-jsfclient\Submit.txt`):

```
String diff = getClientData().getDiffStr();
System.out.println(diff);
boolean dpt_diff_processed = DiffProcessor.synch(getDepartment(),
    diff, "department_Department", "department_Employee");
if (dpt_diff_processed){
    System.out.println("Information successfully processed.");
}
else{
    System.out.println("Error processing the information.");
}
return "";
```

This code executes in the server. All the client data has been sent to the server in a string that represents the differences between the original data and the changes made by the user. We call the static method `synch` to trigger the handling of the differences.

Test the application

Now save the page and test it (Figure 13-18).

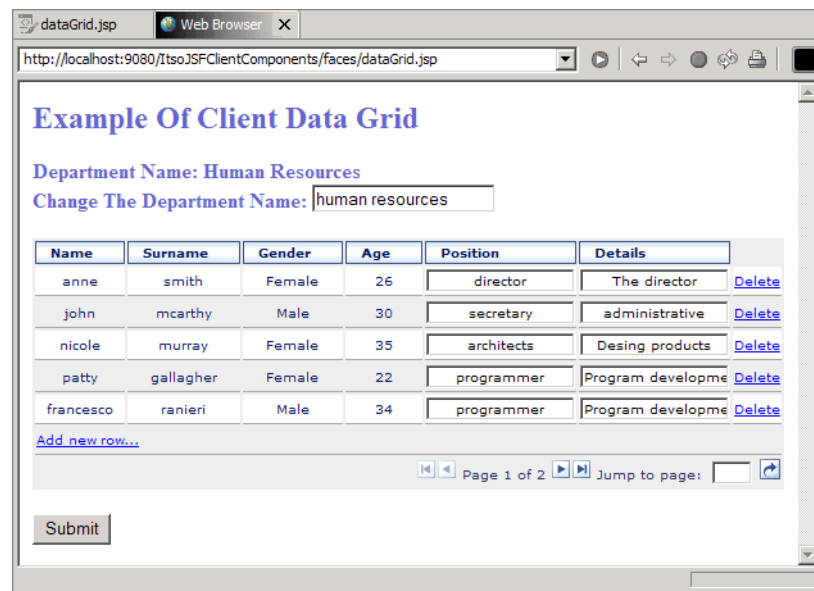


Figure 13-18 The data grid example running

For example, try these changes (Figure 13-19):

- ▶ Change the name of the department to **Administration** and notice how the label is immediately updated when the department name input field loses focus.
- ▶ Now change the values for the job position or job details of any employee, for example change *ueli* into a **staff** position with details **project leader**.
- ▶ Add an employee by selecting *Add new row*. A row of input fields open to enter the new employee information Click *Accept* to make the entry permanent (in the client). The new employee is added to the end of the list.

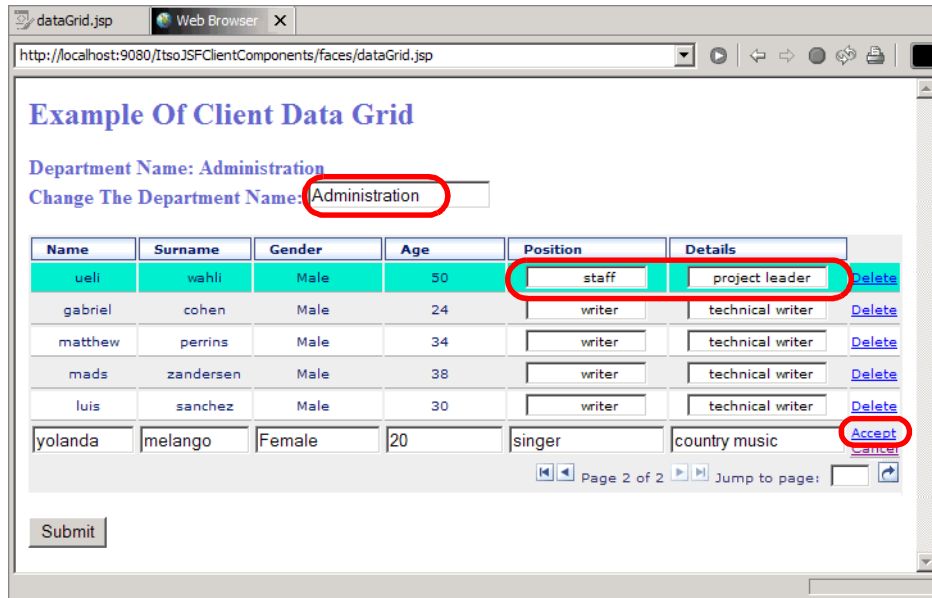


Figure 13-19 Data grid example running

- Click *Submit* and all the changes are sent to the server. Notice the test output in the Console where you can see the current and original data:

```
<Diff ....>
<Current>
  <department_Department ODCstatus="U" xmi:id="_2" ODCid="_2"
    name="Administration" employee="_8 genid_25">
    <employee ODCstatus="C" xmi:id="genid_25" ODCid="genid_25"
      surname="melango" name="Yolanda" jobRole="singer"
      jobDetails="country music" gender="Female" age="20" />
    <employee ODCstatus="U" xmi:id="_8" ODCid="_8"
      surname="wahli" name="ueli" jobRole="staff"
      jobDetails="project leader" gender="Male" age="50" />
  </department_Department>
</Current>
<Original>
  <department_Department xmi:id="_2" ODCid="_2"
    name="human resources">
    <employee xmi:id="_8" ODCid="_8" surname="wahli"
      name="ueli" jobRole="writer" jobDetails="technical writer"
      gender="Male" age="50" />
  </department_Department>
</Original>
</Diff>
```

- Click the column headings to sort the data.

- ▶ Select *Delete* for an employee and it is removed from the data grid. Click *Submit* to process the delete at the server.

important: Delete is not working—the update is not sent to the server. It is a bug that will be resolved in the future.

Sample Faces Client application: Graph

In this example we use the Graph component to display charts based on the ages of the employees from the previous grid example.

Creating the user interface

The instructions here are very short, but should be enough to create the sample:

- ▶ Create a **graph.jsp**, be sure to select the option *Basic with client-side data caching*.
- ▶ Change the default title to Example of Graph Component (heading 2).
- ▶ In the Page Data view select *New -> Java Bean* and add the existing department bean.
- ▶ In the Client Data view add the department bean.
- ▶ Drop a Graph component into the JSP, then bind it to the employee array under the department.

Configuring the graph

The graphic display is configured through the Attributes view of the Graph component (Figure 13-20).

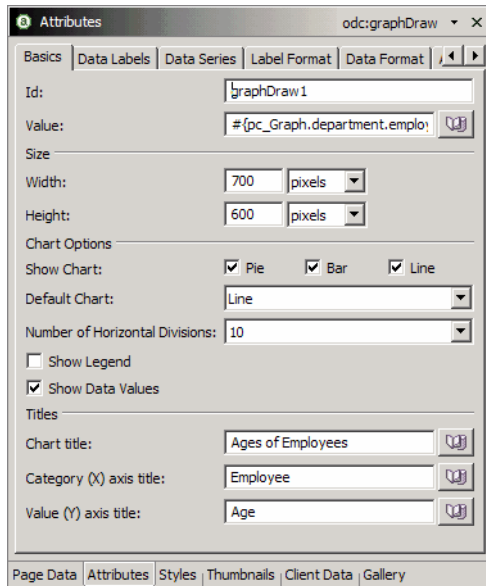


Figure 13-20 Attributes of the graph

- ▶ Basics tab:
 - Change the size to **700 x 600** pixels.
 - Select **Line** as the default chart.
 - Set **10** for the horizontal divisions.
 - **Deselect** *Show Legend*.
 - Select *Show Data Values*.
 - Set the chart title to **Ages of Employees**.
 - Set the axis titles to **Employee** (x) and **Age** (y).
- ▶ Data Labels tab: Select the **name** attribute.
- ▶ Data Series tab: Click *Add*. Set the name to **Age** and the attribute to **age**.
- ▶ Label Format tab: Leave as **String** (for the employees).
- ▶ Data Format tab: Select **Number** for type, **Custom** for style and **0** for format.
- ▶ The design of the graph is shown in Figure 13-21.

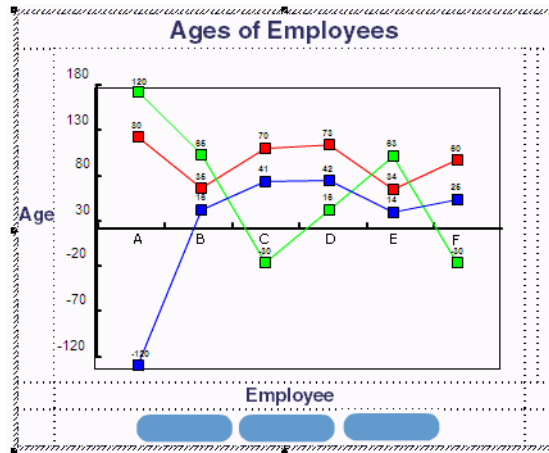


Figure 13-21 Design of the graph

Running the graph example

When we run the example the employee ages are displayed using a line chart (Figure 13-22).

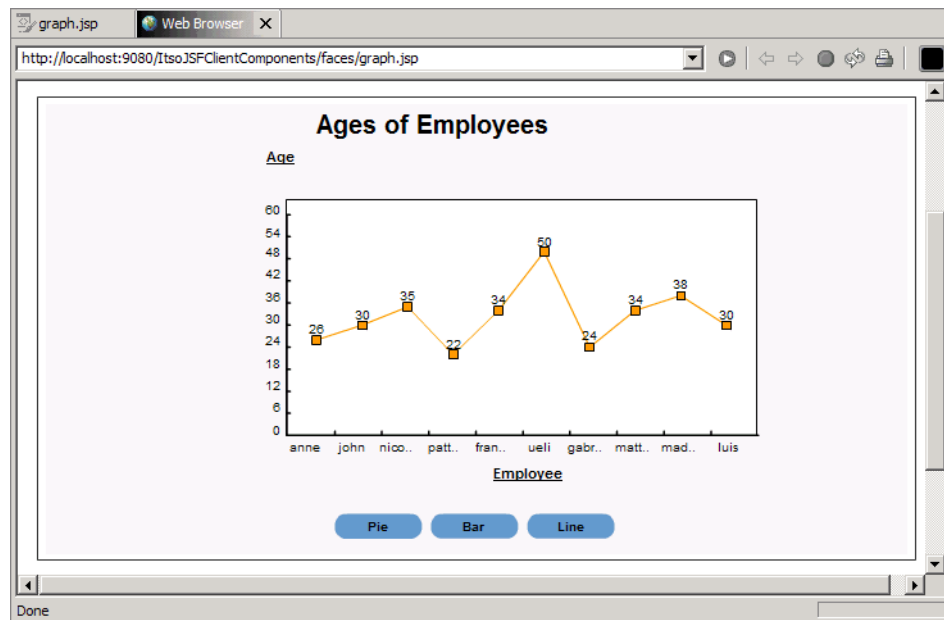


Figure 13-22 Running the graph example

In the context menu of the graph, select *Zoom In*. Using the *Hand* icon you can drag the magnified section over the graph (Figure 13-23).

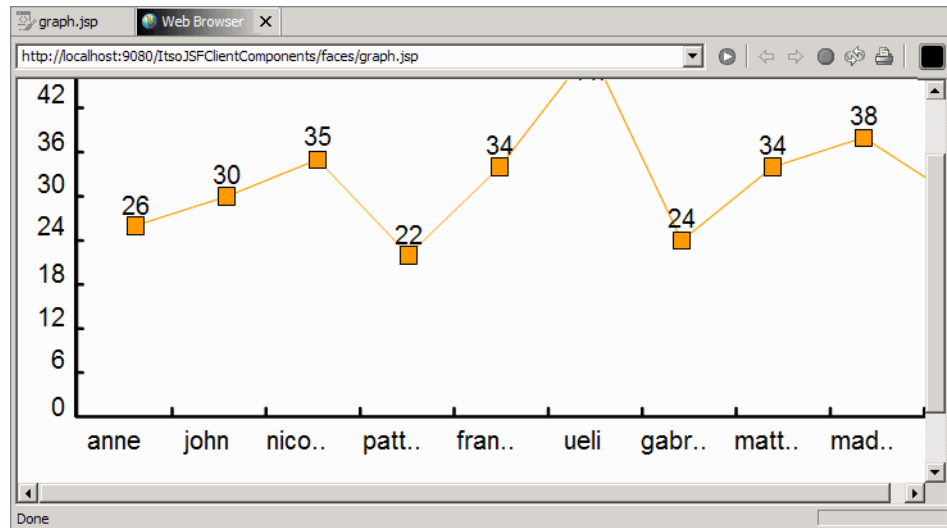
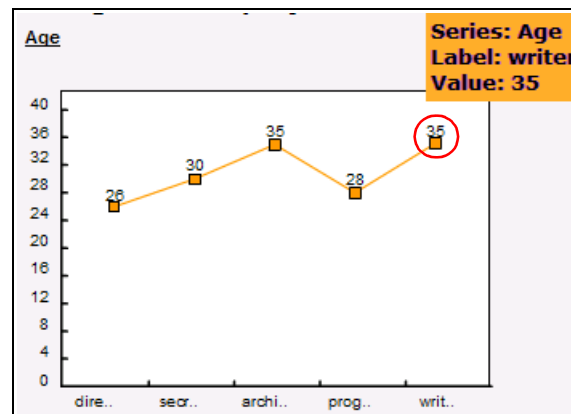


Figure 13-23 Running the graph example (zoom in)

Using the grouping functions

Now let's display the employee ages by job. In the Attributes view, Data Labels tab, select the **jobRole** attribute. In the Data Series tab, select *Group values*. For the age series select **AVG** as group operation. Save and rerun the example. The ages are grouped by jobs and the average is calculated (Figure 13-24).



Details are displayed when you point to a graph point.

Figure 13-24 Average employee ages grouped by job

Other options

Play with the graph parameters and the execution functionality:

- ▶ Try out other grouping functions, for example, `COUNT`, to count employees by job.
- ▶ Click *Pie* or *Bar* to display the result as a pie or bar chart.
- ▶ In this example we cannot draw multiple series in one graph because we only have one numeric field. When using multiple series the values should be in the same range. Also the legend should be selected to label the series by color.

Sample Faces Client application: Tree View

In this example we use the Tree View component to display charts based on the ages of the employees from the previous grid example.

You can easily construct this example (**tree.jsp**) using the same department bean. Define the department bean in the Page Data view and in the Client Data view, then add the bean to the JSP.

Figure 13-25 shows the design and how it looks at execution time. One of the selections in the Attributes view, Basics tab, is *Enable node selection*, which adds check boxes to each node. The selected nodes can then be processed using logic.

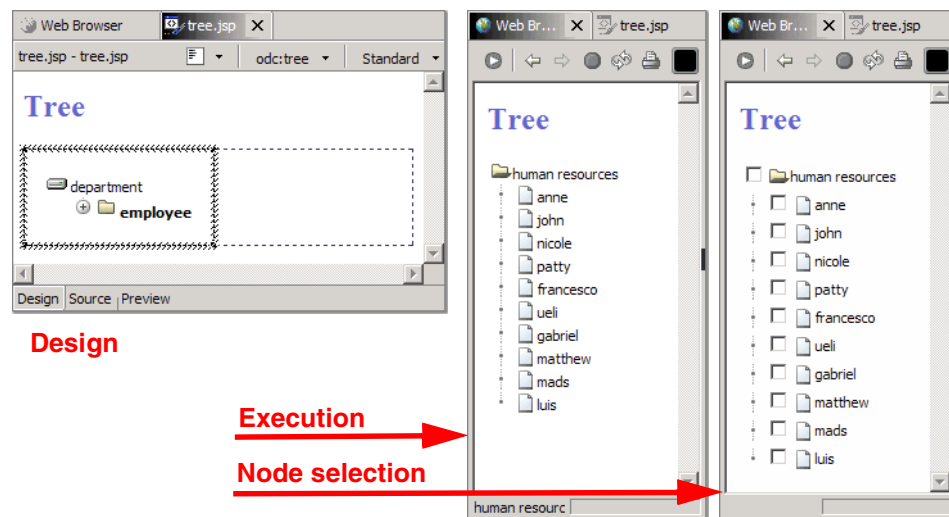


Figure 13-25 Tree View component: design and execution

Processing tree operations

Here is a small example on how to process the tree when a node is highlighted (only one can be) and when nodes are selected (multiple):

- ▶ In the Page Data view define the customer (itso.jsf.Customer) as a managed bean (no scope).
- ▶ Add the customer to the Client Data view.
- ▶ Add an Output component with the text **Highlighted employee:.**
- ▶ Add an Input component, set the id as lastname, and bind it to the customer.lastName.
- ▶ Add a Submit button (Figure 13-26).

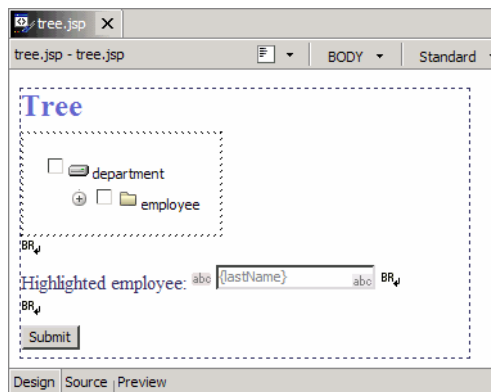


Figure 13-26 Tree view design

We want to pass the highlighted employee's last name to the server. We also pass information about selected employees to the server. This is done using JavaScript logic:

- ▶ Select the employee icon. In the Quick Edit view for the onhighlight event enter a code snippet by selecting *Code Snippet* -> *Bind selected object to another object* (context).

In the dialog that opens, select Employee.surname as source and bind to customer.lastName as target (`{pc_Tree.customer.lastName}`). Click *OK* to insert the event code.

When we highlight an employee the surname is moved into the customer object and displayed in the input field.

- We want to submit all the changes to the tree as a form parameter. Select the Submit button and in the Quick Edit view for the `onClick` event enter:

```
ODCRegistry.saveAllToForm(thisObj);
return "true";
```

- When Tree View component values are submitted to the server all attributes that are primary keys are included. In the Client Data view select the department and *Configure*. Expand the tree, select each employee attribute and deselect *Primary*, except for the surname.
- Finally we implement the action logic. Select the Submit button and in the Quick Edit view add the logic:

```
log("Name="+getRequestParam().get("form1:lastname"));
log("Treestate="+getTree1().getStateString());
return "";
```

Running the Tree View

Run the `tree.jsp` on the server. The department tree is displayed. Highlight an employee (click on the name) and the lastname appears in the input field. Select two employees and click *Submit* (Figure 13-27).

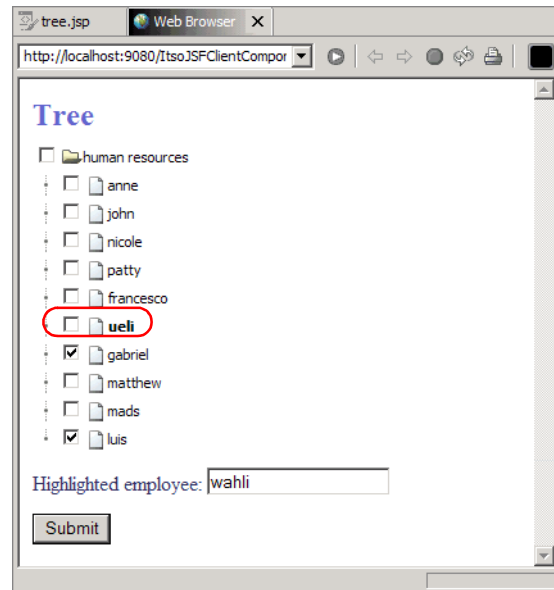


Figure 13-27 Tree View component in action

Look at the Console log. The employee lastname is displayed. In addition a complex state string of the tree is displayed:

```
Treestate=department_Departmenthuman resources:department_Employeewahli:  
department_Employeecohen,department_Employeesanchez
```

There are three sections, separated by colon characters:

- ▶ Department node with name (human resources)
- ▶ Highlighted employee (wahl i)
- ▶ Selected employees (cohen and sanchez)

For each node the primary attributes are concatenated to the node name. This requires parsing of the string and is not very user-friendly, especially if there is more than one attribute. A better way is to write more JavaScript and store information about selected components in other controls.

Note. When you rerun the same page the employees passed to the action logic do not match the selection. This is a bug and will be fixed.

Sample Faces Client application: Web Service

In this example we implement a simple JSF page with client data that invokes an existing Web service. We reuse the `AccountListingSQL` Web service that we used in Chapter 7, “JSF and Web services” on page 277.

Adding the Web service project

If you implemented the example in “Example Web service application” on page 282 then you already have the **ItsoBankingWebService** project in the workspace:

- ▶ If the `ItsoBankingWebService` exists in the workspace, open the EAR deployment descriptor of the `ItsoJSF13Client` enterprise application and on the Module page click *Add* and select the `ItsoBankingWebService` project.
- ▶ Otherwise create a dynamic Web project named `ItsoBankingWebService`. Select *Configure advanced options* and click *Next*. Set the EAR project to `ItsoJSF13Client`. Click *Finish* to exit the wizard. Import an existing Web service. Select the `ItsoBankingWebService` project and *Import*, then select *WAR file*. Click *Browse* to locate the WAR file
`\labscode\dev-jsfwebserv\ItsoBankingWebService.war`.

The target project should be set to `ItsoBankingWebService`. Select *Overwrite existing resources without warning* and click *Finish*.

- ▶ Associate the enterprise application (ItsoJSF13Client) with a server (JSFServer).

Copying the JavaBeans required for Page Data

We use the `getCustomer` method of the `AccountListingSQL` Web service for this example. This method requires input and output in our client application:

- ▶ In the `ItsoJSFClientComponents` project create a package named `itso.bean`.
- ▶ Copy the `Customer` and `CustomerInput` classes from the `itso.bean` package in the `ItsoBankingWebService` project to the `ItsoJSFClientComponents` project (same package).

Creating the JSP to run the Web service

In the `ItsoJSFClientComponents` project create a JSF page named `webService.jsp`:

- ▶ Change the heading to `Web Service From Client`.
- ▶ In the Page Data view, define two managed beans with the names `customer` (`Customer` class) and `customerInput` (`CustomerInput` class). Do not make the beans reusable.
- ▶ In the Client Data view, add the same two beans with the IDs `clientData1` and `clientData2`.
- ▶ Design the page layout as shown in Figure 13-28.

Web Service From Client

Enter a customer ID:

Web Service Result

Customer number	{id}
Title	{title}
First name	{firstName}
Last name	{lastName}

Note: The push button must be a regular HTML component, not the IBM extension JSF Command - Button:

```
<input type="button"
      name="runWebservice"
      value="Run Web Service" />
```

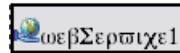
Figure 13-28 Web service invocation design

- ▶ Bind the input field to the client data `customerInput.customerId` field. This creates the binding `@{pc_WebService.customerInput.customerId}`.
- ▶ Bind the four output fields to the four client data `customer` fields.

Adding the Web Service component

To invoke the Web service we add a Web Service component from the Faces Client Components drawer of the palette:

- ▶ Drop a Web Service component under the heading. It may appear in “greek” if your system does have the font installed that was selected by the developer of the component (this will be fixed):



This spells out `webService1` in the Symbol font.

- ▶ The component has the id `webService1`. In the Attributes view enter any Alias, the URL, and the Operation (Figure 13-29):

URL: `http://localhost:9080/ItsoBankingWebService/services/AccountListingSQL/wsd1/AccountListingSQL.wsdl`
Operation: `getCustomer`

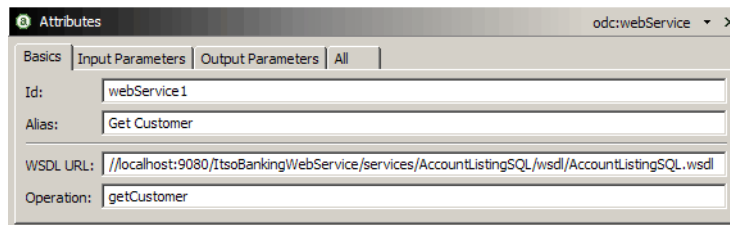
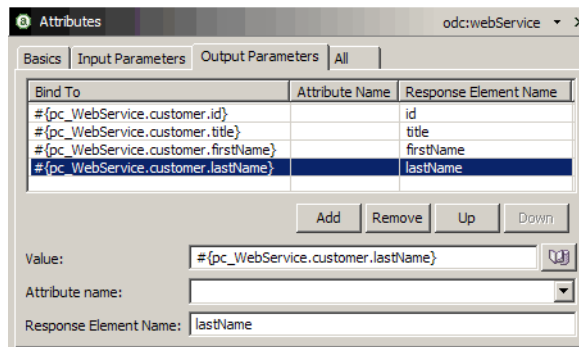


Figure 13-29 Web Service client component attributes

- ▶ On the Input Parameters tab, click *Add*, and bind the parameter to `customerInput.customerId`:

`{pc_WebService.customerInput.customerId}`

- ▶ On the Output Parameters tab, click *Add* four times and bind the four parameters to the four fields of the customer bean (Figure 13-30).



The Response Element Name is the name of the tag in the SOAP response message.

Figure 13-30 Web Service client component parameter binding

Invoking the Web service

To invoke the Web service, select the push button and in the Quick Edit view, select the *onclick* event and add the logic to execute the Web service (using the id of the component):

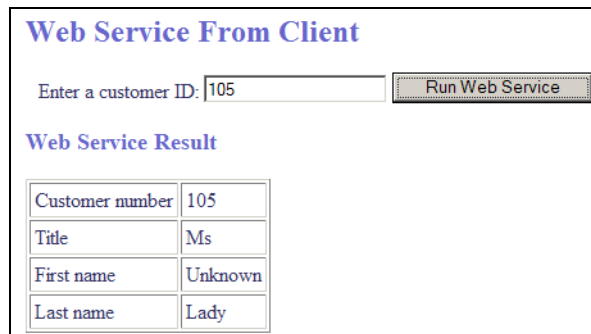
```
executeWebService('webService1');
```

Be sure to use a regular HTML button:

```
<input type="button" name="runWebService" value="Run Web Service"
      onclick="executeWebService('webService1');" />
```

Running the Web service client application

Start the JSFServer with the ItsoJSF13Client enterprise application. Then run the `webService.jsp`. Enter a customer ID and click *Run Web Service*. The customer data is displayed (Figure 13-31). Notice the messages of the Web service invocation in the Console.



Web Service Result	
Customer number	105
Title	Ms
First name	Unknown
Last name	Lady

Figure 13-31 Web Service component run

Complete solution

We provide the complete solution for this exercise in the additional material in the file `\labscode\solution\ItsoJSF13Client.ear`. See “Importing a solution enterprise application” on page 548 for instructions.



Extending the JSF framework

This chapter describes how to develop a custom JSF component and a custom converter using WebSphere Studio Application Developer.

One of the most powerful features of JSF is its ability to be extended. If you have found that the current selection of components does not meet all the demands of your application, you can build a new renderer or full JSF component using WebSphere Studio. This chapter looks at what is required to do this and explains the steps using an example scenario.

This chapter also explains how WebSphere Studio can be extended to support the integration of these components within the Palette view of the Web perspective. Our example enables component developers to understand how to include their custom components within Studio and enables developers of JSF application to get the productivity benefits of using JSF within Studio.

Custom components

JavaServer Faces is a framework which was designed with flexibility in mind. Many aspects of the framework can be replaced, modified, or extended as required. These reusable components can be designed to supply tailored presentation encoding and decoding, generation of new types of Faces events, custom validation, and binding to new types of model data.

Component authors are responsible for creating these extensions of JSF which can be delivered to and exploited by others. Custom components may be required in a variety of circumstances, each of which dictates the nature of the component author:

- ▶ An application developer may author a new component that binds to a type of data object unique to their business environment.
- ▶ A page author might write a custom component to fill a requirement for a special control on a particular page.
- ▶ An internal development team may author custom components to provide a unified look and feel across an entire organization's applications.
- ▶ A team of open source software developers may create portable and robust component libraries for use across the entire industry.

As demonstrated by these examples, the size and nature of the audience for JSF extensions can vary greatly. These extensions may fill an individual requirement of a specific project, or attempt to provide a user interface implementation for an entire market space.

The extension points in JSF are designed such that the implementing components can be packaged and reused across multiple applications. Developers can then pick and choose from repositories of self-contained components, using them as building blocks in the process of rapid application development.

Overview

This chapter is not going to give you a detailed explanation of how to develop JSF components. Instead, this chapter explains how to use WebSphere Studio to develop custom components that can be used in J2EE applications. WebSphere Studio is a powerful Java development environment and is has all the necessary tools to make JSF component development easy and intuitive.

This chapter covers all of the steps to develop a custom component:

- ▶ How to create a dynamic Web project to develop custom components

- ▶ How to develop a component that contains the following pieces:
 - A component object
 - UI component tag
 - A custom renderer
- ▶ How to test a custom component within Studio.
- ▶ How to package a component into a distributable JAR file.

JSF supports a wide variety of customizations. You can develop the following types of JSF components that can be used by JSF developers:

- ▶ Custom validators—Used for validation of input data that is not covered by standard validators (see “Implementing a validator” on page 78)
- ▶ Custom converters—Use for data conversions not covered by standard converters (see “Custom converters” on page 506)
- ▶ Custom components—Used to manage an underlying data model or interface
- ▶ Custom renderers—Can be mapped to a component to render a different user interface or behavior
- ▶ Custom UI component tags—Used to integrate JSF components into JSPs

The custom component that is developed in this chapter is built around a component class that has a separate renderer. This component can then be used in Web application development using the same techniques that have been previously covered in this redbook.

We cover in this chapter how you can extend the Web perspective Palette view and add a custom component that can be dragged into the Page Designer. The ability to do this gives custom component developers many of the same productivity benefits within Studio that can be achieved with the IBM JSF components that are included with WebSphere Studio.

Scenario

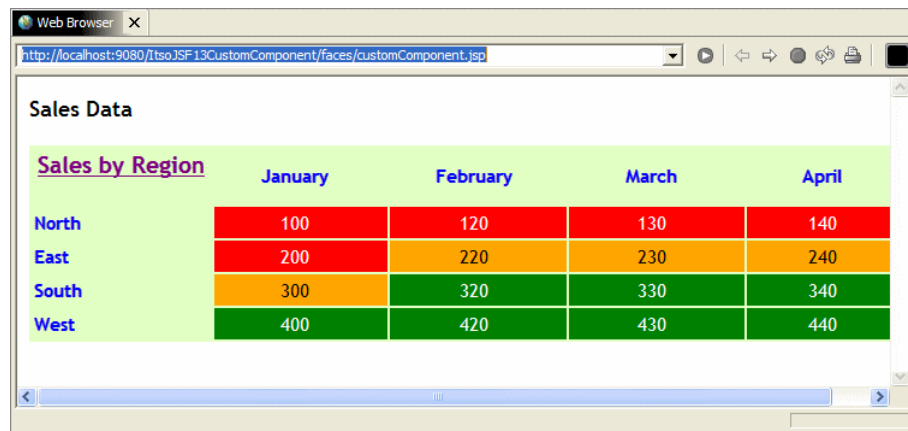
To make the development of the custom component more realistic and meaningful, we are going to use the scenario of a Web project that displays sales data. This data shows high, medium, and low performance of the sales regions. The sales teams have to be able to view this data using the company’s intranet.

This data could be displayed using existing JSP technology, but the development team thinks that this component will be used in a wide variety of sales applications. Using JSPs would mean that a lot of code would be created and it would be hard to maintain. So they have decided to create it as a custom JSF component. This allows the core code to be developed once and the component can be reused in a wide variety of applications, allowing the development team to

focus on delivering business function rather than maintaining complex JSP pages.

The component is populated from data held within a managed bean, which is constructed from an external data source. For the purposes of this scenario we use a managed bean with static test data. The custom JSF component requires a simple and easy to use interface so that Web designers understand how to use the component.

Figure 14-1 displays the layout of the component when it is rendered into a JSF page. You can see that the high, medium, and low styling is applied to the data based on the threshold limits that can be set at runtime.



The screenshot shows a web browser window with the URL `http://localhost:9080/tsoJSF13CustomComponent/faces/customComponent.jsp`. The page displays a table titled "Sales Data". The table has a header row with the title "Sales by Region" and four columns: "January", "February", "March", and "April". The rows represent different regions: "North", "East", "South", and "West". The data values are color-coded: red for high values, orange for medium, and green for low.

<u>Sales by Region</u>	January	February	March	April
North	100	120	130	140
East	200	220	230	240
South	300	320	330	340
West	400	420	430	440

Figure 14-1 Component being used in the sales application

Note: A future enhancement is to allow the row labels and column labels to be selectable and an event triggered back to the server. The underlying data can then be changed and a more interactive component be built. For example, you could click on a region and see the details for the region. You could treat the title as a home link to display the top most layer of data. These are just thoughts—maybe we will implement such extensions in a real application.

Designing the component

The development team decided to call the component a **Traffic Light Grid**, because it uses red, amber, and green colors like a traffic light, and its data is displayed in a grid layout.

To start the development of this custom component we require both a visual design (Figure 14-1) and a tag design (Example 14-1). The design approach taken with the `TrafficLightGrid` component was to develop an HTML sample and to design a tag that allows the key data to be supplied to the component.

Example 14-1 TrafficLightGrid tag design

```
<itso:trafficLightGrid
  id="tlight1"
  title="#{griddata.title}"
  data="#{griddata.data}"
  rowLabels="#{griddata.rowLabels}"
  colLabels="#{griddata.colLabels}"
  min="220"
  max="300">
</itso:trafficLightGrid>
```

The `rowLabels` and `colLabels` attributes accept a `java.lang.ArrayList` containing string objects. The `data` attribute accepts a `java.lang.ArrayList` object that contains `java.lang.ArrayList` objects that represent each row of the table.

The tag design helps you develop the JSF tag handler and the underlying component model. You can see from this design which attributes of the tag are used with value binding elements. This requires extra code for these attributes within the component and component tag.

The other parts of the component design are the name space definition and the component name. In this example we have used **itso** as the name space for the component and **trafficLightGrid** as the component name. We want to associate the component with the vendor developer, so we use http://www.ibm.com/jsf/itso_extended for the taglib URI. The namespace can be used within a page and it is consistent with other component and tag libraries being developed.

Now we can start working through what is required to develop this custom component.

Web project structure

The first task required is to create a dynamic Web project. We do not use this project to develop a typical J2EE Web application, we use it to host the development of the custom component. Because the project is a Web project that supports JSF pages, we can use it as a convenient place to test the custom component as we write it. The following steps take you through the process of setting up the Web project:

- ▶ Create a dynamic Web project called **ItsoJSFCustomComponent** and associate this project with an EAR project called **ItsoJSF14Ext**.
- ▶ Create a Faces JSP page under the WebContent folder and name it **customComponent.jsp**. We will use this page later for the testing of the custom component.
- ▶ While we are in the Web project we can import the style sheet that is used by the custom component and place it in the WebContent\theme folder. Import the file \labscode\dev-ext\custom\trafficLight.css.
- ▶ Study the CSS file (Example 14-2). It contains the simple CSS formatting that is used for the header labels, row labels, and the different threshold styles for each cell.

Example 14-2 Traffic light style sheet for the custom component

```
.title {
    color: black;
    font-size: 14pt;
    font-style: normal;
    font-weight: bold;
    size:14pt;
    color:black; }
.columnLabel {
    color: blue;
    font-size: 11pt;
    font-style: normal;
    font-weight: bold; }
.rowLabel {
    color: blue;
    font-size: 11pt;
    font-style: normal;
    font-weight: bold; }
.minCell {
    background-color: red;
    color: white; }
.maxCell {
    background-color: green;
    color: white }
.normalCell {
    background-color: orange }
```

Note: The IBM custom components automatically include the style sheets in the projects as they are added to the project. For expediency, we are adding the style sheet manually in this example.

You now have a Web project and are ready to start the development of the custom component.

Development steps

Table 14-1 shows the elements that we are developing for the custom component.

Table 14-1 List of items that have to be developed for a custom JSF component

Stage	Description
Tag Library Definition (TLD)	JSP tag library definition that is used to validate the JSF tags within a JSP page
Tag component	Java code to interface to the TLD
UI component	Java code that interfaces with the model and maintains presentation state
Component renderer	Java code that is used by the UIComponent to render its user interface
faces-config.xml	Entries in the faces-config.xml file to register the custom component and the renderer
Ant build script	Ant script used to package the custom component ready for distribution
Testing	Creating a test page for testing the custom component
Exporting	Export the finished package JAR ready for use by other developers or customers

There is no strict order in which you have to develop the first elements of the custom component. We have chosen to start with the JSP tag library interface for the custom component. This helps the developer focus on the interface that has to be developed.

Tip: The main reason for using a Web project for building the custom component is that when you add a Faces JSP page to it, all the necessary JAR files and dependencies are added to the project. When you develop the supporting Java classes you will not receive any errors.

JSF tag library

Each custom component in JSF is exposed through a JSP tag library. This allows the component to be easily added to JSP pages and using the same context and notation that are used for J2EE Web development.

In JSF you reference a Java module that subclasses `UIComponentTag` from the tag library. The `UIComponentTag` is used to collect data and manage properties for the UI component as it is placed within a JSF page.

The TLD definition is built upon the JSP 1.2 specification and all the rules and development guidelines apply. For our design we implement the Traffic Light Grid TLD.

The following steps take you through the process:

- Expand the `WebContent/WEB-INF` folder and create a folder called **tld**. Then import the `trafficLightGrid.tld` file (Example 14-3) into the new folder (`\labscode\dev-ext\custom\trafficLightGrid.tld`).

Example 14-3 Traffic Light Grid TLD definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.2</jspversion>
  <shortname>itso</shortname>
  <uri>http://www.ibm.com/jsf/itso_extended</uri>
  <info>Displays a grid of data and draws the cells in a traffic light color
  depending on min max threshold limits
  </info>
  <tag>
    <name>trafficLightGrid</name>
    <tagclass>itso.jsf.custom.TrafficLightGridTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>Tag Library Definition for TrafficLightGrid custom component</info>
    <attribute>
      <name>id</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>title</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```



```

        <name>data</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>rowLabels</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>colLabels</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>max</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>min</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
</taglib>

```

You can see some important information inside this TLD that is required as we develop the other elements of the component:

- <shortname>** This is used as the namespace for the component.
- <uri>** When the TLD is packaged in the META-INF of the JAR file this URI can be used to reference the package rather than a hard-coded file reference to the TLD. This is good when want to develop custom components that have to be distributed.
- <name>** This is the name of the custom component as it will appear in the JSP page.
- <tagclass>** The class file that is used to manage the interface to the tag when it is placed on the page.

Next we create the Java classes that implement the JSF component.

UI component model

JSF contains a predefined set of UI component models. These can be used or extended. For this example we have designed a new component model for the Traffic Light Grid component that holds the underlying data to render the component. This model is used to hold the native types of data and maps directly the TLD attributes that have been defined.

- ▶ Under the Java Resources folder create a Java package and name it **itso.jsf.custom**.
- ▶ Create a class called **UITrafficLightGrid.java** and include the code from Example 14-4. To save time you can import this Java class from `\labscode\dev-ext\custom\UITrafficLightGrid.java`.

Example 14-4 UIComponent implementation class

```
package itso.jsf.custom;

import javax.faces.component.UIComponentBase;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;

public class UITrafficLightGrid extends UIComponentBase {
    public static final String RENDER_TYPE = "itso.renderer.TrafficLightGrid";
    public static final String FAMILY = "itso.custom.Data";

    public static final String TITLE_VB = "title";
    public static final String DATA_VB = "data";
    public static final String ROWLABELS_VB = "rowlabels";
    public static final String COLLABELS_VB = "collabels";
    public static final String MIN_VB = "min";
    public static final String MAX_VB = "max";

    private Object _data = null;
    private String _title = null;
    private Object _rowLabels = null;
    private Object _collabels = null;
    private Integer _min = null;
    private Integer _max = null;

    public UITrafficLightGrid() {
        setRendererType(RENDER_TYPE);
    }

    public void setMin(Integer min) {
        _min = min;
    }
}
```

```

public void setMax(Integer max) {
    _max = max;
}

public Integer getMin() {
    if (_min != null)
        return _min;
    ValueBinding vb = getValueBinding(UITrafficLightGrid.MIN_VB);
    return vb != null ? (Integer) vb.getValue(getFacesContext()) : null;
}

public Integer getMax() {
    if (_max != null)
        return _max;
    ValueBinding vb = getValueBinding(UITrafficLightGrid.MAX_VB);
    return vb != null ? (Integer) vb.getValue(getFacesContext()) : null;
}

public String getTitle() {
    if (_title != null)
        return _title;
    ValueBinding vb = getValueBinding(UITrafficLightGrid.TITLE_VB);
    return vb != null ? (String) vb.getValue(getFacesContext()) : null;
}

public void setTitle(String title) {
    _title = title;
}

public void setRowLabels(Object value) {
    _rowLabels = value;
}

public Object getRowLabels() {
    if (_rowLabels != null)
        return _rowLabels;
    ValueBinding vb = getValueBinding(UITrafficLightGrid.ROWLABELS_VB);
    return vb != null ? (Object) vb.getValue(getFacesContext()) : null;
}

public void setColLabels(Object value) {
    _colLabels = value;
}

public Object getColLabels() {
    if (_colLabels != null)
        return _colLabels;
    ValueBinding vb = getValueBinding(UITrafficLightGrid.COLLABELS_VB);
    return vb != null ? (Object) vb.getValue(getFacesContext()) : null;
}

```

```

    public void setData(Object value) {
        _data = value;
    }

    public Object getData() {
        if (_data != null)
            return _data;
        ValueBinding vb = getValueBinding(UITrafficLightGrid.DATA_VB);
        return vb != null ? (Object) vb.getValue(getFacesContext()) : null;
    }

    public String getFamily() {

        return FAMILY;
    }
}

```

The class must extend the `UIComponentBase`. This allows it to be a data model that can be used by custom renderers. The class returns its default render type and family, which are used when mapping this UI component to the custom renderers through the `faces-config.xml` file.

The main body of the module is used to manage the attributes for the custom JSF component. The `UIComponent` maintains the values in the native type and returns them to a user of this component. For example the `collabels` is held as an `Object` that allows either a `java.lang.String` or `java.util.ArrayList`. The `set` method for a given property stores the value in a local variable for this component, and the `get` method evaluates the value as a value binding.

Tag library component

The tag library definition class maps directly to the tag attributes that have been developed in the TLD created earlier:

- Import the **TrafficLightGridTag.java** into the `itso.jsf.custom` package from `\labscode\dev-ext\custom\TrafficLightGridTag.java` (Example 14-5).

Example 14-5 TLD implementation Java class

```

package itso.jsf.custom;
import java.util.ArrayList;
import java.util.StringTokenizer;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.faces.webapp.UIComponentTag;

```

```

/**
 * TLD Implmentation class
 */
public class TrafficLightGridTag extends UIComponentTag {

    public static final String RENDER_TYPE = "itso.renderer.TrafficLightGrid";
    public static final String COMPONENT_TYPE = "TrafficLightGrid";
    private String id;
    // hold the data values
    private String data;
    private String min;
    private String max;
    // Hold the Title
    private String title;
    private String rowLabels;
    private String collabels;

    public TrafficLightGridTag() {
    }

    public String getComponentType() {
        return TrafficLightGridTag.COMPONENT_TYPE;
    }

    public String getRendererType() {
        return RENDER_TYPE;
    }

    public String getMax() {
        return max;
    }

    public String getMin() {
        return min;
    }

    public void setMax(String i) {
        max = i;
    }

    public void setMin(String i) {
        min = i;
    }

    public void setData(String string) {
        data = string;
    }

    public String getId() {
        return id;
    }

    public void setId(String string) {

```

```

        id = string;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String string) {
        title = string;
    }

    public String getColLabels() {
        return collabels;
    }

    public String getData() {
        return data;
    }

    public String getRowLabels() {
        return rowLabels;
    }

    public void setColLabels(String string) {
        collabels = string;
    }

    public void setRowLabels(String string) {
        rowLabels = string;
    }

    protected void setProperties(UIComponent component) {
        super.setProperties(component);
        UITrafficLightGrid grid = (UITrafficLightGrid) component;
        // Title Property
        if (title != null) {
            if (isValueReference(title)) {
                ValueBinding vb =
                    FacesContext
                        .getCurrentInstance()
                        .getApplication()
                        .createValueBinding(
                            title);
                grid.setValueBinding(UITrafficLightGrid.TITLE_VB, vb);
            } else {
                grid.setTitle(title);
            }
        }
        // Min
        if (min != null) {
            if (isValueReference(min)) {
                ValueBinding vb =
                    FacesContext
                        .getCurrentInstance()

```

```

        .getApplication()
        .createValueBinding(
            min);
        grid.setValueBinding(UITrafficLightGrid.MAX_VB, vb);
    } else {
        Integer lMin = new Integer(min);
        grid.setMin(lMin);
    }
}
// Max
if (max != null) {
    if (isValueReference(max)) {
        ValueBinding vb =
            FacesContext
                .getCurrentInstance()
                .getApplication()
                .createValueBinding(
                    max);
        grid.setValueBinding(UITrafficLightGrid.MAX_VB, vb);
    } else {
        Integer lMax = new Integer(max);
        grid.setMax(lMax);
    }
}
// rowLabels
if (rowLabels != null) {
    if (isValueReference(rowLabels)) {
        ValueBinding vb =
            FacesContext
                .getCurrentInstance()
                .getApplication()
                .createValueBinding(
                    rowLabels);
        grid.setValueBinding(UITrafficLightGrid.ROWLABELS_VB, vb);
    } else {
        // String tokenize to create array list
        ArrayList newRowLabels = new ArrayList();
        grid.setRowLabels((Object) newRowLabels);
    }
}
// colLabels
if (colLabels != null) {
    if (isValueReference(colLabels)) {
        ValueBinding vb =
            FacesContext
                .getCurrentInstance()
                .getApplication()
                .createValueBinding(
                    colLabels);
        grid.setValueBinding(UITrafficLightGrid.COLLABELS_VB, vb);
    } else {
        // String tokenizer to create array list
        ArrayList newColLabels = new ArrayList();
        grid.setColLabels((Object) newColLabels);
    }
}

```

```

    }
}
// Value
if (data != null) {
    if (isValueReference(data)) {
        ValueBinding vb =
            FacesContext
                .getCurrentInstance()
                .getApplication()
                .createValueBinding(
                    data);
        grid.setValueBinding(UITrafficLightGrid.DATA_VB, vb);
    } else {
        // String tokenizer to create array list
        ArrayList values = new ArrayList();
        grid.setData((Object) data);
    }
}
}
}
}

```

- ▶ You can see that this class implements the getter and setter methods for each property that is defined in the TLD. You can also see that the class subclasses the faces class of `UIComponentTag`. This class helps the Tag live within the Faces life cycle and is called at the key points when the tag has to receive updates from the JSF tags definitions.
- ▶ The other method that is useful is `getRenderType`. This method returns the render type that renders the user interface. This is important as the value matches the renderer name within the `faces-config.xml`. We will explain this later in “Faces configuration file” on page 498.
- ▶ At the bottom of the class is the `setProperties` method that processes the properties and adds them into value binding. Let us explain one example, because this code allows the JSF component attribute to support value binding:
 - The `setProperties` method is called passing the underlying `UIComponent` that maps to the UI object as parameter. Therefore we cast the parameter to `UITrafficLightGrid`.
 - We check if the value passed in is a value reference in the notation of a `{object.attribute}` reference. If true, we create a new value binding object and pass it to the `UIComponent`. Otherwise we can assume that it is static and can directly update the underlying data model component.

Tip: To verify that the value reference binding information is consistent between the `UIComponent` and the `UIComponentTag`, you can use static variables of `UIComponent`, for example `UITrafficLightGrid.TITLE_VB`.

User interface renderer

Renderers are used to render the user interface components based on a given `UIComponent`. So far you have seen that we have created a JSF tag and a `UIComponent`. All that is required to complete the Java coding is the renderer that sends HTML tags to the application client.

For this component we generate an HTML 4.01 `<table>`. The individual cells and header areas have associated style sheet classes (Example 14-2 on page 484) to allow the visual part of the component be separate from the rendered part.

To develop a custom renderer you have to extend the `javax.faces.render.Renderer` class and overwrite a set of methods:

decode	This method is used to decode a response from the J2EE client. The <code>TrafficLightGrid</code> component does not have any update capability so we do not have to add any code to this method.
encodeBegin	This method is called when the component is asked to render its contents. The underlying <code>UIComponent</code> class that is mapped to this renderer is passed to this method, so the renderer has full access to the component's copy of the data model.
encodeEnd	This method can be used to close off any rendering tasks. For example, if the custom component supported sub components, these other components could be rendered and this method will be called once the subcomponents are completed.

Import the **`TrafficLightGridRenderer.java`** into the `itso.jsf.custom` package (`\labscode\dev-ext\custom\TrafficLightGridRenderer.java`):

- In the `encodeBegin` method of the class we put the code that is used to extract the underlying data and to generate the `TrafficLightGrid` and apply the business logic to the threshold limits (Example 14-6).

Example 14-6 encodeBegin method of the TrafficLightGridRenderer

```
public void encodeBegin(FacesContext context, UIComponent component)
    throws IOException {

    if (context == null) {
        throw new NullPointerException("Faces Context is Null");
    }

    if (!component.isRendered())
        return;
```

```

// Get Context so we can starting writing output
ResponseWriter writer = context.getResponseWriter();
String id = component.getClientId(context);

// Get Handle to underlying component of renderer
UITrafficLightGrid grid = (UITrafficLightGrid) component;

// Get data from component
String title = (String) grid.getTitle();
ArrayList data = (ArrayList) grid.getData();
ArrayList cols = (ArrayList) grid.getColLabels();
ArrayList rows = (ArrayList) grid.getRowLabels();

// Get threshold limits
int min = Integer.parseInt(grid.getMin().toString());
int max = Integer.parseInt(grid.getMax().toString());

// Get iterators
if (cols.size() < 1 || rows.size() < 1 || data.size() < 1)
    throw new FacesException();

Iterator iCols = cols.iterator();
Iterator iRows = rows.iterator();
Iterator iData = data.iterator();

// Start Writing out Component
writer.write(
    "<LINK rel=\"stylesheet\" type=\"text/css\" "
    href=\"theme/trafficLight.css\">");

writer.write(
    "<TABLE id=\""
    + id
    + "\" width=\"80%\" border=\"0\" cellpadding=\"2\" "
    cellspacing=\"2\" bgcolor=\"#c0c0c0\">");
writer.write("\t<TBODY>");
writer.write("\t\t<TR>");
writer.write(
    "\t\t\t<TD width=\"150\" nowrap align=\"center\"><H4><SPAN "
    class=\"title\"><A HREF=\"\">
    + title
    + "</A></SPAN></H4>");

writer.write("\t\t\t</TD>");

String colLabel, rowLabel, cellValue = "";
String style = "minClass";

while (iCols.hasNext()) {

```



```
// Write end of Table
writer.write("\t</TBODY>");
writer.write("</TABLE>");
}
```

- ▶ When you read through this method you can see that the `UIComponent` is passed as a parameter to the component. We open a `ResponseWriter` and use this object to send the user interface HTML tags to the client Web browser.
- ▶ Notice the logic where we compare the values against the threshold limits and then make the decision which style to apply to the cell.

Now that all the code is written, we define the component in the JSF configuration so that we can test the component.

Faces configuration file

The `faces-config.xml` is the file that controls the way that the faces components behave within the Web project application (see “Faces configuration file” on page 13).

When developing custom components you have to add the configuration information for the custom component to the `faces-config.xml` file. You also have to maintain a separate `faces-config-comp.xml` file that is used in the packaging phase. This first step allows the component to be tested within the Web project. The second step prepares the packaged JAR file so that the component can be added to another Web project that is running JSF pages and does not require the component definition to be added to the `faces-config.xml` file before it is used.

We now add the two parts, component and renderer, to the `faces-config.xml` file located in the `WebContent/WEB-INF` folder of the Web project (Example 14-7).

Example 14-7 Add component and renderer definition to the `faces-config.xml` file

```
<component>
  <component-type>TrafficLightGrid</component-type>
  <component-class>itso.jsf.custom.UITrafficLightGrid</component-class>
</component>
<render-kit>
  <renderer>
    <component-family>itso.custom.Data</component-family>
    <renderer-type>itso.renderer.TrafficLightGrid</renderer-type>
    <renderer-class>
      itso.jsf.custom.TrafficLightGridRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

```
</render-kit>
</faces-config>
```

The highlighted tags are the component type and the renderer type. Notice that the component type matches the value returned in the `getComponentType` method of the `UIComponentTag`, and the renderer type matches the value returned from the `getRendererType` of the `UIComponent`.

Save this file and create a duplicate file that we use for packaging:

- ▶ Import the **faces-config-comp.xml** file into `WebContent\WEB-INF` (`\labscode\dev-ext\custom\faces-config-comp.xml`). This file has the same content as the `faces-config.xml` file, without the managed bean from our sample JSP.
- ▶ We will use this file when we package the component for deployment (see “How to package JSF components” on page 503).

Do not confuse this file with the real `faces-config.xml` file that is included in this project. Any changes to the master configuration file must be duplicated into the packaging file.

Now that we have completed all the tasks of developing a JSF custom component, we can now look at testing the component in the Web project.

Initial testing

Testing is an important part of any application. Even more so for a component that could be used in a wide variety of situations and applications. We mentioned earlier that using a Web project for the development of the custom component allows not only for the Java classes to be developed, but it is a perfect place to test the component.

When we first created the Web project we created a page called `customComponent.jsp`. We use this JSP for testing of the custom component:

- ▶ First we define the `JavaBean` that holds the data model. Create a package called **itso.jsf.custom.test** and import Java class **GridData.java** from `\labscode\jsf-custom\GridData.java` (Example 14-8).

Example 14-8 GridData.java test code

```
package itso.jsf.custom.test;

import java.util.ArrayList;
/**
 * @author Matthew Perrins
 */
public class GridData {
```

```

ArrayList collLabels = new ArrayList();
ArrayList rowLabels = new ArrayList();
ArrayList data      = new ArrayList();
String title = "Sales by Region";

public GridData()
{
    collLabels.add("January");
    collLabels.add("February");
    collLabels.add("March");
    collLabels.add("April");

    rowLabels.add("North");
    rowLabels.add("East");
    rowLabels.add("South");
    rowLabels.add("West");

    ArrayList d = new ArrayList();
    d.add("100");
    d.add("120");
    d.add("130");
    d.add("140");

    data.add(d);

    d = new ArrayList();
    d.add("200");
    d.add("220");
    d.add("230");
    d.add("240");

    data.add(d);

    d = new ArrayList();
    d.add("300");
    d.add("320");
    d.add("330");
    d.add("340");

    data.add(d);

    d = new ArrayList();
    d.add("400");
    d.add("420");
    d.add("430");
    d.add("440");

    data.add(d);
}

public ArrayList getCollLabels() {
    return collLabels;
}

```

```

public ArrayList getRowLabels() {
    return rowLabels;
}

public String getTitle() {
    return title;
}

public ArrayList getData() {
    return data;
}

public void setColLabels(ArrayList list) {
    colLabels = list;
}

public void setRowLabels(ArrayList list) {
    rowLabels = list;
}

public void setTitle(String string) {
    title = string;
}

public void setData(ArrayList list) {
    data = list;
}
}

```

- ▶ Open the `customComponent.jsp` page and add the `GridData` class as a session scope managed bean named `griddata` to the Page Data view.
`GridData` is the JavaBean that we use to supply the test data to the JSF `TrafficLightGrid` component. It would be nice if we could drop this component from the Palette view, but the custom component has not been integrated into the palette. We add the component to the palette later in “Extending the Studio palette” on page 512.
- ▶ We have to add the component to the JSF manually. The first step is to register the taglib within the page. Go to the Source page and add the following line to the top of the JSF page below the definition for the core library.

```
<%@ taglib uri="/WEB-INF/tld/trafficLightGrid.tld" prefix="itso" %>
```

We are directly referencing the TLD file that we created.

Note: When you receive this component as a distributed component you reference the URI using `http://www.ibm.com/jsf/itso_extended`. This is explained in “Testing custom components” on page 519.

- Change the default text to **<h3>Test Traffic Light Grid</h3>**.
- Under the heading add the tag for the custom component:
 - Select *JSP -> Insert Custom* and in the dialog select the `itso` prefix and the `trafficLightGrid` tag. Click *Insert* and close the dialog. This action inserts the line:


```
<itso:trafficLightGrid></itso:trafficLightGrid>
```
 - Open the Properties view (*Window -> Show View -> Properties*). The properties view displays the attributes of the `trafficLightGrid` tag when the cursor is placed on the tag in the source. Enter the attribute values and verify that they are inserted into the source (Example 14-9).

Example 14-9 Adding trafficLightGrid tags to customComponent.jsp

```
<itso:trafficLightGrid
  id="tlight1"
  title="#{griddata.title}"
  data="#{griddata.data}"
  rowLabels="#{griddata.rowLabels}"
  colLabels="#{griddata.colLabels}"
  min="220"
  max="300">
</itso:trafficLightGrid>
```

- You can see the references to the `griddata` managed bean that we created.
- Save the JSF page and test it within a WebSphere 5.1 Test Environment.

When you run the page you see the grid drawn below the text that was entered. The `trafficLightGrid` expands itself up to the size of its parent area (Example 14-2).

Test Traffic Light Grid				
<u>Sales By Region</u>	January	February	March	April
North	100	120	130	140
East	200	220	230	240
South	300	320	330	340
West	400	420	430	440

Figure 14-2 JSP execution with custom component

After having successfully tested the custom component, we explain what is required to package the component for distribution to other people. Afterwards we create a WebSphere Studio plug-in that allow this component to be integrated into the Web perspective palette. There is a final section explaining how to use the custom component package and the plug-in within a Web project.

How to package JSF components

We have been using the dynamic J2EE Web project to host the development of the JSF component. The issue is that this project format is not compatible with distributing custom JSF components. What we require is a JAR file that can be distributed. The easiest way to do this with WebSphere Studio is to use the built-in Ant support.

There are a couple of key things to remember when packaging the JAR file. To make the tag library accessible by other development tools we have to package the TLD file into the META-INF folder of the JAR file. The second key packaging element is that we also have to include a `faces-config.xml` file into the META-INF folder. This automatically registers the component with a JSF project which includes this JAR in its class path. The final element is to package the compiled component code.

Ant build script

To create a packaging script we create a `build.xml` file that can be run by the Ant tool to create the packaged JSF component:

- ▶ Import the **build.xml** file from `\labscode\dev-ext\custom\build.xml` into the root of the project (Example 14-10).

Example 14-10 Ant script for packaging the custom component

```
<?xml version="1.0"?>
<!--
  Company : IBMM - ITSO
  Name    : BUILD.XML
  Desc    : Build file for JSF Custom Components
-->

<project name="build" default="start" basedir=".">
  <property name="name" value="itso-custom-jsf-components"/>
  <property name="version" value="1_0_0"/>
  <property name="src" value="WebContent/WEB-INF/classes"/>
  <property name="build" value="build"/>
  <property name="package" value="package"/>

  <target name="init">
```

```

        <tstamp/>
        <delete dir="${build}"/>
        <mkdir dir="${build}"/>
        <mkdir dir="${build}/${package}"/>
        <mkdir dir="${build}/${package}/META-INF"/>
    </target>

    <target name="package">
        <copy file="WebContent/WEB-INF/TLD/trafficLightGrid.tld"
            todir="${build}/${package}/META-INF"/>
        <copy file="WebContent/theme/trafficLight.css"
            todir="${build}/${package}/META-INF"/>
        <copy file="WebContent/META-INF/MANIFEST.MF"
            todir="${build}/${package}/META-INF"/>
        <copy file="WebContent/faces-config-comp.xml"
            tofile="${build}/${package}/META-INF/faces-config.xml"/>
        <copy todir="${build}/${package}">
            <fileset dir="${src}">
                <include name="itso/**/*.class"/>
            </fileset>
        </copy>
    </target>

    <target name="jar">
        <jar jarfile="./${name}-${version}.jar" basedir="${build}/${package}" />
        <delete dir="${build}"/>
    </target>

    <target name="start" depends="init,package,jar"/>
</project>

```

- ▶ This build script takes all the artifacts from the Web project and creates the JAR file called `itso-custom-jsf-components-1_0_0.jar`. Notice that the build script moves the `faces-config-comp.xml` to the `META-INF/faces-config.xml` and it moves the TLD into the `META-INF` folder.
- ▶ Select the `build.xml` file and *Run Ant* (context). In the Ant run dialog make sure the `start` item is selected and then click *Run* (Figure 14-3).

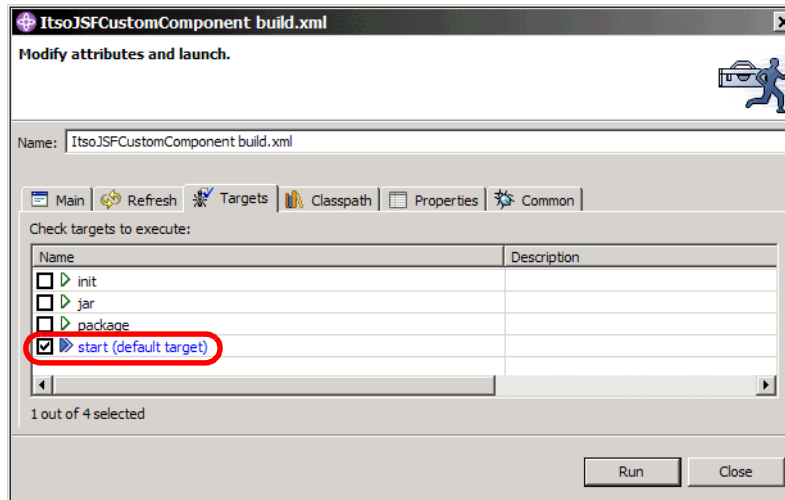


Figure 14-3 Ant build dialog

- ▶ Once the run has completed, you can see a BUILD SUCCESSFUL message in the Console view. Then select the Web project and *Refresh*. This action displays the package JAR file within the project.
- ▶ You can export this JAR or just drag it onto the file system ready for distribution.

To test the distribution package, refer to “Testing custom components” on page 519.

Summary

In this section we discussed how to develop a custom JSF component and how this component can be tested and packaged ready for distribution. Part of the success of JSF is its ability for custom components to be developed and distributed in development tools, packaged solutions, and as software packages for sale. Each of these component packages allows the JSF specification to gain momentum in the industry and allow customers more choice on which components they want to use within their J2EE applications.

You can see that the steps for creating custom components within WebSphere studio are relatively simple and we would recommend that you read one the many books on the market that discussion JSF component development in detail (see “Related publications” on page 551).

Custom converters

Custom converters are used when you need special data conversion between the model and the presentation representation of the data, such as numbers, dates, or currencies.

The process involves writing a class that implements the `javax.faces.convert.Converter` interface, and then registering the converter in the application configuration file. Then you can use the converter in your JSF pages.

Converter interface

The `javax.faces.convert.Converter` interface has two methods:

- ▶ The **`getAsObject`** method converts data from the presentation representation to the model representation:

```
public Object getAsObject (FacesContext context, UIComponent component,
                          String newValue) throws ConverterException;
```

- ▶ The **`getAsString`** method converts data from the model representation to the presentation representation:

```
public String getAsString (FacesContext context, UIComponent component,
                          Object value) throws ConverterException;
```

Registering the converter

You must register the converter before you can use it in an application. The converter class is registered in the `faces-config.xml` configuration file using a unique identifier.

You use the converter element, which must have two sub-elements: `converter-id` and `converter-class`. The `converter-id` element specifies the unique identifier for the converter in the JSF Web application, and the `converter-class` defines the fully qualified Java class implementing the `Converter` interface. The `faces-config.xml` file contains a definition like this:

```
<converter>
  <converter-id>temperatureConverter</converter-id>
  <converter-class>itso.jsf.TemperatureConverter</converter-class>
</converter>
```

Note: Once you have completed a custom converter, you can include it in the custom component package by adding these tag definitions to the `faces-config-comp.xml` (see “Faces configuration file” on page 498). The Ant build file automatically includes the class files into the package. You can then use this custom converter within any JSF application.

Converter example

In this section we develop a custom converter and use it in a simple JSF page. We want to write a simple temperature converter that allows us to convert between celsius degrees and fahrenheit degrees. Follow these steps:

- ▶ Create a new JSF page and name it `temperatureConverter.jsp`. Replace the default text by typing **Converter from Celsius Degrees to Fahrenheit Degrees**.
- ▶ Add an Output component and enter **Enter celsius value:** for its value property.
- ▶ Add an Input component and enter `celsius` as id. Add a new line.
- ▶ Add an Output component and enter **The fahrenheit value is:** for its value property.
- ▶ Add an Output component and enter `fahrenheit` as id. Add a new line.
- ▶ Add a Command - Button component and change the label to **Convert**.
- ▶ Using the Page Data view, create a new variable in the request scope. Set its id to **degrees** and its type to `java.lang.Float`. We are using float values as the values of the temperatures.
- ▶ Bind the request scope variable `degrees` to the `celsius` input field and to the `fahrenheit` output text.
- ▶ The design is complete (Figure 14-4).

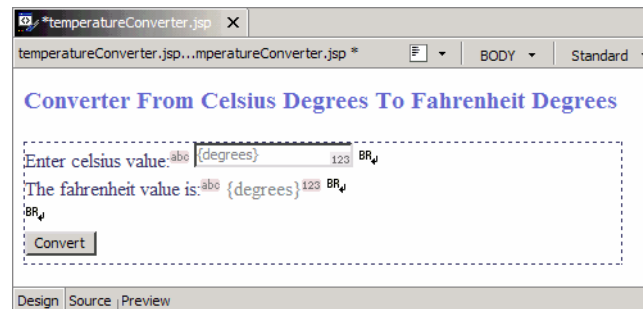


Figure 14-4 Page design for the temperature converter

Now we must write, register, and use the custom converter in the page:

- ▶ In the Web application, select the Java Resources folder and create a **itso.jsf.converter** package. Import the **TemperatureConverter** class from `\labscode\dev-ext\converter\TemperatureConverter.java`.

- The TemperatureConverter implements two methods:

- The getAsObject method converts values from the presentation view to the model view. The presentation view represents the value in celsius degrees, so we must implement a fahrenheit conversion algorithm:

```
public Object getAsObject(FacesContext context,UIComponent component,
                        String value) {
    float celsius = 0;
    Object result = null;
    try {
        celsius = Float.parseFloat(value);
        //convert to celsius
        float fahrenheit = ((celsius*9)/5) + 32;
        //compose result
        result = new Float(fahrenheit);
    } catch (NumberFormatException e) {
        throw new ConverterException
            ("The input value from the view is not a float value.");
    }
    return result;
}
```

- The getAsString method converts values from the model view to the presentation view. The model view represents the value in fahrenheit degrees, so we must implement a celsius conversion algorithm:

```
public String getAsString(FacesContext context,UIComponent component,
                        Object value) {
    float fahrenheit = 0;
    String result = null;
    try {
        fahrenheit = Float.parseFloat(value.toString());
        //convert to celsius
        float celsius = (fahrenheit - 32)*5/9;
        //compose result
        result = (new Float(celsius)).toString();
    } catch (NumberFormatException e) {
        throw new ConverterException
            ("The input value from the model is not a float value.");
    }
    return result;
}
```

- We register the converter at the end of the faces-config.xml file:

```
<converter>
  <converter-id>temperatureConverter</converter-id>
  <converter-class>itso.jsf.converter.TemperatureConverter
  </converter-class>
</converter>
```

- Save the configuration file.
- Now we use the converter in our page. Open the `temperatureConverter.jsp` page. Using the Source page of the Page Designer view, find the `celsius` input component and add the converter property to its tag:

```
<h:inputText styleClass="inputText" id="celsius"
    value="#{requestScope.degrees}" converter="temperatureConverter">
  <del>f:convertNumber</del>
</h:inputText>
```

- You can also add the converter by using the separate `<f:converter>` tag:

```
<h:inputText styleClass="inputText" id="celsius"
    value="#{requestScope.degrees}">
  <del>f:convertNumber</del>
  <f:converter converterId="temperatureConverter"/>
</h:inputText>
```

- Save and test the page (Figure 14-5).

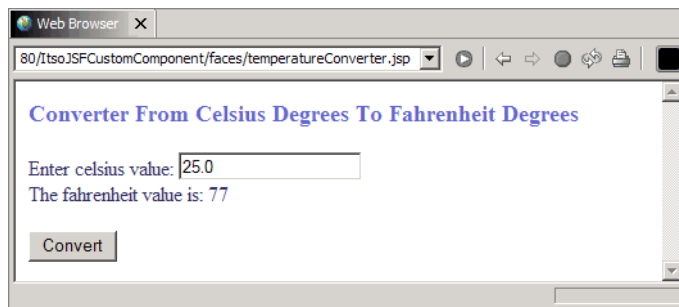


Figure 14-5 Temperature converter in action

- If you want to package the converter with the custom component, add the `<converter>` tags to the `faces-config-comp.xml` file and rerun the Ant build.

Studio support for custom components

In this section we discuss how custom component can be included in the WebSphere Studio development environment.

When a component developer has completed the task of developing JSF components, these components can be delivered to application developers for use within the Faces Web projects. The only restriction in doing this is that generally the tools that developers are using are not aware of the components and have no special support for drag and drop type operations.

Therefore, the developers has to manually add the JSF components tags to the JSF pages. This requires that the components have strong documentation to explain each tag.

What can be done to help the component developer extend WebSphere Studio? Because WebSphere Studio is built on the Eclipse framework it is possible to extend the development environment programmatically through plug-ins. This section explains how a component developer can add components to the Web perspective palette. This makes developers using these custom components more productive.

Adding a custom component to the palette

Note: Its is very important for component developers to be able to extend the Studio palette, but these steps listed below are not officially supported by IBM product support.

You can add the custom components to the Web perspective Palette by using the plug-in development environment within WebSphere Studio. We develop a plug-in with these features:

- ▶ Create a custom palette category.
- ▶ Create a custom JSF component within that palette category.
- ▶ Assign a custom icon for that JSF component.
- ▶ Allow drag and drop into the Page Designer.
- ▶ Generate syntax compliant tags in the JSF page.

The steps for creating a custom plug-in are very straight forward. We assume that the custom component has been developed. The key point is to match the TLD defined in the custom component to the definition in the plug-in. When dragging the palette object into a JSP the tags of Example 14-9 on page 502 must be generated.

Create a plug-in project

To start we create a plug-in project:

- ▶ Select *File -> New -> Project -> Plug-in Development -> Plug-in Project* and then click *Next*.
- ▶ Enter **ItsoJSFPlugin** for the project name and click *Next*.
- ▶ For the Plug-in id field enter **ItsoJSFPlugin**. Make sure the *Create a Java project* is selected and let the rest of values be set to the defaults (Figure 14-6). Click *Next*.

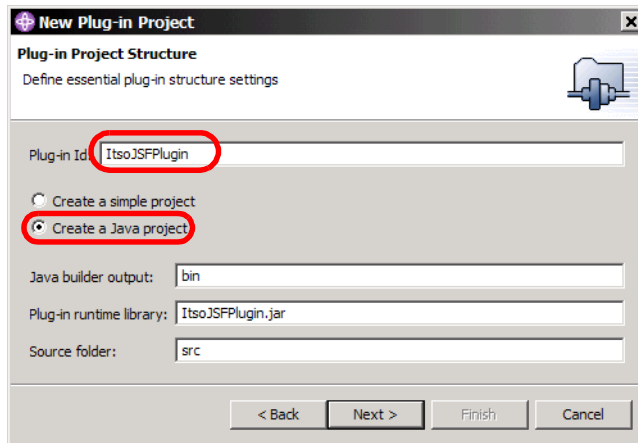


Figure 14-6 Plug-in structure settings

- ▶ In the next page of the wizard, select the *Create a Plug-in project using a code generation wizard* and select *Default Plug-in Structure*. Click *Next*.
- ▶ The next page of the wizard is important for defining the core information about the plug-in. Give the plug-in the name of **ITSO JSF Custom Component** and keep the version number as 1.0.0. Change the provider name to **IBM ITS0**. Then change the class name to **itso.jsf.CustomPlugin** and leave the rest of the settings as default. Click *Finish* (Figure 14-7).

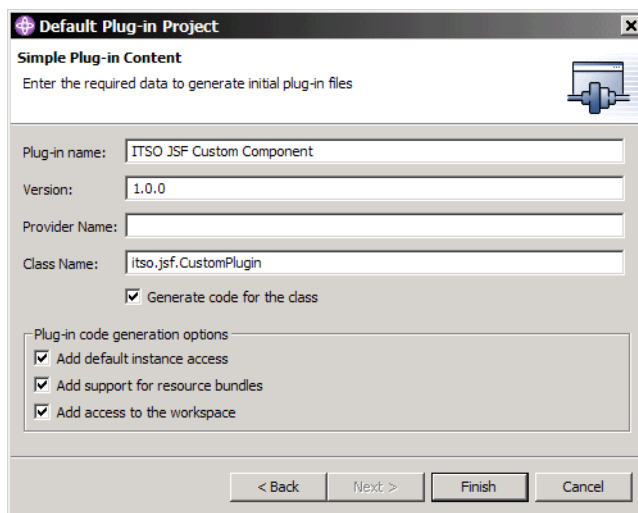


Figure 14-7 Configure the plug-in name details

- ▶ You may be warned that the plug-in required to compile Java classes to avoid compilation problems. This option is enabled for you in the project. Click *OK*.
- ▶ You may be asked if you want to change the perspective to the Plug-in Development perspective. Click *Yes*.

Extending the Studio palette

The palette adds a great deal of productivity to the application developer. Having a vendor or company custom components added to this palette allows developers to use these JSF components just by dragging them into the Page Designer.

The palette uses two icons (a small icon and a large icon) to represent the components. We have to define the icons that are used by the custom component in the Palette view. The component that we defined in the first section of this chapter is the Traffic Light Grid. This component renders to a distinctive view, so we can try to give the icons the same look and feel.

The two icons must be included in the project. To complete this task, create two folders called **iconsSmall** and **iconsLarge** in the project, then import the `\labscode\dev-ext\plugin\images\trafficLightSmall.gif` and `trafficLightLarge.gif` into the respective folder.

Tip: You can drag these files from the Windows explorer directly into the folders in the Studio workbench.

The small icon is a 16x16 pixel GIF image and the large icon is a 24x24 pixel GIF image (Figure 14-8).

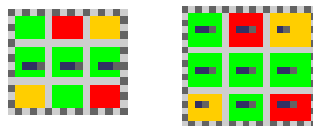


Figure 14-8 Design of the small and large icons for the palette extension

Next we should define the extension tags in the `plugin.xml` file. Before we do this we have to complete the step of creating the Ant build file before we start adding extensions and features to the `plugin.xml` file.

Important: There is a bug with Eclipse that causes the Export of plug-ins to be broken. The process hangs. The work-around is to create an Ant build script straight after the project has been created and before you modify the `plugin.xml` file.

Prepare early for packaged build

To allow this plug-in to be used by other developers or customers, we have to be able to export the plug-in as a ZIP file that can be easily installed within WebSphere Studio environment. Make sure you are in the Plug-in Development perspective.

Select the `plugin.xml` file and *Create Ant Build File* from the context menu. Wait until the progress bar has finished. The `ant build.xml` file is created for you. You can modify this file later to include all the project resources required.

Adding extensions

In the Plug-in Development perspective open the `plugin.xml` file:

- ▶ You see the Welcome screen for the plug-in (Figure 14-9).

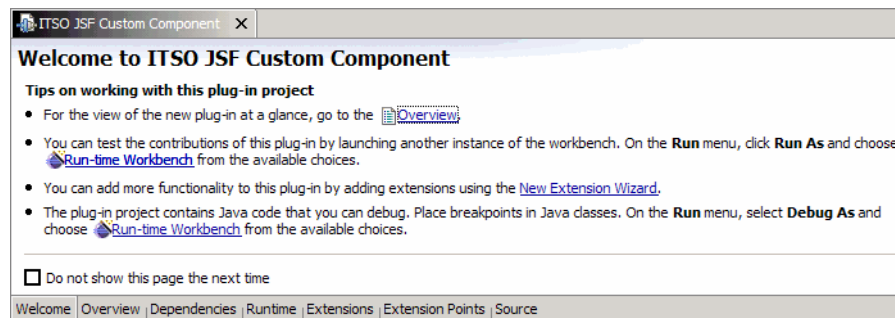


Figure 14-9 `Plugin.xml` file welcome screen

- ▶ Jump to the Source page. If you are an experienced plug-in developer you could use the other pages.
- ▶ The first step is adding other plug-ins that are required by our plug-in. This is done using the `<requires>` tag (Example 14-11).

Example 14-11 Required plug-in definition for `plugin.xml`

```
<requires>
  <import plugin="com.ibm.etools.jsf" version="5.1.2" match="equivalent"/>
  <import plugin="com.ibm.etools.webedit.palette" version="5.1.2"
    match="equivalent"/>
</requires>
```

- ▶ To extend the WebSphere Studio palette you have to use the `com.ibm.etools.webedit.palette.PageDesignerPaletteContributions` extension point. Add the tags in Example 14-12.

Example 14-12 Extending the palette from the palette extension point

```
<!--Adding components to the palette for Drag and Drop-->
<extension id="JSF Palette Contribution"
  name="ITS0 Palette Contributions"
  point="com.ibm.etools.webedit.palette.PageDesignerPaletteContributions">
</extension>
```

- We create a new category in the palette to keep a list of the components and give them a unique area on the palette. The following tags are inserted inside the <extension> tag (Example 14-13).

Example 14-13 Creating a palette category

```
<category
  label= "ITS0 Custom JSF Controls"
  description="ITS0 Custom Components"
  initiallyopen="true"
  visible="true"
  preferredoffset="40"
  id="itso.jsf.custom.html">
  <context>
    <JSF initiallyopen="true" visible="true" />
  </context>
</category>
```

- The important parts to these tags are the unique ID and the <context> tag, which makes sure the category is open and visible. This makes the components easier to see when we open a JSF page in Page Designer.
- Next we add components to the extension (Example 14-14).

Example 14-14 Defining the new custom component in the palette

```
<item
  id="ITS0.Standard.TrafficGrid"
  label="Traffic Light Grid"
  description="Display data with traffic light thresholds"
  category="itso.jsf.custom.html"
  visible="true"
  iconLarge="iconsLarge/trafficlightLarge.gif"
  iconSmall="iconsSmall/trafficlightSmall.gif"
  taglibUri="http://www.ibm.com/jsf/itso_extended"
  preferredoffset="0">
</item>
```

- Notice that you specify the icon references assigned to this palette icon. We also added a reference to the new category. We can also change the label and description that is displayed in the palette.

- ▶ The most important part of this element is the link to the taglib URI. You notice in the first section of this chapter that we defined our taglib with this URI. This makes the definitions with the JSF pages clean and tidy and is good development practice for components.
- ▶ Next we define the attributes that are code generated when we drop this component onto a JSF page. Example 14-15 shows the code that we add inside the `<item>` tag.

Example 14-15 Attributes that map to the custom component attributes

```
<attribute
  actionclass="com.ibm.etools.jsf.palette.actions.JsfDropAction"
  tagname="trafficLightGrid">
  <property attrvalue="Sales by Region" attrname="title"></property>
  <property attrvalue="" attrname="rowLabels"></property>
  <property attrvalue="" attrname="colLabels"></property>
  <property attrvalue="25" attrname="min"></property>
  <property attrvalue="75" attrname="max"></property>
  <property attrvalue="" attrname="data"></property>
  <property attrvalue="itso" attrname="preferredprefix">
  </property>
</attribute>
```

- ▶ We defined the tag `trafficLightGrid`. This name must match the tag name of the custom component. You also notice that we have an action class that is used to handle the drop action into the page designer page. This extension takes the attributes that have been defined and generate the tags within the JSF page.
- ▶ Each attributes that is required by the component has an attribute name and default value. There is a `preferredprefix` property that is used to define the prefix and matches the name space that we defined within the TLD of the custom component.

Attention: These attributes must match the attributes that are defined within the custom component TLD. Otherwise you get errors when the tag is dropped onto page because the generated tags fail TLD validation.

- ▶ The last part we add to the `plugin.xml` file is designed to handle the tag generation when the component is added to the Page Designer Design or Source page.

This extension maps the taglib URI we defined for the custom component to both the `<item>` tags defined earlier and the custom component TLD. If all these values match then the component tags are auto-generated into the page (Example 14-16).

Example 14-16 Extension point for tag generation after drop operation.

```
<extension point="com.ibm.etools.jsf.jsfTaglib">
  <taglibInfo uri="http://www.ibm.com/jsf/itso_extended"></taglibInfo>
</extension>
```

The tags that are generated match the attribute values that are defined in the tag library definition.

You can now save the `plugin.xml` file and prepare for testing the plug-in in WebSphere Studio.

Note: to check that you have created the `plugin.xml` correctly you can compare the file with `\labscode\dev-ext\plugin\plugin.xml`

Testing the plug-in

To test the plug-in, go to the Plug-in Development perspective and select the `plugin.xml`. Select *Run -> Run as -> Run-time Workbench*. This launches another WebSphere Studio runtime workbench with the plug-in running in it. We use this new workbench to test the plug-in.

In this new WebSphere Studio workbench you have a new runtime workspace. The default path for this workspace is the parent directory of the current workspace `<current workspace parent>/runtime-workspace`. You can use Windows Explorer to clear it after testing.

Here are the steps to test the plug-in:

- ▶ Create a dynamic Web project named **ItsoJSFPluginTestWeb** in the **ItsoJSF14Plugin** enterprise application.
- ▶ Create a JSF file **pluginTest.jsp** in this project. Then open the JSF page for editing.
- ▶ If you have followed the steps correctly, you see the new component category on the palette and the custom component registered in it (Figure 14-10).



Figure 14-10 Custom component on the palette

- Drag the traffic light component into the test page; it appears as a question mark (Figure 14-11).



Figure 14-11 Adding the custom component to the page

- The tags are generated into the source code:

```
<itso:trafficLightGrid max="75" styleClass="trafficLightGrid" min="25"
    rowLabels="" colLabels="" data="" title="Sales by Region"
    id="id1">
</itso:trafficLightGrid>
```

- If you save the file you get errors because the custom component is not installed into the Web project.

We will do the real testing later in “Testing custom components” on page 519. For now seeing the component in the palette and dragging it into a JSP is enough.

We have completed the WebSphere Studio plug-in that adds the ITSO custom components to the palette. The next step is to export the plug-in ready for use by developers or customers. Exit the test workbench.

Creating a deployable plug-in

In this section we explain how to modify the Ant build script to generate a ZIP file containing the plug-in. Other developers can use this file to install the plug-in in their development environment.

There are two ways of managing the plug-in so that it can be delivered to developers or customers. You can either export the plug-in, or create an Ant build file that manages the building of the plug-in.

With both options you get a ZIP file that you can include in conjunction with the custom component to distribute the work product. The issue at this stage of writing is that the export option is not working.

To save time you can jump to “Testing custom components” on page 519 and use the `\labscod\dev-ext\custom\ItsoJSFPlugin_1.0.0.zip` file.

Ant build script

We have to make some modifications to the Ant build script to include the icon folders. Open the file `build.xml` and add a new target called `copyicons` above the `<target name="zip.plugin">` (Example 14-17).

Example 14-17 Add copy target to build.xml

```
<target name="copyicons" if="destination.folder">
  <copy todir="${temp.folder}/${full.name}/iconsLarge">
    <fileset dir="iconsLarge"/>
  </copy>
  <copy todir="${temp.folder}/${full.name}/iconsSmall">
    <fileset dir="iconsSmall"/>
  </copy>
</target>
```

- ▶ This Ant task copies the folders into the temporary directory used by the build script for packaging the plug-in.
- ▶ We have to use this new Ant task. Navigate to the `zip.plugin` target task and, above the tag `<antcall target="zip.folder"/>`, add the following lines:

```
<antcall target="copyicons">
  <param name="destination.folder" value="${temp.folder}/${full.name}"/>
</antcall>
```
- ▶ Finally we can run the Ant build script. Select the `build.xml` file and *Run Ant*. You are presented with the Ant dialog (Figure 14-12).

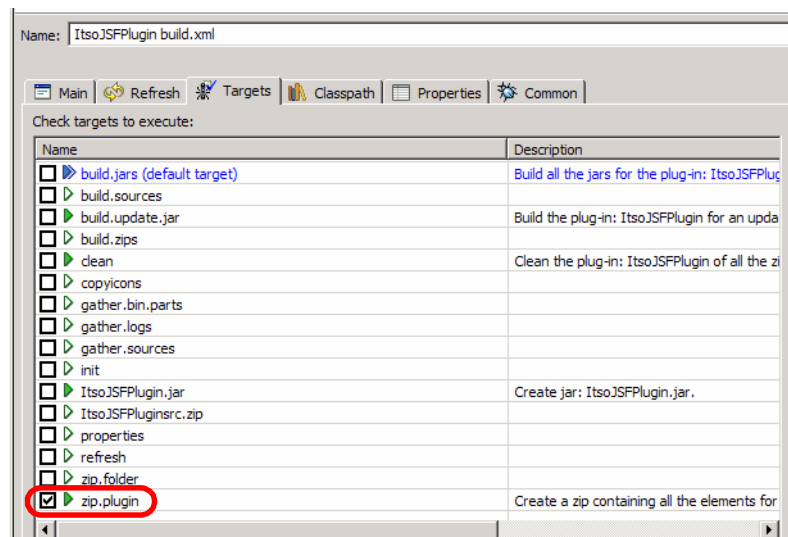


Figure 14-12 Ant build run dialog to generate the plug-in

- ▶ Deselect the default target and select `zip.plugin` target. Click *Run* and watch the Console.
- ▶ Refresh the project and you can see the ZIP file `ItsoJSFPlugin_1.0.0.zip`.
- ▶ You can distribute this ZIP file with the custom component that was developed in the first section:

```
ItsoJSFPlugin_1.0.0.zip  
itso-custom-jsf-component-1_0_0.jar
```

Refer to “Testing custom components” on page 519 for detailed instructions.

Component visualizer and attributes

You can extend the plug-in further by adding even more support for integration within Studio. One of the most common features requested for tag libraries is the ability to display a custom visualisation of the tag when it is displayed in Page Designer. Rather than taking you through a detailed example, we refer you to an article on developerWorks® that will guide you through the process. This article also explains how to develop the custom Attributes view for the component.

http://www-106.ibm.com/developerworks/websphere/library/techarticles/0304_hosokawa/hosokawa.html

Other extensions

You may want to exploit other Studio tools for JSF development, such as the Quick Edit view, Page Data drag and drop data binding, and page code convenience method generation. Unfortunately, these features are not easily available for custom component extensions and we do not cover these features in this book.

Testing custom components

In this section we explain how to install the custom component plug-in in Studio. This allows you to have the custom component in the palette and use it in your Web applications. We also explain how to include the custom tag library for the new component. This tag library is required for the component to run inside the Web application.

Installing custom components in Studio

To use the palette extension in the project, you must install the plug-in in WebSphere Studio Application Developer:

- ▶ Stop Application Developer. New plug-ins are only available after a restart.
- ▶ The developed plug-in is packaged as a ZIP file that contains all the necessary data for the plug-in. This file must be placed into the Studio installation directory.

To install the file, unzip the plug-in file into the Studio installation directory:

```
Source:  \labscode\dev-ext\plugin\ItsoJSFPlugin_1.0.0.zip
Target:  <WSAD_install>\wstools\eclipse\plugins
```

This creates the directory `ItsoJSFPlugin_1.0.0` with the plug-in JAR file and the icon subdirectories.

- ▶ Start Application Developer.
- ▶ Open the `customComponent.jsp` in the `ItsoJSFCustomComponent` project. The Palette view show the new drawer and the Traffic Light Grid component (Figure 14-13).

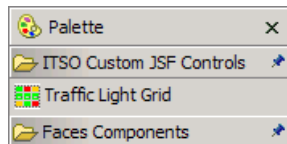


Figure 14-13 Custom component in the palette

Using custom components in a Web application

The plug-in allows you to have the component in the palette and drag and drop it into JSF pages. But you have to install the tag library to use the component in your application.

This installation is done when we create the JSF page in which we are using the custom component. You only have to import the tag library into the project when you create the page:

- ▶ Create a dynamic Web project named **ItsoJSFCustomComponentUse** in the `ItsoJSF14Ext` enterprise application.
- ▶ Create a new JSF page named **customComponentUse.jsp** and select *Configure advanced options*. Click *Next*.
- ▶ In the Tag Libraries page of the wizard, click *Add* and import the tag library (Figure 14-14).

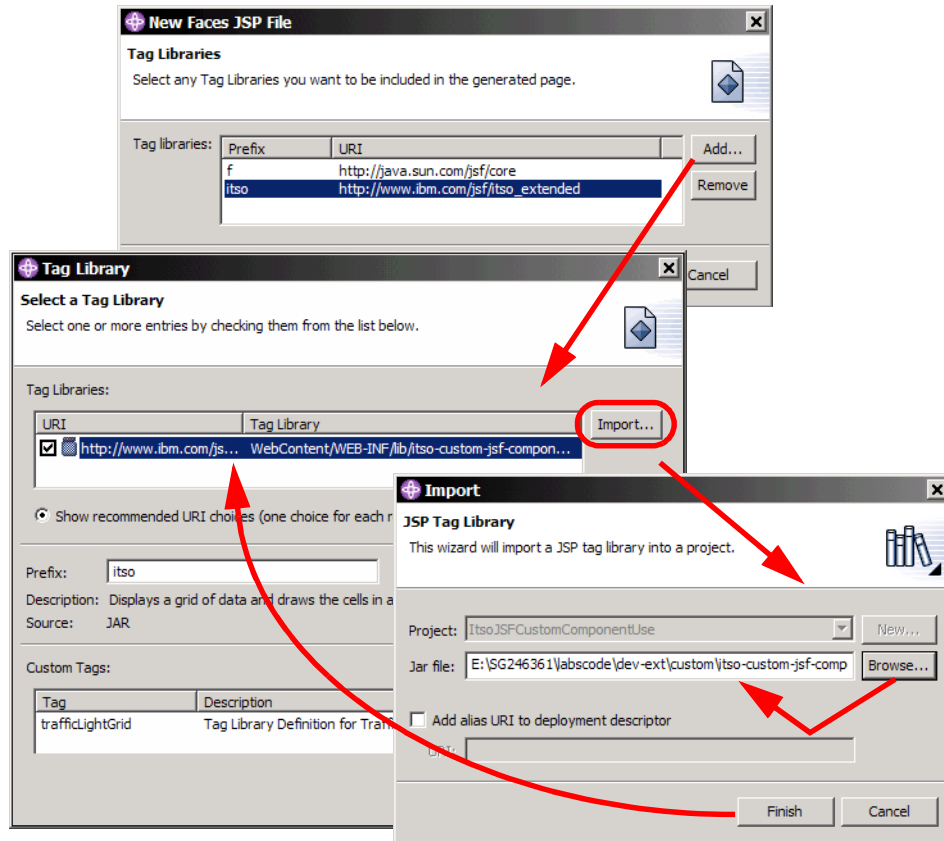


Figure 14-14 Adding a tag library to the project

- Import the custom component JAR file that you built or use the file that we provide:


```
\abscode\dev-ext\custom\itso-custom-jsf-components-1_0_0.jar
```
- After importing, select the tag library from the list and click *OK*.
- The new tag library appears in the list of tag libraries that the project is using.
- ▶ Click *Finish* to create the new page.
- ▶ The tag library is installed in the `WebContent\WEB-INF\lib` folder of the Web project.
- ▶ Apart from importing the library, a new `taglib` tag is added to the JSP:


```
<%@ taglib uri="http://www.ibm.com/jsf/itso_extended" prefix="itso" %>
```

Now we can finish the example and see the custom component working:

- ▶ Import the style sheet (`\labscode\dev-ext\custom\trafficLight.css`) into the theme folder.

Note: If you do not import the style sheet, the component is displayed with no styling at runtime and you will not see all the represented colors.

- ▶ Create a package named `itso.jsf.custom.test` and import the `DataGrid` class (`\labscode\dev-ext\custom\GridData.java`). We use this Java class as a managed bean to populate the component.
- ▶ In the `customComponentUse.jsp` replace the default text with **Custom Component Usage Example** and make it a heading 3.
- ▶ Using the Page Data view, define a JavaBean named `griddata` using the `itso.jsf.GridData` bean. Select *Make this JavaBean reusable* and select *session* scope. Click *Finish*.

- ▶ In the JSF page, drag and drop a new **Traffic Light Grid** component. In the Source page notice the tags that have been generated:

```
<itso:trafficLightGrid max="75" min="25" rowLabels="" colLabels=""
    data="" title="Sales by Region" id="trafficLightGrid1">
</itso:trafficLightGrid>
```

- ▶ Change the parameters of the tag to map their values to the managed JavaBean:

```
<itso:trafficLightGrid
    id="trafficLightGrid1"
    title="#{griddata.title}">
    data="#{griddata.data}"
    colLabels="#{griddata.colLabels}"
    rowLabels="#{griddata.rowLabels}"
    max="320"
    min="200"
</itso:trafficLightGrid>
```

- ▶ Save and test the page. The Traffic Light Grid component should appear in the Web browser (Figure 14-15).

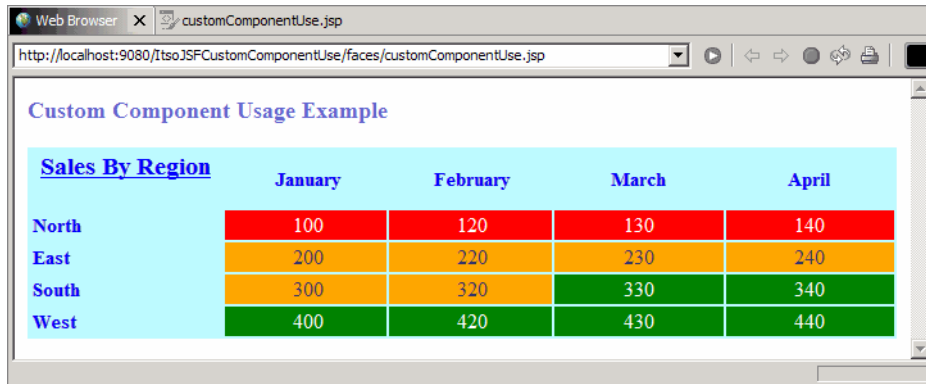


Figure 14-15 Traffic Light Grid component installed and running

Summary

We have developed a tag library to have a custom component available for JSF Web applications. This library allows us to use the custom component in JSF pages.

In addition to the new tag, we have developed a palette extension for it. Using that palette extension, we can design JSF pages and drag and drop the new component visually. This palette extension is a plug-in that is installed into WebSphere Studio Application Developer. This plug-in allows us to use the custom component in the Page Designer in the same way as the JSF standard components.

The combination of the tag library and the plug-in enable us to have custom components available in the palette. Such components help us increase the productivity of Web application development.



Part 5

Appendixes



Installation instructions

This appendix contains the installation instructions for the products used in this redbook:

- ▶ DB2 Universal Database™ Enterprise Edition 8.1.4
- ▶ WebSphere Application Server 5.1
- ▶ WebSphere Application Server Cumulative Fix 3
- ▶ WebSphere Studio Application Developer 5.1.2
- ▶ Portal Toolkit 5.0.2.2

This chapter also contains the instructions for setting up the environment to work with the sample exercises and how to configure WebSphere Application Server 5.1 to deploy the Web applications.

DB2 Universal Database Enterprise Edition 8.1.4

In this section we install the database manager for the sample applications. We install IBM DB2 Universal Database Enterprise Edition 8.1.4 from the installation CD or downloaded code. The installation is straight forward, so you only have to follow the installation panels:

- ▶ Run `setup.exe`. At the welcome page select *Install Product*.
- ▶ Select *DB2 UDB Enterprise Server Edition* and click *Next*. A welcome page to the installation wizard appears. Click *Next*.
- ▶ You must accept the license agreement by selecting *Accept the terms of the license agreement*. Click *Next*.
- ▶ Select typical installation and click *Next*.
- ▶ Select *Install DB2 Enterprise Server Edition on this computer* to continue with the installation. Optionally you can select *Save settings in a response file* for future reference. Click *Next*.
- ▶ Select drive `C:` and enter `C:\SQLLIB\` for the destination folder. Click *Next*.
- ▶ Enter the user information for the DB2 Administration Server. Enter **db2admin** for the user name and **db2admin** for the password. Click *Next*.
- ▶ Select *Local - Create a local list on this system*. Do not select *Enable notification* in the Notification SMTP server frame. Click *Next*. You get a warning. Ignore it and continue with the installation.
- ▶ Leave the default options for the DB2 instances and click *Next*.
- ▶ Select *Do not prepare the DB2 tools catalog* and click *Next*.
- ▶ Select *Defer the task until the installation is complete* and click *Next*.
- ▶ A window with a summary with all the setting appears. Click *Next* to start copying the new files.
- ▶ Click *Finish* in the next window to complete the installation.

WebSphere Application Server

In this section we explain how to install WebSphere Application Server 5.1 for Windows. Follow these steps:

- ▶ Run **install.exe**. The welcome page of the installation wizard appears. Click *Next* to continue with the installation.
- ▶ You must accept the software license agreement by selecting *Accept the terms in the license agreement*. Click *Next*.
- ▶ Select *Full* and click *Next*.
- ▶ Select the installation directories (leave the default options, or enter C:\WebSphere AppServer) and click *Next*.
- ▶ Select **ITSOWAS** for the node name, leave the default system name. Click *Next*.
- ▶ Select *Run WebSphere Application Server as a service* and *Run IBM HTTP Server as a service*. Use an appropriate user and password and click *Next*.
- ▶ A summary appears with all the selections. Click *Next* to install the product.
- ▶ Click *Next* on the Registration page.
- ▶ A summary page appears with the accumulated information. Click *Finish*.

WebSphere Application Server cumulative fix 3 (or fix 4)

We install the cumulative fix number 3 (or number 4) to deploy our JSF Web applications in WebSphere Application Server. To install the fix, follow these steps:

- ▶ Run the `setupCmdLine.bat` command file in the `<was-home>\bin` to setup the required environment variables.
- ▶ Run the `updateWizard.bat` in the installation directory. Select your preferred language for the installation wizard and click *OK*.
- ▶ An information window appears. Click *Next*
- ▶ WebSphere Application Server is found for its update. Click *Next*.
- ▶ Select *Install fix packs*. Click *Next*.
- ▶ Enter the path for the fixes. Let the default selection and click *Next*. The wizard is configured to install the fixes.
- ▶ In the list, select `was510_cf3_win` and click *Next*.
- ▶ An information window appears. Click *Next* to begin the process.
- ▶ Select *Finish* to finish the installation.

WebSphere Studio Application Developer

In this section we show how to install WebSphere Studio Application Developer Version 5.1.2. Follow these steps:

- ▶ Launch `launchpad.exe` from the installation directory or from CD1. The main window of the installation appears (Figure A-1).

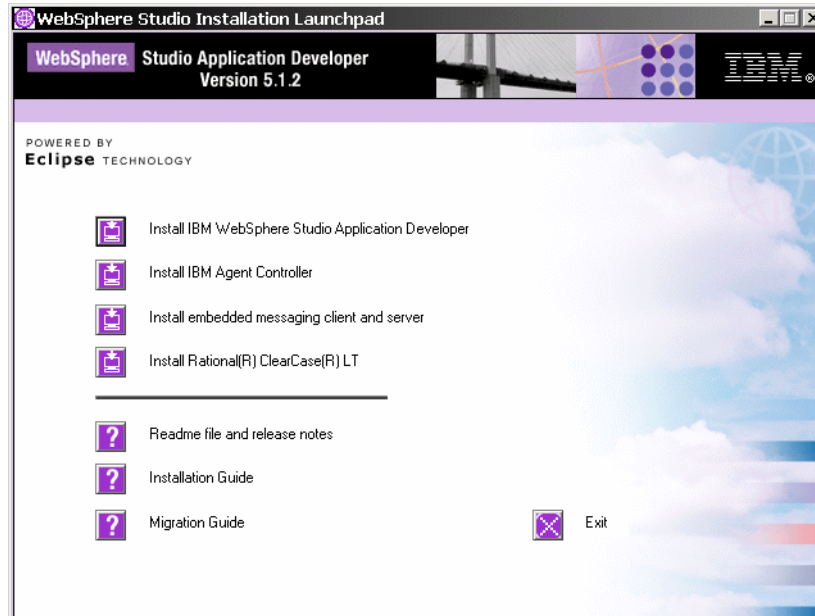


Figure A-1 WebSphere Studio Installation Launchpad

- ▶ Select *Install IBM WebSphere Application Developer* and the WebSphere Application Developer Studio installer starts. Click *Next*.
- ▶ When the license agreement appears, read it. If you accept the terms of the licence, select *I accept the terms in the license agreement* and click *Next*.
- ▶ Select the destination directory for the installation. The default directory is `C:\Program Files\IBM Websphere Studio Application Developer\v5.1.2`. Enter the appropriate directory (for example, `C:\WSAD512`) in the box or browse the hard disk to select the destination and click *Next*.
- ▶ Select the features you may want to install. We recommend select *at least* these integrated test environments:
 - WebSphere Application Server 5.1
 - WebSphere Application Server 5.0.2

For the additional features select:

- Language pack if you are using WebSphere Studio in other languages than English.
- Enterprise Generator Language (EGL) if you are planning to work with backend systems such as a mainframe, or if you want to work through Chapter 12, “JSF and EGL” on page 409.
- Rational® ClearCase® SCM Adapter if you are using ClearCase as the team repository.
- Examples for Eclipse Plug-In Development if you are planning to develop new plug-ins and you want to study examples.

When you have all your selections made, click *Next*. A summary of the selections appears. Click *Next*.

- ▶ The installation process begins. After a few minutes the installer displays the summary of the installation. Click *Finish*.

WebSphere Portal Toolkit

In this section we show how to install WebSphere Portal Toolkit 5.0.2.2. This toolkit requires WebSphere Studio Application Developer 5.1.2 installed on your system. To install this product, follow these steps:

- ▶ Launch `setup.exe` from the CD or the install directory to start the installation wizard, which guides you through the rest of the process. If WebSphere Studio Application Developer is running, please stop it before proceeding with the installation. When you are ready, click *Next*.
- ▶ You must accept the licence agreement by selecting *Accept the terms in the license agreement* and clicking *Next*.
- ▶ Now select the components you want to install (Figure A-2). Select *Portal Toolkit V5.0.0.2* and *WebSphere Portal V5 for Test Environment* and click *Next*.

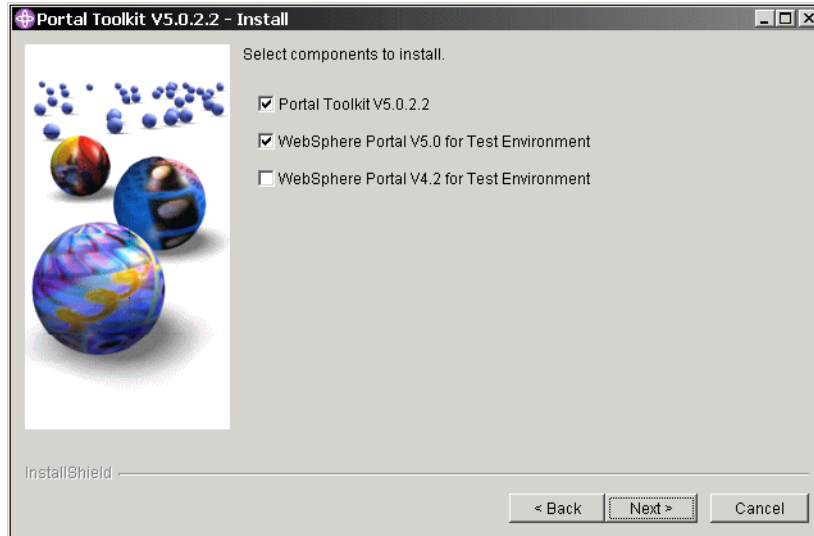


Figure A-2 Selection of the components to install.

Restriction: If you want to install *WebSphere Portal V4 for Test Environment* you must have the runtimes for WebSphere Application Server 4.0 installed on your machine.

- ▶ The wizard automatically detects the location of WebSphere Application Developer Studio and installs the Portal Toolkit in it. Click *Next*.
- ▶ A summary for the installation details appears. Click *Next*.
- ▶ The installation takes about 20 or 30 minutes. After installing the product, the wizard tells you if the product is installed correctly. Click *Finish* to end the wizard.

Setting up the environment for this redbook

In this section we explain how to set up the environment to experiment with the examples covered in the book.

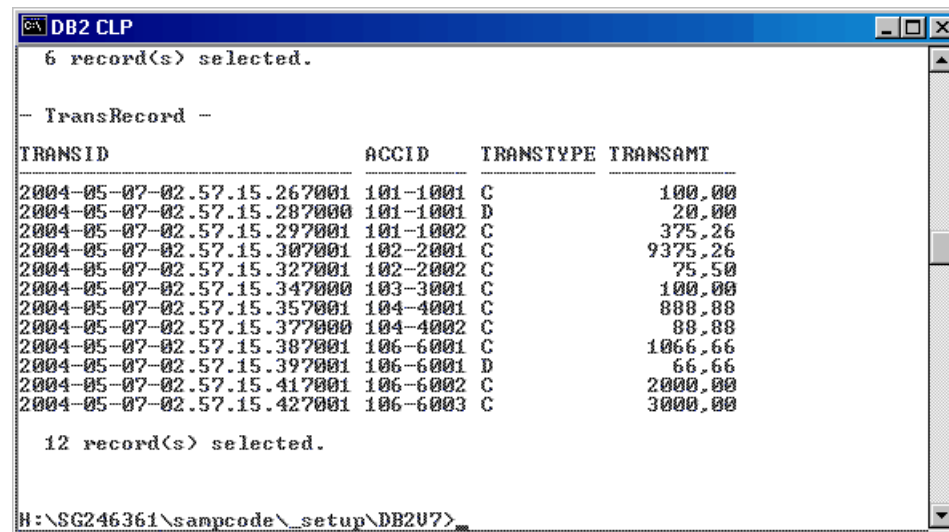
Creating the EJBBANK database

To create the EJBBANK database follow these steps:

- ▶ If you are working with DB2 version 8, locate the directory \labscode_setup\DB2V8. If you are working with DB2 version 7, the directory should be \labscode_setup\DB2V7.
- ▶ Execute the file createbank.bat. This batch program uses the file ejbbank.ddl to create the database and the tables.
- ▶ Execute the file loadbank.bat to populate the database. This file uses ejbbank.sql to insert the sample content in the database. You can rerun this step at any time later to reset the database content to the initial values.

Sample content

To see the content of the EJBBANK database you can execute the file listbank.bat. This batch program uses the file ejbbank.list to retrieve the contents of the sample database. The result is shown in a system window (Figure A-3).



```
DB2 CLP
6 record(s) selected.

-- TransRecord --
TRANSID          ACCID      TRANSTYPE  TRANSAMT
-----
2004-05-07-02.57.15.267001 101-1001 C          100.00
2004-05-07-02.57.15.287000 101-1001 D           20.00
2004-05-07-02.57.15.297001 101-1002 C          375.26
2004-05-07-02.57.15.307001 102-2001 C          9375.26
2004-05-07-02.57.15.327001 102-2002 C           75.50
2004-05-07-02.57.15.347000 103-3001 C          100.00
2004-05-07-02.57.15.357001 104-4001 C           88.88
2004-05-07-02.57.15.377000 104-4002 C           88.88
2004-05-07-02.57.15.387001 106-6001 C          1066.66
2004-05-07-02.57.15.397001 106-6001 D           66.66
2004-05-07-02.57.15.417001 106-6002 C          2000.00
2004-05-07-02.57.15.427001 106-6003 C          3000.00

12 record(s) selected.

H:\SG246361\samrcode\_setup\DB2V7>
```

Figure A-3 Contents of the EJBBANK database

Setting up server targeting

JSF is only supported in WebSphere Application Server Version 5.1. For this reason, our EAR projects must be targeted to use this server. To setup server targeting, follow these steps:

- ▶ In WebSphere Studio Application Developer, select *Window -> Preferences*, then select *J2EE*.
- ▶ The lower frame (at the right side) is named Server Targeting Support (Figure A-4). Select *Enable server targeting support* and click *OK*.

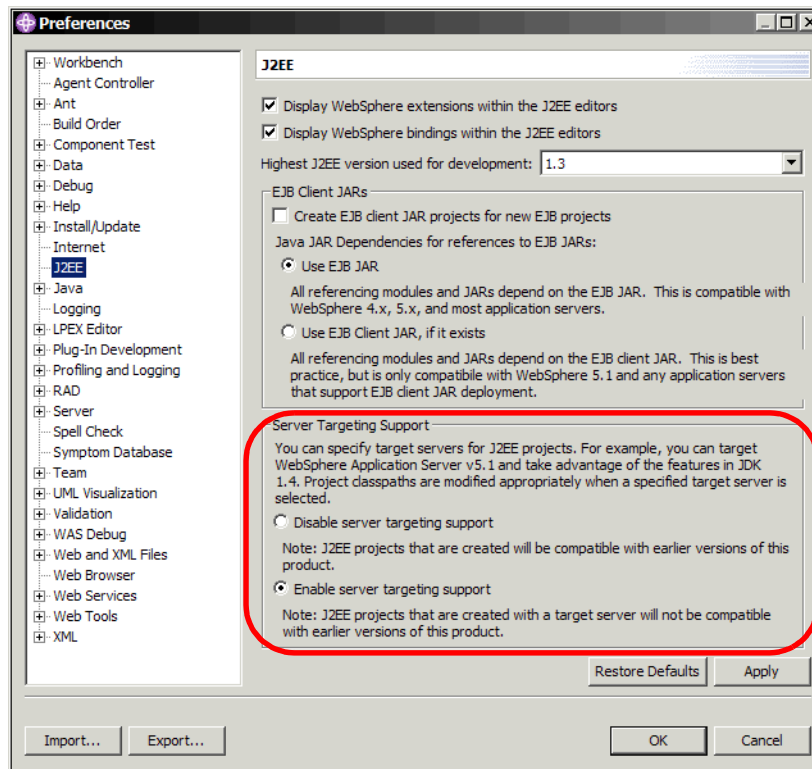


Figure A-4 Enable server targeting support selection

- ▶ With server targeting activated, individual projects can be setup with a target server. We use a Version 5.1 server for all our projects.

Setting up the WebSphere test environment

In this section we explain how to set up a server to test our JSF Web applications.

Defining the JSFServer

We start by creating a server:

- ▶ Open the Server perspective (*Window -> Open Perspective -> Server*).
- ▶ Select *File -> New -> Server and Server Configuration*. Enter **JSFServer** as server name. Leave Servers as folder name. For server type, expand *WebSphere version 5.1* and select *Test Environment*. Click *Finish*.
- ▶ The server then appears in the Server Configuration view (bottom left) and in the Servers view (bottom right) as shown in Figure A-5.

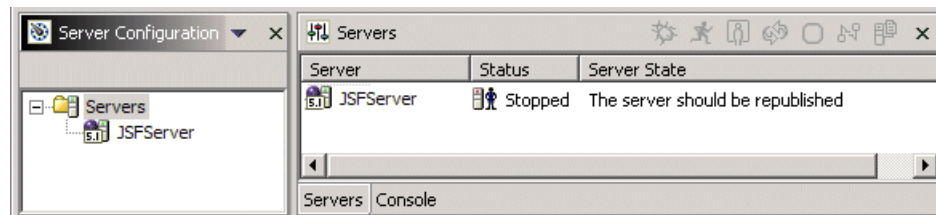


Figure A-5 Server Configuration view and Servers view

Configuring the JSFServer

Next we configure the server to have access the EJBBANK database:

- ▶ Edit the configuration properties by double-clicking the JSFServer in the Server or Server Configuration view.
- ▶ On the Security page click *Add* for JAAS authentication alias (Figure A-6).

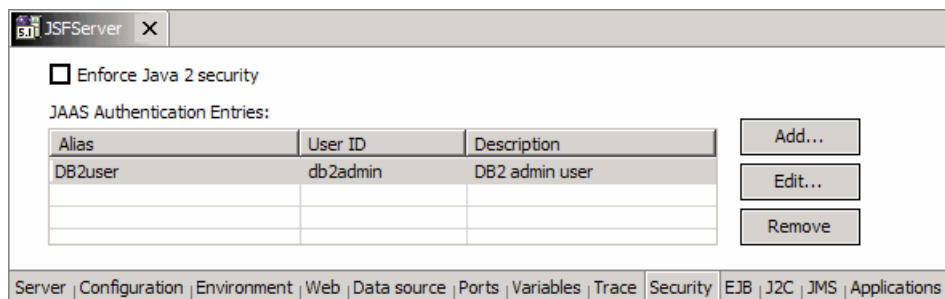


Figure A-6 Defining an authentication alias

- In the dialog, enter **DB2user** as alias and a valid user ID and password (Figure A-7). Click *OK*.

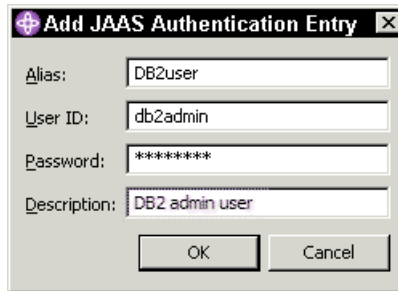
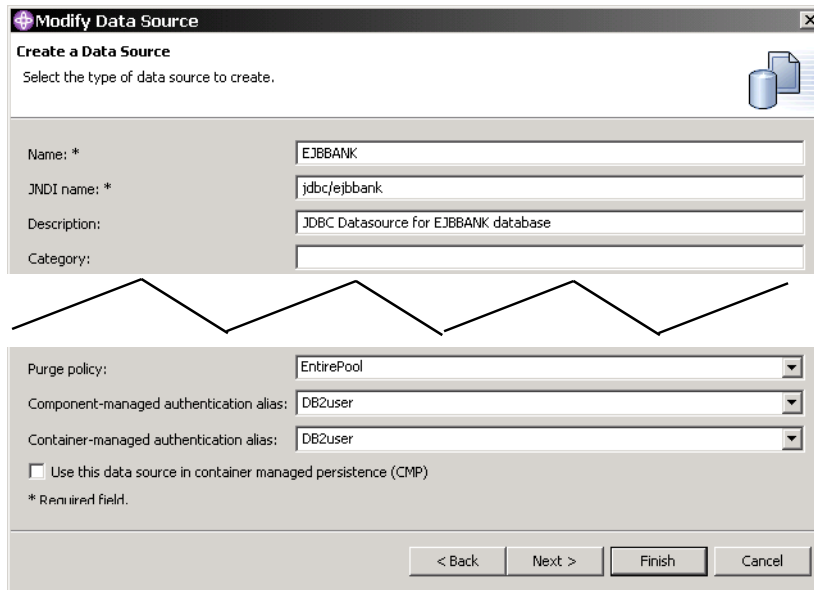


Figure A-7 Entering an authentication alias

- On the Data source page, select the *Default DB2 JDBC Provider*. With the Default DB2 JDBC Provider selected, click *Add* for data source. In the dialog, select *Version 5.0 data source* and *DB2 Legacy CLI-based Type 2 JDBC Driver*. Click *Next*.
- In the next panel (Figure A-8):
 - Enter **EJBBANK** as name and **jdbc/ejbbank** as JNDI name.
 - For *Component-managed authentication alias* and for *Container-managed authentication alias* select **DB2user** from the pull-down.
 - **Deselect** *Use this data source in container managed persistence (CMP)*, because we do not use EJB in our examples.
 - Click *Next*.
 - For the *databaseName* property enter **EJBBANK** as value.
 - Click *Finish*.



Modify Data Source

Create a Data Source
Select the type of data source to create.

Name: * EJBBANK

JNDI name: * jdbc/ejbbank

Description: JDBC Datasource for EJBBANK database

Category:

Purge policy: EntirePool

Component-managed authentication alias: DB2user

Container-managed authentication alias: DB2user

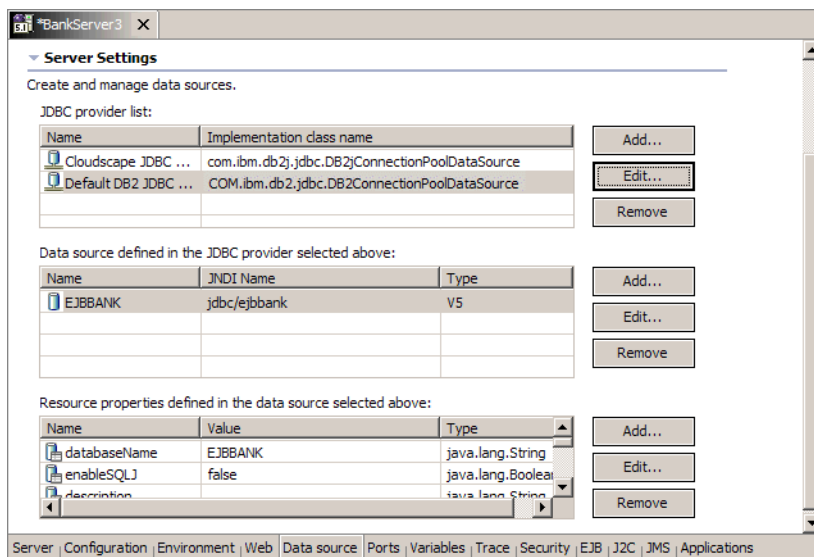
☐ Use this data source in container managed persistence (CMP)

* Required field.

< Back Next > Finish Cancel

Figure A-8 Defining the data source for EJBBANK database

Figure A-9 shows the server settings for data sources. Save and close the server configuration.



BankServer3

Server Settings

Create and manage data sources.

JDBC provider list:

Name	Implementation class name
Cloudscape JDBC ...	com.ibm.db2j.jdbc.DB2jConnectionPoolDataSource
Default DB2 JDBC ...	COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource

Add... Edit... Remove

Data source defined in the JDBC provider selected above:

Name	JNDI Name	Type
EJBBANK	jdbc/ejbbank	V5

Add... Edit... Remove

Resource properties defined in the data source selected above:

Name	Value	Type
databaseName	EJBBANK	java.lang.String
enableSQLJ	false	java.lang.Boolean
description		java.lang.String

Add... Edit... Remove

Server Configuration Environment Web Data source Ports Variables Trace Security EJB J2C JMS Applications

Figure A-9 Data source server settings

Deployment to WebSphere Application Server 5.1

In this section we explain how to configure WebSphere Application Server 5.1 to deploy all the examples covered in this book. We explain how to configure the database and how to install an enterprise application.

Note: We use the JSF calculator enterprise application as an example of how to install and deploy a Web application. To use your own application, export it from WebSphere Studio Application Developer and follow the instructions.

Configure WebSphere Application Server

Before installing any application, we have to configure the database access as we did in with WebSphere test environment (“Setting up the WebSphere test environment” on page 535).

Follow these steps to start the server and the administrative console:

- ▶ Start WebSphere Application Server 5.1 (*Start -> Programs -> IBM WebSphere -> Application Server v5.1 -> Start the server*). You can also open the *First Steps* menu from where you can control the server.
- ▶ Start the administrative console (*Start -> Programs -> IBM WebSphere -> Application Server v5.1 -> Administrative Console*) and login with your normal user ID. Always use the same ID later.
- ▶ Select *Servers* and *Applications* to expand the tree. Select *Application Servers* and you can see the default server (server1). Select *Enterprise Applications* and you can see some sample applications.

Authentication alias

Expand and select *Security -> JAAS Configuration -> J2C Authentication Data* (left side). Click *New*.

- Enter **DB2user** as alias name.
- Enter a valid user ID and password (for example the user ID that was used to install DB2).
- Click *OK*.

JDBC driver path

To set the JDBC driver path:

- ▶ Select *Environment* (left side) and *Manage WebSphere Variables*.
- ▶ Select *DB2 JDBC DRIVER PATH*.

- ▶ Verify or enter the directory of the db2java.zip file (only the directory name, without the db2java.zip file name), for example:

c:\SQLLIB\java or c:\Program Files\SQLLIB\java

- ▶ Click *OK*.

JDBC driver

To define the JDBC driver:

- ▶ Select *Resources* (left side) and select *JDBC Providers*.
- ▶ Select *Node* (should be preselected).
- ▶ Click *New* (under JDBC Providers):
 - Select *DB2 Legacy CLI-based Type 2 JDBC Driver (XA)* from the pull-down and click *Apply*.
 - All the defaults should be fine. Verify that the implementation class is `COM.ibm.db2.jdbc.DB2XADataSource`. Note that the class path entry points to `${DB2_JDBC_DRIVER_PATH}/db2java.zip`.
 - Click *Apply*.

Data source

Now we define the data sources for the JDBC driver:

- ▶ Under Additional Properties select *Data Sources*.
 - Click *New* (Figure A-10).
 - Enter **EJBBANK** as name and **jdbc/ejbbank** as JNDI name (to match what we defined in the Web deployment descriptor).
 - Select *Use this Data Source in container-managed persistence (CMP)*. This may be required for EJBs in other applications.
 - For component- and container-managed authentication alias select the `{node}/DB2user` from the pull-down.
 - Click *Apply*.

Configuration		
General Properties		
Scope	* cells:a23wph18:nodes:a23wph18	<i>i</i> The scope of the configured resource. This value indicates the configuration location for the configuration file.
Name	* EJBBANK	<i>i</i> The required display name for the resource.
JNDI Name	jdbc/ejbbank	<i>i</i> The JNDI name for the resource.
Container managed persistence	☒ Use this Data Source in container managed persistence (CMP)	<i>i</i> Enable if this data source will be used for container managed persistence of EJBs. This will cause a corresponding CMP connection factory which corresponds to this datasource to be created for the relational resource adapter.
Description	New JDBC Datasource	<i>i</i> An optional description for the resource.
Category		<i>i</i> An optional category string which can be used to classify or group the resource.
Statement Cache Size	10 statements	<i>i</i> Number of free prepared statements per connection. This is different from the old datasource which is defined as number of free prepared statements per data source.
Datasource Helper Classname	com.ibm.websphere.rsadapter.DB2C	<i>i</i> The datastore helper that is used to perform specific database functions.
Component-managed Authentication Alias	FUNDY.DB2user	<i>i</i> References authentication data for component-managed signon to the resource.
Container-managed Authentication Alias	FUNDY.DB2user	<i>i</i> References authentication data for container-managed signon to the resource.
Apply OK Reset Cancel		
Additional Properties		
Connection Pool	An optional set of connection pool settings.	
Custom Properties	Properties that may be required for Resource Providers and Resource Factories. For example, most database vendors require additional custom properties for data sources that will access the database.	
Related Items		
J2C Authentication Data Entries	Specifies a list of userid and password for use by Java 2 Connector security.	

Figure A-10 Data source for the EJBBANK database

- Under Additional Properties select *Custom Properties*, then select *databaseName*, enter **EJBBANK** as value and click **OK**.

- ▶ Define a second data source with name **EJBANKsdo** and JNDI name **jdbc/EJBANK_Con1**.
 - **Deselect** *Use this Data Source in container-managed persistence (CMP)*.
 - For component- and container-managed authentication alias select the {node}/DB2user from the pull-down.
 - Click *Apply* and under Additional Properties select *Custom Properties*, then select *databaseName*, enter **EJBANK** as value and click *OK*.

Every data source that was dynamically created for SDO must be defined in the server.

Save the configuration

When you configure the server for JDBC and data source, you have to save the configuration:

- ▶ Select *Save* in the menu bar, then click *Save* and wait for the confirmation message.
- ▶ Select *Logout* in the menu bar.

Stop and start the server

Now you can stop and start the server to activate the configuration changes. Restart the Administrative Console to see the changes and to install your enterprise applications.

Installing an enterprise application

Install the enterprise applications using the Administrative Console:

- ▶ Export an enterprise application from WebSphere Studio using *File -> Export -> EAR file*. Select an enterprise application and a target directory.
- ▶ Copy the exported EAR file into <was-home>\installableApps. You can use our exported files, for example, \labscode\solution\ItsoJSF03Calc.ear.
- ▶ Select *Applications* and *Install New Application*.
- ▶ For local path click *Browse* and locate the EAR file that you exported from Application Developer (Figure A-10):

<WAS-home>\installableApps\ItsoJSF03Calc.ear

Click *Next*.

Path: Browse the local machine or a remote server:

☒ Local path: er\installableApps\itsoJSF3.ear

☐ Server path:

Context Root: Used only for standalone Web modules (*.war)

i Choose the local path if the ear resides on the same machine as the browser, the server path if the ear resides on any of the nodes in your cell context.

i You must specify a context root if the module being installed is a WAR module.

Figure A-11 Enterprise application installation

- ▶ On the Preparing for the application installation panel all the defaults should be fine:
 - You do not want to overwrite existing bindings, everything was configured in Application Developer
 - You can choose the virtual host (instead of default_host)
- Click *Next*.

Installation process: Multiple-steps

Installation of an enterprise application is a multi-step process. The actual number of steps vary depending on the content and requirements of the application.

Review the options and then click *Next* for each step. Because we configured all the deployment information in Application Developer, there is no need to change the options. Click *Finish* on the Summary page and be patient. Messages are displayed in the console.

Save the configuration (click *Save* in the top menu).

SDO runtime

For projects that use Service Data Objects, we have to make the SDO runtime available:

- ▶ A simple way is to copy the SDO runtime JAR files (all the JAR files):

```
From: <wsadhome>\runtimes\base_v51\optionalLibraries\WDO\
To:   <washome>\lib\ext\
```

This process makes the runtime files available to all enterprise applications.

Be careful when applying updates to the Application Server. Updated runtime files have to be copied again.

- An alternative is to define a shared library. Expand *Environment* -> *Shared Libraries* and click *New*. Define the shared library as:

```
Name = WDO Relational Mediator Shared Library
Classpath = ${WAS_INSTALL_ROOT}/optionalLibraries/WDO/wdo-interface.jar
${WAS_INSTALL_ROOT}/optionalLibraries/WDO/emf-runtime.jar
${WAS_INSTALL_ROOT}/optionalLibraries/WDO/emf-event.jar
${WAS_INSTALL_ROOT}/optionalLibraries/WDO/jdbcmediator.jar
${WAS_INSTALL_ROOT}/optionalLibraries/WDO/wdo.jar
${WAS_INSTALL_ROOT}/optionalLibraries/WDO/wdo.xmlmediator.jar
```

Click *OK*.

Select each enterprise application with SDO (under *Applications* -> *Enterprise Applications*) and update the properties:

- Switch the Applications Classloader Mode to *PARENT_LAST*.
- Under Additional Properties select *Libraries* and click *Add*. Select the shared library and click *OK*.

Starting the enterprise application

Expand *Applications* and select *Enterprise Applications*.

Select and start the *ItsoJSF03Calc* enterprise application.

Testing the enterprise application

To test the application, open a browser and enter its URL. The URL for the JSF calculator example is:

```
http://localhost:9080/ItsoJSF03Calc/faces/calculate.jsp
```

For most other applications the URL to run the application is:

```
http://localhost:9080/ItsoXXXXXX/faces/index.jsp
```




B

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246361>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-6361.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
sg2466361code.zip	Sample code for following the samples in the book
ItsoXxxxxx.ear	Exported EAR files of example enterprise applications
corrections.txt	Corrections to the book after publishing

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space	3 GB
Operating System	Windows 2000 or Windows XP
Processor	1 GHz or better
Memory	1 GB or better

How to use the Web material

Unzip the contents of the Web material **sg2466361code.zip** file onto your hard drive. This creates a folder structure **c:\SG2466361\labscode\xxxx**, where **xxxx** refers to a chapter or activity in the book. Place all **ItsoXxxxxx.ear** files into the **solutions** subdirectory.

_setup	Database setup for EJBBANK database
dev-calculator	Sample code of the JSF calculator example
dev-component	Sample code for JSF components
dev-ext	Sample code for JSF custom extensions
dev-java	Sample code for the banking model
dev-jsfbank	Sample code for the JSF banking application
dev-jsfclient	Sample code for the JSF Client framework
dev-jsfegl	Sample code for the JSF banking application using EGL
dev-jsfwebserv	Sample code for the JSF Web service example
dev-patterns	Sample code for the JSF patterns
dev-portlet	Sample code for JSF portlet applications
dev-sdoapi	Sample code for SDO API
dev-sdobank	Sample code for the JSF banking application using SDO
images	Images used in several Web applications
solution	Exported EAR files from Application Developer

Workspace projects

Table B-1 lists the sample projects developed in this redbook. Projects prefixed with a dash (-) belong to the enterprise application listed above the project.

Table B-1 Application Developer projects

Name	Description
ItsoJSF03Calc	EAR file for chapter 3
- ItsoJSFCalc	Web project with JSF calculator
ItsoJSF04Comp	EAR file for chapter 4
- ItsoJSFComponents	Web project with JSF component examples
ItsoJSF05Bank	EAR file for chapter 5
- ItsoJSFBank - ItsoBankModel	Web project with JSF banking application Java project with banking model
ItsoJSF06Pattern	EAR file for chapter 6
- ItsoJSFPatterns	Web project with JSF patterns
ItsoJSF07WebServ	EAR file for chapter 7
- ItsoBankingWebService - ItsoJSFWebService	Web project with existing Web service Web project with JSF using the Web service
ItsoJSF10SDO	EAR file for chapter 10
- ItsoSD0Bank	Web project with JSF banking using SDO
ItsoJSF11Portlet	EAR file for chapter 11
- ItsoJSFPortletCalc - ItsoJSFPortletBank - ItsoBankModel	Web project with JSF calculator as portlet Web project with JSF banking as portlet Java project with banking model
ItsoJSF12EGL	EAR file for chapter 12
- ItsoJSFEGL	Web project with JSF banking using EGL
ItsoJSF13Client	EAR file for chapter 13
- ItsoJSFClientComponents - ItsoBankingWebService	Web project with JSF Client component examples Web project with existing Web service
ItsoJSF14Ext	EAR file for chapter 14
- ItsoJSFCustomComponent - ItsoJSFCustomComponentUse	Web project for developing a custom component Web project using the custom component

Application Developer setup

Before working with the sample code, follow the instructions in “Setting up the environment for this redbook” on page 533. This includes:

- ▶ Creating and populating the EJBBANK database
- ▶ Setting up server targeting
- ▶ Setting up and configuring the WebSphere test environment (JSFServer)

Importing a solution enterprise application

To import an entire enterprise application project into your Application Developer workspace follow these steps:

- ▶ Select *File -> Import*. Select *EAR file* and click *Next*.
- ▶ Browse your disk and find the file `\labscode\solution\ItsoXXXXXX.ear`. The target project name is selected for you.
- ▶ Click *Finish*.

Now you have the complete example of one chapter in your workspace.

Deploying enterprise applications

You can deploy the enterprise applications to a WebSphere Application Server by following the instructions in “Deployment to WebSphere Application Server 5.1” on page 538.

Abbreviations and acronyms

AAT	Application Assembly Tool	JMX	Java Management eXtension
API	Application programming interface	JNDI	Java Naming and Directory Interface
CORBA	Common Object Request Broker Architecture	JSF	JavaServer Faces
CRM	Customer relationship management	JSL	JavaScript Library
DBMS	Database management system	JSP	JavaServer Page
EAR	Enterprise application archive	JSR	Java specification request
EIS	Enterprise information system	MVC	Model-view-controller
EJB	Enterprise JavaBeans	ODC	On Demand Client
EJS	Enterprise Java Server	RAD	Rapid application development
EMF	Eclipse Modeling Framework	RAR	Resource adapter archive
ERP	Enterprise resource planning	RDBMS	Relational database management system
FTP	File Transfer Protocol	RPC	Remote procedure call
GUI	Graphical user interface	SDK	Software Development Kit
HTML	Hypertext Markup Language	SDO	Service Data Objects
HTTP	Hypertext Transfer Protocol	SEI	Service endpoint interface
IBM	International Business Machines Corporation	SOA	Service oriented architecture
IDE	Integrated development environment	SOAP	Simple Object Access Protocol (also known as Service Oriented Architecture Protocol)
ITSO	International Technical Support Organization	SQL	Structured query language
J2EE	Java 2 Enterprise Edition	SSL	Secure Sockets Layer
J2SE	Java 2 Standard Edition	TCP/IP	Transmission Control Protocol/Internet Protocol
JAR	Java archive	TLD	Tag library definition
JAX-RPC	Java APIs for XML based RPC	UDDI	Universal description, discovery, and integration
JCP	Java Community Process	URL	Uniform resource locator
JDBC	Java Database Connectivity	URN	Uniform resource name
JDK	Java Developer's Kit	UTC	Universal test client
JMS	Java Messaging Service	UUID	Universal unique identifier

WAR	Web application archive
WDO	WebSphere Data Objects
WSDL	Web service description language
WS-I	Web services interoperability
WSIF	Web services invocation framework
WSIL	Web services inspection language
WWW	World Wide Web
XMI	XML metadata interchange
XML	eXtensible Markup Language
XSD	XML schema definition

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 553. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook*, SG24-6891
- ▶ *WebSphere Version 5 Application Development Handbook*, SG24-6993
- ▶ *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957
- ▶ *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- ▶ *IBM WebSphere Application Server V5.1 System Management and Configuration WebSphere Handbook Series*, SG24-6195
- ▶ *IBM WebSphere V5.0 Security: WebSphere Handbook Series*, SG24-6573
- ▶ *WebSphere Studio Application Developer Programming Guide*, SG24-6585
- ▶ *Web Services Wizardry with WebSphere Studio Application Developer*, SG24-6292
- ▶ *Self-Study Guide: WebSphere Studio Application Developer and Web Services*, SG24-6407
- ▶ *IBM WebSphere Portal V5: A Guide for Portlet Application Development*, SG24-6076
- ▶ *Portal Application Design and Development Guidelines*, REDP-3829
- ▶ *Architecting Portal Solutions*, SG24-7011
- ▶ *Legacy Modernization with WebSphere Studio Enterprise Developer*, SG24-6806 (information about EGL)

Other publications

These publications are also relevant as further information sources:

- ▶ *JavaServer Faces*, by Hans Bergsten. O'Reilly and Associates, 1st Edition, April 2004. ISBN 0596005393.
- ▶ *JavaServer Faces Programming*, by Budi Kurniawan. McGraw-Hill Osborne Media, 1st Edition, October 2003. ISBN 0072229837.

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ IBM Web sites
<http://www-136.ibm.com/developerworks/>
<http://www-106.ibm.com/developerworks/java/library/j-integrate>
- ▶ Sun Java Web site
<http://java.sun.com/>
<http://java.sun.com/j2ee/javaxserverfaces/index.jsp>
- ▶ Java community process
<http://www.jcp.org/en/jsr/detail?id=127>
<http://www.jcp.org/en/jsr/detail?id=235>
- ▶ Eclipse
<http://www.eclipse.org/articles/Article-UI-Guidelines/v200202/Contents.html>
- ▶ Apache Open Source
<http://cvs.apache.org/builds/jakarta-struts/nightly/struts-faces>
<http://struts.apache.org/>
- ▶ XMethods
<http://www.xmethods.net>
- ▶ XPath
<http://www.w3.org/TR/xpath>
- ▶ Object Management Group
<http://www.omg.org/>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

.jspPersistence 58
.wdo-connections 308
{Error Messages} 65

A

accountDetails.jsp 195, 211, 359
AccountDetailsAction 194
AccountListingSQL 283
accountSelection.jsp 374
accountsOfCustomer.jsp 376
AccountValidator 222
action
 binding 40
 code
 SDO 322
 list 49
 logic
 EGL 420
 SDO 352
ActionEvent 19
administrative console 538
All tab 40, 43
AmountValidator 222
Ant
 build script 503, 518
 run 504
API
 SDO 329
application
 architect 228
 architecture 229–230, 342
 design 227
 developer 6, 228
 JSF 8
 scope 234
 Web 8
 Web service 282
Application Developer
 setup 548
apply request values 11
architecture
 banking application 192

Attributes view 39, 93
authentication alias 538

B

banking application 191
 SDO 341
Banking class 193
BankJDBC 193–194
bar chart 472
batching 440
binding 19, 40
BPEL 288
broken link 55
browser
 framework 21
build
 descriptor 413
 script 503
business logic tier 231
business process execution language 288

C

calculate 73
calculate.jsp 63
calculator
 portlet 387
CalculatorBean 62–63, 71
 portlet 389
Check Box 130
Check Box Group 132
client
 component
 event 450
 data
 binding 448
 object 447
Client Data view 49, 444
client side data caching 385
Combo Box 134
command
 event 32
Command - Button 32, 147
Command - Hyperlink 124

- example 209, 358, 371
- commit 337, 369
- common
 - attributes 89
 - events 94
 - patterns 227
- commons-xxx.jar 64
- component 159
 - attributes 67
 - Check Box 130
 - Check Box Group 132
 - Combo Box 134
 - Command - Button 32, 147
 - Command - Hyperlink 124
 - custom 237
 - Data Grid 450
 - Data Table 141
 - delete 57
 - developer 6, 228
 - Display Error 118
 - Display Errors 120
 - drawer 37
 - Error Message 43
 - File Upload 155
 - Form 102
 - Generic A/V Player 177
 - Graph 453
 - Horizontal Rule 159
 - HTML 101
 - Input 108
 - Input - Hidden 114
 - Input - Password 111
 - Input - Text Area 115
 - JavaBean 46
 - Label 122
 - libraries 20, 95
 - Link 150, 153
 - List box - Multiple Select 139
 - List Box - Single Select 136
 - Macromedia Flash Player 178
 - Macromedia Shockwave Player 180
 - model 86
 - Output 106
 - Output - Formatted Text 106
 - Panel Group Box - List 162
 - Panel Group Box - Snap to Border 160
 - Panel Menu Bar 164
 - Panels - Tabbed 167
 - Radio Button Group 127
 - RealOne Player 182
 - Relational Record 314
 - Relational Record List 314
 - Rich Text Area 174
 - state 86
 - tree 21
 - Tree View 454
 - visualizer 519
 - Web Service 279, 456
- component overview 86
- concurrency 255, 307
- configuration
 - portlet 384
- configuration file 9, 13, 31
- configure
 - server 538
- connection
 - create 312
 - development 311
 - dialog 305
 - information 308
 - runtime 312–313
 - SDO 303
 - wrapper 337
- constants_en.properties 82
- context
 - Edit Faces Command Event 36
 - Edit Faces Value Changed Event 36
 - View Page Code 36
- controller 4–5
- conversion 11, 236
- converter 20, 42
 - custom 481, 506
 - example 507
 - interface 506
 - register 506
- core library 20, 97
- CRUD 239
 - create 252
 - design 242
 - JSF pages 245
 - navigation 249
 - test 255
- cumulative fix 529
- custom
 - component 237, 480
 - installation 520
 - testing 519
 - converter 481, 506

- palette category 510
- renderer 237, 481, 495
- UI component tags 481
- validation 222
- validator 481
- customComponent.jsp 484
- customComponentUse.jsp 520
- CustomerData 201
- customerUpdate.jsp 372
- customize palette 38

D

- data
 - access tier 232
 - binding
 - client 440
 - conversion 236
 - drawer 37
 - graph 300–301, 330
 - mediator 300–301, 331
 - object 300
 - source
 - name 309
 - server 539
 - update 320
 - validation 236
- Data Grid 450
 - example 457
 - update 460
- Data perspective 303, 312
- Data Table 141
 - extensions 186
 - SDO 356
- database tier 232
- DataGraphAccessBean 330
- dataGrid.jsp 457
- DataObject 329
- DB Servers view 312
- DB2
 - installation 528
- debugging
 - EGL 421
- defaultSelectCondition 424
- delete
 - component 57
 - managed bean 57
 - page 55
 - prompt 323

- delta 440
- deluxe pager 187
- deployment 538
- deposit 193
- design
 - considerations 232
- development
 - custom component 485
- DiffGram 441
- DiffHandler 442, 462
- DiffInfo 462
- DiffProcessor 465
- Display Error 118
- Display Errors 120
- drag and drop 37, 49
- dynamic
 - array 425
 - queries 333

E

- Eclipse Modeling Framework
 - see EMF
- EGL
 - action logic 420
 - build descriptor 413
 - compile 415
 - concepts 410
 - considerations 438
 - debug 421
 - editor 417
 - function 418
 - generate 433
 - JSF application 411
 - JSF page 414
 - page handler 410
 - preferences 411
 - retrieve 424
 - SQL record 416
 - Web Exercises 409
 - Web project features 413
- EGL Web Project 410
- EJBBANK database 194, 196
 - create 533
 - data source 536
- EJBBANK_Con1 348
- EMF
 - artifacts 446
 - mapping 446

- model 446
- emitter 441
- encodeURL 405
- enterprise application
 - installation 541
- Enterprise Generation Language
 - see EGL
- error message
 - field 68
- Error Message component 43
- error.jsp 75, 353
- event 9
 - action 19
 - common 94
 - JavaScript 51
 - listener 19
 - logic 29
 - processing 12
 - value change 80
 - value changed 19
- event-driven model 4
- exceptions 194
- Extension Library 146
- ExternalContext 10

F

- facade 193
- Faces
 - Client Components 37
- Faces Client
 - application
 - Data Grid 457
 - Graph 468
 - Tree View 472
 - Web Service 475
 - components 450
 - concepts 440
 - framework 439
 - model 442
 - page 443
 - tools 443
- Faces Client component framework 49
- faces-config.xml 13, 17, 29
 - custom component 498
 - example 77
- faces-config.xml.portlet 401
- FacesContext 63
- FacesServlet 5, 9–10, 13

- File Upload 155
- filter 307
- findComponent 63
- findComponentInRoot 66
- Form 102
- form wizard 257
- Format tab 41
- fragment 33

G

- generate key 307
- Generic A/V Player 177
- getAccount 193
- getAccounts 193
- getAsObject 506
- getAsString 506
- getCustomer 193
- getTransactions 193
- global rules 41
- goto pager 187
- Graph 453
 - example 468
- graph 301
- graph.jsp 468
- grouping function 471
- guideline 54

H

- Horizontal Rule 159
- HTML
 - components 101
 - extension library 146
 - library 20, 101
- HTML Extensions 96
- HttpServletRequest 17
- HTTPSession 17
- hyperlink 209

I

- Image 153
- independence 10
- index.egl 414
- index.jsp 195, 201, 346
- Input 108
- Input - Hidden 114
- Input - Password 111
- Input - Text Area 115

- installation
 - Application Developer 530
 - Application Server 529
 - custom component 520
 - DB2 528
 - enterprise application 541
 - instructions 527
 - Portal Toolkit 531
- integration library 293
- internationalization 4, 236
 - example 82
- invoke application 12
- ItsoBankingWebService 282, 475
- ItsoBankModel 198
- ItsoJSF03Calc 63
- ItsoJSF05Bank 199
- ItsoJSF06Pattern 242
- ItsoJSF07WebServ 282
- ItsoJSF10SDO 345
- ItsoJSF11Portlet 388, 399
- ItsoJSF12EGL 412
- ItsoJSF13Client 457
- ItsoJSF14Ext 484
- ItsoJSF14Plugin 516
- ItsoJSFBank 199
- ItsoJSFCalc 62
- ItsoJSFClientComponents 457
- ItsoJSFCustomComponent 484
- ItsoJSFCustomComponentUse 520
- ItsoJSFEGL 412
- ItsoJSFPatterns 242
- ItsoJSFPlugin 510
- ItsoJSFPluginTestWeb 516
- ItsoJSFPortletBank 400
- ItsoJSFPortletCalc 388
- ItsoJSFSDO 345
- ItsoJSFWebService 282

J

- J2EE 8
- JAAS authentication alias 535
- JAAS entry 310
- JAR
 - JSF 64
- Java
 - event 50
 - Resources 28
- Java Community Process 7

- Java Specification Request
 - see JSR
- JavaBean
 - component 46
 - properties 47
- JavaScript 21, 440
 - event 51
 - Library 441
- JavaServer Faces
 - see JSF
- JCP 7
- JDBC
 - driver 539
 - driver path 538
 - mediator 301, 310
- jdbc/ejbbank 200, 398
- jdbc/EJBANK_Con1 352
- JNDI name 200
- JS Resource Servlet 64
- JSF
 - application 8
 - Application Developer 26
 - banking application 191
 - run 225
 - calculator 61
 - component 85
 - libraries 20
 - concepts 8
 - configuration file 13
 - converter 20, 42
 - custom component 479
 - debugging 52
 - delete 55
 - extension 479
 - FacesServlet 10
 - framework 4
 - introduction 4
 - JAR files 64
 - library 87
 - move 55
 - page 9
 - create 29
 - example 21
 - phases 10
 - processing 13
 - project 27
 - scenario 7
 - source 35
 - specification 7

- Struts 289
- validator 20
- Web services 277
- jsf-api.jar 64
- jsf-ibm.jar 64
- jsf-impl.jar 64
- JSFPortalServer 394
- JSFPortletServer 396
- JSFServer 62
 - configure 535
 - create 535
- JSL 441
- JSP
 - scripting 44, 47
 - SDO 326
- JSR
- JSR-127 7
- JSR-168 383
- JSR-235 300
- jstl 64
- jstl.jar 64
- jstl_el.jar 64

K

- key generation 253, 325

L

- Label 122
- line chart 470
- Link 150
- List Box - Multiple Select 139
- List Box - Single Select 136
- listAccounts.jsp 195, 206, 353
- ListAccountsAction 194, 205
- ListTransactionAction 194
- listTransactions.jsp 195, 217, 367
- log 64

M

- Macromedia Flash Player 178
- Macromedia Shockwave Player 180
- managed
 - bean 9, 17
 - delete 57
 - refactoring 56
 - properties 47
 - property 18

- mask converter 42
- Master.css 63
- mediator 300
 - JDBC 301
- MediatorAccessBean 331
- method
 - applyChanges 323
 - autoGenerateKey 336
 - binding 19, 92
 - calculate 73
 - findComponent 63
 - findComponentInRoot 66
 - getAsObject 506
 - getAsString 506
 - log 64
 - onPageLoadBegin 84
 - saveAllToForm 464
 - validate 78
- mode
 - portlet 384
- model 4–5
- model-view-controller
 - see MVC
- move
 - page 55
- Multi-Column Data Table 317
- MVC 192, 290
 - design pattern 232

N

- navigation 4
 - case 14
 - portlet 384
 - rule 14, 54
 - rules 41
 - scenario 15
- Navigation tab 41
- New Faces JSP File wizard 30
- nonJSF.jsp 326

O

- ODC
 - taglib 447
- OddValidator 78
- OdysseyBrowserFramework 447
- On Demand Client
 - see ODC
- onPageLoad 415, 423

- onPageLoadBegin 84
- optimistic concurrency 255
- Output 106
- Output - Formatted Text 106

P

- package JSF components 503
- packaging script 503
- page
 - author 6, 21, 228
 - code 29
 - move 56
 - delete 55
 - handler 410
 - initialization 83
 - move 55
 - navigation 76
 - welcome 58
- Page Data view 32, 43, 315
- Page Designer 34
 - modes 35
- page statistics 187
- pagecode
 - package 56
 - see page code
- PageCodeBase 33, 56, 63
- paging 186
 - client 440
- palette
 - custom component 510
 - customize 38
 - extension 512, 520
 - settings 38
- Palette view 36
- Panel Group - HTML Panel 159
- Panel Group Box - HTML Panel 159
- Panel Group Box - List 162
- Panel Group Box - Snap to Border 160
- Panel Menu Bar 164
- Panels - Tabbed 167
- PARENT_LAST 450
- passive connection wrapper 337
- pattern
 - CRUD 242
 - form to form 259
 - master to detail 268
- patterns 238
- performance 229

- PerformTransactionAction 194
- perspective
 - Plug-in Development 513
 - Web 34
- phase 10
 - listener 14
- pie chart 472
- plug-in
 - deployment 517
 - project 510
 - testing 516
- Plug-in Development perspective 513
- pluginTest.jsp 516
- portability 230, 233
- portal 382
- Portal Toolkit 381
- portlet 382
 - API 383
 - application 382
 - banking application 396
 - calculator 387
 - client side data caching 385
 - configuration 384
 - file 401
 - fixing JSF pages 403
 - fragment 403
 - JSP models 385
 - mode 384
 - SDO support 386
 - server 394
 - style sheet 405
 - testing 404
 - unsupported components 386
 - URL 405
 - validation 385
- Portlet API 383
- Portlet Development project 387
- Portlet perspective 395
- preferences 52
 - EGL 411
- presentation tier 231
- preview 35
- primary key 307
- process validations 11
- processing
 - JSF 13
- ProductsCreate.jsp 252
- ProductsRead.jsp 245
- ProductsUpdateDelete.jsp 250

- ProductView.jsp 270
- project
 - features 28
 - JSF 27
 - portlet 387
 - rename 58
- Project Navigator view 28
- properties
 - managed 47
- prototyping 6

Q

- query
 - dynamic 333
 - SDO 330
- Quick Edit view 50, 95

R

- RAD
- Radio Button Group 127
- range validator 20
- rapid application development
 - see RAD
- rapid paging 440
- RealOne Player 182
- record
 - delete 335
 - insert 335
 - properties 303
 - SDO 303
 - update 334
- Redbooks Web site 553
 - Contact us xxiv
- refactoring 56
- reference
 - data source 309
- RefreshAccountAction 194
- register
 - converter 506
- Relational Record 46, 301, 314
- Relational Record List 302, 314
- relationship 307
- rename
 - project 58
- render
 - response 12
- render kit 87
- renderer 18, 87, 495

- custom 237
- rendering 18
- request
 - scope 234
 - variable 55
- resource reference 309
- restore component tree 11
- Rich Text Area 174
- Rich Text Editor 96
- rollback 337
- row
 - action support 186
 - example 220
 - category support 186
 - example 376
 - edit support 186
 - example 372
 - select support 186
 - example 374
- rule
 - navigation 14

S

- saveAllToForm 464
- scalability 229
- scenario 7
- scope
 - application 234
 - request 234
 - session 234
 - variable 44, 55
- SDO 299
 - account list 354
 - action code 322
 - API 329
 - architecture 300
 - banking application 341
 - components 314
 - connection 303
 - create 323
 - currency 307
 - customer name 347
 - data source 311
 - Data Table 356
 - delete component 325
 - filter 307
 - for JavaScript 441
 - insert 323

- introduction 300
- key generation 307
- metadata file 307
- object 313
- portlet 386
- query syntax 330
- record 303
- regular JSP 326
- relationship 307
- runtime 542
- specification 300
- wizard 302
- SDO4JS 441
 - emitter 446
- server
 - data source 539
 - portlet 394
 - targeting 52, 534
- Server Configuration view 53
- Servers view 73
- Service Data Objects
 - see SDO 299
- ServletContext 17
- session
 - scope 234
- sg2466361code.zip 546
- simple pager 187
- Single Column Data Table 317
- slush bucket 240
- snippet 51
- source 35
- specification 7
- Struts 289
 - architecture 290
 - comparison 291
 - conversion 296
 - introduction 290
- Struts-Faces
 - integration library 292
 - request 294
- struts-faces.jar 294
- style sheet 63, 65, 248
 - portlet 405
- Styles view 65
- stylesheet.css 29, 65, 248
- Symbol font 477
- synchronization 440

T

- Tabbed Panel 96
- table
 - maintenance 372
- tag
 - f
 - actionListener 98
 - attribute 98
 - convertDateTime 98
 - converter 98, 509
 - convertNumber 70, 98
 - facet 98
 - parm 99
 - selectItem 70, 99
 - selectItems 99
 - subView 99
 - validateDoubleRange 99
 - validateLength 100
 - validateLongRange 70, 100
 - validator 100
 - valueChangeListener 100
 - verbatim 100
 - view 22, 100
 - h
 - commandLink 23, 124
 - dataTable 141
 - form 102
 - inputHidden 114
 - inputSecret 111
 - inputText 23, 108
 - inputTextArea 115
 - message 23, 118
 - messages 23, 120
 - outputFormat 106
 - outputLabel 122
 - outputText 23, 106
 - selectBooleanCheckBox 130
 - selectManyCheckBox 132
 - selectManyListBox 139
 - selectOneListBox 136
 - selectOneMenu 70, 134
 - selectOneRadio 127
 - hx
 - commandExButton 24, 147
 - fileUpload 155
 - graphicImageEx 153
 - jspPanel 159
 - outputLinkEx 150
 - outputSeparator 159

- panelActionBar 164
- panelBox 162
- panelLayout 160
- playerFlash 178
- playerGenericPlayer 177
- playerMediaPlayer 184
- playerRealPlayer 182
- playerShockwave 180
- scriptCollector 23, 83
- itso
 - trafficLightGrid 483
- odc
 - clientBinder 449
 - dataGrid 450
 - graphDraw 453
 - tabbedPannel 167
 - tree 454
 - webService 456
- r
 - inputRichText 174
- wdo
 - commit 327
 - execute 326
 - find 327
 - remove 328
 - useMediator 326
- tag library 9, 37
 - component 490
- TemperatureConverter 507
- template 33
- test environment 53
- tier 231
- TLD 486
- tools vendor 228
- Traffic Light Grid 482
- TrafficLightGridRenderer 495
- TrafficLightGridTag 491
- transaction commit 337, 369
- transactionDetails.jsp 195, 220, 371
- transfer 193
- transferResult.jsp 195, 216, 368
- TransRecord 193
- TransRecordData 201
- tree 21
- Tree View 454
 - example 472
- tree view 49
- tree.jsp 472
- twin box 240

U

- UDDI registry 278
- UI
 - component model 488
 - components 4
- UICommand 124
- UIInput 115
- UIOutput 106
- UISelectMany 132, 139
- UISelectOne 127, 134, 136
- UITrafficLightGrid 488
- update model values 11
- user interface renderer 495
- user roles 6
- utility project 198

V

- validate method 78
- validation 11, 42, 67, 236
 - custom 222
 - portlet 385
 - test 71
- Validation tab 42
- validator 4, 9, 11, 20
 - example 78
- value
 - binding 19, 40, 91
 - expressions 92
- value change event 80
- ValueChangeEvent 19
- variable
 - scope 55
- Variables view 422
- view 4–5
 - Attributes 39, 93
 - Client Data 49, 444
 - DB Servers 312
 - Page Data 32, 43
 - Palette 36
 - Project Navigator 28
 - Quick Edit 50, 95
 - Server Configuration 53
 - Servers 73
 - Styles 65
 - Variables 422
- VisualBasic 410

W

WDO 301

Web

- application 8
- deployment descriptor 309
- perspective 34

Web Service 279, 456

- client 477
- example 475

Web service

- application 282
- invocation 288

Web services 277

- concepts 278
- JSF 281

Web style pager 187

web.xml 29

webService.jsp 476

webserviceInput.jsp 283, 285

webserviceOutput.jsp 283, 287

webservicesclient.xml 281

WebSphere

Application Server

- configuration 538
- deployment 538
- installation 529

Data Objects 301

 Portal Toolkit 383
 installation 531

Studio

- Application Developer 5.1.1 8
- Application Developer 5.1.2 8

 test environment 53
 setup 535

WebSphere Studio

 Application Developer
 installation 530

welcome page 58

Windows Media Player

 component
 Windows Media Player 184

withdraw 193

wizard

- Add Bean 46
- form to form 257
- new JSF page 30

workspace

projects 547

WSDL 278



Redbooks

WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects

(1.0" spine)
0.875" <-> 1.498"
460 <-> 788 pages



Redbooks

WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects

What are JavaServer Faces?

This IBM Redbook describes the new JavaServer Faces (JSF) technology for building Web applications and the WebSphere Studio Application Developer tools supporting it. JSF is based on the Java Specification Request (JSR) 127, of which the first release was finalized in March, 2004.

JavaServer Faces in practice

Service Data Objects in practice

The goal of JSF technology is to provide a complete framework to develop and deploy Web applications easily and rapidly. The framework provides solutions for a variety of tasks found in Web application development, such as user interface design, application navigation and flow, session and object management, validation and error feedback, internationalization, and container portability.

This redbook covers all of the core features of JSF Web development and the tools that are available in WebSphere Studio Application Developer 5.1.2 to design, develop, deploy, and test JSF Web applications.

Service Data Objects (SDO) is a data programming architecture and API for the Java platform that unifies data programming across data source types, provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks