

IBM Workplace Forms:

Guide to Building and Integrating a Sample Workplace Forms Application

Features and functionality

Designing forms

Integration topics



Bernd Beilke
Cayce Marston
Kioko Mwosa
George Poirier
Andreas A. Richter
Deanna Drschiwiski

Redbooks



International Technical Support Organization

**IBM Workplace Forms: Guide to Building and
Integrating a Sample Workplace Forms Application**

July 2006

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

First Edition (July 2006)

This edition applies to IBM Workplace Forms Release 2.5 and 2.6. While the code samples shown and used are from IBM Workplace Forms Release 2.5, the development concepts and approach also applies to IBM Workplace release 2.6.

© Copyright International Business Machines Corporation 2006. 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	vii
Trademarks	viii
Preface	ix
The team that wrote this redbook.	ix
Special acknowledgement for support with Portal Integration of the sample application.	x
Additional contributors to this redbook.	x
Become a published author	xi
Comments welcome.	xi
Chapter 1. Introduction to Workplace Forms	1
1.1 What is a form?	3
1.1.1 Form as a front-end to a business process.	3
1.1.2 Electronic Forms: XML Intelligent documents.	4
1.1.3 Forms marketplace	5
1.2 Overview of IBM Workplace Forms.	6
1.2.1 Value proposition	8
1.2.2 Product positioning	10
1.3 Innovation based on standards: XForms and XFDL	11
1.3.1 XFDL	12
1.3.2 XForms + XFDL in alignment with SOA	12
1.3.3 Sample solutions.	14
1.3.4 Banking and regulated industries	14
1.3.5 Proven eForm technology.	15
1.4 Summary	15
Chapter 2. Features and functionality	17
2.1 Form Document Model	18
2.2 Workplace Forms Component Technology.	19
2.3 Workplace Forms Designer.	19
2.4 Workplace Forms Viewer	20
2.5 Workplace Forms Server	21
2.5.1 Workplace Forms Server API	21
2.5.2 Workplace Forms Webform Server.	23
2.5.3 Workplace Forms Deployment Server	26
Chapter 3. Approaches to integrating Workplace Forms	29
3.1 Integration: what this means within the context of Workplace Forms	30
3.2 Workplace Forms document model and straight-through integration	31
3.2.1 The Workplace Forms document model.	31
3.2.2 Support for arbitrary XML instances	32
3.2.3 Straight-through integration	33
3.3 Aspects of integrating Workplace Forms	33
3.3.1 User Interface (UI) Integration.	33
3.3.2 Data integration.	41
3.3.3 Process integration	48
3.3.4 Security context integration.	49
3.3.5 Client-side device / hardware integration	49
3.4 Integration points summary	50

3.5 Partitioning of features and functionality	50
3.6 Introduction to actual integration scenarios.	51
Chapter 4. Building the base scenario: Stage 1	53
4.1 Introduction to the scenario used throughout this book	54
4.1.1 Starting with a paper-based form	54
4.1.2 From paper-based form to an electronic forms based application	56
4.1.3 Review of the specific forms: End user perspective	59
4.2 Overview of steps: Building Stage 1 of the base scenario	64
4.3 Preparing to build the form template	64
4.3.1 Wizard pages versus traditional form pages.	64
4.3.2 Considerations in advance: Best Practices for implementing traditional forms and wizard pages.	65
4.3.3 Possible starting points for creating a form.	67
4.4 Starting to build forms: Initial creation, design, and layout	70
4.4.1 Designing the layout of the traditional form.	70
4.4.2 Building the traditional form page	70
4.4.3 Creating a scanned template	71
4.4.4 Creating the layout for the wizard pages.	73
4.4.5 Steps to build the wizard pages	75
4.4.6 Setting up the toolbar	75
4.4.7 Adding layout items.	78
4.4.8 Reviewing the layout for the traditional form page	80
4.4.9 Reviewing the layout for the wizard page	83
4.5 Adding input items.	83
4.5.1 Adding input items to the traditional form	84
4.5.2 Adding input items to the wizard pages	86
4.6 Applying formatting and logic	87
4.6.1 Creating a field calculation	87
4.6.2 Creating a custom option	90
4.6.3 Adding a Submit button.	95
4.6.4 Adding Signature buttons	97
4.7 Adding the XML Data Model	102
4.7.1 Creating an XML Data Model	104
4.7.2 Creating XML bindings	105
4.7.3 XML schema validation.	106
4.8 Building the Servlet	108
4.8.1 Where we are in the process: Building Stage 1 of the base scenario.	108
4.8.2 Basic servlet methods.	108
4.8.3 Servlet code skeleton	110
4.8.4 Creating a template repository and form storage structure.	116
4.8.5 Servlet interaction for forms processing	117
4.8.6 Accessing a form through the Workplace Forms API	118
4.8.7 Extraction of form data	121
4.8.8 Signature validation	125
4.8.9 Form storage to local file system	126
4.8.10 Servlet doGet method for application navigation	127
4.9 Creating JSPs	130
4.9.1 Where we are in the process: Building Stage 1 of the base scenario.	130
4.9.2 Form template listing.	142
4.9.3 Approved form listing	143
Chapter 5. Building the base scenario: Stage 2	145

5.1 Overview of steps: Building Stage 2 of the base scenario	146
5.2 Data storage to DB2	146
5.2.1 Installing DB2 Server	147
5.2.2 Creating tables	148
5.2.3 Populating tables	153
5.2.4 Installing DB2 clients on development clients and servers	157
5.2.5 Developing the data access layer (DB2)	157
5.3 Web services	168
5.3.1 Where we are in the process of Building Stage 2 of the base scenario	168
5.3.2 Web services integration	169
5.4 Servlet access to Form data	179
5.4.1 Where we are in the process: Building Stage 2 of the base scenario	180
5.4.2 Servlet Access to form data (prepopulation / data retrieval)	180
5.4.3 Form prepopulation	181
5.4.4 Extraction of form data and storage of entire form	185
5.4.5 Reading form data from DB2	188
5.5 Adjustments to JSPs for Stage 2	190
5.5.1 Where we are in the process: Building Stage 2 of the base scenario	190
5.5.2 Modifying the index.jsp	190
5.5.3 Creating a JSP to view DB2 data	192
5.6 Form prepopulation using Web services	196
5.6.1 Where we are in the process: Building Stage 2 of the base scenario	196
5.6.2 Importing the WSDL file	197
5.6.3 Calling the Web services	199
5.7 Workflow	204
5.7.1 Where we are in the process: Building Stage 2 of the base scenario	204
5.7.2 Approval workflow	205
Chapter 6. Integrating with Portal	209
6.1 Goal of integrating the application with WebSphere Portal	210
6.2 Overview of Portal integration	211
6.3 Writing a portlet	214
6.4 Parking the Workplace Forms Viewer in the Portal	234
6.5 Deploying the portlet	235
Chapter 7. Zero Footprint with WebForm Server	241
7.1 Zero Footprint solution	242
7.2 Form design delta	243
7.3 Web services moves the solution to Viewer only	247
Chapter 8. Integration with IBM DB2 Content Manager	249
8.1 Overview	251
8.2 Basic design of Content Manager integration	253
8.3 Integrating the sales quote sample with DB2 Content Manager	255
8.3.1 Create attributes and item types	255
8.3.2 Add the CM integration in the form	260
8.3.3 Servlet to servlet communication	266
8.3.4 Test the form integration with CM	268
Chapter 9. Domino integration	273
9.1 Introduction to integration of Domino and Workplace Forms	274
9.1.1 How can the two technologies complement each other?	274
9.2 Overview and objective of this integration scenario	275
9.3 Environment overview	276

9.4 Setting up the Domino environment	278
9.5 Domino development	279
9.5.1 Repository database	279
9.5.2 Template Database: components to create a new form from template	286
9.5.3 Template Database: Receiving submitted forms in Domino	313
Appendix A. Additional material	333
Locating the Web material	333
Using the Web material	333
Related publications	337
Online resources	337
How to get IBM Redbooks	337
Help from IBM	337
Index	339

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:

DB2®	IBM®	Redbooks (logo)  ™
DB2 Universal Database™	Lotus®	WebSphere®
Domino®	Lotus Workflow™	Workplace™
Domino Designer®	MVS™	Workplace Forms™
eServer™	Rational®	Workplace Managed Client™
Footprint®	Redbooks™	

The following terms are trademarks of other companies:

Java, JavaScript, JDBC, JSP, J2EE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

Texcel and FormBridge are registered trademarks of Texcel Systems, Inc. in the United States and other countries.

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

Preface

This IBM® Redbook describes the features and functionality of Workplace™ Forms and each of its component products. After introducing the products and providing an overview of features and functionality, we discuss the underlying product architecture and address the concept of integration.

To help potential users, architects, and developers better understand how to develop and implement a forms application, we introduce a specific scenario based on a “Sales Quotation Approval” application. Using this base scenario as a foundation, we describe in detail how to build an application that captures data in a form, then applies specific business logic and workflow to gain approval for a specific product sales quotation.

Throughout the scenario, we build upon the complexity of the application and introduce increasing integration points with other data systems. Ultimately, we demonstrate how an IBM Workplace Forms™ application can integrate with WebSphere® Portal, IBM DB2® Content Manager, and Lotus® Domino®.

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, Cambridge, Massachusetts center.

Bernd Beilke is an IT Specialist for Workplace Forms on the Pan IOT TechWorks Team based in Berlin, Germany. He joined IBM in 1997 and worked as a consultant in various Lotus software projects focusing on large scale deployments and server consolidation in Domino environments. In recent years he has been working on a competitive Presales team to support rather challenging sales situations in the Lotus brand. His areas of expertise include Workplace, Portal, and Collaboration Software.

Cayce Marston is a Senior I/T Specialist within the Lotus Worldwide Technical Sales team. He joined IBM as part of the PureEdge Solutions acquisition in mid-2005, where he held the position of Solutions Engineering Manager. Prior to joining PureEdge, Cayce has worked as a consultant and software architect in the telecom and financial services industries. His areas of expertise include Workplace Forms, XForms, systems integration, and architecture. In late 2005, Cayce authored the RedPaper *Extending SOA with XForms*.

Kioko Mwosa is an IBM Certified Consulting IT Specialist with the Lotus World Wide Technical Sales organization. He has over 8 years of experience at IBM and his areas of expertise include document and Web content management, forms and portals. His current role includes supporting the IBM Sales Team on pre-sales customer calls, assisting customers with system architecture, and providing technical assistance for deployment. He has authored several articles for the Lotus Developer Domain and written a White Paper on Lotus Domino Document Manager.

George Poirier is a member of the World Wide Technical Sales team for Workplace Forms and Workplace Learning. He has been with IBM for 30 years. As an employee of IBM, his roles have included: 7 years in Worldwide Technical Sales, Systems Architect for the Lotus Professional Services (ISSSL), and MVS™ Technical Support specialist in IBM's Dallas System Center. George has developed and delivered several Learning deepdive enablement sessions.

Andreas A. Richter is a system architect working in IBM Software Service for Lotus (ISSL) since 1999, specifically for the IOT Europe North East division. He primarily leads Domino based application development projects, Lotus Workflow projects, and integration projects with SAP HR data in large accounts. Beginning in 2004, he started to focus on J2EE™ application development (IBM Workplace, Workplace Managed Client™, Workplace designer, and Bowstreet Portlet Factory). Since November 2005, he has been concentrating on IBM Workplace Forms while continuing to also engage in Domino projects.

Deanna Drschiwiski is an Information Developer for Workplace Forms in Victoria, BC, Canada. She has been working with Workplace Forms products since 2000 and joined IBM as part of the PureEdge Solutions acquisition in mid-2005. Deanna has written extensively on the Extensible Forms Description Language (XFDL), XForms, and the Workplace Forms product suite.

John Bergland is a Project Leader at the IBM International Technical Support Organization, Cambridge Center. He manages projects that produce IBM Redbooks about Lotus software products. Before joining the ITSO in 2003, John worked as an Advisory IT Specialist with IBM Software Services for Lotus, specializing in Notes and Domino messaging and collaborative solutions.

Special acknowledgement for support with Portal Integration of the sample application

Yves Bollanga is an IBM Certified Consulting IT Specialist with the Lotus World Wide Technical Sales organization supporting Both Lotus Workflow™, WebSphere Portal Server, Workplace Forms, and Workplace collaboration services in pre-sales and post-sales engagements. Yves' current role includes supporting the IBM Sales Team on pre-sales customer calls, assisting customers with system architecture, and providing technical assistance for deployment. Prior to this position, Yves was a Senior Consultant for Lotus Professional Services in France where he was involved in major e-business projects. He has over 7 years of experience at IBM. In his spare time, Yves contributes to the expansion of African Afro-centric culture via both satellite and cable television in Europe, Middle-East, North America, and Africa.

Additional contributors to this redbook

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

Paul A. Chan, Worldwide Director, Forms Marketing, IBM Software Group, WPLC, IBM, Victoria, BC Canada

Bob Levy, Workplace Product Management, IBM Software Group, WPLC, IBM, Cambridge, MA

Steve Shewchuk, Workplace Forms Enablement, IBM Software Group, WPLC, IBM, Victoria, BC Canada

Steve Myerow, President, Texcel Systems, Inc. (<http://www.texcel.com/IBM/>)

Jane L. Wilson, Knowledge Architect, IBM Software Group, WPLC, IBM, Westford, MA

Yvonne Lyon, Editor, IBM International Technical Support Organization, San Jose Center, CA

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 e-mail to:

redbook@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400



Introduction to Workplace Forms

IBM Workplace Forms *unlocks* the enterprise value of information currently trapped within paper forms by dramatically improving the access to accurate and timely information by people and systems.

IBM Workplace Forms enable organizations to streamline core processes and compliance-oriented operations, resulting in reduced costs and improved service levels to customers, suppliers, partners, and employees.

By incorporating IBM Workplace Forms into your organization, and converting from paper-based forms to electronic forms, IBM Workplace Forms provide a security-rich, dynamic, and intelligent front-end to your organization's business processes.

The IBM Workplace Forms product family consists of a server, designer, and client viewer that together enable the creation, deployment, and streamlining of XML forms-based processes. By leveraging open standards to integrate an intelligent user interface with high value back-end systems, IBM Workplace Forms provide public and private sector organizations across many industries with security-rich forms that leverage existing resources and systems to help better serve customers and increase operational efficiency.

This IBM Redbook describes the features and functionality of Workplace Forms and each of its component products. After introducing the products and providing an overview of features and functionality, we discuss the underlying product architecture and address the concept of integration. To help potential users, architects, and developers better understand how to develop and implement a forms application, we introduce a specific scenario based on a "Sales Quotation Approval" application.

Using this base scenario as a foundation, we describe in detail how to build an application that captures data in a form, then applies specific business logic and workflow to gain approval for a specific product sales quotation. Throughout the scenario, we build upon the complexity of the application and introduce increasing integration points with other data systems. Ultimately, we demonstrate how an IBM Workplace Forms application can integrate with WebSphere Portal, IBM DB2 Content Manager, and Lotus Domino.

Note: While the examples in this redbook use IBM products, the integration examples and approach contained in this redbook can be applied to a wide variety of content, document, portal, and workflow systems and architectures due to the open standards architecture of Workplace Forms.

Which specific version of IBM Workplace Forms does this redbook apply to?

This redbook has been written with the intent of showing the value of IBM Workplace Forms, for both IBM Workplace Forms Release 2.5 and where applicable, IBM Workplace Forms Release 2.6:

- ▶ The concepts and value proposition in this introduction chapter apply to both Release 2.5 and 2.6. With Release of 2.6, Workplace Forms provides full support for the Xforms standard.
- ▶ All specific features and functions reviewed in Chapter 2, “Features and functionality” on page 17 refer specifically to Release 2.5
- ▶ All specific examples shown and used when building the sample scenario application are based on the codebase for IBM Workplace Forms Release 2.5.

1.1 What is a form?

As a foundation to discuss the benefits and specific features and technology of IBM Workplace Forms, it is important to first establish a common understanding of the term *form*.

For the context of this product, the following definition of a *form* is a good starting point:

- ▶ **Form:** A blank document or *template* to be filled in by the user. The form serves a structured mechanism to capture data.

In the most traditional sense, forms have been largely paper-based. For the context of this book, we will focus on benefits of electronic forms and the ability to process and reuse the data captured in forms. We will demonstrate how an XML-enabled eForm is more than just a form. It is a Dynamic User Interface that captures structured data at the front-end of the business process and enables a rich user experience.

1.1.1 Form as a front-end to a business process

Given the information-intensive nature of business, both documents and forms are essential components in driving and supporting all types of processes. These include processes and procedures, regulatory requirements and compliance issues, and inter-agency/inter-departmental and constituent/customer communications. Document types range from spreadsheets to engineering drawings to Web information. They can come from desktop users or output from back-end systems such as ERP, CRM, content management, and legacy systems.

Why are documents so critical?

- ▶ They are a necessary and ubiquitous part of business. In many cases, they are the “end product” of a service (a business permit, or an insurance policy, for example).
- ▶ Documents are familiar, readily available in paper or online forms, and deliver high levels of visual quality.
- ▶ Documents meet regulatory or compliance requirements that often dictate a document’s precise look and feel (insurance policies, tax forms, permits, etc.).
- ▶ Documents are universally accepted: they can be used offline.

Forms are also a critical part of business processes and procedural transactions.

Filling out forms — from business permits to filing taxes — is familiar to anyone who has dealt with government agencies. Forms are used to capture information from individuals and organizations. Much of this information ends up in core IT systems.

While the form is a structured mechanism to capture data, much of the business value lies on how the data is processed once it has been entered. The form acts as a UI for entering the data, and is in turn a starting point for a business process.

For example, one of your customers might fill out a paper form to open a new account. Once that form is completed and passed to the correct department, the process of creating that account begins. This concept is fundamental to understanding and appreciating the business value of electronic forms. A form is a front-end to a process. Ultimately, IBM Workplace Forms provides a solution which goes beyond merely converting paper-based forms to electronic forms. The real value is the ability to *integrate* people, information, and processes.

Key concept: A form is a mechanism for capturing data and represents a front-end to a *business process*.

1.1.2 Electronic Forms: XML Intelligent documents

When considering the benefits that result from evolving toward electronic forms and XML intelligent documents, there are several levels of integration to be considered. The extent to which electronic forms have been fully integrated into your organization's business processes impacts the level of value and ROI from your investment in electronic form processing.

As shown in Figure 1-1, you can look at eForms from a Content-centric view or a Process-centric view. The entry point or Level 1 is the Print and Fill form. For example, you might download a form from the Web, print it, fill it out, and then most likely mail it in for processing. There is *some* value with this (namely, the company provided the user or customer with the form from a Web site versus having to mail them a copy), but this is just the prelude to what is possible. At this level, the back-end processing of the form has not changed — only the delivery method has been improved.

But as you go up the value curve, you will start to realize a larger return on your investment, and it can grow exponentially when you start to cross into the process spectrum. For example, Level 2 represents an environment in which you allow the user or customer to electronically fill out a form on the Web, optionally sign it, and submit it for back-end processing. Tremendous investments are being made to put paper processing online, and using automation to do more of the work. A Fill and Submit type of an eForm (Level 3) is an excellent example of this.

The other levels of eForms processing (Level 4 and Level 5) are extensions to the Fill and Submit example, where you are truly adding an electronic workflow to the form once it has been submitted. At Level 5 you make the Content (the form) part of an ECM or Enterprise Content Management environment. In this case, you allow for re-use of the form or Content in other applications or part of a larger overall storage strategy. An example of this might be a Customer Folder concept — where the eForm is one piece of content in the overall electronic customer folder.

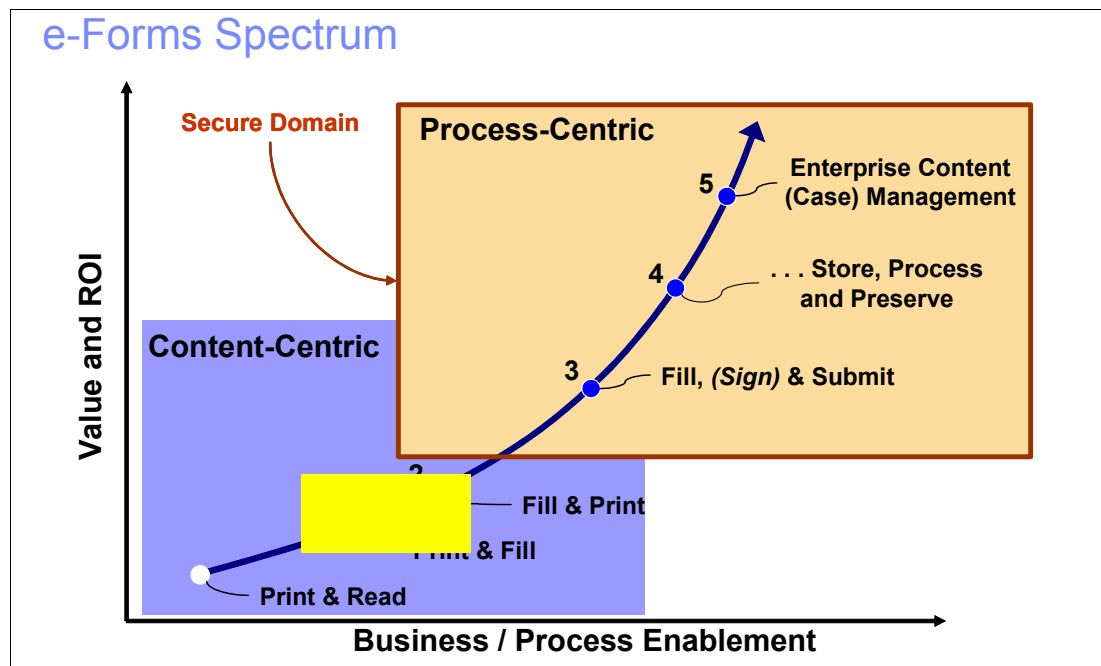


Figure 1-1 Spectrum of eForm integration

Finally, the value from implementing electronic forms can be viewed in the following terms:

- ▶ **Meeting customer and partner demands:**
 - Faster, easier access to online forms and services
 - Consistent experience across programs and delivery channels
- ▶ **Increasing operational efficiencies:**
 - Automate paper processes
 - Integrate services delivery
 - Hard dollar savings with accelerated ROI
- ▶ **Ensuring security:**
 - Protect customer / partner privacy
 - Control access
 - Ensure content integrity
 - Authenticate people and processes

The degree to which your organization can leverage these benefits depends upon where you are in the spectrum of integrating eForms.

1.1.3 Forms marketplace

According to industry analysts, eForms represent a \$500M Market, with the following notable points:

- ▶ XML-enabled eForms will double in use as a standard enterprise document format.
- ▶ By 2009, 25% of enterprises will use XML-based document processes.
- ▶ An XML-enabled eForm is more than just a form. It is a dynamic user interface that captures structured data the front-end of the business process – and ultimately enables a rich user experience:
 - It enforces business rules and validates data at the glass.
 - It is a computation engine without server refresh.
 - It provides a secure transaction captured in a standard data model with digital signatures.
 - It functions offline/online – thin or thick client.
 - It is a storable and retrievable document object that traverses a business process via workflow engine, e-mail, or Web services.

Figure 1-2 illustrates the positioning of IBM Workplace Forms relative to other industry players. In terms of both Enterprise Value and Functionality, Workplace Forms ranks the highest due to its advanced functionality, ability to integrate with and drive business processes, and finally, its ability to meet regulatory compliance requirements. Alternatively, competitors such as Microsoft® and Adobe provide *Proprietary Forms Solutions* at the bottom of the middle tier.

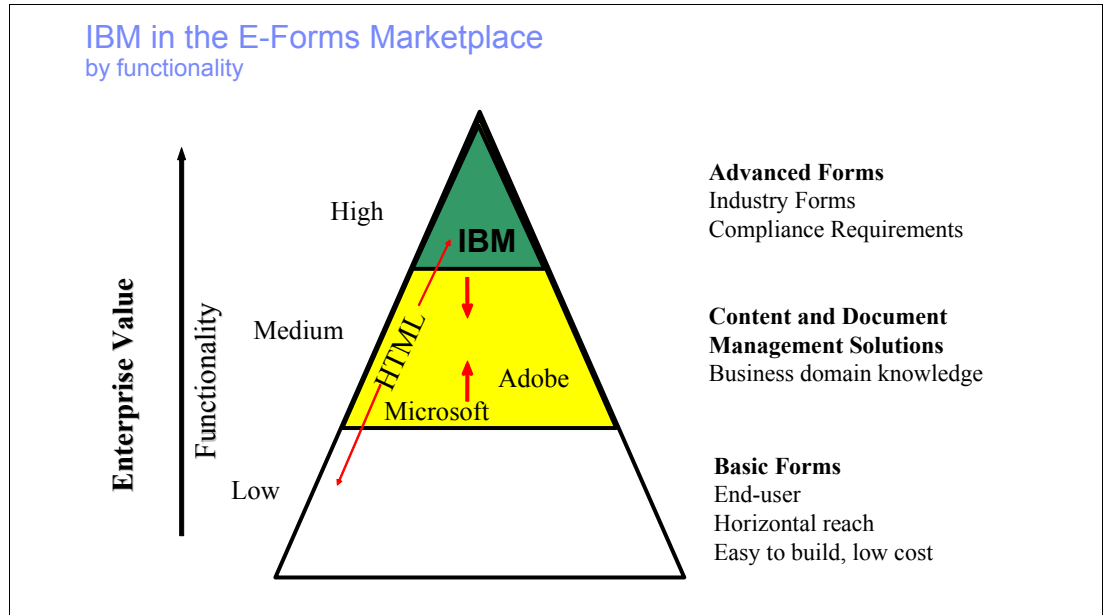


Figure 1-2 Positioning of IBM Workplace Forms

1.2 Overview of IBM Workplace Forms

The IBM Workplace Forms suite includes the following products:

- ▶ IBM Workplace Forms Viewer
- ▶ IBM Workplace Forms Designer
- ▶ IBM Workplace Forms Server

Together, these tools allow you to create, fill, and submit forms, and integrate those forms with your back-end processes. Ultimately, IBM Workplace Forms tools work together to make each of the functions possible (creating, submission, and integration with other data systems) and can be used to create an end-to-end solution.

Figure 1-3 illustrates a conceptual overview of how the products work together within the context of a complete Web based forms application.

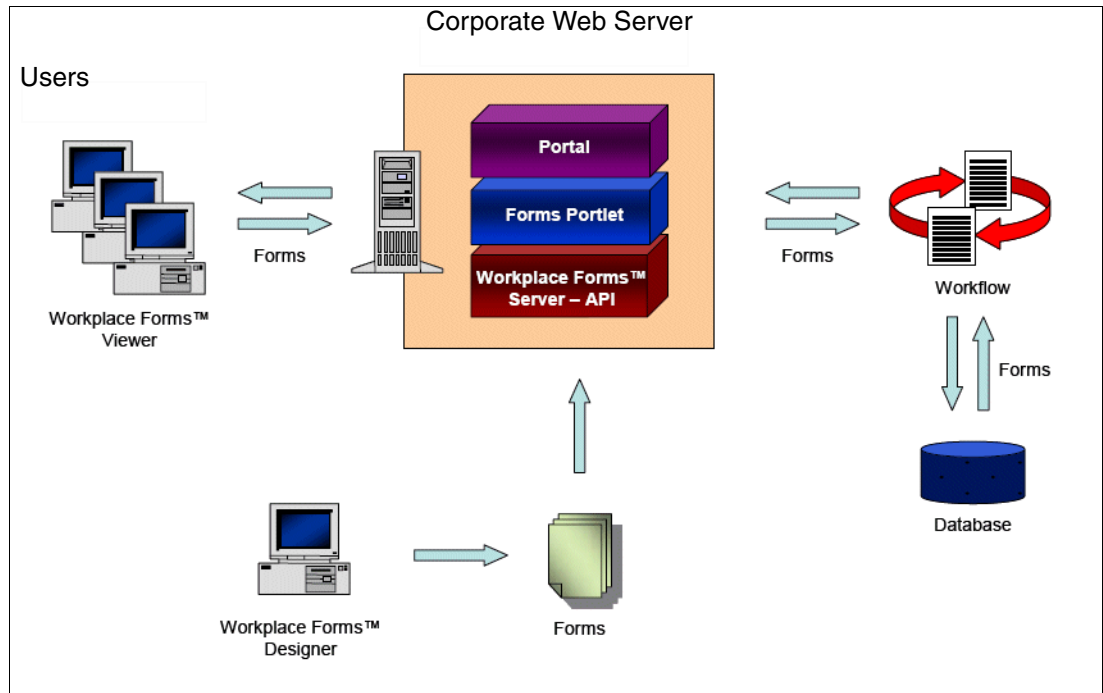


Figure 1-3 Conceptual overview of Workplace Forms

Figure 1-4 illustrates a more in-depth view of how IBM Workplace Forms works with other products in the IBM Software portfolio.

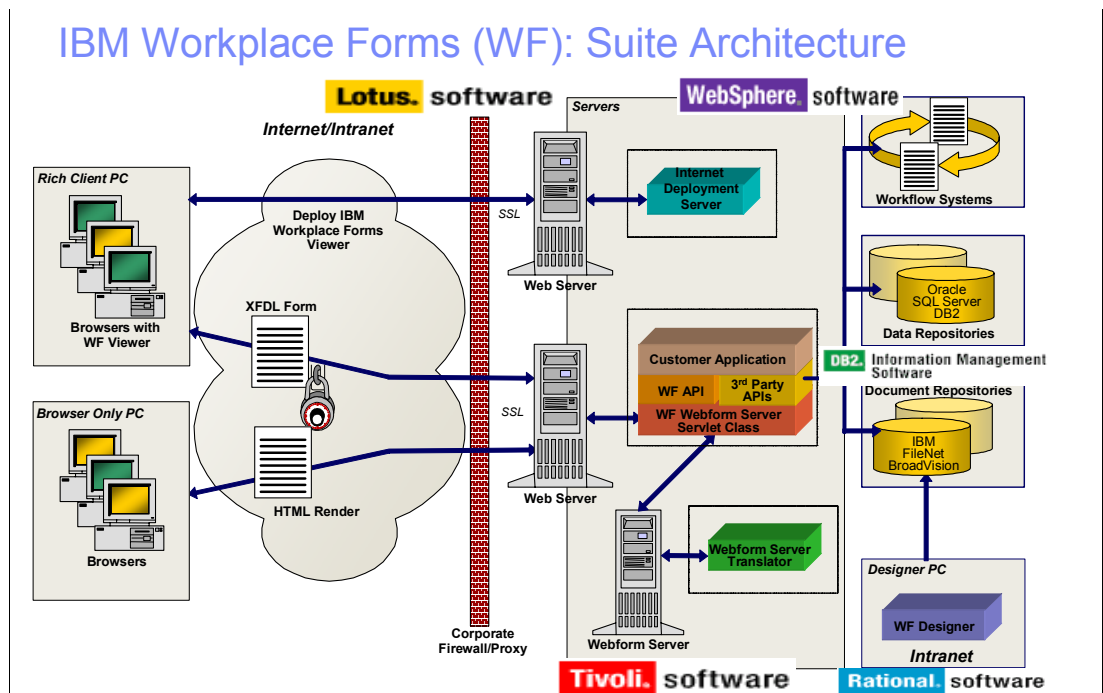


Figure 1-4 Architecture of Workplace Forms

IBM Workplace Forms Viewer is a feature-rich desktop application used to view, fill, sign, submit, and route eForms, and is able to function on the desktop or within a browser. It enables full connectivity with real-time integration using Web services. The open standards framework of IBM Workplace Forms enables Workplace Forms Viewer to operate in portal or stand-alone environments, in online or offline modes, and as a plug-in for thin or rich-client browsers.

IBM Workplace Forms Designer is an easy-to-use WYSIWYG eForm design environment that supports the drag-and-drop creation of precision forms with an open and accessible, XML-schema-derived data model. IBM Workplace Forms Designer leverages open standards to deliver advanced forms-based business process automation solutions that integrate seamlessly across lines of business applications and IT infrastructure.

IBM Workplace Forms Server enables the creation and delivery of XML forms applications. It provides a common, open interface to enable integration of eForms data with server-side applications using industry-standard XML schemas. IBM Workplace Forms Server also delivers a true zero-footprint solution, providing eForms to external users quickly and efficiently within a browser without requiring additional downloads or plug-ins. IBM Workplace Forms Server provides a low administration solution for data-capture requirements while supporting precise viewing and printing, automated validation, and the security and compliance capabilities of XML eForms.

1.2.1 Value proposition

In this section we highlight key value points where IBM Workplace Forms can benefit your organization.

IBM Workplace Forms allows your organization to more efficiently and effectively integrate people, information, and processes.

Starting with the traditional paper-based forms:

- ▶ Workplace Forms *unlocks the value* trapped in paper forms.
- ▶ Workplace Forms then make this information *more accessible*.
- ▶ Workplace Forms helps *manage* the information and associated processes *more efficiently*.

IBM Workplace Forms enables you to *extract value* from paper-based data. This ensures:

- ▶ **Accurate information:** Data you can rely on.
 - Data and schema validation provided at the client and server levels using business logic that understands business process, data types, and schema requirements.
 - **Business impact:** Ensures accurate data required by all systems that touch the data or are involved in the business process. No expensive downstream re-work. Ensures that business can process data efficiently.
- ▶ **Timeliness of data:** Enabling the right data at the right time.
 - On demand data capture from employees, customers, suppliers, and partners. Captures information as required by real-time processes, and uses business rules and Web services to get the *right data at the right time*.
 - **Business impact:** The right data at the right time — This enables straight-through processing by capturing all necessary data as required to optimize a business process.

- ▶ **Enhanced user productivity:** Role-based personalization.
 - Business logic is used to create personalized role-based “wizards” that can work online or offline to dramatically decrease the time required for each participant in a forms-based process.
 - **Business impact:** Users have a personalized form-filling experience to minimize data capture tasks and maximize productivity.
- ▶ **Leading forms technology:** Next generation online forms.
 - Best in class technology standards support include Web 2.0 (Ajax) for powerful browser based forms experience: XForms — the only W3C approved standard for the next generation of Internet forms. The product is based upon DSig standards that IBM helped shape. XML, XML Schema, and Web services are all supported since Workplace Forms was designed from the ground up to support online forms.
 - **Business impact:** Best Forms Technology on the Market built on Common Open Standards.
- ▶ **Enterprise accessibility of data:** Leveraging information across the enterprise.
 - Interoperable forms based on Open Standards (XML, XML Schema, XForms). Forms operate across various workflow, repository, portal, collaboration and Document management systems. Thin server architecture that supports J2EE and .NET implementations ensures high performance and throughput that can be easily scaled and grow with the customer's needs. The largest forms customers in the world use Workplace Forms.
 - **Business impact:** Forms that work across system, division and corporate boundaries provide business flexibility to enable and extend forms-based processes to create an On Demand business.
- ▶ **Flexible and rapid deployment:** Rich or thin client options.
 - Rich client, thin (browser) or hybrid solutions provide flexibility of deployment for customers that require offline, online and varying process or deployment requirements. “Design Once Render Many” paradigm. Fast deployment using robust migration tools (from static formats such as PDF) and an Eclipse design tool that can create reusable form components.
 - **Business impact:** Flexibility to deploy for any process that spans organizations and diverse customer process requirements. Rapid time-to-value. Proven references.
- ▶ **Manage information and processes more efficiently:** Lower operating costs.
 - Workplace Forms supports role-based workflows with flexible support for document and data driven workflows and support for complex process flows that can involve multiple and overlapping authorizations and signatures.
 - **Business impact:** Workplace Forms enables straight through processing and optimization of forms-based processes to reduce costs and decrease process cycle times.
- ▶ **Best of breed compliance document:** A robust transactional record.
 - A Workplace Form acts as document throughout the lifecycle of a process — from initiation to archiving a record of the transaction. Based on a declarative business rules that avoid document integrity issues associated with scripting languages where digital signatures are used.
 - **Business impact:** No separate paper or imaged copies required to document a transaction or compliance of a process.

- ▶ **Transactional document for SOA:** The business process document for SOA.
 - Workplace Forms uses a “thin architecture” that ensures high performance while adding an intelligent document that adds business process rules on top of SOA. Leverages Web service support in the WorkPlace Forms client and adds the process rules that add unique knowledge to XML Schemas to drive process automation.
 - **Business impact:** A process-aware document that instantiates and drives process improvement on top of SOA.

1.2.2 Product positioning

IBM Workplace Forms builds upon the value of the IBM Software portfolio. Advanced eForms are a critical component of Industry Solutions due to their broad applicability to a variety of business processes. IBM Workplace Forms serves as a *common front end* to many different products within the IBM Software Portfolio (Figure 1-5).

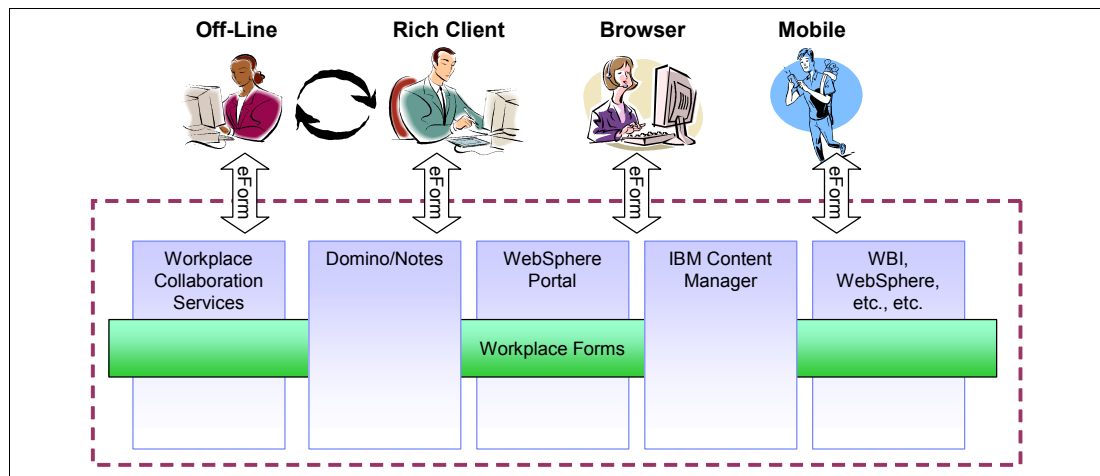


Figure 1-5 How Workplace Forms builds upon existing IBM Software products

- ▶ Workplace Forms provides a market leading business process automation framework composed of products, partnerships, and services to create manage and deploy XML forms-based processes.
 - This addresses customers’ demand for increased efficiencies in business process management.
- ▶ Workplace Forms is helping to further drive Open Standards, especially XForms, a key open industry standard that IBM is supporting. Workplace Forms also supports Java™, XML, and Eclipse technology.
 - This accelerates adoption of open industry standards (like XML and XForms) to ensure that business information is not locked in a proprietary format.
- ▶ Delivering high value industry solutions:
 - This fulfills customer demand for industry specific applications to replace manual forms.
- ▶ Workplace Forms leverages complementary technology.
 - Workplace Forms is a logical extension to the extensive forms-based application development business in Lotus Notes. Additionally it complements Lotus Domino and IBM Workplace solutions for risk and compliance management.

1.3 Innovation based on standards: XForms and XFDL

Workplace Forms is built upon proven, standards based technology, including XForms and Extensible Forms Description Language (XFDL). As Web based electronic forms have become more ubiquitous, XForms has emerged as the W3C specification for Web forms to overcome limitations with HTML Forms.

The design goals of XForms meet the shortcomings of HTML forms point for point:

- ▶ Excellent XML integration (including XML Schema)
- ▶ Provide commonly-requested features in a declarative way, including calculation and validation
- ▶ Device independent, yet still useful on desktop browsers
- ▶ Strong separation of purpose from presentation
- ▶ Universal accessibility

Xforms enables eForms that can be used with a wide variety of platforms, including desktop computers, handhelds, information appliances, and even paper. It aims to combine XML and forms.

Important: W3C XForms is an open standard for forms on the Web that builds on the syntax of XML data schemas. It provides the necessary rules and user interface abstraction to enable a forms data processing model that ensures interoperability for customers and partners, and that speeds time-to-market and reduces costs.

XForms: Business benefits and customer value

The W3C XForms standard provides the following major benefits to customers:

- ▶ Standardization at a technology and industry transaction level enable interoperable B2B processes.
- ▶ Standardization of a forms data processing model enables reusable components that integrate with SOA, enabling faster time-to-market with lower form application deployment and maintenance costs.

This standardization is accomplished in the following ways:

- ▶ XForms supports existing industry schemas.
- ▶ XForms can extend industry data schemas to support forms processing rule.
- ▶ XForms provides standards organizations with a more complete form definition to enable industry participants a faster time-to-market and a greater level of interoperability.

Ultimately, this results in the following business benefits:

- ▶ **Enables application interoperability:** XForms are available on any device...in any language...for any able/ impaired person...in any role within a business process.
- ▶ **Enables industry transactional standards:** XForms supports industry schemas along with transactional rules/UI.
- ▶ **Lowers application development costs:** XForms enables reusable form components with multiple client deployment options.
- ▶ **Enhances and complements SOA:** XForms provides a forms data processing model and supports active content using declarative rules and Web services.

1.3.1 XFDL

Extensible Forms Description Language (XFDL), developed by UWI.Com and Tim Bray, is an application of XML that allows organizations to move their paper-based forms systems to the Internet while maintaining the necessary attributes of paper-based transaction records. XFDL was designed for implementation in business-to-business electronic commerce and intra-organizational information transactions.¹

XFDL is a highly-structured XML protocol designed specifically to solve the body of problems associated with digitally representing paper forms on the Internet. The features of the language include support for a high-precision interface, fine-grained computations and integrated input validation, multiple overlapping digital signatures, and legally-binding transaction records.

What does XFDL add to XForms?

The benefits include:

- ▶ Document-centricity
- ▶ XFDL stores the data in the document, creating a single record
- ▶ Precision layout and printing
- ▶ Can faithfully reproduce paper forms
- ▶ Wizard-based, dynamic forms
- ▶ Can guide the user through filling process, change on the fly, and reduce errors
- ▶ Broad support for signatures
- ▶ Locks both the XFDL presentation and the XForms data
- ▶ Extension points for integration with other technologies
- ▶ Can embed .jar files in the form to extend the functionality

What does XForms add to XFDL?

The benefits include:

- ▶ New items
- ▶ Table, pane, checkgroup/radiogroup, slider
- ▶ XForms event handlers
- ▶ Value-changed, read-only, read/write, submit-error, etc.
- ▶ XForms functions
- ▶ Boolean-from-string, avg, min, max
- ▶ Device Independence

Common XML Data Model

Workplace Forms gives a common XML Data Model (based on the W3C XForms standard) that can work in heterogeneous IT environments. These are common in most organizations that can combine diverse J2EE, .NET, legacy, CRM, ERP, HRMS, Content, Document, and Workflow environments. Also, this common XForms model within Workplace Forms provides the ultimate flexibility in providing personalized, role-based “views” of this data for improved user productivity (wizard is an example of this). This common XForms model also allows for multiple system “views” or schemas necessary for straight through processing of this same data as required for back-end integration.

1.3.2 XForms + XFDL in alignment with SOA

In addition to standardizing an eForm document model, XForms technology has excellent alignment with both the principles and technical requisites of service-oriented architectures

¹ XFDL: Creating Electronic Commerce Transaction Records Using XML: Barclay T. Blair and John Boyer
<http://www8.org/w8-papers/4d-electronic/xfdl/xfdl.html>

(SOA). XFDL + XForms provides Workplace forms with an enabler for Service-Oriented eForm solutions; while strong technical alignment also reduces barriers.

At a functional level, one often sees a similar set of activities occur throughout different forms applications and forms-based processes. Let us consider some of the most common interactions that take place in Web and Portal eForm applications. Standard interactions in eForm applications include:

- ▶ Server-side form prepopulation, that is, the merging of an empty form template with data
- ▶ Submission of a form into a Content Management system at various stages of a process or workflow
- ▶ Submission of a completed, signed form to a Record Management System as a transaction record at the conclusion of a process
- ▶ Presentation of a form to users on laptops (mobile computers), tablet, or handheld devices both in online and offline modes
- ▶ Storage of form data into a database (often for reporting or for use by other systems)
- ▶ Transmission of form data into one or more Line-of- Business (LOB) systems.
- ▶ Validation of digital signatures as part of an approval process.

At a solution architecture level, SOA gives us a great story. Figure 1-6 provides a representative example of how eForm application functionality can be effectively used within the business tier of a larger Web application.

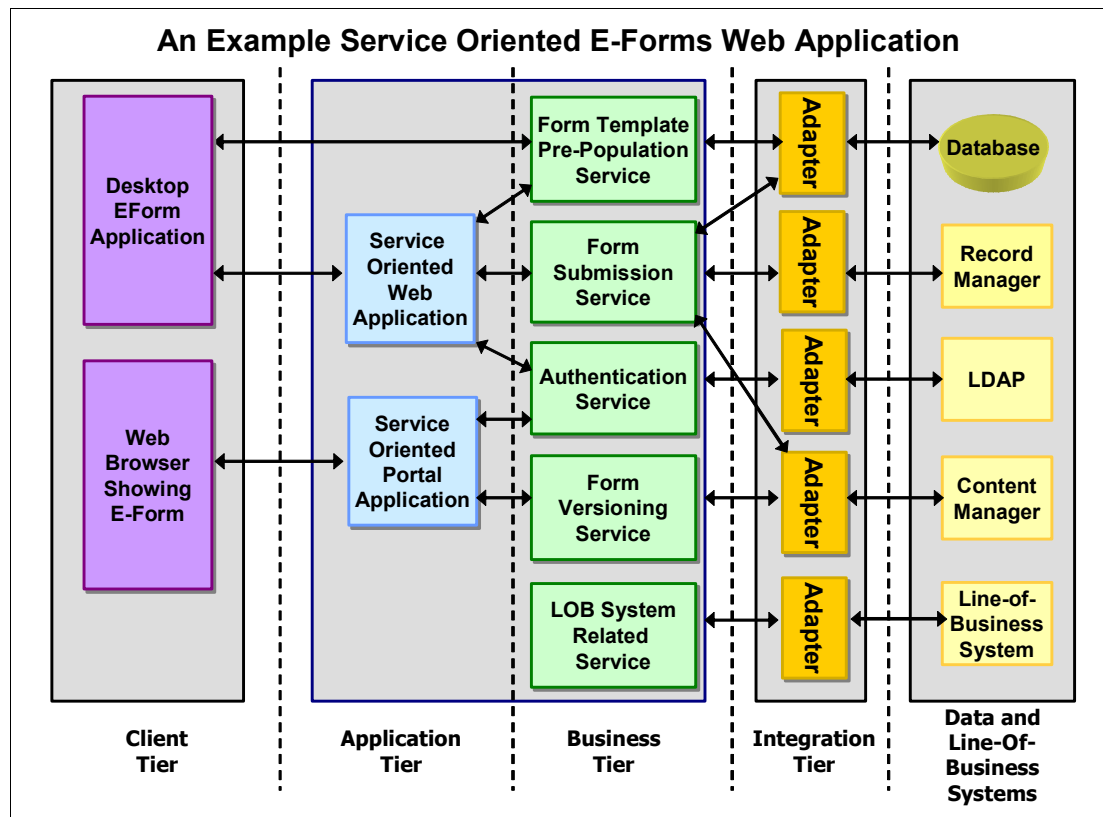


Figure 1-6 Example Service Oriented eForms Web Application

As you can see, a number of the services within the business tier are encapsulations of specific form application-related functionality, *designed for reuse across multiple applications*.

1.3.3 Sample solutions

The flexibility of the IBM Workplace Forms tools allow you to create solutions for any form-based process. The example scenario used for building the sample application in this book was intentionally kept generic — illustrating aspects of a Workplace Forms application that will apply to businesses across many industries, as well as illustrating concepts and benefits that apply to both small or larger organizations.

The following summaries discuss some possible industry specific solutions that you can create using these tools.

Government program registration

Government agencies often need to enroll the general public in a variety of programs. In these cases, the government is trying to serve a large and diverse population, with various levels of computer knowledge and Internet connectivity. The Webform Server component of the Workplace Forms suite offers the perfect solution for these situations. Users can log onto a central Web site and complete a registration form using only their Web browser. This saves them from having to download and install Workplace Forms Viewer, which can be intimidating for some users and time consuming over low bandwidths.

Additionally, while it is not possible to issue a digital certificate to each citizen, you can use Clickwrap signatures to capture some information about the user and confirm their approval. Clickwrap signatures simulate the “click to accept” process that is common on many Web sites today, and add a measure of security to the form. They can also include information about the user, such as a pass phrase, that you can use later to identify them.

1.3.4 Banking and regulated industries

Banking and other regulated industries must comply with a variety of government regulations, and must be able to produce reliable audit trails to demonstrate their compliance. In these cases, records must be maintained for many years, and those records must be secure.

Workplace Forms support a wide range of digital signature technologies ranging from simple authenticated password acceptance to biometric (retinal scans, fingerprint readers) and PKI certificates. Once a form is signed, the signature makes the form tamper evident. This means that if any of the information in the form is changed, the signature itself breaks, indicating that it can no longer be trusted. Furthermore, Workplace Forms provides the ability to sign a form template to know whether anyone has tampered with the template itself. Finally, most digital signature technologies reliably identify the signer. These features combine to create reliable records: you can reliably identify who signed each form, and you can easily judge whether the form has been changed in any way.

Additionally, the forms themselves are written to comply with open standards promoting interoperability and reducing the total cost of ownership through support of standards such as XML, XForms, JSR-168/170, etc. Workplace Forms also uses a native XML document type that future proofs archival records of forms so organizations can feel confident that forms records and the data contained within can be recovered within the necessary retention and record keeping requirements.

Secured communications

Some organizations may require more than just signatures to secure their forms. This is sometimes the case due to privacy laws or other legal requirements that insist that the forms themselves cannot be viewed by other people. In these cases, signature technology falls short, because while it will reveal tampering with a form, it does not prevent simple viewing.

Using the API, you can create an extension for Workplace Forms Viewer that will encrypt each form before it is sent. The server that processes the forms can then use the API to decrypt the form once it is safely behind a firewall.

The form is encrypted before it leaves the user's computer. If it is intercepted during transmission, or copied from a public server, it will be completely unreadable. However, once it is safely behind a firewall, it can be decrypted and processed with ease.

This encryption capability allows organizations the flexibility to provides differing levels of encryption as appropriate to the application or individual form and allows this to be updated as new encryption standards come into the market.

1.3.5 Proven eForm technology

While the name of this product includes IBM Workplace (intended to reinforce the integration with other Workplace products in the IBM Software portfolio), IBM Workplace Forms is based on proven technology from PureEdge solutions, acquired by IBM in 2005. The product has evolved over more than a decade of improvements, based on input from more than two hundred customers and more than 5 million users. Beginning with a solid foundation of eForm technology, from PureEdge Solutions — a leading provider of secure standards-based eForm solutions for automating business processes — coupled with IBM's experience and customer base, Workplace Forms products provide a valuable addition to the IBM software portfolio and enhances IBM's market leadership in providing industry specific eForms solutions.

Key industries where Workplace Forms have been implemented include: government, insurance, banking, manufacturing and healthcare.

One customer example worth calling out is : the US Army, which selected Workplace Forms over the competition for what Gartner Group calls "the largest eForms implementation" in the world. The Army's Forms Management Content Program (FCMP) involves the automation of their inventory of 100,000 forms used by 1.4M personnel worldwide. Estimated cost savings of this forms implementation surpass \$1.3 annually. The Army forms program integrates Workplace Forms with CM in a Portal environment.

Reasons that the Army chose Workplace Forms over the competition included: sectional signing within a form, ability to generate a wizard-like interface (like Turbo-tax), and offline use among others.

1.4 Summary

This redbook provides a both an overview of the features and functionality of IBM Workplace Forms, while also providing a specific sample application scenario. After clarifying the features of the product and defining the options for integration, the remainder of the book, beginning with Chapter 4, "Building the base scenario: Stage 1" on page 53, helps you gain hands-on experience building a sample forms based application. Using this sample application as a foundation, we then discuss integration options with WebSphere Portal, IBM DB2 Content Manager and Domino.



Features and functionality

This chapter provides an overview of the features and functionality of the components that make up IBM Workplace Forms. The chapter begins with an introduction to the Workplace Forms Document Model and an overview of the Workplace Forms Component Technology. Working from this foundation, it then discusses functionality of the components, including:

- ▶ “Workplace Forms Designer” on page 19
- ▶ “Workplace Forms Viewer” on page 20
- ▶ “Workplace Forms Server” on page 21
- ▶ “Workplace Forms Server API” on page 21
- ▶ “Workplace Forms Webform Server” on page 23
- ▶ “Workplace Forms Deployment Server” on page 26

2.1 Form Document Model

Workplace Forms is built upon proven, standards based technology, including XForms and Extensible Forms Description Language (XFDL). As Web-based electronic forms have become more ubiquitous, XForms has emerged as a W3C specification for Web forms to overcome limitations with HTML Forms, while XFDL is a highly-structured XML protocol designed specifically to solve the body of problems associated with digitally representing paper forms on the Internet.

Extensible Forms Description Language (XFDL) is an XML syntax for describing complex, intelligent business forms. XFDL allows you to create powerful, complex forms that integrate with server-side applications such as workflow, databases, and security structures.

XFDL is:

- ▶ Human-readable plain text
- ▶ Open standard
- ▶ Extensible (allows you to add custom items, options, and external code functions)
- ▶ Document-centric

The advantages of XFDL include:

- ▶ Document-centric forms that maintain the data, logic, and presentation layers of a form in a single, legally binding document.
- ▶ Signatures on document-centric forms identify the signer and are representative of the context in which they were signed — just like signatures on a paper form.
- ▶ XFDL forms are ideal for high-value business transactions, regulated industries, or other security conscious organizations.

XFDL is written as plain text. Each form is a single text file containing standard XML syntax. As a result, XFDL is a tagged language. This means you must use both opening and closing tags that are wrapped by angle brackets. Opening tags indicate the start of a specific form element. Closing tags, marked with a slash before the tag name, indicate the end of that description (Example 2-1).

Example 2-1 Opening and closing tags

```
<PAGE> </PAGE>
```

Opening and closing tags surround information that describes a specific element of your form. For example, the following tags identify the form as XFDL and also give the namespaces used, and must appear at the beginning of the form (Example 2-2).

Example 2-2 XFDL form tag

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XFDL xmlns="http://www.PureEdge.com/XFDL/6.5"
xmlns:custom="http://www.PureEdge.com/XFDL/Custom" xmlns:xfdl="http://www.PureEdge.com/
XFDL/6.5">
...Form Data...
</XFDL>
```

2.2 Workplace Forms Component Technology

Although a paper form is composed of paper and ink, its structure and content is made up of graphics and text. For example, a certain form may be three pages long and printed on grey paper. It also contains a number of carefully positioned labels and fields, and has different font colors for titles and text. A document replicates these elements electronically.

Documents are composed of five types of elements:

- ▶ **Form:**

The form itself. Every form has exactly one form element. Its purpose is to identify the code as an XFDL form.

- ▶ **Page:**

Every form contains at least one page. Pages control the background appearance of your form and contain your form items in much the same way paper pages do.

- ▶ **Item:**

The objects that appear on the page and make up the form content. These include fields, labels, and buttons.

- ▶ **Option:**

The elements that describe the appearance and actions of items. You can use options to set an item's color and font, or set the defaults for a print button.

- ▶ **Argument::**

Arguments contain the settings used by options. For example, arguments store the three numbers that make up an RGB value, such as 255, 255, 255.

2.3 Workplace Forms Designer

IBM Workplace Forms Designer is a drag-and-drop design program that allows form designers to create highly detailed, powerful XFDL forms. In addition to easy item placement, alignment, and configuration, the Designer also allows easy access to the underlying source code of the form, enabling form designers to include complex logic and calculations in the forms. The Designer works in tandem with IBM Workplace Forms Viewer. Form designers use the Designer to build the form, and use the Viewer to check form appearance, layout, and logic as they proceed.

Workplace Forms Designer is a drag-and-drop design tool that allows you to create detailed XFDL forms. The Designer has four views, allowing form designers to control every aspect of their form. These views include:

- ▶ **Form View:**

Offers easy item placement, alignment, and configuration.

- ▶ **Tab Order View:**

Simplifies setting the tab order for the form.

- ▶ **Tree View:**

Shows the build order at a glance and allows you to quickly modify form elements, especially option settings and references.

- ▶ **Code View:**

Displays the XFDL code for the form, allowing you to add complex logic and calculations.

Figure 2-1 illustrates the Designer's primary interface.

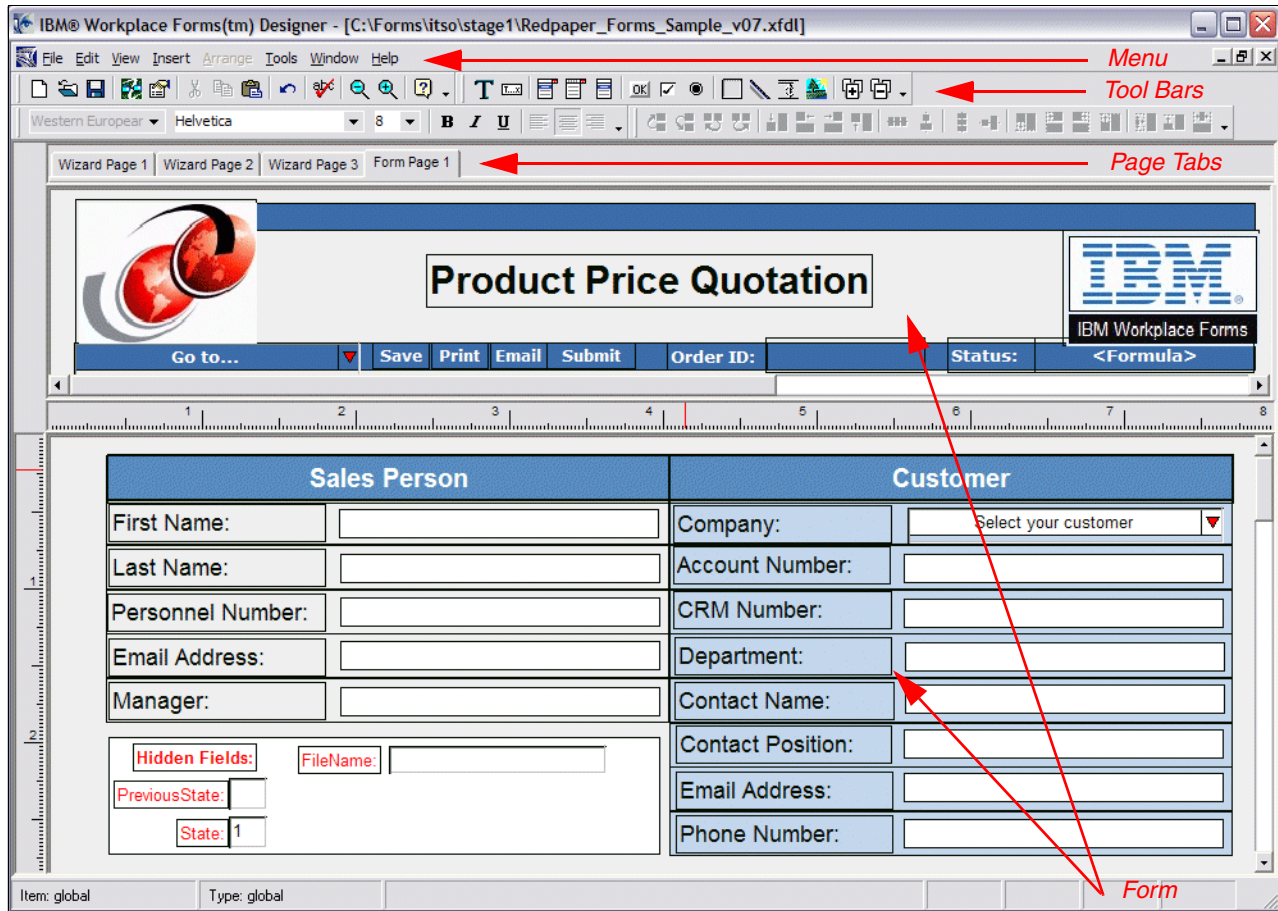


Figure 2-1 Workplace Forms Designer Primary Interface

We will describe the Workplace Forms Designer in greater detail when we build our sample application in Chapter 4, “Building the base scenario: Stage 1” on page 53, and Chapter 5, “Building the base scenario: Stage 2” on page 145.

2.4 Workplace Forms Viewer

Workplace Forms Viewer provides a single interface for users to open, review, or complete XFDL forms. Additionally, the Viewer is interactive. While users complete forms, the Viewer responds to user input.

The Viewer has two modes:

- ▶ Standalone
- ▶ Browser interface

Both modes have full online and offline functionality.

Figure 2-2 shows the Workplace Forms Viewer interface.

The screenshot displays the IBM Workplace Forms Viewer interface. At the top is a title bar labeled 'Form Page 1'. Below it is a toolbar with various icons. The main content area features a 'Product Price Quotation' form. The form has a header with the IBM logo and the text 'IBM Workplace Forms'. Below the header is a navigation bar with buttons: 'Go to...', 'Save', 'Print', 'Email', 'Submit', 'Order ID:', 'Status:', and 'Template'. The form itself is divided into two columns: 'Sales Person' and 'Customer'. The 'Sales Person' column contains fields for 'First Name:', 'Last Name:', 'Personnel Number:', 'Email Address:', and 'Manager:'. The 'Customer' column contains fields for 'Company:', 'Account Number:', 'CRM Number:', 'Department:', 'Contact Name:', 'Contact Position:', 'Email Address:', and 'Phone Number:'. Red arrows point to the 'Title Bar', 'Tool Bar', 'Form Toolbar (optional)', and 'Form'.

Figure 2-2 Workplace Forms Viewer Interface

2.5 Workplace Forms Server

In this section we describe the various features of the Workplace Forms Server.

2.5.1 Workplace Forms Server API

The IBM Workplace Forms Server - Application Programmer Interface (API) is a collection of specialized functions that you can use to develop applications that interact with XFDL forms. The API provides a C and Java interface for Windows® and UNIX® computers, as well as a COM interface for Windows platforms.

The API is divided into two sections, reflecting two sets of functionality:

- **The Form Library:**

The Form Library is a collection of functions that enables you to access and manipulate XFDL forms as structured data types. These functions provide your applications with a means of reading and writing forms, retrieving information contained in graphical form elements, and assigning information to these items. Once the API loads the form into memory, you can use Java methods (or C functions) to populate the form, retrieve data, duplicate and destroy form elements, extract enclosures, and even verify digital signatures.

► **The FCI Library:**

The FCI library is a collection of specialized functions that enables you to develop your own functions. XFDL forms can then access these custom functions, thereby extending the capabilities of the Internet Commerce System, without requiring an upgrade to either your forms software or to the form description language (XFDL). This allows you to provide custom form functions that you can call from within the form. This is similar to calling system form functions such as set or toggle.

Typical API uses

The Workplace Forms API is used in a variety of ways and in several places. On the client side of things, developers may create extensions to the Workplace Forms Viewer, using the API, that allow form manipulation — that is, modifications to the form (usually in response to user input) as the user completes it. These extensions are referred to as IFXs. An IFX can also be used with the client-side Viewer to integrate with other hardware, typically hardware that collects information that must eventually end up in a form, such as scanners, bar code readers, or GPS (Global Positioning Systems).

On the server side, the API may be used within a Java servlet or similar piece of code that is designed as an interface between Web-based forms and a database. This database interaction permits automatic retrieval of data to prepopulate form items, and seamless submission of form data (or the entire form) to the database. Workflow integration can be accomplished in much the same way.

Occasionally it may be necessary to integrate with additional libraries. On either the client side or the server side, using the API will allow you to use function libraries not provided with Workplace Forms API, such as voice input libraries.

Compatible technologies

Any application that has its own application programmer interface in either Java, COM, or C can be integrated with the Workplace Forms API. Most modern databases also support XML natively so storage and retrieval of form or XML model data can be quick and seamless and extensive field to database coding can be minimized.

Additionally, because the Workplace Forms form is a complete document, forms can be stored directly in the database and retrieved as a whole document.

Standard architectures

On the server the API is commonly used with servlets, Java Server Pages (JSP™), and Common Gateway Interfaces (CGI).

Custom client side functionality is generally supported through custom IFX extensions (code extensions) provided in either Java, C, or both. Java extensions that do not require any system access may be embedded directly in a form.

Workplace Forms API versus XML Parsers

Some developers may prefer to use a third party XML parser to interface with their forms. Depending on the parser and the types of tasks to be performed, this may provide some benefits in terms of speed or developer comfort (if the developer has a great deal of experience with XML parsers).

However, the API is able to perform some tasks that are impossible for an XML parser. For example, computes can continue to run in the form (or be activated by the API) while the form is in memory, so any form data that is dependent on continuously evaluated computes will remain accurate. Additionally, since most applications use compressed forms, which cut down

transmission speeds and use of disk space dramatically, the API is able to automatically decode Base 64 data and uncompresses forms as it reads them. With an XML parser, you must force it to decode and uncompress the forms to run them.

The API also provides methods for verifying and handling digital signatures. Most applications need to verify signatures on the server side, and occasionally even apply signatures on the server. These tasks are virtually impossible to perform without the Workplace Forms API.

The API can also encode and decode data items stored such as images, enclosures, and digital signatures. In the case of images and enclosures, this means that using the API it is possible to extract attachments or images from the form and store them separately on the server., or insert attached files into forms as they are sent out to the user.

2.5.2 Workplace Forms Webform Server

IBM Workplace Forms Webform Server is a zero footprint solution that allows users to open, complete, and submit forms using only a Web browser. Webform Server is generally the best solution if your forms contain little logic and you need to distribute them to a large user base, such as the general public.

Webform Server uses a collection of server components to intercept requests for XFDL forms, translate those forms into HTML and JavaScript™, and return the translated forms to the user. Webform Server can then receive completed HTML forms, translate them back into the original XFDL, and pass the completed XFDL forms to a processing application.

This allows end-users to view and fill the forms without installing any client software — instead, the user works with an enhanced HTML form in their regular Web browser. However, because there is no client software, Webform Server does not offer all of the functionality that is available with the Viewer.

Basic architecture

Webform Server relies on three central components: a portlet or servlet, a Translator, and a Log Server. The Translator also contains two sub-components: an Access Control Server (ACS) and a File Cache.

Figure 2-3 shows how these components are set up in a typical installation.

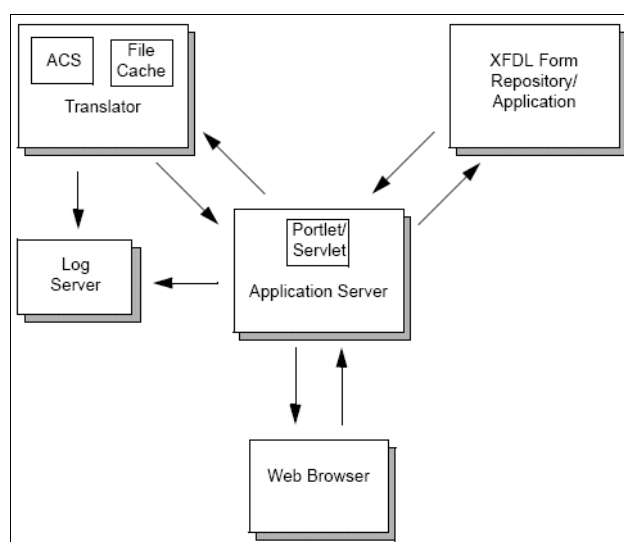


Figure 2-3 Workplace Forms Webforms Server Architecture

The portlet/servlet controls the basic incoming and outgoing form processing. This component passes form requests to the Translator for form conversion between XFDL and HTML. Optionally, it interacts with other applications, and/or an external forms repository. Each form application has its own portlet/servlet; it is up to the developer to create the portlet/servlet for a particular application. The Webform Server provides a framework for creating the portlet/servlet. The Webform Server documentation also includes complete documentation for how to create, configure, and implement your own portlets and servlets, including descriptions of the available methods and (for more advanced programmers) APIs. (If you plan to use APIs, you must install the separate Workplace Forms Server - API.)

Servlets and portlets can be relatively simple, or more sophisticated. You can even design a portlet that uses multiple “sub-portlets” (useful for displaying multiple panes to the user). To help you get started, the Webform Server comes with a sample servlet and sample portlet, which you can use and adapt to the needs of your own forms application.

The Translator consists of two sub-components, the access control server and the file cache. The Translator performs the conversion from XFDL to HTML, and back again. When it translates a form into HTML, the Translator stores the original XFDL form in the file cache, and metadata in the access control server. To convert the form from HTML back to XFDL, the Translator retrieves the original form from the file cache, and transfers the data received from the completed HTML form.

The Log Server records all transactions performed by the portlet/application and Translator. This can be useful for performance analysis, error checking, and troubleshooting.

These components work together to execute the three basic tasks performed by the Webform Server: forms requests (retrieving a requested XFDL form, translating it to HTML or JavaScript, and presenting it to the user), forms submissions (receiving a completed HTML form from the user, converting it back into XFDL, and processing it as appropriate), and “special actions” (for example, re-computing data on the form).

Differences between Webform Server and the Viewer

Webform Server allows users to complete and submit forms without the need for any client-side software other than a Web browser. However, in the absence of specialized client-side software, Webform Server cannot support the full-range of functionality that is offered by IBM Workplace Forms Viewer.

In many cases, these differences in functionality require a different approach to form design. For instance, forms designed for use with the Viewer may include rich text fields and computes that rely on the event model. However, since neither of these features is supported by Webform Server, these forms would not work in a Webform Server environment.

As a general rule, any form that works with Webform Server will also work with the Viewer, but the reverse is not true. If you are designing forms for an environment that uses both Webform Server and the Viewer, be sure to restrict the functionality of the forms to those features that Webform Server supports.

Table 2-1 lists a number of other differences between the Webform Server and the Viewer.

Table 2-1 Differences Between Webform Server and the Viewer

Functionality	Webform Server	Viewer
Action Items	Updates computes when the user refreshes the form	Updates in realtime
Appearance of forms	Slightly different look due to HTML widgets	Pixelperfect Rendering

Functionality	Webform Server	Viewer
Attachments	One file at a time	Multi-select files
Computes	User must click button to update computes	Updates computes while user is filling out the form
Calendar Widget	Not supported	Supported
Event Model	No Compute system is only updated upon submission to server	Updated while user is working with the form
Printing	Preview is a PNG image of the form that is generated on the server	Page size can be changed from form to form
Signatures	Allows users to sign forms using Clickwrap signatures with limited security	Full range of supported digital signatures can be used.
Type Checking and Predictive Input Checking	Since predictive input checking relies on an immediate response to user it is not supported	Checked while user is filling out the form
URLs	Does not allow users to submit forms to multiple URLs at the same time.	Multiple URLs are supported
XML Data Model	Data Model is not updated while the user is filling out the form.	Updated while user is filling out the form
Email	Partial support - Users must save forms to their local computer and email them as attachments via email program	Full support
Form version support	Version 6.0 and later	Versions 4.4 and later
Localization	English only	English and French (Canadian)
Realtime error and format checking	Not supported	Supported
Rich Text Fields	Not supported	Supported
Schema	Server-side only	Client and Server
Screen readers	JAWS only	MSAA compliant
Smartfill	Not supported	Supported
Spellchecking	Not supported	Supported
User modification of display or print preferences	Not supported	Supported
Viewer functions, such as fileOpen, messageBox, setCursor, and so on.	Not supported	Supported
Web services	Not supported	Supported
Zoom capability	Not supported	Supported

2.5.3 Workplace Forms Deployment Server

Deployment Server allows you to deploy software to your users through standard Web browsers, including Microsoft Internet Explorer® and Netscape Navigator. Deployment Server automates this process, requiring little or no user interaction, and ensures that the right components are installed for each user.

Deployment Server relies on three key components: the Deployment Server applet, which manages the deployment process, the Deployment Server servlet, which passes files to the applet, and the Deployment Server file system, which stores the instructions and the files the applet uses to install the software. The applet runs in the user's Web browser, and communicates with the servlet, which is installed on the server.

Basic architecture

Deployment Server has three key components:

- ▶ An applet that controls the installation of software on the end-user's computer.
- ▶ A servlet that communicates with the applet and passes both instructions and files to the applet.
- ▶ A file system that stores all of the instructions and files necessary for deployment.

Figure 2-4 shows how these components work together.

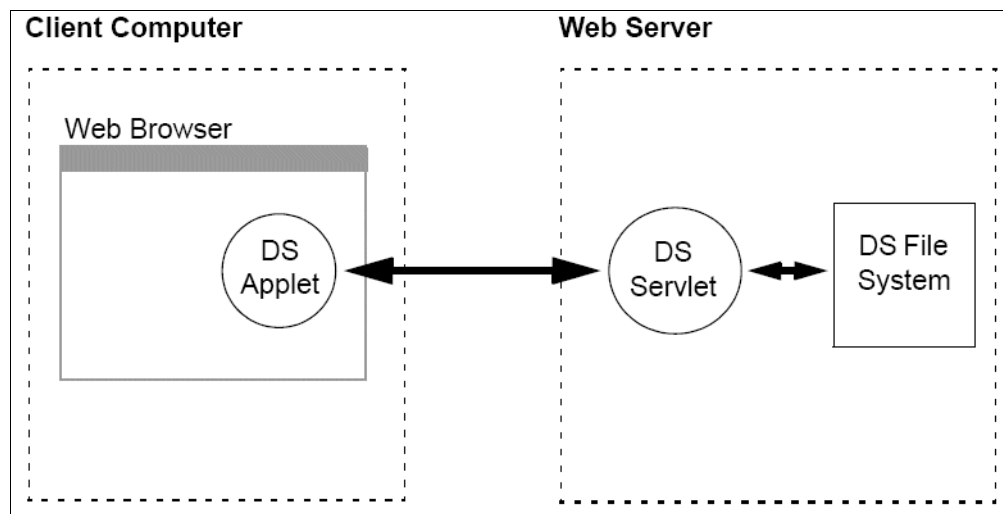


Figure 2-4 Workplace Forms Deployment Server Architecture

The file system stores two types of files, called manifests and packages. Each manifest is a set of instructions that the applet follows when deploying an application. Each package is a set of installation files that may install either a complete application or a portion of an application.

A typical software deployment follows these steps:

1. The user opens a Web page that contains the Deployment Server applet.
2. The applet runs, and requests updated instructions (the current manifests) from the servlet.
3. The servlet retrieves the manifests from the file system, and passes them to the applet.
4. The applet reads the manifests and checks the configuration of the user's computer.
5. Based on the logic in the manifests, the applet decides which packages to install.

6. The applet requests the necessary packages from the server.
7. The server retrieves the packages from the file system, and passes them to the applet.
8. The applet runs each package in turn.
9. When run, each package installs an application or a portion of an application on the user's computer.
10. The applet monitors the installation, and loads a success or failure page, depending on the results.

During installation, the user will normally see a dialog that lists the components being installed, and may have the option of refusing some or all of the components. Once the installation is complete, the dialog will disappear and the applet will load a success or error page.



Approaches to integrating Workplace Forms

This chapter provides describes a range of approaches for integrating Workplace Forms into one's environment. We begin by defining the context of integrations, then describe in greater detail the different levels possible for integration.

We consider these topics:

- ▶ Integration: what this means in the context of Workplace Forms
- ▶ Workplace Forms document model and straight-through integration
- ▶ Aspects of integrating Workplace Forms
- ▶ Integration points summary
- ▶ Partitioning of features and functionality

3.1 Integration: what this means within the context of Workplace Forms

Workplace Forms provides is a component technology that is based on open standards. The discussion of integration typically depends on the context of the customer scenario, infrastructure, and solution scope.

As a level-set, the simplest use of Workplace Forms is standalone — one can launch the Workplace Forms Viewer by double-clicking a form on their desktop. This will cause the Workplace Forms Viewer to launch as a standalone application and present the form for data entry or viewing (Figure 3-1).

The screenshot shows a desktop application window titled "Page1". The window contains the IBM Workplace Forms logo and a red header bar with navigation buttons: "Save", "Print", "Close", "Help", "Wizard", and "Next >>". The main form is titled "Heavy Highway Vehicle Use Tax Return" and is for the period July 1, 2003, through June 30, 2004. It includes instructions to attach both copies of Schedule 1 to this return. The form is divided into sections: "Type or Print", "Check here if:", "Part I Figuring the Tax", and "Part II Statement in Support of Suspension". The "Type or Print" section includes fields for Name, Employer identification number, Address (number, street, and room or suite no.), City, state, and ZIP code (For Canadian or Mexican address, see page 3 of the instructions.), and Final return. The "Check here if:" section includes checkboxes for Address change and Final return. The "Part I Figuring the Tax" section includes lines 1 through 6 for calculating the tax. The "Part II Statement in Support of Suspension" section includes statements 7 and 8a regarding vehicle suspension.

IBM Workplace Forms

Save | Print | Close | Help | Wizard | Next >>

Heavy Highway Vehicle Use Tax Return

Heavy Highway Vehicle Use Tax Return
For the period July 1, 2003, through June 30, 2004
Attach both copies of Schedule 1 to this return.
See the separate instructions.

Keep a copy of this return for your records.

Type or Print

Name

Employer identification number

USE ONLY

Check here if:

Address change ☐

Final return ☐

City, state, and ZIP code (For Canadian or Mexican address, see page 3 of the instructions.)

Part I Figuring the Tax

1 Was the vehicle(s) reported on this return used on public highways during July 2003 ?
If YES, enter 200307 in the boxes to the right. If NO, see page 3 of the instructions. ▶ 1 YYYY MM

2 Total tax. Enter the Totals from Form 2290, page 2, column (4). ▶ 2

3 Additional tax from increase in taxable gross weight. See page 3 of the instructions ▶ 3

4 Credits. See page 3 of the instructions ▶ 4

5 Tax as adjusted. Add lines 2 and 3, then subtract line 4 from the total. This is the amount you owe. If paying in installments, go to line 6. If payment through EFTPS, check here. ☐ ▶ 5 \$0.00

6 Installment payment. See page 5 of the instructions ▶ 6

Part II Statement in Support of Suspension (Complete the statements that apply. Attach additional sheets if needed.)

7 I declare that the vehicles listed in Part II of Schedule 1 are expected to be used on public highways (check the boxes that apply):
☐ 5,000 miles or less ☐ 7,500 miles or less for agricultural vehicles
during the period July 1, 2003, through June 30, 2004, and are suspended from the tax. Complete and attach Schedule 1.

8a I declare that the vehicles listed as suspended on the Form 2290 filed for the period July 1, 2002, through June 30, 2003, were not subject to the tax for that period except for any vehicles listed on line 8b. Check this box if applicable. ☐

Figure 3-1 Baseline example: the Workplace Forms Viewer running standalone as a desktop application

The Workplace Forms Viewer installation package provides us with the Workplace Viewer both as a standalone application and also as a browser plug-in.

Now, let us consider the standard means of integrating Workplace Forms. At a high level, there are a number of kinds of integrations:

- ▶ User Interface (UI) Integration
- ▶ Data Integration
- ▶ Process Integration
- ▶ Security Context Integration
- ▶ Client-Side Device / Hardware Integration

3.2 Workplace Forms document model and straight-through integration

Insight into the Workplace Forms document model will help us to understand what happens within the form when we prepopulate a form template, enter data, validate signatures, extract data, or interact with a process.

3.2.1 The Workplace Forms document model

First, let us examine the high-level components that make up a Workplace Forms document.

Workplace Forms are structured XML files, which contain separate data model and user-interface (Figure 3-2).

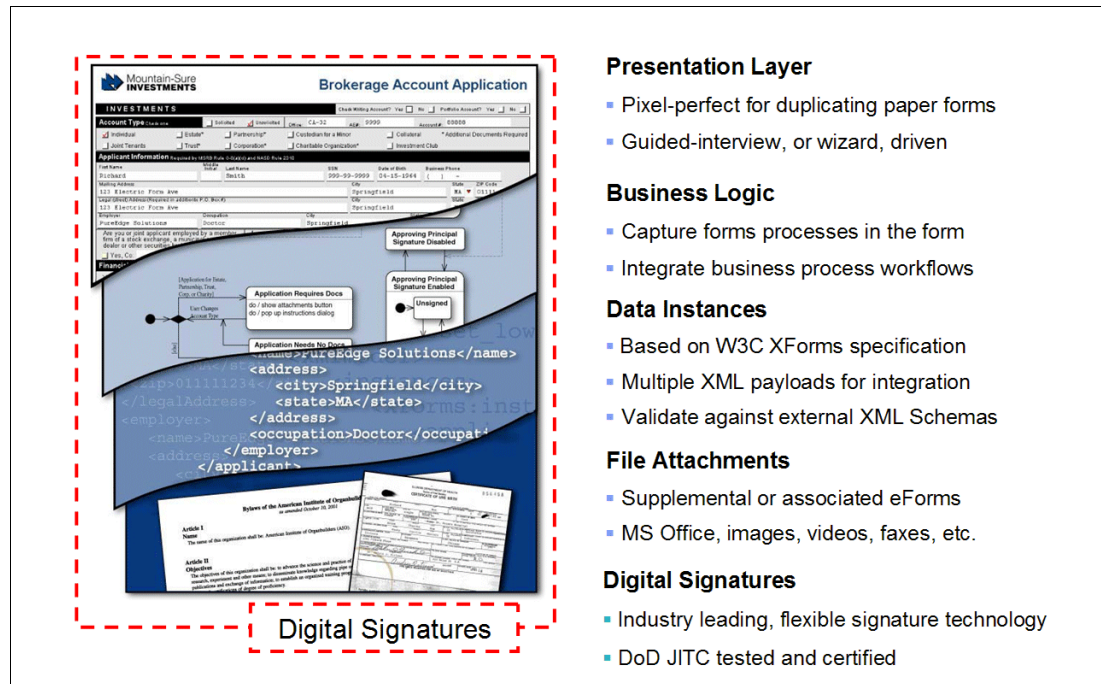


Figure 3-2 Workplace Forms: structured XML files with separate data model and user interface

Next, Figure 3-3 shows a conceptual view of a Workplace Form document model.

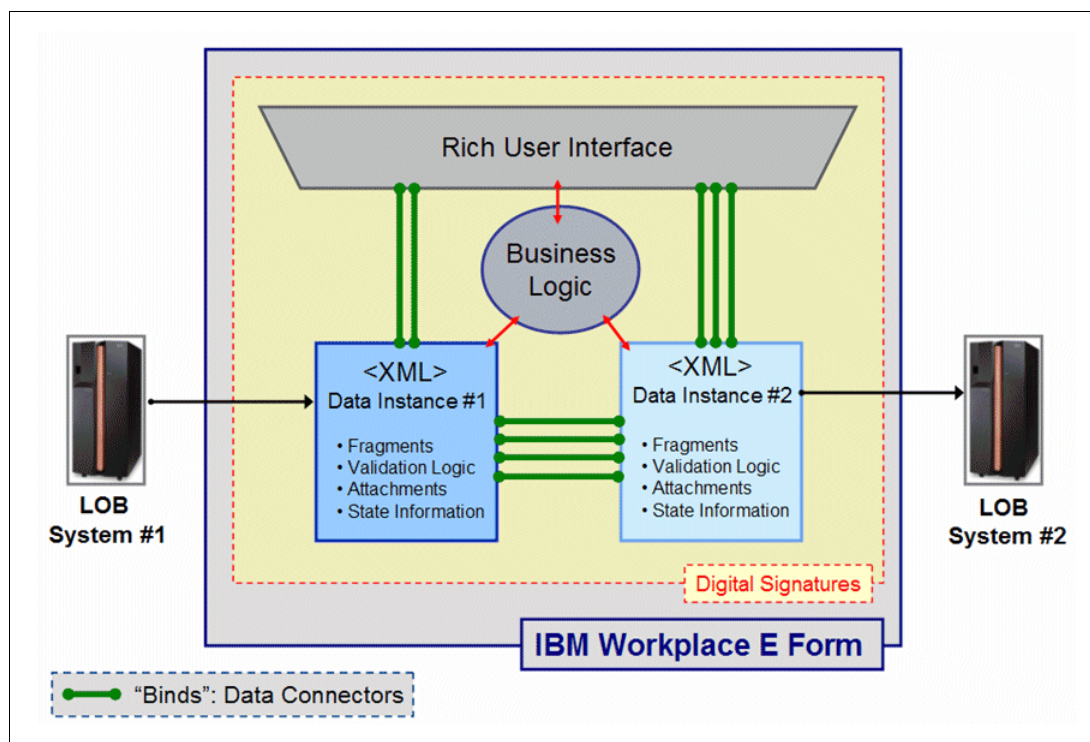


Figure 3-3 Conceptual view of a Workplace Form document model

In Figure 3-3 above, note that it is a conceptual representation of an IBM Workplace Form. The form provides separation of user-interface (also referred to as presentation layer) and data model (from 0 → N discrete data instances).

In the above example, the form data model consists of two data instances. In practice, the form could contain no data instance at all, or many different data instances. Data Instance #1 is used for form prepopulation while Data Instance #2 is used for data integration to a Line-Of-Business system.

3.2.2 Support for arbitrary XML instances

Support for arbitrary XML instances refers to the fact that you can take an XML schema from any external system (or alternately make create your own), generate a compliant XML data instance, then embed that instance into the form. The elements of this instance are related to other aspects of the form by *binds*, shown in green in the figure above. The form can contain 0 → n instances.

3.2.3 Straight-through integration

Straight-through integration refers to the ability of Workplace Forms to produce XML schema compliant data instances without the need for translation — the compliant data instance, or instances, reside within the form.

The usefulness of Workplace Forms' straight-through integration functionality is enhanced by the ability to integrate data to multiple, disparate systems without requiring any translation. Furthermore, data can be presented in different ways to each system! For example, if an employee number must be 8 digits for a green-screen legacy system, and is expected to be 16 digits for a complementary, modern HR system, a single form can contain two different data instances, and populate the data elements within it in different ways, as needed by each system. Once again, no translation is required.

Lastly, you will notice in both of the previous two figures, digital signatures can be applied to each aspect of the form, ensuring that the forms are tamper-proof, or, at a minimum "tamper-evident".

3.3 Aspects of integrating Workplace Forms

Now that we have covered the basic structure of Workplace Forms, let us drill-down and examine each of the different types of integration in more depth.

3.3.1 User Interface (UI) Integration

In this section we begin our discussion of integration.

Display of Workplace Forms within a Web Page

We start by illustrating the most straightforward type of integration, namely, a form rendered within the context of a Web page. Figure 3-4 is an example of a pixel-precise traditional form page, suitable for printing.

03_Commercial Line_of_Credit.xfdl - Microsoft Internet Explorer

File Edit View Go To Favorites Help

Back Forward Stop Home Address D:\cmarston\Sales\03.Commercial Line_of_Credit.xfdl Links

IBM Workplace Forms

Go To Wizard Submit Form << Back Next >>

IBM Account Origination Demonstration Form IBM_WP_Form_IBM_D01

21st March 2006

New Account Application

I. ACCOUNT INFORMATION				[BANK USE ONLY]	
Account Applied for:	<input type="checkbox"/> Personal Account	<input checked="" type="checkbox"/> Commercial Line-of-Credit	Application #	C.R. Required	Credit Score
	<input type="checkbox"/> Commercial Account		320-26927		
Company Name			Interest Type	<input type="checkbox"/> Fixed Rate	Comments
Desired Credit \$				<input type="checkbox"/> Variable Rate	

Applicant				II. BORROWER INFORMATION				Co-Applicant			
First Name		Middle Initial	Last Name	First Name		Middle Initial	Last Name	First Name		Middle Initial	Last Name
Social Security Number		Age	Education:	Social Security Number		Age	Education:	Social Security Number		Age	Education:
Marital Status:		# Dependents:		Marital Status:		# Dependents:		Marital Status:		# Dependents:	
Address				Address				Address			
City				City				City			
State				State				State			
Zip Code				Zip Code				Zip Code			
Phone				Phone				Phone			
() -				() -				() -			
If residing at present address for less than two years, complete the following:											
Address				Address				Address			
City				City				City			
State				State				State			
Zip Code				Zip Code				Zip Code			
Phone				Phone				Phone			
() -				() -				() -			

Applicant				III. EMPLOYMENT INFORMATION				Co-Applicant			
Name and Address of Employer		Current Status:	Employed	Yrs. on this job		Yrs. of work		Name and Address of Employer		Current Status:	Employed

Unknown Zone

Figure 3-4 Workplace Forms Viewer displaying a form within the Web browser

Display of forms to end users within a Web application context is perhaps the most common and basic form of User Interface (UI) integration. In Figure 3-4, the Workplace Forms Viewer is running within the browser as a plug-in. Users typically navigate to forms by following links within a company Internet or intranet site. Alternately, HTTP links to Workplace Forms can be provided to users via e-mail, although this involves application-tier integration with an e-mail system or daemon.

Figure 3-5 illustrates an example of an interview-style, Wizard form page.

03_Commercial Line_of_Credit.xfdl - Microsoft Internet Explorer

File Edit View Go To Favorites Help

Back Address D:\marston\Sales\03_Commercial Line_of_Credit.xfdl Links

IBM Workplace Forms

Commercial Line of Credit Account Application

Traditional Form Next >>

Account Type Selection

Account Type

Applicant

Co-Aplicant

Employment Info

TBD

Save Incomplete

E-Mail

Print

Thank-you for your interest. Please select an account type to start the application process.

☐ Personal Account

☐ Commercial Account

☒ Commercial Line-of-Credit

Company Name:

Desired Credit Limit:
[Maximum \$500,000]

The information that you provide will be kept confidential and secure and will not be sold or redistributed.

Figure 3-5 The Workplace Forms Viewer displaying another form page within a Web browser

Workplace Forms are 100% XML, and have a mime-type of:
application/vnd.xfdl

For file extensions:
.xfd and .xfd1

Once you set the return mime-type, when you return a Workplace Form, the browser will recognize and launch the Workplace Forms Viewer within the browser as a plug-in.

It is important to note that when creating forms applications, standard Web application design considerations apply.

Display of Workplace Forms within a portal page

Figure 3-6 shows a Workplace Forms Viewer displaying a mortgage pre-approval form within a Portlet.

The screenshot displays the ON DEMAND BUSINESS portal interface. At the top, there is a navigation bar with links for 'My Workplace', 'News', 'Docs and Forms', 'Business', and 'My Tasks'. A user greeting 'Welcome Cindy!' and a 'My Portal' button are visible. Below the navigation bar, a 'Documents' section shows a list of documents, including 'Mortgage Pre-Approval(submitted).xfd'. The main content area displays the 'Mortgage Pre-Approval' form, which includes fields for 'Mortgage Amount' (\$98,000.00), 'Down Payment' (with a tooltip stating 'This item is digitally signed and cannot be altered'), 'Interest Rate' (6%), 'Payment Frequency' (Monthly), 'Amortization Period' (30 Years), and 'Start Date' (2006-01-01). A 'Mortgage Calculator' pop-up window is also visible, showing fields for 'Calculate Monthly Payment' and 'Calculate Purchase Price'.

Figure 3-6 The Workplace Forms Viewer displaying a mortgage pre-approval form within a Portlet

Figure 3-6 illustrates an example of how Workplace Forms can be displayed within a Portal application. When delivered via Portlets, Workplace Forms can be used as elements of composite Portal applications. Inter-Portlet communication provides us with a convenient avenue for data-integration, such as form template prepopulation. As an example, one might compose a UI with a client-list JSP on the left side of the UI, and an eForm Portlet on the right. Selection of a client within the client-list JSP could be used to prepopulate a form template to the right.

Zero Footprint display of Workplace Forms

In certain situations, it may not be possible or desirable to install the Workplace Forms Viewer onto each user's computer. Typically, this situation results from:

- ▶ Desktop "lockdown" — users not having sufficient privileges to install the application
- ▶ Infrequent form users who do not wish to install another program
- ▶ Dial-up or low-bandwidth users who cannot download the Workplace Forms Viewer in a timely manner
- ▶ Version management / upgrade concerns
- ▶ Software licensing considerations

Whatever the reason, it may be desirable to provide access to forms without requiring the installation and management of the Workplace Forms Viewer. In this case, the Workplace Forms WebForm Server provides us with the ability to translate server-side XFDL documents into HTML and Javascript, so that they can be displayed within standard browsers — with no Viewer required. This is called the Zero Footprint display of Workplace Forms, as shown in Figure 3-7.

Attention: Please also refer to Chapter 7, “Zero Footprint with WebForm Server” on page 241, where we discuss in greater detail the considerations when implementing a Zero Footprint® solution using the sample scenario application presented in the redbook.

Wizard Page 1 - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Address http://localhost:8085/translator/Translate?pwsFormName=03_Commercial+Line_of_Credit.xfdl&pwsAction=pagedone Links

IBM Workplace Forms

Commercial Line of Credit Account Application

Traditional Form Next >>

Account Type

Account Type

Applicant

Co-Applicant

Employment Info

TBD

Save Incomplete

E-Mail

Print

Thank-you for your interest. Please select an account type to start the application process.

☒ Personal Account

☐ Commercial Account

☐ Commercial Line-of-Credit

The information that you provide will be kept confidential and secure and will not be sold or redistributed.

Local intranet

Figure 3-7 Rendering of Workplace Form using Zero Footprint option

An important note is that on the server-side, WebForm Server maintains the complete, XFDL form, giving us the previously described benefits of the Workplace Forms for straight-through integration to one or more systems while also enabling end-users without the Workplace Forms Viewer.

Display of Workplace Forms within Notes / Domino

Notes and Domino provide us with an excellent foundation onto which we can overlay Workplace Forms. Domino extends the Workplace Forms Products with a number of benefits

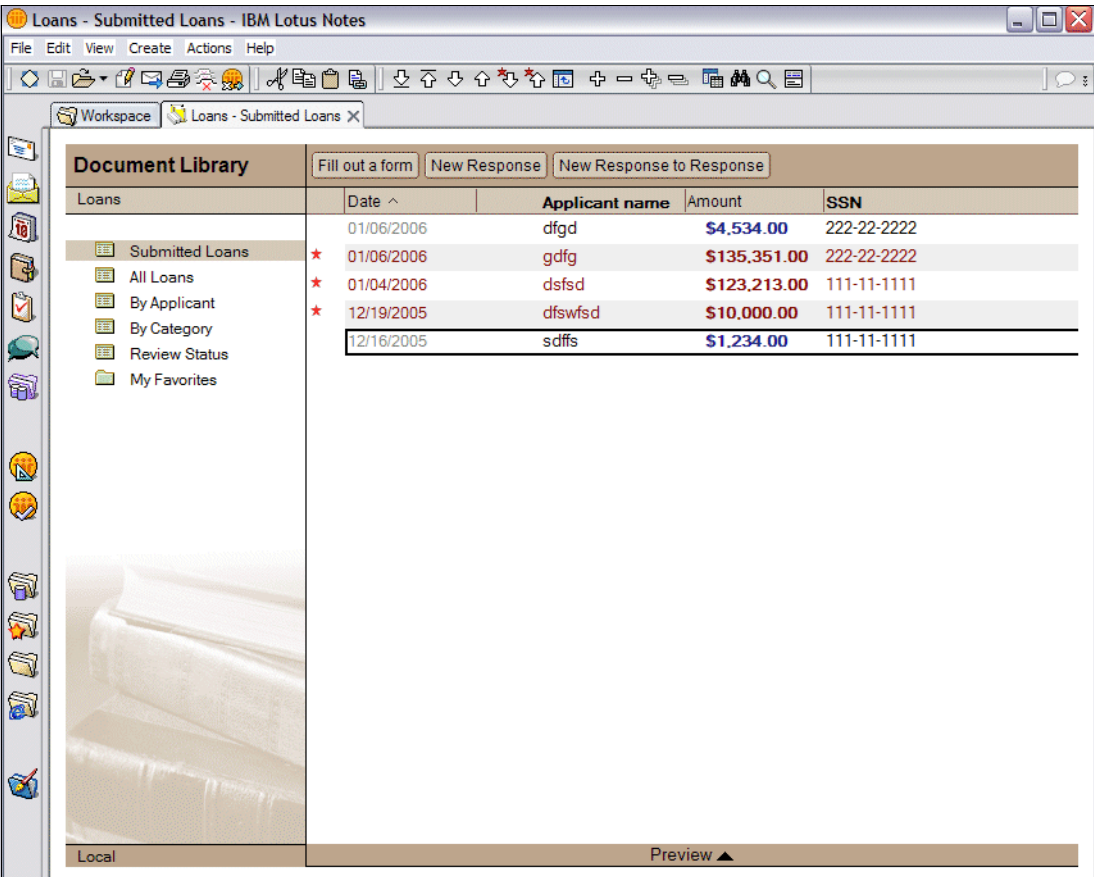
- ▶ Forms Management:
 - Domino agent reads attached form and displays in Domino view
 - Domino Forms live (attachments)
- ▶ Document-Based Workflow:
 - Place Workplace Forms into existing workflow
- ▶ Use of Domino mail as a transport
- ▶ Replication / Server-side
- ▶ Encryption
- ▶ As a Supporting Technology:
 - Support Workplace Forms with additional Domino content (FAQs, Travel policies, etc.)

Looking at it from the opposite perspective, Workplace Forms extends the capabilities of Notes / Domino with:

- ▶ Pixel Perfect Form Layout
- ▶ Guided, Wizard Page Front-Ends
- ▶ Overlapping Digital Signatures
- ▶ W3C XForms Support
- ▶ Form Extension (FCI / IFX), such as:
 - Device Integration (biometrics, signing tablets, etc.)
 - 3rd Part Encryption

Example screen displays of Workplace Forms used with Notes / Domino are given in the following sections.

Figure 3-8 illustrates a database of Workplace Forms.



Document Library				
Fill out a form New Response New Response to Response				
Loans	Date ^	Applicant name	Amount	SSN
Submitted Loans	01/06/2006	dfgd	\$4,534.00	222-22-2222
All Loans	01/06/2006	gdfg	\$135,351.00	222-22-2222
By Applicant	01/04/2006	dsfsd	\$123,213.00	111-11-1111
By Category	12/19/2005	dfswfsd	\$10,000.00	111-11-1111
Review Status	12/16/2005	sdffs	\$1,234.00	111-11-1111
My Favorites				

Figure 3-8 Database containing Workplace Forms

Clicking **Fill out a form** opens a new form template, shown in the following sample screen (Figure 3-9).

The screenshot displays the IBM Lotus Notes application window titled '(Untitled) - IBM Lotus Notes'. The interface includes a menu bar (File, Edit, View, Help), a toolbar with various icons, and a workspace area. The workspace shows several open documents: 'Workspace', 'Bemd Beilke - Inbox', 'Replication', 'Sales Quote Approval - Submitte...', 'Loans - Submitted Loans', and '(Untitled)'. The main content area displays a 'Mortgage Pre-Approval Form' template. The form has a sidebar on the left with a 'Steps' section containing links for 'Mortgage Requirements', 'Mortgage Details', 'Borrower Personal Data', 'Co-Borrower Personal Data', and 'Finalize Pre-Approval'. The 'Mortgage Requirements' section is currently active, showing a 'To Traditional Form' button and a 'Mortgage Requirements' heading. Below this, it asks to 'Provide the following mortgage information:' and includes fields for 'Mortgage Amount' (\$90,000.00), 'Down Payment' (\$10,000.00), 'Interest Rate' (5 %), 'Payment Frequency' (Monthly), 'Amortization Period' (20 Years), and 'Start Date' (2006-04-01). A 'Mortgage Calculator' pop-up window is also visible, featuring two sections: 'Calculate Monthly Payment' and 'Calculate Purchase Price'. The 'Calculate Monthly Payment' section includes fields for 'Purchase Price' (\$100,000.00), 'Down Payment' (10 % OR \$10,000.00), 'Interest Rate' (5 %), 'Amortization Period' (20 Years), and a 'Calculate Monthly Payment' button showing \$593.98. The 'Calculate Purchase Price' section includes fields for 'Monthly Payment You Can Afford', 'Down Payment', 'Interest Rate' (%), 'Amortization Period' (Years), and a 'Calculate Purchase Price' button.

Figure 3-9 Workplace Form displayed within Notes / Domino

Display of workplace forms within Eclipse

Figure 3-10 shows an example of a Workplace Form displayed within the Eclipse platform.

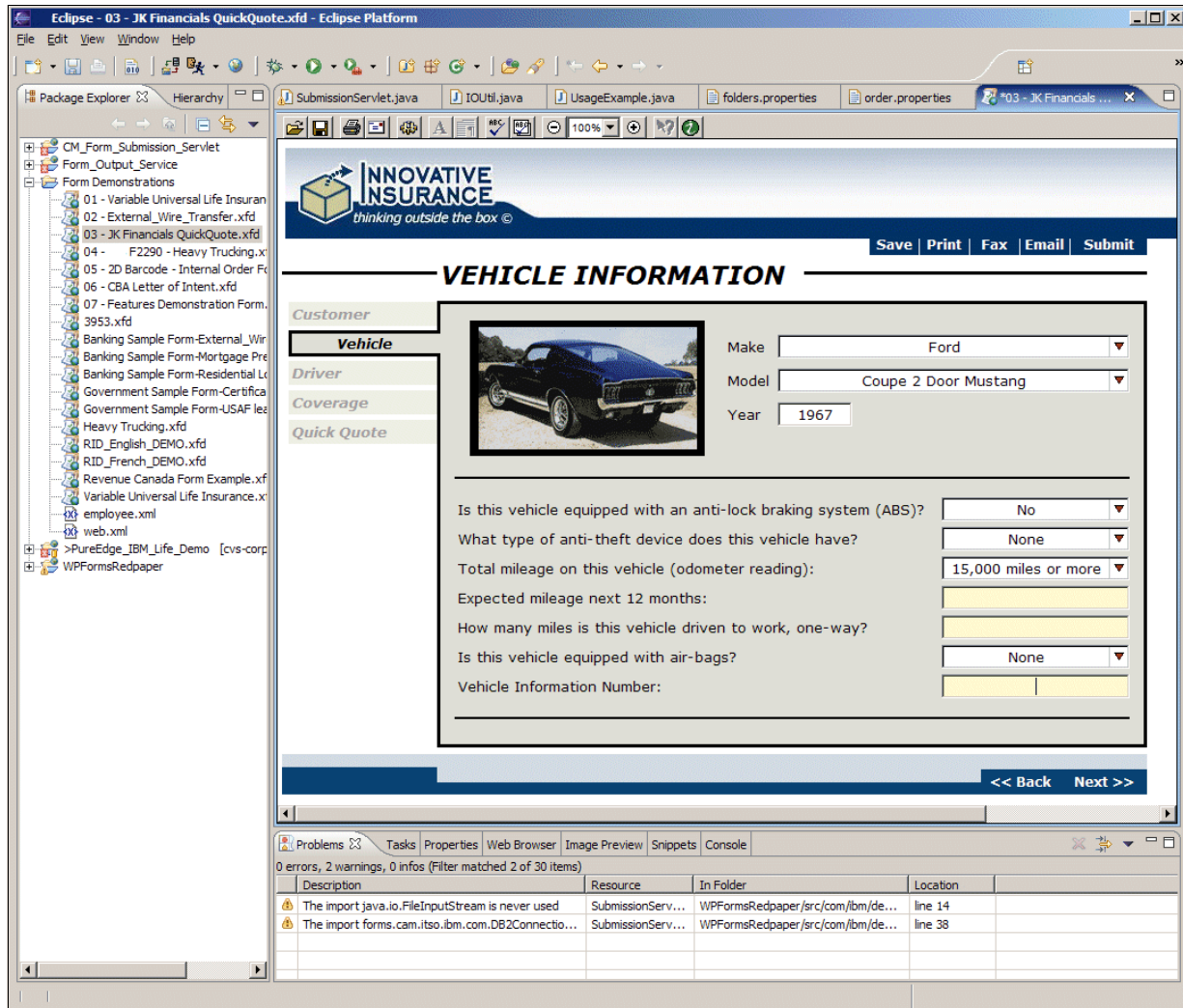


Figure 3-10 Example of a Workplace Form displayed within the Eclipse platform

It is also possible to run the Workplace Forms Viewer as a plug-in within Eclipse. As of the 2.6 release, the Workplace Forms Designer will be based on the Eclipse Platform.

This capability opens up a range of possibilities when one considers deploying forms and the Workplace Forms Viewer within a server-managed client infrastructure.

3.3.2 Data integration

Data integration is a broad topic and, depending on your definition of “data” it can have a number of meanings. Typically, data refers to information that is inserted, entered into, or extracted from forms. However, from a transactional (content or records management) perspective, the form itself can be considered data. Let us examine the most common data integration scenarios.

Storage of form templates and completed forms

Workplace Forms are 100% XML documents. While this may sound obvious, it bears repeating as the question often arises as to where form templates and completed forms are stored. Since the forms are XML, they are platform independent and can be stored onto the file-system of their choosing, to a database, content or records management system. Most commonly, form templates are stored:

- ▶ Within the WebRoot of a Web or Portal Application
- ▶ On the Web or Portal Server filesystem
- ▶ On a remote, mounted filesystem
- ▶ In DB2, Domino, or another database
- ▶ In Content Manager or Records Manager
- ▶ In Portal Document Manager

Retrieval of forms templates and persistence of completed forms typically occurs in the application tier. In J2EE based solutions, this occurs in the Servlet or Portlet. In Notes/Domino applications, this occurs within the application itself.

Server-side prepopulation of form templates

This is the most common form of prepopulation. At the time of the request for the form template, data is acquired from one or more systems and inserted into the form. The Workplace Forms API provides us with a number of methods/functions that make data insertion easy.

At a high level, one can either set individual data elements within the UI or data model, or, more commonly, enclose an entire data instance within a form. If desired, one can prepopulate a form with multiple data instances from different sources.

From an end-user experience perspective, this happens behind-the-scenes. One simply clicks on a link or button to request a new form, and is presented with a prepopulated form (Figure 3-11).

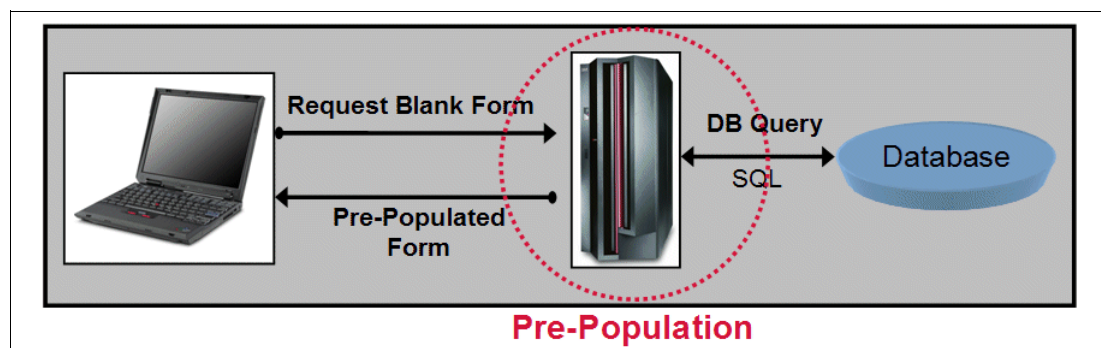


Figure 3-11 High-level example of server-side form template prepopulation call flow

It is important to note that Workplace Forms based solutions rely on standard Web and Portal Application authentication mechanisms to determine user identity and/or roles.

Real-time data ingestion via Web services

In some situations, data can be time sensitive, or, the form may require results based on calculations and data contained by external systems. In this case, the suggested approach is to make Web service calls from the client-side (Figure 3-12). When designing Workplace Forms, one can embed *Web Service Definition Language (WSDL)* files directly into the form itself and make Web service calls to:

- ▶ Submit / update information via an external service
- ▶ Obtain data from an external service
- ▶ Submit data for processing and obtain the result
- ▶ Initiate or claim a task, or update the state of an existing task (Workflow Integration)

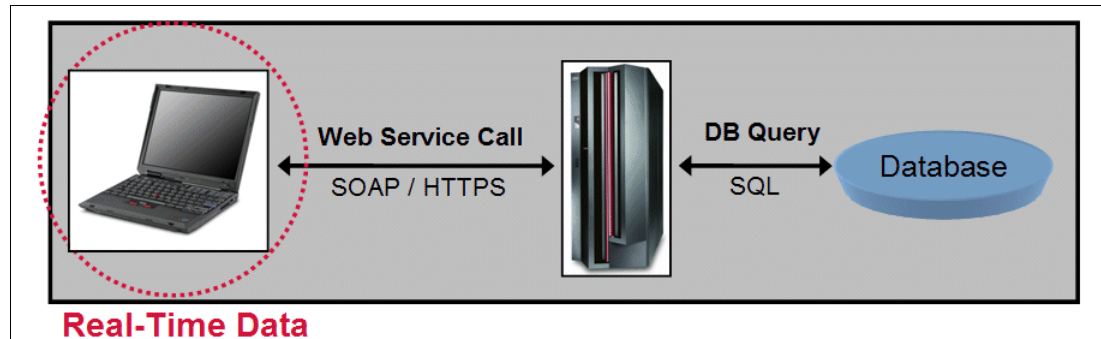


Figure 3-12 High-level example of client-side Web-service call flow

The Workplace Forms Viewer provides us with Web service support, so if you need to interact with a Web service when operating in Zero Footprint mode, then those calls will have to be made from the server side (server-to-service).

Local prepopulation — Smartfill

Smartfill is a means for prepopulating a Workplace Form with a user's locally cached XML data fragments. Over the years, a number of customers have had the need to support prepopulation, however, they have been in a situation in which users were either offline, or, for legal reasons, they were not able to store end-user data on the server-side.

When using Smartfill, data fragments are cached locally on a per-user basis, and access is protected by filesystem privileges. Note that Smartfill is not intended to store sensitive data such as credit-card numbers, banking information, passwords, or personal-ID numbers.

The Workplace Forms Viewer is required for Smartfill, and forms must be designed specifically to support this capability. At design-time, one can designate one or more Smartfill data fragments. These fragments can be either single elements, deeper XML data fragments, or entire instances. A single form can support from 0 →n Smartfill data fragments.

When using a Smartfill enabled form, when saving the form locally, users have the option of exporting Smartfill data for later reuse. The location of this data is automatically managed by the Workplace Forms Viewer (Figure 3-13).

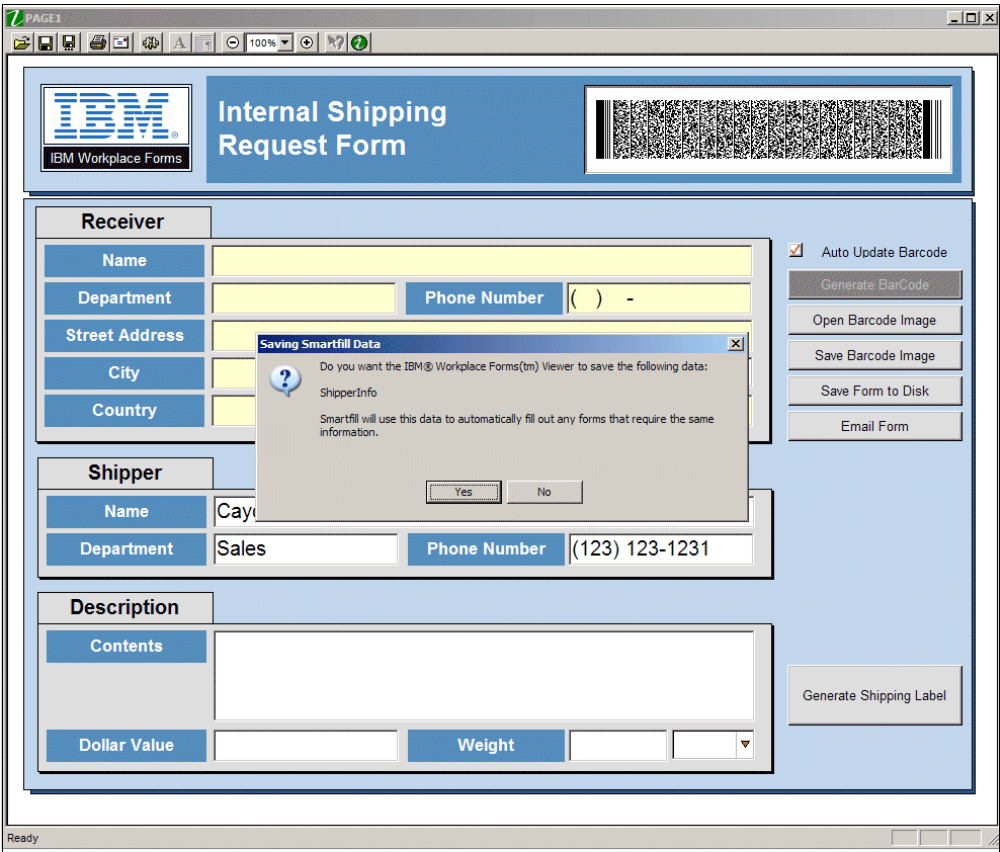


Figure 3-13 Smartfill prompt asking if user wishes to cache local data

Once Smartfill data has been cached locally, the next time that the corresponding form template is launched by that same user, the user will be presented with a dialog asking if they wish to load the data (Figure 3-14).

The screenshot shows a web browser window displaying the 'Internal Shipping Request Form' from IBM Workplace Forms. The form is divided into several sections: 'Receiver' (with fields for Name, Department, Street Address, City, and Country), 'Shipper' (with fields for Name and Department), and 'Description' (with fields for Contents, Dollar Value, and Weight). A 'Barcode' section is visible at the top right. A 'Loading Smartfill Data' dialog box is overlaid on the form, asking: 'Do you want the IBM® Workplace Forms(tm) Viewer to load the following data: ShipperInfo'. Below this, it states: 'Smartfill will use this data to automatically complete portions of the form.' The dialog has 'Yes' and 'No' buttons. On the right side of the form, there are buttons for 'Generate BarCode', 'Open Barcode Image', 'Save Barcode Image', 'Save Form to Disk', 'Email Form', and 'Generate Shipping Label'. The status bar at the bottom of the browser window shows 'Ready'.

Figure 3-14 Smartfill load data 'ShipperInfo' [Yes / No] prompt

In this case, **Yes** was selected, resulting in the presentation of the form shown in Figure 3-15. This figure shows that the Shipper section has been prepopulated with the locally cached Smartfill data. The user was *not* online at this time.

Figure 3-15 Resultant form template; the Shipper section has been prepopulated with data

Integration of an XML data instance with a Line-Of-Business System

Workplace Forms provide us with an excellent means of capturing and validating data for systems that require human input. Many systems provide access via well defined, XML interfaces, most often described by XML Schema. As touched upon previously, Workplace Forms provides us with an elegant way to provide complete, valid data to such systems.

Let us walk through the steps involved in performing this kind of straight-through-integration.

1. Create the form (User Interface, Input Validation Logic, Signatures) using the Workplace Forms Designer.
2. Obtain the XML Schema definition of the System to which we need to provide data.
3. Create a “prototypical” data instance — an instance that complies with this XML Schema.
4. Using the Workplace Forms Designer, embed this data instance within the form’s data model, providing it with a unique namespace and identifier.
5. Using the dialogs provided in Workplace Forms Designer, create “binds” between the elements within the data instance, the user interface (fields, combo-boxes etc.) and/or other data models as needed. The Workplace Forms Designer provides us with a visual (point-and-click) means to relate the UI and data model elements.
6. If desired, enclose the original schema to provide an additional level of validation of this data instance.
7. Save the form and deploy it to your users or into your Web / Portal / Domino application.

When data is entered into the form by end-users, or, by prepopulation of other data instances, the binds we have defined will automatically propagate these values into the data instance we have just defined. In terms of form processing and data integration, the high-level steps are:

1. Receive the form submission (typically HTTP Post, or SOAP/HTTP).
2. Validate digital signatures, if desired, to ensure that the form has not been tampered with.
3. Using the Workplace Forms API, call the `extractInstance()` method to obtain the desired data instance.
4. Connect to the remote system and provide the valid, populated data instance.

Integration of multiple XML data instances to separate systems

A natural extension of integration to a single system is integration to multiple, disparate systems. Given the fact that Workplace Forms can simultaneously support more than one data instance, and store different representations of the same data in each instance (different date formats for example: DD/MM/YYYY versus MM/DD/YY), one need only repeat the previously described steps for external XML Schema.

To make this more concrete, let us examine a specific example. The insurance industry in the United States of America has formed a group called ACORD that has defined a set of standard messages for insurance data and information processing systems. These messages are numbered and each has a different purpose. Let us take the example of a Life Insurance Policy application form. This form contains three data instances:

1. ACORD 111 compliant instance, used for prepopulating client data
2. ACORD 103 instance, used for integration to underwriting
3. XML Database compliant instance, used for reporting

Initially, the form is prepopulated by inserting a completed ACORD 111 instance. The data provided is propagated though to the form UI and the other data instances. As the end-users fill out the rest of the form data, this information will populate the rest of the required elements within the ACORD 103 and XML Database instance. Once data entry is complete, this form will be submitted to the server, signatures will be validated, and the data instances will be extracted. The ACORD 103 instance will be passed along to the underwriting system, and the XML database instance will be provided to the database.

It is important to note that quite often, the entire form document is also retained as a transaction record that can be audited at a later date if need be (Figure 3-16). This figure shows a high-level example of form data integration to multiple systems, with submission of the completed form.

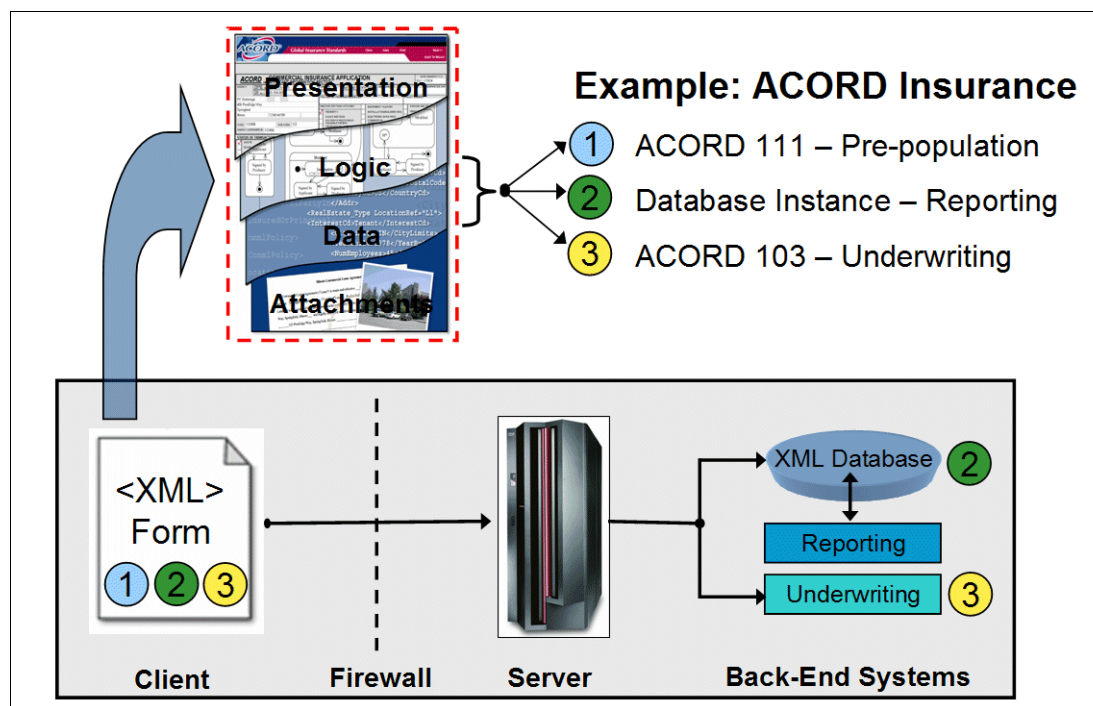


Figure 3-16 High-level example of form data integration to multiple systems

Real-time data integration

In certain situations, it is desirable to provide (or obtain data) in real-time. Client-side Web services provide us with the best means of doing so, however, alternate Apaches include round-trip submission of the form to the server, or the use of an IFX to handle socket or stream based communications with external systems.

Client-side Web service integration: The use of Web service calls directly from the form itself. This requires the Workplace Forms Viewer.

Server-side Web service integration: This is integration with external Web services in the Application Tier, typically within a Servlet or Portlet. These interactions can be performed either at the time of form template request, for prepopulation, part-way through the form filling process, based on an event (such as one pressing a button) within a form that triggers a round-trip submission, or on processing of a completed and submitted form.

3.3.3 Process integration

Workplace Forms provide us with a high degree of flexibility with regards to process integration. This results, in part, from the fact that Workplace Forms is a component technology that is designed to overlay onto existing customer infrastructure investments (Application or Portal Server, Workflow, Database, Content or Records Management Systems, etc.). Additionally, architects have a high-degree of freedom with regard to how much (or little) business logic is built into the form, and how much is managed externally.

For enterprise applications, the use of a process-oriented or workflow product is often essential. Examples include, but are not limited to:

- ▶ WebSphere Process Server
- ▶ WebSphere MQ Workflow
- ▶ WebSphere Portal Document Manager
- ▶ Content Manager Workflow

From a high-level perspective, there are several standard models for form interactions with process or workflow engines, as explained in the following sections.

Initiation of a task or workflow based on form submission or completion

In this case, data collected within a Workplace Form is used to initiate or kick-off a new process.

Form-based data collection as a human-task within a workflow

As part of a human task, data is collected within a form. On submission, the form data is processed, the task state is updated accordingly, and the process continues.

Calls to process server for real-time process interaction

In some situations, real-time business rule processing is important. One example may be the routing of a form for approval to an individual based on business logic, and role-based task capabilities, using data collected within the form as decision criterion. Web service calls could be made using either the client-side or server-side models that we discussed earlier.

3.3.4 Security context integration

Considered at a high-level, forms run within the security context of the user and host environment; for example, if a user launches a form off their desktop, then the form runs with all of the privileges and rights associated with that user's account on their operating system. If, however, one accesses a form through their Web browser, then there are several different security implications. First, if the Web site is secured and requires authentication, then the Workplace Forms Viewer runs within the context of the user's session with the server. Session time-out handling should be taken into account, for example, if one expects users to complete, then submit a lengthy form. Second, local privileges may be restricted based on the "sand-box" to which the Web browser is restricted.

3.3.5 Client-side device / hardware integration

The Workplace Forms Viewer provides us with an extension point that allows us to integrate with a broad range of hardware and system. This extension interface is called the *Function Call Interface* (FCI), and individual extensions or extension packages are often referred to as *Internet Form Extensions* (IFX).

The FCI Library is a collection of methods for developing custom-built functions that form developers may call from within Workplace Forms. By allowing one to create custom functions, it is possible extend the capabilities of forms without requiring an upgrade to either your forms software. IFX can be installed onto the end user's systems, or embedded within Workplace Forms. In addition to providing individual functions to Form Designers, one can also specify how and when the functions are used. For example, it is possible to specify that a function should run when a form opens, when it closes, and so on.

Using C or Java IFX, it is possible to interface with the local filesystem, device drivers, create socket based connections, and perform almost any action that you can implement in the programming language that you have selected. Examples of this could include Digital Signature Pad, Biometric Device, and GPS hardware integration.

For extensive, detailed information about the FCI, refer to the Workplace Forms API documentation (C and Java versions):

<http://www-128.ibm.com/developerworks/workplace/documentation/forms/>

3.4 Integration points summary

To sum up, here are the key integration points for Workplace forms:

1. The form data model
2. The Workplace Forms Viewer, running as a plug-in within a supported browser or Eclipse
3. The Workplace Form, rendered into HTML within a compliant browser
4. Web services on the client, or
5. The Workplace Forms Viewer FCI / IFX
6. The Workplace Forms Viewer via “Smartfill”

Although it may sound simple, the implications of the first point are significant. The Workplace Forms API gives us the ability to quickly and easily move data into or out of a form’s data model, and because of the support for arbitrary data instances, this gives us broad latitude at design-time. For example, we could “push” attachments into the form data model as part of prepopulation. In a Portal environment, we can prepopulate generated URI, used for form submission back to the Portlet. Alternately, the data model can enclose BPEL that contains the current state of the form or even logos and branding, depending on the role of the end user / customer who is accessing the form.

3.5 Partitioning of features and functionality

Creating stand-alone forms is one thing, but when architecting an overall eForm solution or process, we are faced with numerous design-time trade-offs. As with other projects, a needs assessment or requirements specification is essential to set parameters for partitioning of where specific features or functionality are addressed.

For example, using the stateful nature of the form in conjunction with the compute system, it is possible to construct a state-machine based on conditional logic, and to tie it to the data entered into the form. This gives us basic, in-form workflow capabilities to set end-points for routing via e-mail, submission, etc. Often, however, clients have existing investments in enterprise grade workflow products such as WebSphere Process Server or MQ Workflow, which give us much more robust, and centralized capabilities. The use of Workplace Forms in conjunction with these products is both expected and intended; from an architectural perspective, this gives us freedom with regard to how we partition functionality.

At one extreme, one can build forms that contain no business-logic at all, forms that instead rely on real-time interaction with a process-engine for all decisions and state information. At the other extreme, one can create forms that operate independently and contain complex state-machines, managing all aspects of their own behavior and routing. An example of a Workplace Form as a standalone application is the game of Blackjack shown in Figure 3-17.

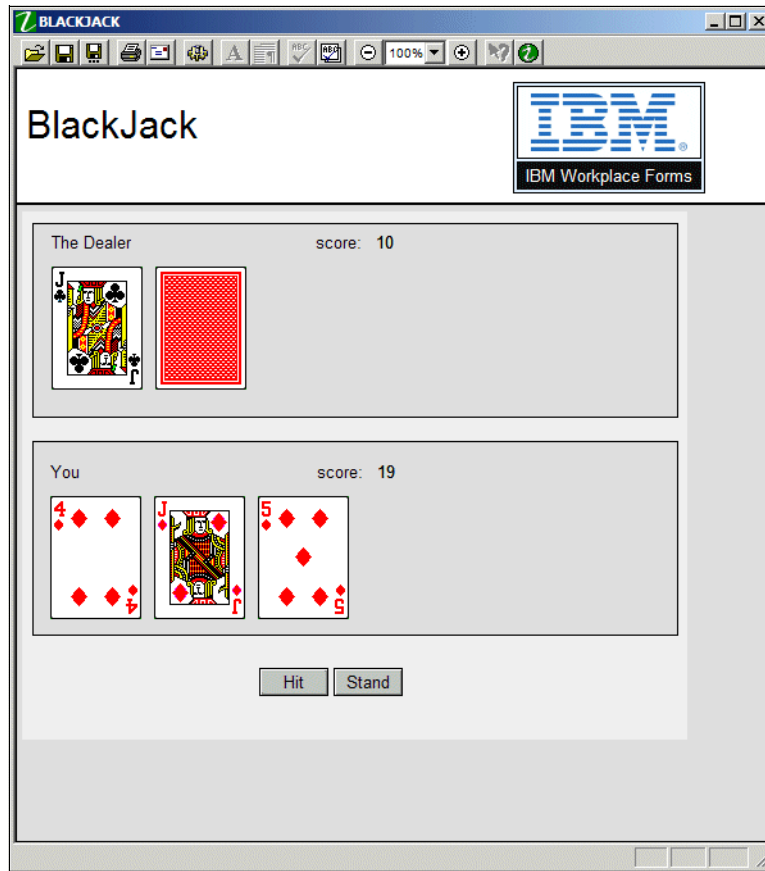


Figure 3-17 Example of an Application - Blackjack - implemented within a Workplace Form

When designing the example created for this redbook, we made the decision to show as much form functionality as possible. As such, we decided to manage all state information within the form itself. If such a solution were implemented in a production environment, we might wish to repartition the solution to move much of the business logic to the application tier (Process Server or Content Manager Workflow, for example).

3.6 Introduction to actual integration scenarios

Throughout the remainder of this book, we provide hands-on information about how to build a sample scenario application, beginning first with a standalone J2EE Workplace Forms application, then illustrating how to integrate this application with WebSphere Portal, Lotus Domino and IBM DB2 Content Manager.

The base scenario application is based on a Sales Quotation Approval Application, granting approval for price quotations and discounts for customers. The application has built-in business logic and workflow, determining which quotations and discounts must be granted either manager or director level approval based upon the price of the sale. This sample scenario application is described in much greater detail beginning in 4.1, "Introduction to the scenario used throughout this book" on page 54.



Building the base scenario: Stage 1

In this chapter we build a base scenario with a form depicting a sales quote process to be signed and approved by different roles defined by an approval workflow.

This building of this base scenario, together with the building steps provided in Chapter 5, “Building the base scenario: Stage 2” on page 145, serves as the foundation for the sample application used throughout the remainder of this book.

The base scenario is comprised of the following components:

- ▶ A form that collects the relevant data
- ▶ A servlet that processes the form
- ▶ Several JSPs to start the sales quote process and to view forms

Note: The code used for building this sample scenario application is available for download. For specific information about how to download the sample code, please refer to Appendix A, “Additional material” on page 333.

Note: All specific examples shown and used when building the sample scenario application are based on the codebase for IBM Workplace Forms Release 2.5.

4.1 Introduction to the scenario used throughout this book

For this redbook, we have created a base scenario application that serves as a foundation example throughout the book. The base scenario application is based on a Sales Quotation Approval Application, granting approval for price quotations and discounts for customers. The application has built-in business logic and workflow, determining which quotations and discounts must be granted either manager or director level approval based upon the price of the sale.

As we have discussed in the opening chapter, much of the business value from Workplace Forms is realized when converting paper-based forms into electronic forms. This conversion allows for a process to be formalized by capturing data through a structured front end (the electronic form) and incorporating formal business logic, business rules, workflow, and security. Ad hoc processes become formalized, the method to capture data becomes consistent, and opportunities for efficiently processing and leveraging this data are exposed.

4.1.1 Starting with a paper-based form

The foundation for the scenario is a paper-based form (illustrated in Figure 4-1), which served as the starting point for creating a Sales Quote approval.

Using the *paper-based* form, the process for obtaining a Sales Quote Approval would go as follows:

1. The sales representative would fill in the basic information, such as name, manager, customer name, account number, and information about the product, quantity, and proposed quotation pricing.
2. Once the information is filled in, the form would be faxed to a manager for review. The manager would review the information, verify the details of the customer account, and then sign their approval.
 - a. In many cases, numerous phone calls and e-mail exchanges between the manager and sales representative might be required to discuss details and confirm the proper quotation.
3. Finally, once the manager has approved a specific quotation, the signed form would be faxed back to the sales representative.
4. If the sales quotation led to an actual sale, the form would then be faxed to a customer service person in the sales department, who would then *manually* enter the information into the main sales and inventory system.

Sales Person				Customer			
First Name:				Company:			
Last Name:				Account Number:			
Personnel Number:				CRM Number:			
Email Address:				Department:			
Manager:				Contact Name:			
				Contact Position:			
				Email Address:			
				Phone Number:			

Products						
Item	Item number	# in stock	quantity	price	discount	total
Grand Total						

Attach fax and/or supporting documents

Originator's Position	Originated By:
Approval Level:	Authorized By:
Approval Level:	Authorized By:

Figure 4-1 Example of the original paper-based form used for generating a Sales Quote Approval

As you can imagine, the process for obtaining a sales quotation approval using this paper-based method is considered inefficient and burdensome. Responses and approvals from management could easily be delayed, and the business logic / rules applied for approving specific quotations are not always consistent. By converting this Sales Quotation Approval process from a *paper-based* form to an *electronic form-based* application with formal business logic, workflow, and back-end integration with other systems, the entire process can be made much more efficient.

4.1.2 From paper-based form to an electronic forms based application

In building the electronic forms based application, this demonstrates the value in converting from a paper-based form to an electronic forms based application. Furthermore, this chapter, together with Chapter 5, “Building the base scenario: Stage 2” on page 145 provides step by step instruction on how this application is built.

Description of the scenario

The base scenario depicts a sales quote process where a sales representative provides relevant data for sales quote to be approved by different roles. The sales quote is comprised of different data entities that are collected from the user by the form:

- ▶ Organizational data of the sales representative
- ▶ Customer data of the company inquiring the quote
- ▶ Order data with products and prices being quoted

In the basic scenario we create a form with wizard functionality to provide a guided interview for the user. We are not integrating with a back-end system or doing any prepopulation in the form yet. However, we have implemented a simple workflow of different roles that have to sign the form depending on the total amount of the sales quote. At the end, the user submits the form to a servlet that does further handling of the form such as:

- ▶ Controlling access to the form
- ▶ Saving the form to the filesystem in dedicated folders according to the workflow
- ▶ Processing the workflow by presenting a workbasket to the respective roles

Figure 4-2 provides a conceptual overview of the application from an end user perspective. The sales person logs into the system, and is guided through a quick series of forms to enter specific information about the customer and product. Much of the information is prepopulated based on the salesperson's log in credentials. Once the sales person has chosen the customer, the product, and the quantity, specific business rules are applied to determine if the quotation can be automatically approved, or if a manager or director needs to approve the quotation. Based on the required approvals, the form will be routed to the appropriate management approver via automated workflow.

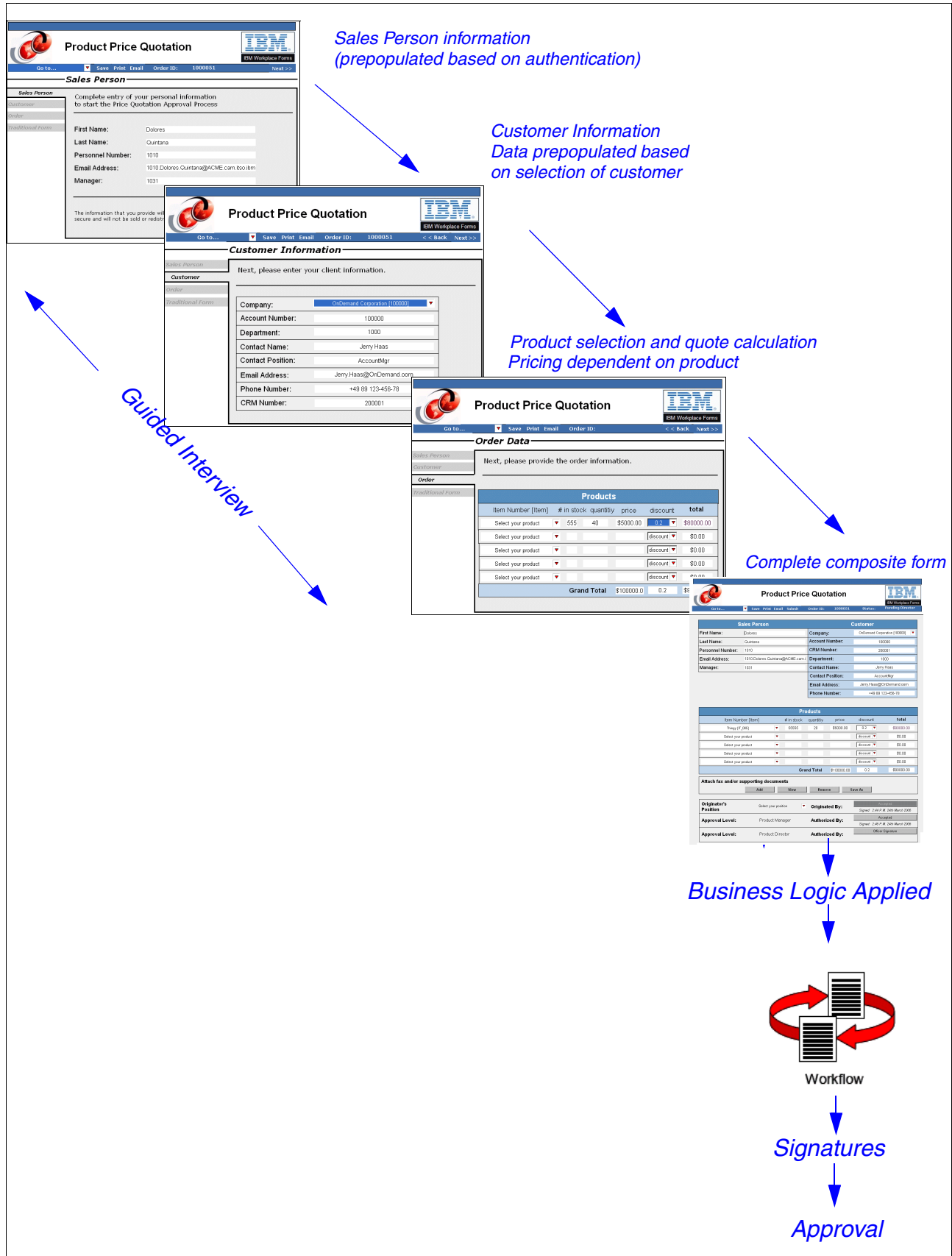


Figure 4-2 Overview of base scenario application

Figure 4-3 illustrates an overview of the workflow and business logic contained within the application.

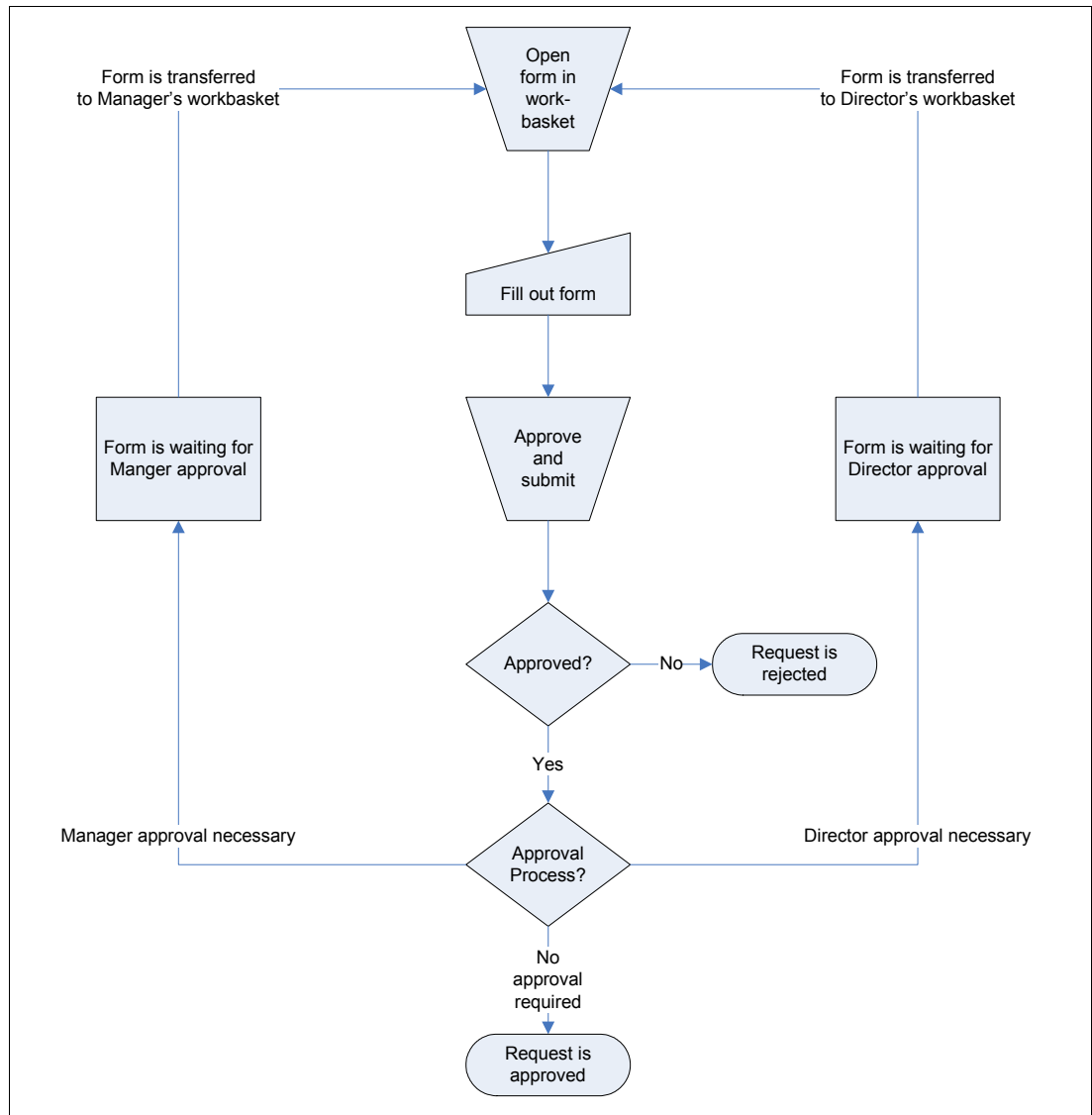


Figure 4-3 Overview of workflow for sample application

Within this example application, the following features and functionality of IBM Workplace Forms is incorporated:

- ▶ Approval for quotation and discounts for customers
- ▶ Prepopulation of data:
 - From servlet via Forms Server API
 - From Web services
- ▶ Business logic
- ▶ Workflow
- ▶ Signatures
- ▶ Submit to servlet
- ▶ Extract data and store form to DB2

Illustration of possible integration with other systems, including WebSphere Portal, Content Manager, and Domino is provided in subsequent chapters.

Figure 4-4 illustrates the sample application within the context of a three-tier architecture. While the basis of this scenario is built using J2EE, different integration techniques/ touchpoints with other products are examined in subsequent chapters.

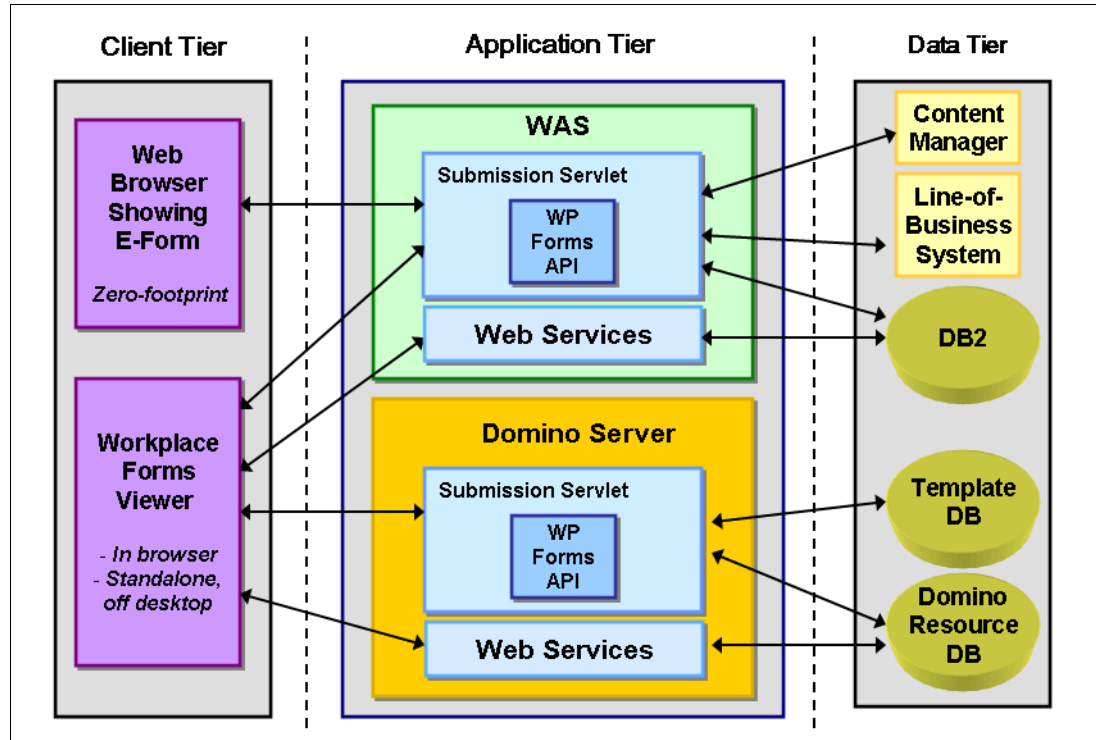


Figure 4-4 Illustrating the scope of the sample application within context of a three-tier architecture

4.1.3 Review of the specific forms: End user perspective

This section describes the key parts of the application from an end user perspective. Note that the application is role based, and the options available for each user will be different depending upon whether they are an Employee user (such as a Sales Person), a Manager, or a Director. Only Managers and Directors have the ability to make formal approvals.

Upon logging in, the user is presented with choices related to the task they need to accomplish. They can either create a new order for a quotation, or they can view their workbasket to view pending required approvals, or any other forms which have been approved or rejected. (See Figure 4-5.)

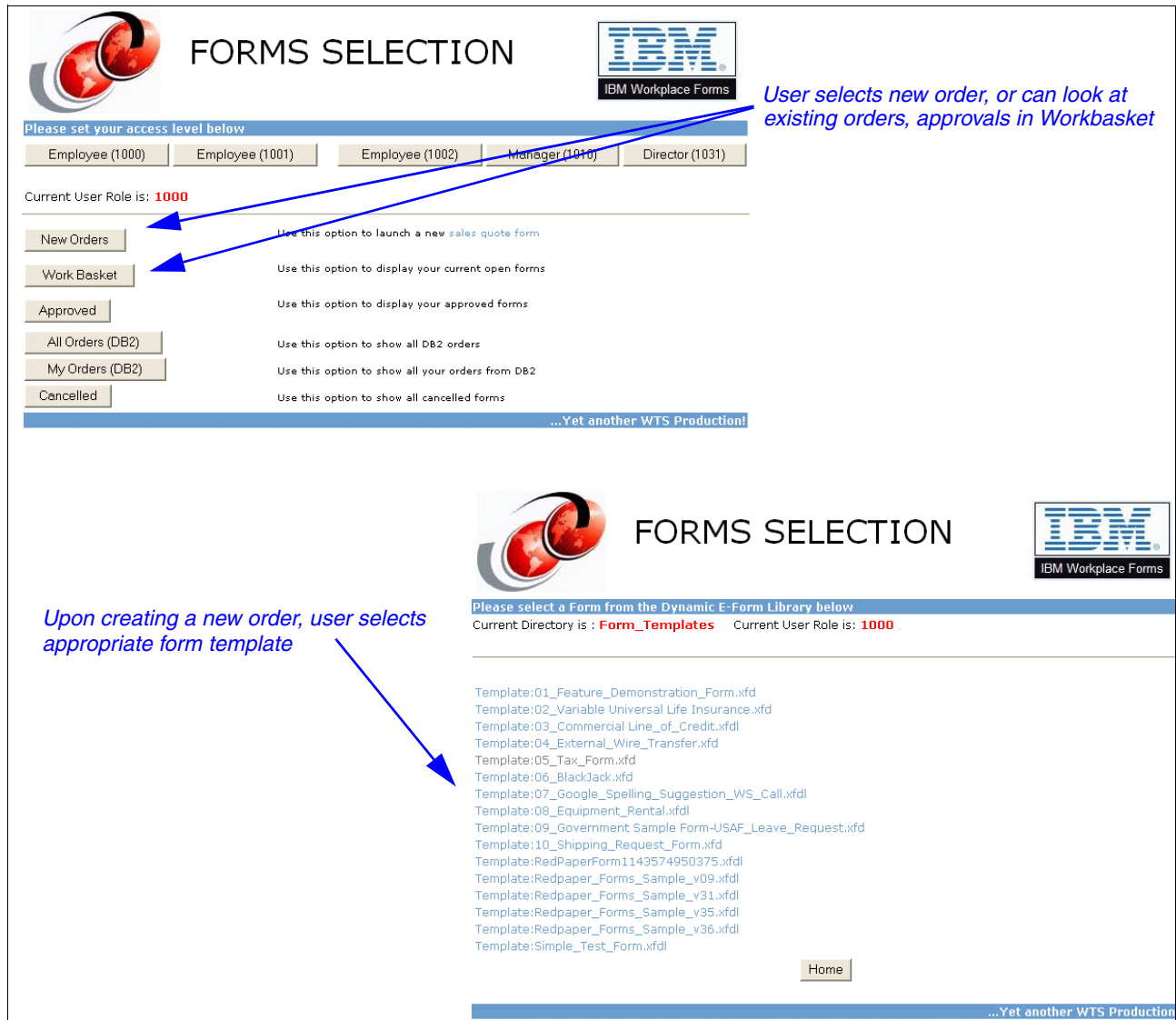



Figure 4-5 Logging in to create a new order


From within the Sales Quotation form template, the end user enters / validates data related to:

- Sales Person information
- Customer Information
- Product Information

Figure 4-6 illustrates the form component pertaining to the Sales Person data.

Note: Sales person data for this form is prepopulated based on user authentication credentials, using data from the corporate directory.


IBM Workplace Forms



Product Price Quotation

Go to... Save Print Email Order ID: 1000051 Next >>

Sales Person

Sales Person

Customer

Order

Traditional Form

Complete entry of your personal information to start the Price Quotation Approval Process

First Name:

Dolores

Last Name:

Quintana

Personnel Number:

1010

Email Address:

1010.Dolores.Quintana@ACME.cam.itso.ibm

Manager:

1031

The information that you provide will be kept confidential and secure and will not be sold or redistributed.

Navigating between the form components

Prepopulated based on user log-in credentials

Figure 4-6 Sales person data

When the users clicks the **Customer** component of the form (see Figure 4-7), they must select the customer name, and the remaining data fields will be populated automatically.

Product Price Quotation

Go to... Save Print Email Order ID: 1000051 < Back Next >>

Customer Information

Sales Person

Customer

Order

Traditional Form

Next, please enter your client information.

Company:	OnDemand Corporation [100000]
Account Number:	100000
Department:	1000
Contact Name:	Jerry Haas
Contact Position:	AccountMgr
Email Address:	Jerry.Haas@OnDemand.com
Phone Number:	+49 89 123-456-78
CRM Number:	200001

Once customer name is chosen, remaining data is populated via a Web service

Figure 4-7 Customer information - prepopulated via a Web service

Finally, the end user can enter information specific to the product. (See Figure 4-8). Total pricing is automatically calculated.

Once the form is completed, it can be submitted for approval.

The following business logic is applied for this scenario.

- ▶ \$0 → \$10,000: Pre-Approved
- ▶ \$10,001 → \$50,000: Manager Approval Required
- ▶ \$50,001 or higher: Director Approval Required

Product Price Quotation

IBM Workplace Forms

Go to... Save Print Email Order ID: < Back Next >

Order Data

Sales Person
Customer
Order
Traditional Form

Next, please provide the order information.

Products					
Item Number [Item]	# in stock	quantity	price	discount	total
Select your product ▼	555	40	\$5000.00	0.2 ▼	\$80000.00
Select your product ▼				discount ▼	\$0.00
Select your product ▼				discount ▼	\$0.00
Select your product ▼				discount ▼	\$0.00
Select your product ▼				discount ▼	\$0.00
Grand Total			\$100000.0	0.2	\$80000.00

Stock quantity and pricing populated via a Web service, based upon the specific product selection.

Figure 4-8 Entering specific data about the product

Upon submitting the form, it then routes via workflow to the appropriate Work Basket for further processing upon approval.

Throughout the remainder of this chapter, together with the steps outlined in Chapter 5, “Building the base scenario: Stage 2” on page 145, we provide you with step-by-step guidance on how to build this scenario application.

4.2 Overview of steps: Building Stage 1 of the base scenario

The diagram in Figure 4-9 is intended to provide an overview of the key steps involved to build the base scenario. This focuses on building the Form, the JSPs, and the Servlet.

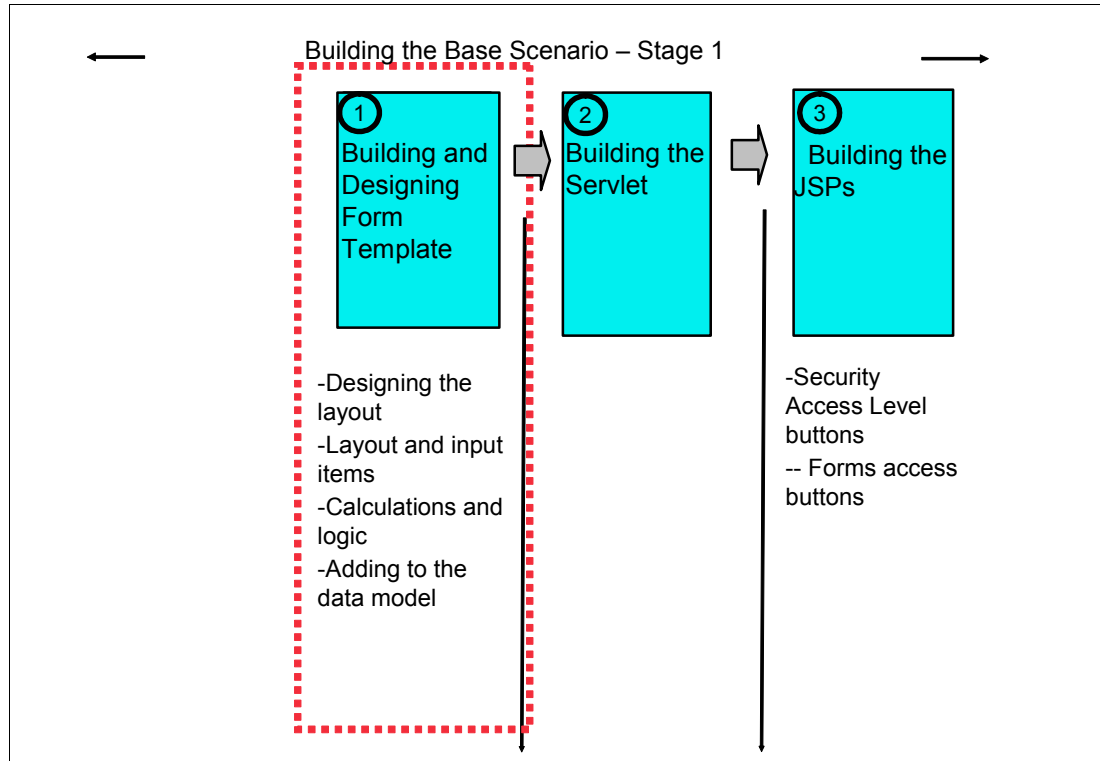


Figure 4-9 Overview of major steps involved in building the core base scenario application

4.3 Preparing to build the form template

The form we are designing is made up of a traditional form page and three wizard pages. The traditional form page holds all pieces of information as a summary page. The three wizard pages are intended to guide the user through the form in a step-by-step or interview fashion.

To create the form template, we will be using the Workplace Forms Designer. The Designer's workspace has a number of features that make it easy to create professional-looking forms quickly. In this section you will be looking at a few of them, and exploring the workspace.

4.3.1 Wizard pages versus traditional form pages

It can be difficult for users to enter complex data into crowded traditional forms, especially if they are unfamiliar with the rules governing the data entry. This task can be made quicker, easier, and more consistent with less chance of error if you provide an interface for the data entry. This interface is referred to as *wizard* pages. Where traditional eForms can be crowded and unstructured, with little room for user instruction, these pages can be used to lead the user through the data entry process in a logical and controlled manner. The user can enter the data in an order more suitable to their understanding, and have it displayed in a more business-appropriate order on the traditional form. Also, users only have to enter data once in fields that may be repeated throughout the traditional form.

The use of wizard pages within IBM Workplace Forms eForms is a common practice used to make the entry of data an easier process. Here are some benefits of using wizard pages:

- ▶ Wizard pages are generally smaller, fit to the screen, and easier to read, with no scrolling required. Page real estate is not an issue as it can be with some dense traditional forms.
- ▶ They allow the user to enter information once and have it displayed in multiple locations on the traditional form, making data entry more consistent and decreasing the risk of errors.
- ▶ They provide a flexibility that is not present in a traditional form — the ability to add labels to make the form easier to read, expand acronyms, and add code definitions that normally would not fit on a traditional form. If a user does not know the business rules needed to fill in a form, wizard pages can be used to guide them through the process in an intuitive manner.
- ▶ The appearance and formatting of fields on the traditional form can be changed based on user input in wizard pages. For example, fields on the traditional form might be active or not, based on the response entered on a wizard page. This would be especially useful when using Workplace Forms Webform Server, as dynamic updating of the form is only possible through either a page refresh or a page flip.
- ▶ Wizard pages allow for a “role based” display of fields — it can be used to walk a user through their portions of the form without displaying fields that do not affect them.
- ▶ This allows instructions to be displayed without overcrowding the traditional form.
- ▶ Users can enter multiple values on a wizard page and have the single calculated value based on that input displayed in the traditional form. This can save valuable real estate. (Note that this methodology restricts two-way data transfer between traditional form and wizard.)
- ▶ Users can select multiple values from a pop-up list on a wizard page and then have only codes representing those values concatenated in a single field on the traditional form. This not only saves real estate on the traditional form, but increases the clarity of the possible selections by providing more information than a code allows. (Note that this methodology restricts two-way data transfer between traditional form and wizard.)
- ▶ Users can also have multiple wizard page fields concatenated into a single field on the traditional form, again saving form real estate.

4.3.2 Considerations in advance: Best Practices for implementing traditional forms and wizard pages

A well-designed wizard page will make life easier for your users and help to ensure that forms are filled out with the correct data. The following sections take you through some Best Practices and recommendations to consider as you venture into building the forms.

Step 1: Design and implement the traditional form first. This will allow you to copy items from the traditional form to wizard pages after formatting the fields, preserving the format on the wizard pages. Here are some recommendations:

- ▶ Make sure the field size is the same on the wizard page as on the traditional form and that they can hold the same amount of data proportionally, especially if using different fonts between the traditional form and the wizard page.
- ▶ If the format is different between the traditional form item and the wizard page item, the user may not be able to enter the required amount of data, or data entered on the wizard page may be truncated or invalid on the traditional form.

Step 2: Design the template for your wizard pages.

A template typically consists of a page containing a sizing information, a background box (or two if you want a shading effect), a heading, and navigation buttons.

By using a template, you ensure that each wizard page has a consistent size, look, and feel. This also makes it much easier for you to create the individual wizard pages since the background and any buttons are done for you, and all you need to do in most cases is copy and paste the items from the traditional form onto your wizard pages.

Step 3: Make liberal use of help messages and have areas on the page to display help as opposed to only using tool tip help. As mentioned above, reference the same help message that the traditional form field references for ease of maintenance.

To do this, you can set up computes in labels to display the Help messages within labels rather than simply as mouse-over tool tips. The best way to implement this is to have a single label item on each wizard page that makes use of the page global focused item option. This is essentially a pointer to whatever item currently has focus. You can then de-reference that item's help item value, and display that text in the label.

Step 4: Use XML Data Model to bind items on wizard pages to fields on the traditional form.

The basics of how to use the XML Data Model are covered in the Advanced XFDL documentation (either training guide or document) section titled XML Data Model.

Using the XML model to make a wizard work is relatively easy. You need to create a data model instance containing one node for each piece of data that is represented in both the traditional form pages and the wizard. Then, when you set up the bindings (you can do this easily in the Designer) you simply create two bindings for each node in the instance — one connecting the data entered in the wizard page and one connecting the data entered in the traditional form page. When the data in one form element changes, the data in the other element linked to that node in the XML model will be updated as well — so there is no need for computes pushing and pulling data in every form item.

Step 5: Break wizard pages down into common or logical sections and have the user enter the data in a logical order. Fields may not appear in an intuitive manner on the traditional form and it may make more sense to the user on the wizard page if the order is changed. Also, it may be effective to group items on the same wizard page when they will be required on the traditional form under the same conditions. This wizard page could then be bypassed if the condition is not true and the data is not required.

Do not overcrowd wizard pages and make them too busy. This defeats the purpose.

Step 6: Decide whether your users will be allowed to exit the wizard pages before completing them and to save the data they have already entered. As an alternative, you may want to give the user the opportunity to bypass the wizard pages altogether.

If you do provide the opportunity to bypass the wizard pages, it is a good idea to let the user return to the wizard pages if they wish. If you have a lot of wizard pages, the user will find it helpful to be returned to the last wizard page they were on rather than to be taken right back to the beginning. In order to achieve this, you will need to set a global “return-to” option in your form that a “back to wizard” button can use. This global option should be set whenever a user clicks a button that takes him or her to the traditional form, and should contain the page reference of that button.

4.3.3 Possible starting points for creating a form

The Designer allows form designers to create forms in four ways:

- ▶ From scratch, starting with a blank, white page
- ▶ Using a sample form as a starting point, and customize
- ▶ From a template image of a paper form
- ▶ Using Texcel® FormBridge® to convert an existing format

Designing a form from scratch

If the form you are creating has no paper counterpart or appropriate template, you will need to start from scratch with a blank screen in the Designer. The layout tools provided – rulers, guides, and alignment tools – will help ensure that the items on your form are of a consistent size and line up correctly. One of the benefits of starting from scratch is that you can take full advantage of the different item types since you are not constrained by paper-based layout formats. Designing a form from scratch, however, takes a great deal of planning.

You should design a form from scratch if:

- ▶ The form will request company-specific information, but no paper counterpart exists.
- ▶ You decided to move away from a paper-based layout paradigm.
- ▶ Pre-existing applications will use the form and it needs to fit a certain format.

When you design a form from scratch, planning is essential. Forms can become large and complex, and if you do not have a firm plan to work from, you can easily find yourself facing a complete rearrangement near the end of the form design process.

Defining the purpose of the form

As you begin to plan, the first step is to consider the purpose of the form in detail. You may want to ask the following questions.

- ▶ What information will this form collect?
- ▶ In what order should users enter the information?
- ▶ Does the form need distinct sections? If so, how many?
- ▶ Should the form have more than one page? If so, what should go on each page?
- ▶ What security features are necessary?
- ▶ Should the form support digital signatures?
- ▶ To whom or to where will users submit the form?
- ▶ When the form is loaded, will items need to be updated by a database?
- ▶ Will any user input be submitted to a database?

Your answers to these questions will dictate how the form looks and acts in the end, and should help you build a list of all the items that you need to add to the form.

Determining your item type needs

Once you have a firm list of all the information your form needs to collect, you will also need to think about how the form will collect the information. Different types of information require different methods of collection. Your form can gather information in different ways by using different input items. Consider each piece of information displayed or collected by the form, and ask the following questions:

- ▶ What type of information will this item collect (numbers, text, “yes” or “no”, and so on)?
- ▶ Can I display necessary requests or instructions in the item?
- ▶ If any items ask the users to make a choice, should the form present the choices as mutually exclusive?
- ▶ If the form presents the choices as a list, should users have the option to type in other selections if none of the list choices fit (combo box)?

Make sure you take other factors into account as you go, such as the need to reformat data, especially if the information gathered by the form will be used by other applications.

Using a sample form

You might want to use an existing sample form that fits your requirements regarding layout, wizard, or functionality to a certain degree as a basis for your forms project. Numerous industry specific XFDL samples are shipped with the product and available on the Web.

Tip: You have to be careful working with copy and paste of forms objects from other XFDL projects. All the properties and calculations are being copied as well, so that you might run into errors regarding references that are outside of your project. Be sure to make a syntax check of your form in the Workplace Forms Designer (F key) and check your source code for undesirable name references.

Using scanned paper forms

Form Designers may use the Designer to create a form that already has a paper version but with no electronic version available. If this is the case, one way to produce the form with Designer is to use a template image. A template image consists of the scanned image of the paper form, saved in .bmp or .jpg format. This image is loaded into the Designer workspace, where the form designer can trace all the elements of a paper form, reproducing the exact layout, look, and feel of the paper form.

If you need to convert a pre-existing paper-only form to an eForm, and you want to maintain the look and feel of the paper version, you may use the Template Image feature in the Designer. This feature allows you to load a scanned image of a paper form into the Designer and place items on top of their paper counterparts, thereby recreating the layout of the original.

Use a template Image (scanned paper form) if:

- ▶ You need to duplicate the look and feel of a paper form.
- ▶ The form is very company-specific.
- ▶ There is no sample form available, and you want to use the layout of a paper form as a reference.
- ▶ The form is not available in any electronic format, exists only as a paper copy, and the scanned paper form gives poor results when processed with OCR (optical character recognition) software.

This process essentially transforms a paper form into an eForm. Because eForms use a different paradigm than paper, you can improve your transformation by following the tips provided here.

Before you start, you may want to think about the following questions:

- ▶ Do I want to use the same item types as the paper form? For instance, if a paper form has a section in which users are meant to check only one choice, you may want to use radio buttons rather than check boxes on the eForms.
- ▶ Does the form require a toolbar? If the form requires the user to save, print, or submit, you may want to use a toolbar – a non-scrolling region at the top of a form that contains buttons or information the user may want to use at any time. Toolbars save space because they are not printed and are not considered to be within the margins of the form.

- ▶ Do I want the form to specify user-input constraints for certain items and to provide help messages? These can save space also, since the “instructions” are hidden until the user tabs into the field or activates the help.
- ▶ Does the form require a signature for it to be considered completed? If so, include a signature button, allowing the user to digitally sign the form. Signing the form in this manner ensures that no changes are made in transit. You can create a signature button the same size as the signature space already provided on the form, but you may need to alter the instructions for signing slightly, to prompt the user to click the signature button.

Using Texcel FormBridge to convert an existing format

FormBridge, from Texcel Systems, converts forms from PDF, Microsoft Word, and many other sources to Workplace Forms. The converted forms are fully editable in the Designer, just as if they were created by hand. FormBridge preserves the appearance of the original forms and is the fastest way to convert existing forms into Workplace Forms.

Use FormBridge to create your form if:

- ▶ You want to duplicate the look and feel of the original form.
- ▶ You want to reuse information from the original form to create wizard pages.
- ▶ The form is available as a file from a software application.
- ▶ The form is available only on paper and gives good results when processed with OCR (optical character recognition) software.

FormBridge features:

- ▶ Offers accurate and precise conversion of the original form layout.
- ▶ Automatically generates fields by analyzing the page layout when converting non-fillable forms.
- ▶ Converts existing fields and field properties (e.g. name, data type, help text...) from fillable PDF and other applications such as JetForm and FormFlow.
- ▶ Batch translations can be performed with a single operation.

Advantages of using FormBridge:

- ▶ Re-creating a form by hand can take hours. FormBridge reduces the time to create a form from hours to minutes.
- ▶ FormBridge eliminates the mistakes and reduces the proofing and cleanup associated with manual forms creation and scanning. It is a true digital conversion — no scanning is required when converting from an electronic file.
- ▶ FormBridge is easy to use and no specialized forms design expertise is required. Many organizations have a shortage of skilled forms designers, but almost anyone can create forms with FormBridge.
- ▶ It is much easier to work with a FormBridge generated form in the Workplace Forms Designer than to start from scratch, especially for a new user.
- ▶ Conversion from paper-only forms can be done with FormBridge by first scanning and processing with OCR software.

For more information about FormBridge and to download a FormBridge demonstration, go to:

<http://www.texcel.com/ibm/>

4.4 Starting to build forms: Initial creation, design, and layout

Throughout these next sections, we discuss how to create the initial forms and establish the key design and layout of the forms. We approach the issue in terms of what needs to be done: first, for the traditional form pages, and second, for the subsequent wizard pages.

In our Sales Quote Approval sample we used a scanned image as a template for the traditional form page. You then add the items to the page and usually remove the template when you are done designing the traditional form.

Attention: For your help in building this sample application, we provide the image we use for the scanned form. This file `Form_Page_scanned_image.jpg` can be downloaded from the Additional Materials with the book. See Appendix A, “Additional material” on page 333 for detailed information about how to download the file.

See Appendix 4.4.3, “Creating a scanned template” on page 71 to review the approach for starting with a scanned form.

The wizard pages are then designed from scratch starting at a blank page. First, we defined the layout of the form before we added the items. You may want to sketch the layout on paper before you start building it in the Designer.

4.4.1 Designing the layout of the traditional form

The following sections discuss the first step in building your forms — namely, beginning with the correct layout.

Page setup

The first step is to create the necessary pages for the traditional form and wizard pages that you designed for your layout. Typically the wizard pages and the traditional pages have different page size and layouts that are consistent throughout the entire form.

4.4.2 Building the traditional form page

First we outline how to begin by creating the template for the traditional form page. Once this is built, we will follow on with information about how to build the wizard pages.

For the traditional form page, you want to use a preset page size of the paper format like Letter or A4 since this page will typically be used for printing the form. Figure 4-10 shows the scanned image that we created to be used as a template for the page.

Sales Person		Customer	
First Name:		Company:	
Last Name:		Account Number:	
Personnel Number:		CRM Number:	
Email Address:		Department:	
Manager:		Contact Name:	
		Contact Position:	
		Email Address:	
		Phone Number:	


Products						
Item	Item number	# in stock	quantitiy	price	discount	total
Grand Total						

Attach fax and/or supporting documents

Originator's Position	Originated By:
Approval Level:	Authorized By:
Approval Level:	Authorized By:

Figure 4-10 Scanned Image to be used as a template for the traditional form page

4.4.3 Creating a scanned template

- To create a scanned template for the traditional form page, follow these steps:
1. Scan your paper form, and save it in a compatible format (.bmp, .jpg, .ras, .png). It is important that the scanned image be scaled to fit the Designer form area (945 x 1220 pixels in our case).
 2. In the Designer, click **New**  and double-click anywhere on the blank form to open the **Page Properties** dialog box. Under **Template Image**, type the file path of the scanned image, or click **Browse** to locate it. Make sure the page size matches that of the original paper form.)

Tip: If the template is too obtrusive in the Designer, adjust the Image Fading bar to a setting that is comfortable.

3. Give the first page of the form a name (such as **Form Page 1**) by typing it in the *Page Label* field. Using the paper form name will make the form more recognizable in its new eForm format.
4. If desired, you may also change the background color of your form by selecting the **Appearance** tab and clicking on the panel under **On Screen**, next to **Background**. This background color will not be visible until you preview the form or remove the template image. You can also set the font to the most commonly used font on the form. You may also wish to include a toolbar at this point by selecting **Add Toolbar to Page**, but we will do this in 4.4.6, “Setting up the toolbar” on page 75.
5. Click **OK** to finish setting up the page. (See Figure 4-11.)

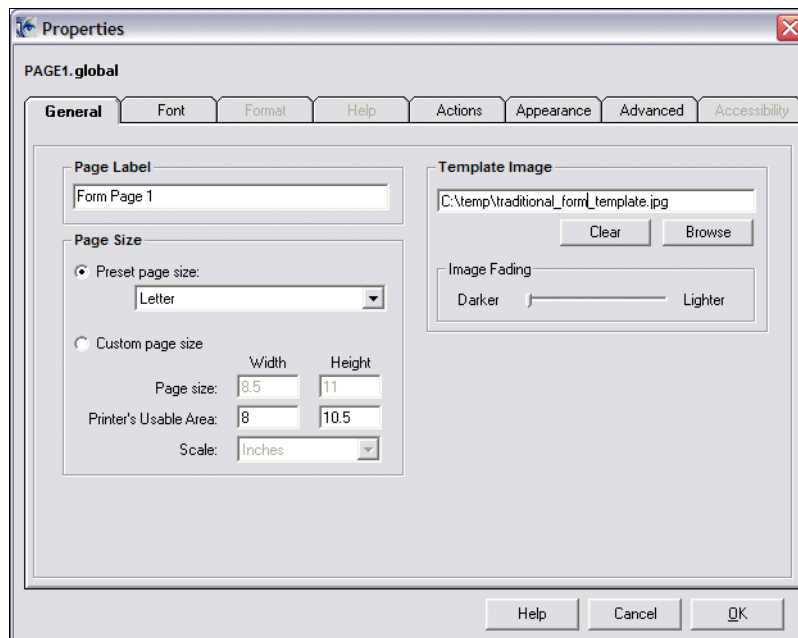


Figure 4-11 Page Properties of Traditional Form Page

Tip: It is a good idea to decide on the page name, background color, default font, and page size before you begin, because while you can change the page properties at any time, changing them while there are items already on the form can cause problems with sizing, appearance, and, if the page name is changed, references from other pages.

In Figure 4-12 you can see the scanned image as a background template on a traditional page in the Workplace Forms Designer. The template now serves as a pattern to precisely position our layout item and imitate the original paper form. After you have added the layout items to the form page, you should remove the image by clicking **Clear** on the **Properties** dialog box as seen in Figure 4-11.

Before we add the layout items onto the page, as described in 4.4.7, “Adding layout items” on page 78, we will create the wizard pages and set up a toolbar for the traditional form page.

The screenshot displays the IBM Workplace Forms Designer interface. The main form area contains the following sections:

Sales Person		Customer	
First Name:		Company:	
Last Name:		Account Number:	
Personnel Number:		CRM Number:	
Email Address:		Department:	
Manager:		Contact Name:	
		Contact Position:	
		Email Address:	
		Phone Number:	

Products						
Item	Item number	# in stock	quantity	price	discount	total

Grand Total	

Attach fax and/or supporting documents

Originator's Position	Originated By:
Approval Level:	Authorized By:
Approval Level:	Authorized By:

The status bar at the bottom shows: Item: PAGE1.global, Type: Page Global.

Figure 4-12 Traditional form page with scanned Image Template inserted

4.4.4 Creating the layout for the wizard pages

A wizard page typically gets a resolution of 600X800 pixels. For our form we will create three wizard pages to guide the user through the following information areas:

- ▶ Sales representative
- ▶ Customer information
- ▶ Order information

Preview of the wizard page to be built


Figure 4-13 illustrates in advance what one of the wizard pages you are building will look like. This is the wizard page for the form component pertaining to the Sales representative data.

When the users click on the **Customer** component of the form (see Figure 4-13), they must select the customer name, and the remaining data fields will be populated automatically.

Note: Sales person data for this form is prepopulated based on user authentication credentials, using data from the corporate directory.

Figure 4-13 Sales representative data

4.4.5 Steps to build the wizard pages

In our example we insert additional form pages by clicking  on the toolbar until we have the correct number of pages for the form. We will be adding three additional pages and will configure each of them as shown in the Page Properties (Figure 4-14).

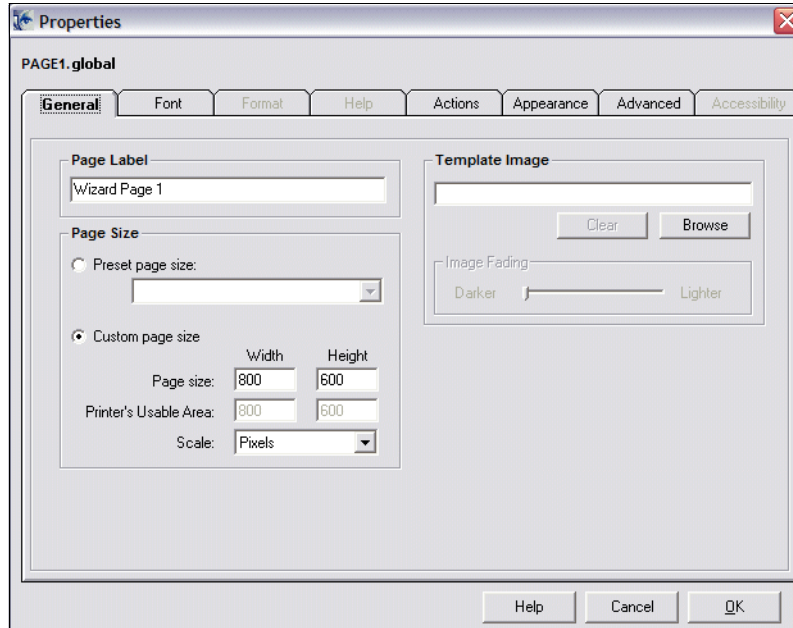


Figure 4-14 Page Properties of Wizard Page 1

Page globals specify settings (such as Next and Save Format) and characteristics (like bgcolor) for the page within which they appear. Page globals appear within a global item at the top of each page definition, and apply to the whole page. They can be overridden by option settings within items.

4.4.6 Setting up the toolbar

A toolbar is a separate and fixed area at the top of the page. It functions much like a toolbar in a word processing application. Items placed in the toolbar are always visible at the top of the form, no matter what portion of the page they are viewing. The toolbar is visible no matter what portion of the page body is visible. However, if the toolbar is larger than half the form window, it is necessary to scroll to see everything it contains. Since the toolbar on a page will not be printed, we only used it on the traditional form page.

Normally, your paging controls and company logos are located on the toolbar. If you do not have paging controls set up, you will not be able to access all the pages when you test the form in the Viewer. In our example we created following controls:

- ▶ Page navigation pop-up
- ▶ Save form to filesystem
- ▶ Print form
- ▶ E-mail form
- ▶ Show order ID
- ▶ Page back
- ▶ Page next

To insert a toolbar in your page, select the **Toolbar** option from the Insert Menu as shown in Figure 4-15.

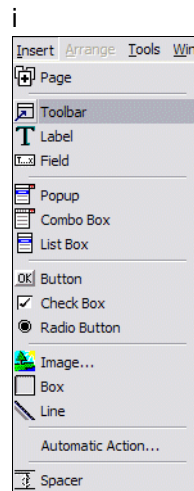


Figure 4-15 Insert Toolbar menu

Figure 4-16 shows the inserted toolbar on the traditional form page. We already added some basic layout items like a heading, logos, and design bars.

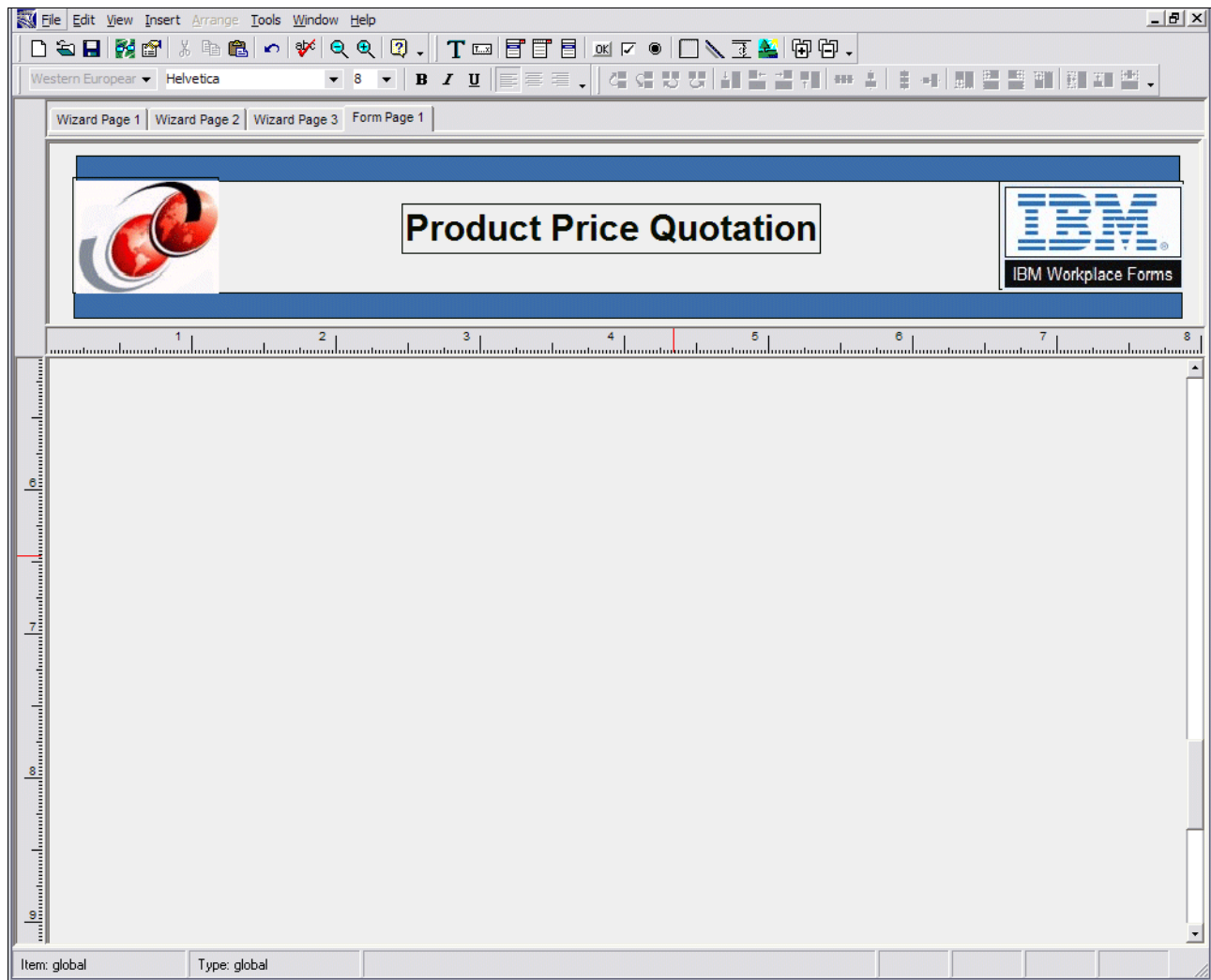


Figure 4-16 Traditional form page with toolbar

Tip: If you want to use the toolbar on several pages of the form, it is best to have the same toolbar on each page. The easiest way to accomplish this is as follows:

1. Lay out the toolbar on the first page.
2. Select every item on the toolbar.
3. Copy and paste the items onto the toolbars of the subsequent pages.

4.4.7 Adding layout items

First, add the lines and box panels you want to use, then add the labels for each section. Once all the labels are in place, preview the form to make sure it looks correct in the Viewer.

Use the Form View to add items to your form. This view allows you to place, resize, cut, copy and paste items here. It only shows one page of your form at a time, but additional pages are accessed with tabs at the top of the form. Form View offers item layout and arrangement tools, including zoom functionality.

The Insert toolbar displays all of the user input and layout items, allowing you to select and place items with your mouse. You can place the following form elements with this toolbar. (See Figure 4-17.)

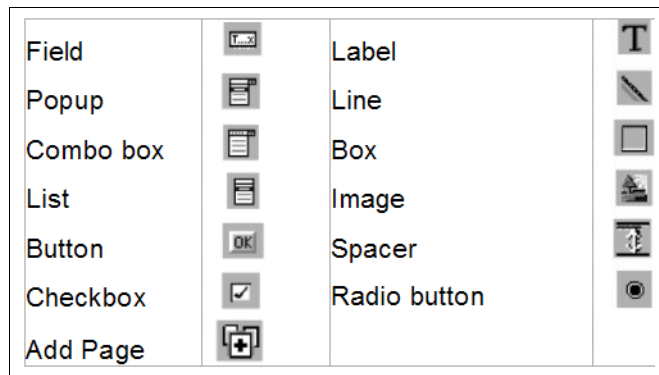


Figure 4-17 Toolbar items in the Workplace Forms Designer

Input items, such as fields, checkboxes, and combo boxes, allow users to enter data. Layout items, such as lines, labels, and boxes, are used to provide visual effects in forms.

Item properties

Item properties are characteristics, like color and font style. When you add an item to a form, it has certain default properties. For example, a button's default color is gray.

You can change these properties and give items new properties via the Properties dialog box. The Properties dialog box has several tabs, which we describe here:

- General Properties:

Sets general characteristics for items. The contents of this tab vary, depending on the type of item you are configuring. Typically, it includes any text the item displays, the size setting, toggles for making the item active or inactive, and any specialized information specific to the type of item you have selected.

- Font Properties:

Sets the style of the text displayed by an item, such as font type, size, and weight.

- Format Properties:

Sets how users input information into your form, and how that information is displayed. You can determine data types, set constraints on user input, and set predefined formats in this tab.

- Help Properties:

Lets you create context-sensitive help for each visible item on a page. When users turn on Help Mode, help messages appear when the mouse pointer passes over an item with a help message.

- ▶ **Choice Properties:**
Allows you to add, remove, and edit the choices (or cells) that appear in pop-ups, comboboxes, and lists.
- ▶ **Appearance Properties:**
Allows you to modify an item's appearance, such as color, visibility, and borders. If you want an item to display an image, you can use the Image section to insert an image into buttons and labels.
- ▶ **Advanced Properties:**
Allows you to add custom options, usually using the 'custom' namespace. These options may contain computes.
- ▶ **Accessibility Properties**
Allows you to add accessibility messages for users with visual disabilities.

Layout tools

The Designer interface provides a number of tools to help you in precisely designing items:

- ▶ **Rulers:**
Horizontal and vertical rulers that measure in pixels or inches. You can choose whichever units you prefer.
- ▶ **Guides:**
Horizontal and vertical lines that you can place throughout form to ensure that items line up. The top or left side of items placed guide automatically "snap" into alignment along the guide.
- ▶ **Grid:**
A grid of uniformly placed dots that are super-imposed on the form to assist you in item placement. You can adjust the spacing of the grid to either pixels or inches. Additionally, the snap-to-grid feature automatically aligns the top left corner of items to the nearest point on the grid.

Absolute and relative positioning

You can use both absolute and relative positioning to set an item's position on a page. This position is recorded in the *itemlocation* option.

Absolute positioning sets the top left corner of an item at absolute x and y coordinates, measured from the top left corner of the form window (Example 4-1).

Example 4-1 XFDL code for absolute positioning of an item

```
<itemlocation>
  <ae>
    <ae>absolute</ae>
    <ae>144</ae>
    <ae>67</ae>
  </ae>
</itemlocation>
```

Relative positioning defines an item's position relative to an anchor item (Example 4-2).

Example 4-2 XFDL code for relative positioning of an item

```
<itemlocation>
  <ae>
    <ae>before</ae>
    <ae>FIELD1</ae>
  </ae>
</itemlocation>
```

Regardless of whether you choose to use absolute or relative positioning, you can use the Arrange tools to assist you in placing items. The Arrange tools include these:

- ▶ **Reposition:**
Allows you to precisely place an item above, below, or to the right or left of another item.
- ▶ **Align:**
Aligns an edge or center of an item with the edge or center of another item.
- ▶ **Resize:**
Allows you to move an item's border so that it lines up with another item's edge or center.

If you use absolute positioning, the Arrange tools place items using x and y coordinates. If you use relative positioning, the tools define item positions relative to the anchor item.

Building the layout

To build the layout of the form follow these basic steps:

- ▶ **Place Lines and Boxes:**
Begin by placing all the necessary lines and boxes on your form. Start at the top and place each horizontal or vertical line as you work down the paper form.
- ▶ **Place Labels:**
Begin at the top of the page and work through the paper form placing each label as you come to it.
- ▶ **Preview:**
Once you have placed all lines, boxes and labels on your form, preview and print the form to check that everything lines up correctly and prints properly. Repeat the preview, correction, and modification process until all you have resolved any problems with the form.

4.4.8 Reviewing the layout for the traditional form page

We have divided the traditional form page into five parts:

- ▶ Sales person section
- ▶ Customer section
- ▶ Products section
- ▶ File attachments section
- ▶ Signature and approval section

For each of these sections, we created different boxes that produced a table layout for the labels and fields to be positioned on. You can use different background colors for the boxes to mark up headings and sections of your page as shown in Figure 4-18.

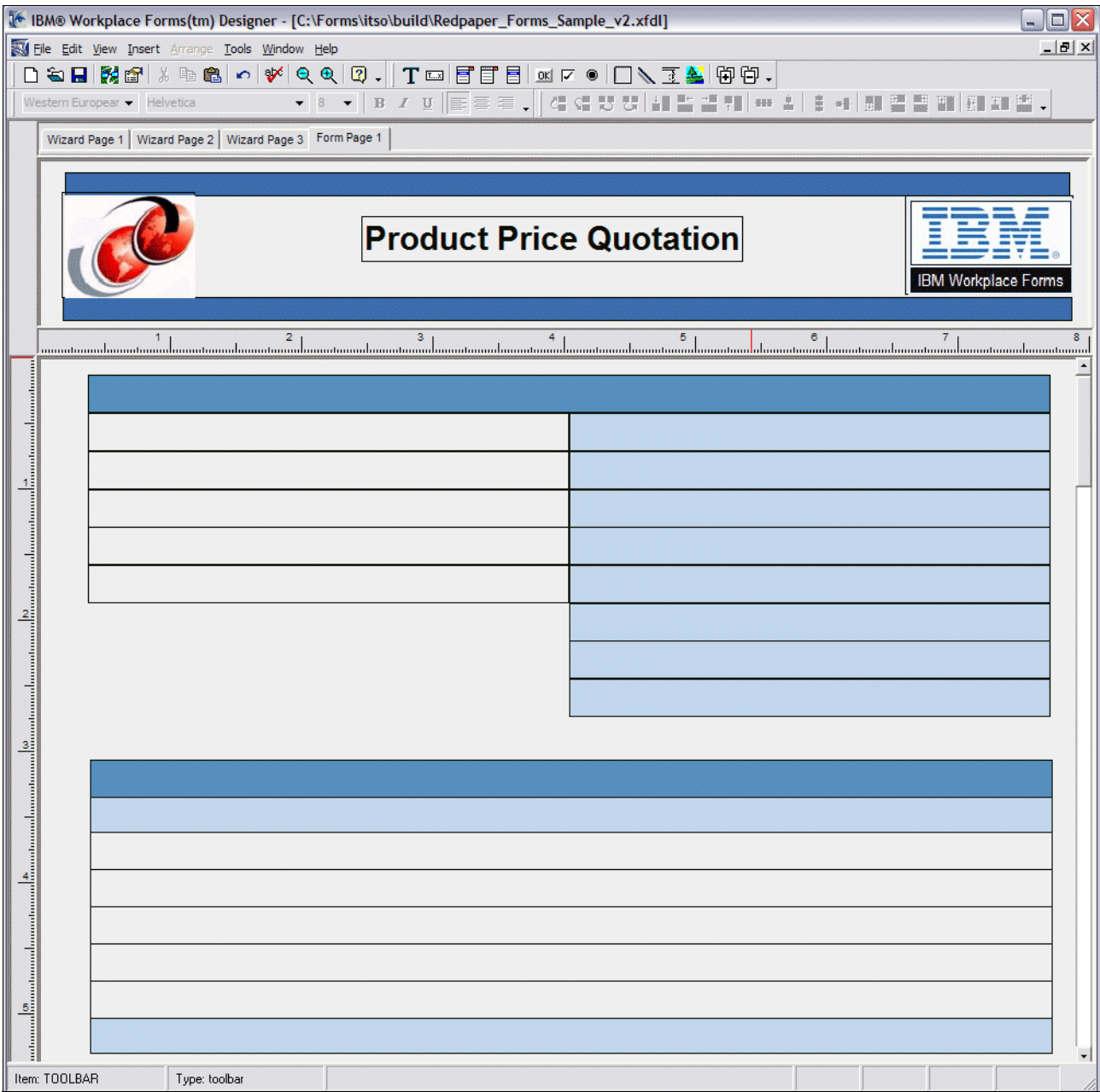


Figure 4-18 Basic layout of the traditional form page

You can configure the background colors in the Appearance section of the item properties by clicking into the background color box as shown in Figure 4-19.

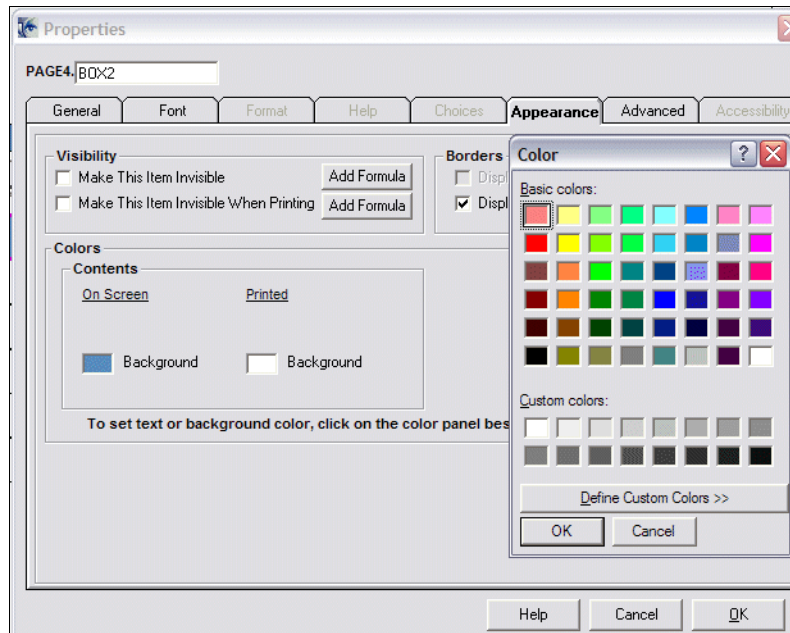


Figure 4-19 Background color selection of box items

4.4.9 Reviewing the layout for the wizard page

The layout of the wizard pages is usually more distinct and graphical than the design of the traditional form pages. We used boxes and lines to also create a navigation through the wizard pages of the form and to highlight the current position within the guided interview process. The layout we used is illustrated in Figure 4-20.

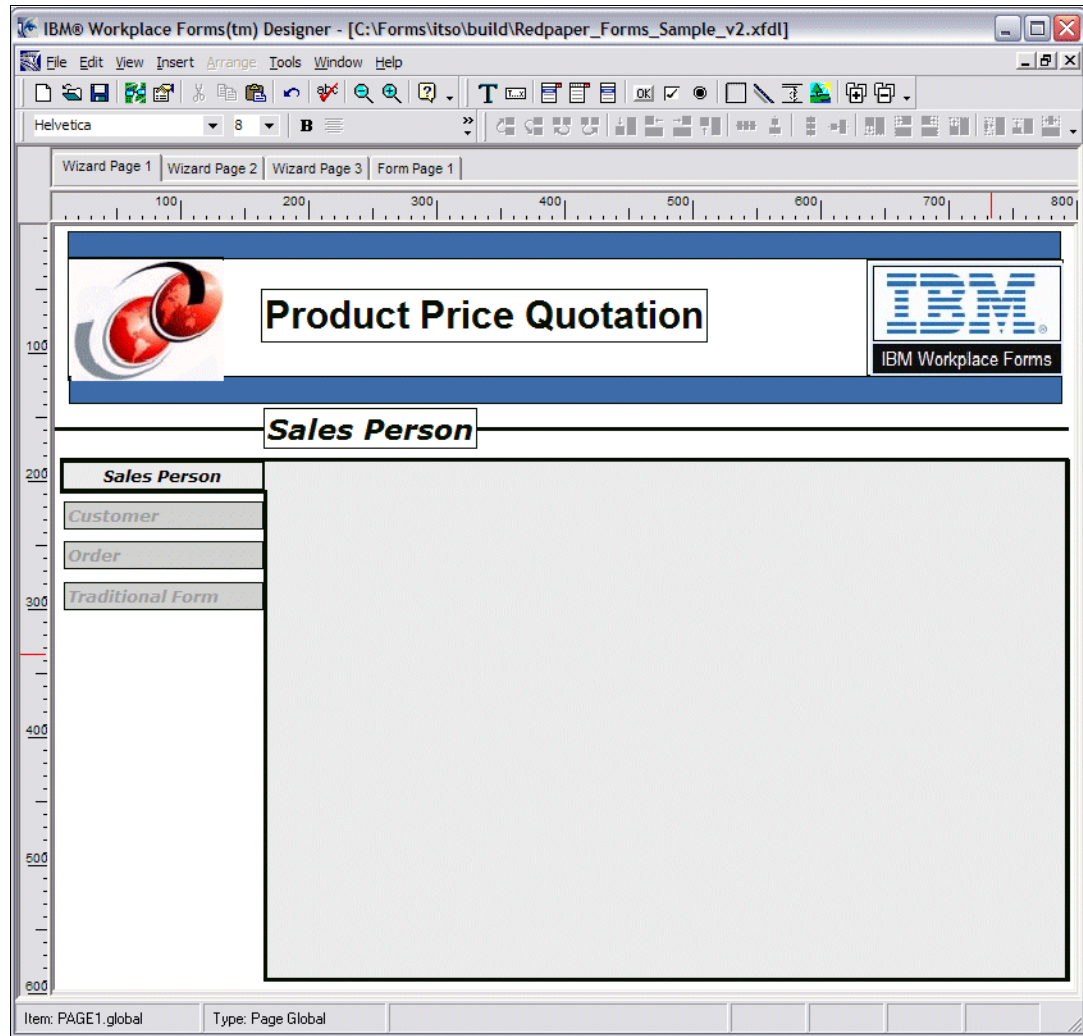


Figure 4-20 Basic Layout of wizard pages

4.5 Adding input items

In the next step we add the input items such as fields, pop-ups, and labels to our form pages. You may find necessary to modify your layout slightly if some of the input items will not fit.

Again, we follow the model of focusing on what needs to be done for the traditional form, as well as the subsequent wizard pages.

If you place each item in the order in which users should work through the form, you will not have to adjust the tab order of the items; the default tabbing should move through the form properly. Do not add logic or formatting to the form at this stage. Applying formatting and logic will be done in the next stage.

4.5.1 Adding input items to the traditional form

First you should create the input items on the traditional form page. We used mainly fields and some pop-up boxes to collect and display the necessary data for the Product Quotation Form.

IBM® Workplace Forms(tm) Designer - [C:\Forms\itso\stage2\Redpaper_Forms_Sample_v29.xfdl]

File Edit View Insert Arrange Tools Window Help

Western European Helvetica 8 B I U

Wizard Page 1 Wizard Page 2 Wizard Page 3 Form Page 1

Product Price Quotation

IBM Workplace Forms

Go to... Save Print Email Submit Order ID: Status: <Formula>

Sales Person		Customer	
First Name:		Company:	Select your customer ▼
Last Name:		Account Number:	
Personnel Number:		CRM Number:	
Email Address:		Department:	
Manager:		Contact Name:	
		Contact Position:	
		Email Address:	
		Phone Number:	

Products					
Item Number [Item]	# in Stock	Quantity	Price	Discount	Total
Select your product ▼				discount ▼	<Formula>
Select your product ▼				discount ▼	<Formula>
Select your product ▼				discount ▼	<Formula>
Select your product ▼				discount ▼	<Formula>
Select your product ▼				discount ▼	<Formula>
Grand Total			<Formula>	<Formula>	<Formula>

Item: global Type: global

Figure 4-21 Input Item on traditional form

The Workplace Forms Viewer can perform real-time consistency checks and formatting of the data entered by the user while the fields in the form are being filled. This assures that the data entered is always in correct format for its further purpose in a back-end system or application. You can configure the following format properties for the data contained in the fields. See Table 4-1.

Table 4-1 Properties for formatting input data

Format	Description
Data Type	Specifies what type of input to accept: ASCII Text (standard characters), Integer (whole numbers), Dollar, Float (floating point number), Full Date (year, month, and day), Year, Month, Day, Day of Month, Day of Week, or Time.
Text Format	Specifies how to reformat text that the user types in. Choose from lower case, UPPER CASE or Title Case. Applies only to data types ASCII, Full Date, Year, Month, Day, Day of Month, and Day of Week.
Numeric Format	Specifies how to reformat Integer, Float or Dollar data. Comma delimit means that for numbers greater than 999, the thousands are separated by a comma (3,689,995). Space delimit means that no comma is to be used, only a space (3 689 995).
Numeric or Dollar	Specifies how to reformat data of types Integer, Dollar or Float. Bracket negative means that negative numbers will be denoted with brackets around rather than a negative sign. For instance, -2 would be (2). Add Dollar Sign means that if the item is of Dollar format, a dollar sign will be added if the user does not type one in.
Date Format	Specifies how to display dates. The user can enter a date in any form, and the Viewer will reformat it to a consistent display. Possible formats are Short Format, Numeric, Long Format and Abbreviated.
Required Status	Determines whether the user must fill in the item. You can set this property to be determined by user actions by clicking Formula.
Constraints on User Input	The user input can be required to match in any or all of the following manners: <ul style="list-style-type: none"> ► Case Sensitive - The case in the input must match the template or format exactly. ► Range - Input must fall within the alphabetical or numeric range you specify. ► Length - Input must fall within the length you specify. ► Template - Choose a template from the list, or type your own.
Help Message	Message to display if the user types an incorrect entry, or if the user's entry becomes invalid because of something else added to the form.

As an example, we formatted the price and amount with the following properties as shown in Figure 4-22.

- Set the *Data Type* to **Dollar**.
- Set the *Numeric Format* to **Comma delimit**.
- In the *Constraints on Input* section, set the *Length* to **1 to 10**.

If you do not find an appropriate constraint for you purpose you can define an individual template in the Constraints on Input section to match your requirements on input formatting.

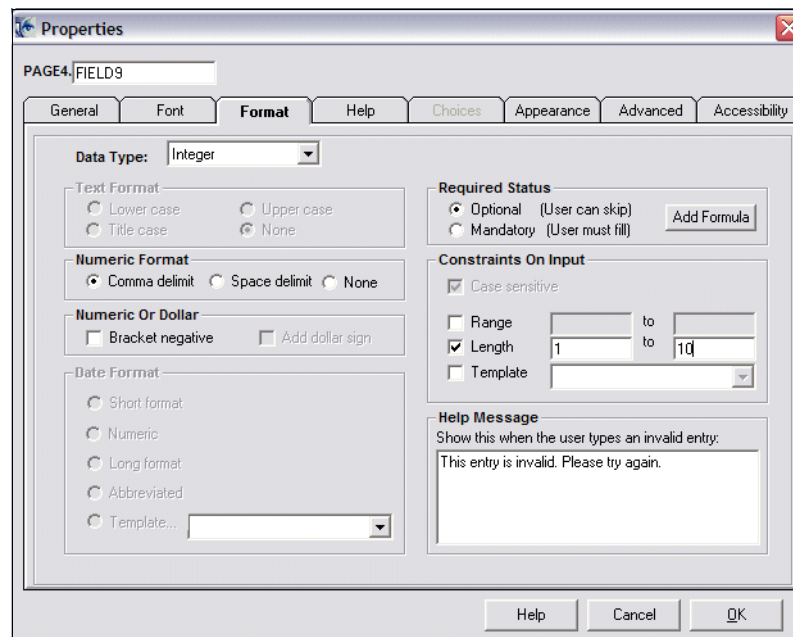


Figure 4-22 Field Format Properties Dialog

4.5.2 Adding input items to the wizard pages

The purpose of the wizard pages is to perform a guided interview for the user that makes it a lot easier and more intuitive to fill out a complex form. The individual wizard pages break up the traditional form page in meaningful parts of the information for the user to fill in. A good approach in adding the input items to these pages is to copy and paste the respective fields from the traditional form to the wizard pages, since this will also copy your computes along with the formats. You may then reformat the fields and layout items to fit your needs. (See Figure 4-23.)

Figure 4-23 Input Items on wizard page

4.6 Applying formatting and logic

Once you have all the visible items on the form, you can apply formatting, calculations and logic.

Unlike paper forms, XFDL forms can perform sophisticated checks and calculations while the user fills out the form. This can prevent mistakes right at the source and provide a high level of comfort while filling out the form.

4.6.1 Creating a field calculation

In this section we create a field calculation.

Calculation by decision

In our Sales Quote example we want to add a multiplication calculation to the amount field of the item based on the quantity and item the user selected from the product list. To accomplish this, we will use the formula wizard in the Workplace Forms designer by following these steps:

1. Open the Properties window for the first field (FIELD12) in the Total column of the Product section of the traditional form.
2. Click the **Add Formula** button in the *Field Contents* section on the *General* tab as seen in Figure 4-24.

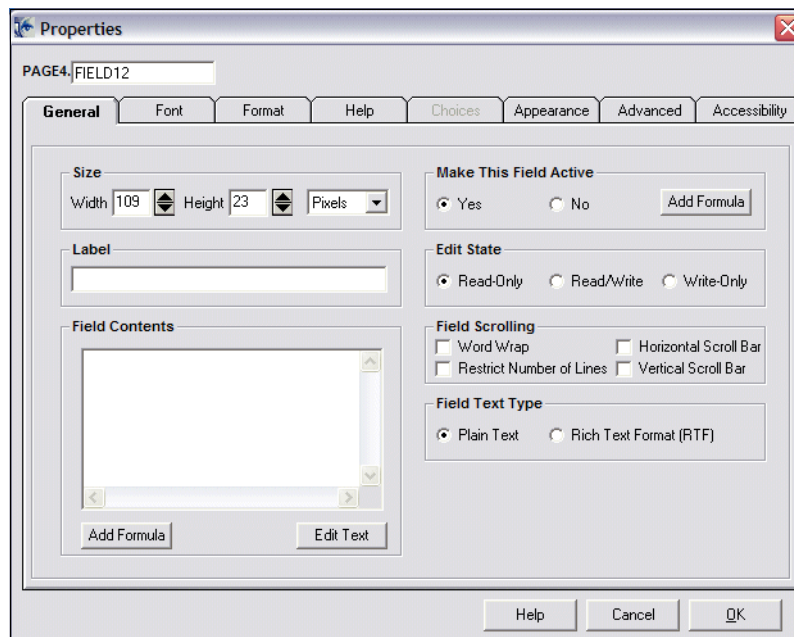


Figure 4-24 Properties box of the Total field

3. Select **set by a calculation of two values** from the drop-down.
4. In the Formula section, as seen in Figure 4-25:
 - a. Select **value of** from the first drop-down; a hand icon will appear, click it.
 - b. On the form page that is now visible, choose the first field in the quantity column.
 - c. Select **multiplied by** in the middle drop-down list.
 - d. Select **value of** from the second drop-down; a hand icon will appear, click it.
 - e. On the form page that is now visible, choose the corresponding price field.
 - f. Exit the Calculation window by clicking **OK**, then exit the Properties window by clicking **OK**.

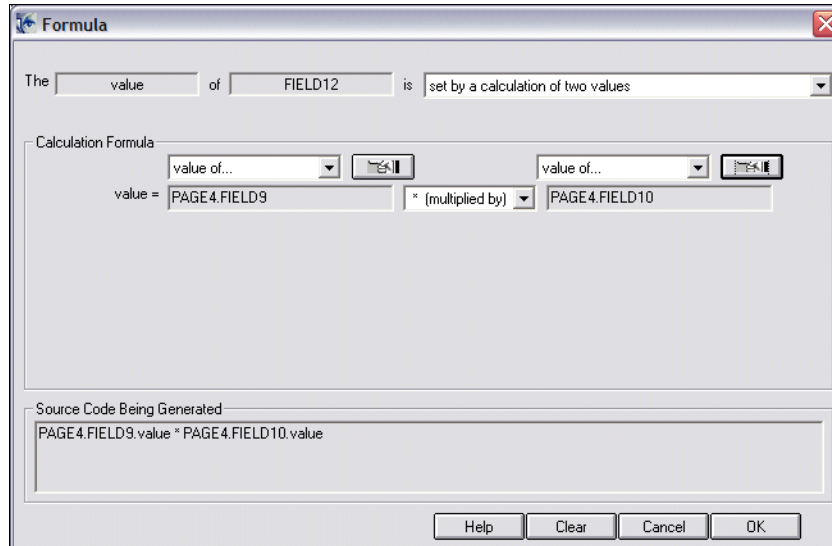


Figure 4-25 Formula wizard of Field Contents property of the Total field

Sum calculation

As a second example, we want to create a sum field that calculates the total amount of the products in the sales quote to be approved. To accomplish we follow these steps:

1. Open the Properties window for the Order Total field on the lower right end of the Products section.
2. Click the **Add Formula** button in the Field Contents section on the General tab. (See Figure 4-26.)

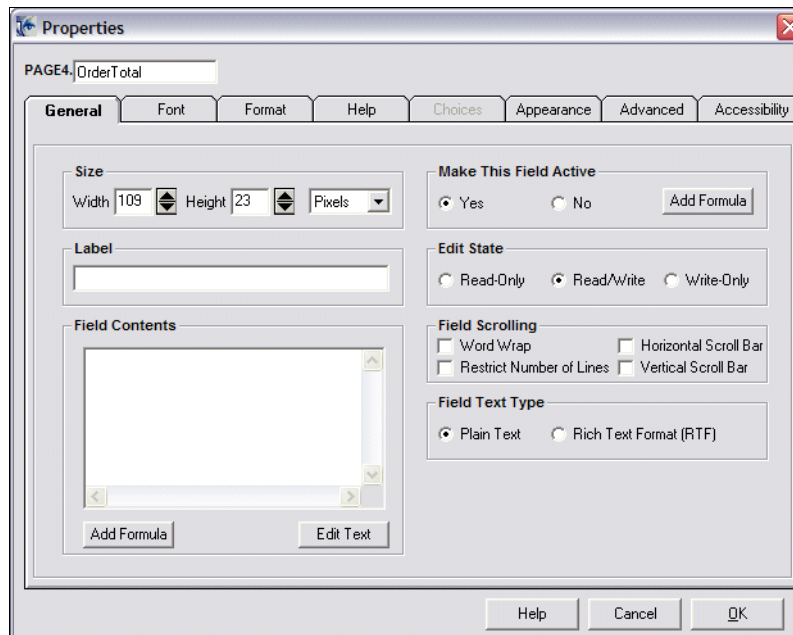


Figure 4-26 Properties box of the Order Total field

3. Select “equal to the sum of multiple fields on the form” from the drop-down.
4. In the formula section, as seen in Figure 4-27:
 - a. Select the hand icon.
 - b. On the form page that is now visible, select all of the fields in the Amount column and then click **OK** as seen in Figure 4-27.

Figure 4-27 Pick wizard for selecting fields for a sum calculation

- c. Exit the Calculation window by clicking **OK**, then exit the Properties window by clicking **OK**. (See Figure 4-28.)

Figure 4-28 Formula wizard of Field Contents property of the Order Total field

4.6.2 Creating a custom option

Custom options are options that are not part of standard XFDL. You create these options yourself, and put them in a different namespace, such as Custom. Custom options are ignored by the Viewer, and never affect the appearance of an item. Use custom options to integrate your form with other applications, to store other information in an item, or to include a specialized formula.

Custom options are used for many purposes, but typically the first purpose that new forms designers encounter is holding toggle computes. Because the purpose of these computes is to watch one form option and then change another one, you cannot place the compute actually within either of the options or you would interfere with both operations. In order to get around this, you can create a custom option, which can go in any form item *or* in the form global. A custom option will not affect the way a form item looks on the screen.

Building a toggle compute

Toggle computes are used frequently in forms design. They allow you to detect when something happens and subsequently cause something else to happen. For instance, you may have a check box and a field on a form and you only want the field to be visible if the check box is on. This might be the case if you have a list of items with a last selection of “Other” — you would want to provide a field so the user can explain what “Other” is, but you would not want to allow the user to do that unless the “Other” check box was actually checked.

Most toggle computes use the “if/then/else” compute structure, the toggle() function and the set() function. The “if/then/else” structure simply means that the compute tells the Viewer “if [this] happens, do [that], otherwise do [something else]”. The [this] portion of the compute is what the toggle() function figures out. The toggle function keeps an eye on a form option that is provided to it, and if it changes from one provided value to another, then the [that] portion is executed.

Generally, to define what we want to happen in the [that] portion, we use the set() function. This allows us to alter any option or options in the form we want, including visibility, color, active, read-only, etc. You need to provide the set function with two pieces of information: the option you want to alter, and the new value.

The tricky part comes in the [something else] portion — you do not actually want anything else to happen if what you are looking for does not happen. In this case, you simply leave the “else” portion blank.

In our example we want to set the approval date and next approval level of the form when a person signs the form. To accomplish this, we select the first signature button on the traditional form page, open the Properties box, and follow these steps:

1. On the **Advanced** tab in the Properties Box of the Signature Button (BUTTON3) in the Custom Option section as shown in Figure 4-29:
 - a. Select **custom** from the drop-down for Namespace Prefix.
 - b. Enter **set_date** in the Name field for your custom option.
 - c. Click **Edit Formula** to open the Formula dialog.

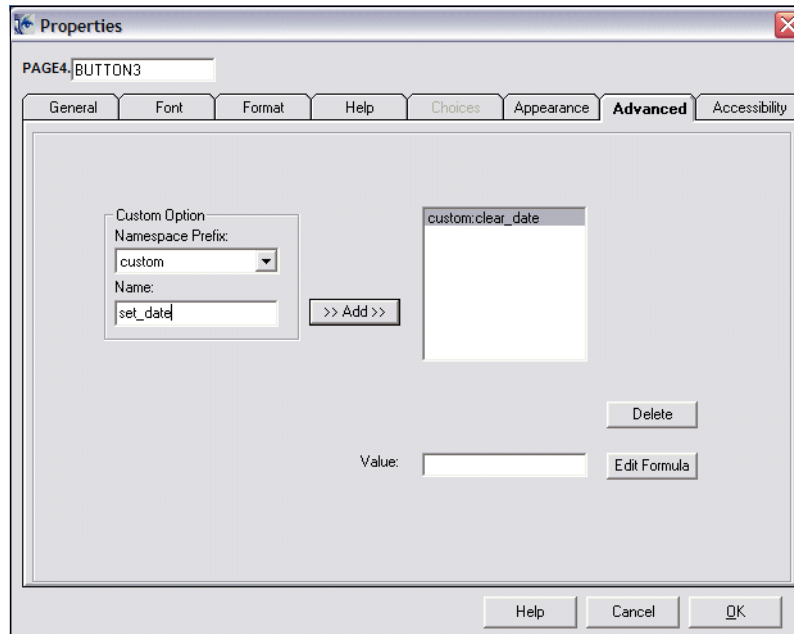


Figure 4-29 Advanced tag in the Properties of the Signature button

2. In the Formula dialog, select **determined by a decision (If/Then/Else)** from the pop-up as shown in Figure 4-30.
3. In the first pop-up, select a function and click the **Function** button that appears.

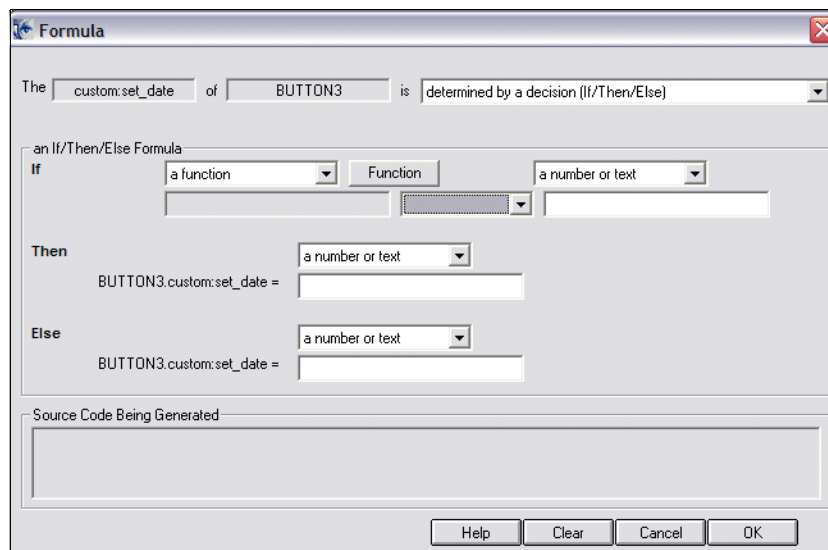


Figure 4-30 Formula dialog of the Custom option

4. In the new dialog that appears (Function Call) as shown in Figure 4-31:
 - a. Select **toggle** from the function pop-up,
 - b. Select the action in the reference pop-up that will trigger the formula:
 - i. For a button, select **activated event of...** and click the button with the hand icon, and click the button that is being monitored by this compute.

Note: Other events to choose from are: value of..., active status of..., format of..., focused event of..., mouseover event of... Choose the one that best fits what you want to accomplish. The “from” and “to” parameters may not apply in every case; please refer to the Workplace Forms the XFDL Event Model 2.5.

- c. Select **off** in the *from* pop-up
- d. Select **on** in the *to* pop-up
- e. Click **OK** to exit the Function Call Dialog.

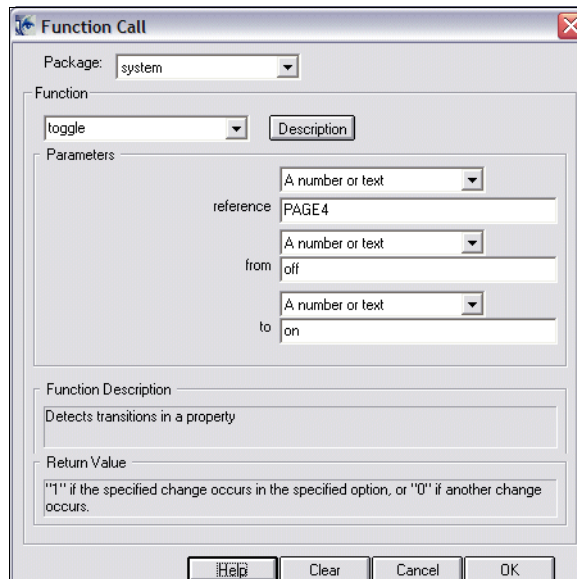


Figure 4-31 Function Call dialog of the Custom option

5. Select **==** (equal to) in the middle pop-up as shown in Figure 4-32.
6. Select a number or text in the next pop-up and enter 1 in the last field of the IF section.
7. Click **OK** to exit the Formula dialog.

Since we want to set values outside of the button holding the toggle compute, we have to exit the Formula dialog and open it again to enter a manually created formula. The code frame for the decision loop will be preserved.

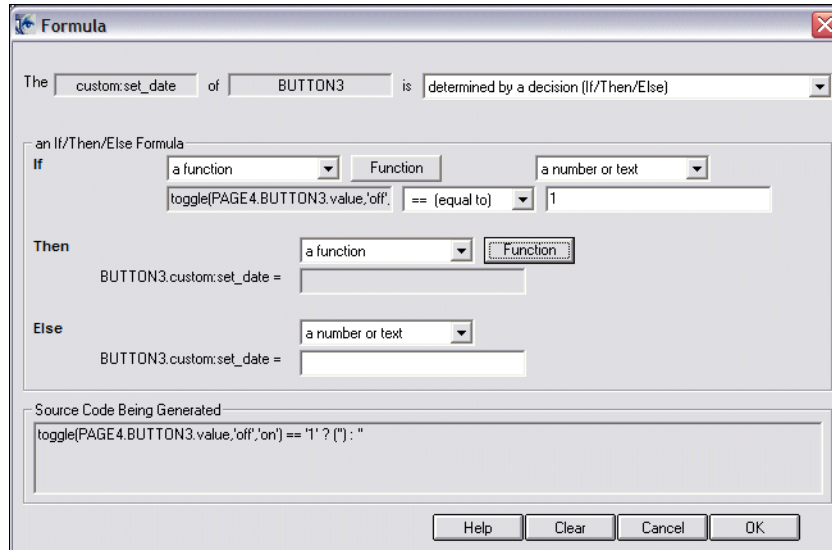


Figure 4-32 Formula dialog after toggle configuration

8. Click **Edit Formula** again to open the *Formula* dialog.
9. Leave the **set by a manually created formula** on as shown in Figure 4-32.
10. In the Formula section, edit the formula code to match the example shown in Figure 4-33.

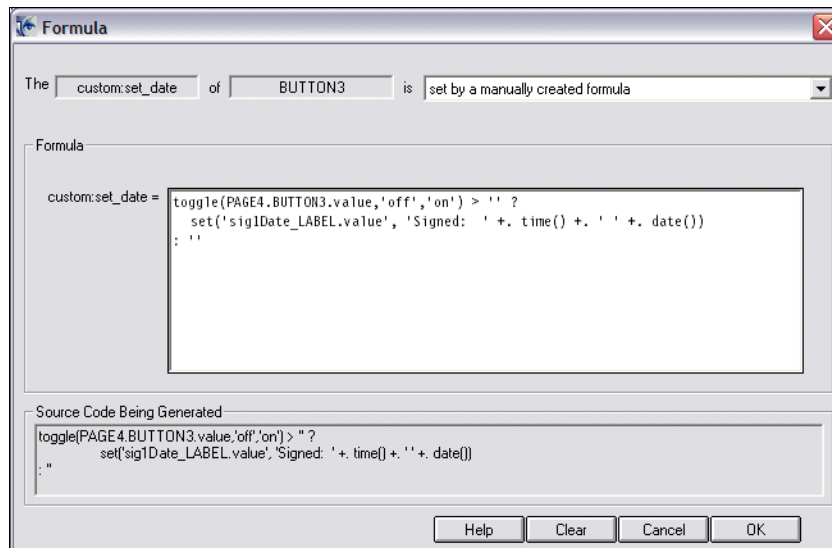


Figure 4-33 Formula dialog to insert a manually created formula

A simplified XFDL code sample from the Sales Quote form is shown in Example 4-3.

Example 4-3 Toggle compute for a button

```
toggle(PAGE4.BUTTON3.value,'off','on') > '' ?
    set('sig1Date_LABEL.value', 'Signed: ' + time() + ' ' + date())
: ''
```

4.6.3 Adding a Submit button

To add user-triggered actions to your XFDL form, you can add buttons to it that can execute a broad range of actions, as listed in Table 4-2.

Table 4-2 Actions of buttons within XFDL

Action	Description
Enclose File	Allows users to enclose multiple files
Display Enclosure	Allows users to select an enclosed file to view
Extract Enclosure	Allows users to extract an enclosed file and save it
Remove Enclosure	Allows users to remove an enclosed file from the form
Enclose Single Item	Allows users to enclose a single file in the form; if there is already a file enclosed, the new file replaces it
Extract Single Item	Allows users to extract the item and save it to disk
Display Single Item	Displays the file
Remove Single Item	Deletes the file
Go To Page	Switches the view to another page in the form
Link to File	Opens a file from the Internet or the user's computer and displays it in either the Viewer or a Web browser
Replace with File	Replaces the form with another file; this file can reside on the Internet or on the user's computer
Submit Data	Submits form data to a server, leaving the form open
Submit Then Cancel	Submits form data to a server and closes the form
Signature	Allows users to sign the form
Select	Allows users to indicate a choice
Cancel	Closes the form without submitting or saving
Print	Prints the form
Save	Saves the form

You can place buttons both in the form and in the form's toolbar. Buttons placed in the toolbar are fully functional, but are not printed as part of the form. Placing eForm-only items, such as e-mail, print, or next page buttons, in a toolbar, allows your form to function as an eForm and a paper form.

To create a Submit button as in our example, do the following steps:

1. Create a button and double-click it.
 - The *Properties* dialog box opens.
2. In the *General* tab, click the pop-up list under **Perform This Action**.
 - The pop-up list opens, displaying a number of actions.

3. Select **Submit Then Cancel** and click **Details** as seen in Figure 4-34.
 - The *Submit Then Cancel Action Details* dialog box opens (Figure 4-35).

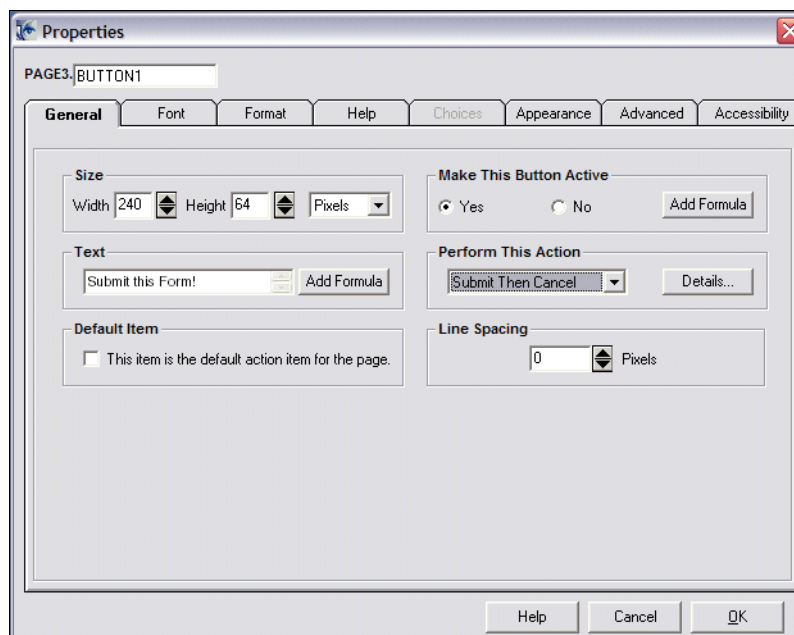


Figure 4-34 General properties of Submit button

4. Under *Transmit Format* leave **XFDL** selected.
5. Under *Compression* leave **Uncompressed** selected.
6. Under *URL* enter the appropriate URL of your submission servlet, either in absolute or relative fashion.
 - a. Enter relative URL: `/WPFormsRedpaper/SubmissionServlet?action=store`
 - b. Click **Add**.
 - c. Exit the Details window by clicking **OK**, then exit the Properties window by clicking **OK**.

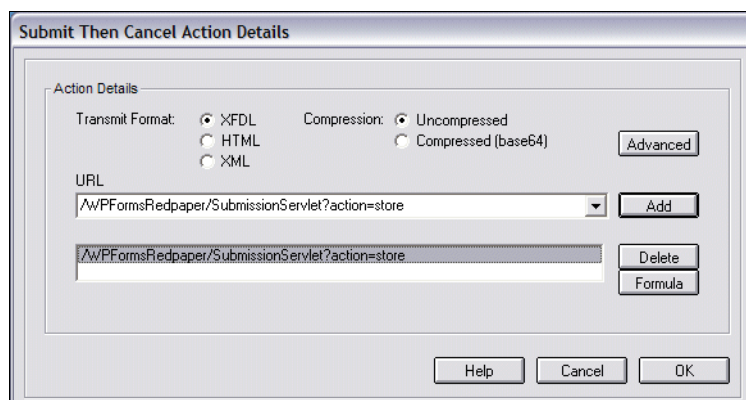


Figure 4-35 Submit Then Cancel Action Details dialog box

4.6.4 Adding Signature buttons

Digital signatures have two purposes:

- Identifying signers:

Digital signatures affix the signer's name and e-mail address to the document.

- Secure documents:

Digital signatures use encryption algorithms and digital certificates to guarantee form security.

When users sign a document, a “snapshot” of the document is taken and hashed to produce a unique number representing your document. If the document changes, hashing produces a different number. As a result, you can think of a hash as a document's digital fingerprint, unique and unmistakable.

In other words, once users have signed a form, any alterations to the signed portion of the form breaks the signature. For example, if a malicious user tampers with the form's XFDL code, the Viewer detects the change. The Viewer then displays an error message warning that one or more of the form's digital signatures is invalid. It also changes the label on the signature button to read “INVALID”. This means that if someone tried to alter a signed form to change the meaning of the form, subsequent users would be alerted to the fact that the form had been changed. This security measure does not prevent the original signers from making changes to the form — if they wish to make changes, they can simply delete their signatures from the form, modify their entries, and re-sign the form.

While creating digital signature buttons can be complicated in XFDL, the Designer provides a simple interface that allows you to create them easily. To create a signature button:

1. Create a button on your form.
2. Double-click the button.
 - a. The Properties dialog box opens.
3. Under Perform This Action, select **Signature** from the pop-up.
4. Click **Details**.
 - a. The Signature Action Details dialog box opens.
5. Ensure that Sign Entire Form is selected, and click **OK**.

The Designer allows you to sign the entire form, or create custom signatures.

Signature filters

Occasionally, you may find that you do not want certain items or options to be signed. You may wish to create different form sections to be signed by different users, or you may want a custom item to continue working after the form is signed. You can accomplish this with signature filters.

Signature filters allow you to set which form elements you want to sign. There are two kinds of signature filters:

- **omit:** The *Omit* filters let you specify the form elements you do not want to sign, and ensures that the rest of the form is signed. This filter guarantees that everything in the form is secured, except the form elements that you choose.
- **keep:** The *Keep* filters let you specify the form elements you want to sign, leaving the rest of the form unsigned. This filter only secures the form elements that you choose, leaving the remainder of the form unprotected.

Note: Omit filters provide greater security than keep filters. It is good practice to use *omit* filters rather than *keep* filters. If you must use a keep filter, use it in conjunction with an omit filter. For example, you might want to omit the items in a “For office use only” section from a user’s signature, but you would secure the location and appearance of these items with a keep filter.

Workplace Forms Designer makes it easy to create secure signature filters. It allows you to select specific items that you want to omit from the signature, while automatically retaining item position information. This filter option, which is called *signitemrefs*, allows you to specify individual items that the signature will omit or keep. For example, you might set the filter to omit BUTTON1 on PAGE1.

The Designer also provides an interface that assists you in creating custom signature filters. Custom signature filters allow you to omit or keep specific form elements, such as:

- ▶ **Items:**
Specifies the types of items that the signature will omit or keep. For example, you might set the filter to omit all button items from the signature.
- ▶ **Options:**
Specifies the types of options that the signature will omit or keep. For example, you might set the filter to omit all *triggeritem* options from the signature.
- ▶ **Groups:**
Specifies one or more groups, as defined by the group option, that the signature will either omit or keep. This filters any radio buttons or cells belonging to that group, but does not filter list, pop-up, or combo box items. For example, if you had a pop-up containing a cell for each State, you might set the filter to omit the State group, which would omit all cells in that group.
- ▶ **Data groups:**
Specifies one or more data groups, as defined by the data group option, that the signature will either omit or keep. This filters data items belonging to that data group, but does not filter any action, button, or cell items. For example, if you had an enclosure button containing references, you might set a filter to omit the References data group, which would omit all data items in that group.

Creating multiple signatures

Forms often require multiple signatures. In fact, it is very common for some signatures to endorse other signatures. A combination of signatures is called overlapping signatures.

In our example, the Sales Quote form requires employees to complete personal, customer and order information sections and sign them. Managers must then endorse the employee data depending on the total amount. Finally, a director might receive the form, depending on the amount and endorses the entire form. As you can see, this would require three different signature buttons, all with differing signature filters.

XFDL allows an unlimited number of signatures on a form. The signatures can sign separate and overlapping sections of the form, as well as endorsing other signatures, depending on the signature filters.

Creating Clickwrap signatures

Clickwrap signatures are electronic signatures that do not require digital certificates. While they offer a measure of security due to an encryption algorithm, Clickwrap signatures are not security tools. Instead, Clickwrap signatures offer a simple method of obtaining electronic evidence of user acceptance to an electronic agreement. The Clickwrap signing ceremony authenticates users through a series of questions and answers, and records the signer's consent. Clickwrap style agreements are frequently found in licensing agreements and other online transactions.

The simplest Clickwrap signing ceremony requests that users click the **Accept** button to sign a form. However, you can include a number of options in your Clickwrap signing ceremony. For example, you can add company information, the text of your agreement, questions and answers, and echo text. Echo text allows you to designate text that the user must re-type before signing the form. This ensures that the user has read vital information before indicating their agreement.

To create a Clickwrap signature button in the Sales Quote form, follow these steps:

1. Create a button on your form.
2. Double-click the button.
 - The Properties dialog box opens as shown Figure 4-36.

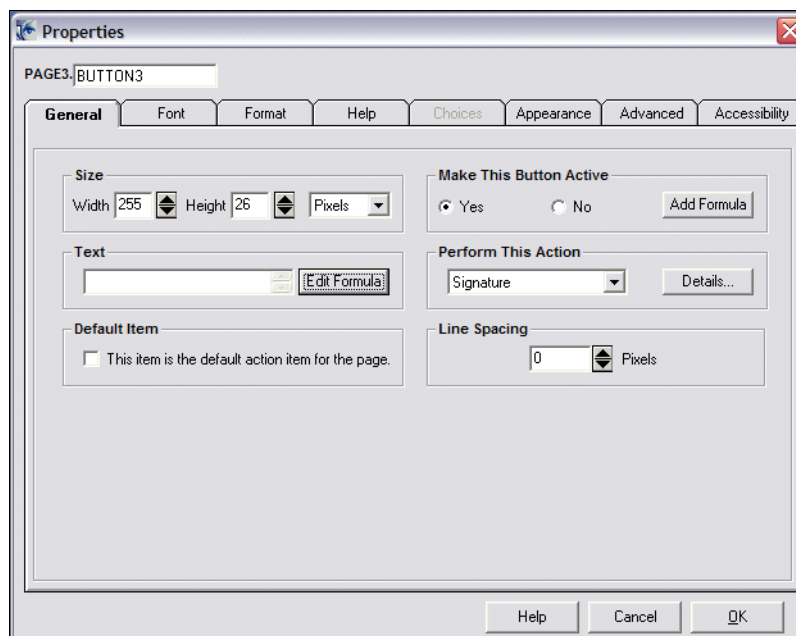


Figure 4-36 Properties dialog box of the Signature button

3. Under Text, select **Edit Formula**.
 - The Formula dialog box opens.
4. Leave the selection, *is set by a manually created formula*.
5. Enter the conditional formula for the label text as shown in Figure 4-37.
6. Click **OK** to return to the Properties dialog box.

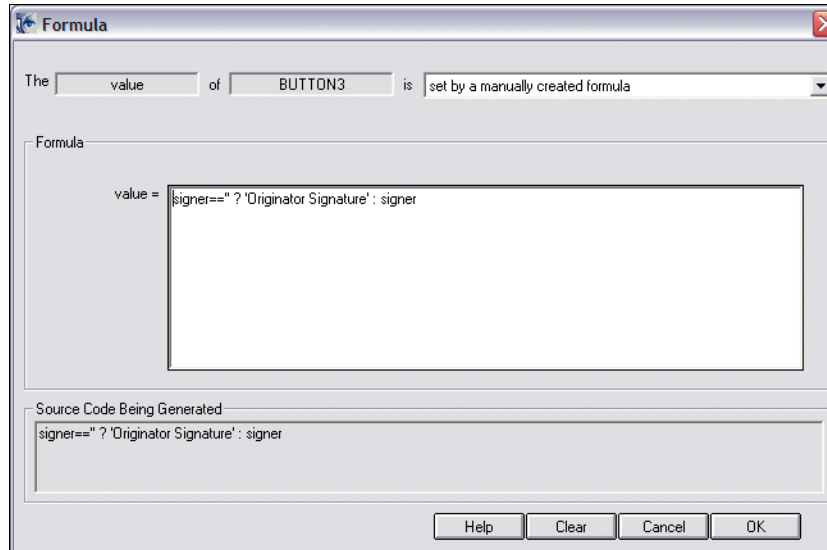


Figure 4-37 Formula dialog box for the label of a signature button

7. Under Perform This Action, select **Signature** from the pop-up.
8. Click **Details**.
 - The Signature Action Details dialog box opens.
9. Select the **Signature Engine Settings** tab.
 - The *Signature Action Details* dialog opens as shown in Figure 4-38.
10. From the *Use This Digital Signature Engine* pop-up, select **ClickWrap**.
 - The contents of the Parameters section changes to reflect the new setting.
11. Type text into the parameters you want to use in your Clickwrap signing ceremony.
12. Click **OK**.

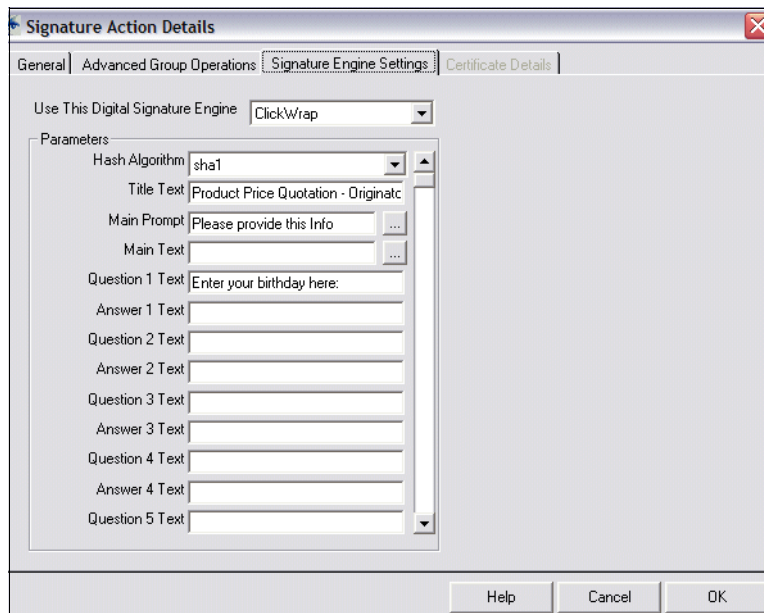


Figure 4-38 Signature Action Details dialog box

13. Select the **General** tab (refer to Figure 4-40 on page 102).
14. Select **Keep** in the top right corner of the box indicating that you want to include the objects selected in the next step in your signature.
15. Select the hand icon.
 - The form will open with a hand cursor.
16. Select the fields you want to be signed with this button, as shown in Figure 4-39 here, and click **OK**.

IBM® Workplace Forms(tm) Designer - [C:\Forms\itso\stage1\Redpaper_Forms_Sample_v06.xfd]

File Edit View Insert Arrange Tools Window Help

Western European Helvetica

Wizard Page 1 Wizard Page 2 Form Page 1

Submit this Form! **Product Price Quotation** IBM Workplace Forms

To select items to be signed or omitted, click on them in the form.
Click on "OK" when done.

Status: <Formula>

Products						
Item	Item number	# in stock	quantity	price	discount	total
Select your product ▼	<Formula>	<Formula>		<Formula>	discount ▼	<Formula>
Select your product ▼	<Formula>	<Formula>		<Formula>	discount ▼	<Formula>
Select your product ▼	<Formula>	<Formula>		<Formula>	discount ▼	<Formula>
Select your product ▼	<Formula>	<Formula>		<Formula>	discount ▼	<Formula>
Select your product ▼	<Formula>	<Formula>		<Formula>	discount ▼	<Formula>
Grand Total				<Formula>	<Formula>	<Formula>

Attach fax and/or supporting documents Add View Remove Save As <Formula>

Originator's Position Select your position ▼ **Originated By:** <Formula>

Approval Level: Product Manager **Authorized By:** <Formula>

Approval Level: Product Director **Authorized By:** <Formula>

Item: BUTTON3 Type: button X: 5.558" Y: 6.267" W: 2.125" H: 0.217"

Figure 4-39 Selecting fields to be included in signature

17. On the **General** tab you will now see the objects included in the signature as shown in Figure 4-40. Click **OK** to exit the dialog.

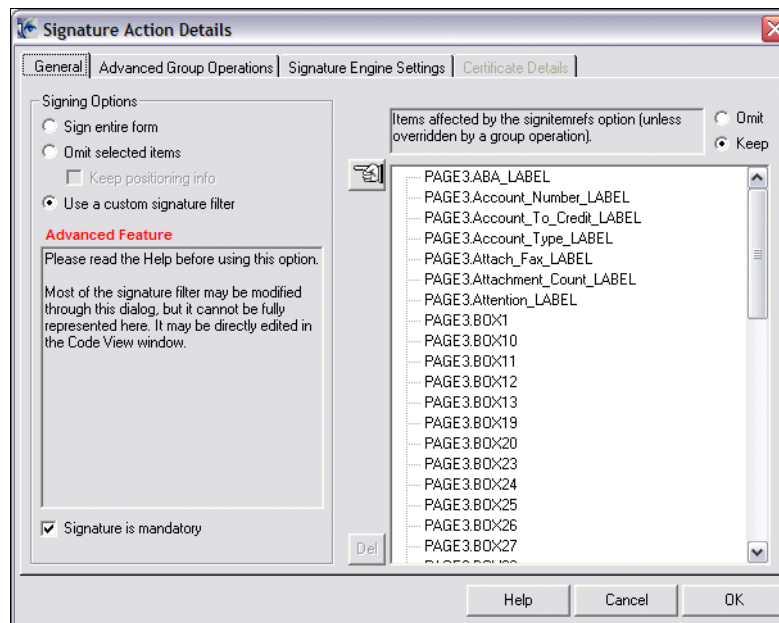


Figure 4-40 Signature Action Details showing the list of objects kept in the signature

Note: You cannot unselect a selected item. Finish the procedure by clicking **OK** and then remove the item from the *Signature Action Details* dialog, by selecting the item and clicking the **Del** button.

4.7 Adding the XML Data Model

The XML Data Model makes it easier to integrate XFDL forms with other applications by:

- ▶ Separating the form data into a separate block of XML. This makes the data easy to locate and parse within the form, while also allowing the data to conform to any valid XML structure, such as that dictated by a schema.
- ▶ Enabling schema validation of data. This ensures that all data collected adheres to a defined schema, thereby reducing input errors.

Additionally, you can use the XML Data Model to enable Smartfill functionality in the Viewer, which can automatically complete portions of the form for the user.

The XML Data Model allows form designers to create separate blocks of arbitrary XML within an XFDL form that share data with form elements, such as fields. This is useful for integrating with applications that require data in a particular XML format, such as schema compliant data. The XML Data Model is based on the XForms standard, as published by the W3C, but is not limited to XForms. Refer to the XFDL Specification for more information.

The XML Data Model consists of instances and bindings. Instances are the base data structures for the data model and the document. By carefully modeling each instance element, integration with interfaces to other business processes. The ability to have multiple instances allows for a variety of integration techniques. Bindings represent a relationship between a Workplace Forms form element (field, radio, list, etc.) and an element or attribute of an instance. The value of the bind is that data from the presentation layer is moved in and out of the data layer (instances) seamlessly during prepopulation or while a user is completing a form.

Another benefit of the XML Data Model is easy integration with other application and systems. With XML instances that comply with the XML schemas of other applications, processing of the form for integration with other applications is kept to an absolute minimum. Any application can simply retrieve the instance without having to process a Workplace Forms form or message XML for each data item.

Also, since multiple options can be bound to a single data instance, this alleviates the need for multiple “set” statements to synchronize data within the document. All options that are bound to a data instance are kept in sync by the Workplace Forms Viewer automatically.

When to use the XML Data Model

You can use the XML Data Model in any of the following scenarios:

- ▶ **XML applications:**

The XML Data Model is most useful when integrating eForms with applications that already use XML, especially if those applications already offer XML interfaces. In these cases, you can design forms that will submit the XML data directly to the application, and will not need to program a custom module that extracts the data from the form. Furthermore, you can format the data to match any schema, and validate the data against the schema before submission.

- ▶ **Non-XML applications:**

Even if an application does not use XML, you can still benefit from using the XML Data Model. The data model simplifies copying information from one page to another, making wizard-style forms easier to create and manage. Furthermore, although custom programming is still required for back-end processing, the data model makes it far easier to extract data from the form.

- ▶ **Automatic form completion:**

If a form requires users to repeatedly enter the same information, such as, their name and contact information, you can set up the form to use Smartfill. Smartfill can automate portions of form completion by capturing frequently used information and giving the user the option to automatically load that information while they are completing a form.

Overview of the XML Data Model

The XML Data Model serves several purposes. Its core function is to provide a way to achieve interoperability with other applications. In addition, it provides schema validation capabilities, and allows you to enable the Viewer’s Smartfill feature, which can automatically complete portions of the form for the user.

The XML Data Model contains three core parts working together to create a complete model:

- ▶ **Data instances:**

Data instances are arbitrary blocks of XML. A data model may contain any number of data instances, and each instance is normally created to serve a particular purpose. For example, if your form provides data to both an accounting application and a shipping application, you may want to create two data instances — one for each application.

- Bindings:

Each data instance has associated bindings. Bindings tie one element in the data instance to one or more elements in the form description. For example, if a form had a firstName field on both the first and second pages, you might bind the firstName element in your data instance to both fields. Once this is done, all three elements will share data, meaning that if one element is changed the other two elements are updated to mirror that change.

- Submission rules:

Each data instance may have an associated set of submission rules. These rules control how a data instance is transmitted when it is submitted for processing. This is an optional feature, and is only necessary when you want to submit the data instance by itself, without the rest of the form. There are many cases in which you may want to submit the entire form, and then retrieve the data instance from the form during processing. This is particularly true when you are using signatures on your forms.

4.7.1 Creating an XML Data Model

When creating an XML Data Model, it is a good idea to create your data instances one at a time, and to set up the bindings and submission rules for that instance before moving on to the next data instance.

To create an XML Data Model, you must:

- Declare the XML Data Model in the form.
- Create a data instance.
- Bind the elements of the data instance.
- Set up submission rules for the instance (optional).
- Create a submission button for the instance (optional).

The data model is always declared as an option in the global item of a form's global page, and begins with the <xmlmodel> tag. Each data instance is inserted within an <instances> tag in the XML model. Essentially, the XML Data Model is a block of XML that is placed at the beginning of a form, within the global page's global item, as shown in Example 4-4.

Example 4-4 Example of the XML Data Model

```
<globalpage sid="global">
  <global sid="global">
    <xmlmodel>
      <instances>
        <xforms:instance xmlns="" id="FormOrgData">
          <FormOrgData>
            <FirstName></FirstName>
            <LastName></LastName>
            <ID></ID>
            <ContactInfo></ContactInfo>
            <Manager></Manager>
          </FormOrgData>
        </xforms:instance>
      </instances>
    </xmlmodel>
  </global>
</globalpage>
```

This block of XML allows for arbitrary data, meaning that it can contain any data and can be formatted in any manner. Furthermore, individual elements in the data model can be bound to one or more elements in the form description. This binding causes the elements to share data. If one element is changed, the other elements are updated to mirror that change.

This allows you to create a separate block of data within the form, format it any way you like, and bind it to form elements so that data entered by the user is automatically copied to the data model. For example, you could include the block of data that is required by an application (such as an IBM Content Manager based system), format the data so that it complies with a specific schema, and then bind that data model to the form description.

The result is a block of XML data that can be structured to meet any needs, extracted easily by other applications, and transmitted without the rest of the form.

4.7.2 Creating XML bindings

Once you have created a data instance, you need to bind the data elements. A bind creates a link between two elements in the form. This link causes those elements to share information, meaning that if one of the elements is changed, the other element is updated to mirror that change.

There are two types of binds:

- Data element to form element:

In this case, one element in the data instance is bound to one option in the form description. This type of bind links your data model to the form description, so that information entered by the user is copied to your data model. For example, you might create a bind that links the firstName element in your data instance to the firstNameField.value option in your form description.

- Data element to data element:

In this case, one element in the data instance is bound to another element in the data model. This type of bind is often used to perform special calculations or to copy information from one part of the data model to another. For example, you might have a data holder element that performs a calculation. You could then bind this element to copy the result of the calculation into your data instance.

Each bind creates a one-to-one relationship: one data element to one data or form element. However, you can also create a one-to-many relationship — one data element to many data or form elements — by creating additional binds. For example, you could bind the firstName data element to both the firstName field on page one of the form and the firstName field on page two of the form. All of the bindings are contained within a <bindings> tag in the data model, as shown:bind a form element to more than one data element.

An XML binding is a way of linking the fields on your form to the XML instance that describes the captured data.

The following steps describe how to use the XML Data dialog to create your XML bindings:

1. Open the XML Data Model dialog from the toolbar, **Tools** → **XML Data Model** → **Create/Edit Manually**, and click the **Bindings** tab.
2. Click the button that looks like a file menu and then select a data element from the XML instance.
3. Click the button with the hand icon; the workspace will be brought to the front. Select the corresponding field.
4. Click the **Add** button to add the binding to the list.

5. Repeat steps 2, 3, and 4 until bindings have been created for all the elements in your XML instance.
6. When you are finished, click **OK** to exit the XML Data Model Dialog.

All of the bindings are contained within a <bindings> tag in the data model, as shown in Example 4-5.

Example 4-5 Sample XML binding in the instance FormOrgData

```
<xmlmodel>
  <instances>
    ...
  </instances>
  <bindings>
    <bind>
      <instanceid>FormOrgData</instanceid>
      <ref>[null:FormOrgData][null:FirstName]</ref>
      <boundoption>PAGE1.OrgFirstName.value</boundoption>
    </bind>
    <bind>
      <instanceid>FormOrgData</instanceid>
      <ref>[null:FormOrgData][null:LastName]</ref>
      <boundoption>PAGE1.OrgLastName.value</boundoption>
    </bind>
  </bindings>
</xmlmodel>
```

Each bind contains tags that determine which elements are bound together. The first bound element must be part of a data instance, and is identified by an <instanceid> tag and a <ref> tag. The second bound element can be part of the data model or part of the form description, and is identified by a <boundoption> tag.

4.7.3 XML schema validation

As we have seen, the core of the data model consists of data instances, their bindings, and submission rules. In addition, the data model can reference one or more schemas, which allow you to validate data instances against them. A schema can be embedded in the form by including a <schemas> tag in the data model.

When adding schema validation, you can choose to either embed one or more schema files in the form itself, or refer to external schema files that are saved on the user's computer. Embedding schemas in the form will increase the overall size of the form and may affect performance, especially when low bandwidth is available. However, referring to external schema files requires you to distribute those files to client computers. The architecture of your overall application will probably dictate which solution you should use.

Normally, each instance in the data model is validated against all available schemas. However, if a schema is defined for a particular namespace, only those instances that belong to that namespace are validated against it. This allows you to apply specific schemas to specific data instances. Finally, you must restrict all schemas to a single, self-contained file. The Viewer does not support the use of the import or include tags.

To add schema validation to a form, you must:

1. Embed the schemas you want to include in the form.
2. Register the embedded schemas.
3. Register any external schemas.
4. Add the xmlmodelValidate function to the form.

Adding the xmlmodelValidate function

The data model is validated against all registered schemas when you call the `xmlmodelValidate` function in the form. In most cases, you will want to tie this function to the submission of the form, so that the data model is validated just before the form is submitted.

Optionally, you may prefer to validate the data model during processing on your back-end systems. In this case, you can use any available XML schema tools to validate the data model.

Enabling Smartfill

Smartfill enables users to store frequently used form information on their local computers and to re-use this information when completing other forms that require it. Smartfill is intended to make it easier to fill out forms that require commonly used information, such as the user's name and address.

Smartfill works by creating data fragments. As the name suggests, these are small groupings (or fragments) of data from the form. Each data fragment represents a particular set of data. For example, you might create one data fragment for the user's home address and a second data fragment for the user's work address.

The first time a user submits or saves a Smartfill enabled form, the system will save the data fragments defined in the form to the user's local computer. The next time the user completes a form requiring those data fragments, the Viewer will offer to automatically load the information into the form.

Once a data fragment has been saved, the Viewer will offer to load that data into each form that requires it. However, once loaded, the user can modify the data at any time. Any changes the user makes will be saved over the old data fragment when the user saves or submit the form.

Because data fragments are stored as XML files within the user's profile on the local computer, they can be accessed by other applications. As such, they should only capture commonly used information, such as names and addresses. Never use them to store confidential information, such as credit card numbers or passwords.

The Smartfill feature allows you to include any number of data fragments in your form, and you should give careful consideration to how you want to define and arrange your data fragments.

4.8 Building the Servlet

In the following scenario we show you how to build a Servlet.

4.8.1 Where we are in the process: Building Stage 1 of the base scenario

The following diagram is intended to provide an overview of the key steps involved to build the base scenario. This focuses on building the Form, the JSPs, and the Servlet (Figure 4-41.)

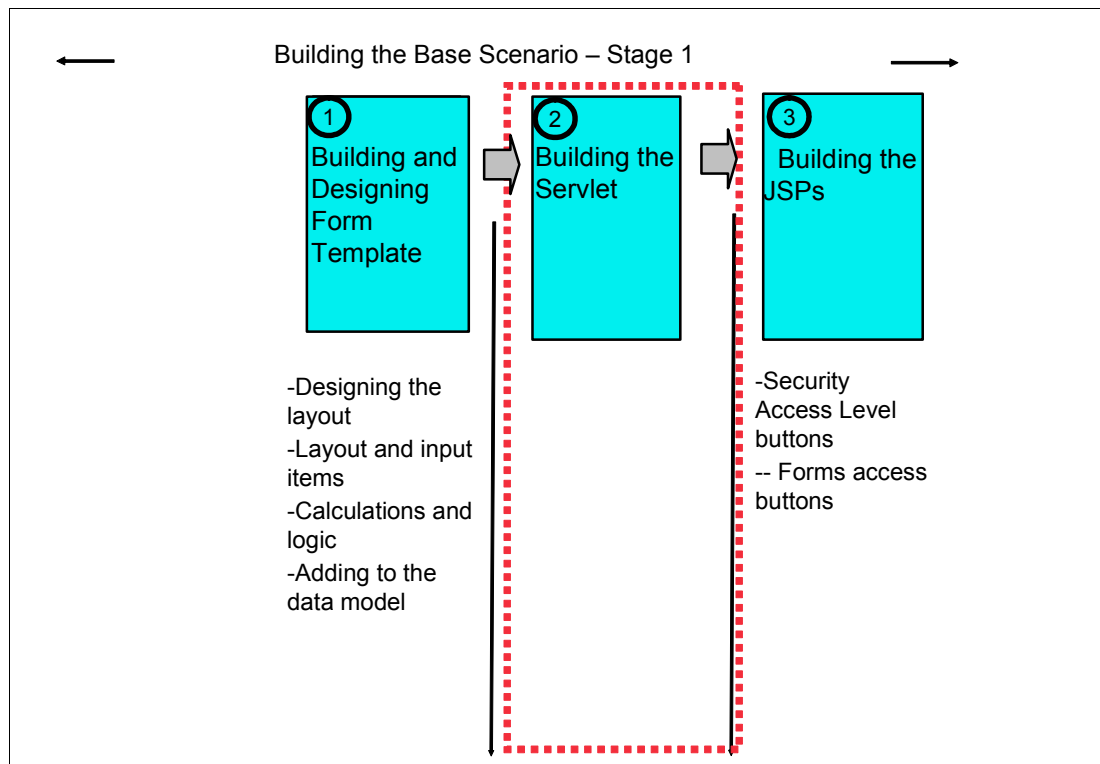


Figure 4-41 Overview of major steps involved in building the core base scenario application

Stage 1 presents a “stand alone” form scenario, where the form design does not require any interaction with external data (such as pre-filling it with some instantiation data or interaction with any external data sources during work on the form at client side). The only interaction of the built application with the form is in the extraction of a form state value after submission. This value decides about the directory to store the form in. That way the servlet will represent the main business logic of the application.

4.8.2 Basic servlet methods

In the basic application we use a servlet as Web application. The servlet can be called by a URL submitted by a browser (or any other Web client) and exposes four basic methods to interact with the environment:

- ▶ The init method does basic servlet initiation and initiates the Workplace Forms API for future use
- ▶ The doGet method is called, whenever the client submitted a GET request. This is a URL pointing to the servlet. It may contain additional parameters, that can be evaluated in servlet doGet method. This is a convenient way to request different actions from the servlet.

- The doPost method is called by any POST request submitted to the servlet. The POST request works like a GET request but submits an additional data stream. In this data stream, Forms Viewer ships the submitted XFDL data.
- The destroy method cleans up by destroying all necessary objects to free the allocated memory.

In our project we use the following actions for GET and POST (see Table 4-3).

Table 4-3 List of supported actions in a GET request

Request	Parameters	Action performed
GET	action=listTemplates	Calls dirlisting1.jsp with the template folder as target.
GET	action=workbasket	Calls dirlisting1.jsp with one of the work folders as target (workbasket, manager approval, director approval) depending on current user role.
GET	action=listApproved	Calls dirlisting1.jsp with the folder with approved forms as target.
GET	action=listCancelled	Calls dirlisting1.jsp with the folder with cancelled forms as target.
GET	action=setRole&userRole=XXXX	Assigns a new user role (simulation of a new login with an other user name / password). Called from index1.jsp using the available role buttons (1000 = Employee.... 1031 = Director)
GET	action=getJSP&jsp=XXXX	Requests the servlet to open a new JSP for the browser. This is a convenient method to route navigation from one JSP to another via the servlet as proxy. So the servlet can control the parameters passed to every JSP called. The feature is used in navigation buttons on index1.jsp and Home buttons on all other JSPs.
POST	action=bounceback	Submits to the browser the received post data as XML content. This method is useful for tests. It requires a special button in the form with a submit URL containing the assigned parameter (http://servletURL?action=bounceback)
POST	action=store	Retrieves the order state and stores the received POST data to file system in a folder depending on the detected form state (submissions, manager approval, director approval, approved or completed orders). After this, the success1.jsp is called.

4.8.3 Servlet code skeleton

To create the servlet, create a new Dynamic Web Project in RAD6 and a new package (you can name it *wpFormsRedpaper* or give it any other name of your choice).

Then create a new Servlet in this project, as follows.

Right-click the new package and select from the property box **New - Other - Servlet**. (See Figure 4-42.)

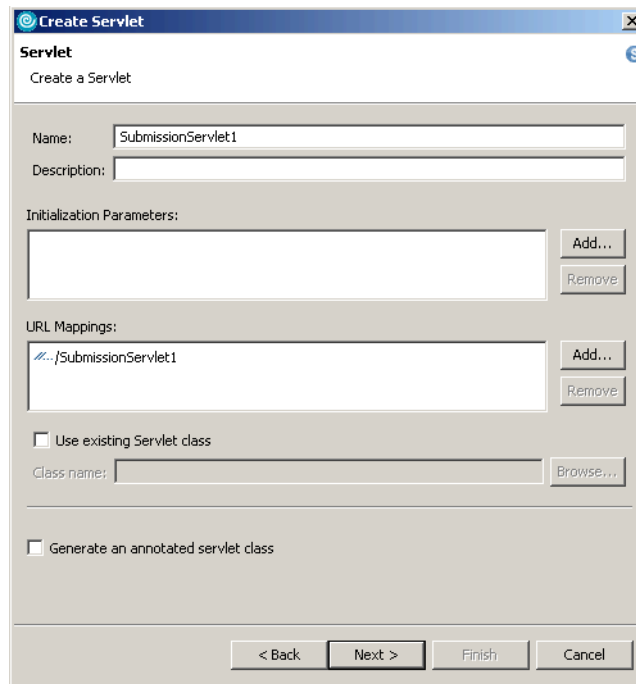


Figure 4-42 New servlet basics

Give a servlet name *SubmissionServlet1*, disable the *Generate an annotated servlet class* option, and click **Next** (see Figure 4-43).

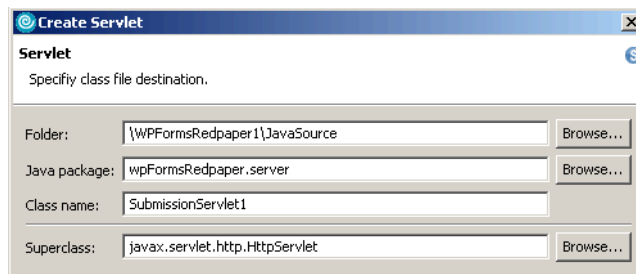


Figure 4-43 *SubmissionServlet* - package

Assign the new created package and click **Next**. (See Figure 4-44.)

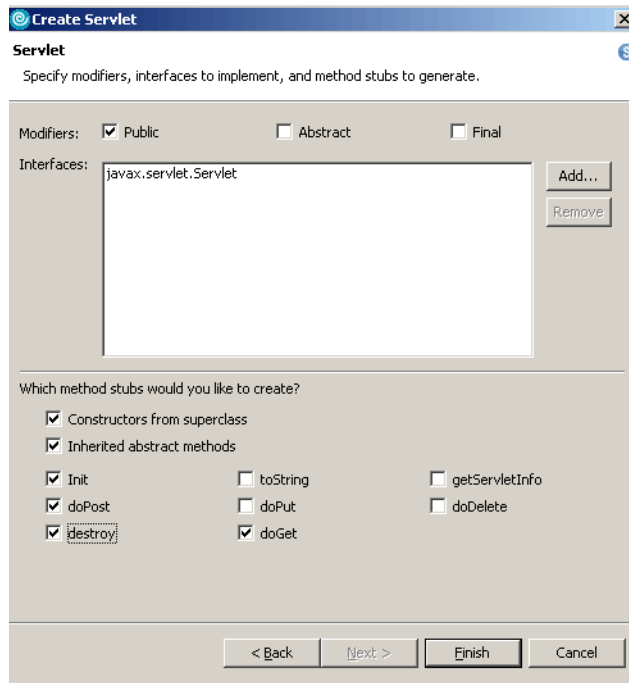


Figure 4-44 *SubmissionServlet methods*

Make sure all four basic methods (Init, doGet, doPost, destroy) are selected and click **Finish**.

The servlet comes with the skeleton shown in Example 4-6. Note that the methods are reordered, all comments are deleted to make the example short, and the method `SubmissionServlet1()` is removed.

Example 4-6 *New Servlet code skeleton*

```
package wpFormsRedpaper.server;

import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SubmissionServlet1 extends HttpServlet implements Servlet {

    public void init(ServletConfig arg0) throws ServletException {
    }
    protected void doGet(HttpServletRequest arg0, HttpServletResponse arg1) throws
ServletException, IOException {
    }
    protected void doPost(HttpServletRequest arg0, HttpServletResponse arg1) throws
ServletException, IOException {
    }
    public void destroy() {
    }
}
```

After the new servlet creation, an error message can be reported in the *Problems* view.

We found that the generated description file was sometimes not accepted. In this case, double-click the **Deployment Descriptor** file in the project outline and open the **Source** tab.

Search here for the following entries:

```
<!-- @generated
wpFormsRedpaper.server.SubmissionServlet1#web/servlet.wpFormsRedpaper.server.SubmissionServlet1 -->
    <servlet>
        <description>Form handler for stage 1</description>
        <display-name>SubmissionServlet1</display-name>
        <servlet-name>SubmissionServlet1</servlet-name>
        <servlet-class>wpFormsRedpaper.server.SubmissionServlet1</servlet-class>
    </servlet>
```

After deleting the `<description>` and `<display-name>` tags and saving, the errors should have gone.

To prepare the servlet for “productive use” we will add some external libraries and code for initiation.

First add necessary external jars for Workplace Forms API. Right-click the project, choose **Properties** from the context menu, and select **Java Build Path** and go to tab **Libraries**. (See Figure 4-45.)

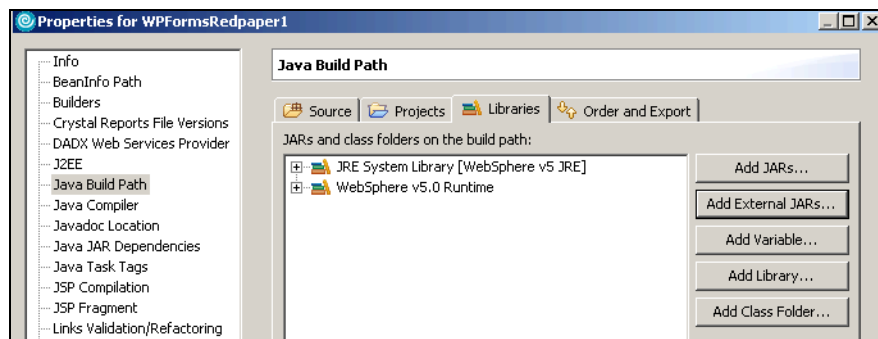


Figure 4-45 Configure Java Build Path

Click **Add External JARs** and select in the folder of the installed API [System32]\PureEdge\65\java\classes pe_api.jar and uwi_api.jar. (See Figure 4-46.)

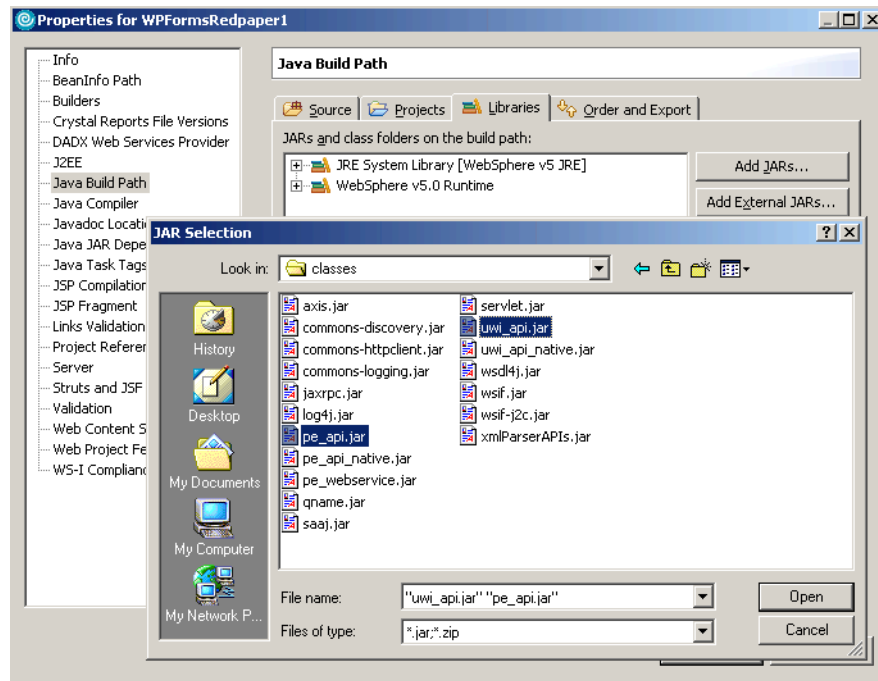


Figure 4-46 Adding external jars

Close property box and open the created new servlet code file (SubmissionServlet1.java).

Add here all necessary imports (The concrete imports can vary depending on the additional functionality you plan to implement — we will reference here all classes finally used in this stage independent from the actual progress of coding. So we will have, over this time, some warnings about unused imports. Do not worry about this.

Insert the following imports (Example 4-7).

Example 4-7 Init method code sample

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.util.Date;
import java.util.Properties;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.omg.CORBA.Any;

import com.PureEdge.DTK;
```

```
import com.PureEdge.IFSSingleton;
import com.PureEdge.error.UWException;
import com.PureEdge.xfdl.FormNodeP;
import com.PureEdge.xfdl.XFDL;
```

Now we can start to add class attributes and initiation code. To store folder names mapping to state values and file name prefix for stored forms, we use a `folders.properties` file as shown in Example 4-8.

Example 4-8 folders.properties file

```
TEMPLATE_FOLDER = \\Redpaper_Demo\\Form_Templates\\
1=\\Redpaper_Demo\\Form_Templates\\

MANAGER_FOLDER=\\Redpaper_Demo\\Manager_Forms\\
2=\\Redpaper_Demo\\Manager_Forms\\

DIRECTOR_FOLDER = \\Redpaper_Demo\\Director_Forms\\
3=\\Redpaper_Demo\\Director_Forms\\

APPROVED_FOLDER = \\Redpaper_Demo\\Approved_Forms\\
4=\\Redpaper_Demo\\Approved_Forms\\

SALES_REP_FOLDER = \\Redpaper_Demo\\Sales_Rep_Forms\\
5=\\Redpaper_Demo\\Sales_Rep_Forms\\

CANCELLED_FOLDER = \\Redpaper_Demo\\Cancelled_Forms\\
6=\\Redpaper_Demo\\Cancelled_Forms\\
FORM_NAME_PREFIX=RedPaperForm
```

Create this file in `WEB-INF` folder of the project and now start adding class parameters and code to the `init` method. Insert the initial body code to the created methods as shown in Example 4-9.

Example 4-9 SubmissionServlet1 class with attributes init and destroy method

```
public class SubmissionServlet1 extends HttpServlet implements Servlet {

    private final static int STREAMING_BLOCK_SIZE = 32768;
    private final static String PID_START_TAG = "<PID>";
    private final static String PID_END_TAG = "</PID>";
    //Form Storage Parameters
    private Properties props = null;
    private static String TEMPLATE_FOLDER;
    private static String MANAGER_FOLDER;
    private static String DIRECTOR_FOLDER;
    private static String APPROVED_FOLDER;
    private static String SALES_REP_FOLDER;
    private static String CANCELLED_FOLDER;
    private static String FORM_NAME_PREFIX;
    //Database Properties
    private Properties orderProps = null;
    //Form Meta-Data Related Parameters
    private final static String METADATA_INSTANCE_ID = "FormMetaData";
    private static ServletConfig conf;

    public void init(ServletConfig config) throws ServletException {
        conf = config;
        System.out.println("SubmissionServlet: init(): started");
        //Initialize the Workplace Forms API
```

```

    try {
        DTK.initialize("RedpaperDemo", "1.0.0", "6.5.0");
    } catch (UWException initE) {
        System.out
            .println("SubmissionServlet: init(): exception occurred initializing
Workplace Forms API: "
                + initE.toString());
    } catch (Exception anE) {
        System.out
            .println("SubmissionServlet: init(): exception occurred: "
                + anE.toString());
    }
    //Read in the properties file for the submission folder names
    try {
        ServletContext ctx = config.getServletContext();
        //ctx.getRealPath()
        InputStream inputStream = ctx
            .getResourceAsStream("/WEB-INF/folders.properties");
        props = new java.util.Properties();
        props.load(inputStream);
        TEMPLATE_FOLDER = props.getProperty("TEMPLATE_FOLDER");
        MANAGER_FOLDER = props.getProperty("MANAGER_FOLDER");
        DIRECTOR_FOLDER = props.getProperty("DIRECTOR_FOLDER");
        APPROVED_FOLDER = props.getProperty("APPROVED_FOLDER");
        SALES_REP_FOLDER = props.getProperty("SALES_REP_FOLDER");
        CANCELLED_FOLDER = props.getProperty("CANCELLED_FOLDER");
        FORM_NAME_PREFIX = props.getProperty("FORM_NAME_PREFIX");

    } catch (Exception anE) {
        System.out
            .println("SubmissionServlet: init(): exception occurred reading properties
file: "
                + anE.toString());
    }
    System.out.println("SubmissionServlet: init(): completed");
}
public void destroy() {
    super.destroy();
}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    // TODO Auto-generated method stub
}
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    // TODO Auto-generated method stub
}
}

```

There is no forms specific code except the `DTK.initialize` statement. It connects us to Workplace Forms API of version 6.5.0. This will allow us to access the XFDL form in the code.

Now we are prepared to implement the `doGet` methods (operating in this stage only for navigation), and later on, the `doPost` methods handling the submitted form.

4.8.4 Creating a template repository and form storage structure

In Stage 1 we have no other storage medium as file system (see Figure 4-47). To make it available to the application, we will create in the WebContent folder of the project a sub-folder named Redpaper_Demo with subfolders as named in the supplied folders.properties file:

```
Redpaper_Demo/Form_Templates
Redpaper_Demo/Manager_Forms
Redpaper_Demo/Director_Forms
Redpaper_Demo/Approved_Forms
Redpaper_Demo/Sales_Rep_Forms
Redpaper_Demo/Cancelled_Forms
```

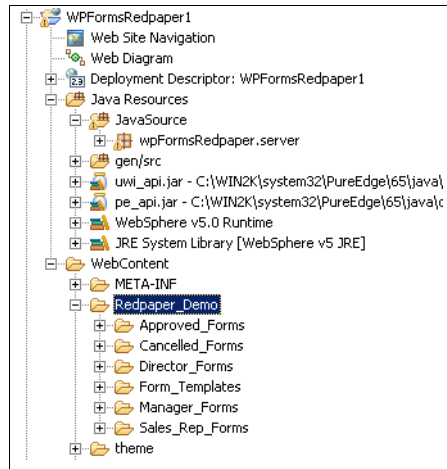


Figure 4-47 Form storage directory Redpaper_Demo in WebContent folder

In the folder Redpaper_Demo/Form_Templates, paste the created forms. (See Figure 4-48.) Just to have a real list, we pasted some arbitrary forms in the directory as well. The form we will use at this stage is named Redpaper_Forms_Sample_v10.xfdl.

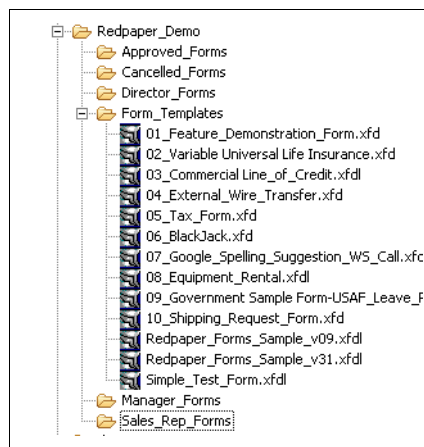


Figure 4-48 Same forms folder inside the project structure

This folder will make up the file storage in the directory `<WASRoot>/installedapplications/<applicationName>` on the application server. The real paths used for storage will differ in the production system and the test server running in RAD6 IDE. The code provided here is valid for both cases, but the code provided in the JSPs to display the file list and generate appropriate links will contain some if/then statements to match both environments. For details on this topic, see the next sub-chapter.

4.8.5 Servlet interaction for forms processing

The entry point of the application is index1.jsp (Call in the browser *http://servername:portname/WPFormsRedpaper/index1.jsp* to open the JSP. Clicking the **New Order** button, dirlisting1.jsp is activated and the list of available templates shows up. This is the starting point in the Stage 1 form processing scenario. (See Figure 4-49.)

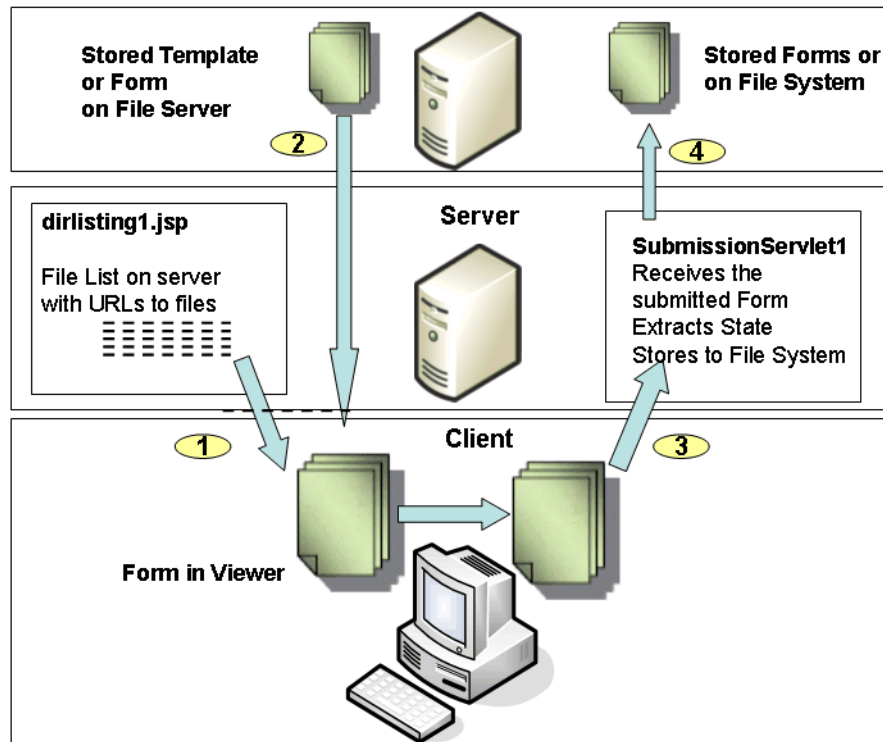


Figure 4-49 Stage 1 form processing scenario

Figure 4-49 shows a diagram of a basic form lifecycle:

1. dirlisting1.jsp shows a list of all available forms in a file system directory. The directory shown depends on the task to proceed and in some cases on the employee status selected in the welcome page:
 - New Order → template directory is shown.
 - Work Basket / Employee: Directory with draft submissions is shown.
 - Work Basket / Manager: Directory with submissions for manager approval is shown.
 - Work Basket / Director: Directory with submissions for director approval is shown.
 - Approved: Directory with finally approved forms is shown.
 - Canceled: Directory with cancelled forms is shown.
2. Clicking one of the links in the file list, a URL is generated that retrieves the selected form from the file server. In this stage the application server does not access the form at all on form load.
3. After working on the form, the client submits it using a URL stored in the form. It points to the SubmissionServlet1 and activates there the doPost method. The method receives the form, extracts some basic values (mainly the form name and state) and creates a form name, if the form does not contain a valid name.

The `SubmissionServlet1 / doPost` method creates a file name based on the `formName` and stores the form to the file system. The folder to store is calculated based on the form state.

So, as we see, in this stage the servlet does not really interact with the template or form on opening time — just the called JSP exposes a URL enabling the browser to get the template or form from the file system via the HTTP server. The real interaction takes place when a form is submitted. Before we can code it, let us have a short look at how to use the Workplace Forms API.

4.8.6 Accessing a form through the Workplace Forms API

The IBM Workplace Forms Server - Application Programmer Interface (API) consists of a collection of programming tools to help you develop applications that can interact with XFDL forms. These tools are available for both C and Java programming environments. An API for COM interface is available as well. The API enables you to access and manipulate forms as structured data types.

The API is divided into two libraries: the Form Library and the Function Call Interface (FCI) Library. The Form Library allows you to create applications that:

- ▶ Read and write forms.
- ▶ Retrieve information from form elements.
- ▶ Add cells to certain form items.
- ▶ Insert information into form elements.

This is the part of the API (namely, the Form Library) that we will focus on in this chapter.

The Function Call Interface (FCI) Library provides additional methods that:

- ▶ Create, duplicate, or delete form elements.
- ▶ Manipulate and verify digital signatures.
- ▶ Handle attachments.
- ▶ Create custom functions for use within XFDL forms.

For extensive, detailed information about the FCI, refer to the Workplace Forms Servers API documentation (C and Java versions):

<http://www-128.ibm.com/developerworks/workplace/documentation/forms/>

The idea of forms API is to open a form from stream or file system as a complex tree structure and present a huge number of methods to navigate in the form, read and write data or read / alter the form structure adding, changing or deleting nodes.

In this chapter we will focus on some basic methods to read/write data. For full information about the Forms API, see the product documentation available at the following URL:

<http://www-128.ibm.com/developerworks/workplace/documentation/forms/>

Refer to the following files:

- ▶ *WPForms API Setup Guide (22914850.pdf)*
- ▶ *WPForms API Java User Manual (22914870.pdf)*
- ▶ *WPForms API C User Manual (22914860.pdf)*
- ▶ *WPForms API COM User Manual (22914880.pdf)*

Accessing form data, in most cases, is done in three steps:

1. Opening the form as `formNodeP` object. This step will give us the root node of the form as starting point to other actions performed on the node structure.
2. Navigation to the data or structure object we are interested in (the field, the XFDL data instance, a button, an included WSDL file, an attribute, or any other node available in the form).
3. Access to the object (read / write / delete).

For each of these tasks, there are multiple methods available. We highly recommend that you consider the notes attached to function descriptions in the manual, since similar navigation and data access methods will have different side effects (such as creating new nodes, if the requested node does not exist, or just to raise an error in the same case). Be aware that any changed node structure or data in signed areas of a form will break the signature.

A basic API program (here shown as a servlet) accessing a form available as a file will look like Example 4-10. (However, please do *not* enter this code into your sample project. This code is for discussion purposes only).

Example 4-10 Simple sample servlet using XFDL API

```
import java.io.*;
//servlet support
import javax.servlet.*;
import javax.servlet.http.*;

//WPForms API references
import com.PureEdge.DTK;
import com.PureEdge.xfdl.FormNodeP;
import com.PureEdge.xfdl.XFDL;
import com.PureEdge.IFSSingleton;

public class ProcessXFDL extends HttpServlet {

    private static FormNodeP theForm;

    //in do post we can read the submitted xfdl file as stream
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {

        ServletOutputStream out = response.getOutputStream();
        ServletInputStream theStream = request.getInputStream();

        try {
            DTK.initialize("DominoIntegration", "1.0.0", "6.5.0");

            XFDL theXFDL;
            //initialize the API
            theXFDL = IFSSingleton.getXFDL();

            if (theXFDL == null)
                throw new Exception("Could not find interface");

            //get access to the form from stream
            theForm = theXFDL.readForm(theStream, XFDL.UFL_SERVER_SPEED_FLAGS);

            //if the form is available as a file, the following code would fit:
            //theForm = theXFDL.readForm( "myPath/myfile.xfdl", 0);
```

```

//now we can work with the form, e.g. retrieve a value of FIELD1 on
// PAGE1
String temp = theForm.getLiteralByRefEx(null, "PAGE1.FIELD1.value",
    0, null, null);

//or get a data instance like this
theForm.extractInstance("formData", null, null, "mypath/exp.xml",
    0, null, null, null);

//or set some additional fields
theForm.setLiteralByRefEx(null, "PAGE1.FIELD2.value", 0, null,
    null, "new value");

//we can save the form to file system or stream
theForm.writeForm("mypath/mySavedForm.xfdl", null, 0);

//finally we should free up memory
theForm.destroy();

} catch (Exception ex) {
    ex.printStackTrace();
}
} // end of method Post
} // end of class

```

Attention: Actually, there is a limitation to initialize the Forms API only once on a server. As a best practice, create a library containing the initialization statement and avoid subsequent initializations in the applications using the API.

Once having opened the XFDL file or stream, we accessed the form in two different ways in this example:

- ▶ Accessing nodes directly
- ▶ Writing data to XFDL data instances

In this example we used the methods `getLiteralByRefEx` / `setLiteralByRefEx` assigning a text value to a node. These functions cannot be used to assign complete XML trees, since the assigned text is automatically rendered in XML conform encoding — for example, changing `<` to `<`.

Accessing nodes directly can read/alter/delete any node. Using the API methods `getLiteralByRefEx` / `setLiteralByRefEx` as shown in this example, we would create new nodes in case the referenced node does not exist. To react in any other way on missing nodes, we can use code pieces as shown in Example 4-11.

Example 4-11 Sample coding for missing nodes

```

if ((tempNode = theForm.dereferenceEx(null, "PAGE1.COLORFIELD", 0,
    FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
{
    throw new UWException("Could not locate COLORLABEL node.");
}
tempNode.setLiteralByRefEx(null, "PAGE1.COLORFIELD.VALUE", 0,
    null, null, "Purple");

```

Both ways work fine accessing “real” field values (as shown in the examples) or other field properties addressable “by name”. We will use these methods for data extraction from data instances as well, but there are some limitations, as we will see when creating the code.

4.8.7 Extraction of form data

For form data extraction, basically three methods are available:

- ▶ Extracting single values using API
- ▶ Extracting complete data instances using API
- ▶ Access to any stored information using text parsers

There are different insertion points for the extracting code, depending on the way the form is submitted. The Forms Viewer has three basic methods to return a form:

- ▶ Store on a file system
- ▶ Submit as a mail attachment
- ▶ Submit as an HTTP POST action

Storing requests to the file system or submitting by mail is out of scope of this redbook. Nevertheless, the techniques shown here can be applied to both scenarios.

The form will make sure in our case to submit the completed form in a POST action. Back to the application scenario, we are ready now to code parts of the doPost method responsible to receive the submitted form and extract some data. There we can get the submitted form, retrieve data, and store it in the database. The following code is a short example of how to access the form data.

First we will implement some helper methods in the class:

- ▶ `returnText` — composes a simple HTML file as an error/message page sent to the browser
- ▶ `getFormAsString` — returns the form as string; useful for debugging
- ▶ `getFormBytes` — returns a form as a byte array ready to store in a file
- ▶ `returnJSP` — opens a named JSP in the browser (the method used for success action and for application navigation later on)

The coding is shown in Example 4-12.

Example 4-12 Helper methods for form handling in doPost

```
//return any error messages to web client (helper for the time being until we have an
error.jsp)
private void returnText(HttpServletResponse response, String outputString,
    String mimeType) throws IOException {
    //Set Headers so that the response is not cached
    response.setStatus(HttpServletResponse.SC_OK);
    response.setHeader("Pragma", "No-cache");
    response.setHeader("Cache-Control", "no-cache");
    response.setDateHeader("Expires", new Date().getTime());
    response.setHeader("Expires-Absolute", "Thu, 01 Dec 1994 16:00:00 GMT");
    response.setContentType(mimeType);

    System.out
        .println("SubmissionServlet: writing formString to response OutputStream");
    PrintWriter pw = new PrintWriter(response.getOutputStream());
    pw.write(outputString);
    pw.flush();
    pw.close();
}

//read form as String
private static String getFormAsString(FormNodeP theForm)
    throws UWIException, IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```

        theForm.writeForm(baos, null, 0);
        baos.flush();
        return baos.toString();
    }
    //read the form as byte[]
    private byte[] getFormBytes(FormNodeP theForm) throws UWIException,
        IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        theForm.writeForm(baos, null, 0);
        baos.flush();
        return baos.toByteArray();
    }
    //proxy method calling any jsp identified by jsp file name jspName
    private void returnJSP(HttpServletRequest request,
        HttpServletResponse response, String jspName)
        throws ServletException, IOException {
        //Return response page
        response.setStatus(HttpServletResponse.SC_OK);
        response.setHeader("Pragma", "No-cache");
        response.setHeader("Cache-Control", "no-cache");
        response.setDateHeader("Expires", new Date().getTime());
        response.setHeader("Expires-Absolute", "Thu, 01 Dec 1994 16:00:00 GMT");
        RequestDispatcher view = request.getRequestDispatcher(jspName);
        view.forward(request, response);
        System.out.println("CMSSubmissionServlet: returning response JSP");
    }
}

```

Now create the code for the doPost method (Example 4-13). The basic flow is as follows:

1. Detect the operation evaluating the action parameter.
2. Access the form with the API.
3. Process the action.
 - a. Bounceback for debug, or
 - b. Normal operation: extract values.
4. Send a response to the browser.

Example 4-13 Form handling in doPost - value extraction

```

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    System.out.println("SubmissionServlet: doPost started");

    //Initialize member variables
    String action = null; //Controls the processing action, response from
    // Servlet
    XFDL theXFDL = null; //The form
    FormNodeP theForm = null; //Represents nodes of the XFDL form
    String formString = null; //String representation of the form. Used for
    // the 'bounceback' feature.

    //Form State variables
    String formState = null; //The current state of the form
    String previousFormState = null; //The previous state of the form
    String formName = null; //Used in Stage 1 as the file name when
    // persisting the form to the file system

    /**
     * Processing. <BR>

```

```

* Determine the type of request. Currently supported options include:
* [store] Store form, display 'submissionComplete' JSP. If needed,
* the previous version is removed. <BR>
* [bounceback] For test purposes, returns the form as an text/xml
* response page. <BR>
*/
try {
    action = request.getParameter("action");
    if (action.equalsIgnoreCase("bounceback")) {
        formString = getFormAsString(theForm);
    } else if (action.equalsIgnoreCase("store")) {
        System.out
            .println("SubmissionServlet: detected -->store<-- request.");
        action = "store";

        /**
         * Read in the form
         */
        System.out.println("SubmissionServlet: doPost: reading Form");
        theXFDL = IFSSingleton.getXFDL();
        theForm = theXFDL.readForm(request.getInputStream(),
            XFDL.UFL_SERVER_SPEED_FLAGS);

        /**
         * OK _ HERE WE SHOULD DO SIGNATURE VALIDATION
         */

        /**
         * Extract the form State. The following commented-out lines are
         * examples of a variety of valid ways of extracting the
         * FormMetaData instance element State. In this case we will use
         * an explicit reference to the element rather than a positional
         * reference. This allows us to add data instances
         */

        formState = theForm.getLiteralByRefEx(null,
            "global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:State]", 0, null, null);
        previousFormState = theForm.getLiteralByRefEx(null,
            "global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:PreviousState]",
            0, null, null);
        System.out.println("SubmissionServlet: doPost: FormState: " + formState);
        System.out.println("SubmissionServlet: doPost: PreviousFormState by
getLiteralByRefEx: " + previousFormState);
        //If this is the first submission of a form, set a new filename
        // into the form
        formName = theForm.getLiteralByRefEx(null,
            "global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:FileName]",
            0, null, null);
        if (formName == null)
            formName = "";
        if (formName.equals("")) {
            formName = "" + System.currentTimeMillis();
            theForm.setLiteralByRefEx(null,
                "global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:FileName]",
                0, null, null, formName);
        }

        formName = FORM_NAME_PREFIX + formName + ".xfd1";
        System.out.println("SubmissionServlet: doPost: formName : " + formName);
    }
}

```

```

//*****
// OK _ HERE WE COULD STORE THE FORM - E.G. TO FILE SYSTEM
//*****

}

if (theForm != null) {
    System.out.println("SubmissionServlet: doPost: calling destroy() on theForm");
    theForm.destroy();
}
} catch (Exception processingE) {
    System.out.println("SubmissionServlet: doPost: Exception processing request: "
        + processingE.toString());
    returnText(response, "SubmissionServlet: doPost: Exception occurred: "
        + processingE.toString(), "text/plain");
    return;
}

/**
 * Return the appropriate response based on the action variable. <BR>
 */
try {
    //Switch based on the specified action
    if (action.equalsIgnoreCase("store")) {
        returnJSP(request, response, "/success1.jsp");
    } else {
        throw new Exception(
            "SubmissionServlet: unexpected state, taking no action");
    }
} catch (Exception anE) {
    try {
        if (theForm != null) {
            theForm.destroy();
        }
    } catch (Exception anotherE) {
        System.out.println("SubmissionServlet: Nested Exception: "
            + anotherE.toString());
    }
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter(response.getOutputStream());
    out.write(anE.toString());
    out.flush();
    out.close();
    System.out.println("SubmissionServlet: Exception: "
        + anE.toString());
}
System.out.println("SubmissionServlet: doPost: completed.");
}
}

```

In the middle section we see several references to read data from the form using the `getLiteralByRefEx` method. It is a best practice to access the form via XFDL instances as a stable interface to the forms values rather than to access form fields directly. This is a much more stable interface than fields in a form, since development can simply change the field location (moving a field on another page) or change the internal data model in any other way.

The reference to the extracted data item that we used is written like this:

```
global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:FileName]
```

The reference points to a particular data instance, but we could not make this run by assigning the data instance by name. The cause is, data instances are created using the same tag (<instance>) and the differentiation is made using an ID attribute (Example 4-14).

Example 4-14 XFDL structure with the data instance FormMetaData / field FileName to prepopulate

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XFDL xmlns="http://www.PureEdge.com/XFDL/6.5" ..... >
  <globalpage sid="global">
    <global sid="global">
      .....
      <xmlmodel>
        <instances>
          ....
          <xforms:instance xmlns="" id="FormMetaData">
            <FormMetaData>
              <FileName></FileName>
              .....
            </FormMetaData>
          </xforms:instance>
          ....
        </xmlmodel>
      </global>
    </globalpage>
```

Here we can find the problem to address elements in data instances by name (see path `global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:FileName]` in the code above using a reference by position. This is a better approach than accessing the field directly, but can break when other instances are created or deleted. To overcome this, the method `extractInstance` could be used. For an example, see Chapter 9, “Domino integration” on page 273.

Having the values of interest extracted (for the first stage, these are state, previousState, and formName), we can go on to do some more sophisticated actions and store the form.

4.8.8 Signature validation

Another interesting action when receiving a form is signature checking. Insert the following code in the place with the signature comment (Example 4-15).

Example 4-15 Sample coding for signature checking

```
/**
 * Validate form signatures. If any signatures are invalid, then
 * return. TODO: Add a JSP response that indicates tampering.
 */
if (!allSignaturesAreValid(theForm)) {
    System.out
        .println("SubmissionServlet: doPost: WARNING -- signatures were
invalid");
    returnText(response,"Error, one or more signatures were invalid!! Form
submission processing halted.", "text/plain");
    theForm.destroy();
    return;
} else {
    System.out.println("SubmissionServlet: doPost: validation OK");
}
```

The code is supported by an additional helper method to insert in the servlet class - it is pretty simple to detect any changes to signed items (Example 4-16).

Example 4-16 Signature verification using Forms API

```
private static boolean allSignaturesAreValid(FormNodeP theForm)
    throws Exception {
    return (theForm.verifyAllSignatures(false) == FormNodeP.UFL_SIGS_OK);
}
```

There are other methods available to check validity of a single signature (`verifySignature`) or even to read signature validity status on the last signature check to track newly occurred violations (`getSignatureVerificationStatus`).

Attention: There is no method to detect which item was changed in the form with a broken signature, since a signature is basically a hash code including all items to sign in a step. So the validation will fail, if any item is changed.

4.8.9 Form storage to local file system

Having all tasks completed, we can store the file to file system. The code provided for now already created a file name (`formName` variable). The code to add would perform the following actions:

- ▶ Evaluate the form state to determine the target directory to store the form.
- ▶ Store the form.
- ▶ Remove earlier versions of this form in any other directory (we just want to keep the actual form only).

Tip: It is not always necessary to store the form. In some cases it can be the right way to extract only some values and store them to a back-end system.

We have already created the file system structure for storage — so we can use it now. Insert the following code, as shown in Example 4-17, at the right place in the `doPost` method (see the comment for file storage there).

Example 4-17 Status indication, target folder calculation, file storage and clean-up

```
/**
 * Store the form into the folder indicated by the formState.
 */
String folderPath = props.getProperty(formState);

ServletContext ctx = conf.getServletContext();
String path = ctx.getRealPath(folderPath);

writeBytesToFile(path + formName, getFormBytes(theForm));

/**
 * Remove the previous instance of the form. *STAGE 1*
 */
if (previousFormState == null) {
    System.out.println("SubmissionServlet: doPost: FormMetaData: ERROR:
PreviousFormState element was null");
} else if (previousFormState.equalsIgnoreCase("2")
|| previousFormState.equalsIgnoreCase("3"))
```

```

        || previousFormState.equalsIgnoreCase("5")) {
    System.out
        .println("SubmissionServlet: doPost: FormMetaData: previousFormState ["+
            previousFormState+ "] indicates previous file deletion is necessary");
    props.getProperty(previousFormState);
    System.out.println("SubmissionServlet: doPost: FormMetaData:
PreviousFormState element contained value: "
        + previousFormState+ ", which does not indicate deletion.");
    }
}

```

As you can see, the provided code contains the main part of the business logic (storing files according to state). The rest of the application logic is contained in the JSPs offering all application navigation mainly dealing with the doGet method calling the servlet URL with different action parameters attached.

4.8.10 Servlet doGet method for application navigation

The doGet method in Stage 1 provides mainly assisting functions. All are not related to the Workplace Forms functionality. This will change in Stage 2, when form prepopulation comes into the game.

For now, we are doing only navigation support and user identity management here. To have an easy way for user identification, we decided not to implement a server security at this stage. Instead, we will provide an application based security model, that passes the employee ID to any calls (JSP or servlet) doGet method. It will be used to set this ID for the session to any of the available employee IDs.

Actually there are no new helper methods to implement, since we did all necessary steps for doPost in previous section. The servlet will accept the following action parameters:

- ▶ listTemplates — calls dirlisting1.jsp listing the templates directory.
- ▶ workbasket — calls dilisting1.jsp listing one of the folders for sales rep. forms, manager approval, or director approval (forms in state 1, 2 or 3) depending on the user role.
- ▶ listApproved — calls dilisting1.jsp listing approved forms (forms in state 4).
- ▶ listCancelled — calls dilisting1.jsp listing approved forms (forms in state 6).
- ▶ getJSP — navigates just to a dedicated JSP.

Each time the servlet activates a JSP, it provides the user context (user ID) and possible other application state data to it.

The doGet code is shown in Example 4-18.

Example 4-18 Servlet doGet method

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    try {
        /**
         * Obtain the user identity
         */
        //TODO: Replace this with real user id lookup
        String userID = null;
        if (request.getSession().getAttribute("userRole") != null) {
            userID = (String) request.getSession().getAttribute("userRole");
            System.out
                .println("SubmissionServlet: doGet: request parameter 'userRole' was
loaded from Session as: "

```

```

        + userID);
    } else {
        System.out
            .println("SubmissionServlet: doGet: request parameter 'userRole' was not
specified, using default value of 1000");
        userID = "1000";
    }
    request.setAttribute("userRole", userID);

    /**
     * Obtain the type of request from the key=value params and set the
     * necessary folder into the session
     */
    String action = request.getParameter("action");
    if (action == null || action.equalsIgnoreCase("listTemplates")) {
        System.out
            .println("SubmissionServlet: doGet: detected -->listTemplates<--
request.");
        request.setAttribute("FOLDER", TEMPLATE_FOLDER);
    } else if (action.equalsIgnoreCase("workbasket")) {
        System.out
            .println("SubmissionServlet: doGet: detected -->workbasket<--
request.");
        if (userID.equalsIgnoreCase("1010")
            || userID.equalsIgnoreCase("1020")
            || userID.equalsIgnoreCase("1030")) {
            request.setAttribute("FOLDER", MANAGER_FOLDER);
            System.out
                .println("SubmissionServlet: doGet: workbasket: setting folder to: "
                    + MANAGER_FOLDER);
        } else if (userID.equalsIgnoreCase("1031")) {
            request.setAttribute("FOLDER", DIRECTOR_FOLDER);
            System.out
                .println("SubmissionServlet: doGet: workbasket: setting folder to: "
                    + DIRECTOR_FOLDER);
        } else {
            request.setAttribute("FOLDER", SALES_REP_FOLDER);
            System.out
                .println("SubmissionServlet: doGet: workbasket: setting folder to: "
                    + SALES_REP_FOLDER);
        }
    } else if (action.equalsIgnoreCase("listApproved")) {
        System.out
            .println("SubmissionServlet: doGet: detected -->listApproved<--
request.");
        request.setAttribute("FOLDER", APPROVED_FOLDER);
    } else if (action.equalsIgnoreCase("getJSP")) {
        System.out
            .println("SubmissionServlet: doGet: detected -->getJSP<-- request.");
        String jsp = request.getParameter("jsp");
        if (jsp == null) {
            returnText(
                response,
                "SubmissionServlet: for getJSP, parameter jsp must not be null.",
                "text/plain");
        } else {
            System.out
                .println("SubmissionServlet: doGet: getJSP: jsp = "
                    + jsp);
            returnJSP(request, response, jsp);
        }
    }
}

```

```

        return;
    }
} else if (action.equalsIgnoreCase("listCancelled")) {
    System.out
        .println("SubmissionServlet: doGet: detected -->listCancelled<--
request.");
    request.setAttribute("FOLDER", CANCELLED_FOLDER);

} else if (action.equalsIgnoreCase("setRole")) {
    System.out
        .println("SubmissionServlet: doGet: detected -->setRole<-- request.");
    String userRole = request.getParameter("userRole");
    if (userRole == null) {
        System.out
            .println("SubmissionServlet: setRole: parameter userRole was null.
Defaulting to Employee role, 1000");
        request.getSession().setAttribute("userRole", "1000");
    } else {
        System.out
            .println("SubmissionServlet: setRole: parameter userRole was: "
                + userRole
                + ". Storing into session for later use.");
        request.getSession().setAttribute("userRole", userRole);
    }
    request.setAttribute("userRole", userRole);
    returnJSP(request, response, "/index1.jsp");
    return;
} else {
    System.out
        .println("SubmissionServlet: doGet: unexpected action param detected: "
            + action);
}

/**
 * Store the foldername into the session for use in the JSP -
 * initial, default state is 1
 */
returnJSP(request, response, "/dirlisting1.jsp");
} catch (Exception doGetE) {
    System.out
        .println("SubmissionServlet: doGet: Exception processing request: "
            + doGetE.toString());
    returnText(response,
        "SubmissionServlet: doGet: Exception occurred: "
            + doGetE.toString(), "text/plain");
}
}

```

This method is actually strongly related to the JSPs in the application.

4.9 Creating JSPs

In the following discussion, we show you how to create the JSPs.

4.9.1 Where we are in the process: Building Stage 1 of the base scenario

Figure 4-50 is intended to provide an overview of the key steps involved to build the base scenario. This focuses on building the Form, the Servlet, and the JSPs.

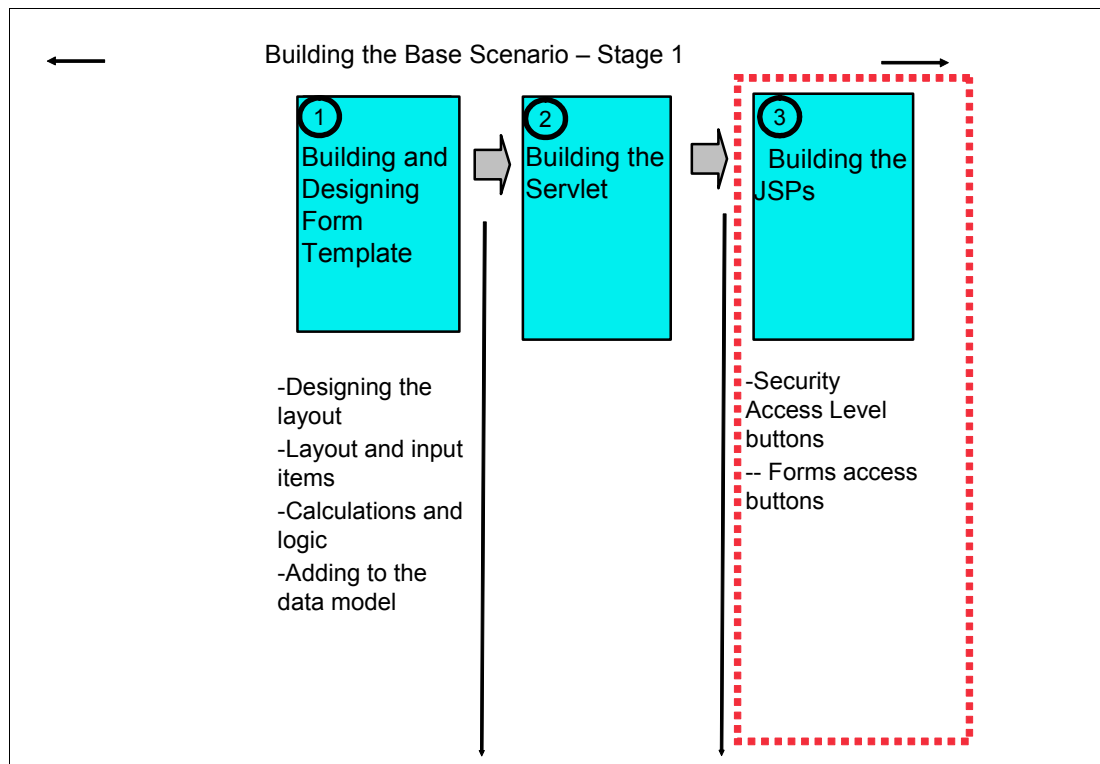


Figure 4-50 Overview of major steps involved in building the core base scenario application

Java Server Pages (JSP)s allow you to develop dynamic, content driven Web pages that separate the user interface (UI) from constantly changing data that may be generated from an external source. JSP tags can be combined with HTML to allow you to format the appearance of the content on the Web page.

For the example used in this stage of the Redbook, you will need to create 3 JSPs namely: *index1.jsp*, *dirlisting1.jsp* and *success1.jsp*. You can use any Web application development tool such as Rational® Application Developer (RAD), Eclipse or DreamWeaver in order to create these JSPs. All the JSPs contained here were created using RAD and subsequently all the screen-shots will have a RAD context.

index1.jsp

The *index1.jsp* is the main entry point and navigation page for the Web application that your sales team will use as a launching point to access all the forms that they need to initiate or complete a sale as well as to see all the data associated with their customers.

Employees, Managers, and Directors are able to see unique views to all the forms that are stored on the file system — new orders, any items in their own workbasket that require review and approval, all approved forms, and all forms that have been rejected or cancelled.

Figure 4-51 shows what the final page is going to look like. You can change the HTML layout, stylesheet, and graphics to suit your own taste. The important functionality, however, is contained in the buttons that you see on the page.

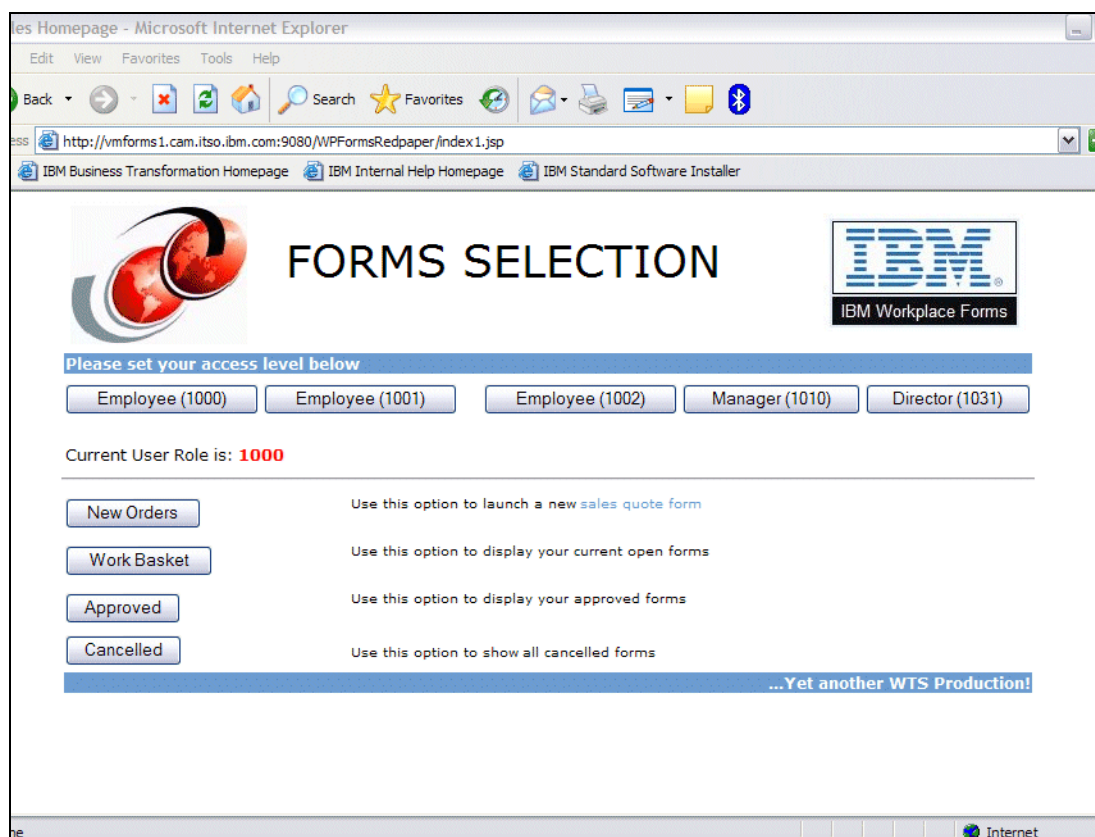


Figure 4-51 index1.jsp

Important: In order to run this Web application on WebSphere Application Server correctly, you will need to save all the JSPs to the root **WebContent** directory and not in the **WEB-INF** directory. All graphics and cascaded style-sheets should be stored in a directory called **themes**, also in the root.

1. Using your Web application development tool create a new JSP file, name it *index1.jsp*, and save it to the root *WebContent* directory (Figure 4-52).

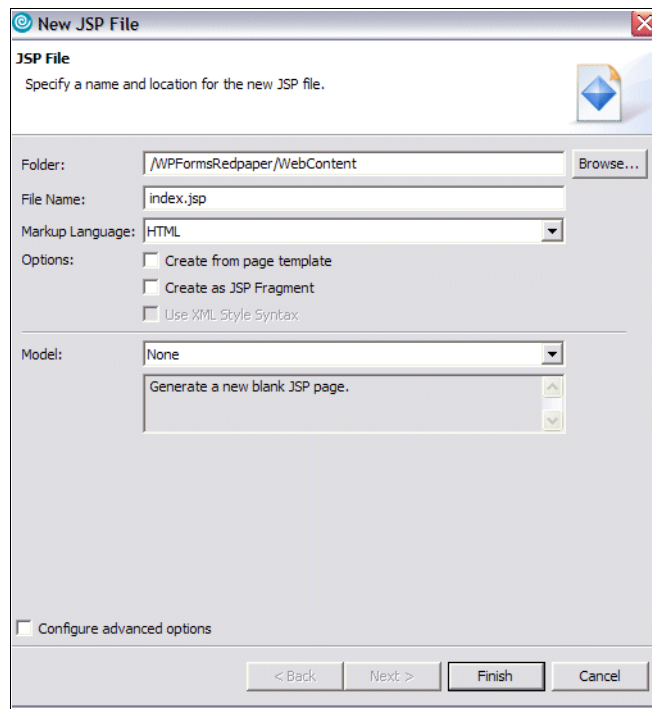


Figure 4-52 New JSP creation

2. In the <HEAD> section of the JSP file, give the path to the cascaded style-sheet you will use for the page in the <LINK href="theme/..."> section and enter the <TITLE> as *Sales Homepage* as shown in Example 4-19.

Example 4-19 Coding example (<TITLE> = Sales Homepage)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM Software Development Platform">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/blue.css" rel="stylesheet" type="text/css">
<TITLE>Sales Homepage</TITLE>
</HEAD>
```

3. Next, in the <BODY> section, you will create some tables to display:
 - a. Logo graphics
 - b. Security access level buttons
 - c. Form access buttons

Logo graphics

In Example 4-20 we simply add a table with three columns, with a logo graphic on the left, a header in the middle, and a graphic on the right. Note that all the graphics are contained in the folder called *theme*.

Example 4-20 Logo graphics

```
<BODY>
<!-- Table to display the logo graphics -->
<CENTER>
<TABLE border="0" cellpadding="2" width="760">
  <TBODY>
    <TR>
      <TD><IMG border="0" src="theme/redbook_logo.jpg" width="138"
        height="114" align="left"></TD>
      <TD align="left">
        <H1>FORMS SELECTION</H1>
      </TD>
      <TD><IMG border="0" src="theme/Workplace_Forms.jpg" width="158"
        height="92" align="right"></TD>
    </TR>
  </TBODY>
</TABLE>
</CENTER>
```

Security Access Level buttons

This table (see Example 4-21) contains important security access level information in buttons that are to be submitted to the servlet. These in turn will control the information that is returned and displayed from the other JSPs, which are described later.

Each button has a *userRole* value, when clicked, submits an action that sets the user credentials in the session and passes this to the servlet. The users from 1000 - 1002 have the role of *employee*, user 1010 has the role of *manager*, and user 1031 has the role of *director*.

We also use the user ID number to prepopulate the form with some metadata about the employee from the DB2 database. This includes: First Name, Last Name, Personnel Number, E-mail Address, and the Manager's ID.

Attention: We decided to implement security in this way, rather than by using a separate login page, to make the demonstration flow more easily. In a real-world scenario, all authentication should pass through a login page that would pass the user credentials to the servlet.

Example 4-21 Security Access Level buttons

```
<!-- Table to Display Access Level Selection Buttons -->
<CENTER>
<%@ page session="false" contentType="text/html"
  import="java.util.*,java.io.File"%>
<%
  String userID = (String) request.getAttribute("userRole");
  if (userID== null) userID="1000";
  if (userID.equals("")) userID="1000";
  %>

<TABLE width="760">
```

```

<TR>
  <TD colspan="3">

    <TABLE>
    <TR>
    <TD>
      <FORM method=get action="SubmissionServlet">
      <INPUT type="submit" value="Employee (1000)">
      <INPUT type="hidden" name=action value="setRole">
      <INPUT type="hidden" name=userRole value="1000">
      </FORM>
    </TD>
      <TD width="304">
        <FORM method=get action="SubmissionServlet">
        <INPUT type="submit" value="Employee (1001)">
        <INPUT type="hidden" name=action value="setRole">
        <INPUT type="hidden" name=userRole value="1001">
        </FORM>
      </TD>
      <TD>
        <FORM method=get action="SubmissionServlet">
        <INPUT type="submit" value="Employee (1002)">
        <INPUT type="hidden" name=action value="setRole">
        <INPUT type="hidden" name=userRole value="1002">
        </FORM>
      </TD>
      <TD>
        <FORM method=get action="SubmissionServlet">
        <INPUT type="submit" value="Manager (1010)">
        <INPUT type="hidden" name=action value="setRole">
        <INPUT type="hidden" name=userRole value="1010">
        </FORM>
      </TD>
      <TD>
        <FORM method=get action="SubmissionServlet">
        <INPUT type="submit" value="Director (1031)">
        <INPUT type="hidden" name=action value="setRole">
        <INPUT type="hidden" name=userRole value="1031">
        </FORM>
      </TD>
    </TR>
    <TR>
      <TD colspan="3"><%= "Current User Role is: <STRONG>" + userID + "</STRONG>"
%></TD></TR>
  </TABLE>
</CENTER>

```

Forms Access buttons

The behavior of Forms Access buttons is controlled by the `userRole` values that were set in the previous step above. If a user does not set their security access role, then the default role that is set is *employee*. Once the `userID` has been set and passed to the *dirlisting.jsp* by the servlet, the data that is returned when the user clicks on any button is dependent on their access level. For example, clicking the **Workbasket** button will display only the forms that are specific to the `userID`.

There are four buttons that you need to create:

1. **New Orders:** This button submits an action with a value of *listTemplates* to the SubmissionServlet, which returns all the new forms stored in a directory on the server file system named *Form_Templates*.
2. **Workbasket:** This button submits an action with a value of *workbasket* and passes in the userID to the SubmissionServlet. This returns only the forms that the user has created and submitted. These forms are stored in a directory on the server file system named *Sales_Rep_Forms*.
3. **Approved:** This button submits an action with a value of *listApproved* to the servlet which returns all the forms in the *Approved_Forms* directory that have been approved by the manager or director role. All forms that have a value over \$10,000.00 require manager approval, and all forms over \$50,000.00 require director approval.
4. **Cancelled:** This button submits an action with a value of *cancelled* to the servlet which returns all the new forms stored in a directory on the server file system named *Cancelled_Forms*. These are the forms that have not been approved by the manager or director.

We show you how to create these buttons in Example 4-22.

Example 4-22 Forms selection

```
<!-- Table to display the Forms Selection -->
<hr>
<TABLE width="760" align="center">
  <TR>
    <TD>
      <FORM method="get"
        action="/WPFormsRedpaper/SubmissionServlet">
        <INPUT type="submit" value="New Orders">
        <INPUT type="hidden" name="action" value="listTemplates">
      </TD>
    <TD><FONT size="-2">Use this option to launch a new <A
      href="/WPFormsRedpaper/SubmissionServlet">sales quote form</A></FONT>
    </FORM>
    </TD>
  </TR>
  <TR>
    <TD>
      <FORM method="get"
        action="/WPFormsRedpaper/SubmissionServlet"><INPUT
        type="submit" value="Work Basket">
        <INPUT type="hidden" name="action" value="workbasket">
        <INPUT type="hidden" name="id" value="<%=userID%>">
      </TD>
    <TD><FONT size="-2">Use this option to display your current open forms</FONT>
    </FORM>
    </TD>
  </TR>
  <TR>
    <TD>
      <FORM method="get" action="SubmissionServlet"><INPUT
        type="submit" value="Approved">
        <INPUT type="hidden" name="action" value="listApproved">
      </TD>
    <TD><FONT size="-2">Use this option to display your approved forms</FONT>
    </FORM>
    </TD>
  </TR>
</TABLE>
```

```

        <TR>
        <TD>
        <FORM method=get action="SubmissionServlet">
        <INPUT type="submit" value="Cancelled">
        <INPUT type="hidden" name=action value="listCancelled">
        </TD>
        <TD><FONT size="-2">Use this option to show all cancelled forms</FONT>
        </TD>
        </FORM>
        </TD>
    </TR>
</TABLE>

</FORM>

```

The JSP shown above (see Example 4-22) creates several actions calling the `SubmissionServlet` with additional parameters. To create the appropriate links, we created for each button a separate HTML form element containing the necessary parameters. Each input element with type *hidden* will create an parameter in the URL. The sample below (Example 4-23) would create a URL like this:

```
http://servername/WPFormsRedpaper/SubmissionServlet?action=action_param_val&param_1_name=add_param_1_val&...&param_n_name=add_param_n_val
```

Example 4-23 Creating a GET action with additional parameters

```

<FORM method="get" action="/WPFormsRedpaper/SubmissionServlet">
    <INPUT type="submit" value="[Button Label]">
    <INPUT type="hidden" name=action value="[action_param_val]">
    <INPUT type="hidden" name=[param_1_name] value="[add_param_1_val]">
    ....
    <INPUT type="hidden" name=[param_n_name] value="[add_param_n_val]">
    <FONT size="-2">#button description#
    <A href="/WPFormsRedpaper/SubmissionServlet">sales quote form</A>
    </FONT>
</FORM>

```

dirlisting1.jsp

The *dirlisting1.jsp* is used to display all the forms that are currently stored on the file system of the server and it is launched by the servlet. Depending on the button that is clicked on the *index1.jsp* and which security role is set, the *dirlisting1.jsp* will dynamically update the forms listed in the folder on the file system for that user. This JSP looks specifically for forms in six folders on the server's file system, located in the following directory:

```
C:\AppServer\installedApps\vmforms1\WPFormsRedpaper_war.ear\WPFormsRedpaper.war\Redpaper_Demo\:
```

These are the six folders:

- ▶ Form_Templates
- ▶ Sales_Rep_Forms
- ▶ Approved_Forms
- ▶ Cancelled_Forms
- ▶ Manager_Forms
- ▶ Director_Forms

Here are the steps that you should follow to create this JSP:

1. Create a new JSP named *dirlisting1.jsp*, and save it to the root **WebContent** directory
2. The contents of the JSP should be as shown in Example 4-24.

Example 4-24 dirlisting1.jsp provides links to the available files in an assigned folder in file system

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta http-equiv="Content-Style-Type" content="text/css">
<link rel="stylesheet" href="theme/blue.css" type="text/css">
<title>IBM Workplace Forms Selection</title>
</head>

<CENTER>
<TABLE border="0" cellpadding="2" width="760">
  <TBODY>
    <TR>
      <TD><IMG border="0" src="theme/redbook_logo.jpg" width="138"
        height="114" align="left"></TD>
      <TD align="left">
        <H1>FORMS SELECTION</H1>
      </TD>
      <TD><IMG border="0" src="theme/Workplace_Forms.jpg" width="158"
        height="92" align="right"></TD>
    </TR>
  </TBODY>
</TABLE>
</CENTER>

<%@ page session="false" contentType="text/html"
  import="java.util.*,java.io.File"%>

<%String theDir = (String) request.getAttribute("FOLDER");
  String prepop = (String) request.getAttribute("PrePop");
  String userID = (String) request.getAttribute("userRole");
  //prepop will control link behavior
  if (prepop == null)
    prepop = "No";
    prepop = "Yes";
  //The path to the current template, including the template name
  String servletPath = application.getRealPath(request
    .getServletPath());

  //Remove subdirs including WEB-INF
  System.out.println("servletPath: " + servletPath);
  String projectName = "WPFormsRedpaper";
  String path = servletPath;
  String linkPath = "";
  if (path.indexOf(java.io.File.separator + "WebContent"
    + java.io.File.separator) > 0) {
    path = path.substring(0, path.lastIndexOf("WebContent") - 1);
    path = path + java.io.File.separator + "WebContent";
    linkPath = path.substring(path.indexOf(projectName) - 1, path
      .lastIndexOf(java.io.File.separator) - 1);
    linkPath = "/" + projectName + theDir;
  } else {
    path = path.substring(0, path
      .lastIndexOf(java.io.File.separator));
```



```

</TR>
</TABLE>

<TABLE width="760" align="center">
  <tr bgcolor="#699ccf">
    <td align="right"><B>...Yet another WTS Production!</B></td>
  </tr>
</TABLE>

```

Adaptive handling of absolute and relative paths to stored XFDL forms

A basic concept of this JSP is the adaptive handling of absolute and relative paths to the stored XFDL forms in both the runtime test environment in RAD6 and the production environment in the application server directory structure.

The JSP code first detects the path to the servlet rendered from the JSP (Variable `servletPath`). If this path contains the folder `WebContent`, we are in the test environment, if not, we are in the deployed Web application. According to the environment detection, the code creates a variable containing the path to the project directory (variable `path`). Next it computes the relative path to the template (`linkPath`) used in the action retrieving the template from the server and the absolute path to the file on file system (variable `templatePath`).

In case of prepopulation (`prepop = "Yes"`), the original link to the template file is replaced with a URL calling the `SubmissionServlet` with the appropriate parameters (`action= ...`, `template=`)

Created links

Table 4-4 illustrates the links generated within `dirlisting1.jsp`.

Table 4-4 Examples for the generated actions in `dirlisting1.jsp` (Stage 1)

	Generated URL
RAD6 test environment	
New form	<code>http://localhost:9080/WPFormsRedpaper/Redpaper_Demo/Form_Templates/Redpaper_Forms_Sample_S2_v41.xfdl</code>
Open stored form	<code>http://localhost:9080/WPFormsRedpaper/Redpaper_Demo/Approved_Forms/Quote_Approval_Form1000143.xfdl</code>
Deployed application	
New form	<code>http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/Redpaper_Demo/Form_Templates/Redpaper_Forms_Sample_S2_v41.xfdl</code>
Open stored form	<code>http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/Redpaper_Demo/Approved_Forms/Quote_Approval_Form1000143.xfdl</code>

success1.jsp

This JSP is used to inform a user that their form has been successfully submitted. The servlet is responsible for launching this form once the user has gone through the steps of filling it out and submitting it for approval.

Figure 4-53 is a sample of what the *success1.jsp* looks like.

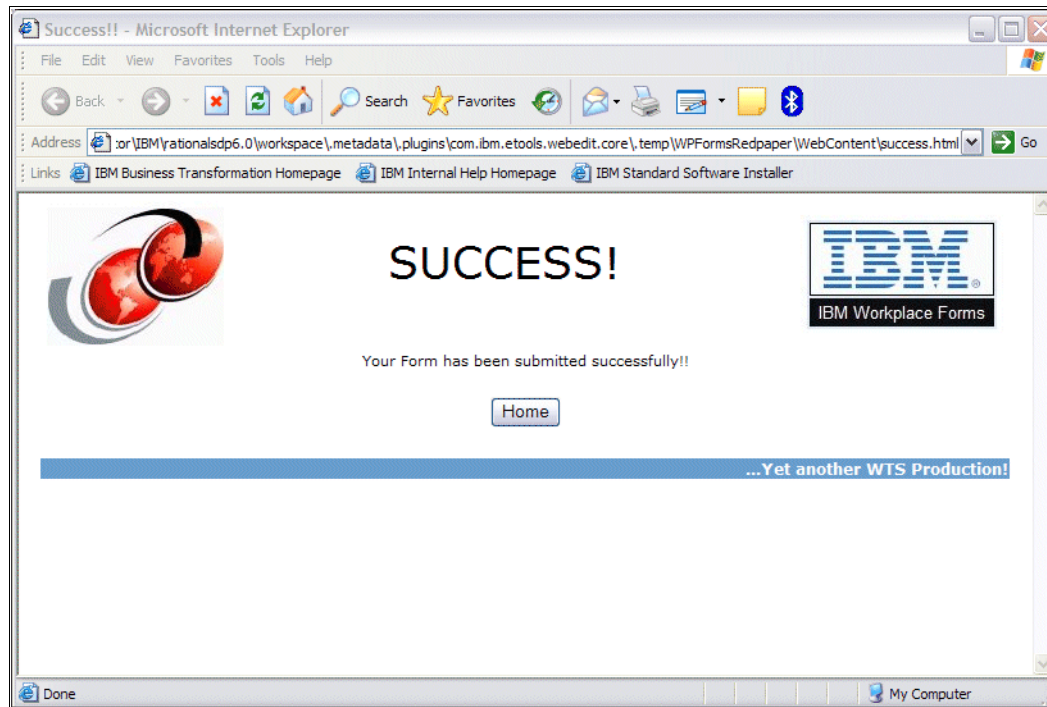


Figure 4-53 Rendering of success 1.jsp

Example 4-25 shows a sample of the code that can go into this JSP.

Example 4-25 success1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM Software Development Platform">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href=theme/blue.css" rel="stylesheet" type="text/css">
<TITLE>Success!!</TITLE>
</HEAD>
<BODY>
<CENTER>
<TABLE border="0" cellpadding="2" width="760" >
  <TBODY>
    <TR>
      <TD><IMG border="0" src="theme/redbook_logo.jpg" width="138"
        height="114" align="left"></TD>
      <TD align="left"><H1 align="center">SUCCESS!</H1></TD>
      <TD><IMG border="0" src="theme/Workplace_Forms.jpg" width="158"
        height="92" align="right"></TD>
    </TR>
  </TBODY>
</TABLE>
</CENTER>

<CENTER>
Your Form has been submitted successfully!!

</CENTER>
<P>
<TABLE align="center">
  <TR>

    <TD align="center">
      <FORM method=get action="SubmissionServlet">
        <INPUT type="submit" value="Home">
        <INPUT type="hidden" name=action value="getJSP">
        <INPUT type="hidden" name=jsp value="index1.jsp">
      </FORM>

    </TD>
  </TR>
</TABLE>

<TABLE width="760" align="center">
  <tr bgcolor="#699ccf">
    <td align="right"><B>...Yet another WTS Production!</B></td>
  </tr>
</TABLE>
```

4.9.2 Form template listing

Once a user has selected their role and then selected the button to create a new Sales Quote Order, the *dirlisting1.jsp* displays a list of the relevant form templates that they are authorized to use. The user can then click one of the links to launch a new form and begin the Sales Quote Order process.

Figure 4-54 shows a sample of all the forms listed in the Form_Templates folder from the file system that a user (employee) is able to see (1000).

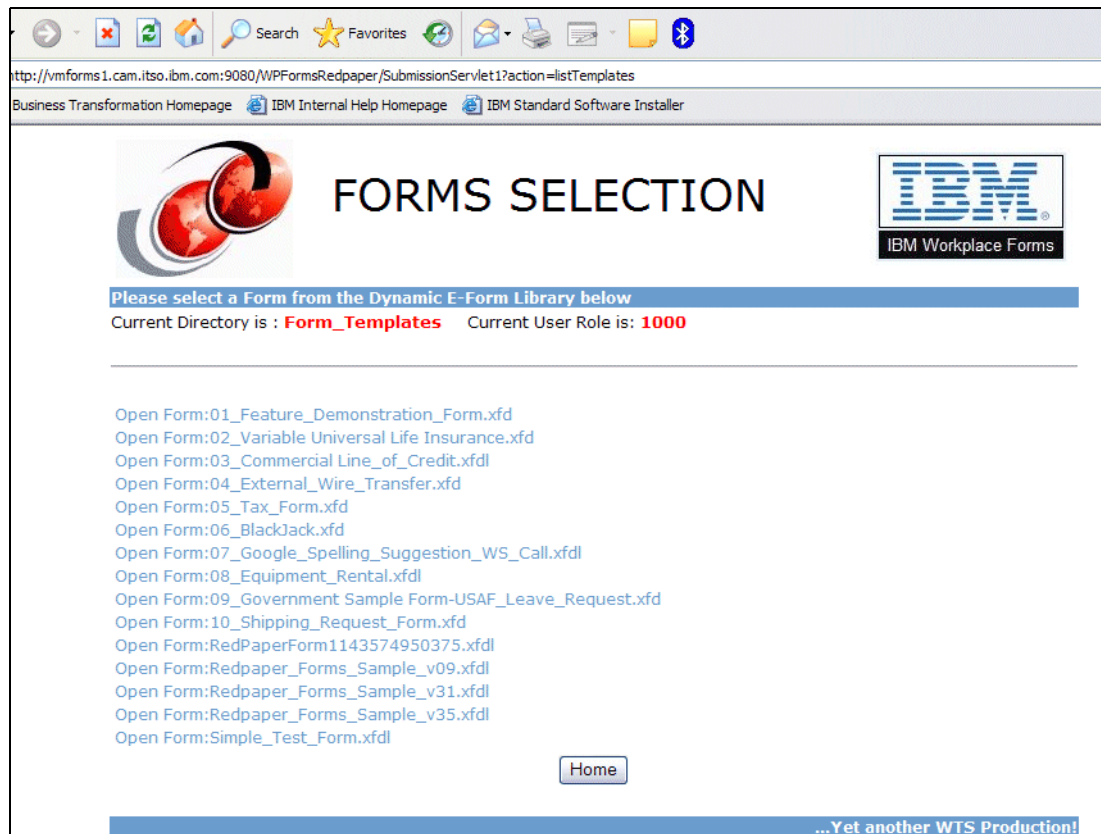
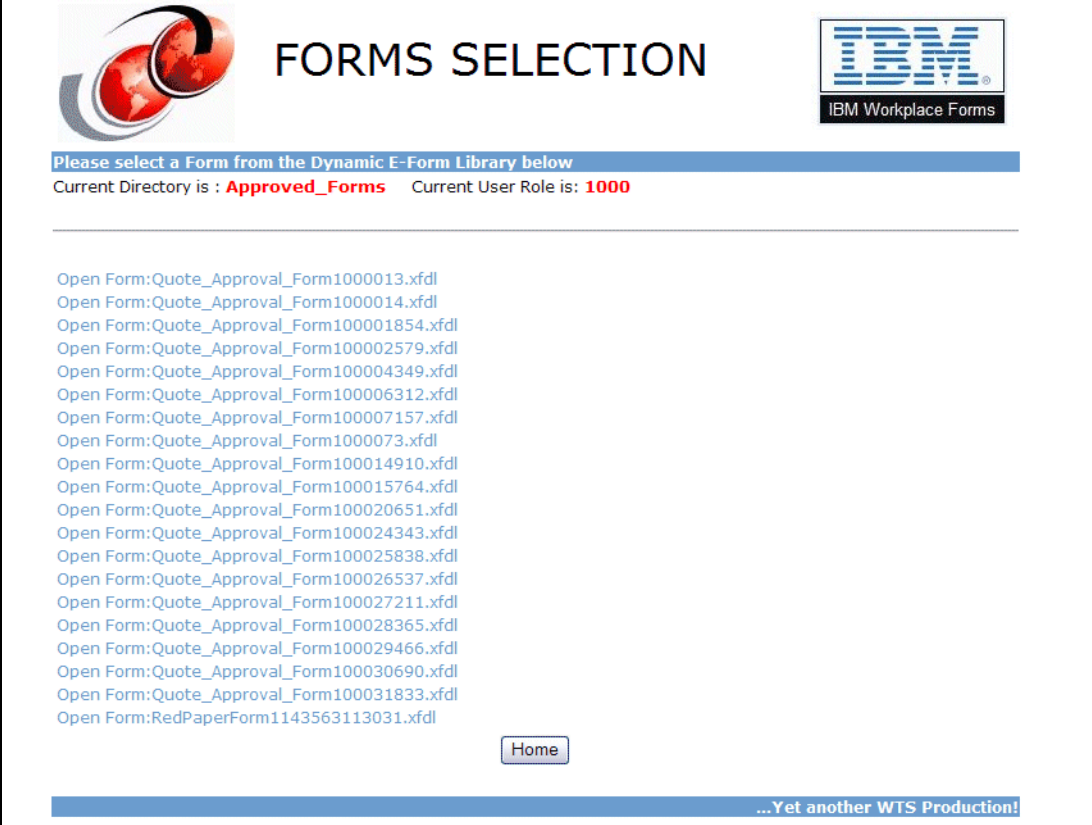


Figure 4-54 Form-Template listing using *dirlisting1.jsp*

4.9.3 Approved form listing

The **Approved** button allows the user to view all forms that have gone through the approval process. Clicking this button utilizes the *dirlisting1.jsp*, which displays the contents of the Approved_Forms folder.

Figure 4-55 shows a sample Approved_Forms listing view.



FORMS SELECTION

Please select a Form from the Dynamic E-Form Library below

Current Directory is : **Approved_Forms** Current User Role is: **1000**

Open Form:Quote_Approval_Form1000013.xfdl
Open Form:Quote_Approval_Form1000014.xfdl
Open Form:Quote_Approval_Form100001854.xfdl
Open Form:Quote_Approval_Form100002579.xfdl
Open Form:Quote_Approval_Form100004349.xfdl
Open Form:Quote_Approval_Form100006312.xfdl
Open Form:Quote_Approval_Form100007157.xfdl
Open Form:Quote_Approval_Form1000073.xfdl
Open Form:Quote_Approval_Form100014910.xfdl
Open Form:Quote_Approval_Form100015764.xfdl
Open Form:Quote_Approval_Form100020651.xfdl
Open Form:Quote_Approval_Form100024343.xfdl
Open Form:Quote_Approval_Form100025838.xfdl
Open Form:Quote_Approval_Form100026537.xfdl
Open Form:Quote_Approval_Form100027211.xfdl
Open Form:Quote_Approval_Form100028365.xfdl
Open Form:Quote_Approval_Form100029466.xfdl
Open Form:Quote_Approval_Form100030690.xfdl
Open Form:Quote_Approval_Form100031833.xfdl
Open Form:RedPaperForm1143563113031.xfdl

[Home](#)

...Yet another WTS Production!

Figure 4-55 Approved Form Listing view using *dirlisting1.jsp*



Building the base scenario: Stage 2

In this chapter we extend the base J2EE scenario we created in Chapter 4, “Building the base scenario: Stage 1” on page 53. We are now adding a DB2 environment. The reason for this is to take into consideration the potential data growth and subsequent server performance implications when storing a huge amount of large XDFL files on the file system. Storing the forms in DB2 gives you scalability, reliability, and database search capabilities.

We describe the following topics:

- ▶ Installing DB2 clients and servers
- ▶ Creating and populating DB2 tables
- ▶ Developing a DB2 data access layer
- ▶ Working with Web services
- ▶ Using the Workplace Forms API
- ▶ Creating JSPs to view DB2 data

Note: The code used for building this sample scenario application is available for download. For specific information about how to download the sample code, please refer to Appendix A, “Additional material” on page 333.

Note: All specific examples shown and used when building the sample scenario application are based on the codebase for IBM Workplace Forms Release 2.5.

5.1 Overview of steps: Building Stage 2 of the base scenario

The diagram in Figure 5-1 is intended to provide an overview of where we are within the key steps involved to build Stage 2 of the base scenario. This focuses on adding storage capabilities to DB2, incorporating Web services, modifying the JSPs, and adding an approval workflow.

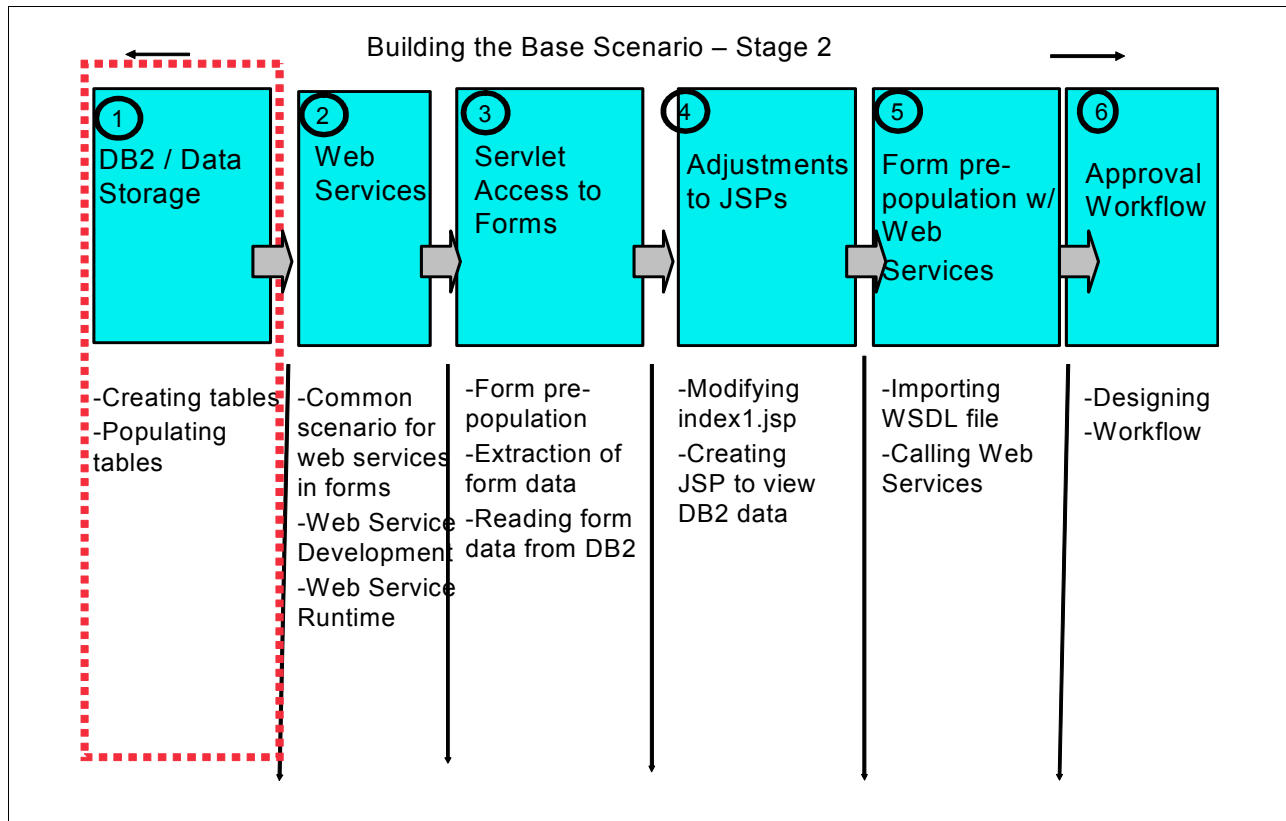


Figure 5-1 Overview of major steps involved in building Stage 2 of base scenario application

5.2 Data storage to DB2

A main topic of Stage 2 in building the sample application is moving the leading data storage to a relational database to take advantage of database search capabilities, scalability, and reliability. The following data should be available as SQL data for the project:

- ▶ Basic application parameters (such as business rules)
- ▶ Organization data (such as Employee data, Manager)
- ▶ Inventory (such as Product description, stock level)
- ▶ Customer registry (such as ID, Name, related sales person)
- ▶ Order metadata (such as ID, name, submitting sales person)
- ▶ Order forms (complete XFDL files as CLOB)
- ▶ Counter for new order number

Due to the use of large data objects (XFDL files can reach the multiple megabyte size range), we decided to use a separate DB2 table for data storage. For each of the data objects mentioned above, there is one dedicated table holding all available data and one additional table to store the entire amount of submitted forms.

The data should be accessed using the Class 2 JDBC™ driver available in the install package.

Note: Access to CLOB / BLOB is not a JDBC standard, nevertheless the chosen JDBC driver supports these operations. Make sure your driver / database combination will handle BLOB or CLOB objects when storing templates or submitted files in a database. Forms containing one or more megabytes are rarely used, but sometimes required.

5.2.1 Installing DB2 Server

All tables are assigned to one single database instance with a standard DB2 setup on a Windows 2003 server machine. We processed a standard DB2 UDB 8.2 installation using the following setup parameters (Table 5-1).

Table 5-1 DB2 Universal Database™ setup parameters

Setup parameter	Parameter value	Comments
Setup type	Standard	Creates a DB2 instance and all necessary administration utilities (Control Center, Command line processor)
Instance name	DB2	Default instance name
Instance node name	TCP8D534	Generated during setup
Administration user	wpsadmin	Same user as portal / WebSphere Application Server (WAS) administrator
Administration user password	wpsadmin	Be careful about user name and password - setup will create a system account with full administration permissions - never use here any common known (default) passwords in production or sensitive environments
Maintenance mode	Low administration / low performance	Arbitrary - keep things simple for this project
Admin notification mode	Defer	Arbitrary - keep things simple for this project

After setup, we created a new DB2 database (Table 5-2).

Table 5-2 DB2 Universal Database Setup Parameters

Setup Parameter	Parameter Value	Comments
Database name	VMFORMS	
Database alias	VMFORMS	

All database tables necessary for the project are created in this database.

5.2.2 Creating tables

The project uses the following tables, as listed in Table 5-3. (For detailed column attributes, see the table setup scripts in Example 5-1 on page 151 and Example 5-2 on page 154.)

Table 5-3 DB2 table descriptions

Table name	Column name	Comments
WPF_Param		Table containing administrative parameters, business rules and other setup data on a key / value concept; to store numeric and text data, the table contains two parameter value columns dedicated to the two value types - no administration utilities to maintain data in the application - use DB2 administration tools for data maintenance
	Par_Key	Key value for parameter lookup
	Par_NumValue	Numeric parameter value
	Par_TextValue	Text string assigned to the parameter.
WPF_ORG		Organization data (Employee data and reporting line) - no administration utilities to maintain data in the application - use DB2 administration tools for data maintenance.
	Org_ID	Employee ID (used as key and role identification)
	Org_FirstName	Employee first name
	Org_LastName	Employee last name
	Org_ContactInfo	Employee mail address or other contact data
	Org_MGR	Managers ID - references to the Org_ID of any other employee

Table name	Column name	Comments
WPF_CUST		Customer data (name, contact person data and responsible sales person) - no administration utilities to maintain data in the application - use DB2 administration tools for data maintenance
	CUST_ID	Customer company ID (unique key for data lookup)
	CUST_NAME	Company name
	CUST_AMGR	ID of responsible sales person - related or column ORG_ID in table WPR_ORG
	CUST_CONTACT_NAME	Name of contact person on customer site
	CUST_CONTACT_POSITION	Position of contact person on customer site
	CUST_CONTACT_EMAIL	E-mail address of contact person on customer site
	CUST_CONTACT_PHONE	Phone number of contact person on customer site
	CUST_CRM_NO	ID for the customer in the local CRM system
WPF_ITEMS		Product catalog used for product lookup. No administration utilities to maintain data in the application - use DB2 administration tools for data maintenance.
	IT_ID	Item ID (used for item details lookup)
	IT_NAME	Item name (short item description)
	IT_PRICE	Item price per unit
	IT_STOCK	Stock availability

Table name	Column name	Comments
WPF_ORDERS		Table storing active and archived order metadata - the application reads data for list display / order selection and writes/updates data on form submit
	ORD_ID	Order ID (unique) used for data lookup
	ORD_CUST_ID	Customer ID for the related order
	ORD_AMOUNT	Total order amount
	ORD_DISCOUNT	Total order discount
	ORD_SUBMITTER_ID	ID of the sales person creating this order (related to field ORG_ID in table WPF_ORG)
	ORD_STATE	Order state: 1 - new order not processed yet 2 - order waiting for manager approval 3 - order waiting for director approval 4 - completed (finally accepted) order 5 - rejected order waiting for rework 6 - canceled order
	ORD_CREATION_DATE	Order creation date
	ORD_COMPLETION_DATE	Order completion date
	ORD_OWNER	ID for the actual owner (sales person) for the owner - initial the submitting same person (related to field ORG_ID in table WPF_ORG)
	ORD_VERSION	Order version - the version number of the order form used
	ORD_APP_1	ID of approving manager (related to field ORG_ID in table WPF_ORG)
	ORD_APP_DATE_1	Date of manager approval
	ORD_APP_COMMENT_1	Comment for manager approval
	ORD_APP_2	ID of approving director (related to field ORG_ID in table WPF_ORG)
	ORD_APP_DATE_2	Date of director approval
	ORD_APP_COMMENT_2	Comment for director approval
WPF_ORDXFDL		Table storing the entire XFDL form containing detail order information and signing information. Each record related to a corresponding order metadata record stored in table WPF_ORDERS with the same Order ID. The application reads data for on form open and writes/updates data on form submit.
	ORD_ID	Order ID (unique) used for data lookup relates to order metadata in table WPF_ORDERS by same ORD_ID.
	ORD_XFDL	CLOB field (1 MB maximum) to store entire XFDL file.

Table name	Column name	Comments
WPF_ORDNOCNT		Counter for new order numbers
	ORD_ID	Last used order number - will increase by 1 for each new order

Tip: To avoid multi-megabyte CLOB column definitions used for form storage, the read/write routines could join/split the entire file into smaller blocks stored in consecutive records. This technique could help to overcome database or driver limitations but would result in general in a somewhat lower performance.

Note: There is no DB2 based repository for available XFDL templates in or project. These files could be stored in DB2 similar to the submitted forms. We did not take this approach just for development time limitations. So we saved time on development for special (template) file upload and maintenance facilities using file system storage as in Stage 1 scenario.

On project start, these tables were created using the DB2 command line processor on the server machine with the following SQL script (Example 5-1).

Example 5-1 DB2 table creation script

```
connect to WPFORMS user wpsadmin using wpsadmin
;
DROP TABLE WPF_PARAM;
DROP TABLE WPF_ORG;
DROP TABLE WPF_CUST;
DROP TABLE WPF_ITEMS;
DROP TABLE WPF_ORDERS;
DROP TABLE WPF_ORDXFD;
DROP TABLE WPF_ORDNOCNT;
```

```
CREATE TABLE WPF_Param (
    Par_Key VARCHAR (20) NOT NULL DEFAULT ,
    Par_NumValue DECIMAL (10,2) ,
    Par_TextValue VARCHAR (100) DEFAULT ,
    PRIMARY KEY (PAR_KEY)
);
select * from WPF_Param;
```

```
CREATE TABLE WPF_ORG (
    Org_ID VARCHAR (10) NOT NULL DEFAULT ,
    Org_FirstName VARCHAR (30) NOT NULL DEFAULT ,
    Org_LastName VARCHAR (30) NOT NULL DEFAULT ,
    Org_ContactInfo VARCHAR (100) NOT NULL DEFAULT ,
    Org_MGR VARCHAR (10) NOT NULL ,
    PRIMARY KEY (ORG_ID)
);
select * from WPF_ORG;
```

```

CREATE TABLE WPF_CUST (
    CUST_ID VARCHAR (10) NOT NULL DEFAULT ,
    CUST_NAME VARCHAR (30) NOT NULL DEFAULT ,
    CUST_AMGR VARCHAR (10) NOT NULL DEFAULT ,
    CUST_CONTACT_NAME VARCHAR (30) NOT NULL DEFAULT ,
    CUST_CONTACT_POSITION VARCHAR (30) NOT NULL DEFAULT ,
    CUST_CONTACT_EMAIL VARCHAR (50) NOT NULL DEFAULT ,
    CUST_CONTACT_PHONE VARCHAR (20) NOT NULL DEFAULT ,
    CUST_CRM_NO VARCHAR (10) NOT NULL DEFAULT ,
    PRIMARY KEY (CUST_ID)
);
select * from WPF_CUST;

```

```

CREATE TABLE WPF_ITEMS (
    IT_ID VARCHAR (10) NOT NULL DEFAULT ,
    IT_NAME VARCHAR (30) NOT NULL DEFAULT ,
    IT_PRICE DECIMAL (10,2) NOT NULL DEFAULT ,
    IT_STOCK INTEGER NOT NULL DEFAULT 0,
    PRIMARY KEY (IT_ID)
);
select * from WPF_ITEMS;

```

```

CREATE TABLE WPF_ORDERS (
    ORD_ID VARCHAR (10) NOT NULL ,
    ORD_CUST_ID VARCHAR (10) DEFAULT '',
    ORD_AMOUNT DECIMAL (10,2) DEFAULT 0,
    ORD_DISCOUNT DECIMAL (10,2) DEFAULT 0,
    ORD_SUBMITTER_ID VARCHAR (10) ,
    ORD_STATE VARCHAR (10) DEFAULT '',
    ORD_CREATION_DATE DATE ,
    ORD_COMPLETION_DATE DATE ,
    ORD_OWNER VARCHAR (10) DEFAULT '1.0',
    ORD_VERSION VARCHAR (10) DEFAULT '1.0',
    ORD_APP_1 VARCHAR (10) DEFAULT '',
    ORD_APP_DATE_1 DATE ,
    ORD_APP_COMMENT_1 VARCHAR (200) DEFAULT '',
    ORD_APP_2 VARCHAR (10) DEFAULT '',
    ORD_APP_DATE_2 DATE ,
    ORD_APP_COMMENT_2 VARCHAR (200) DEFAULT '',
    PRIMARY KEY (ORD_ID)
);
select * from WPF_ORDERS;

```

```

CREATE TABLE WPF_ORDXFD (
    ORD_ID VARCHAR (10) NOT NULL ,
    ORD_XFDL CLOB (1000000) DEFAULT '',
    PRIMARY KEY (ORD_ID)
);

```

```
);
select * from WPF_ORDXFD;
```

```
CREATE TABLE WPF_ORDNOCNT (
    ORD_ID VARCHAR(10) NOT NULL ,
    PRIMARY KEY (ORD_ID)
);
select * from WPF_ORDXFD;
```

```
SELECT COUNT (*) FROM WPF_Param ;
SELECT COUNT (*) FROM WPF_ORG ;
SELECT COUNT (*) FROM WPF_CUST ;
SELECT COUNT (*) FROM WPF_ITEMS ;
SELECT COUNT (*) FROM WPF_ORDERS ;
SELECT COUNT (*) FROM WPF_ORDXFD ;
SELECT COUNT (*) FROM WPF_ORDNOCNT ;
```

```
SELECT * FROM WPF_Param ;
SELECT * FROM WPF_ORG ;
SELECT * FROM WPF_CUST ;
SELECT * FROM WPF_ITEMS ;
SELECT * FROM WPF_ORDERS ;
SELECT * FROM WPF_ORDXFD ;
SELECT * FROM WPF_ORDNOCNT ;
```

The script initially deletes all used tables (DROP statements) to have a simple redeployment whenever tables or example data changes.

Next, all tables are created (CREATE statements). Finally, all tables are hit to have an easy way to see if any error occurred during setup (SELECT statements).

Tip: This procedure is really helpful when deploying large database structures with multiple dependencies between tables, views, and stored procedures. Actually, the created structure is rather simple. To keep code, installation, and documentation as simple as possible, we did not set up any relational references, or special lookup views inheriting related data. In reality, there are various interdependencies between the data structures, making setup validation a mandatory task.

5.2.3 Populating tables

To have a manageable project start, we filled the tables with example data. This makes it easy to design all read only functionality (Web services, table display JSPs, prepopulation functionality) independent from any data creating modules. These are usually created in more advanced project phases.

Initial data population was done using the following script (Example 5-2) right after table creation (INSERT statements will create one table row, SELECT statements will return the inserted data for error detection).

Example 5-2 Example of data prepopulation to DB2 tables

```

INSERT INTO WPF_Param (Par_Key, Par_NumValue, Par_TextValue) VALUES ('MgrThreshold', 10000,
'Manager Approval Threshold');
INSERT INTO WPF_Param (Par_Key, Par_NumValue, Par_TextValue) VALUES ('DirThreshold', 50000,
'Director Approval Threshold');
INSERT INTO WPF_Param (Par_Key, Par_NumValue, Par_TextValue) VALUES ('Discount_1', 10,
'Rabate 1 in percent');
INSERT INTO WPF_Param (Par_Key, Par_NumValue, Par_TextValue) VALUES ('Discount_2', 20,
'Rabate 2 in percent');
INSERT INTO WPF_Param (Par_Key, Par_NumValue, Par_TextValue) VALUES ('Discount_3', 30,
'Rabate 3 in percent');

SELECT * FROM WPF_Param;

INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1000', 'Christine', 'Haas', '1000.Christine.Haas@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1001', 'Michael', 'Thompson', '1001.Michael.Thompson@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1002', 'Sally', 'Kwan', '1002.Sally.Kwan@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1003', 'John', 'Geyer', '1003.John.Geyer@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1004', 'Irving', 'Stern', '1004.Irving.Stern@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1005', 'Eva', 'Pulaski', '1005.Eva.Pulaski@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1006', 'Eileen', 'Henderson', '1006.Eileen.Henderson@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1007', 'Theodore', 'Spenser', '1007.Theodore.Spenser@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1008', 'Vincenzo', 'Lucchessi', '1008.Vincenzo.Lucchessi@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1009', 'Sean', 'Connell', '1009.Sean.Connell@ACME.cam.itso.ibm.com', '1010');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1010', 'Dolores', 'Quintana', '1010.Dolores.Quintana@ACME.cam.itso.ibm.com', '1031');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1011', 'Heather', 'Nicholls', '1011.Heather.Nicholls@ACME.cam.itso.ibm.com', '1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1012', 'Bruce', 'Adamson', '1012.Bruce.Adamson@ACME.cam.itso.ibm.com', '1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1013', 'Elizabeth', 'Pianka', '1013.Elizabeth.Pianka@ACME.cam.itso.ibm.com', '1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1014', 'Masatoshi', 'Yoshimura', '1014.Masatoshi.Yoshimura@ACME.cam.itso.ibm.com',
'1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1015', 'Marilyn', 'Scoutten', '1015.Marilyn.Scoutten@ACME.cam.itso.ibm.com', '1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1016', 'James', 'Walker', '1016.James.Walker@ACME.cam.itso.ibm.com', '1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1017', 'David', 'Brown', '1017.David.Brown@ACME.cam.itso.ibm.com', '1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1018', 'William', 'Jones', '1018.William.Jones@ACME.cam.itso.ibm.com', '1020');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1019', 'Jennifer', 'Lutz', '1019.Jennifer.Lutz@ACME.cam.itso.ibm.com', '1020');

```

```

INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1020', 'James', 'Jefferson', '1020.James.Jefferson@ACME.cam.itso.ibm.com', '1031');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1021', 'Salvatore', 'Marino', '1021.Salvatore.Marino@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1022', 'Daniel', 'Smith', '1022.Daniel.Smith@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1023', 'Sybil', 'Johnson', '1023.Sybil.Johnson@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1024', 'Maria', 'Perez', '1024.Maria.Perez@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1025', 'Ethel', 'Schneider', '1025.Ethel.Schneider@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1026', 'John', 'Parker', '1026.John.Parker@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1027', 'Philip', 'Smith', '1027.Philip.Smith@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1028', 'Maude', 'Setright', '1028.Maude.Setright@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1029', 'Ramlal', 'Mehta', '1029.Ramlal.Mehta@ACME.cam.itso.ibm.com', '1030');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1030', 'Wing', 'Lee', '1030.Wing.Lee@ACME.cam.itso.ibm.com', '1031');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('1031', 'Jason', 'Gounot', '1031.Jason.Gounot@ACME.cam.itso.ibm.com', '1031');
INSERT INTO WPF_ORG (Org_ID, Org_FirstName, Org_LastName, Org_ContactInfo, Org_MGR) VALUES
('wpsadmin', 'Admin', 'Portal&WAS', 'wpsadmin@ACME.cam.itso.ibm.com', '1031');

```

```

SELECT * FROM WPF_ORG;

```

```

INSERT INTO WPF_CUST (CUST_ID, CUST_NAME, CUST_AMGR, CUST_CONTACT_NAME,
CUST_CONTACT_POSITION, CUST_CONTACT_EMAIL, CUST_CONTACT_PHONE, CUST_CRM_NO) VALUES
('100000', 'OnDemand Corporation', '1000', 'Jerry Haas', 'AccountMgr',
'Jerry.Haas@OnDemand.oom', '+49 89 123-456-78', '200001');
INSERT INTO WPF_CUST (CUST_ID, CUST_NAME, CUST_AMGR, CUST_CONTACT_NAME,
CUST_CONTACT_POSITION, CUST_CONTACT_EMAIL, CUST_CONTACT_PHONE, CUST_CRM_NO) VALUES
('100001', 'Workplace Early Adopter Inc', '1000', 'Mary F Thompson', 'DeptMgr',
'Mary.F.Thompson@Workplace-Early-Adopter.com', '1 756-568-123', '200002');
INSERT INTO WPF_CUST (CUST_ID, CUST_NAME, CUST_AMGR, CUST_CONTACT_NAME,
CUST_CONTACT_POSITION, CUST_CONTACT_EMAIL, CUST_CONTACT_PHONE, CUST_CRM_NO) VALUES
('100002', 'Portal Application Surfacing', '1001', 'Hiu Kwan', 'Director',
'Hiu.Kwan@p-app.surf.org', '+43 623-644', '200003');
INSERT INTO WPF_CUST (CUST_ID, CUST_NAME, CUST_AMGR, CUST_CONTACT_NAME,
CUST_CONTACT_POSITION, CUST_CONTACT_EMAIL, CUST_CONTACT_PHONE, CUST_CRM_NO) VALUES
('100003', 'Workplace Forms Redpapers Inc', '1001', 'Max Ritter', 'AccountMgr',
'Max.Ritter@wpfrp.ibm.com', '1 756-123-456', '200004');
INSERT INTO WPF_CUST (CUST_ID, CUST_NAME, CUST_AMGR, CUST_CONTACT_NAME,
CUST_CONTACT_POSITION, CUST_CONTACT_EMAIL, CUST_CONTACT_PHONE, CUST_CRM_NO) VALUES
('100004', 'Global Security Trust Center Inc', '1001', 'Toni Tester', 'DeptMgr',
'Toni.Tester@gstc.de', '1 756-568-124', '200005');
INSERT INTO WPF_CUST (CUST_ID, CUST_NAME, CUST_AMGR, CUST_CONTACT_NAME,
CUST_CONTACT_POSITION, CUST_CONTACT_EMAIL, CUST_CONTACT_PHONE, CUST_CRM_NO) VALUES
('100005', 'Mobile Devices Corporation', '1002', 'Monique Lille', 'Director',
'Monique.Lille@md-corp.fr', '1 756-568-125', '200006');

```

```

SELECT * FROM WPF_CUST;

```

```

INSERT INTO WPF_ITEMS (IT_ID, IT_NAME, IT_PRICE, IT_STOCK) VALUES ('IT_001', 'Nut', 21.15,
11111);

```

```

INSERT INTO WPF_ITEMS (IT_ID, IT_NAME, IT_PRICE, IT_STOCK) VALUES ('IT_002', 'Bolt', 30.00,
22222);
INSERT INTO WPF_ITEMS (IT_ID, IT_NAME, IT_PRICE, IT_STOCK) VALUES ('IT_003', 'Widget',
512.99, 33333);
INSERT INTO WPF_ITEMS (IT_ID, IT_NAME, IT_PRICE, IT_STOCK) VALUES ('IT_004', 'Gadget',
12345, 44444);
INSERT INTO WPF_ITEMS (IT_ID, IT_NAME, IT_PRICE, IT_STOCK) VALUES ('IT_005', 'Thingy',
5000, 55555);

SELECT * FROM WPF_ITEMS;

```

```

INSERT INTO WPF_ORDERS (ORD_ID, ORD_CUST_ID, ORD_AMOUNT, ORD_DISCOUNT, ORD_SUBMITTER_ID,
ORD_STATE, ORD_CREATION_DATE, ORD_COMPLETION_DATE, ORD_OWNER, ORD_VERSION, ORD_APP_1,
ORD_APP_DATE_1, ORD_APP_COMMENT_1, ORD_APP_2, ORD_APP_DATE_2, ORD_APP_COMMENT_2) VALUES
('10000', '100001', 1000, 0, '1000', 'SUBMITTED', '2006-03-01', '9999-12-31', '1000',
'01.0', NULL, NULL, '1000', NULL, NULL, NULL);
INSERT INTO WPF_ORDERS (ORD_ID, ORD_CUST_ID, ORD_AMOUNT, ORD_DISCOUNT, ORD_SUBMITTER_ID,
ORD_STATE, ORD_CREATION_DATE, ORD_COMPLETION_DATE, ORD_OWNER, ORD_VERSION, ORD_APP_1,
ORD_APP_DATE_1, ORD_APP_COMMENT_1, ORD_APP_2, ORD_APP_DATE_2, ORD_APP_COMMENT_2) VALUES
('10001', '100002', 2000, 10, '1002', 'APPROVED1', '2006-03-02', '9999-12-31', '1010',
'01.0', 'APPROVED', '2006-03-02', '2000', 'APPROVED', NULL, NULL);
INSERT INTO WPF_ORDERS (ORD_ID, ORD_CUST_ID, ORD_AMOUNT, ORD_DISCOUNT, ORD_SUBMITTER_ID,
ORD_STATE, ORD_CREATION_DATE, ORD_COMPLETION_DATE, ORD_OWNER, ORD_VERSION, ORD_APP_1,
ORD_APP_DATE_1, ORD_APP_COMMENT_1, ORD_APP_2, ORD_APP_DATE_2, ORD_APP_COMMENT_2) VALUES
('10002', '100003', 3000, 20, '1021', 'APPROVED2', '2006-03-03', '9999-12-31', '1020',
'01.0', 'APPROVED', '2006-03-03', '3000', 'APPROVED', '2006-03-03', NULL);
INSERT INTO WPF_ORDERS (ORD_ID, ORD_CUST_ID, ORD_AMOUNT, ORD_DISCOUNT, ORD_SUBMITTER_ID,
ORD_STATE, ORD_CREATION_DATE, ORD_COMPLETION_DATE, ORD_OWNER, ORD_VERSION, ORD_APP_1,
ORD_APP_DATE_1, ORD_APP_COMMENT_1, ORD_APP_2, ORD_APP_DATE_2, ORD_APP_COMMENT_2) VALUES
('10003', '100002', 4000, 30, '1000', 'COMPLETED', '2006-03-04', '2006-03-17', '1031',
'01.0', 'APPROVED', '2006-03-04', NULL, 'APPROVED', '2006-03-04', NULL);
INSERT INTO WPF_ORDERS (ORD_ID, ORD_CUST_ID, ORD_AMOUNT, ORD_DISCOUNT, ORD_SUBMITTER_ID,
ORD_STATE, ORD_CREATION_DATE, ORD_COMPLETION_DATE, ORD_OWNER, ORD_VERSION, ORD_APP_1,
ORD_APP_DATE_1, ORD_APP_COMMENT_1, ORD_APP_2, ORD_APP_DATE_2, ORD_APP_COMMENT_2) VALUES
('10004', '100001', 5000, 0, '1000', 'REJECTED', '2006-03-05', '9999-12-31', '1010',
'01.0', 'REJECTED', NULL, NULL, NULL, NULL, NULL);
INSERT INTO WPF_ORDERS (ORD_ID, ORD_CUST_ID, ORD_AMOUNT, ORD_DISCOUNT, ORD_SUBMITTER_ID,
ORD_STATE, ORD_CREATION_DATE, ORD_COMPLETION_DATE, ORD_OWNER, ORD_VERSION, ORD_APP_1,
ORD_APP_DATE_1, ORD_APP_COMMENT_1, ORD_APP_2, ORD_APP_DATE_2, ORD_APP_COMMENT_2) VALUES
('10005', '100002', 6000, 0, '1002', 'CANCELED', '2006-03-06', '9999-12-31', '1010',
'01.0', NULL, NULL, NULL, NULL, NULL, NULL);

```

```

SELECT * FROM WPF_ORDERS;

```

```

INSERT INTO WPF_ORDXFD (ORD_ID, ORD_XFDL) VALUES ('10000', NULL);
INSERT INTO WPF_ORDXFD (ORD_ID, ORD_XFDL) VALUES ('10001', NULL);
INSERT INTO WPF_ORDXFD (ORD_ID, ORD_XFDL) VALUES ('10002', NULL);
INSERT INTO WPF_ORDXFD (ORD_ID, ORD_XFDL) VALUES ('10003', NULL);
INSERT INTO WPF_ORDXFD (ORD_ID, ORD_XFDL) VALUES ('10004', NULL);
INSERT INTO WPF_ORDXFD (ORD_ID, ORD_XFDL) VALUES ('10005', NULL);

```

```

SELECT * FROM WPF_ORDXFD;

```

```

INSERT INTO WPF_ORDNOCNT (ORD_ID) VALUES ('1000000');

```

```

SELECT * FROM WPF_ORDNOCNT;

```

The available sample data was adjusted to project needs over in iterative steps. Each step results in a new table setup and table example data population.

The lifecycle of this data differs for the various objects:

- ▶ Sample order data (Data in Table WPF_ORDERS and WPF_ORDXFDL) is not complete (we did not store XFDL files as sample data). So, after having created the first full orders in Stage 2, we deleted the initial example data in WPF_ORD table to avoid exceptions when opening orders with missing form data.
- ▶ Organization data, product catalog and project parameters (Tables WPF_ORG, WPF_ORD, WPF_PARAM) stayed unchanged during the project development. Maintenance routines for these tables are out of scope for this project.

5.2.4 Installing DB2 clients on development clients and servers

In this section we describe the installation of DB2 clients on the development systems.

Development workstations

For work with the used JDBC driver, each developer testing DB2 related code needs to run a DB2 client on the local workstation:

- ▶ Servlet development for data prepopulation and data storage on new order or order submit events
- ▶ JSP development showing available order lists based on data stored in DBs
- ▶ Web service provider development

Therefore we installed on those workstations a DB2 client and DB2 Control Center to have a convenient setup for the necessary database registration. There are various ways to register the database to the client — we used DB2 Control Center wizards. Make sure to assign the same alias to the database on client registration as used on the server, because the code will run using a fix server name database and alias to identify the connection.

Workstations used for Workplace Forms (XFDL) development and end user tests do not need a DB2 client installation.

Production servers

In a distributed environment the following server would need to install a DB2 client and register the target database:

- ▶ WebSphere/Tomcat Application Server running the created servlet and JSPs
- ▶ WebSphere/Tomcat Application Server running the Web service provider applications

In our project one server machine handled all server tasks (DB2 Server, WebService provider, application server, http server). So we did not install additional DB2 clients on remote servers in this project.

5.2.5 Developing the data access layer (DB2)

To make parallel development of different components possible from start up — without time to develop a sophisticated data model — we decided to make interfaces between parallel tracks as flexible as possible.

Two basic assumptions were made:

- ▶ Design interfaces were only based on simple Java data types (String, String[] and String[][]). This saved much time, usually necessary for modelling interface data objects.

- No publishing was done on internal DB2 artifacts (table gnomon, column names, queries) outside the DB2 data access layer to reduce impact on any changes in DB2 structures, queries, etc. on the business logic and presentation layer.

These presumptions led to a simple 2-layer concept for basic data access, resulting in a 2-level Java library (WPFormsDBsConnectionT) with the following functionality:

- Class DB2Connection for basic DB2 connectivity:

- Open connection
- Read row data
- Insert row data
- Update row data

This module is only accessed by the access by modules in DB2ConnectionForms. The available methods, in general, accept input data formed as a valid SQL query and return output as simple strings or string arrays.

- Class DB2ConnectionForms exposing an easy-to-use and robust interface for business logic modules abstracting mainly from table and field names, creating for each data object the corresponding read/write operations as dedicated methods. So in most of the cases, reading or writing data could be reduced to a “one-liner” in the higher level application functionality.

Procedures for opening and closing the JDBC connection are shown here (Example 5-3).

Example 5-3 Basic JDBC routines for opening and closing a connection

```

/*
 * Created on Mar 10, 2006
 *
 * Basic DB2 Connection routines using a level 2 driver
 * db2java.zip / db2java.jar from DB2 8.2 UDB release
 */
package forms.cam.itso.ibm.com;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @author Andreas Richter ISSL
 *
 * application independent db2 calls (NO table and column namens here)
 */
public class DB2Connection {

    /**
     * Comment for <code>db2con</code> application independent db2 calls (NO
     * table and column namens)
     */
    //connection object
    private static Connection db2con;

    // For local tests set this attribute in the calling code like this:
    //DB2Connection.env = "LocalTestEnvironment"
    // For productonnuse specify an empty string

```

```

public static String env = "";

/* Generic procedures */

/**
 * create a jdbc connection
 */
public static void connect() {

    //set connection credentials
    String driver = "COM.ibm.db2.jdbc.app.DB2Driver";
    String url = Messages.getString("DB2_url");
    String user = Messages.getString("DB2_username");
    String password = Messages.getString("DB2_password");
    //initiate connection object
    db2con = null;

    try {
        // Load the DB2 JDBC Type 2 Driver with DriverManager
        Class.forName(driver);
        //System.out.println("Driver found: " + driver);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        System.out.println("Driver not found:" + driver);
        db2con = null;
    }

    //open the connection
    try {
        db2con = DriverManager.getConnection(url, user, password);
        //in real life set it to false and use commit method after write
        // operations
        db2con.setAutoCommit(true);
        //System.out.println("Connected: " + url);
    } catch (SQLException e1) {
        System.out.println("NOT Connected " + url + " User: " + user
            + " pw: " + password);
        e1.printStackTrace();
        //try to reconnect
        if (db2con != null) {
            try {
                db2con.close();
                db2con = DriverManager.getConnection(url, user, password);
                db2con.setAutoCommit(true);
            } catch (SQLException e2) {
                // TODO Auto-generated catch block
                System.out.println("no connection: " + url);
            }
        }
        //no connection - clear object
        db2con = null;
    }
}

/**
 * commit method
 */
public static void commit() {
    try {
        db2con.commit();
    }
}

```

```

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    /**
     * close connection
     */
    public static void disconnect() {
        try {
            db2con.close();
        } catch (SQLException e1) {
            System.out.println("could not close connection");
        }
    }

    // Data retrieval routines

    // .....
}

```

Here is an example for the property file we used (Example 5-4):

Example 5-4 File db2connection.properties

```

DB2_url=jdbc:db2:wpforms
DB2_username=wpsadmin
DB2_password=wpsadmin

```

For read only connections, we mainly used statement oriented methods submitting a query such as “SELECT <variables> FROM <tablename> WHERE <selection criterion>”. A basic call looks always like this (Example 5-5).

Example 5-5 Example method to read one-column value

```

package forms.cam.itso.ibm.com;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DB2Connection {

    .....

    /**
     * get a specific field from a table row by sql query *
     *
     * @param query
     *         valid sql query for read (SELECT ...)
     * @return field value as String
     */
    public static String getField(String query) {
        String result = "";
        Statement stmt;
    }
}

```

```

        ResultSet rs = null;

        //open the connection
        DB2Connection.connect();

        try {
            //execute query
            stmt = db2con.createStatement(); // Create a Statement object
            rs = stmt.executeQuery(query);

            //read result as string
            result = "";
            try {
                rs.next();
                result = rs.getString(1); // Retrieve
                rs.close(); // Close the ResultSet
                stmt.close();
            } catch (SQLException e2) {
                // TODO Auto-generated catch block
                //e2.printStackTrace();
                System.out.println("No results: " + query);
                result = "";
            }
        } catch (SQLException e) {
            e.printStackTrace();
            result = e.toString();
        } finally {
            //always disconnect
            DB2Connection.disconnect();
        }
        return result;
    }

    ....
}

```

Reading and writing Character Large Objects (CLOBs), used to retrieve and store complete XFDL documents, is not contained in the JDBC standard. Nevertheless, it could be done with the chosen JDBC driver. To read CLOBs, we could use the `getField` method above. To insert and update CLOBs, we used a driver-specific method, which may not work on other drivers (Example 5-6).

Example 5-6 writing / inserting Character Large Objects (CLOBs)

```

/**
 *
 * Updates (or inserts) a clob field This method is NOT jdbc standard and
 * can fail depending on the used driver If the assigned row does not exist,
 * a new entry is created. If the assigned row exists, the clob is updated
 *
 * @param tableName
 *         table name to insert
 * @param id
 *         id (key field value)
 * @param id_field
 *         key field name
 * @param clob
 *         clob value
 * @param clob_field

```

```

*           field name for clob
*/
public static void updateCLOB(String tableName, String id, String id_field,
    String clob, String clob_field) {

    PreparedStatement stmt;
    ResultSet rs = null;
    String query = "";

    //connect
    DB2Connection.connect();
    //System.out.println("UPDATE CLOB: " + id);
    try {
        //create query
        query = "UPDATE " + tableName + " SET (" + clob_field + ") = (?) WHERE " +
id_field + " = '" + id + "'";
        stmt = db2con.prepareStatement(query); // Create a Statement object
        //assign clob value
        stmt.setString(1, clob);
        //update
        stmt.execute();
        db2con.commit();
        //close
        stmt.close();

    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } finally {
        //close connection
        DB2Connection.disconnect();
    }

}

```

To meet forms-specific needs in data prepopulation (assigning multiple values to an XFDL data instance (for example, complete employee data or customer data), we implemented a generic function (`getResultsXML`) returning one or more rows of an SQL table rendered as XML instances. The implemented code is shown here (Example 5-7).

Example 5-7 Returning data structures as XML instances

```

/**
 *
 * execute a query and return data rendered as xml string (convert all data
 * to string) tag names for data entities are created from column names tag
 * names for rows are created from tagInstance parameter
 *
 * @param query
 *         valid sql query
 * @param instanceTag
 *         tag name for the xml instance to create
 * @return xml instance like this <instanceTag> <column1>val
 *         </column1> <column2>val </column2> <column3>val </column3> ...
 *         <columnN>val </columnN> </instanceTag> .... <instanceTag sid=X>
 *         <column1>val </column1> <column2>val </column2> <column3>val
 *         </column3> ... <columnN>val </columnN> </instanceTag>
 *
 */
public static String getResultsXML(String query, String instanceTag) {

```

```

Statement stmt;
String resultXML = "";
String resultXMLrow = "";
ResultSet rs = null;
int row = 0; //row counter for sid generation
//connect
DB2Connection.connect();

try {
    //execute query
    stmt = db2con.createStatement(); // Create a Statement object
    rs = stmt.executeQuery(query);
    //get column number (need for row fetch)
    int cols = rs.getMetaData().getColumnCount();
    rs = stmt.executeQuery(query);
    row = 0;
    //read data and create xml
    try {
        while (rs.next()) { // Position the cursor
            resultXMLrow = "";
            for (int col = 0; col < cols; col++) {
                //add result to row data
                resultXMLrow = resultXMLrow
                    + createTag(rs.getString(col + 1), rs
                        .getMetaData().getColumnName(col + 1),
                        "");
            }
            //add row to return string
            resultXML = resultXML
                + createTag(
                    "\n" + resultXMLrow, instanceTag, "");
            row++;
        }
        rs.close(); // Close the ResultSet
        stmt.close();
    } catch (SQLException e2) {

        e2.printStackTrace();
    }

    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        //always close connection
        DB2Connection.disconnect();
    }
    return resultXML;
}

/**
 *
 * helper method to create an xml tags from a value
 *
 * @param tagData
 *         data to include
 * @param tagName
 *         name of the tag
 * @param attributes
 *         opt. additional attributes

```

```

*
* @return String containing xml represenatation of the query
*/
private static String createTag(String tagData, String tagName,
    String attributes) {
    String tag = "";
    if (attributes.equals("")) {
        tag = tag + "<" + tagName + ">";
    } else {
        tag = tag + "<" + tagName + " " + attributes + ">";
    }
    tag = tag + tagData;
    tag = tag + "</" + tagName + ">" + "\n";
    return tag;
}

```

A call to that method, with the following parameters, would return a string as in Example 5-8:
`getResultsXML("Select * from WPF_ORG","employee")`

*Example 5-8 Resulting XML Fragement for Query "Select * from WPF_ORG"*

```

<employee>
  <ORG_ID>1000</ORG_ID>
  <ORG_FIRSTNAME>Christine</ORG_FIRSTNAME>
  <ORG_LASTNAME>Haas</ORG_LASTNAME>
  <ORG_CONTACTINFO>1000.Christine.Haas@ACME.cam.itso.ibm.com</ORG_CONTACTINFO>
  <ORG_MGR>1010</ORG_MGR>
</employee>
<employee>
  <ORG_ID>1001</ORG_ID>
  <ORG_FIRSTNAME>Michael</ORG_FIRSTNAME>
  <ORG_LASTNAME>Thompson</ORG_LASTNAME>
  <ORG_CONTACTINFO>1001.Michael.Thompson@ACME.cam.itso.ibm.com</ORG_CONTACTINFO>
  <ORG_MGR>1010</ORG_MGR>
</employee>

.....

<employee>
  <ORG_ID>wpsadmin</ORG_ID>
  <ORG_FIRSTNAME>Admin</ORG_FIRSTNAME>
  <ORG_LASTNAME>Portal&WAS</ORG_LASTNAME>
  <ORG_CONTACTINFO>wpsadmin@ACME.cam.itso.ibm.com</ORG_CONTACTINFO>
  <ORG_MGR>1031</ORG_MGR>
</employee>

```

These XML fragments can easily be surrounded with the necessary tags and used for any data instance updates to the XFDL form using the XFDL API routine, `theForm.encloseInstance`. These fragments meet the xForms standard. In the project, we used that technique only to prepopulate the order number for new orders. See the related code in 4.8.2, "Basic servlet methods" on page 108 describing the servlet methods.

We did not create any data object specific interface classes in this project, since available time and resources did not allow us to do so. This was a good choice in most of the cases. Only when updating order metadata, we missed those dedicated objects, but once implemented, the model was not changed.

In a real project, the data model and functionality represented by the DB2ConnectionForms library might extended in the following way:

- ▶ Normalizing data structure (such as separating objects for customer site and customer contact data)
- ▶ Defining data object specific classes represented by interface classes exposing getters, setters, and all object specific data maintenance methods
- ▶ Enabling commit/rollback for DB2 transactions
- ▶ Providing appropriate data validation and translation methods
- ▶ Adding missing insert / update and write methods

After considering the topics above and any possible additional requirements (such as server platform considerations or support for special data types for another project), we could decide to use another driver type than JDBC or to use the driver with other functionality.

The created DB2 interface (class DB2ConnectionForms) exposes the methods in Table 5-4.

Table 5-4 Descriptions of data object methods and parameters

Data object method	Parameters	Description
Employee:		
getEmployeeData	emp_id (Employee ID as String) returns String [] array of emp. data	Reads all columns for a selected employee Used for prepopulation with submitter data. Called from SubmissionS.rvlet.
Customer:		
getCustomerData	cust_id (Customer ID as String) returns String [] array of cust. data	Reads all columns for a selected customer. Used for customer detail data. Called by a Web service inside the form.
getCustomerList	returns String [] array of cust. data	Reads full customer table and returns a String containing all customer IDs and Names. Used to build customer choices list. Called by a Web service inside the form.
getCustomersAll	returns String [] array of cust. data	Reads full customer table and returns a String [][] Array containing all customer data (actually not used).
getCustomersEmp	emp_id (Employee ID as String) returns String of selected cust. names and IDs	Reads customer table and returns a String containing all customer IDs and Names for a responsible sales person. The intent is to reduce the transferred data by a reasonable preselection based on submitter ID. Used to build customer choices list. Called by a Web service inside the form (not implemented yet but ready to run).

Data object method	Parameters	Description
Order:		
getOrderData	order_id (Employee ID as String) returns String [] array of order data	Reads all order data details for a given order id. Used: (actually not used). Called: (actually not used).
getOrderListEmp	emp_id (Employee ID as String) returns String [] array of cust. data	Reads all order data details for a given employee (acting as submitter, manager approver or director approver). Used for data display in personalized order table Called in JSP db2listing from servlet inline code
writeOrderData	orderData (all columns as String[]) returns String [] array of cust. data	Writes all order metadata extracted from a submitted form. Used to store actual order metadata in a DB2 table. Called in SubmissionServlet when receiving a submitted form containing the data instance "FormOrderData".
getOrderXFDL	order_id (Order ID as String) returns String [] array of cust. data	Reads a complete XFDL form as String. Used to get XFDL form data on form opening in Stage 2. Called in SubmissionServlet on showForm event.
readRowXFDL	order_id (Order ID as String) returns String [] array of cust. data	Reads a complete XFDL form as String. Used to get XFDL form data on form opening in Stage 2. Called in SubmissionServlet on showForm event. Same functionality as getOrderXFDL, but other internal code.
updateRowXFDL	order_id (Order ID as String) returns String [] array of cust. data	Writes (insert/update) a complete XFDL form as CLOB. Used to store XFDL form data on form submit in Stage 2. Called in SubmissionServlet on submitForm event.
writeOrderXFDL	order_id (Order ID as String) returns String [] array of cust. data	Writes (insert/update) a complete XFDL form as CLOB. Used to store XFDL form data on form submit in Stage 2. Called in SubmissionServlet on submitForm event. Same functionality as writeRowXFDL, but other internal code.

Data object method	Parameters	Description
Parameters:		
getParamNumValue	par_key (Key as String) returns Number value converted to String	Reads a numeric parameter value based on the given key. Used to get prepopulation data (business rules) when creating new forms Called in SubmissionServlet on createForm event.
getParamTextValue	par_key (Key as String) returns Text value as String	Reads a text parameter value based on the given key. Used: (actually not used). Called: (actually not used).
Inventory:		
getInventoryAll	returns String [] array of cust. data	Reads full inventory table and returns a String [][] Array containing all item data (actually not used).
getInventoryItemData	returns String [] array of cust. data	Reads all columns for a selected item. Used for item detail data . Called by a Web service inside the form for each item row.
getInventoryList	returns String [] array of cust. data	Reads full inventory item IDs and Names. Used to build item choices list for each of the item rows. Called by a Web service inside the form
Generic function in DN2Connection		
getNewOrderNumber	returns String containing a new order number	Increments the order number in table WPF_ORDNOCNT and returns the new number. Used: intended for order number. prepopulation in SubmissionServlet. Called: (actually not used).
getNewOrderNumber XML	returns String containing a new data instance for the order number	Increments the order number in table WPF_ORDNOCNT and returns the new number. Used for data instance prepopulation for new orders. Called from SubmissionServlet.
Generic function in DN2Connection:		
getResultsXML	query String containing any valid SQL query instanceTag String defining an XML tag name returns String containing the result rendered as an XML instance	Reads any data stored in DB2 defined by the incoming query Used: Actually only used for order number prepopulation. Creates XML instances useful to make data prepopulation easy by replacing internal data instances in the form with the composed XML instance. Called from SubmissionServlet

Note: There are two important points regarding the implemented functions:

- ▶ Functions `getCustomerList()`, `getCustomersEmp()` and `getInventoryList()` will return data in a somewhat “strange” format as a single string like “item1 [id1]~item2 [id2]~...~itemN [idN]”. This is a customized return format meeting best the requirements coming with both the Workplace Forms Web service implementation (single String convention) and the Workplace Forms drop down box implementation (name [ID] construct). These topics will be discussed in the next section (Web services) .
- ▶ All functions accepting an ID (user ID, customer ID, order ID) as an input parameter will accept both the ID or a string in the format “<arbitrary text>[<ID>]” as an input parameter value. This is a contribution to the behavior of the drop-down box in Workplace Forms. The DB2 layer was the most efficient place to code the data extraction (single function).

5.3 Web services

In this section we discuss various considerations regarding Web services.

5.3.1 Where we are in the process of Building Stage 2 of the base scenario

The following diagram is intended to provide an overview of where we are within the key steps involved to build Stage 2 of the base scenario. This focuses on adding storage capabilities to DB2, incorporating Web services, modifying the JSPs, and adding an approval workflow (Figure 5-2).

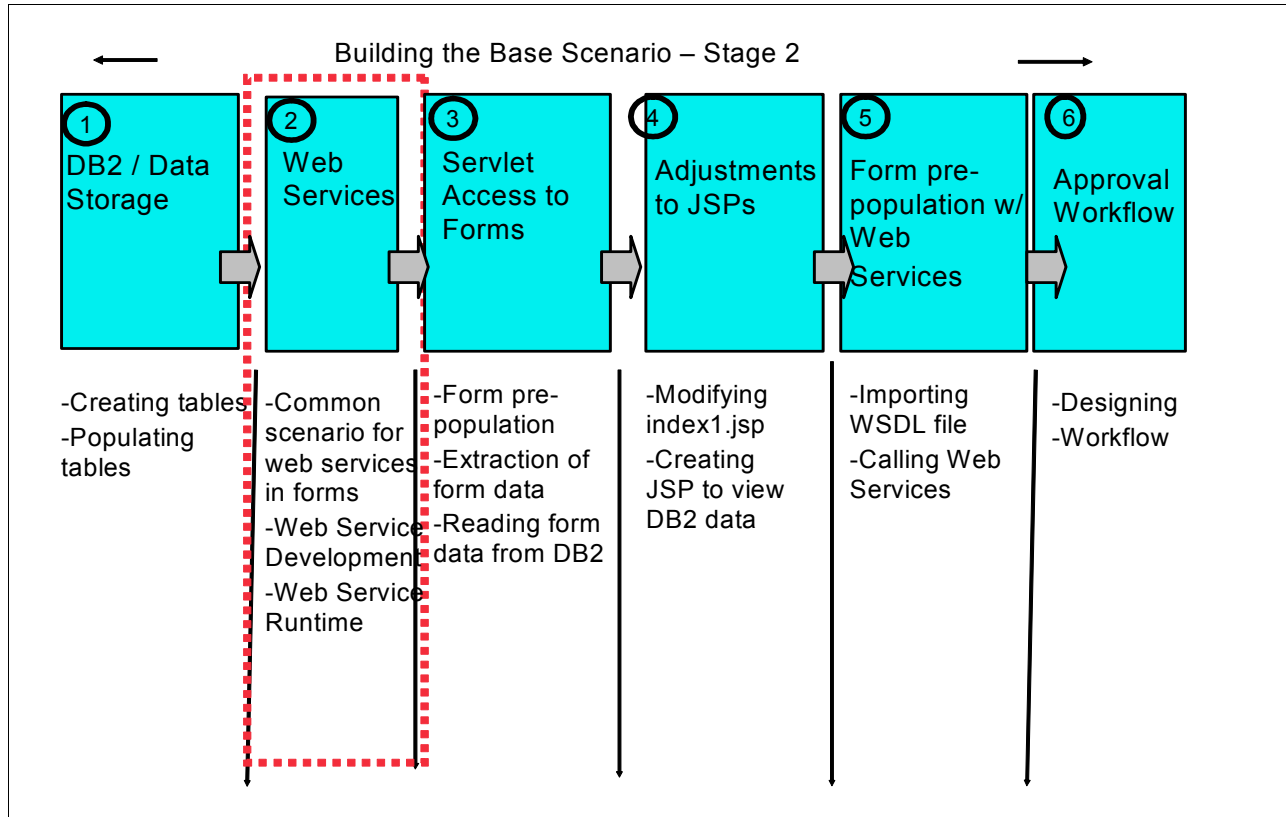


Figure 5-2 Overview of major steps involved in building Stage 2 of base scenario application

5.3.2 Web services integration

Web service integration in Workplace Forms content is a specific way to dynamically get hold of content specific data (as custom object for a selected object) or real time data changing during the time working in the form (such as actual stock prices or exchange rates). A Web service integration (for simple cases, which make up more than 90% of all Web service use cases) is a rather simple task to do, since there are efficient tools available to support Web service development and implementation.

In a global context, Web services are a convenient method for data exchange between different systems. In most cases we find here a “request / response” use case. The Web service consumer submits a request to the provider, and the provider sends a response back. This works regardless of platform, operating system, and programming languages of both the service provider and the service consumer, because the used transmission protocol and internal data structure of the request and response are defined in an independent format, the Web Service Description Language (WSDL).

Web service integration targets mainly the used form (implementing the Web service calls in XFDL structure) and the service provider (usually an http/soap proxy server connected to the data source). So the developed modules are not necessarily related to the application server dealing with the forms handling. That is why the created provider code is not included in the Forms application but is available as separate stand-alone applications ready to run on different servers.

Common scenario for Web services in forms

Workplace Forms Viewer contains a built in Web service consumer. The consumer is configured by a WSDL contained in the launched form. Usually this WSDL is included in design time, but it can be changed all over the forms life cycle. Calling a Web service, the Workplace Forms Viewer will read the WSDL, and configure the request message to send and analyze the incoming response according to the actual Web service description included in the form. This capabilities suggest to use the Workplace Viewer as Web service consumer and create the Web service provider for the server maintaining back-end data.

We will use Web services to retrieve data from back-end systems. It is possible to write back data as well, but in this project we did not implement those services. The way of implementation would be the same for both cases — only the meaning (and amount) of data transmitted in both directions would vary (see Table 5-5).

Table 5-5 Web service data flow

	Data retrieval Web service	Data submission Web service
Request	No data, one or more parameters for data selection	One or more submitted fields and a record selection
Response	One or multiple retrieved objects	Simple response (OK / NOK) or extended transaction information as logs, created IDs, processed data. and so forth.

There are many different approaches to define the internal data structure of an request:

- ▶ Service style (document, document-wrapped, or RPC)
- ▶ Encoding (encoded or literal)

Depending on the chosen style for the service, objects of different structures can be declared. Implementing Web services for Workplace Viewer, we will have to match the following restrictions:

- ▶ Web services must not include the underscore character (_) in either service or port names, but can include it in operation names.
- ▶ Web services must not use mandatory headers, as defined by the soap:mustUnderstand tag name.
- ▶ Web services are restricted to the 46 primitive data types as defined by schema. Third party extensions to the primitive data types are not supported.
- ▶ Web services may use basic or digest authentication. In either case, authentication must be performed before calling any functions in the Web service. This is accomplished by calling the setNetworkPassword function, which is created in a package with the same name as the Web service.
- ▶ Web services running in Workplace Forms 25. viewer do not support SSL.

Some of the constrains above (no SSL, authentication type, basic types only, naming conventions) have to be considered when connecting to an existing Web service. Missing SSL support will give hard limitations to the sensibility of transferred data.

Note: These restrictions apply only to the built-in Web service implementation. However, we can develop Web service consumers with other profiles and include them as a Java library in a custom IFX extension (written in Java or C). These extensions can be called from XFDL computes as additional function libraries.

Finally, we created three Web services (for each of the three potentially used object types, one dedicated Web service) with five service methods (Table 5-6).

Table 5-6 List on implemented Web service operations

Object (Web Service Description File)	Operation name	Purpose
Employee data (EmployeeInfo.wsdl)	GETEMPLOYEEINFO	Gathering employee detail data based on employee number. Returns a complex object with detail data. Not used in the project, since employee data was finally added to the form as prepopulation not using a Web service.
Customer data (CustomerInfo.wsdl)	GETCUSTINFO	Gathering customer detail data based on customer number. Returns a complex object with detail data. Activated on any change in customer selection field.
	GETCUSTOMERLIST	Imports complete customer list (name and customer account number) as a single string. Activated on first form load.
Product data (ProductInfo.wsdl)	GETPRODUCTINFO	Gathering product detail data based on item number. Returns a complex object with detail data. Activated on any change in an item selection field.
	GETPRODUCTLIST	Imports complete product list (name and product number) as a single string. Activated on first form load.

Web service development

Even though this chapter focuses on Web service based integration with a J2EE based environment, we begin to see and realize the eventual consequences for the Domino integration scenario discussed in Chapter 9, “Domino integration” on page 273. The main reason was to define the Web services implementation in a way usable for both environments with no changes or only a minimum of changes in the created form.

Very often the used development tools will make up additional constraints to the concrete implementation. As Web service provider, we selected Tomcat 5.0 gathering data from a DB2 server (J2EE integration scenario) and Domino 7.0 Server (Domino integration scenario). As Web service development tools for these platforms, in the project we use IBM Rational Application Developer Version 6 (RAD6) and Domino Designer® Version 7.0.

Web service development can be done basically following one of two strategies:

- ▶ “Bottom up” = create a WSDL description of a Web service based on a given class or data structure
- ▶ “Top down” = create a skeleton of classes and data objects based on a given WSDL description

In the project we will use both techniques. Due to the project target — show an integration scenario using WebSphere Application Server (WAS)/Portal and a complementary scenario using Domino — we will have to implement two Web server providers for each service. The idea is to implement a service in one environment bottom-up, and based on the generated WSDL file, implement the second provider top-down. This should ensure that we can use the same form (the same Web service consumer) in both environments.

So we always started building a class in Domino Designer, exported the WSDL to RAD 6, and implemented the complementary service provider using Java for the WebSphere Application Server (WAS)/Portal environment. The reason for starting with Domino was that we had required classes in Domino available. There was no hard stop to do the work starting with J2EE and RAD6.

The reason for starting bottom-up is simple: Defining a valid WSDL for a Web service manually is really demanding work, which can be done for simple objects much better using a tool with bottom-up capabilities.

Figure 5-3 shows the development roadmap used in the redbook project.

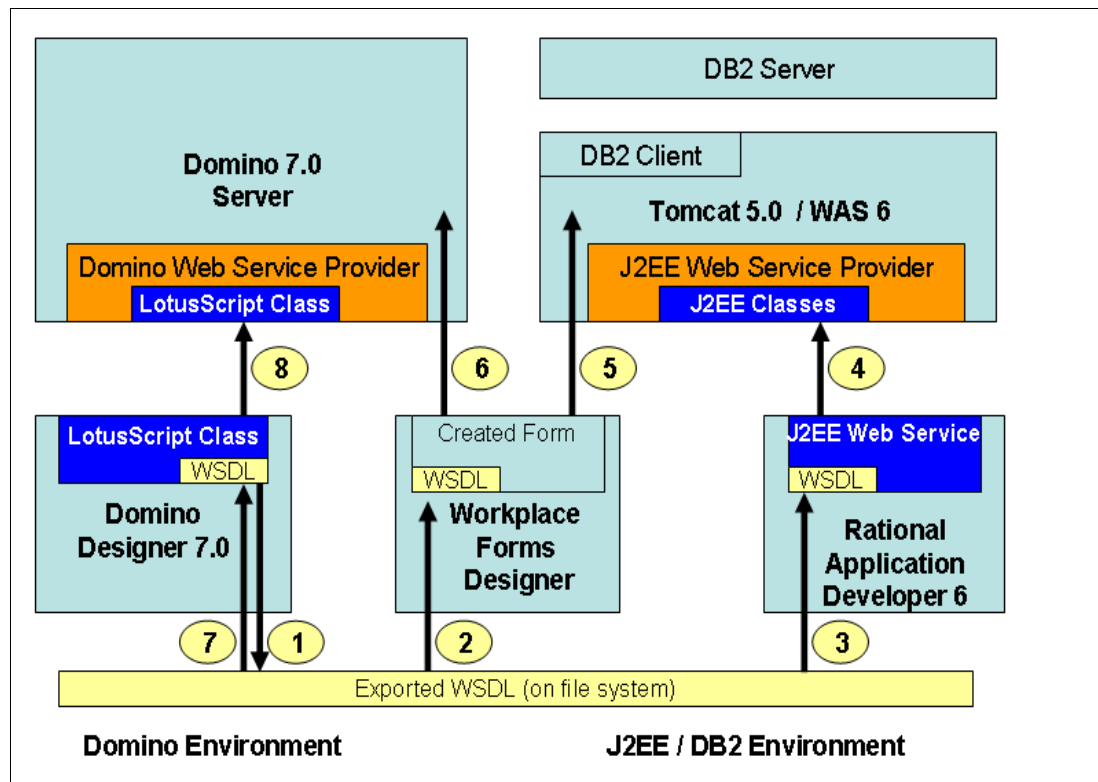


Figure 5-3 Architecture of Web service design (development roadmap)

These are the steps shown in Figure 5-3:

1. Create the object class definition and WSDL export to file system; make cosmetic changes (such as eliminating Domino from any names used in the WSDL file).
2. WSDL import to Workplace Forms Designer (Tools / Enclose WSDL) and coding for Web service invocation in the XFDL form and create a function calling the service.
3. WSDL import to RAD 6 and J2EE Web service provider / test client development.
4. Deployment of Web service consumer as a J2EE application (WAR file) to WebSphere Application Server (WAS) / Tomcat server.
5. Deployment of developed form as template to WebSphere Application Server (WAS) server file system to make it available to the J2EE integration scenario.
6. Deployment of developed form as template to Domino database to make it available to Domino Integration scenario (Chapter 9, "Domino integration" on page 273).
7. Re-import the generalized WSDL and do the necessary Lotus Script development in the created class skeleton (the top-down approach).

8. Deployment of the Web service to Domino server (which is automatically available, when the development template resides on an http-enabled Domino server).

Web service development in Forms Designer is limited to three rather basic actions using the WSDL created in step 2 (include WSDL in the form, create a XFDL function call to invoke the service with a specified target for the response object and data retrieval from the received response object. Using the built-in Web service consumer there is no need (and no chance) for additional coding in low level procedures. For details on the necessary steps, see Chapter 2, “Features and functionality” on page 17.

For details how to create Web service providers in Domino, see *Domino 7.0 Designer Help*, in the chapter, “Programming Domino for Web Applications / Web services”.

For details on how to create Web services in RAD6, see *Eclipse Help in RAD6* — choose from the menu, **Help / Help Contents**, and select the chapter, “Developing Web services.”

The main steps for RAD6 are as follows:

1. Create a new Dynamic Web Project.
2. Import a WSDL to the project root.
3. Right-click the WSDL, choose **Web Services / Generate Java Bean skeleton**.
4. Select **Generate a proxy** and **Create Test Client** (optional, but recommended).
5. Click **Finish**.
6. Search for**SoapBindingImpl.java** file in
 <project>/JavaResources/JavaSource/<your package> folder.
7. Edit the generated class skeleton(s) to create the desired return object.

See Example 5-9 and Example 5-10 for the coding involved.

Example 5-9 Empty class skeleton after creation in file ProductCatalogPortSoapBindingImpl.java

```
/**
 * ProductCatalogPortSoapBindingImpl.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package WPFormsRedpaper;

import forms.cam.itso.ibm.com.DB2ConnectionForms;

public class ProductCatalogPortSoapBindingImpl implements
WPFormsRedpaper.ProductCatalogPortType{
    public java.lang.String GETPRODUCTLIST(java.lang.String FILTER) throws
java.rmi.RemoteException {
        return null;
    }

    public WPFormsRedpaper.PRODUCT GETPRODUCTINFO(java.lang.String IT_ID) throws
java.rmi.RemoteException {
        return null;
    }
}
```

Example 5-10 Web service implementation after completing with implementation code

```
/**
 * ProductCatalogPortSoapBindingImpl.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package WPFormsRedpaper;

import forms.cam.itso.ibm.com.DB2ConnectionForms;

public class ProductCatalogPortSoapBindingImpl implements
WPFormsRedpaper.ProductCatalogPortType{
    public java.lang.String GETPRODUCTLIST(java.lang.String FILTER) throws
java.rmi.RemoteException {
        String prodList = DB2ConnectionForms.getInventoryList();
        return prodList;
    }

    public WPFormsRedpaper.PRODUCT GETPRODUCTINFO(java.lang.String IT_ID) throws
java.rmi.RemoteException {
        WPFormsRedpaper.PRODUCT it = new WPFormsRedpaper.PRODUCT();
        String[] itemData = DB2ConnectionForms.getInventoryItemData(IT_ID);
        it.setIT_ID(itemData[0]);
        it.setIT_NAME(itemData[1]);
        it.setIT_PRICE(Double.valueOf(itemData[2]).doubleValue());
        it.setIT_STOCK(Integer.valueOf(itemData[3]).intValue());
        return it;
    }
}
```

Creating Web services for Tomcat 5.0 and WebSphere Application Server (WAS) 6.0, we could use all default settings proposed by RAD6. Developing for WebSphere Application Server (WAS) 5.1, we had to manually set the target protocol to *Apache AXIS 1.0* protocol (*IBM SOAP* and *IBM WebSphere* protocol did not accept the created WSDL files). Nevertheless, the generated classes did not compile to acceptable results, since they did not contain suitable serializers/deserializers for the created classes.

This problem could be solved only by extended debugging of the generated code or a redefinition of the used WSDL (such as implementing a Web service using another style, encoding, or object structure). As a result, in the project, the Web services were generated for WebSphere Application Server (WAS) 6 / Tomcat 5 using the initial WSDL files and deployed on the Tomcat 5.0 server coming with the IBM Workplace Forms Server on the same machine as WebSphere Application Server (WAS) 5.1.

For the demonstration of Forms and Web service integration capabilities, a main goal was to reuse the same WSDL description for different Web service providers. The minimum change required to make a Web service running in different environments is the adjustment of the Web service endpoint definition (URL and port to the target Web service provider).

Forms implementation does not provide a dedicated functionality to change parts of WSDL information, but this can be done using different techniques outside XFDL language and API (such as text parsing and / or Forms API methods) in different phases of a forms lifecycle:

- In design time (before importing the WSDL file to Workplace Form in Workplace Forms Designer) - we used this approach for J2EE / DB2 integration scenario to switch the URL

from default URL (<http://localhost>) to the target URL of the J2EE Web service provider just after WSDL export to file system.

- In deployment time (when storing the form to the template repository in the production system) — not used in this project.
- In runtime (for example, just before template download to the Forms Viewer as a part of form prepopulation) — we use this approach in the Domino environment.

Additionally, we edited the WSDL, changing some internal names and namespace containing the original source platform (“Domino”) before beginning any Web service development. These names were changed to a project related name such as “WPFormsRedpaper”.

An important note for development is to check the tolerance of all involved systems to the structure of the deployed WSDL and the structure of the exchanged soap messages. We had to make minor adjustments to the WSDL before importing to RAD6 (removing some empty structures in type definitions), but this step can cause a complete rework of the Web service definition in other projects. Processing the full development cycle for one target system without a compatibility check with all other potential components involved can be risky.

Web service runtime

Having all components created, we can deploy the components to the different systems and run a first test.

Figure 5-4 shows data flow during runtime. Web services are called in our sample form directly after first form load (reading choices lists for customers and products from the repository) and during work in the form (gathering detail data after selecting a product or customer). For each Web service, call activities 2-8 are processed.

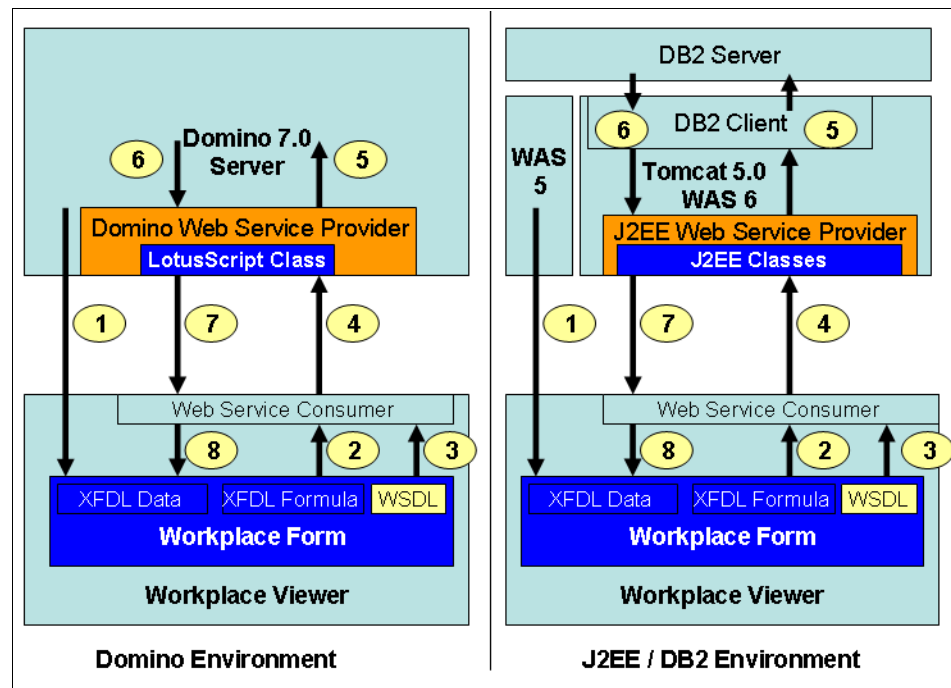


Figure 5-4 Runtime Web service activities (data flow)

Figure 5-4 shows the following steps for both the J2EE and Domino environments:

1. Form download (with or without value prepopulation initiated on the server side)

2. Web service invocation by a XFDL formula (on first load or any other form events, like a button clicked, or a value change in a field), including transfer on all input parameters for this request
3. Web service consumer pre-configuration according to the selected WSDL (target URL, message structures)
4. Compose and submit request (done by built-in Web service client), Web service request parsing on Web service provider (WebSphere Application Server (WAS) or Domino)
5. Data query to target source (DB2 or Domino) coded in the J2EE or Lotus Script classes on provider side
6. Data retrieval form data source (DB2 or Domino) coded in the J2EE or Lotus Script class on provider side
7. Response message composition on provider side, decomposition in Forms Viewer Web service consumer
8. Data storage to the target specified in the initiating Web service invocation function executed in step

As Web service target to store the response object, XFDL can assign only values to one single object. If the response contains a complex object, it will overwrite the assigned target XFDL node with the content of the response message. This can (and will) change locally the structure of the XFDL document, if the initial structure for the target is not the same as in the incoming message (changed number, order, names of child elements additional attributes and missing or additional XML subtrees are possible. That is why a best practice is to inspect and double-check not only the Web service description but the incoming responses from the provider too. Different providers can create differently structured response message structures based on identical WSDLs and incoming messages.

This happened, for example, when switching between a Domino based and a J2EE based Web service provider. See Example 5-11, showing customer data as one complex object containing eight elements (CUST_ID, CUST_NAME, CUST_AMGR and some others):

Example 5-11 Extract from WSDL for CustomerInfo object

```

.....
<wsdl:types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://WPFormsRedpaper">
    <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
    <complexType name="CUSTOMER">
      <sequence>
        <element name="CUST_ID" type="xsd:string"/>
        <element name="CUST_NAME" type="xsd:string"/>
        <element name="CUST_AMGR" type="xsd:string"/>
        <element name="CUST_CONTACT_NAME" type="xsd:string"/>
        <element name="CUST_CONTACT_POSITION" type="xsd:string"/>
        <element name="CUST_CONTACT_PHONE" type="xsd:string"/>
        <element name="CUST_CONTACT_EMAIL" type="xsd:string"/>
        <element name="CUST_CRM_NO" type="xsd:string"/>
      </sequence>
    </complexType>
  </schema>
</wsdl:types>
<wsdl:message name="GETCUSTINFORequest">
  <wsdl:part name="CUST_ID" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="GETCUSTINFOResponse">
  <wsdl:part name="GETCUSTINFOReturn" type="impl:CUSTOMER"/>

```

```

</wsdl:message>
<wsdl:portType name="CustomerInfo">
....
<wsdl:operation name="GETCUSTINFO" parameterOrder="CUST_ID">
  <wsdl:input message="impl:GETCUSTINFORequest" name="GETCUSTINFORequest"/>
  <wsdl:output message="impl:GETCUSTINFOResponse" name="GETCUSTINFOResponse"/>
</wsdl:operation>
</wsdl:portType>
....

```

The corresponding data instance in XFDL form intended to receive the Web service response message containing the employee data looks like this (Example 5-12).

Example 5-12 XFDL data instance in form template containing CustomerInfo object

```

<xforms:instance xmlns="" id="FormCustomerData">
  <GETCUSTINFOResponse>
    <GETCUSTINFOReturn>
      <CUST_ID></CUST_ID>
      <CUST_NAME></CUST_NAME>
      <CUST_AMGR></CUST_AMGR>
      <CUST_CONTACT_NAME></CUST_CONTACT_NAME>
      <CUST_CONTACT_POSITION></CUST_CONTACT_POSITION>
      <CUST_CONTACT_PHONE></CUST_CONTACT_PHONE>
      <CUST_CONTACT_EMAIL></CUST_CONTACT_EMAIL>
      <CUST_CRM_NO></CUST_CRM_NO>
    </GETCUSTINFOReturn>
  </GETCUSTINFOResponse>
</xforms:instance>

```

This structure reflects exactly the estimated response object including message name and operation name (GETCUSTINFOResponse and GETCUSTINFOReturn). The incoming message from Domino looks like this (Example 5-13).

Example 5-13 Response message sent by Domino Web service provider

```

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
  <soapenv:Body>
    <ns1:GETCUSTINFOResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://WPFormsRedpaper">
      <GETCUSTINFOReturn xsi:type="ns1:CUSTOMER"><CUST_ID
        xsi:type="xsd:string">100001</CUST_ID>
        <CUST_NAME xsi:type="xsd:string">XXX</CUST_NAME>
        <CUST_AMGR xsi:type="xsd:string">1001</CUST_AMGR>
        <CUST_CONTACT_NAME xsi:type="xsd:string">Mr. Last</CUST_CONTACT_NAME>
        <CUST_CONTACT_POSITION xsi:type="xsd:string">tester</CUST_CONTACT_POSITION>
        <CUST_CONTACT_PHONE xsi:type="xsd:string">PH</CUST_CONTACT_PHONE>
        <CUST_CONTACT_EMAIL xsi:type="xsd:string">xx@xxx</CUST_CONTACT_EMAIL>
        <CUST_CRM_NO xsi:type="xsd:string">200001</CUST_CRM_NO></GETCUSTINFOReturn>
      </ns1:GETCUSTINFOResponse>
    </soapenv:Body>
  </soapenv:Envelope>

```

The J2EE based Web service provider on the same request returns messages like this (Example 5-14).

Example 5-14 Response message sent by J2EE Web service provider

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:GETCUSTINFOResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://WPFormsRedpaper">
      <GETCUSTINFOReturn href="#id0"/>
    </ns1:GETCUSTINFOResponse>
    <multiRef id="id0" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:CUSTOMER"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns2="http://WPFormsRedpaper">
      <CUST_ID xsi:type="xsd:string">100001</CUST_ID>
      <CUST_NAME xsi:type="xsd:string">Workplace Early Adopter Inc</CUST_NAME>
      <CUST_AMGR xsi:type="xsd:string">1000</CUST_AMGR>
      <CUST_CONTACT_NAME xsi:type="xsd:string">Mary F Thompson</CUST_CONTACT_NAME>
      <CUST_CONTACT_POSITION xsi:type="xsd:string">DeptMgr</CUST_CONTACT_POSITION>
      <CUST_CONTACT_PHONE xsi:type="xsd:string">1 756-568-123</CUST_CONTACT_PHONE>
      <CUST_CONTACT_EMAIL
xsi:type="xsd:string">Mary.F.Thompson@Workplace-Early-Adopter.com</CUST_CONTACT_EMAIL>
      <CUST_CRM_NO xsi:type="xsd:string">200002</CUST_CRM_NO>
    </multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

The significant differences between both messages are shown in *italics*: The operation contains a reference to the object only, the object is then attached as a “multiRef” element. Creating those messages is a valid behavior used commonly to create messages (potentially) containing circular references between objects. This is not the case in our example, but in the result, the data instance in the XFDL form changes after a Web service call to a J2EE provider like this (Example 5-15) .

Example 5-15 Changed data instance structure after J2EE Web service call

```
<xforms:instance xmlns="" id="FormCustomerData">
  <GETCUSTINFOResponse>
    <multiRef xmlns:ns2="http://WPFormsRedpaper"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="id0" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:CUSTOMER">
      <CUST_ID xsi:type="xsd:string">100002</CUST_ID>
      <CUST_NAME xsi:type="xsd:string">Portal Application Surfacing</CUST_NAME>
      <CUST_AMGR xsi:type="xsd:string">1001</CUST_AMGR>
      <CUST_CONTACT_NAME xsi:type="xsd:string">Hui Kwan</CUST_CONTACT_NAME>
      <CUST_CONTACT_POSITION xsi:type="xsd:string">Director</CUST_CONTACT_POSITION>
      <CUST_CONTACT_PHONE xsi:type="xsd:string">+43 623-644</CUST_CONTACT_PHONE>
      <CUST_CONTACT_EMAIL
xsi:type="xsd:string">Hui.Kwan@p-app.surf.org</CUST_CONTACT_EMAIL>
      <CUST_CRM_NO xsi:type="xsd:string">200003</CUST_CRM_NO>
    </multiRef>
  </GETCUSTINFOResponse>
</xforms:instance>
```

The outcome is a changed tag name (<GETCUSTINFOReturn> to <multiRef>) that must be considered as creating bindings to the inner structure of the object.

For this project, the workaround used was relative addressing of any inner elements and making code tolerant to changes of the relevant tags:

```
[GETCUSTINFOResponse][0][CUST_ID].value
```

in place of:

```
[GETCUSTINFOResponse][GETCUSTINFOReturn][CUST_ID].value
```

Another workaround would be a redefinition of the Web service description using an explicit RCP datastructure with the following object definitions (Example 5-16).

Example 5-16 Redefined WSDL using simple data types only

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://WPFormsRedpaper"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://WPFormsRedpaper" xmlns:intf="http://WPFormsRedpaper"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
  </wsdl:types>
  <wsdl:message name="GETCUSTINFORequest">
    <wsdl:part name="CUST_ID" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="GETCUSTINFOResponse">
    <part name="CUST_ID" type="xsd:string"/>
    <part name="CUST_NAME" type="xsd:string"/>
    <part name="CUST_AMGR" type="xsd:string"/>
    <part name="CUST_CONTACT_NAME" type="xsd:string"/>
    <part name="CUST_CONTACT_POSITION" type="xsd:string"/>
    <part name="CUST_CONTACT_PHONE" type="xsd:string"/>
    <part name="CUST_CONTACT_EMAIL" type="xsd:string"/>
    <part name="CUST_CRM_NO" type="xsd:string"/>
  </wsdl:message>
  <wsdl:operation name="GETCUSTINFO" parameterOrder="CUST_ID">
    <wsdl:input message="impl:GETCUSTINFORequest" name="GETCUSTINFORequest"/>
    <wsdl:output message="impl:GETCUSTINFOResponse" name="GETCUSTINFOResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

Doing so would result in a complete new design of the J2EE Web service implementation class switching from accessing objects with attributes of a basic type (String, int, ...) to work with placeholder parameters (java.lang.String placeholder, ...). We did not take this approach. After rewriting the binding definitions stored in XFDL form for customer and product detail data, we considered the Web service implementation complete.

5.4 Servlet access to Form data

In this section we describe how to provide servlet access to the Form data.

5.4.1 Where we are in the process: Building Stage 2 of the base scenario

The diagram in Figure 5-5 is intended to provide an overview of where we are within the key steps involved to build Stage 2 of the base scenario. This focuses on adding storage capabilities to DB2, incorporating Web services, modifying the JSPs and adding an approval workflow.

As a starting point, we created a copy in the J2EE project from the servlet (SubmissionServlet1) and all JSPs and gave them new names (SubmissionServlet, dirlisting.jsp and similar). We did this so that we could continue work on the created code structures without destroying the Stage 1 application.

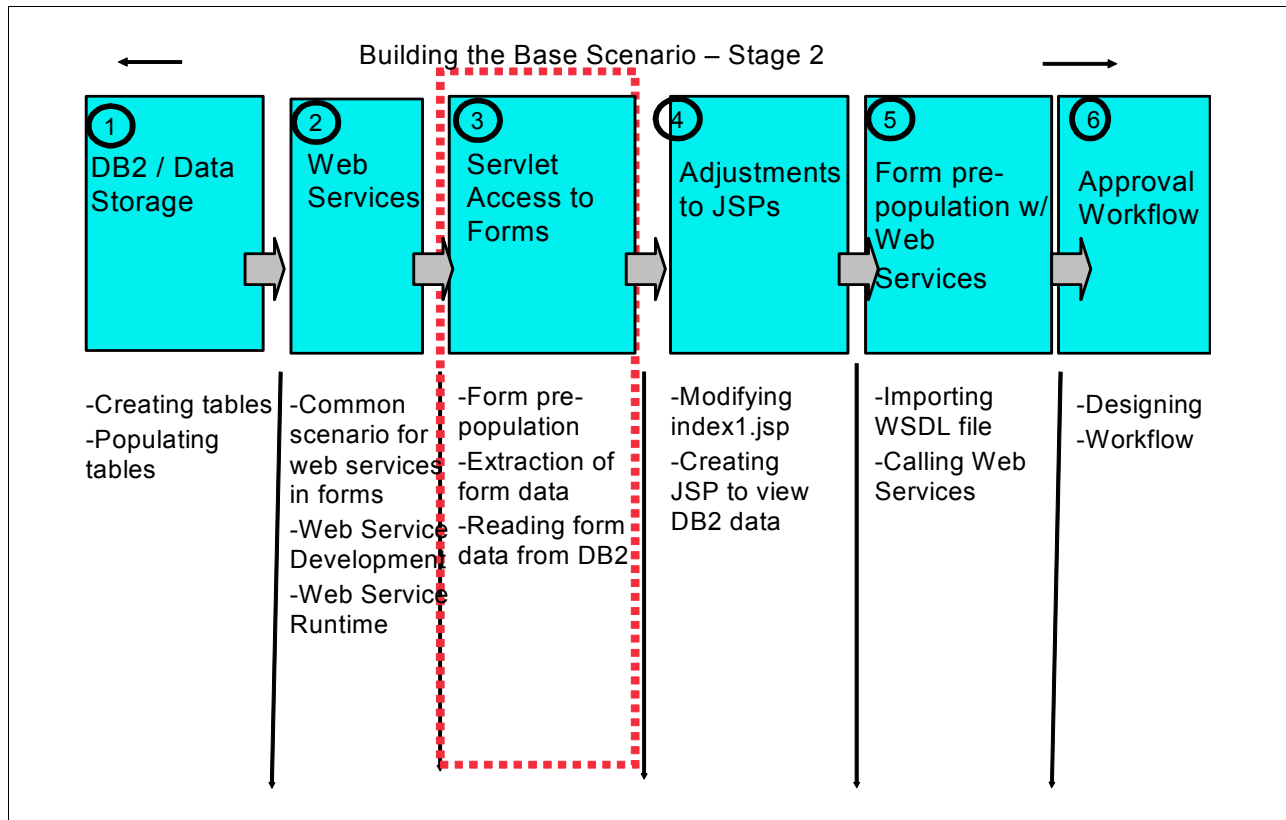


Figure 5-5 Overview of major steps involved in building Stage 2 of base scenario application

5.4.2 Servlet Access to form data (prepopulation / data retrieval)

While Web service integration is a good choice for runtime data exchange during work in a form opened in the Forms viewer, data prepopulation and retrieval is done on server side before the form is presented to the end user and after a submitted form is received on the server. Both actions are basically an access to the form available as a file or a stream.

There are two different access methods:

- ▶ Access through Workplace Forms API
- ▶ Access by text parsing

In this redbook we provide both techniques. In the servlet context we use API based methods only. For Domino integration we discuss the text parsing methods as well. Both scenarios will access the form in form open event (filling a template with initial data on first open) and data extraction on form submit.

Some developers may prefer to use a third party XML parser to interface with their forms. Depending on the parser and the types of tasks to be performed, this may provide some benefits in terms of speed or developer comfort (if the developer has a great deal of experience with XML parsers).

However, the API is able to perform some tasks that are impossible for an XML parser. For example, computes can continue to run in the form (or be activated by the API) while the form is in memory, so any form data that is dependent on continuously evaluated computes will remain accurate. Additionally, since most applications use compressed forms, which cut down transmission speeds and use of disk space dramatically, the API is able to automatically decode Base 64 data and uncompresses forms as it reads them. With an XML parser, you must force it to decode and uncompress the forms to run them.

The API also provides methods for verifying and handling digital signatures. Most applications need to verify signatures on the server side, and occasionally even apply signatures on the server. These tasks are virtually impossible to perform without the Workplace Forms API. The API can also encode and decode data items stored such as images, enclosures, and digital signatures. In the case of images and enclosures, this means that using the API it is possible to extract attachments or images from the form and store them separately on the server, or insert attached files into forms as they are sent out to the user.

5.4.3 Form prepopulation

In this stage we will provide server side prepopulation (Figure 5-6) using the Forms API. Server side form prepopulation takes place before a user opens a form. The engaged module (such as a servlet) takes control over the initial XFDL form (stored form or empty template as file or stream), accesses the internal values (using API or text parsing) and submits the changed XFDL file / stream to the invoking instance (the viewer or any other module requesting the form).

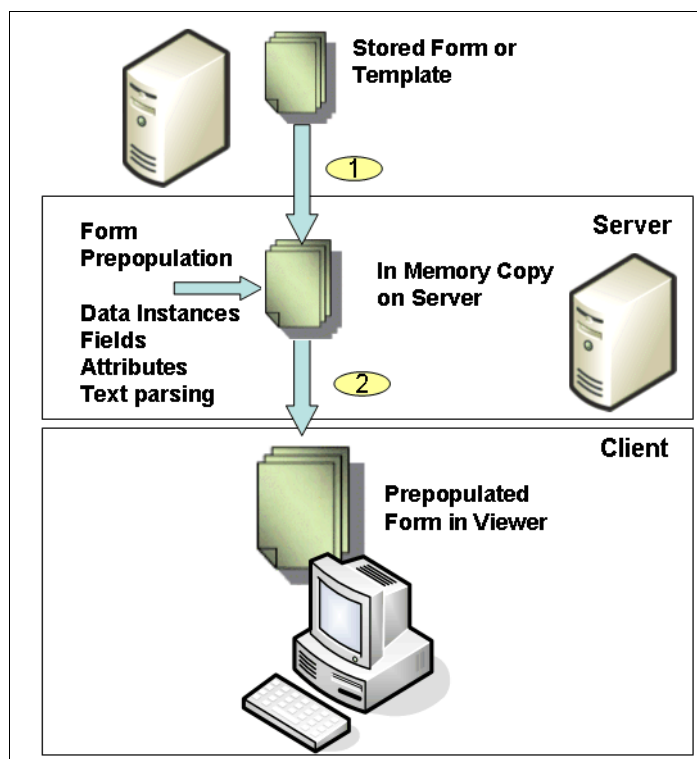


Figure 5-6 Server side form prepopulation

The basic idea is here to get the original XFDL information (template), transform it adding external data to the form and then send it to the client. Assuming an invocation URL fired from the client browser to open a form, in a servlet environment, prepopulation will occur usually in the doGet method. The URL must now point to the servlet (not directly to the stored template on file system) and contain a reference to the chosen template.

Form download based on a POST action fired by a browser or another system would be handled in the doPost method. This will not occur in our scenario, but it is possible to do so. The coding in Example 5-17 gives a simple example of how prepopulation can be done in the doGet method.

Example 5-17 Prepopulation executed in doGet servlet method

```

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    try {
        //get the template path from the request parameter "template="
        String formTemplate = request.getParameter("template");
        FileInputStream fis = new FileInputStream(formTemplate);

        XFDL theXFDL = IFSSingleton.getXFDL();
        FormNodeP theForm = theXFDL.readForm(fis,XFDL.UFL_SERVER_SPEED_FLAGS);

        ....

        //Set some internal values like this
        //address the internal element to update using xfdl specific pathes.
        String mgrThreshold = "10000";
        theForm.setLiteralByRefEx(null,
"global.global.xmlmodel[xfdl:instances][3][null:BusinessRuleParams][null:QuoteLevelOneThres
hold]",0, null, null, mgrThreshold);

        ....

        //Return the prepopulated form
        response.setContentType("application/vnd.xfdl");
        theForm.writeForm(response.getOutputStream(), null, 0);

    } catch (Exception doGetE) {
        System.out.println("SubmissionServlet: doGet: Exception processing request: "
            + doGetE.toString());
        returnText(response,"SubmissionServlet: doGet: Exception occurred: "
            + doGetE.toString(), "text/plain");
    }
}

```

As discussed in Stage 1, it is a best practice to access form data using data instances, not field values. The example above does access an data instance (in the code, see reference `global.global.xmlmodel[xfdl:instances][3][null:BusinessRuleParams][null:QuoteLevelOneThreshold]`), but the instance is selected by position ([3]) - not by name. This can break when instances are moved in the form. As in value extraction, there is a better way to reference the instance by instance ID.

The next example shows how to address a field value synchronized with an XFDL data instance by position or by name. See the following data instance in the XFDL form (Example 5-18).

Example 5-18 XFDL structure with the data instance OrderNumber / field ORD_ID to prepopulate

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XFDL xmlns="http://www.PureEdge.com/XFDL/6.5" ..... >
  <globalpage sid="global">
    <global sid="global">
      .....
      <xmlmodel>
        <instances>
          ....
          <xforms:instance xmlns="" id="OrderNumber">
            <OrderNumber>
              <ORD_ID></ORD_ID>
            </OrderNumber>
          </xforms:instance>          ....
        </xmlmodel>
      </global>
    </globalpage>
```

To assign a value to this field ORD_ID in data instance OrderNumber, the code in Example 5-19 could be used (reference by position to the data instance):

Example 5-19 Assigning a value (orderNumber) to a form node in a XFDL data instance

```
//theForm contains a reference to the form root node
private static void setOrderNumber(FormNodeP theForm) throws UWException {
  //get orderNumber from DB2
  String orderNumber = DB2ConnectionForms.getNewOrderNumber();
  //set ordernumber in data instance
  theForm.setLiteralByRefEx(null,
    "global.global.xmlmodel[xfdl:instances][7][null:OrderNumber][null:ORD_ID]",
    0, null, null, orderNumber);
}
```

The code used is quite simple, but has one disadvantage: We cannot reference the data instance by name — all data instances have the tag name “instance” and no SID assigned. They are differentiated by the attribute ID. This attribute cannot be parsed by those methods as setLiteralByRefEx. As a result, we had to code a relative reference to the object:

```
global.global.xmlmodel[xfdl:instances][7][null:OrderNumber][null:ORD_ID]
```

Here, the part [7] indicates to access the 8th XFDL data instance. Adding or deleting data instances can break this reference.

There are at least two ways to overcome this problem. We could use a parsing algorithm to check all data instances in a loop for the required instance ID, detect the instance number, and write back data. This code is not shown here, but is possible to create. The other way is to use a dedicated API function for data instance updates operating with the instance ID as parameter. See the code in Example 5-20.

Example 5-20 Updating a data instance directly with an XML fragment

```
private static void setOrderNumber(FormNodeP theForm) throws UWException {
  //get db2 data as XML fragment
  String orderNumberXML = DB2ConnectionForms.getNewOrderNumberXML();
  //returns something like "<OrderNumber><ORD_ID>1000016</ORD_ID></OrderNumber>";

  //convert the String into a Stream
  StringReader sr = new StringReader(orderNumberXML) ;
}
```

```

// update the instance by name
// public void encloseInstance(theInstanceID,theStream,theFlags,theScheme,
//     theRootReference,theNSNode,replaceNode)
theForm.encloseInstance("OrderNumber", sr , 0, null, "[0]", null,true);
}

```

The method `encloseInstance` overwrites an existing element in an instance with a new XML fragment. This makes it easy to update much more complex instances than this in the example. The only restriction for this method is to match exactly the tags in the updated elements to save all existing bindings to other objects in the form. Be aware of the addressing used in the call. The parameter `theRootReference` is filled with "[0]" - a pointer to the first node inside the data instance. See Example 5-21 to understand this concept.

Example 5-21 XML fragment representing the data instance and the first element <OrderNumber>

```

<xforms:instance xmlns="" id="OrderNumber">
  <OrderNumber>
    <ORD_ID></ORD_ID>
  </OrderNumber>
</xforms:instance>

```

The way this works is as follows:

1. The parameter `theInstanceName` (filled with the value "OrderNumber") points the update method to the Tag `<xforms:instance xmlns="" id="OrderNumber">`.
2. The parameter `theRootReference` filled with "[0]" points to the first inner element, that is `<OrderNumber>` in the example.
3. This element and all child elements are updated (overwritten) by the update.

We found that when updating the complete instance, including the `<xforms:instance>` elements would cause namespace problems with the xforms namespace.

The following code snippets will give you an idea of how to use the `theRootReference` parameter for internal addressing. All the samples below would work (Example 5-22).

Example 5-22 Different ways to address data instance objects

```

// set the entire first element in data instance - addressing by position
String orderNumberXML = "<OrderNumber><ORD_ID>1000016</ORD_ID></OrderNumber>";
StringReader sr = new StringReader(orderNumberXML) ;
theForm.encloseInstance("OrderNumber", sr , 0, null, "[0]", null,true);

// set the a named element in data instance - addressing by name
String orderNumberXML = "<OrderNumber><ORD_ID>1000016</ORD_ID></OrderNumber>";
StringReader sr = new StringReader(orderNumberXML) ;
theForm.encloseInstance("OrderNumber", sr , 0, null, "[0][null:ORD_ID]", null,true);

// set the value only - mixed addressing
String orderNumberXML = "<ORD_ID>1000016</ORD_ID>";
StringReader sr = new StringReader(orderNumberXML) ;
theForm.encloseInstance("OrderNumber", sr , 0, null, "[0][null:ORD_ID]", null,true);

// set the value only - addressing by position
String orderNumberXML = "<ORD_ID>1000016</ORD_ID>";
StringReader sr = new StringReader(orderNumberXML) ;
theForm.encloseInstance("OrderNumber", sr , 0, null, "[0][0]", null,true);

```

Knowing how to prepopulate data, we can explore how to extract data from a submitted form.

5.4.4 Extraction of form data and storage of entire form

Extracting data from a form will be done as in the Stage 1 doPost method. In Stage 2, we will:

- ▶ Extract much more data (state changes and order detail data).
- ▶ Complete the extracted data with additional history data (time stamps for approvals and approver IDs).
- ▶ Store the data to DB2.
- ▶ Store the entire form to DB2.

The main methods for data extraction stay the same as in Stage 1. See the code in the extended scenario for doPost (Example 5-23).

Example 5-23 extended scenario for doPost

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    XFDL theXFDL = null; //The form
    FormNodeP theForm = null; //Represents nodes of the XFDL form
    String formString = null; //String representation of the form. Used for
    // the 'bounceback' feature.

    try {
        //get action from url - aimed to offer different processin modes defined in
        //the form submit button target url
        action = request.getParameter("action");
        if (action.equalsIgnoreCase("store")) {
            // Read in the form
            theXFDL = IFSSingleton.getXFDL();
            theForm = theXFDL.readForm(request.getInputStream(),
                XFDL.UFL_SERVER_SPEED_FLAGS);
            // Validate form signatures. If any signatures are invalid, then
            if (!allSignaturesAreValid(theForm)) {
                System.out.println("SubmissionServlet: doPost: WARNING -- signatures
invalid");
                returnText(response,"Error, one or more signatures were invalid!! Form
submission processing halted.", "text/plain");
                theForm.destroy();
                return;
            } else {
                System.out.println("SubmissionServlet: doPost: signature OK");
            }

            //Example: Extract the form State
            formState = theForm.getLiteralByRefEx(null,
                "global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:State]",0, null, null);
            //Store the form into the folder indicated by the formState. (as in stage 1)
            String folderPath = props.getProperty(formState);
            //get the absolute path to store the file
            ServletContext ctx = conf.getServletContext();
            String path = ctx.getRealPath(folderPath);
            //write it
            IOUtil.writeBytesToFile(path + formName, getFormBytes(theForm));

            //store extracted order datato DB2
            writeOrderDatatoDB(orderData);

            //finally store form to DB2
            writeFormToDB(orderData[0], getFormAsString(theForm));
        }
    }
}
```

```

    } catch (Exception processingE) {
        System.out.println("SubmissionServlet: doPost: Exception processing request: "
            + processingE.toString());
        returnText(response, "SubmissionServlet: doPost: Exception occurred: "
            + processingE.toString(), "text/plain");
        return;
    }

    //Return the appropriate response based on the action variable
    try {
        //Switch based on the specified action
        if (action.equalsIgnoreCase("store")) {
            returnJSP(request, response, "/success.jsp");
        } else {
            throw new Exception("SubmissionServlet: unexpected state, taking no action");
        }
    } catch (Exception anE) {
        try {
            if (theForm != null) {
                theForm.destroy();
            }
        } catch (Exception anotherE) {
            System.out.println("SubmissionServlet: Nested Exception: " +
anotherE.toString());
        }
        response.setContentType("text/html");
        PrintWriter out = new PrintWriter(response.getOutputStream());
        out.write(anE.toString());
        out.flush();
        out.close();
        System.out.println("SubmissionServlet: Exception: " + anE.toString());
    }
    System.out.println("SubmissionServlet: doPost: completed.");
}

```

As in the prepopulation discussion, here we can find the problem to address elements in data instances by name — in the code above, see the path:

```
global.global.xmlmodel[xfdl:instances][6][null:FormMetaData][null:State]
```

To overcome this problem, the method `extractInstance` could be used. For an example, see Chapter 9, “Domino integration” on page 273.

The application in Stage 2 stores the files in both the file system and the DB2 database (see method `writeFormToDB`). The corresponding code for DB2-storage is quite simple. It just calls the corresponding `DB2ConnectionForms` method. The method will check for any update or insert operations internally. The key for insert/update procedures is the first element of the `orderData` array containing the extracted data.

After this, the order state stored in DB2 is checked (`lastOrderState`). If any approvals are detected (new order state 3 or 4), the corresponding history fields are filled (Example 5-24).

Example 5-24 Code storing order data and entire form to DB2

```
private static void writeOrderDataToDB(String[] orderData) {
    DB2ConnectionForms.writeOrderData(orderData);
}

private static void writeFormToDB(String orderID, String theForm) {
    DB2ConnectionForms.writeOrderXFDL(orderID, theForm);
}
```

The way to create orderData array was straightforward here — we just parsed the FormOrderData instance using the API for registered field names in orderdata.properties file and extracted the values using getLiteralByRefEx. This should be improved in a production scenario using a dedicated order object populated with data from FormOrderInstance using XML parsing. Code for extracting FormOrderInstance as an XML fragment can be seen in the Domino integration chapter. The actually working code fragment for the WebSphere Application Server (WAS) servlet is shown in Example 5-25.

Example 5-25 Extracting order data and detecting workflow state changes

```
/**
 * Extracts the order data from the form and creates an array
 * representation. <BR>
 * The elements and their order are defined in the order.properties file.
 *
 * @param theForm
 * @param orderProps
 * @return
 * @throws UWIException
 */
private static String[] extractOrderData(FormNodeP theForm,
    Properties orderProps) throws UWIException {
    int numElements = (new Integer(orderProps.getProperty("NUM_ELEMENTS")))
        .intValue();
    String[] orderData = new String[numElements];
    String[] lastOrderData = new String[numElements];
    String orderState = "0";
    String lastOrderState = "0";
    GregorianCalendar cal = new GregorianCalendar();
    //Extract data from the form
    String propName = "ORDER_ELEMENT_NAME_" + i;
    String propValue = orderProps.getProperty(propName);
    String refString =
"global.global.xmlmodel[xfdl:instances][4][null:FormOrderData][null:"
        + propValue + "];";
    orderData[i] = theForm.getLiteralByRefEx(null, refString, 0, null,
        null);
}

// read the state, get last state and check for history fields to write
orderState = orderData[5];
//read last order metadata
System.out.println("Read last order data");
if (!orderState.equals("1"))
    lastOrderData = DB2ConnectionForms.getOrderData(orderData[0]);
System.out.println("Read last order data OK");
if (lastOrderData == null)
    lastOrderData = new String[numElements];
lastOrderState = lastOrderData[5];
if (lastOrderState==null) lastOrderState="0";
if (lastOrderState.equals("")) lastOrderState="0";
```

```

//now set approval dates and ids if the states match
if ((orderState.equals("3") || orderState.equals("4"))
    && (lastOrderState.equals("2"))) {
    System.out.println("MGR approval");
    orderData[10] = "userRole2"; //manager id
    orderData[11] = currentDate(); //app date mgr
    orderData[12] = "accepted by manager"; //comment manager
    System.out.println("MGR approval OK");
} else if (orderState.equals("4")
    && (lastOrderState.equals("3"))) {
    System.out.println("DIR approval");
    orderData[13] = "userRole3";
    orderData[14] = currentDate();
    orderData[15] = "accepted by director";
    System.out.println("DIR approval OK");
} else if (orderState.equals("4")
    && (!lastOrderState.equals("4"))) {
    System.out.println("AUTO approval");
    System.out.println("AUTO approval OK");
}
if (orderState.equals("4") //completed
    && (!lastOrderState.equals("4"))) orderData[7] = currentDate();
if (lastOrderState.equals("0")) orderData[6] = currentDate();

return orderData;
}

```

For details on the workflow state meanings, see 5.7.2, “Approval workflow” on page 205.

5.4.5 Reading form data from DB2

Having at least one form submitted and stored in DB2, we can work on the read procedure to read the form on any subsequent form invocation (such as for an approval or further processing in other applications).

The place to modify is doGet method. In Stage 1 this method does not access the form at all. Now we have to implement an additional action (showForm) with an additional parameter (orderNumber) to retrieve the form and send it to the browser.

The differences from the New Order scenario in Stage 1 are as follows:

- ▶ The New Order scenario will read the template from the file system as in Stage 1. We will read the submitted form when re-opened from DB2 now.
- ▶ The New Order scenario will do form data prepopulation — reopened forms are not changed when they are opened.

We insert some additional code into the doGet method detecting the value “showForm” in the if... else if ... chain evaluation for the action parameter processing form load for re-opened forms previously stored in DB2 (Example 5-26).

Example 5-26 Additional code to read XFDL form DB2 and send to the Viewer

```

        //insert in ... else if ... chain evaluation for the action parameter
    } else if (action.equalsIgnoreCase("showForm")) {
        System.out.println("SubmissionServlet: doGet: detected -->showForm<--
request.");
        String orderNumber = request.getParameter("orderNumber");
        if (orderNumber == null) {
            returnText(
                response,
                "SubmissionServlet: for showForm, parameter orderNumber must not be
null.",
                "text/plain");
        } else {
            System.out
                .println("SubmissionServlet: doGet: showForm: orderNumber = "
                    + orderNumber);
            //Load Form From DB
            String formString = DB2ConnectionForms.readRowXFDL(orderNumber);
            System.out.println("SubmissionServlet: doGet: showForm: loaded form from
DB2. Length = "+ formString.length());
            //Return Form to requestor
            returnForm(response, formString);
            return;
        }
    }

```

The called returnForm helper method was already used in prepopulation scenario for new forms from template. The code assumes a URL to the servlet like this:

`http://servername:port/servletPath?action=showForm&orderNumber=XXXXX`

The inserted code will retrieve the order number, read the form data using the readRowXFDL method and return the form to the client. To create suitable URLs, a new JSP was required (db2listing.jsp). It will read DB2 data for all available forms, render it in a table for browser display, and create the appropriate links to open the forms.

5.5 Adjustments to JSPs for Stage 2

In this section we explain some adjustments to the JSPs for Stage 2.

5.5.1 Where we are in the process: Building Stage 2 of the base scenario

Figure 5-7 is intended to provide an overview of where we are within the key steps involved to build Stage 2 of the base scenario. This focuses on adding storage capabilities to DB2, incorporating Web services, modifying the JSPs, and adding an approval workflow.

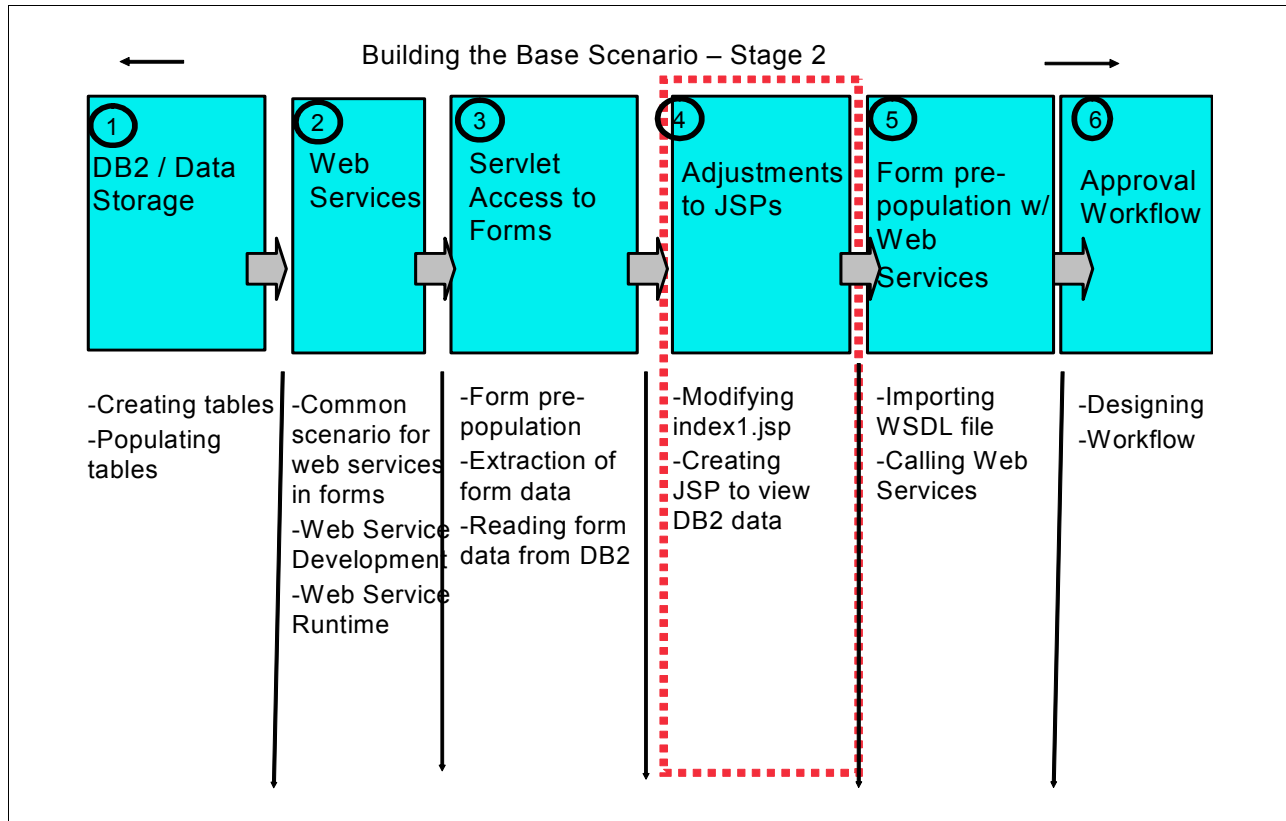


Figure 5-7 Overview of major steps involved in building Stage 2 of base scenario application

5.5.2 Modifying the index.jsp

There are two additional buttons that you will need to add to the *index.jsp* in order to connect to DB2:

1. **All Orders (DB2):** This button submits a request to the *db2listing.jsp*, which returns all the orders in the DB2 database.
2. **My Orders (DB2):** This button submits a request to the *db2listing.jsp*, which returns only the orders of the user making the request in the DB2 database.

Figure 5-8 shows what the final JSP is going to look like.

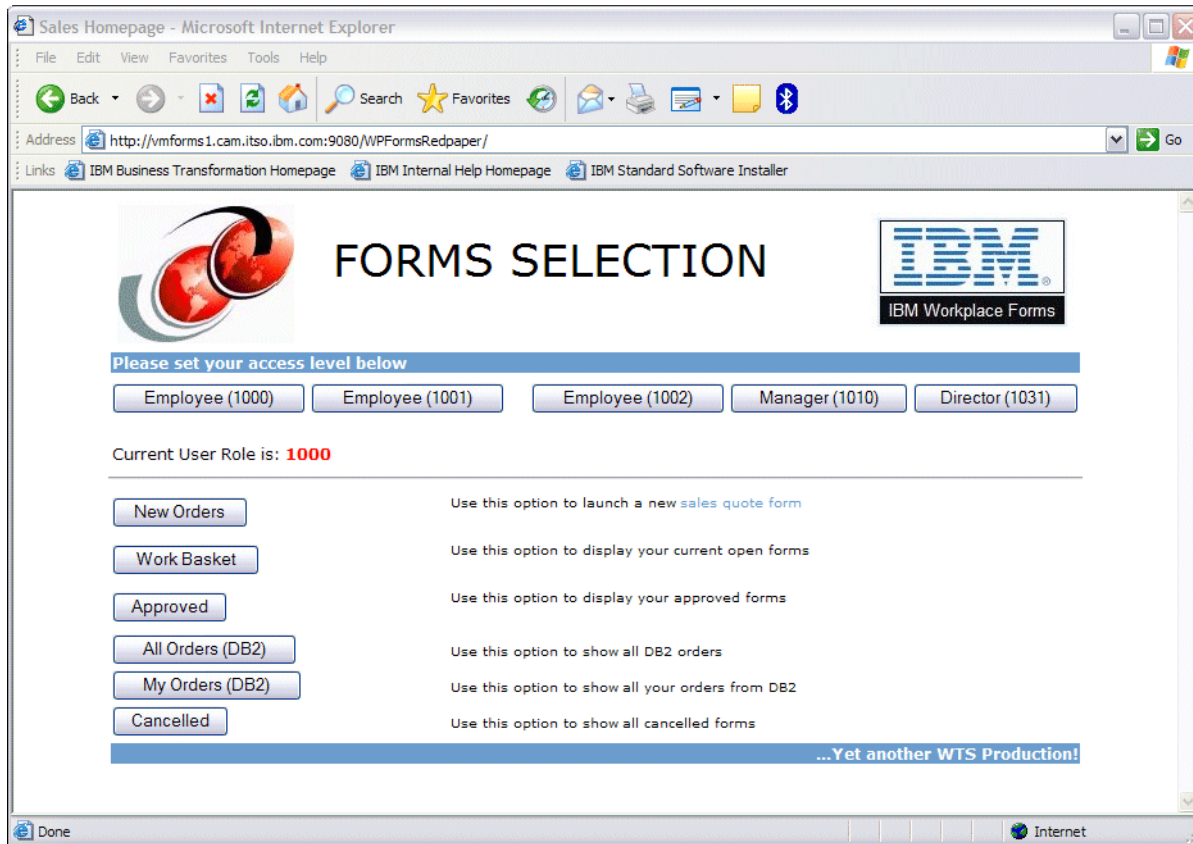


Figure 5-8 index.jsp

These are the steps to add these buttons:

1. Open *index.jsp* using your application development tool.
2. In the HTML for the **Forms Access** button table, add the code shown in Example 5-27.

Example 5-27 Code to add the DB2 buttons

```
<TR>
  <TD>
    <FORM method=get action="db2listing.jsp">
      <INPUT type="submit" value="All Orders (DB2)">
    </TD>
    <TD><FONT size="-2">Use this option to show all DB2 orders</FONT>
    </TD>
  </FORM>
</TD>
</TR>
<TR>
  <TD>
    <FORM method=get action="SubmissionServlet">
      <INPUT type="submit" value="My Orders (DB2)">
      <INPUT type="hidden" name="action" value="getJSP">
      <INPUT type="hidden" name="jsp" value="db2listing.jsp">
    </TD>
    <TD><FONT size="-2">Use this option to show all your orders from DB2</FONT>
    </TD>
  </FORM>
```

```

</TD>
</TR>

```

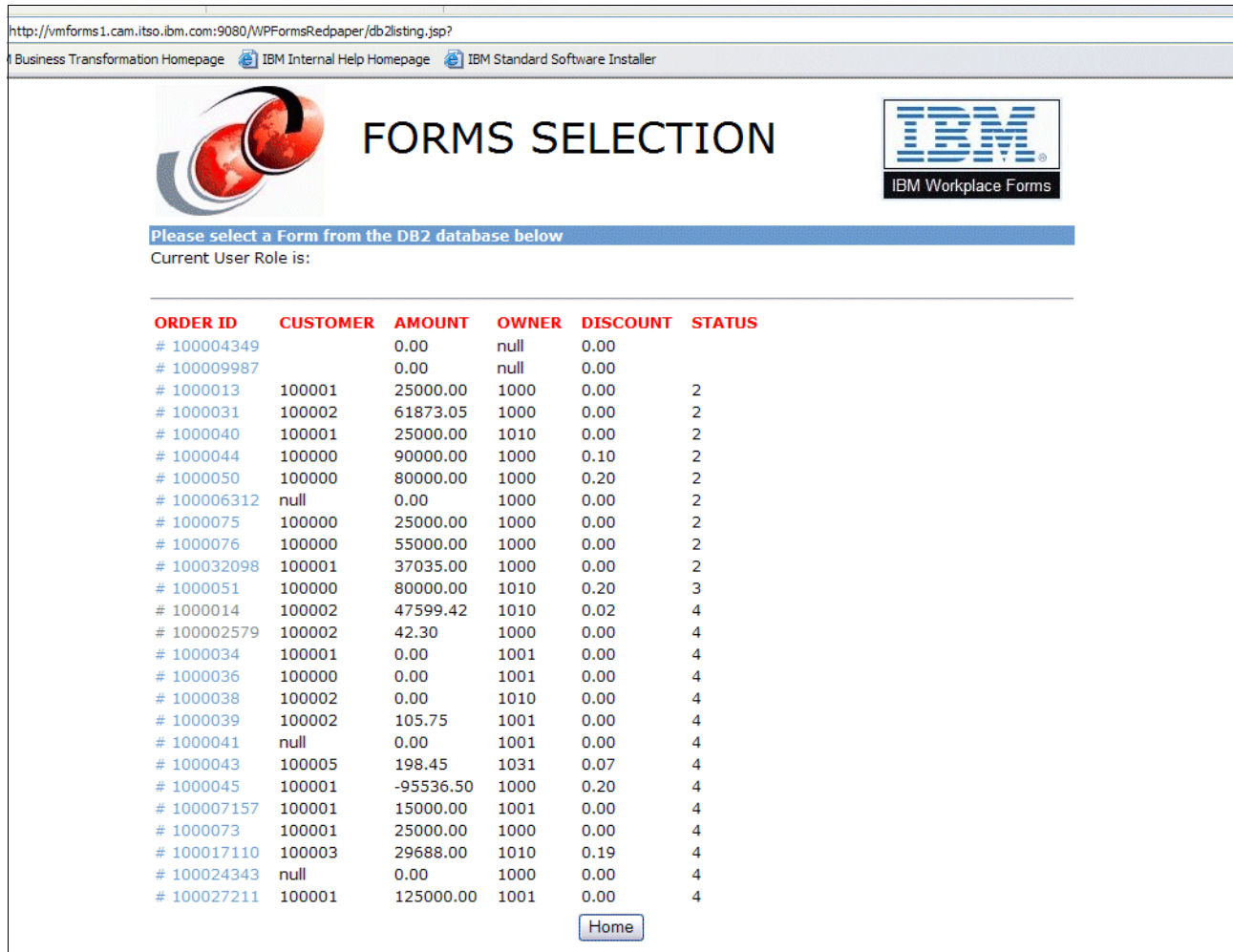
3. Save the updated file as *index.jsp* so as to separate this from the original *index.jsp*.

5.5.3 Creating a JSP to view DB2 data

In order to display the data from the DB2 database, you will need to create a JSP that reads the data from the tables and creates the output in HTML format. This JSP uses the DB2 Access Layer connector mentioned previously in 5.1.5 to make the queries to the database and to store the form back.

The *db2listing.jsp* has a similar function to the *dirlisting1.jsp* mentioned in Chapter 4. The main difference is that the forms in this example are stored in a table in DB2 and not on the file system. This JSP is used two different ways. The first scenario is from the *index.jsp* where a user has the option to select the button, **All Orders (DB2)**. This button does not pass in any userID information to the servlet and all data from the database is returned. The second time that this JSP is called is from the **My Orders (DB2)** button. In this case, the userID is passed to the JSP from the servlet, and only those forms that the user has created will be displayed.

Figure 5-9 shows what the **All Orders** view from the *db2listing.jsp* looks like.



ORDER ID	CUSTOMER	AMOUNT	OWNER	DISCOUNT	STATUS
# 100004349		0.00	null	0.00	
# 100009987		0.00	null	0.00	
# 1000013	100001	25000.00	1000	0.00	2
# 1000031	100002	61873.05	1000	0.00	2
# 1000040	100001	25000.00	1010	0.00	2
# 1000044	100000	90000.00	1000	0.10	2
# 1000050	100000	80000.00	1000	0.20	2
# 100006312	null	0.00	1000	0.00	2
# 1000075	100000	25000.00	1000	0.00	2
# 1000076	100000	55000.00	1000	0.00	2
# 100032098	100001	37035.00	1000	0.00	2
# 1000051	100000	80000.00	1010	0.20	3
# 1000014	100002	47599.42	1010	0.02	4
# 100002579	100002	42.30	1000	0.00	4
# 1000034	100001	0.00	1001	0.00	4
# 1000036	100000	0.00	1001	0.00	4
# 1000038	100002	0.00	1010	0.00	4
# 1000039	100002	105.75	1001	0.00	4
# 1000041	null	0.00	1001	0.00	4
# 1000043	100005	198.45	1031	0.07	4
# 1000045	100001	-95536.50	1000	0.20	4
# 100007157	100001	15000.00	1001	0.00	4
# 1000073	100001	25000.00	1000	0.00	4
# 100017110	100003	29688.00	1010	0.19	4
# 100024343	null	0.00	1000	0.00	4
# 100027211	100001	125000.00	1001	0.00	4

Figure 5-9 All Orders from DB2 using *db2listing.jsp*

Note: The values in the STATUS column on the right-hand side of the screen refer to the workflow stage, which we will talk about later in 5.7.2, “Approval workflow” on page 205

Figure 5-10 shows what the *My Orders* view from the *db2listing.jsp* looks like.

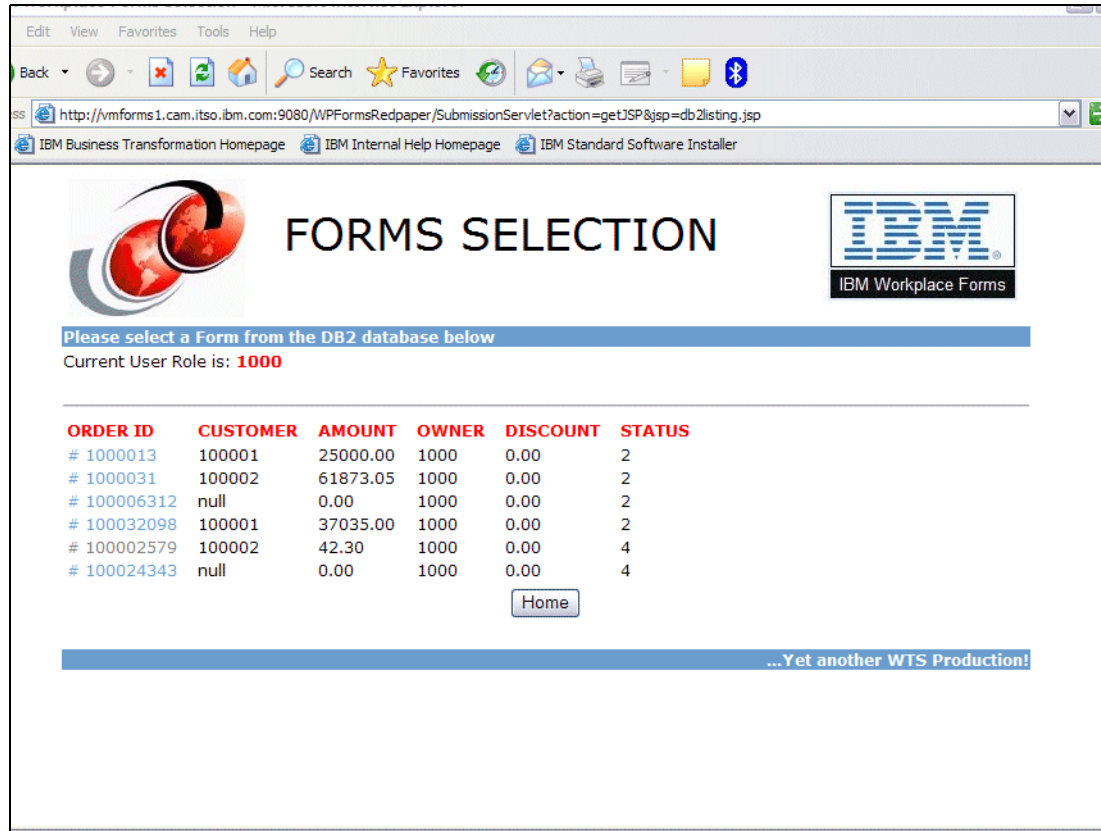


Figure 5-10 *My Orders* from DB2 view using *db2listing.jsp*

1. Create a new JSP named *db2listing.jsp* and save it to the *WebContent* folder on the Web server.
2. Copy the code below into the JSP. Example 5-28 shows the code to get the tables from DB2 and then create an HTML table with the metadata values and a link to the forms that are stored in the database.
3. Save and close the new JSP.

Example 5-28 Code to create db2listing.jsp

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta http-equiv="Content-Style-Type" content="text/css">
<link rel="stylesheet" href="theme/blue.css" type="text/css">
<title>IBM Workplace Forms Selection</title>
</head>

<CENTER>
<TABLE border="0" cellpadding="2" width="760" >
  <TBODY>
```

```
<TR>  
    <TD><IMG border="0" src="theme/redbook_logo.jpg" width="138"  
        height="114" align="left"></TD>  
    <TD align="left"><H1>FORMS SELECTION</H1></TD>  
    <TD><IMG border="0" src="theme/Workplace_Forms.jpg" width="158"  
        height="92" align="right"></TD>  
</TR>  
</TBODY>  
</TABLE>  
</CENTER>  
  
<%@ page session="false" contentType="text/html"  
import="java.util.*, java.io.File"%>  
  
<!-- import DB2 connection library to make order query available -->  
<%@ page import="forms.cam.itso.ibm.com.DB2ConnectionForms"%>  
  
<TABLE border="0" width="760" align="center">  
    <TR>  
        <TD bgcolor="#699ccf"><B>Please select a Form from the DB2 database below</B><BR>  
    </TD>  
    </TR>  
    <TR>  
        <TD>  
            <%  
String userID = (String) request.getAttribute("userRole");  
  
if (userID == null) userID = "";  
String [][] resArr = DB2ConnectionForms.getOrderListEmp(userID);  
  
String html = "Current User Role is: <STRONG>" + userID + "</STRONG><P><HR>";  
  
html = html + "<TABLE cell padding=5>";  
String n = "<TD>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~</TD>";  
String projectName = "WPFormsRedpaper";  
  
html = html + "<TD><STRONG>ORDER  
ID</STRONG></TD>"+n+"<TD><STRONG>CUSTOMER</STRONG></TD>"+n+"<TD><STRONG>AMOUNT</STRONG></TD>  
>"+ n;  
html = html +  
"<TD><STRONG>OWNER</STRONG></TD>"+n+"<TD><STRONG>DISCOUNT</STRONG></TD>"+n+"<TD><STRONG>STA  
TUS</STRONG></TD>";  
for (int i = 0; i < resArr.length; i++){  
html = html + "<TR>";  
html = html + "<TD>";  
html = html + "<A href="/" + projectName +  
"/SubmissionServlet?action=showForm&amp;orderNumber=" + resArr[i][0]+ ">#  
"+resArr[i][0]+"</A>";  
html = html + "</TD>"+n;  
html = html + "<TD>" + resArr[i][1] + "</TD>"+n;  
html = html + "<TD>" + resArr[i][2] + "</TD>"+n;  
html = html + "<TD>" + resArr[i][4] + "</TD>"+n;  
html = html + "<TD>" + resArr[i][3] + "</TD>"+n;  
html = html + "<TD>" + resArr[i][5] + "</TD>";  
html = html + "</TR>";  
}  
html = html + "</TABLE>";  
%>  
<%=html %>
```

```

        </TD>
    </TR>
<TR>

    <TD align="center">
        <FORM method=get action="SubmissionServlet">
        <INPUT type="submit" value="Home">
        <INPUT type="hidden" name=action value="getJSP">
        <INPUT type="hidden" name=jsp value="index.jsp">
        </FORM>

</TR>
</TABLE>

<TABLE width="760" align="center">
<tr bgcolor="#699ccf">
    <td align="right"><B>...Yet another WTS
        Production!</B></td>
</tr>
</TABLE>

```

Created links

Table 5-7 and Table 5-8 illustrate the links generated within dirlisting.jsp and db2listing.jsp.

Table 5-7 Examples for the generated actions in dirlisting.jsp (Stage 2)

	Generated URL:
RAD6 test environment:	
New form	http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/SubmissionServlet?action=prepop&template=C:\WEBSPH~1\APPSER~1\installdApps\vmforms1\WPFormsRedpaper_war.ear\WPFormsRedpaper.war\Redpaper_Demo\Form_Templates\Redpaper_Forms_Sample_S2_v41.xfdl
Open stored form	not available (processed by db2listing.jsp)
Deployed application:	
New form	http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/SubmissionServlet?action=prepop&template=C:\WEBSPH~1\APPSER~1\installdApps\vmforms1\WPFormsRedpaper_war.ear\WPFormsRedpaper.war\Redpaper_Demo\Form_Templates\Redpaper_Forms_Sample_S2_v41.xfdl
Open stored form	not available (processed by db2listing.jsp)

Table 5-8 Examples for the generated actions in db2listing.jsp (Stage 2)

	Generated URL:
RAD6 test environment:	
New form	Not available (processed by dirlisting.jsp)
Open stored form	http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/SubmissionServlet?action=showForm&orderNumber=1000044
Deployed application:	
new form	Not available (processed by dirlisting.jsp)
open stored form	http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/SubmissionServlet?action=showForm&orderNumber=100009987

5.6 Form prepopulation using Web services

In this section we discuss the use of Web services for form prepopulation.

5.6.1 Where we are in the process: Building Stage 2 of the base scenario

The diagram in Figure 5-11 is intended to provide an overview of where we are within the key steps involved to build Stage 2 of the base scenario. This focuses on adding storage capabilities to DB2, incorporating Web services, modifying the JSPs and adding an approval workflow.

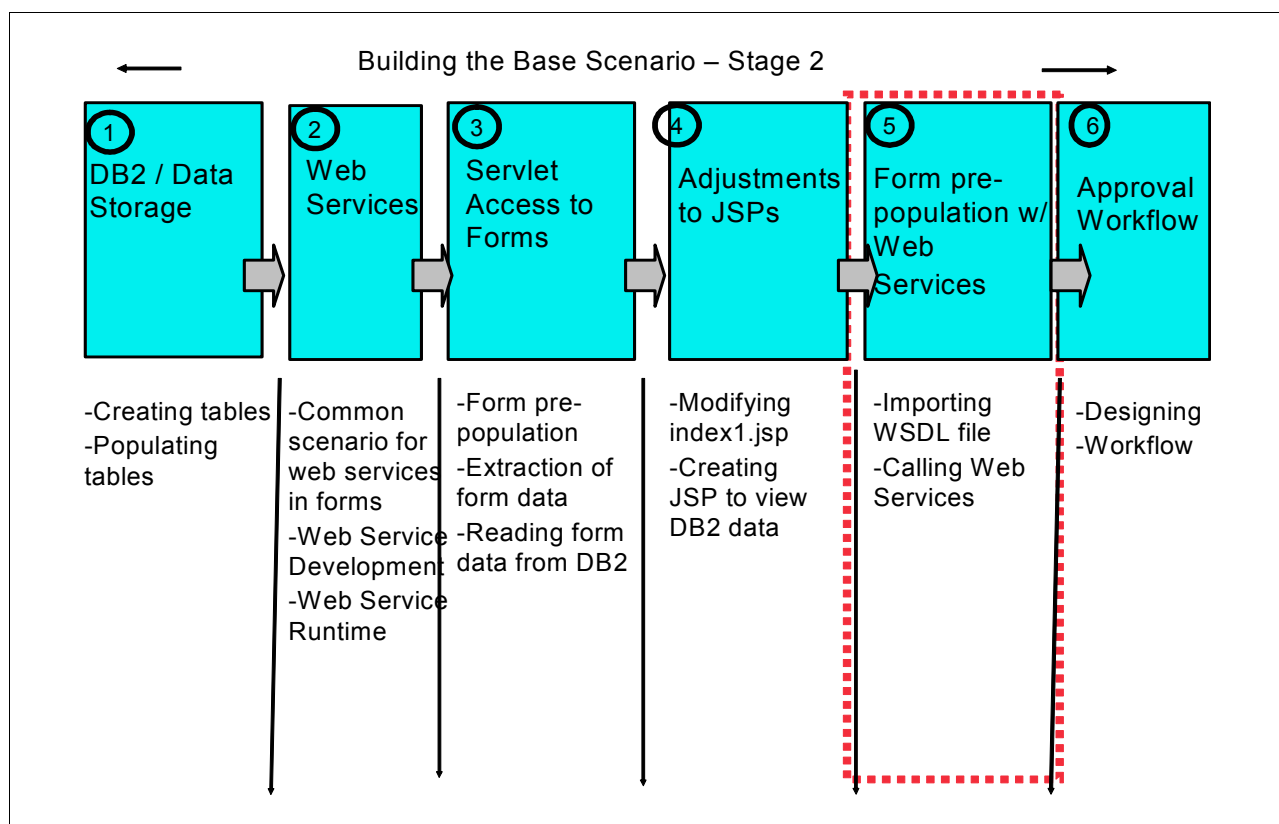


Figure 5-11 Overview of major steps involved in building Stage 2 of base scenario application

Web services are useful for real time form population. Instead of the form being submitted to the server, the server filling in the form data, then returning the form, a Web services Description Language (WSDL) document can populate the form in real time. Web services can be called from within the XFDL language. Through embedding a WSDL document within a form, the API automatically creates XFDL function calls that represent the valid ports within the WSDL document.

5.6.2 Importing the WSDL file

When you embed a WSDL document within a form, the functions defined in the WSDL document become available to the form as though they were XFDL functions.

To enclose WSDL files, the Designer provides a menu option under the Tools menu. An input field is provided for the file. When adding a new WSDL file this column shows the file name including the path. The WSDL Name column shows the WSDL file name only (no path) for the already enclosed WSDL files. If the form is edited by hand and the WSDL name is not present this column should show the label “Name is not defined”. The Target Namespace column shows the targetNamespace attribute of the WSDL document contained in the <definitions> tag. If not present this column should show the label “<targetNamespace> is not defined” and if the <definitions> tag is not present show the label “<definitions> tag is not defined”.

To import a WSDL in our sample form application, follow these steps:

1. From the *Tools* menu, select **Enclose WSDL...**
 - The *Enclose WSDL* Dialog box opens as shown in Figure 5-12.
2. Under *Add WSDL File*, type the path to the WSDL file or browse to locate it.
3. Click **Add**.
4. Repeat steps 2-3 to add additional WSDL documents.
5. Click **OK**.

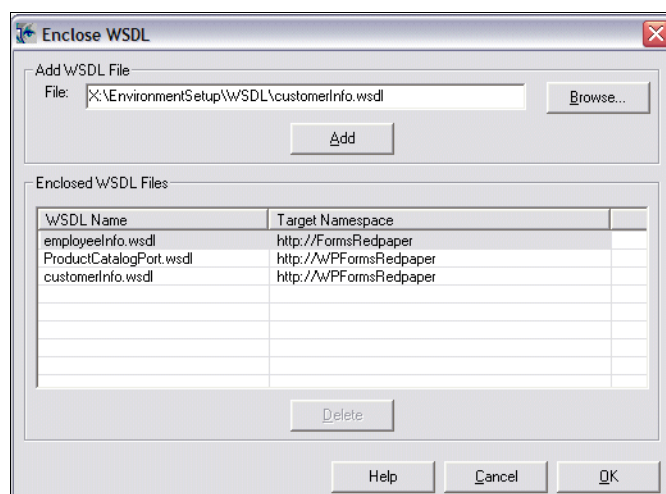


Figure 5-12 Enclose WSDL dialog box

A new top-level option will be added to the form's global item (within the globalpage) called <webservices>. Only one <webservices> element may exist within the <global> element. Within the <webservices> element only <wsdl> elements may exist, which must contain a name attribute. The name attribute is the name of the Web service. The <wsdl> element contains the actual WSDL data. The <wsdl> tag has an attribute called “name” that contains the original file name of the WSDL. When the Designer writes the “name attribute” into the form the path is striped and only stores the file name (see Example 5-29).

```

<webservices>
  <wsdl name="customerInfo.wsdl">
    <wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://WPFormsRedpaper"
      xmlns:intf="http://WPFormsRedpaper" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://WPFormsRedpaper">
      <wsdl:types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://WPFormsRedpaper">
          <import namespace="http://schemas.xmlsoap.org/soap/encoding/"></import>
          <complexType name="CUSTOMER">
            <sequence>
              <element name="CUST_ID" type="xsd:string"></element>
              <element name="CUST_NAME" type="xsd:string"></element>
              <element name="CUST_AMGR" type="xsd:string"></element>
              <element name="CUST_CONTACT_NAME" type="xsd:string"></element>
              <element name="CUST_CONTACT_POSITION" type="xsd:string"></element>
              <element name="CUST_CONTACT_PHONE" type="xsd:string"></element>
              <element name="CUST_CONTACT_EMAIL" type="xsd:string"></element>
              <element name="CUST_CRM_NO" type="xsd:string"></element>
            </sequence>
          </complexType>
        </schema>
      </wsdl:types>
      <wsdl:message name="GETCUSTINFORequest">
        <wsdl:part name="CUST_ID" type="xsd:string"></wsdl:part>
      </wsdl:message>
      <wsdl:message name="GETCUSTINFOResponse">
        <wsdl:part name="GETCUSTINFOReturn" type="impl:CUSTOMER"></wsdl:part>
      </wsdl:message>
      <wsdl:message name="GETCUSTOMERLISTResponse">
        <wsdl:part name="GETCUSTOMERLISTReturn" type="xsd:string"></wsdl:part>
      </wsdl:message>
      <wsdl:message name="GETCUSTOMERLISTRequest">
        <wsdl:part name="FILTER" type="xsd:string"></wsdl:part>
      </wsdl:message>
      <wsdl:portType name="CustomerInfo">
        <wsdl:operation name="GETCUSTOMERLIST" parameterOrder="FILTER">
          <wsdl:input message="impl:GETCUSTOMERLISTRequest" name="GETCUSTOMERLISTRequest">
            </wsdl:input>
          <wsdl:output message="impl:GETCUSTOMERLISTResponse" name="GETCUSTOMERLISTResponse">
            </wsdl:output>
          </wsdl:operation>
        <wsdl:operation name="GETCUSTINFO" parameterOrder="CUST_ID">
          <wsdl:input message="impl:GETCUSTINFORequest" name="GETCUSTINFORequest">
            </wsdl:input>
          <wsdl:output message="impl:GETCUSTINFOResponse" name="GETCUSTINFOResponse">
            </wsdl:output>
          </wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="WPFormsCustSoapBinding" type="impl:CustomerInfo">
          <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http">
            </wsdlsoap:binding>
          <wsdl:operation name="GETCUSTOMERLIST">
            <wsdlsoap:operation soapAction=""></wsdlsoap:operation>
            <wsdl:input name="GETCUSTOMERLISTRequest">
              <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://WPFormsRedpaper" use="encoded"></wsdlsoap:body>
            </wsdl:input>
          </wsdl:operation>
        </wsdl:binding>
      </wsdl:definitions>
    </wsdl>
  </webservices>

```

```

        <wsdl:output name="GETCUSTOMERLISTResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://WPFormsRedpaper" use="encoded"></wsdlsoap:body>
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GETCUSTINFO">
        <wsdlsoap:operation soapAction=""></wsdlsoap:operation>
        <wsdl:input name="GETCUSTINFORequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://WPFormsRedpaper" use="encoded"></wsdlsoap:body>
        </wsdl:input>
        <wsdl:output name="GETCUSTINFOResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://WPFormsRedpaper" use="encoded"></wsdlsoap:body>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
<wsdl:service name="CustomerInfoService">
    <wsdl:port binding="impl:WPFormsCustSoapBinding" name="WPFormsCust">
        <wsdlsoap:address
location="http://vmforms1.cam.itso.ibm.com:8085/WpfWsCustomerT/services/WPFormsCust"></wsdlsoap:address>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
</wsdl>
</webservices>

```

5.6.3 Calling the Web services

There are two types of Web services that we are using to prepopulate the form.

- ▶ Web services returning simple type objects
- ▶ Web services returning complex type objects

While the Integration of Web services returning a simple type object can be accomplished using custom computes on the field level, Web services returning complex objects have to be integrated into your XML Data Model.

Web services returning simple type objects

Web services returning simple type objects send back a single string as return value. This value can then either be filled as a single value into a field or split up into several values as choices for a pop-up menu for example. In our scenario we use this type of Web services to prepopulate two pop-up menus:

- ▶ Customer selection pop-up
- ▶ Product item selection pop-up

To parse the single string returned from the Web service into a choices list for our pop-up menus we had to realize a nested loop functionality as a custom option in XFDL as described in the following section.

For loops and nesting loops in XFDL

The *For* function creates a counter that you can use to simulate a “for loop” (as found in most programming languages).

This function uses an option in your form (such as a custom option) as an index that stores the current count of the loop. As the loop counts, it sets this option to the current count. For example, if the for loop counted from 1 to 3, it would first set the option to 1, then to 2, then to 3.

You can then trigger other action in the form based on the value of this option. For example, you could use the toggle function to detect each change in the option's value, and to update some aspect of the form each time.

A form with a for loop will begin counting that loop as soon as the form opens, unless the loop itself relies on a triggering event, such as a key-press event or a toggle function.

You can create a loop that counts once by setting the initial count and the final count to be equal. For example, a loop that counts from 1 to 1 will count once.

To create a nested loop, you must trigger the second loop off the value of the first. For example, your first loop might change the value of the custom:loop1 option. You can then create a toggle function that detects any change in the custom:loop1 option, and that triggers its own loop, as shown in Example 5-30.

Example 5-30 Nesting Loops in XFDL

```
toggle(custom:loop1) == '1' ? for('custom:loop2', '1', '5') : ''
```

The second loop is triggered each time the value of custom:loop1 changes. In other words, the second loop is triggered each time the first loop counts once. Additionally, the second loop will process completely before processing of the first loop resumes. In this case, that means the second loop will count from 1-5 before the first loop counts again.

Tip: When creating the option that acts as an index for the loop, ensure that the starting value of the option does not equal the starting value of the loop. If it does, your loop will work incorrectly, since setting the first count will not result in a detectable change.

For example, you might create a loop that counts from 1-5, and your index might be set to a value of 1. In this case, the first count of the loop will set the index to 1, but since the index already equals 1, there will be no detectable change to the value. This means that any toggle function used to detect a change in that value will not fire.

Example 5-31 shows the custom computes used in the customer selection pop-up to prepopulate the choices by consuming the Customer Data Web service described in 5.3, “Web services” on page 168.

Example 5-31 Nested Loop to prepopulate Customer pop-up using a Web service

```
<custom:getCustomerChoices xfdl:compute="
  (toggle(custom:CustomerChoicesCounter) == '1' and (custom:CustomerChoicesCounter == '0'))
  or (CustomerChoices_1.value=='')
  ? set('custom:CustomerChoicesList', CustomerInfoService_WPFormsCust.GETCUSTOMERLIST('') + '~')
  + for('custom:CustomerChoicesCounter', '1', strlen(custom:CustomerChoicesList))
  : ''">
</custom:getCustomerChoices>

<custom:getCustomerChoices2 xfdl:compute="
```

```

toggle(custom:CustomerChoicesCounter) == '1' and (custom:CustomerChoicesCounter > '0')
  ? (custom:CustomerChoicesCounter > '1'
    ? destroy('CustomerChoices_' + custom:CustomerChoicesCounter, 'item')
    : '' )
  + (strlen(custom:CustomerChoicesList) > '1'
    ? (custom:CustomerChoicesCounter > '1'
      ? duplicate('CustomerChoices_1','item', 'CustomerChoices_'
        + (custom:CustomerChoicesCounter - '1'), 'item','after_sibling', 'CustomerChoices_'
        + custom:CustomerChoicesCounter)
      : '' )
    + set('CustomerChoices_' + custom:CustomerChoicesCounter + '.value',
      substr(custom:CustomerChoicesList, '0', strlen(custom:CustomerChoicesList, '~')-1))
    + set('custom:CustomerChoicesList', substr(custom:CustomerChoicesList
      ,strlen(custom:CustomerChoicesList, '~') + '1',strlen(custom:CustomerChoicesList)))
    : '' )
: '' ">
</custom:getCustomerChoices2>

```

Web services returning complex type objects

Web services returning complex type objects send back an array of strings or an XML fragment that can be populated into an XML data instance. This way you can easily use a Web service to prepopulate an entire set of fields by updating the values in the data instance that the fields are bound to. In our example we used complex type Web services to prepopulate following data instances:

- ▶ Customer data based on the customer selected
- ▶ Product data based on the item selected in the order

When writing the data returned from the Web service directly to the data instance, we do not need to parse the return value into an array or different fields. The Web services client built into the Workplace Forms Viewer asks for the customer ID as primary key and 3 additional parameters when identifying a Web service returning complex type objects in the WSDL file. The following parameters are required for the GetCustomerInfo Web service in our example:

▶ CUST_ID

This is the customer ID as the primary key to identify the customer on which we want the additional information.

Our value for this parameter: **PAGE4.Company.value->value**

▶ CustomerInfo_Return

This is a pointer to the data instance where the Web service return stream is to be integrated.

Our value for this parameter: **getInstanceRef('FormCustomerData')**

▶ CustomerInfo_ReturnScheme

This is a pointer to the location of the XML model within the XFDL file.

Our value for this parameter: **[0][0]**

▶ CustomerInfo_ReturnReplacei

This is an indicator if you want to replace or update the existing data in the instance.

Our value for this parameter: **replace**

The last three parameters define the position within the data instance and update mode when inserting the response data into the data instance. Figure 5-13 shows the input parameters required by the Web service when using the function call wizard in the Workplace Forms Designer.

Figure 5-13 Input Parameters for GetCustomerInfo Web service

We are calling this Web service out of the customer selection pop-up whenever the user selects a customer. The remaining fields to be filled by this Web service call are bound to the FormCustomerData data instance that is being filled by the call. Thereby the data filled into the data instance by the Web service is surfaced to the respective fields holding the meta information. Example 5-32 shows the Web service call using a toggle function in the customer selection pop-up.

Example 5-32 Toggle function in the customer selection pop-up to call GetCustomerInfo Web service

```
toggle(PAGE4.Company.value->value) == '1'
? set('PAGE4.FIELD2_HIDDEN.value', PAGE4.Company.value->value)
+ CustomerInfoService_WPFormsCust.GETCUSTINFO(PAGE4.Company.value->value,
getInstanceRef('FormCustomerData')+. '[0][0]', 'XFDL','replace')
+ xmlmodelUpdate()
: ''
```

In Example 5-33 we show the corresponding data instance that we build in the form before being filled by the Web service.

Example 5-33 FormCustomerData XML instance to be filled by our Web service

```
<xforms:instance xmlns="" id="FormCustomerData">
  <GETCUSTINFOResponse>
    <GETCUSTINFOReturn>
      <CUST_ID></CUST_ID>
      <CUST_NAME></CUST_NAME>
      <CUST_AMGR></CUST_AMGR>
      <CUST_CONTACT_NAME></CUST_CONTACT_NAME>
      <CUST_CONTACT_POSITION></CUST_CONTACT_POSITION>
      <CUST_CONTACT_PHONE></CUST_CONTACT_PHONE>
```

```

        <CUST_CONTACT_EMAIL></CUST_CONTACT_EMAIL>
        <CUST_CRM_NO></CUST_CRM_NO>
    </GETCUSTINFOReturn>
</GETCUSTINFOResponse>
</xforms:instance>

```

In Example 5-34 we show that data instance after it was filled by the Web service.

Example 5-34 FormCustomerData XML instance after being filled by our Web service

```

<xforms:instance xmlns="" id="FormCustomerData">
  <GETCUSTINFOResponse>
    <multiRef xmlns:ns2="http://WPFormsRedpaper"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="id0" soapenc:root="0"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xsi:type="ns2:CUSTOMER">
      <CUST_ID xsi:type="xsd:string">100000</CUST_ID>
      <CUST_NAME xsi:type="xsd:string">OnDemand Corporation</CUST_NAME>
      <CUST_AMGR xsi:type="xsd:string">1000</CUST_AMGR>
      <CUST_CONTACT_NAME xsi:type="xsd:string">Jerry Haas</CUST_CONTACT_NAME>
      <CUST_CONTACT_POSITION xsi:type="xsd:string">AccountMgr</CUST_CONTACT_POSITION>
      <CUST_CONTACT_PHONE xsi:type="xsd:string">+49 89 123-456-78</CUST_CONTACT_PHONE>
      <CUST_CONTACT_EMAIL
        xsi:type="xsd:string">Jerry.Haas@OnDemand.oom</CUST_CONTACT_EMAIL>
      <CUST_CRM_NO xsi:type="xsd:string">200001</CUST_CRM_NO>
    </multiRef>
  </GETCUSTINFOResponse>
</xforms:instance>

```

The result of this Web services integration is a Real-Time access and pre-filling to customer data from the DB/2 database back-end.

Tip: When you create the bindings for the customer data fields to the corresponding data instance you should reference the second root node of by position and not by name. This is due to the fact that the root node <GetCustomerInfoReturn> changed to <multiRef> by the complex type data returned by the Web service. When integrating the Domino hosted complex type Web service this is not the case. A sample binding with a reference by position from the FormCustomerData instance to a field is shown in example 4-24.

Example 5-35 shows a sample binding of a field to the FormCustomerData instance. The data populated to the instance elements is bound to the field in the form that way.

Example 5-35 Data Instance Binding for the Account Number Field

```

<bindings>
  <bind>
    <instanceid>FormCustomerData</instanceid>
    <ref>[null:GETCUSTINFOResponse][0][null:CUST_ID]</ref>
    <boundoption>PAGE4.AccountNumber.value</boundoption>
  </bind>
</bindings>

```

We will use a similar Web service to prepopulate our form with a Lotus Domino hosted Web service in Chapter 9, “Domino integration” on page 273.

5.7 Workflow

In this section we consider the workflow in our sample application.

5.7.1 Where we are in the process: Building Stage 2 of the base scenario

The diagram in Figure 5-14 is intended to provide an overview of where we are within the key steps involved to build Stage 2 of the base scenario. This focuses on adding storage capabilities to DB2, incorporating Web services, modifying the JSPs, and adding an approval workflow.

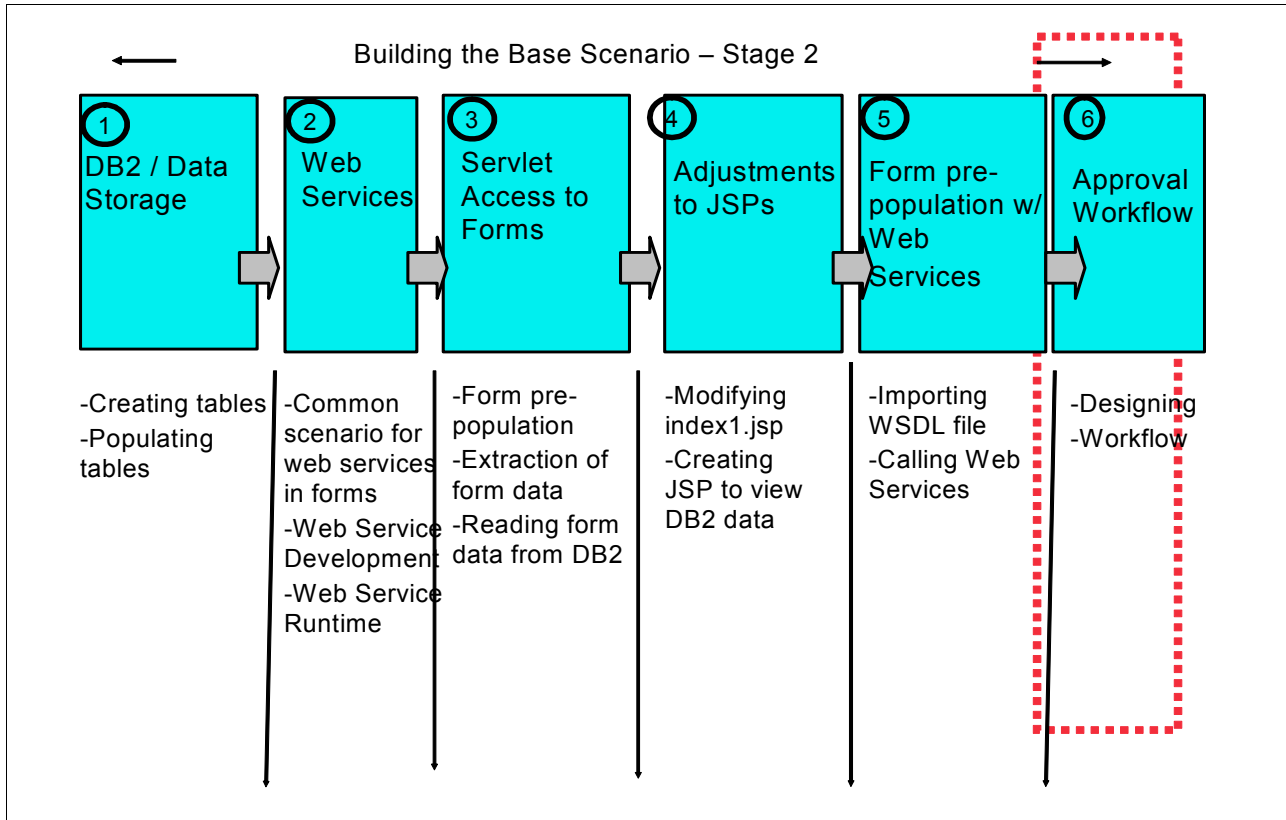


Figure 5-14 Overview of major steps involved in building Stage 2 of base scenario application

5.7.2 Approval workflow

To bring our sample application closer to a real life scenario, we designed an approval workflow that assigns different approval levels to the sales quote according to thresholds or business rules that we prepopulate into the form.

Figure 5-15 illustrates an overview of the workflow and business logic contained within the application.

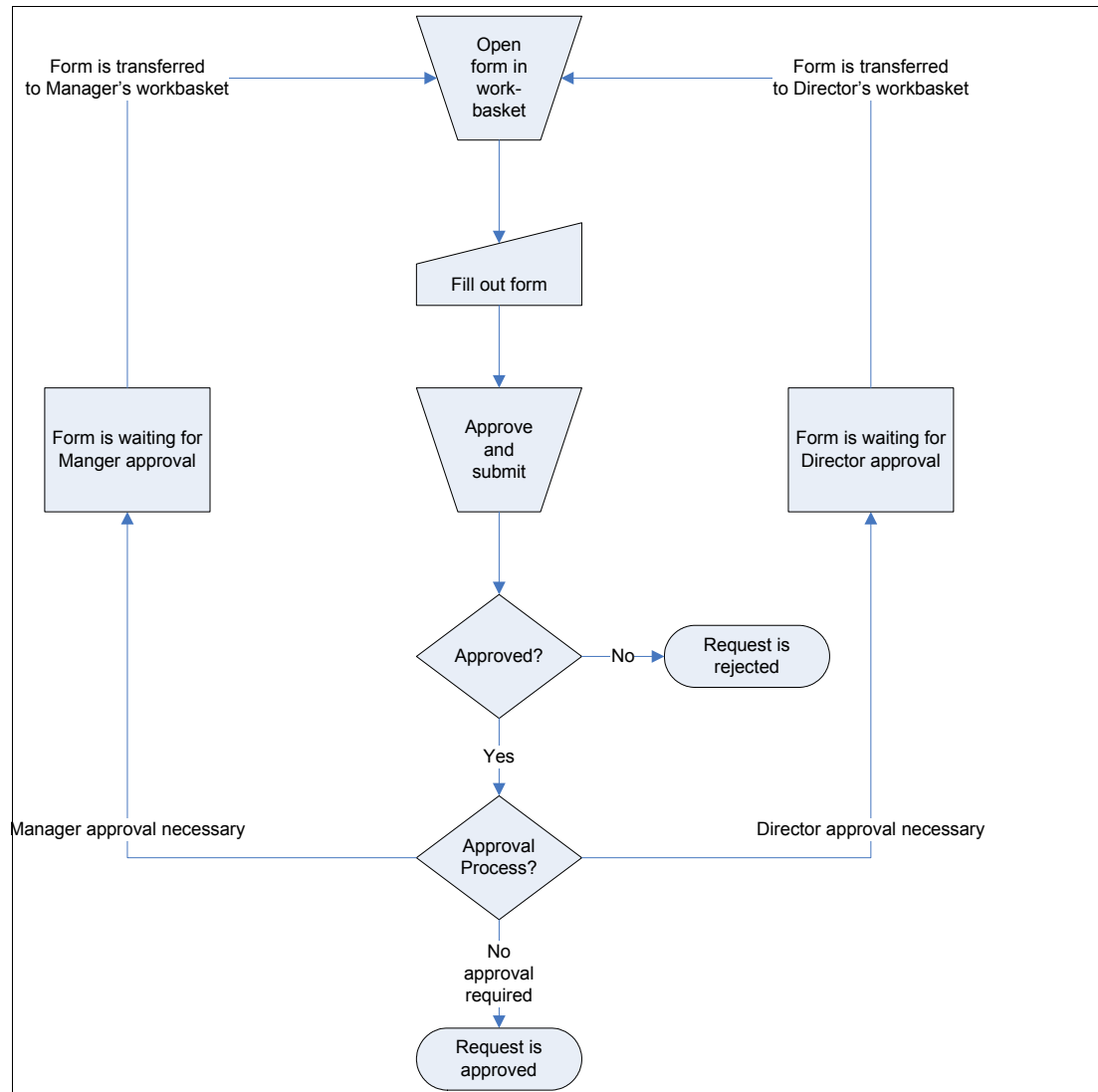


Figure 5-15 Overview of workflow for sample application

The workflow is comprised of the following stages:

1. Template
2. Waiting for Manager Approval
3. Waiting for Director Approval
4. Approved
5. Rejected
6. Terminated

The different approval levels are assigned to three roles of users accessing the form:

1. Requestor
2. Manager Level Approver
3. Director Level Approver

Figure 5-16 shows the workflow stages and the basic flow.

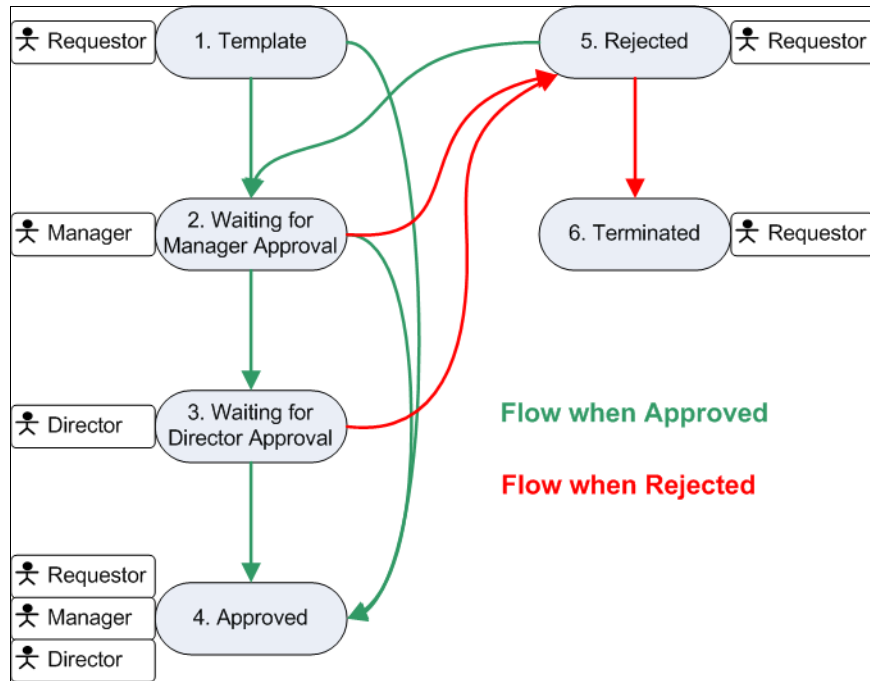


Figure 5-16 Workflow stages, access roles and basic flow

On the form we are doing the workflow processing when a user signs the form. The signature buttons hold different custom computes that sets two fields of the form (State and PreviousState) to the appropriate state depending on the business rules. The business rules are also prepopulated by the servlet and have following rule set.

- ▶ If the total amount of the sales quote is less than \$10,000, the form is approved when the requestor signs it
- ▶ If the total amount is between \$10,000 and \$50,000, a Manager Level Approval is necessary to approve the quote.
- ▶ If the total amount is greater than \$50,000, a sequential signing with Manager and Director Level Approval is necessary to approve the quote.

Example 5-36 shows the custom compute on the requestor's signature button to process the form within the workflow. The computes on the other two signature buttons work in the same fashion to control the workflow in the higher stages.

When the form is submitted, the servlet reads the values for State and PreviousState from the data instance to store the form in the corresponding folder or workbasket of the respective role that has to take action. Thereby the control over the workflow remains in the business logic within the form.

Example 5-36 Custom compute on the requestor's signature button to process the form in the workflow

```
((toggle(signer) == '1') and (signer != ''))
  ? set('siglDate_LABEL.value', 'Signed: '
    +. formatString(time(), 'PAGE4.global.custom:time_format') +. ' '
    +. formatString(date(), 'PAGE4.global.custom:date_format'))
    + (global.global.xmlmodel[xfdl:instances][3][null:BusinessRuleParams][null:QuoteLevelOneThreshold] >
OrderTotal.value)
    ? set('PAGE4.PreviousState.value', PAGE4.State.value)
      + set('PAGE4.State.value', '4')
      + viewer.messageBox('Because the quote is less than $10,000, a Manager is not required to sign.',
'Director Signature NOT Required', 'OK')
        : set('PAGE4.PreviousState.value', PAGE4.State.value)
          + set('PAGE4.State.value', '2')
          + viewer.messageBox('Because the quote is $'
+ global.global.xmlmodel[xfdl:instances][3][null:BusinessRuleParams][null:QuoteLevelOneThreshold]
+ ' or greater, a Manager must sign for this quote to proceed.', 'Manager Signature Required', 'OK')
    : ''
```



Integrating with Portal

This chapter describes the process to deploy the Sales Quotation application to WebSphere Portal. Using the base sample scenario application described in Chapter 4, “Building the base scenario: Stage 1” on page 53, and Chapter 5, “Building the base scenario: Stage 2” on page 145, we will show how to integrate this same sample application with WebSphere Portal.

In this chapter, we discuss these topics:

- ▶ Overview of Portal integration
- ▶ Writing a portlet that displays a form
- ▶ Inter-portlet communication
- ▶ Parking the Workplace Forms Viewer in the Portal
- ▶ Deploying the portlets

Note: The code used for building this sample scenario application is available for download. For specific information about how to download the sample code, please refer to Appendix A, “Additional material” on page 333.

6.1 Goal of integrating the application with WebSphere Portal

Up until this point in the redbook, we have developed and used a J2EE application that processed the form (as described in Chapter 4, “Building the base scenario: Stage 1” on page 53, and Chapter 5, “Building the base scenario: Stage 2” on page 145.) While this method has its advantages, it still requires Java development skills and time to create the servlets and JSPs to handle the processing and viewing of the forms.

WebSphere Portal provides a standards-based, composite application framework that allows Workplace Forms to work with other applications through Portal’s extended presentation layer. For example, a Portal Administrator can easily add a form viewer portlet to a page and place it among other portlets. This allows a user to have a contextual view of their form, and interact with the other portlets on the page to supplement the information that they have in front of them.

Figure 6-1 illustrates how the portlets fit together on a Portal page. There are a number of elements working together that make up the complete page on the portal. Each portlet window is pointing to an independent application; for example, you can use the People Finder portlet to search for users without needing to refresh any of the other portlets. You can incorporate interportlet communication to allow two portlets to send and receive data to each other. The ClicktoAction Sender portlet and the WPFRedpaper portlet use interportlet communication to display the correct form that a user selects.

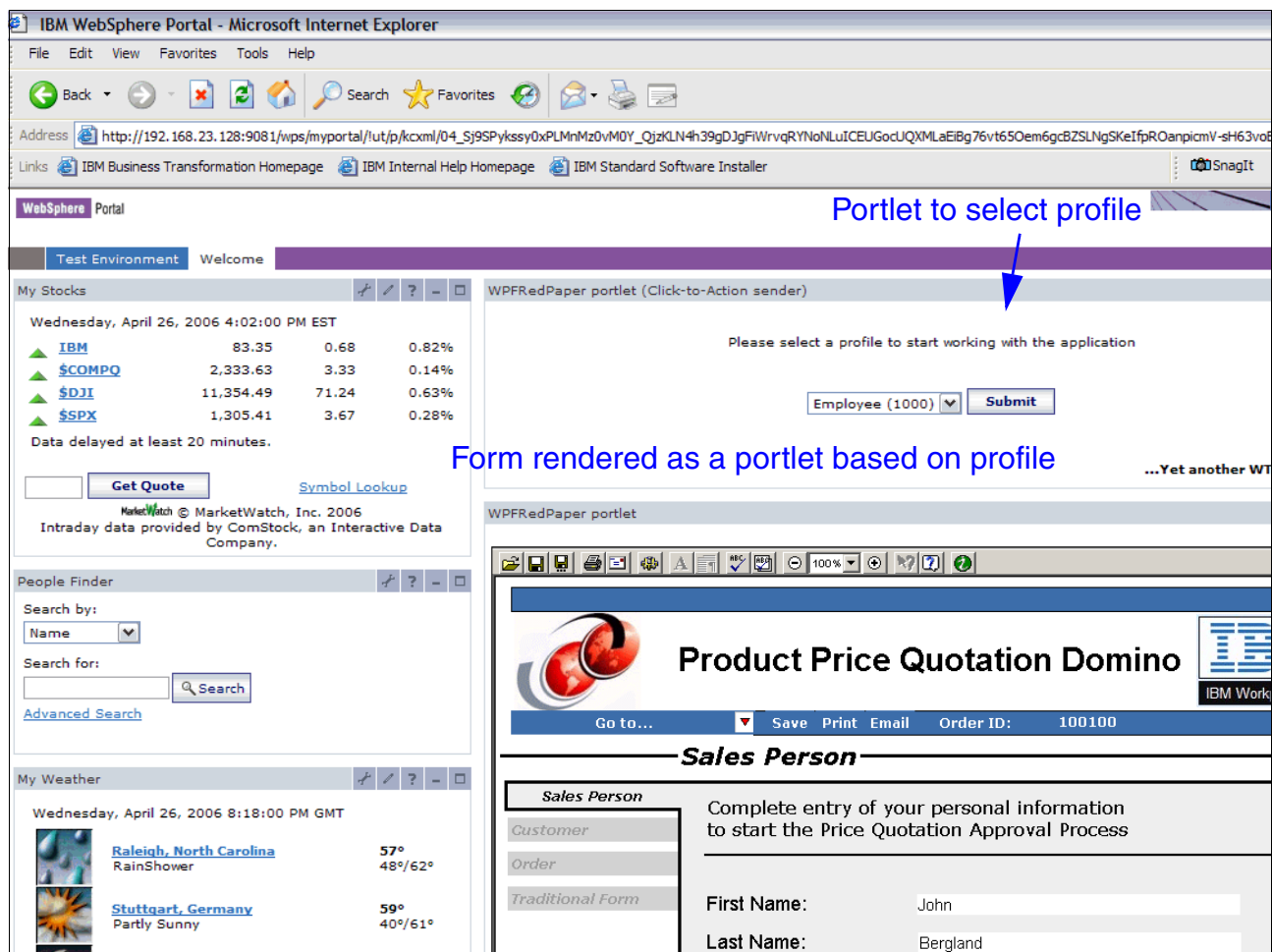


Figure 6-1 How portlets will work together on the Portal page

6.2 Overview of Portal integration

You will need to create and deploy two portlets to the WebSphere Portal server. These portlets can be placed anywhere on the same page. We recommend that they be next to each other since they communicate with each other using WebSphere Portal's Click-to-Action capabilities.

The first portlet is the Click-to-Action Sender portlet and is used to allow the user to set their role. Once the user role has been set, the same Click-to-Action Sender portlet then displays the buttons that a user can select to display the contents of their own unique directories that contain the forms (Figure 6-2).

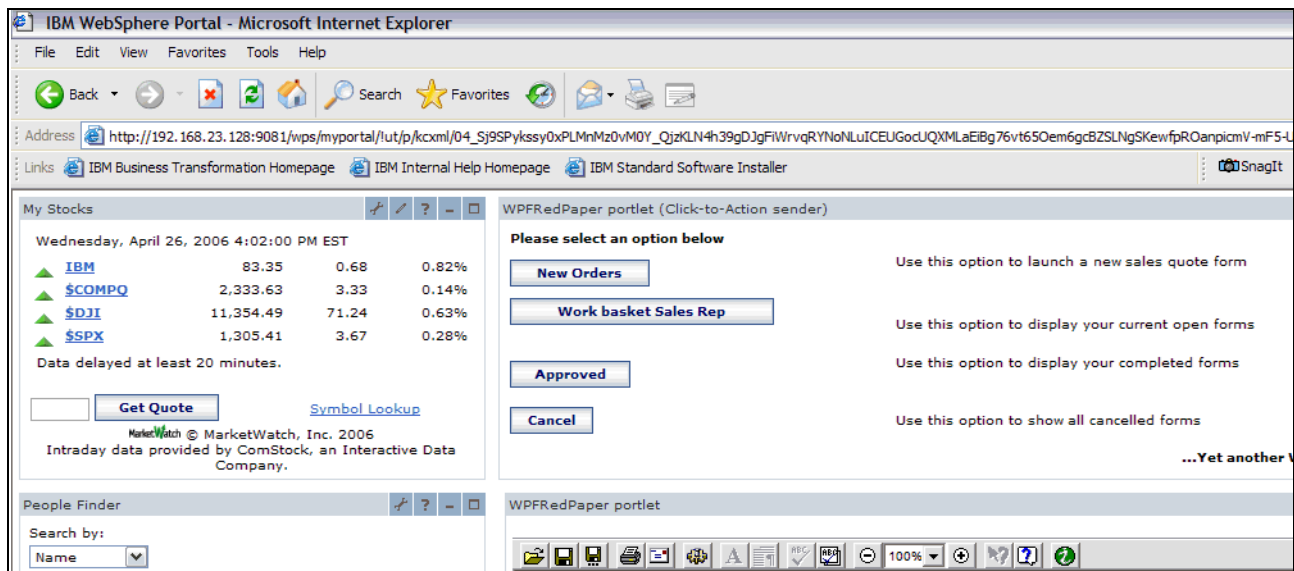


Figure 6-2 Click-to-Action Sender portlet with buttons

When the user clicks one of the buttons, the Click-to-Action Sender portlet reads the directories on the file system of the WebSphere Portal server and then displays a list of forms that a user can select from (Figure 6-3).

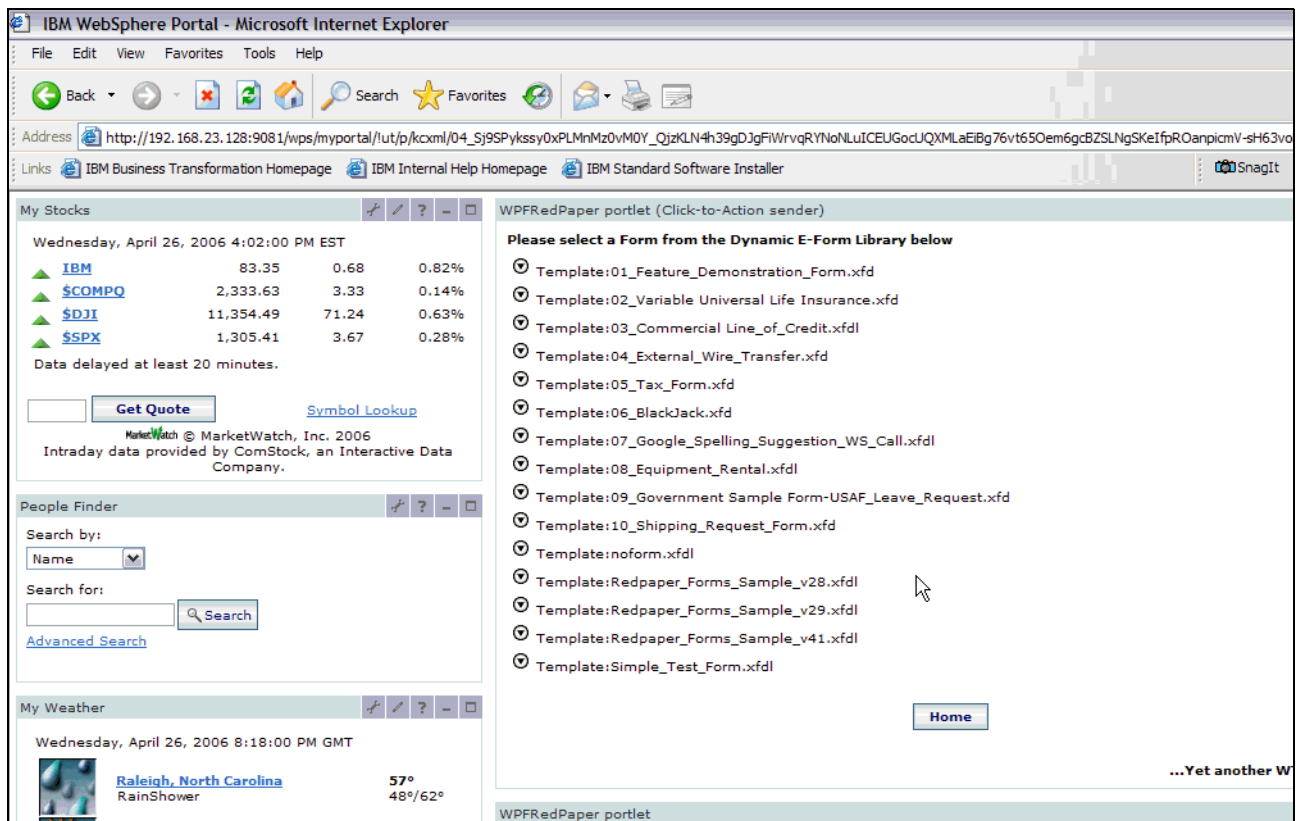


Figure 6-3 Click-to-Action Sender portlet displaying list of forms

The user then clicks on the link to select a form. The Click-to-Action sender portlet uses the Click-to-Action interportlet communication to send the requested form to the second portlet, which is the WPFRedPaper Forms Viewer portlet (Figure 6-4).

WebSphere Portal - Microsoft Internet Explorer

Portlet to select profile

WPFRedPaper portlet (Click-to-Action sender)

Please select a profile to start working with the application

Employee (1000) Submit

WPFRedPaper portlet

Product Price Quotation Domino

Go to... Save Print Email Order ID: 100100 Next

Sales Person

Complete entry of your personal information to start the Price Quotation Approval Process

First Name: John

Last Name: Bergland

Personnel Number: 1010

Email Address: jr@itso.com

Manager: 1031

The information that you provide will be kept confidential and secure and will not be sold or redistributed.

Form rendered as a portlet based on profile

Figure 6-4 WPFRedpaper portlet displaying a Workplace Forms form

Once the form is displayed in the Forms Viewer in the WPFRedpaper portlet, a user can then fill it out and submit it as they would normally do. The completed form is then saved to the file system of the WebSphere Portal server and then goes through the workflow process for approval if required.

6.3 Writing a portlet

In this section we make an assumption that the reader has good knowledge about portlet development. The goal is not to demonstrate the end-to-end process of portlet creation, rather, it is to give an overview of the process flow of information in the portlet.

Note: This portlet example is a complicated one due to the fact that we took an existing J2EE application and rewrote it to fit the *WebSphere Portal environment*. Not everything that we do in this example is required to display a form in the Portal. We will also highlight an easier way to create a portlet that displays a form.

To begin writing a portlet, you need an application development tool such as WebSphere Studio Application Developer (WSAD) or the Rational Application Developer (RAD). For this example we used RAD, and the steps identified below will have a RAD context.

When create a new project using RAD, you will find that a number of files are automatically added to the project, for example, the *portlet.xml* file. You will also need to create some additional files, including some JSPs and Java files, all of which will be explained next.

portlet.xml

This file is used to store the configuration information about the portlet. It is essentially a properties file. When you initialize the portlet, this file is read to set the attributes of the portlet. These attributes are saved in memory so that they can be called at anytime in the code.

The attributes that we set in this file contain the paths to the directories on the file system where we store the forms that the users will process. You will notice that we use both an absolute path to the directories, and a relative path. For example, the absolute path to the directory that the user is pointed to when they click on the **New Orders** button is *C:\Redpaper_Demo\Form_Templates*, and the relative path is */wpfredpaper/forms/Form_Templates/*.

The reason we need both paths is that when you call the method reading the file system in the Java file, you need an absolute path in order for it to be recognized as a directory, whereas once the JSP that is launched to render the contents of the directory, it needs a relative path to render correctly. (Refer to Example 6-1 for the coding.)

Important: You need to manually create the following file directories on the WebSphere Portal's file system:

```
C:\Redpaper_Demo\Form_Templates
C:\Redpaper_Demo\Sales_Rep_Forms
C:\Redpaper_Demo\Manager_Forms
C:\Redpaper_Demo\Director_Forms
C:\Redpaper_Demo\Approved_Forms
C:\Redpaper_Demo\Cancelled_Forms
```

Example 6-1 portlet.xml contents

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE portlet-app-def PUBLIC "-//IBM//DTD Portlet Application 1.1//EN"
"portlet_1.1.dtd">
<portlet-app-def>
  <portlet-app uid="wpfredpaper.WPFRedPaperPortlet.477c98c6a0" major-version="1"
minor-version="0">
    <portlet-app-name>WPFRedPaper application</portlet-app-name>
```

```

    <portlet id="wpfredpaper.WPFRedPaperPortlet"
href="WEB-INF/web.xml#wpfredpaper.WPFRedPaperPortlet" major-version="1" minor-version="0">
    <portlet-name>Display Workplace Form Portlet</portlet-name>
    <cache>
        <expires>0</expires>
        <shared>no</shared>
    </cache>
    <allows>
        <maximized/>
        <minimized/>
    </allows>
    <supports>
        <markup name="html">
            <view />
        </markup>
    </supports>
</portlet>
    <portlet id="wpfredpaper.WPFRedPaperPortletC2ASender"
href="WEB-INF/web.xml#wpfredpaper.WPFRedPaperPortletC2ASender" major-version="1"
minor-version="0">
    <portlet-name>Select Workplace Form Portlet</portlet-name>
    <cache>
        <expires>0</expires>
        <shared>no</shared>
    </cache>
    <allows>
        <maximized />
        <minimized />
    </allows>
    <supports>
        <markup name="html">
            <view />
        </markup>
    </supports>
</portlet>
</portlet-app>
<concrete-portlet-app uid="wpfredpaper.WPFRedPaperPortlet.477c98c6a0.1">
    <portlet-app-name>WPFRedPaper application</portlet-app-name>
    <concrete-portlet href="#wpfredpaper.WPFRedPaperPortlet">
        <portlet-name>Display Workplace Form Portlet</portlet-name>
        <default-locale>en</default-locale>
        <language locale="en">
            <title>WPFRedPaper portlet</title>
            <title-short></title-short>
            <description></description>
            <keywords></keywords>
        </language>
        <config-param>
            <param-name>c2a-action-descriptor</param-name>
            <param-value>wpfredpaper/wsd1/WPFRedPaperPortletC2A.wsd1</param-value>
        </config-param>
        <config-param>
            <param-name>c2a-nls-file</param-name>
            <param-value>wpfredpaper.nls.WPFRedPaperPortletC2A</param-value>
        </config-param>
    </concrete-portlet>
</concrete-portlet-app>
<concrete-portlet-app uid="wpfredpaper.WPFRedPaperPortletC2ASender.477c98c6a0.1">

```

```

    <portlet-app-name>WPFRedPaper application (Click-to-Action
sender)</portlet-app-name>
    <concrete-portlet href="#wpfredpaper.WPFRedPaperPortletC2ASender">
        <portlet-name>Select Workplace Form Portlet</portlet-name>
        <default-locale>en</default-locale>
        <language locale="en">
            <title>WPFRedPaper portlet (Click-to-Action sender)</title>
            <title-short></title-short>
            <description></description>
            <keywords></keywords>
        </language>
        <config-param>
            <param-name>TEMPLATE_FOLDER</param-name>
            <param-value>C:\\Redpaper_Demo\\Form_Templates</param-value>
        </config-param>
        <config-param>
            <param-name>MANAGER_FOLDER</param-name>
            <param-value>C:\\Redpaper_Demo\\Manager_Forms</param-value>
        </config-param>
        <config-param>
            <param-name>DIRECTOR_FOLDER</param-name>
            <param-value>C:\\Redpaper_Demo\\Director_Forms</param-value>
        </config-param>
        <config-param>
            <param-name>APPROVED_FOLDER</param-name>
            <param-value>C:\\Redpaper_Demo\\Approved_Forms</param-value>
        </config-param>
        <config-param>
            <param-name>SALES_REP_FOLDER</param-name>
            <param-value>C:\\Redpaper_Demo\\Sales_Rep_Forms</param-value>
        </config-param>
        <config-param>
            <param-name>CANCELLED_FOLDER</param-name>
            <param-value>C:\\Redpaper_Demo\\Cancelled_Forms</param-value>
        </config-param>

        <config-param>
            <param-name>TEMPLATE_FOLDER_RELATIVE</param-name>
            <param-value>wpfredpaper/forms/Form_Templates/</param-value>
        </config-param>
        <config-param>
            <param-name>MANAGER_FOLDER_RELATIVE</param-name>
            <param-value>wpfredpaper/forms/Manager_Forms/</param-value>
        </config-param>
        <config-param>
            <param-name>DIRECTOR_FOLDER_RELATIVE</param-name>
            <param-value>wpfredpaper/forms/Director_Forms/</param-value>
        </config-param>
        <config-param>
            <param-name>APPROVED_FOLDER_RELATIVE</param-name>
            <param-value>wpfredpaper/forms/Approved_Forms/</param-value>
        </config-param>
        <config-param>
            <param-name>SALES_REP_FOLDER_RELATIVE</param-name>
            <param-value>wpfredpaper/forms/Sales_Rep_Forms/</param-value>
        </config-param>
        <config-param>
            <param-name>CANCELLED_FOLDER_RELATIVE</param-name>
            <param-value>wpfredpaper/forms/Cancelled_Forms/</param-value>
        </config-param>
    </concrete-portlet>
</portlet>

```

```
</concrete-portlet>
</concrete-portlet-app>
</portlet-app-def>
```

When you launch a portlet in Portal, the first thing that it does is look for the Java source and then reads the class of the Java file to find the *doView* method. The *doView* method tells the portlet what to display when it is loaded and then executes the business logic. The *actionPerformed* method reads the *doView* method and then forwards the request to a JSP.

The JSP that is displayed depends on the appropriate action to perform. When the portlet is launched it displays a JSP to render the portlet. We add a *JSP2Display* flag depending on where the user is in the process. For example, the first time a user selects their profile in the drop-down list, we use a variable *JSP2Display* flag to open up the *index.jsp* for that user.

Process flow of information in a portlet

Basically, the information flow occurs as follows:

1. A user selects a portlet to open.
2. The portlet is loaded and calls the *doView* method.
3. The *doView* method sets a *JSP2Display* flag to display the correct JSP.
4. The JSP has an action that has a parameter, *actionName*.
5. The user sets parameters for the form action by clicking a button such as **New Orders**.
6. The submit form action goes back to the *actionPerform* method, which now has all the parameters set.
7. The *actionPerform* method sets the variables in the *doView* method and then calls the *doView*, which forwards the request to launch the next JSP.

WPFRedpaperPortletC2ASender.java

Example 6-2 is an example of the coding that the Java source should contain, including all the methods we just mentioned above. Create a Java file in your project named *WPFRedpaperPortletC2ASender.java*, copy this coding to it, and save the file.

Example 6-2 Contents of WPFRedpaperPortletC2ASender.java

```
public class WPFRedPaperPortletC2ASender
    extends PortletAdapter
    implements ActionListener, MessageListener {

    public static final String LIST_TEMPLATES = "listTemplates";
    // Action name to display list of available forms

    public static final String LIST_WORKBASKET_SALES = "workbasketsales";
    // Action name to display list of available forms

    public static final String LIST_WORKBASKET_MANAGER = "workbasketmanager";
    // Action name to display list of available forms

    public static final String LIST_WORKBASKET_DIRECTOR = "workbasketdirector";
    // Action name to display list of available forms

    public static final String LIST_CANCEL = "listCancelled";
    // Action name to display list of available forms
```

```

public static final String LIST_APPROVED = "listApproved";
// Action name to display list of available forms

public static final String OPEN_FORM = "openSelected";
// Action name to open selected form

public static final String PROFILE_FORM = "profileSelected";
// Action name to open selected form


public static final String TEST_JSP = "/wpfredpaper/jsp/TestBean.";
// Action name to open user profile for the application

public static final String FORM_PROFILE_JSP = "/wpfredpaper/jsp/profile.";
// Action name to open user profile for the application

public static final String OPEN_FORM_JSP = "/wpfredpaper/jsp/open.";
// Action name to open selected form

public static final String INDEX_JSP = "/wpfredpaper/jsp/index.";
// JSP file name to be rendered on the view mode

public static final String LIST_JSP = "/wpfredpaper/jsp/list.";
// JSP file name to be rendered on the view mode

public static final String VIEW_JSP =
    "/wpfredpaper/jsp/WPFRedPaperPortletC2ASenderView.";
// JSP file name to be rendered on the view mode
public static final String VIEW_BEAN =
    "formredpapersample.FormRedPaperSamplePortletViewBean";
// Bean name for the view mode request
public static final String SESSION_BEAN =
    "formredpapersample.FormRedPaperSamplePortletSessionBean";
// Bean name for the portlet session
public static final String FORM_ACTION =
    "formredpapersample.FormRedPaperSamplePortletFormAction";
// Action name for the orderId entry form
public static final String TEXT =
    "formredpapersample.FormRedPaperSamplePortletText";
// Parameter name for general text input
public static final String SUBMIT =
    "formredpapersample.FormRedPaperSamplePortletSubmit";
// Parameter name for general submit button
public static final String CANCEL =
    "formredpapersample.FormRedPaperSamplePortletCancel";
// Parameter name for general cancel button

/**
 * @see org.apache.jetspeed.portlet.Portlet#init(PortletConfig)
 */
public void init(PortletConfig portletConfig) throws UnavailableException {
    super.init(portletConfig);
}
}

```

```

/**
 * @see org.apache.jetspeed.portlet.PortletAdapter#doView(PortletRequest,
PortletResponse)
 */
public void doView(PortletRequest request, PortletResponse response)
    throws PortletException, IOException {
    // Check if portlet session exists
    WPFRedPaperPortletSessionBean sessionBean =
        getSessionBean(request);
    if (sessionBean == null) {
        response.getWriter().println("<b>NO PORTLET SESSION YET</b>");
        return;
    }

    // Read in the Portlet.xml file for the submission folder names
    PortletSettings settings = request.getPortletSettings();

    String TEMPLATE_FOLDER = settings.getAttribute("TEMPLATE_FOLDER");
    String TEMPLATE_FOLDER_RELATIVE =
settings.getAttribute("TEMPLATE_FOLDER_RELATIVE");

    String MANAGER_FOLDER = settings.getAttribute("MANAGER_FOLDER");
    String MANAGER_FOLDER_RELATIVE =
settings.getAttribute("MANAGER_FOLDER_RELATIVE");

    String DIRECTOR_FOLDER = settings.getAttribute("DIRECTOR_FOLDER");
    String DIRECTOR_FOLDER_RELATIVE =
settings.getAttribute("DIRECTOR_FOLDER_RELATIVE");

    String APPROVED_FOLDER = settings.getAttribute("APPROVED_FOLDER");
    String APPROVED_FOLDER_RELATIVE =
settings.getAttribute("APPROVED_FOLDER_RELATIVE");

    String SALES_REP_FOLDER = settings.getAttribute("SALES_REP_FOLDER");
    String SALES_REP_FOLDER_RELATIVE =
settings.getAttribute("SALES_REP_FOLDER_RELATIVE");

    String CANCELLED_FOLDER = settings.getAttribute("CANCELLED_FOLDER");
    String CANCELLED_FOLDER_RELATIVE =
settings.getAttribute("CANCELLED_FOLDER_RELATIVE");

    // Save Variables in Memory
    request.setAttribute("TEMPLATE_FOLDER", TEMPLATE_FOLDER);
    request.setAttribute("TEMPLATE_FOLDER_RELATIVE", TEMPLATE_FOLDER_RELATIVE);

    request.setAttribute("MANAGER_FOLDER", MANAGER_FOLDER);
    request.setAttribute("MANAGER_FOLDER_RELATIVE", MANAGER_FOLDER_RELATIVE);

    request.setAttribute("DIRECTOR_FOLDER", DIRECTOR_FOLDER);
    request.setAttribute("DIRECTOR_FOLDER_RELATIVE", DIRECTOR_FOLDER_RELATIVE);

    request.setAttribute("APPROVED_FOLDER", APPROVED_FOLDER);
    request.setAttribute("APPROVED_FOLDER_RELATIVE", APPROVED_FOLDER_RELATIVE);

    request.setAttribute("SALES_REP_FOLDER", SALES_REP_FOLDER);
    request.setAttribute("SALES_REP_FOLDER_RELATIVE", SALES_REP_FOLDER_RELATIVE);

```

```

        request.setAttribute("CANCELLED_FOLDER",CANCELLED_FOLDER);
        request.setAttribute("CANCELLED_FOLDER_RELATIVE",CANCELLED_FOLDER_RELATIVE);

// Make a view mode bean
WPFRedPaperPortletViewBean viewBean = new WPFRedPaperPortletViewBean();
request.setAttribute(VIEW_BEAN, viewBean);

// Set actionURI in the view mode bean
PortletURI formActionURI = response.createURI();
formActionURI.addAction(FORM_ACTION);
viewBean.setFormActionURI(formActionURI.toString());

;

// Invoke the JSP to render based on jsp2Display flag
String actionName = (String) request.getAttribute("actionName");
String jsp2Display = (String) request.getAttribute("jsp2Display");

if (jsp2Display == null) {

    getPortletConfig().getContext().include(
        VIEW_JSP + getJspExtension(request),
        //TEST_JSP + getJspExtension(request),
        request,
        response);

}

else if (jsp2Display=="profileSelected") {

    getPortletConfig().getContext().include(
        INDEX_JSP + getJspExtension(request),
        request,
        response);

}

else if (jsp2Display=="listTemplates") {

    getPortletConfig().getContext().include(
        LIST_JSP + getJspExtension(request),
        request,
        response);

}

else if (jsp2Display=="index") {

```

```

        getPortletConfig().getContext().include(
            INDEX_JSP + getJspExtension(request),
            request,
            response);
    } else if (jsp2Display=="openSelected") {

        getPortletConfig().getContext().include(
            OPEN_FORM_JSP + getJspExtension(request),
            request,
            response);}

    else if (jsp2Display=="workbasketsales") {

        getPortletConfig().getContext().include(
            LIST_JSP + getJspExtension(request),
            request,
            response);}

    else if (jsp2Display=="workbasketdirector") {

        getPortletConfig().getContext().include(
            LIST_JSP + getJspExtension(request),
            request,
            response);}

    else if (jsp2Display=="workbasketmanager") {

        getPortletConfig().getContext().include(
            LIST_JSP + getJspExtension(request),
            request,
            response);}

    else if (jsp2Display=="listApproved") {

        getPortletConfig().getContext().include(
            LIST_JSP + getJspExtension(request),
            request,
            response);}

    else if (jsp2Display=="listCancelled") {

        getPortletConfig().getContext().include(
            LIST_JSP + getJspExtension(request),
            request,
            response);}

}

/**
 * @see org.apache.jetspeed.portlet.event.ActionListener#actionPerformed(ActionEvent)
 */
public void actionPerformed(ActionEvent event) throws PortletException {
    if (getPortletLog().isDebugEnabled())
        getPortletLog().debug("ActionListener - actionPerformed called");
}

```

```

// ActionEvent handler
String actionString = event.getActionString();

// Add action string handler here
PortletRequest request = event.getRequest();
WPFRedPaperPortletSessionBean sessionBean = getSessionBean(request);

if (PROFILE_FORM.equals(actionString)) {

    String profile=request.getParameter("listProfiles");
    String action=request.getParameter("action");
    request.setAttribute("userRole",profile);
    request.setAttribute("actionName",action);
    request.setAttribute("jsp2Display","index");

}

else if (LIST_TEMPLATES.equals(actionString)) {
    // Set form text in the session bean
    //sessionBean.setFormText(request.getParameter(TEXT));

    String action=request.getParameter("action");
    request.setAttribute("actionName",action);
    request.setAttribute("jsp2Display",LIST_TEMPLATES);

}

else if (OPEN_FORM.equals(actionString)) {
    // Set form text in the session bean

    String action=request.getParameter("action");
    String formselected=request.getParameter("eFormList");
    String relativelink=request.getParameter("relativeLink");
    String folderselected=request.getParameter("folderselected");

    request.setAttribute("actionName",action);
    request.setAttribute("jsp2Display",OPEN_FORM);

    request.setAttribute("form2Open",formselected);
    request.setAttribute("folderselected",folderselected);

    request.setAttribute("relativeLink",relativeLink);

}

else if (LIST_WORKBASKET_SALES.equals(actionString)) {

```

```

        // Set form text in the session bean
        String action=request.getParameter("action");
        request.setAttribute("actionName",action);
        request.setAttribute("jsp2Display",LIST_WORKBASKET_SALES);}

    else if (LIST_WORKBASKET_MANAGER.equals(actionString)) {
        // Set form text in the session bean
        String action=request.getParameter("action");
        request.setAttribute("actionName",action);
        request.setAttribute("jsp2Display",LIST_WORKBASKET_MANAGER);}

    else if (LIST_WORKBASKET_DIRECTOR.equals(actionString)) {
        // Set form text in the session bean
        String action=request.getParameter("action");
        request.setAttribute("actionName",action);
        request.setAttribute("jsp2Display",LIST_WORKBASKET_DIRECTOR);}

    else if (LIST_APPROVED.equals(actionString)) {

        String action=request.getParameter("action");
        request.setAttribute("actionName",action);
        request.setAttribute("jsp2Display",LIST_TEMPLATES);

    }

    else if (LIST_CANCEL.equals(actionString)) {

        String action=request.getParameter("action");
        request.setAttribute("actionName",action);
        request.setAttribute("jsp2Display",LIST_TEMPLATES);

    }

}

/**
 * @see org.apache.jetspeed.portlet.event.MessageListener#messageReceived(MessageEvent)
 */
public void messageReceived(MessageEvent event) throws PortletException {
    if (getPortletLog().isDebugEnabled())
        getPortletLog().debug("MessageListener - messageReceived called");
    // MessageEvent handler
    PortletMessage msg = event.getMessage();
    // Add PortletMessage handler here
    if (msg instanceof DefaultPortletMessage) {

```

```

        String messageText = ((DefaultPortletMessage) msg).getMessage();
        // Add DefaultPortletMessage handler here
    } else {
        // Add general PortletMessage handler here
    }
}

/**
 * Get SessionBean.
 *
 * @param request PortletRequest
 * @return FormRedPaperSamplePortletSessionBean
 */
private WPFRedPaperPortletSessionBean getSessionBean(PortletRequest request) {
    PortletSession session = request.getPortletSession();
    if (session == null)
        return null;
    WPFRedPaperPortletSessionBean sessionBean =
        (WPFRedPaperPortletSessionBean) session.getAttribute(
            SESSION_BEAN);
    if (sessionBean == null) {
        sessionBean = new WPFRedPaperPortletSessionBean();
        session.setAttribute(SESSION_BEAN, sessionBean);
    }
    return sessionBean;
}

/**
 * Returns the file extension for the JSP file
 *
 * @param request PortletRequest
 * @return JSP extension
 */
private static String getJspExtension(PortletRequest request) {
    String markupName = request.getClient().getMarkupName();
    return ".jsp";
}
}

```

Creating the JSP files to display in the Portal

You will need to create four JSP files in order to display the correct information in the portlets. These JSPs are:

1. WPFFormsRedPaperPortletC2ASenderView.jsp
2. index.jsp
3. list.jsp
4. WPFFormsRedPaperPortletView.jsp

WPFormsRedPaperPortletC2ASenderView.jsp

This file is used to display the profile selection so that a user can set their role for the session (Figure 6-5).

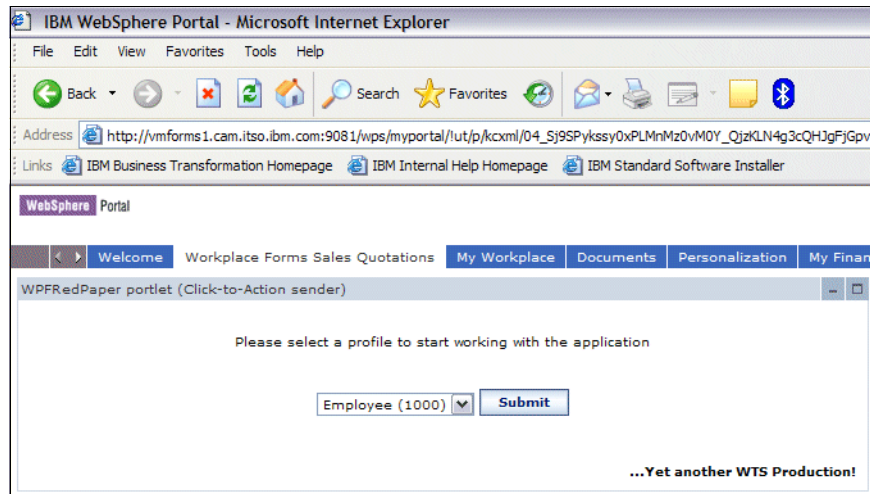


Figure 6-5 WPFormsPortletC2ASenderView.jsp

The contents of the file should be as follows (Example 6-3).

Example 6-3 Contents of WPFormsPortletC2ASenderView.jsp

```
<%@ taglib uri="/WEB-INF/tld/c2a.tld" prefix="C2A" %>
<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix="portletAPI" %>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
session="true"
import="java.util.*,java.io.File, wpfredpaper.*"%>

<portletAPI:init/>

<P>
<TABLE border="0" cellpadding="2" width="100%">
  <TBODY>
    <TR>
      <TD colspan="3" ALIGN=MIDDLE><BR>
        <FORM method="POST" action="<portletAPI:createURI><portletAPI:URIAction
name='profileSelected' /></portletAPI:createURI>">
          <INPUT type="hidden" name=action value="profileSelected">

          Please select a profile to start working with the application<P><BR>
          <SELECT name="listProfiles" size="0">
            <OPTION value="1000">Employee (1000)</OPTION>
            <OPTION value="1010">Manager (1010)</OPTION>
            <OPTION value="1031">Director (1031)</OPTION>
          </SELECT>

          <INPUT class="wpsButtonText" type="submit" value="Submit">
        </FORM>
      <P><BR>
    </TD>
  </TR>
</TABLE>
```

```

        </TR>
        <TR bgcolor=""><TD COLSPAN=3 ALIGN=RIGHT><B>...Yet another WTS
Production!</B></TD></TR>
    </TBODY>
</TABLE></CENTER>

```

index.jsp

This file is used to display the buttons that a user can select once their profile has been set (Figure 6-6).

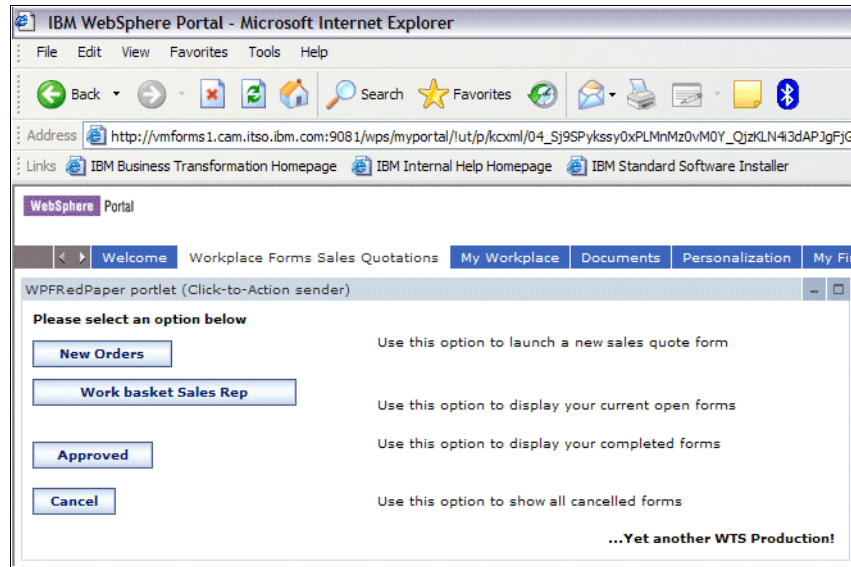


Figure 6-6 index.jsp

The contents of this file should be as shown in Example 6-4.

Example 6-4 index.jsp contents

```

<%@ taglib uri="/WEB-INF/tld/c2a.tld" prefix="C2A" %>
<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix="portletAPI" %>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
session="true"
import="java.util.*,java.io.File, wpfredpaper.*"%>

<portletAPI:init/>

<jsp:useBean id="RedPaper"class="wpfredpaper.WPFRedPaperPortletViewBean"
scope="session"></jsp:useBean>
<jsp:setProperty name="RedPaper" property="userRole"
value='<%=request.getAttribute("userRole")%>' />

<TABLE border="0" width="100%" align="center">
    <TR>
        <TD><B>Please select an option below</B><BR>
        </TD>
    </TR>

```

```

</TABLE>

<CENTER>

<TABLE width="100%">
  <TR>
    <TD>
      <FORM method="POST"
        action="<portletAPI:createURI><portletAPI:URIAction
name='listTemplates' /></portletAPI:createURI>">

        <INPUT class="wpsButtonText" type="submit" value="New Orders"> <INPUT type="hidden"
          name=action value="listTemplates">
        </TD>
      <TD><FONT size="-2">Use this option to launch a new sales quote form</FONT>
      </FORM>
    </TD>
  </TR>
  <TR>
    <TD>

      <%
String Sales="1000";
String Director="1031";
String Manager="1010";

if (request.getAttribute("userRole").equals(Sales)) { %>
  <FORM method="POST" action="<portletAPI:createURI><portletAPI:URIAction
name='workbasketsales' /></portletAPI:createURI>">
    <INPUT class="wpsButtonText" type="submit" value="Work basket Sales Rep"> <INPUT
type="hidden" name=action value="workbasketsales"></form>

    <% } else if (request.getAttribute("userRole").equals(Director)) { %>

      <FORM method="POST" action="<portletAPI:createURI><portletAPI:URIAction
name='workbasketdirector' /></portletAPI:createURI>">
        <INPUT class="wpsButtonText" type="submit" value="Work Basket Director"> <INPUT
type="hidden" name=action value="workbasketdirector"></form>

        <% } else if (request.getAttribute("userRole").equals(Manager)) { %>

          <FORM method="POST" action="<portletAPI:createURI><portletAPI:URIAction
name='workbasketmanager' /></portletAPI:createURI>">
            <INPUT class="wpsButtonText" type="submit" value="Work Basket Manager"> <INPUT
type="hidden" name=action value="workbasketmanager"></form>
            <%}%>

          </TD>
        <TD><FONT size="-2">Use this option to display your current open forms</FONT>
        </FORM>
      </TD>
    </TR>
    <TR>
      <TD>
        <FORM method="POST" action="<portletAPI:createURI><portletAPI:URIAction
name='listApproved' /></portletAPI:createURI>">

```

```

        <INPUT class="wpsButtonText" type="submit" value="Approved"> <INPUT type="hidden"
name=action value="listApproved">
    </TD>
    <TD><FONT size="-2">Use this option to display your completed forms</FONT>
</FORM>
</TD>
</TR>
<TR>
    <TD></TD>
    <TD></TD>
</FORM>
</TD>
</TR>
<TR>
    <TD></TD>
    <TD></TD>
</FORM>
</TD>
</TR>
<TR>
    <TD>
<FORM method="POST" action="<portletAPI:createURI><portletAPI:URIAction
name='listCancelled'/></portletAPI:createURI>">
        <INPUT class="wpsButtonText" type="submit" value="Cancel"> <INPUT type="hidden"
name=action value="listCancelled">

        </TD>
        <TD><FONT size="-2">Use this option to show all cancelled forms</FONT>
</TD>
</FORM>
</TD>
</TR>

<TR>
    <TD colspan="3">

        <TABLE>
            <TR>
                <TD colspan="3"></TD>
            </TR>
        <TABLE>

            </TD>
        </TR>
    </TABLE>

    </FORM>

    <TABLE width="100%" align="center">
        <tr>
            <td align="right"><B>...Yet another WTS Production!</B></td>
        </tr>
    </TABLE>
</CENTER>

</CENTER>

```

Important: We decided to implement security in this way for demonstration purposes. In a real portal environment, you would need to modify the LDAP directory to add the roles of the employee, manager, and director.

list.jsp

This file displays the contents of all the forms that are stored in the folders on the file system of the Portal server. This list changes based on the user role selected and the view of the button they choose (Figure 6-7).

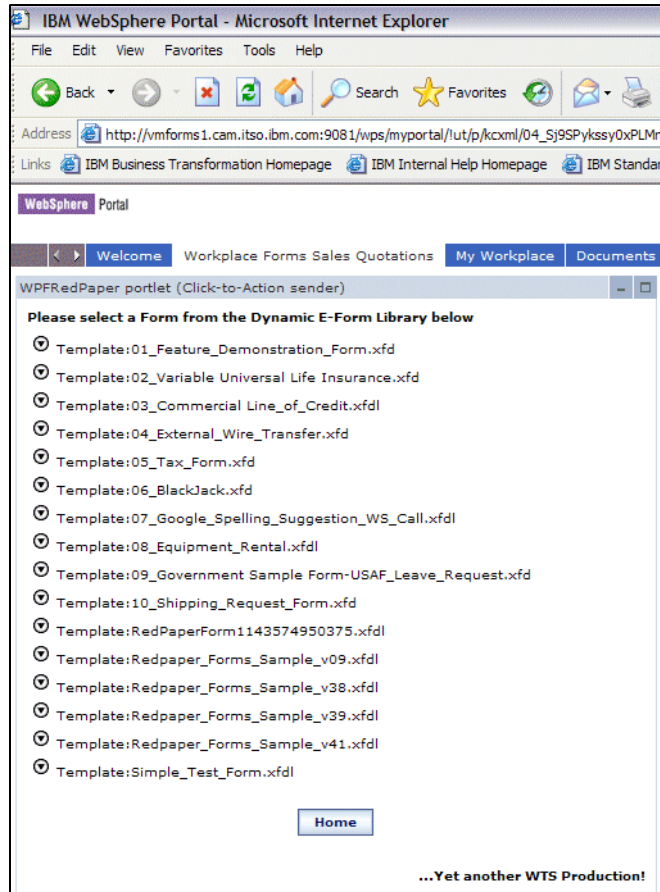


Figure 6-7 list.jsp

The contents of this file are as follows (Example 6-5).

Example 6-5 Contents of list.jsp

```
<%@ taglib uri="/WEB-INF/tld/c2a.tld" prefix="C2A" %>
<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix="portletAPI" %>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
session="true"
import="java.util.*,java.io.File, wpfredpaper.*,org.apache.jetspeed.portlet.*,
org.apache.jetspeed.portlet.event.*" %>

<portletAPI:init/>
```

<%

```
String action = (String)request.getAttribute("actionName");

    if (action.equalsIgnoreCase("listTemplates")) {

        request.setAttribute("FOLDER", request.getAttribute("TEMPLATE_FOLDER"));
        request.setAttribute("FOLDER_RELATIVE",
request.getAttribute("TEMPLATE_FOLDER_RELATIVE"))
        ;}

        else if (action.equalsIgnoreCase("workbasketsales")) {
            request.setAttribute("FOLDER",
(String)request.getAttribute("SALES_REP_FOLDER"));
            request.setAttribute("FOLDER_RELATIVE",
(String)request.getAttribute("SALES_REP_FOLDER_RELATIVE"));
        }

        else if (action.equalsIgnoreCase("workbasketmanager")) {
            request.setAttribute("FOLDER",
(String)request.getAttribute("MANAGER_FOLDER"));
            request.setAttribute("FOLDER_RELATIVE",
(String)request.getAttribute("MANAGER_FOLDER_RELATIVE"));
        }

        else if (action.equalsIgnoreCase("workbasketdirector")) {
            request.setAttribute("FOLDER",
(String)request.getAttribute("DIRECTOR_FOLDER"));
            request.setAttribute("FOLDER_RELATIVE",
(String)request.getAttribute("DIRECTOR_FOLDER_RELATIVE"));
        }

        else if (action.equalsIgnoreCase("listApproved")) {
            request.setAttribute("FOLDER", request.getAttribute("APPROVED_FOLDER"));
            request.setAttribute("FOLDER_RELATIVE",
request.getAttribute("APPROVED_FOLDER_RELATIVE"));
        }

        else if (action.equalsIgnoreCase("listCancelled")) {
            request.setAttribute("FOLDER",
(String)request.getAttribute("CANCELLED_FOLDER"));
            request.setAttribute("FOLDER_RELATIVE",
(String)request.getAttribute("CANCELLED_FOLDER_RELATIVE"));
        }

String path = (String) request.getAttribute("FOLDER");
String theDir = path;
String prepop = (String) request.getAttribute("PrePop");

//userID="1002";
//prepop will control link behavior
if (prepop == null)
    prepop = "No";
```

```

prepop = "Yes";

//The path to the current template, including the template name
String servletPath = application.getRealPath(request.getServletPath());

File dir = new File(theDir);
try {
    if (dir.isDirectory()) {
        String[] children = dir.list();
    } %>

<TABLE border="0" width="100%" align="center">
    <TR>
        <TD><B>Please select a Form from the Dynamic E-Form
        Library below</B><BR>
        </TD>
    </TR>
    <TR>
        <TD>

            <TABLE border="0">
                <FORM method="POST"
                    action="<portletAPI:createURI><portletAPI:URIAction
name='openSelected'/></portletAPI:createURI>">
                    <%String[] child = dir.list();
for (int i = 0; i < child.length; i++) {
    File Eform = new File(dir, child[i]);%>
                    <TR>
                        <TD>

                            <%String formPath=request.getAttribute("FOLDER_RELATIVE")+Eform.getName();%>

                            <C2A:encodeProperty
namespace="http://www.ibm.com/wps/c2a/examples/FormRedPaper"
type="FormName" value="<%=formPath%>" broadcast="true"
generateMarkupWhenNested="true"/>
                            <%= "Template:" + Eform.getName() %></TD>
                        </TR>
                        <%}%>

                        <%} catch (Exception ex) {
ex.printStackTrace();
}%>

                        <tr>
                            <td>
                                <p><br> <INPUT type="hidden"
                                name=action value="openSelected">
                            </td>
                        </tr>
                    </form>
                </TABLE>
            </TD>
        </TR>
    <TR>
        <TD align="center">
            <FORM method="POST"
                action="<portletAPI:createURI><portletAPI:URIAction
name='index'/></portletAPI:createURI>">

```

```

        <INPUT class="wpsButtonText" type="submit" value="Home"> <INPUT type="hidden"
name=action
        value="index"></FORM>
    </TR>
</TABLE>

<TABLE width="100%" align="center">
    <tr>
        <td align="right"><B>...Yet another WTS Production!</B></td>
    </tr>
</TABLE>

```

WPFormsRedPaperPortletView.jsp

This file allows you to display the IBM Workplace Forms Viewer in a portlet window and then to show the selected form (Figure 6-8).

The screenshot displays a web browser window showing the IBM Workplace Forms Viewer. The browser's address bar shows a URL starting with 'http://www.ibm.com/...'. The page has a navigation bar with links like 'My Portal', 'Administration', 'Edit my profile', and 'Log out'. Below this is a search bar and a 'My Favorites' dropdown. The main content area is titled 'Product Price Quotation' and features a sidebar with navigation links: 'Sales Person', 'Customer', 'Order', and 'Traditional Form'. The 'Sales Person' link is selected, and the form displays fields for 'First Name' (Christine), 'Last Name' (Haas), 'Personnel Number' (1000), 'Email Address' (1000.Christine.Haas@ACME.cam.itso.ibm.co), and 'Manager' (1010). The form also includes a 'Go to...' dropdown, 'Save', 'Print', 'Email', and 'Order ID: 1000126' buttons. A disclaimer at the bottom states: 'The information that you provide will be kept confidential and secure and will not be sold or redistributed.'

Figure 6-8 WPFormsRedPaperPortletView.jsp

The contents of this file should be as follows (Example 6-6).

Example 6-6 Contents of WPFormsRedPaperPortletView.jsp

```
<%@ page session="true" contentType="text/html" import="java.util.*, wpfredpaper.*"%>
<%@ taglib uri="/WEB-INF/tld/c2a.tld" prefix="C2A" %>
<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix="portletAPI" %>
<portletAPI:init/>

<jsp:useBean id="RedPaperC2A" class="wpfredpaper.WPFRedPaperPortletSessionBean"
scope="session"></jsp:useBean>

<%
    WPFRedPaperPortletViewBean viewBean =
    (WPFRedPaperPortletViewBean)portletRequest.getAttribute(WPFRedPaperPortlet.VIEW_BEAN);
    WPFRedPaperPortletSessionBean sessionBean =
    (WPFRedPaperPortletSessionBean)portletRequest.getPortletSession().getAttribute(WPFRedPaperP
ortlet.SESSION_BEAN);
%>

<DIV style="margin: 6px">

<H3 style="margin-bottom: 3px"></H3>
<DIV style="margin: 12px; margin-bottom: 36px"><% /***** Start of sample code *****/
%>

<%
    String formText = sessionBean.getFormText();
    String urlComputed=null;
    if( formText.length()>0 ) {
        %>
<!--
This is the path to the form to be open : '<%=formText%>'.<% }

        %> -->
<FORM method="POST" action="<%=viewBean.getFormActionURI()%>"><LABEL
    class="wpsLabelText"
    for="<portletAPI:encodeNamespace value='<%=WPFRedPaperPortlet.TEXT%>' />"></LABEL></FORM>
</DIV>
</DIV>

<!--To make sure that I display some Text in case no form is selected yet-->

<% if( formText.length()>0 ) { %>

<OBJECT id="Object1" height=640 width=700 border="0"
    classid="CLSID:354913B2-7190-49C0-944B-1507C9125367">
    <PARAM NAME="XFIDLID" VALUE="XFIDLData">
    <PARAM NAME="refresh_url" VALUE="envAware.html">
    <PARAM NAME="TTL" VALUE="17">
    <PARAM NAME="retain_Viewer" VALUE="off">
</OBJECT>

<SCRIPT language="XFIDL" id="XFIDLData"
    type="application/vnd.xfdl; wrapped=comment">
<!--
```

```

<jsp:include page='<%=formText%>' />

-->
</SCRIPT>
<p><br>

<jsp:setProperty name="RedPaperC2A" property="formText" value='test' />

<% } else { %> <!--Here is the else in case No form is selected yet-->
Please select a form to be displayed in the Select form portlet <}%>

```

6.4 Parking the Workplace Forms Viewer in the Portal

Embedding an XFDL form in an HTML page allows users to view forms inside the portal environment. Furthermore, it allows application designers to manipulate the embedded form like any other HTML object. Unlike other embedded objects, such as HTML or XML forms, embedded XFDL forms maintain user data even after the Web page is changed or refreshed.

To embed a form, you must use two HTML elements:

- **objects** — An object allows you to display non-HTML data in the browser. It also allows you to define the size and borders of an object.
- **scripts** — The script contains and loads the form. You can also use other scripts to contain data that is used to modify the form, such as XML instances.

The HTML *object* is a placeholder. It specifies the location where the object will be displayed in relation to the other elements on the Web page.

The *script* element is not displayed as part of the Web page, so it can be placed anywhere in your HTML code.

Tip: The only piece of code required to display the IBM Workplace Forms viewer in a portlet is the following which you can add to any JSP. This is a simpler method of showing this integration.

```

<OBJECT id="Object1" height=640 width=700 border="0"
  classid="CLSID:354913B2-7190-49C0-944B-1507C9125367">
  <PARAM NAME="XFDLID" VALUE="XFDLData">
  <PARAM NAME="refresh_url" VALUE="envAware.html">
  <PARAM NAME="TTL" VALUE="17">
  <PARAM NAME="retain_Viewer" VALUE="off">
</OBJECT>
<SCRIPT language="XFDL" id="XFDLData"
  type="application/vnd.xfdl; wrapped=comment">
<!--
<jsp:include page='<%=formText%>' />

-->
</SCRIPT>

```

6.5 Deploying the portlet

Once you have packaged your portlet, you are ready to deploy it to your portal server (see Figure 6-9). To do this, follow these steps:

1. Log into your portal server as the administrator and click the **Administration** link.
2. Select the **Portlet Management** → **Web Modules** link on the menu on the left.
3. Click the **Install** button.

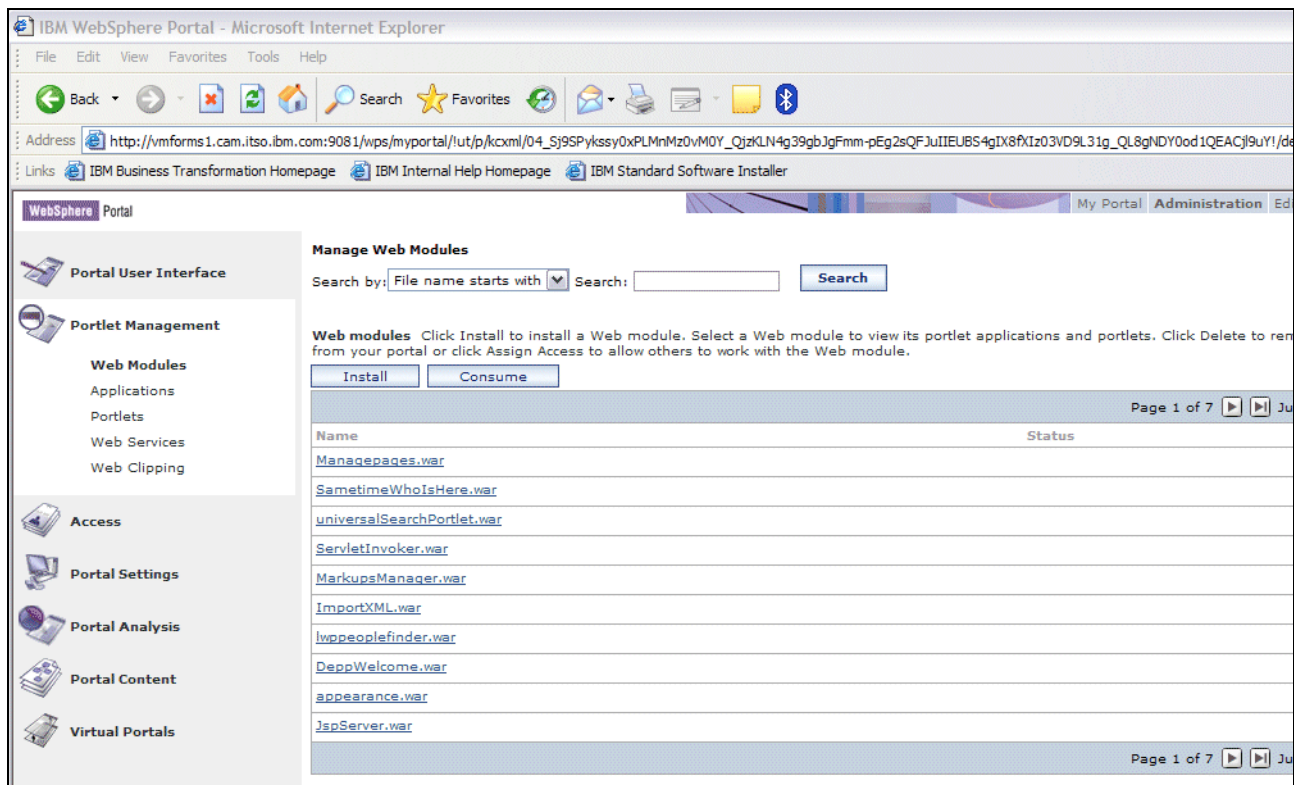


Figure 6-9 Portlet Management Administration Page

4. Click the **Browse** button to locate the WPFRedpaper.war file (Figure 6-10).

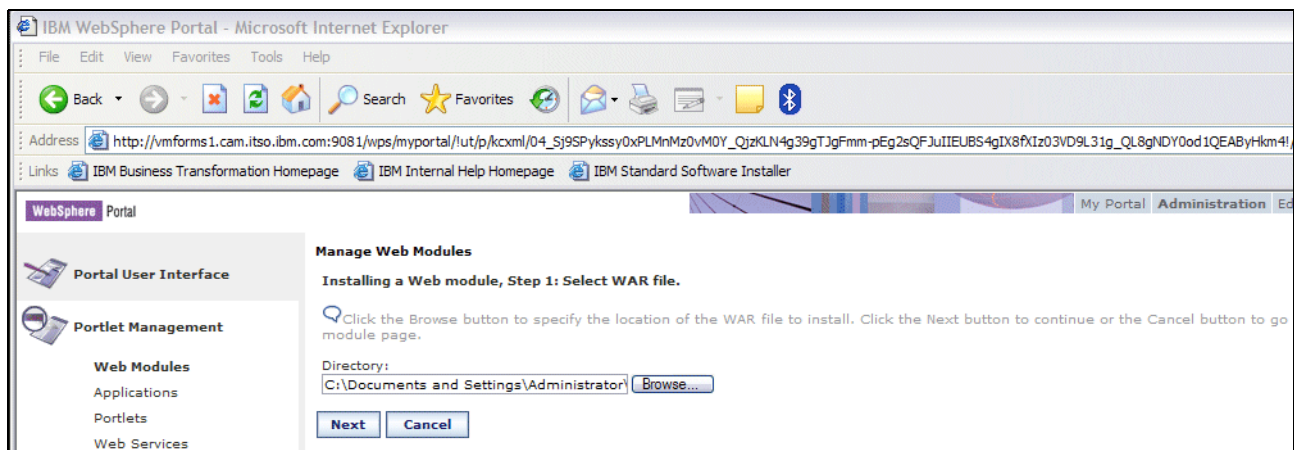


Figure 6-10 Selecting the WAR file

5. On the next screen (Figure 6-11) you will see the contents of the WAR file displayed, and this includes a *FormRedPaper application* and a *FormRedPaper application (Click-to-action Sender)*. Click the **Finish** button to complete.

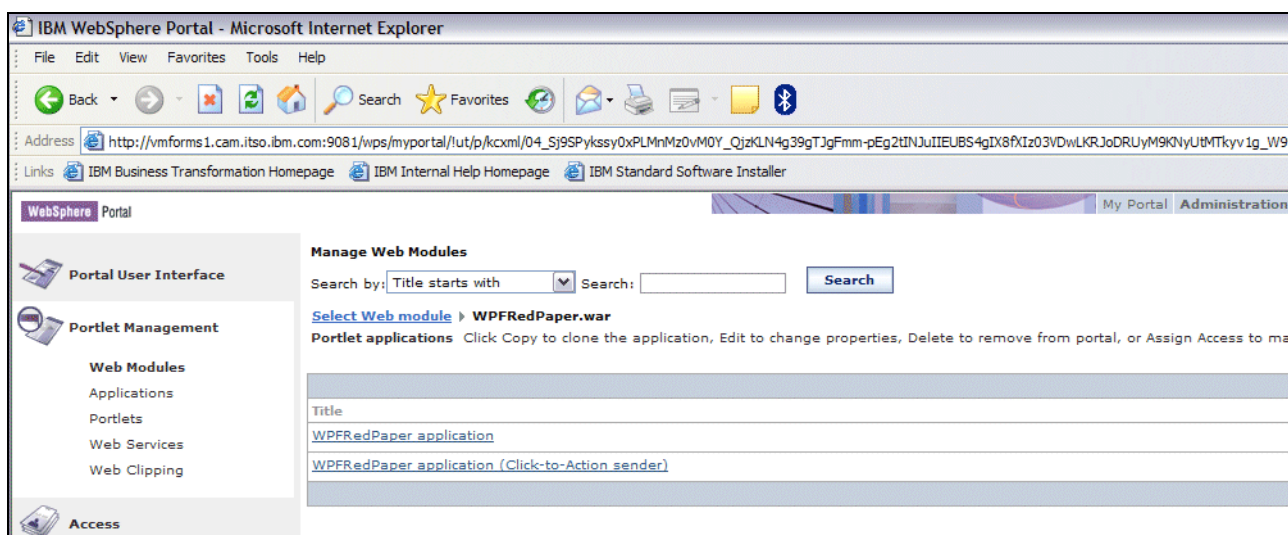


Figure 6-11 WAR file contents

Adding the portlet to a page

The next step is to create a page with which to view the portlet. In order to do this you need to click on the **Portal User Interface** → **Manage Pages** link on the left hand menu of the **Administration** page. You can create a new page and place it anywhere.

For this example, we are going to create a new page named *Workplace Forms Sales Quotations* and store it under the **My Portal** page.

1. Click the **New Page** button and fill it out as shown in Figure 6-12.

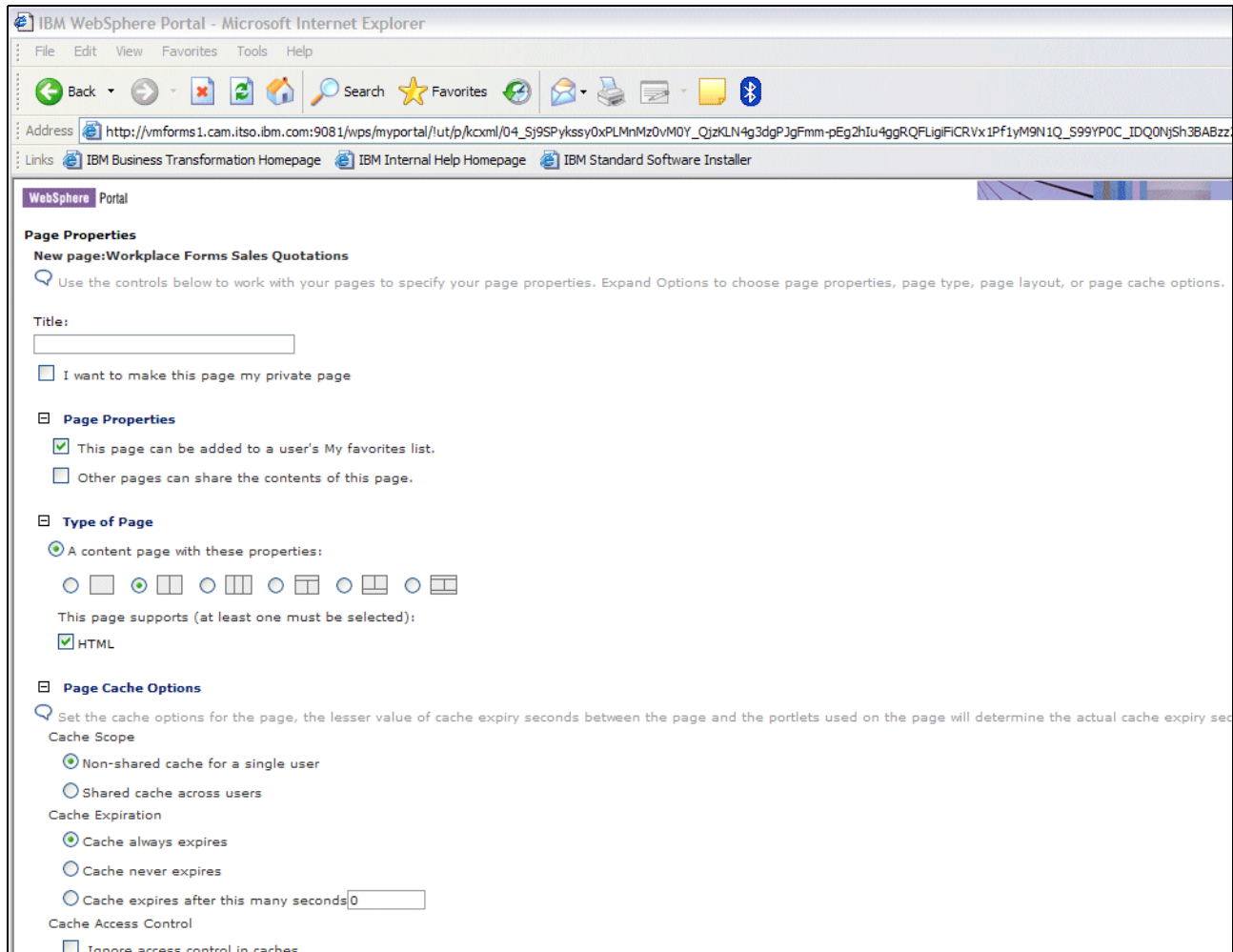


Figure 6-12 Creating a new portal page

2. Click the **OK** button to continue.
3. Click the Edit Page icon (the second icon from the left that looks like a pencil) in order to add a portlet to the page.
4. Click the **Add Portlet** button on the left, select the **WPFRedPaper portlet (Click-to-action) Sender** and then click **Done**.
5. Click the **Add Portlet** button on the right, select the **WPFRedPaper portlet**, and then click **Done**. At this point, you should see the following display (Figure 6-13).

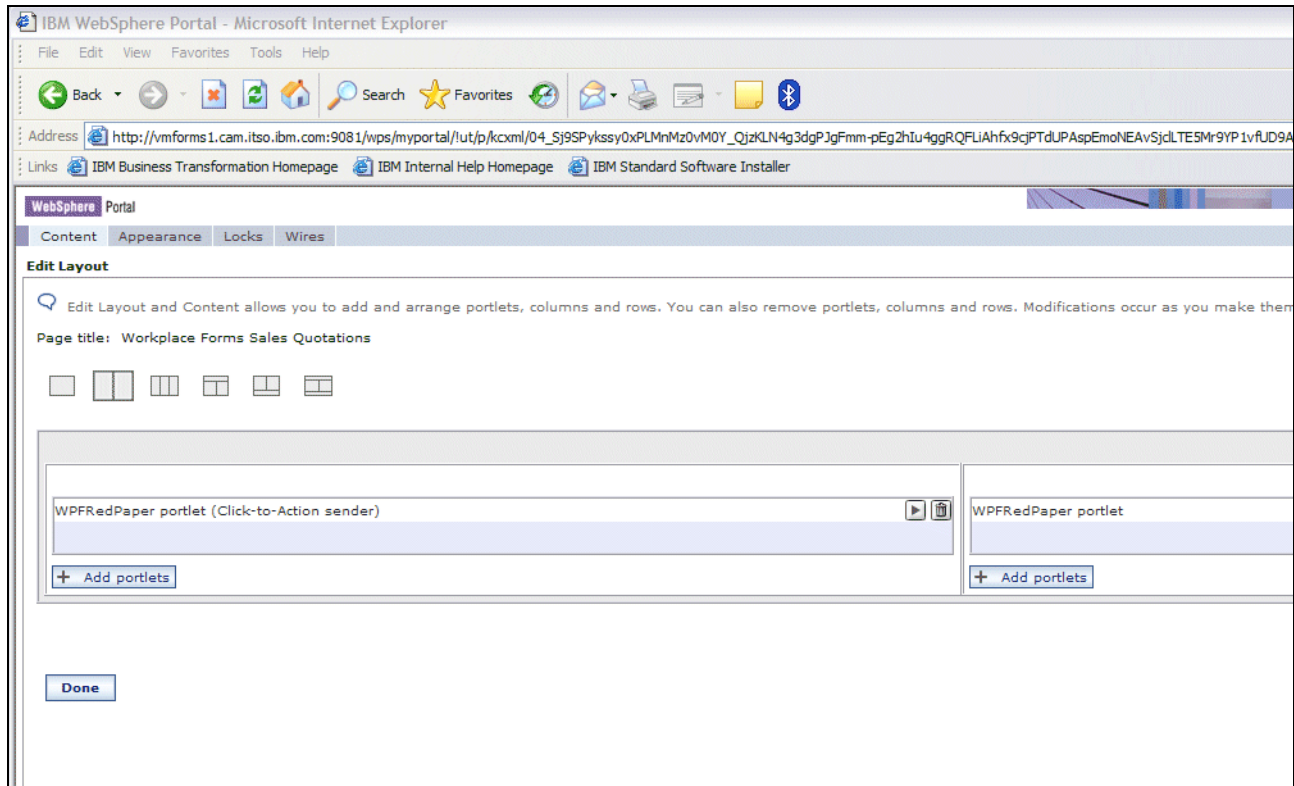


Figure 6-13 Page title: Workplace Forms Sales Quotations

6. Go to the Workplace Forms Sales Quotations Page under My Portal. You will now see the portlets deployed on the page (Figure 6-14).

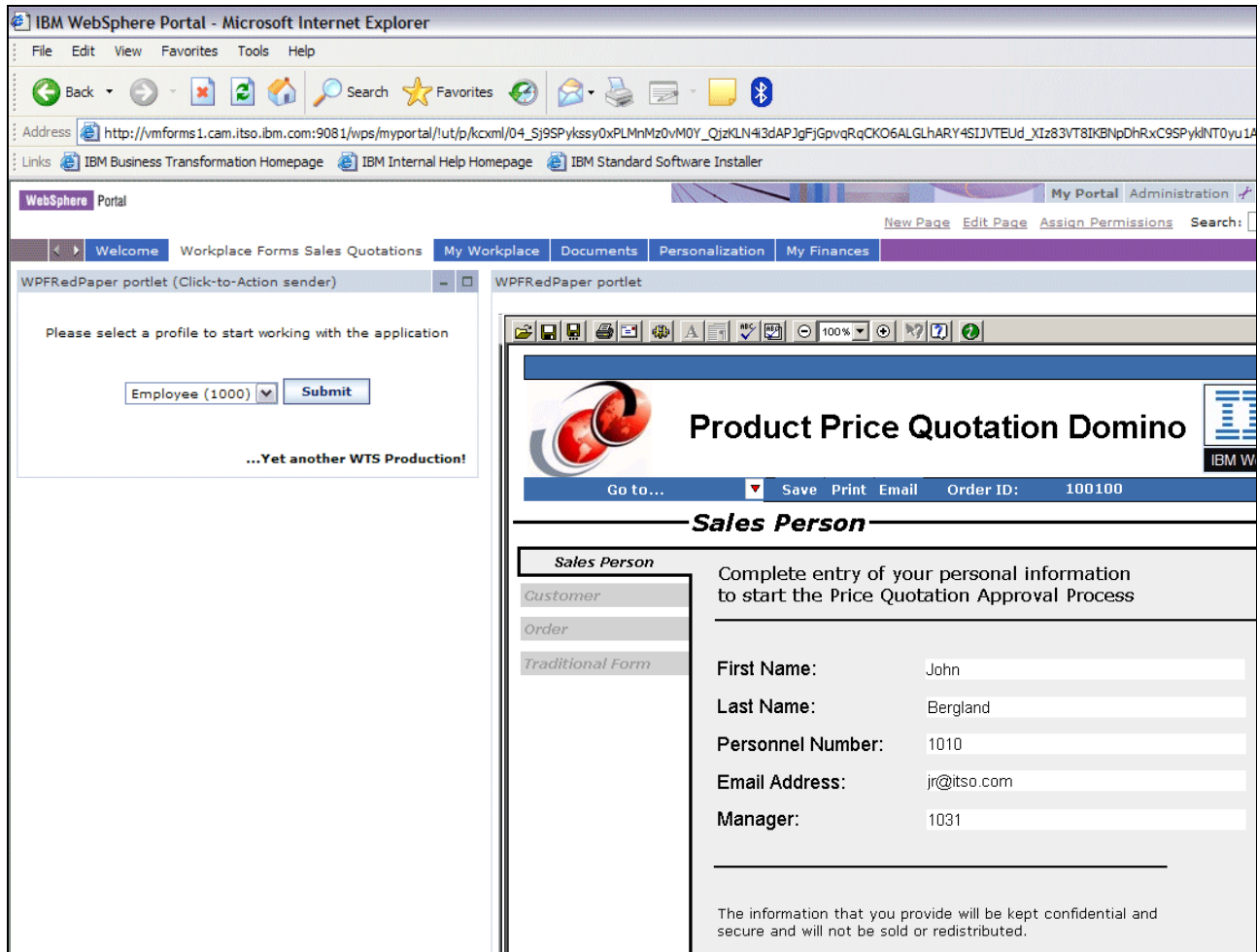


Figure 6-14 Profile Selection and Viewer portlets



Zero Footprint with WebForm Server

When deploying Workplace forms applications, the functionality to enable end users to work with forms is provided through one of two possibilities:

- ▶ Using the IBM Workplace Forms Viewer, which is a feature-rich desktop application used to view, fill, sign, submit, and route eForms. The Viewer is able to function on the desktop or within a browser.
- ▶ Alternatively, using the IBM Workplace Forms Server – Webform Server. You can provide a Zero Footprint solution that allows users to open, complete, and submit forms using a Web browser.

This chapter provides information about the capability of Zero Footprint functionality in the base scenario. (For a complete description of the base scenario, see 4.1, “Introduction to the scenario used throughout this book” on page 54.)

In the process of building the base scenario sample application, when Stage 2 of the development process was implemented (described in Chapter 5, “Building the base scenario: Stage 2” on page 145), the ability to use a Zero Footprint UI was eliminated. Web services for data integration is a Workplace Forms Viewer only alternative.

This chapter describes:

- ▶ The differences in the user experience when using the Workplace Forms Viewer versus a Browser
- ▶ Why and how specific technical implementation details within the base sample application scenario have eliminated the option for a Zero Footprint solution approach
- ▶ The unsupported functions and other considerations in a Zero Footprint approach

7.1 Zero Footprint solution

IBM Workplace Forms Server – Webform Server is a Zero Footprint solution that allows users to open, complete, and submit forms using a Web browser. Webform Server is generally the best solution if your forms contain little logic and you need to distribute them to a large user base, such as the general public.

In a typical scenario, the user goes to a Web site and clicks a link to request a form. Webform Server translates that form into a collection of HTML and Javascript and sends that information to the user's Web browser. The browser displays the translated HTML form to the user, who can then complete the form and submit it back to the server.

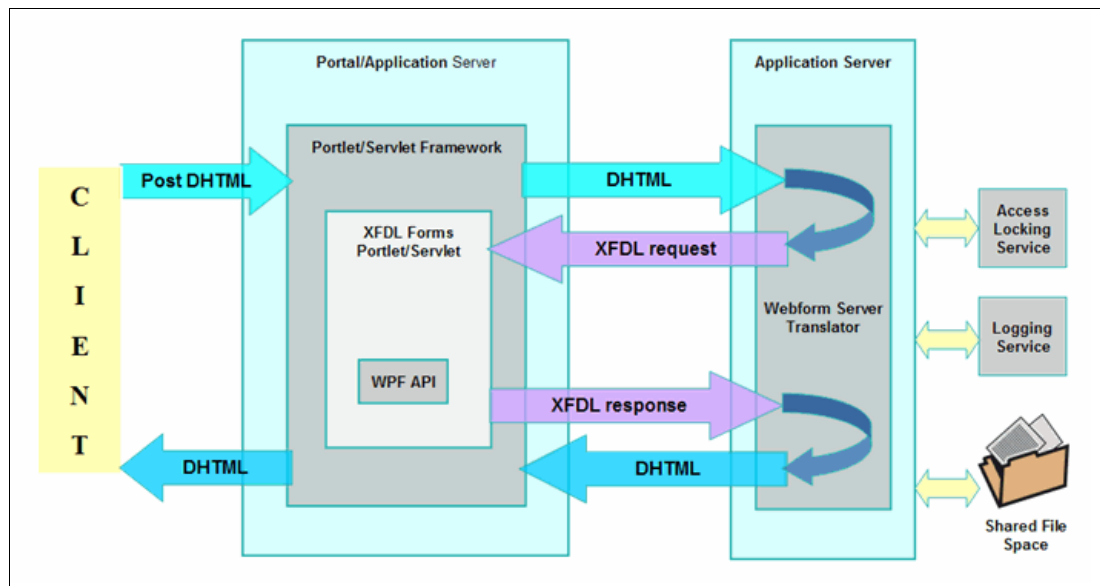


Figure 7-1 Zero Footprint Implementation

Because Webform Server is a Zero Footprint solution, not all of the logic in a form can be processed on the client side. While the form that is sent to the user can perform some simple calculations, there are many things it cannot do without help from the server. Because of this, the user's Web browser will need to make calls back to the server when required. Because of these factors, the user cannot work with the form offline. Instead, the user must remain connected to the Webform Server to complete the form.

However, users can still save forms to their local computer. Furthermore, each form is saved as a single XFDL file, rather than a collection of HTML and javascript. This allows users to save work in progress, or to route forms to other people – a marked advantage over typical HTML forms. Furthermore, Webform Server does not support some features that may be required in your forms, such as digital signatures, Client-side Web services, or the use of Workplace Forms Viewer extensions.

7.2 Form design delta

Throughout Stage 1 development, a browser could be used. A comparison of the two possible user experiences follows.

The Toolbelt at the top of the image and the Go to... wizard graphic are the only two indications that the user experience is with the Viewer (Figure 7-1).

Product Price Quotation

IBM Workplace Forms

Go to... Save Print Email Order ID: Next >>

Sales Person

Sales Person

Customer

Order

Traditional Form

Complete entry of your personal information to start the Price Quotation Approval Process

First Name:

Last Name:

Personnel Number:

Email Address:

Manager:

The information that you provide will be kept confidential and secure and will not be sold or redistributed.

Figure 7-2 The price quotation with the Viewer

With the browser, the functionality available in the Toolbelt is significantly different. Additionally, the Go to... Wizard graphic is subtly different (Figure 7-2).

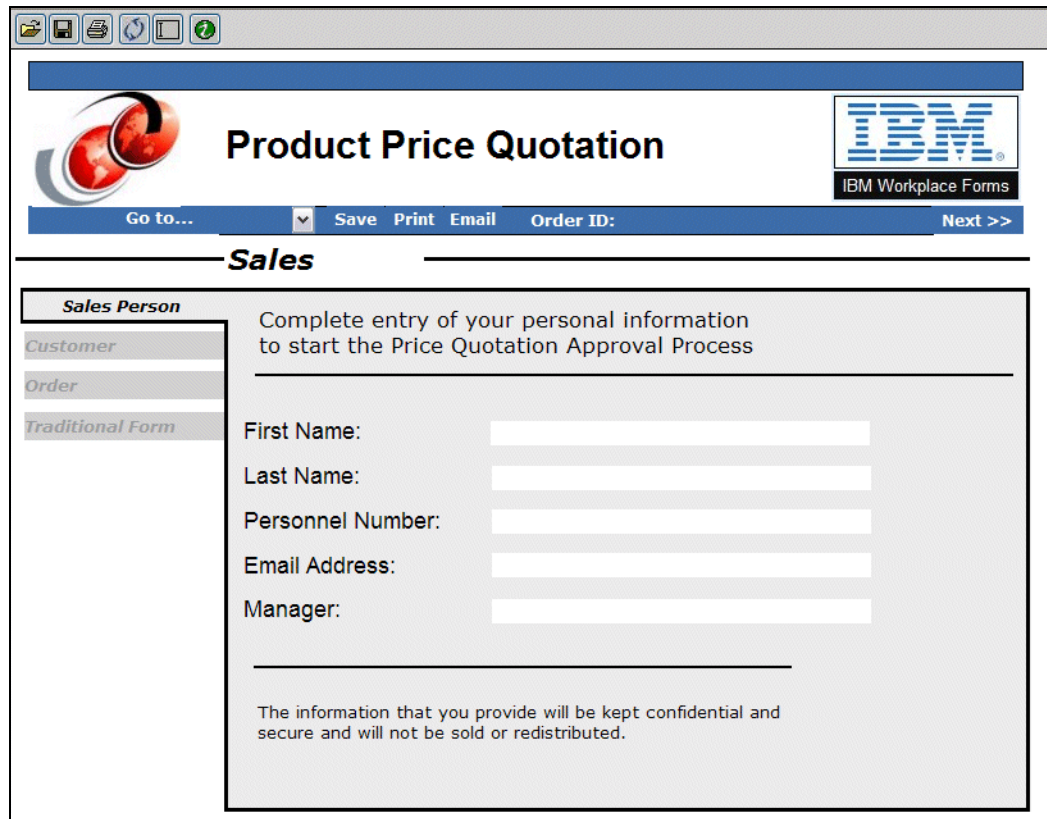


Figure 7-3 The price quotation with the browser

For a description of the icons in the browser toolbelt, see Table 7-1.

Table 7-1 The Browser Toolbelt

Icon	Description	Tag
Open	Opens a new form	open
Save	Save the current form	save
Print	Prints the current form	print
Refresh	Refreshes the current page	refresh
Accessibility	Toggles accessibility mode on and off	accessibility

These and other Toolbelt functions can be changed in `uvf_settings`.

About the `uvf_settings` option

The `uvf_settings` option is declared in either the form global or any page global. As with other options, the global page settings override the global form settings.

The `ufv_settings` option belongs to the XFDL namespace. The option can control one or more features, and follows this syntax:

```

<ufv_settings>
  <feature1>
    <ae>setting1</ae>
    <ae>settingn</ae>
  < /feature1>
</featuren>
  <ae>setting1</ae>
  <ae>settingn</ae>
</featuren>
</ufv_settings>

```

Note: Page settings override the global settings. For example, if you set the `validoverlap` globally, then set the `errorcolor` for page one, then page one will not inherit the `validoverlap` setting. If you want to add page specific settings to your form, you must repeat the form's global settings in the settings for that page.

```

<globalpage sid="global">
  <global sid="global">
    <ufv_settings>
      <menu>
        <save>off</save>
        <open>off</open>
      </menu>
    </ufv_settings>
  </global>
</globalpage>


```

The most significant usability differentiator occurs when embedded form calculations exist. The Viewer contains the calculation engine similar to a spreadsheet application. This capability enables instantaneous feedback when a calculation occurs. The browser does not have this capability. In order to force the form to perform the calculation, a “Refresh Form” has to be performed so that the Webform Server can perform and re-render the results of the calculation(s). The following example from the Stage 1 implementation describes this process.

When calculations exist in the form, the Viewer performs the functions immediately. In the “Widget” line in Figure 7-4, when “Widget” was selected, “Item #”, “# in stock”, and “price” are retrieved and displayed automatically. Additionally, when a value is entered into “quant.,” the “total” and “Grand Total” are immediately calculated and displayed. If a discount is applied, the “total” and “Grand Total” are immediately recalculated and displayed.

Products						
Item	Item #	# in stock	quant.	price	discount	total
Widgets	ITEM_001	121	1	\$199.00	0.1	\$179.10
Select your					discount	\$0.00
Select your					discount	\$0.00
Select your					discount	\$0.00
Select your					discount	\$0.00
Grand Total				\$199.00	0.1	\$179.10

Figure 7-4 Order entry with the Viewer

Using the browser results in a much different user experience. As presented in Figure 7-5, when “Widget” is selected, no associated data is retrieved. Instead, the “Item #”, etc., remain blank. Additionally, when “quantity” is entered, no calculations immediately occur. In this environment the user has to click a button to cause a “Refresh” with the Webform Server. By default, the Refresh button () on the Toolbelt is used to perform this action. Additionally, a Refresh action could be added to other location(s) on the form to improve the user experience.











Products						
Item	Item #	# in stock	quant.	price	discount	total
Widgets 			1		0.1 	\$0.00
Select your pr 					discount 	\$0.00
Select your pr 					discount 	\$0.00
Select your pr 					discount 	\$0.00
Select your pr 					discount 	\$0.00
Grand Total				\$0.00	0	\$0.00

Figure 7-5 Order entry with the browser

For the differences between Webform Server and Viewer, see Table 7-2.

Table 7-2 Differences between Webform Server and Viewer

Functionality	Webform Server	Viewer
Calendar Widget	Not supported	Supported
E-mail	Partial support — Users must save forms to their local computer and e-mail them as attachments via e-mail program	Full support
Form version support	Version 6.0 and later	Versions 4.4 and later
Localization	English only	English and French (Canadian)
Realtime error and format checking	Not supported	Supported
Rich Text Fields	Not supported	Supported
Schema	Server-side only	Client and Server
Screen readers	JAWS only	MSAA compliant
Smartfill	Not supported	Supported
Spellchecking	Not supported	Supported
User modification of display or print preferences	Not supported	Supported
Viewer functions, such as fileOpen, messageBox, setCursor, and so on.	Not supported	Supported
Web services	Not supported	Supported
Zoom capability	Not supported	Supported

Webform Server allows users to sign forms using Clickwrap signatures, and to verify other types of signatures that have already been applied to the form. However, Webform Server does not allow users to sign forms using any other signature types. This means that signature-based security is limited when using Webform Server. While Clickwrap signatures prove acceptance of a document, they do not provide the same level of authentication as digital signatures.

7.3 Web services moves the solution to Viewer only

Stage 2 of the implementation forced the solution to a Viewer only client experience because of the decision to use Web services as the integration technology. Since Web services is Viewer based solution, a Zero Footprint environment was eliminated as an option. This simplified the Forms Design process since there was no longer a need to be concerned with a browser only environment.

The effort to solve this challenge exceeded the scope of the redbook. The following approach could have allowed a Zero Footprint solution.

Client-side Web service calls are not supported in Zero Footprint. In order to enable this capability, the Web services functionality would have to be re-partitioned so that the calls to Web services from the form (client side) are moved to the Web Application (Servlet or Portlet).

The likely solution would include creating a button that submits the form and putting an action on it (`url?action=wscall`). When the Servlet detects the request, it would perform the Web service call within your Servlet/Portlet [Server <—> Service]. When the Web service call is successful, the API would be used to insert the data into the form and return the result.



Integration with IBM DB2 Content Manager

This chapter describes the integration of IBM Workplace Forms with DB2 Content Manager. The automation of forms is often a critical requirement in many Content Manager (CM) installations cross-industry. IBM Workplace Forms can be easily integrated with Content Manager and thereby delivers a high level of synergy.

To remain within the context of the sample scenario application described throughout this redbook, we will consider that the Content Manager could ultimately serve as the final repository for signed forms.

We will provide detailed information on how to integrate Workplace Forms with CM to expand record management capabilities and build complex workflows for development of document centric applications. When integrating IBM Workplace Forms with CM, you create security-rich front-end transaction records with a streamlined, secure content management infrastructure.

Note: The code used for building this sample scenario application is available for download. For specific information about how to download the sample code, please refer to Appendix A, “Additional material” on page 333.

Note: All specific examples shown and used when building the sample scenario application are based on the codebase for IBM Workplace Forms Release 2.5.

Note: The Content Manager Demo Platform referred to within this chapter is available as a VMWare or Ghost Image. It provides a range of products and assets related to DB2 Content Manager, Portal, Workplace Forms, and other products. It is intended for use in pre-sales/sales and proof-of-concepts.

The Content Manager Demo Platform (CM Demo Platform) can be used internally, or provided to partners, and contains a working example of a simple point-connector-based integration of Workplace Forms with CM. In this reference implementation, forms are made completely self-describing with regard to how they are stored into Content Manager (item type, item attributes, and optionally, libserver, etc.). The CM Demo Platform reference integration is genericized and is intended to enable one to integrate any form with Content Manager.

8.1 Overview

Workplace Forms can be stored in Content Manager as items with specific attribute arrays that in turn can be used to store metadata describing the form. Content Manager can serve as a central repository with different purposes and functionalities in regard to eForms:

- ▶ Document management
- ▶ Workflow modelling and processing
- ▶ Data search and retrieval
- ▶ Archive and storage

Figure 8-1 gives you a functional overview of the three tiers involved when integrating DB2 Content Manager.

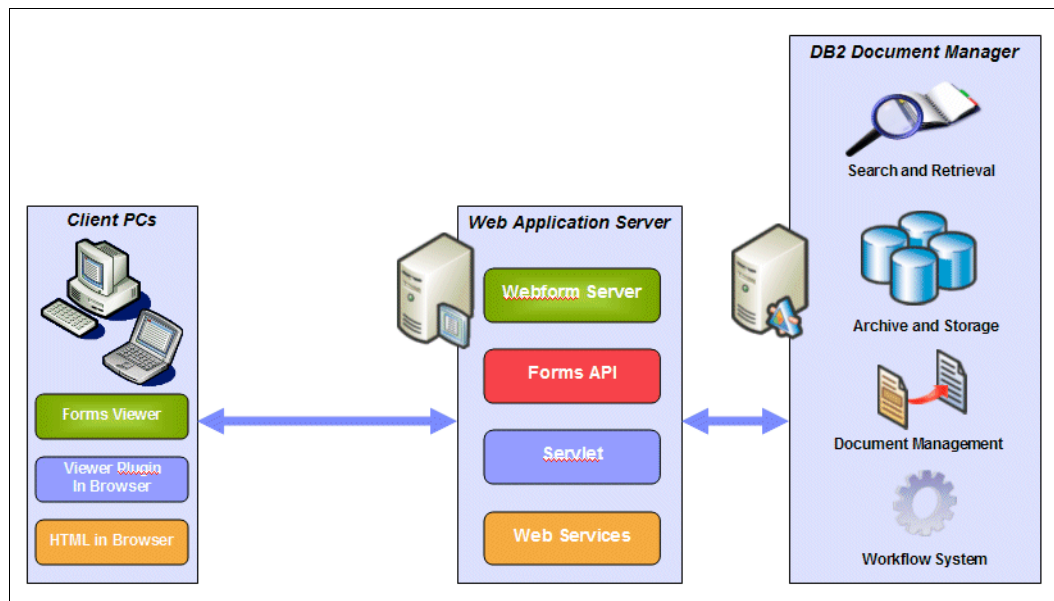


Figure 8-1 3-Tier overview of DB2 Content Manager integration

The interaction of Workplace Forms with DB2 Content Manager usually follows a specific sequence of steps, as shown in Figure 7-2.

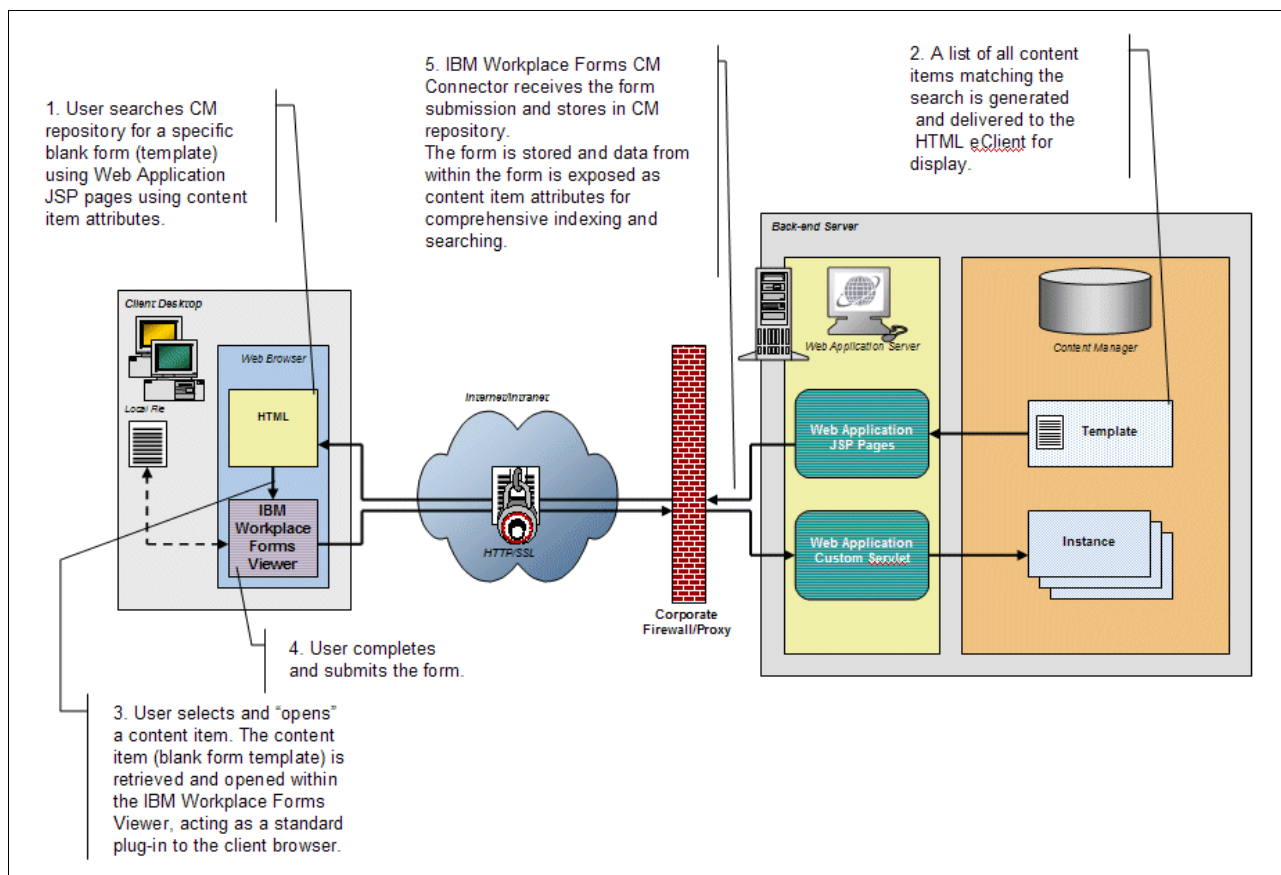


Figure 8-2 Typical sequence of steps for Workplace Forms and CM interaction

The described CM connector is in fact a custom servlet that performs the interaction with the DB2 CM instance. In the following section we used a sample implementation of a CM Submission servlet that is part of the IBM DB2 Content Manager Demo platform. However, the basic concept and functionality of the servlet described here is applicable to any type of Web application that performs this integration.

The described CM connector is in fact a custom servlet that performs the interaction with the DB2 CM instance. In the following section, we have used a sample implementation of a CM Submission servlet that is part of the IBM DB2 Content Manager Demo platform. However, the basic concept and functionality of the servlet described here is applicable to any type of Web application that performs this integration.

The typical sequence of steps for the CM integration is as follows:

1. The user searches the CM repository for a specific blank form (template) using Web Application JSP pages using content item attributes.
2. A list of all content items matching the search is generated and delivered to the HTML eClient for display.
3. The user selects and “opens” a content item. The content item (blank form template) is retrieved and opened within the IBM Workplace Forms Viewer, acting as a standard plug-in to the client browser.
4. The user completes and submits the form.
5. IBM Workplace Forms CM Connector receives the form submission and stores in CM repository. The form is stored and data from within the form is exposed as content item attributes for comprehensive indexing and searching.

For the scenario we describe in this redbook, the Content Manager does not serve as a repository for form templates. We extended our example described in the previous chapters by both submitting completed forms to the Content Manager using a Submit button, and by submitting only approved forms to the Content Manager using servlet to servlet communication as described in 8.3.3, “Servlet to servlet communication” on page 266.

8.2 Basic design of Content Manager integration

A basic connector to the Content Manager as described here makes it simple for developers to:

- ▶ Store forms as items in Content Manager (with attribute values set based on form data).
- ▶ Retrieve form items from Content Manager.
- ▶ Update existing form items within Content Manager.

Figure 8-3 illustrates the basic design of the Forms-CM integration.

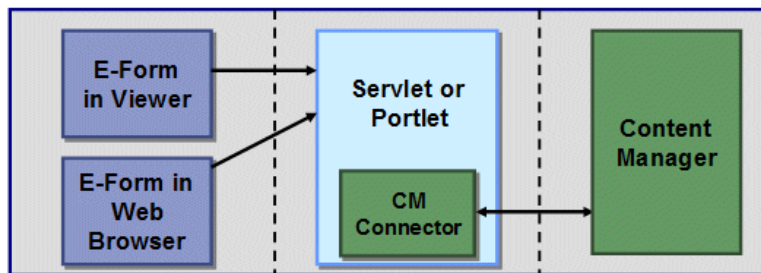


Figure 8-3 Basic Design of Workplace Forms and CM Integration

On the left is the Workplace Forms Viewer invoked either standalone or from within a Web browser. The form is filled in as needed and then submitted via a Submit button. This causes the servlet (middle section) to be executed, which will process the form, extracting the Item Type, Attributes, and other form data, and will store the entire, filled-in form into CM. If the Form Viewer was invoked to display an existing Form saved to CM using this servlet, then the stored form will be updated.

The ContentManagerMetaData instance contains all of the information needed to describe this form’s integration to Content Manager – thus making the form document completely self-describing with regards to how it is stored or represented in Content Manager.

We will add the Instance Data Model required and bind certain fields in the Form to this Instance Data Model. The current servlet provided on the CM Demo Platform has some very specific requirements for integration. Specifically, two entities must exist:

1. An invisible field is added to the form for the CM PID (generated when a Form is stored in CM) and must be called "PID".
2. Example 8-1 shows the syntax required by the servlet (shown here in the Form's XML format with my comments added). This is an example of what must be defined in the Data Instance Model.

Example 8-1 Data Instance Model required by the CMSubmission Servlet

```
<!--the ID MUST BE CMAttributes -->
<xforms:instance xmlns=http://www.PureEdge.com/XFDL/Custom id="CMAttributes">
<!--the ContentManagerMetaData tag is required -->
  <ContentManagerMetaData>
    <!--insert the name of the Item Type in CM -->
    <ItemType>MyForm</ItemType>
    <!--insert the number of Attributes that will be used -->
    <NumberItemAttributes>4</NumberItemAttributes>
    <!-- repeating tags occur now as necessary for the number of Attributes -->
    <!--insert the Attribute Name here -->
    <ItemAttributeName0>MyFirstAttr</ItemAttributeName0>
    <!--this will contain the Attribute Value when the form is completed -->
    <ItemAttributeValue0></ItemAttributeValue0>
    <ItemAttributeName1>MySecondAttr</ItemAttributeName1>
    <ItemAttributeValue1>External Wire Transfer Request</ItemAttributeValue1>
    <ItemAttributeName2>MyThirdAttr</ItemAttributeName2>
    <ItemAttributeValue2></ItemAttributeValue2>
    <ItemAttributeName3>MyFourthAttr</ItemAttributeName3>
    <ItemAttributeValue3></ItemAttributeValue3>
    <!-- the tags above must follow the sequence up to the required number -->
    <!--PID is a required entry and will be filled in when the form is completed -->
    <PID></PID>
    <!--CMUserName will default to icmadmin if not provided here -->
    <CMUserName>icmadmin</CMUserName>
    <!--CMUserPassword will default to password if not provided here -->
    <CMPassword>password</CMPassword>
    <!--CMLibServerName will default to icmnlbdb if not provided here -->
    <CMLibServerName>icmnlbdb</CMLibServerName>
    <!--CMSchemaName will default to SCHEMA=ICMADMIN if not provided here -->
    <CMSchemaName>SCHEMA=ICMADMINicmadmin</CMSchemaName>
    <!--ItemMimeType will default to application/vnd.xfdl if not provided here -->
    <ItemMimeType>application/vnd.xfdl</ItemMimeType>
  </ContentManagerMetaData>
</xforms:instance>
```

The preceding example shows the syntax required by the servlet. The tag values (that is, the information between the tag pairs) are just examples in this case.

In the next sections we describe how to enhance our existing Sales Quote Application to store the forms along with some metadata attributes in DB2 Content Manager.

8.3 Integrating the sales quote sample with DB2 Content Manager

To integrate our Sales Quote sample application with IBM DB2 Content Manager, we have to perform essentially two tasks:

1. Create attributes and item types in DB2 Content Manager.
2. Add the CM Integration in the form.
3. Enhance our submission servlet to post the form to the Content Manager submission servlet.

The necessary steps are described in the next two sections.

8.3.1 Create attributes and item types

To create the Content Manager item type and attributes you may follow these steps:

1. Start the IBM Content Manager System Administration Client by selecting **Start** → **Programs** → **IBM DB2 Content Manager Enterprise Edition** → **System Administration client**.
 - a. Login with the respective UID and password.
 - b. You will see the System Administration Client as shown in Figure 8-4.

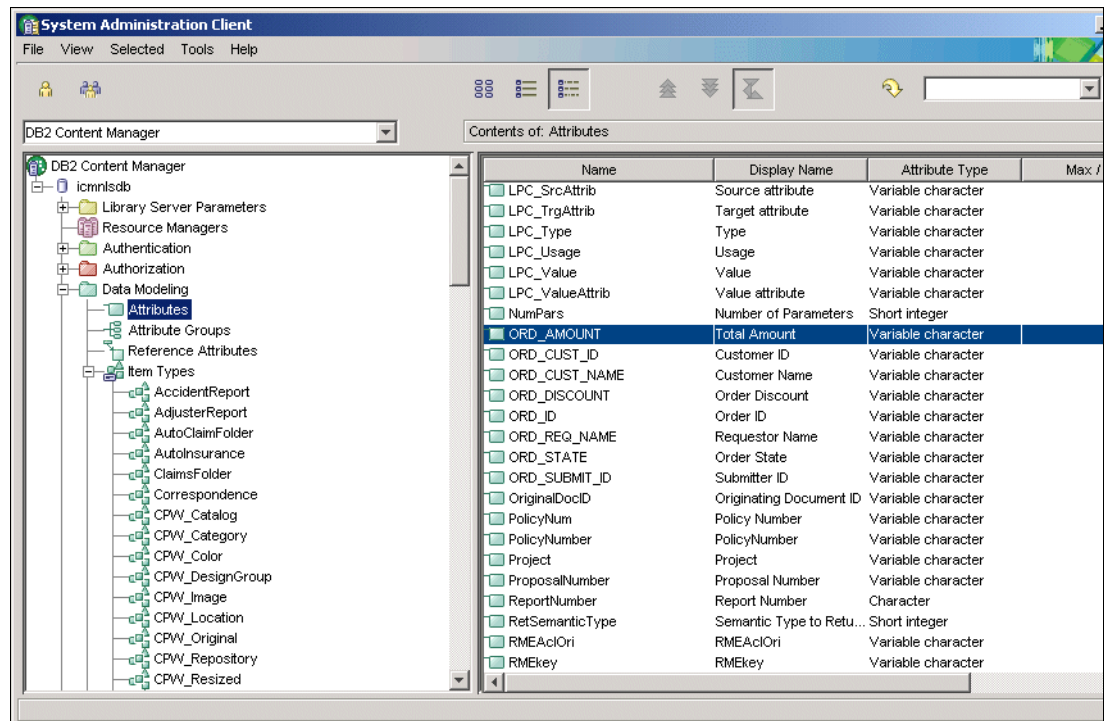
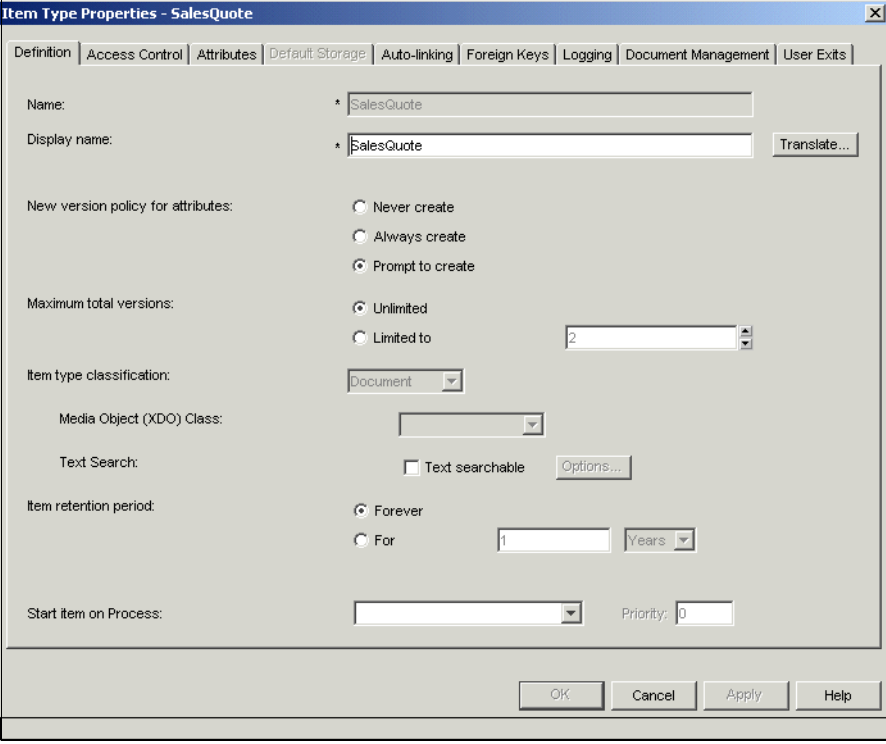


Figure 8-4 DB2 Content Manager System Administration Client

2. Create a new item type for the form.
 - a. Expand the **Data Modeling** tree in the left-hand pane and right-click **Item Types**.
 - b. Select **New**.
 - c. You will see the **Item Type Properties** dialog as seen in Figure 8-5.



The image shows the 'Item Type Properties - SalesQuote' dialog box. It has a tabbed interface with the following tabs: Definition, Access Control, Attributes, Default Storage, Auto-linking, Foreign Keys, Logging, Document Management, and User Exits. The 'Definition' tab is active. The dialog contains the following fields and controls:

- Name:** * SalesQuote
- Display name:** * SalesQuote (with a 'Translate...' button)
- New version policy for attributes:** Radio buttons for 'Never create', 'Always create', and 'Prompt to create' (selected).
- Maximum total versions:** Radio buttons for 'Unlimited' (selected) and 'Limited to' (with a value of 2).
- Item type classification:** Document (dropdown)
- Media Object (XDO) Class:** (empty dropdown)
- Text Search:** ☐ Text searchable (with an 'Options...' button)
- Item retention period:** Radio buttons for 'Forever' (selected) and 'For' (with a value of 1 and a 'Years' dropdown).
- Start item on Process:** (empty dropdown)
- Priority:** 0

At the bottom are buttons for OK, Cancel, Apply, and Help.

Figure 8-5 Item Type Properties dialog

3. On the **Definition** tab:
 - a. Enter **SalesQuote** for the Name and Display Name.
 - b. Select **Prompt to create** as the **New version policy attribute**.

4. On the **ACL** tab, as shown in Figure 8-6:
 - a. Select **PublicReadACL** for Item type access control list (ACL).

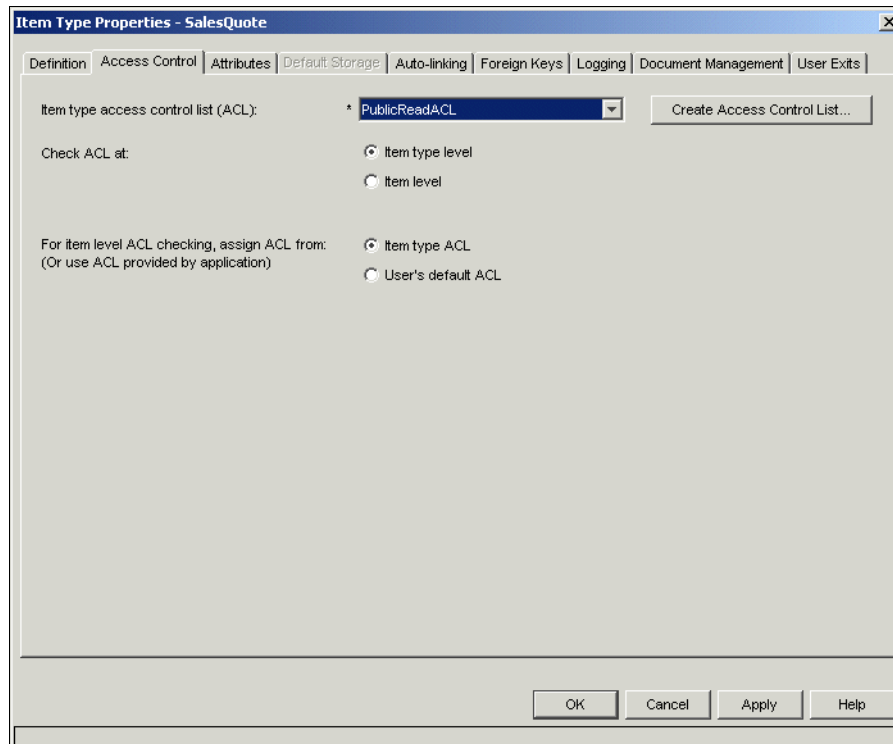



Figure 8-6 Access Control tab of Item Type Properties dialog

5. On the **Attributes** tab:
 - a. Click the **New Attribute** button 
 - b. The New Attribute dialog will open as seen in Figure 8-7.

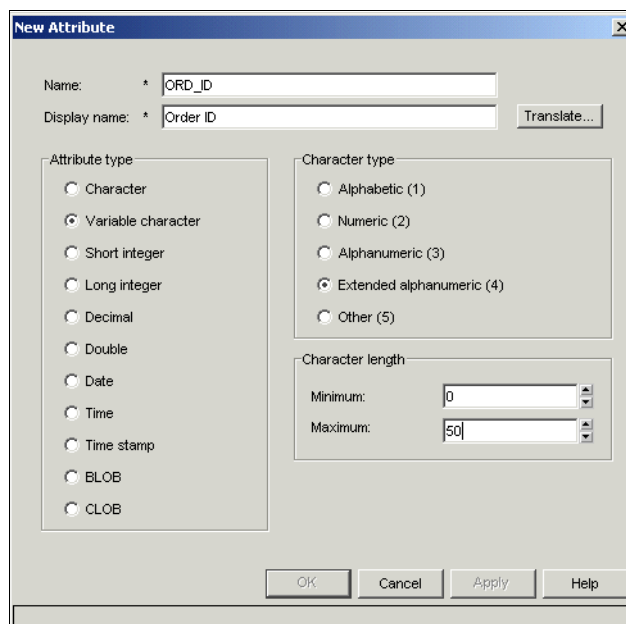


Figure 8-7 New Attribute Dialog

- c. Enter the following values:
 - i. Name = ORD_ID
 - ii. Display name = Order ID
 - iii. Attribute type = Variable Character
 - iv. Character type = Extended alphanumeric
 - v. Length Maximum = 50
 - vi. Click **Apply**.
- d. Repeat this step for add these attributes with the following Names and Display Names:
 - ORD_SUBMIT_ID - Requestor ID
 - ORD_REQ_NAME - Requestor Name
 - ORD_CUST_ID - Customer ID
 - ORD_CUST_NAME - Customer Name
 - ORD_AMOUNT - Total Amount
 - ORD_DISCOUNT - Discount
 - ORD_STATE - Order State
- e. Now select these four attributes (use the Ctrl key to have multiple selections) and click the **Add >** button as seen in Figure 8-8.

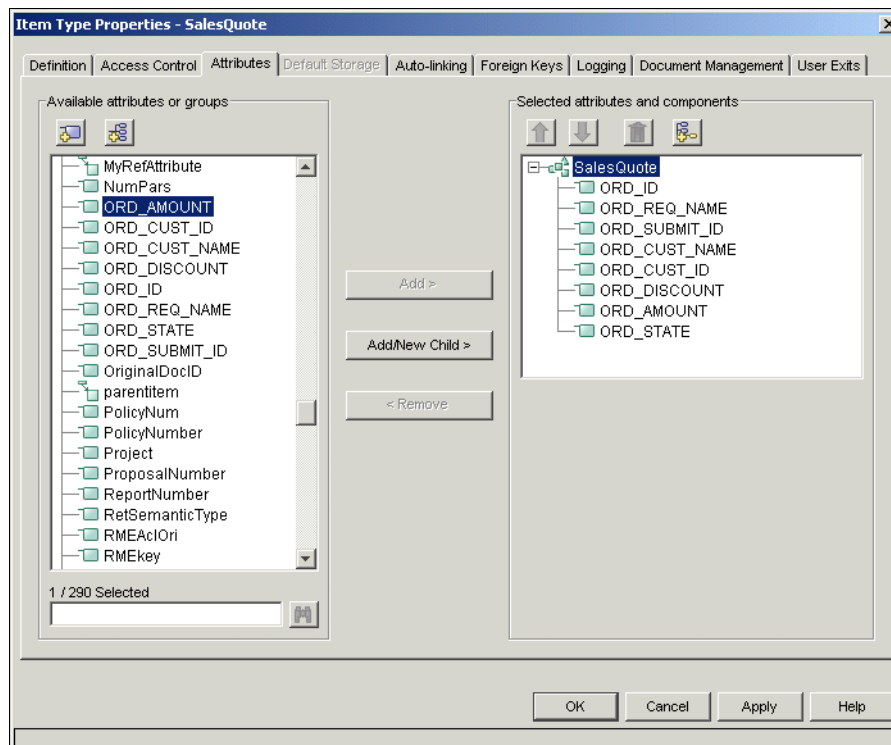


Figure 8-8 Attributes tab of the Item Types dialog

6. On the **Document Management** tab, as seen in Figure 8-9, make the following selections.

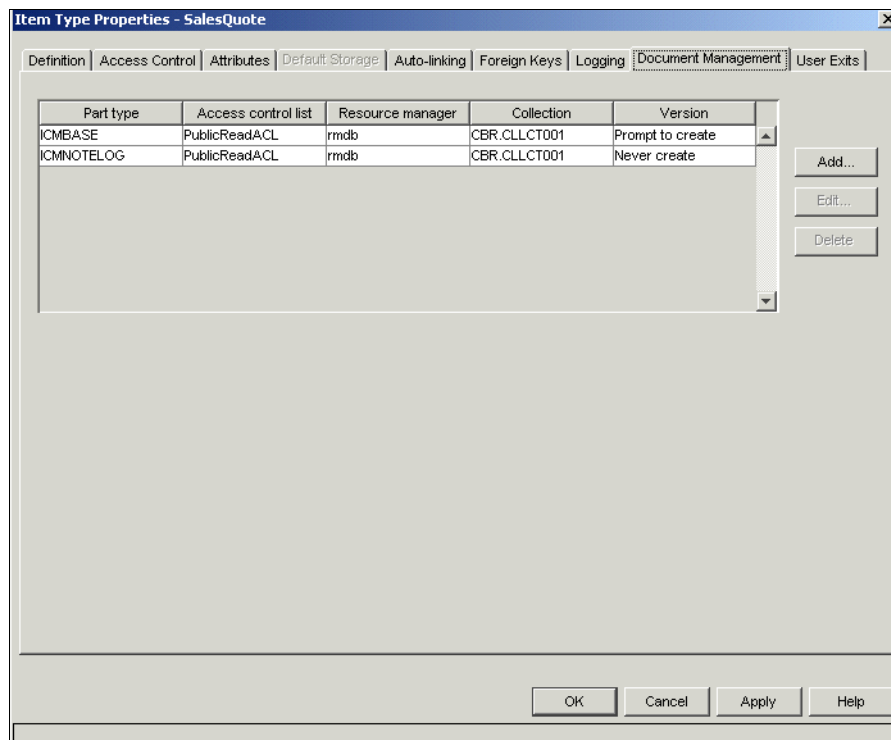


Figure 8-9 Document Management tab of the Item Types Dialog

- a. Click **Add**:
 - i. The Define Document Management Relations dialog opens as seen in Figure 8-10.
- b. Select **ICMBASE** for the **Part Type** field.
- c. Select **PublicReadACL** for the Access control list field.
- d. Click **OK**.

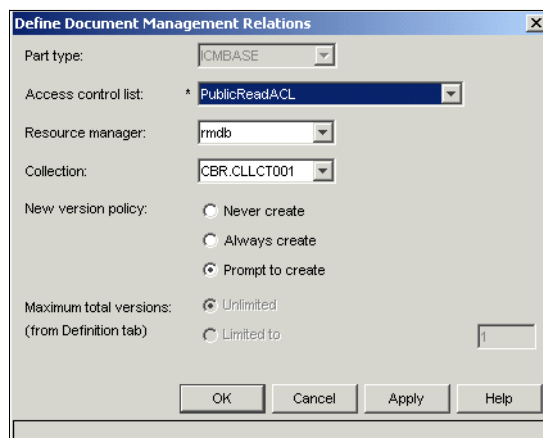


Figure 8-10 Define Document Management Relations dialog


- e. Click **Add** again.
 - f. Select **ICMNOTELOG** for the **Part Type** field.
 - g. Click **OK** and click **OK** again. The CM Item Type **SalesQuote** should appear in the list of Item Types (right-hand pane).
7. Add the **XFDL MIME Type** to Content Manager.
 - a. Click on the **MIME Types** entry in the right-hand pane Tree. Sort the list Z-A. You should see **XFDL** near the top.
 - b. Double-click on **XFDL** and you will see the properties required:
 - i. Name = XFDL
 - i. Display name = XFDL
 - i. MIME type = application/vnd/xfdl
 - i. Suffixes = .xfd
 - c. Click **Cancel**.
 8. Add XFDL MIME type processing for the Client for Windows.
 - a. Start the IBM Content Manager Client for Windows by selecting **Start → Programs → IBM DB2 Content Manager Enterprise Edition → Client for Windows**.
 - a. Login with the respective UID and password.
 - b. Select **Options - Preferences... - Helper Applications**. Scroll to the bottom of the list to find XFDL. This information comes from the Library Server and doesn't need to be changed. The Client for Windows relies on Windows to launch the correct application (in this case the Workplace Forms Viewer).
 9. Add XFDL MIME type processing for the eClient.
 - a. View the file **IDMAdminDefaults.properties** located at **C:\IBM\db2cmv8\CMClient**.
 - b. Search for **xfd** and you will see that a line was added for the XFDL MIME type. That is, application/vnd.xfdl=launch was added to the MIME Type action list of MIME types.

8.3.2 Add the CM integration in the form

To be able to store our form in DB2 Content Manager, we have to add some CM specific items to the form. Open the form in Workplace Forms Designer and perform the following steps to define a data instance model for Content Manager:

1. Create a hidden Field. This field will contain the CM PID for when the Form is stored in CM.
 - a. Click the Field Icon and position the field on the traditional form page below the hidden fields box.
 - b. Shorten it by dragging the right edge to the left so that it does not extend beyond the right side of the box.
 - c. Edit the Properties and change the name to **PID**.
 - d. Click the **Appearance** Tab. Check both of the "invisible" conditions.
 - e. Click **OK**.

2. Create the **Submit** button.

- a. Click the Button Icon. Position the button in the toolbar of the traditional form page, towards the left edge of the box.
- b. Edit the Properties and click **Add Formula** for the Text field.
- c. Select **determined by a decision (If/Then/Else)** in the “is” field
- d. In the **If** field, select **value of...**. Click the  button and then click on the PID field. Select **==** (equal to) for the conditional. Leave the value blank! Enter **Store a new Quote to CM** for the **Then** value. Enter **Update an existing Quote in CM** for the “Else” value as shown in Figure 8-11.

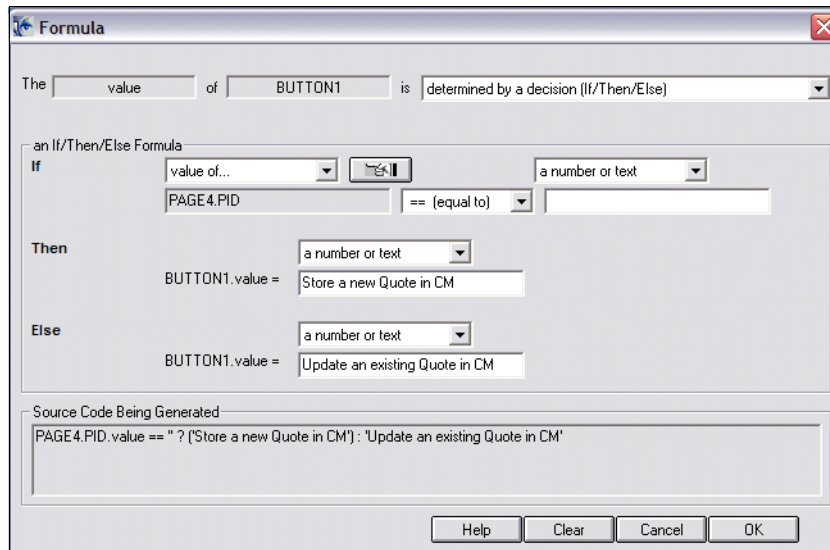



Figure 8-11 Label formula for the CM Submit button

- e. Click **OK**.
- f. Select **Submit Then Cancel** for the **Perform This Action** field.
- g. Then click **Details...**
- h. Click the **Formula** button.
- i. Select **determined by a decision (If/Then/Else)** in the **is** field
- j. In the **If** field, select **value of...**. Click the  button and then click on the **PID** field. Select **==** (equal to) for the conditional. Leave the value blank!

For the **Then** value, enter:

<http://cmdemo.svl.ibm.com/formdemo/CMSubmissionServlet?action=store>

For the **Else** value, enter:

<http://cmdemo.svl.ibm.com/formdemo/CMSubmissionServlet?action=update>

The formula dialog is shown in Figure 8-12.

The of is determined by a decision (If/Then/Else)

an If/Then/Else Formula

If

Then

Else

Source Code Being Generated

```
PAGE4.PID.value == " ? ('http://cmdemo.svl.ibm.com/formdemo/CMSubmissionServlet?action=store') :
'http://cmdemo.svl.ibm.com/formdemo/CMSubmissionServlet?action=update'
```

Help Clear Cancel OK

Figure 8-12 Formula dialog for the CM Submit button URL

3. Modify the scope of the Signature. Since the PID field and submit button will be modified after the form has been signed, these fields must be omitted from the scope of the signature.
 - a. Since we are using a keep filter in items for our Signature, there are no further actions required. The field and the button should not be on the keep list.
4. Create the Data Instance and then bind it to fields in the form.
 - a. Select **Tools** → **XML Data Model** → **Create/Edit Manually**
 - b. Enter **CMMetaData** in the ID field, and select custom in the prefix field. Now click the **Add >>** button. This immediately pops up a window where the XML will be edited for the Instance Data Model.
 - c. Enter the data shown in Example 8-2. We selected eight attributes to be stored with the form in DB2 Content Manager.
 - d. Click **OK** and click **OK** again to save the changes.

Example 8-2 Code listing of data attributes

```
<xforms:instance xmlns="" id="CMMetaData">
  <ContentManagerMetaData>
    <PID></PID>
    <ItemType>SalesQuote</ItemType>
    <NumberItemAttributes>8</NumberItemAttributes>
    <ItemAttributeName0>ORD_ID</ItemAttributeName0>
    <ItemAttributeValue0></ItemAttributeValue0>
    <ItemAttributeName1>ORD_CUST_ID</ItemAttributeName1>
    <ItemAttributeValue1></ItemAttributeValue1>
    <ItemAttributeName2>ORD_AMOUNT</ItemAttributeName2>
    <ItemAttributeValue2></ItemAttributeValue2>
    <ItemAttributeName3>ORD_DISCOUNT</ItemAttributeName3>
    <ItemAttributeValue3></ItemAttributeValue3>
    <ItemAttributeName4>ORD_STATE</ItemAttributeName4>
    <ItemAttributeValue4></ItemAttributeValue4>
    <ItemAttributeName5>ORD_SUBMIT_ID</ItemAttributeName5>
    <ItemAttributeValue5></ItemAttributeValue5>
    <ItemAttributeName6>ORD_REQ_NAME</ItemAttributeName6>
    <ItemAttributeValue6></ItemAttributeValue6>
    <ItemAttributeName7>ORD_CUST_NAME</ItemAttributeName7>
    <ItemAttributeValue7></ItemAttributeValue7>
  </ContentManagerMetaData>
</xforms:instance>
```

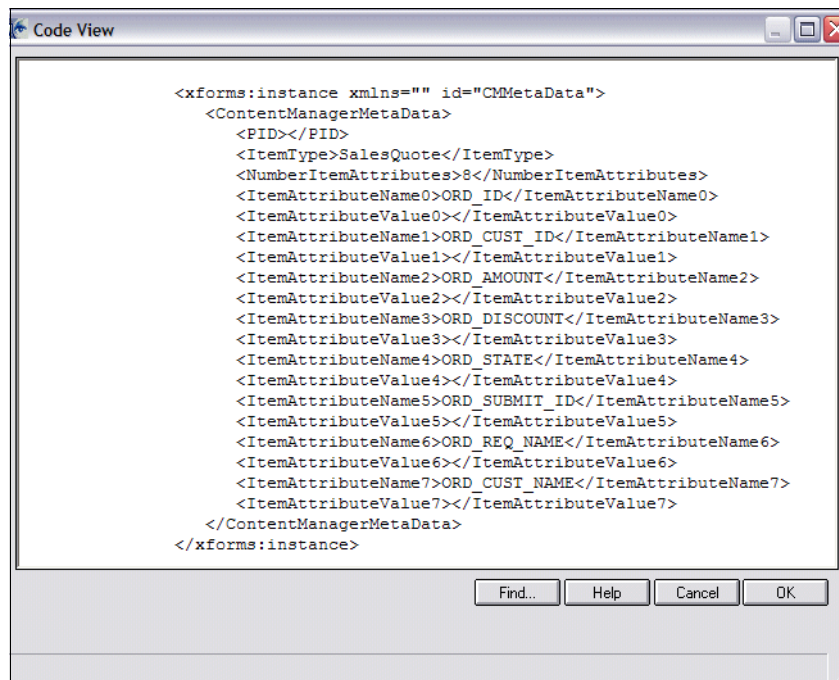


Figure 8-13 Code view of the CMMetaData instance

5. Next we have to bind the fields in to the data instance. Since the fields we selected for CM are already bound to other data instances, we will bind the CMMetaData attributes to these instance attributes. We will show this example for the Order ID attribute:

- a. Select **Tools** → **XML Data Model** → **Create/Edit Manually**.

- b. On the Bindings tab, select **FormOrderData** in the Data Instances list (Figure 8-14).

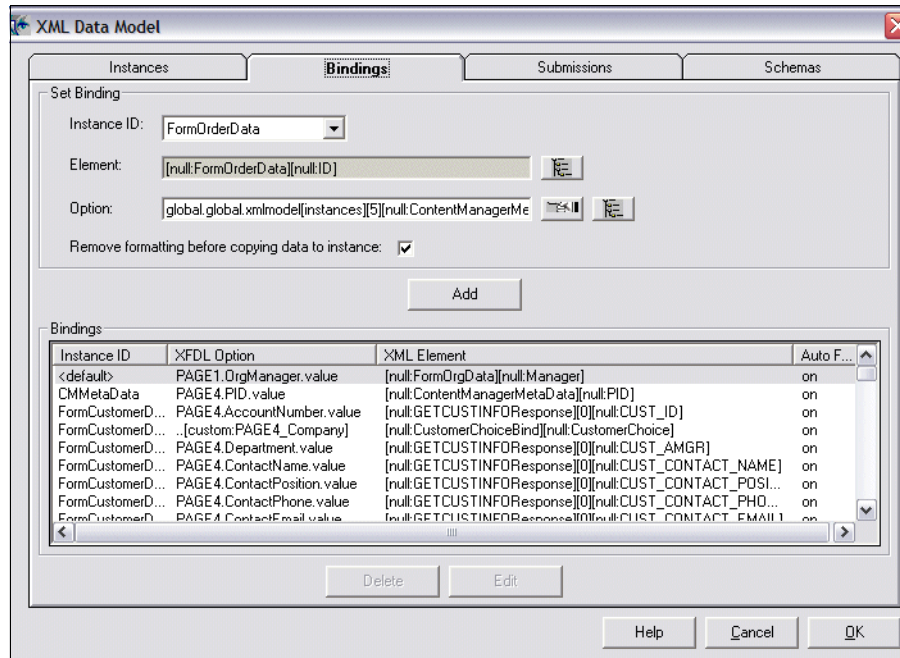


Figure 8-14 Binding tag of the XML data model dialog

- c. Click the Element Tree button for the **Element** field and select the **ID** element. Click **Done**.
- d. Click the Element Tree button for the **Option** field and select the **ItemAttributeValue0** element as shown in Figure 8-15. Click **Done**.

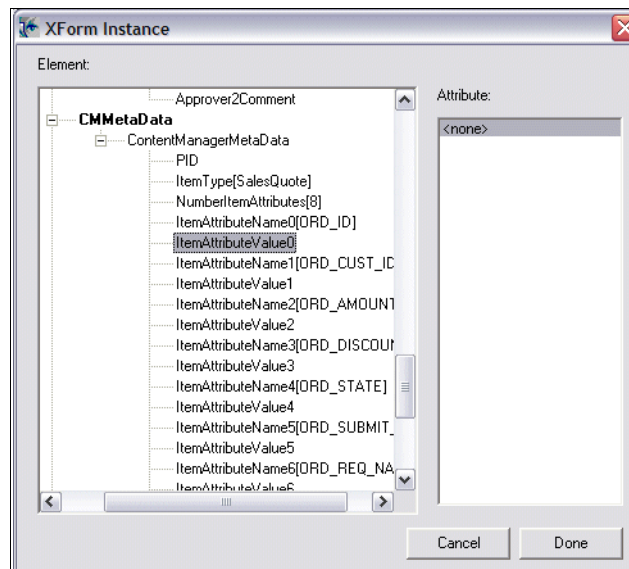


Figure 8-15 XForm Instance dialog for binding

- e. Click **Add**. You must click **Add** or the mapping will not take effect.

- f. Bind the other seven attributes (Instance:Attribute) and the PID attribute:
 - i. FormOrderData:CustomerID to ORD_CUST_ID
 - ii. FormOrderData:Amount to ORD_AMOUNT
 - iii. FormOrderData:Discount to ORD_DISCOUNT
 - iv. FormMetaData:State to ORD_STATE
 - v. FormOrgData:ID to ORD_SUBMIT_ID
 - vi. FormOrgData:LastName to ORD_REQ_NAME
 - vii. FormCustomerData:Cust_Name to ORD_CUST_NAME
 - viii.CMMetaData:PID to Field PID on Page 4
- g. Click **OK** and click **OK** again to save your changes.

Example 8-3 shows the bindings you just created in the XML source code view of the form.

Example 8-3 CMMetaData bindings in the XML source code view

```
<bindings>
  <bind>
    <instanceid>FormOrderData</instanceid>
    <ref>[null:FormOrderData][null:ID]</ref>
    <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
      [null:ItemAttributeValue0]</boundoption>
  </bind>
  <bind>
    <instanceid>FormOrderData</instanceid>
    <ref>[null:FormOrderData][null:CustomerID]</ref>
    <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
      [null:ItemAttributeValue1]</boundoption>
  </bind>
  <bind>
    <instanceid>FormOrderData</instanceid>
    <ref>[null:FormOrderData][null:Amount]</ref>
    <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
      [null:ItemAttributeValue2]</boundoption>
  </bind>
  <bind>
    <instanceid>FormOrderData</instanceid>
    <ref>[null:FormOrderData][null:Discount]</ref>
    <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
      [null:ItemAttributeValue3]</boundoption>
  </bind>
  <bind>
    <instanceid>FormMetaData</instanceid>
    <ref>[null:FormMetaData][null:State]</ref>
    <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
      [null:ItemAttributeValue4]</boundoption>
  </bind>
  <bind>
    <instanceid>FormOrgData</instanceid>
    <ref>[null:FormOrgData][null:ID]</ref>
    <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
      [null:ItemAttributeValue5]</boundoption>
  </bind>
  <bind>
    <instanceid>FormOrgData</instanceid>
    <ref>[null:FormOrgData][null:LastName]</ref>
    <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
```

```

[null:ItemAttributeValue6]/>boundoption>
</bind>
<bind>
  <instanceid>FormCustomerData</instanceid>
  <ref>[null:GETCUSTINFOResponse][0][null:CUST_NAME]</ref>
  <boundoption>global.global.xmlmodel[instances][5][null:ContentManagerMetaData]
    [null:ItemAttributeValue7]/>boundoption>
</bind>
<bind>
  <instanceid>CMMetaData</instanceid>
  <ref>[null:ContentManagerMetaData][null:PID]</ref>
  <boundoption>PAGE4.PID.value</boundoption>
</bind>
</bindings>

```

Now we have created the form items, data instance, and bindings for the DB2 Content Manager Integration. We will test the integration in the following section.

8.3.3 Servlet to servlet communication

To integrate the existing submission servlet with the supposed Content Manager environment we have to enable the SubmissionServlet installed on the WebSphere Application Server (WAS) 5.1 server to submit a received form to Content Manager. The new integration scenario is shown in Figure 8-16.

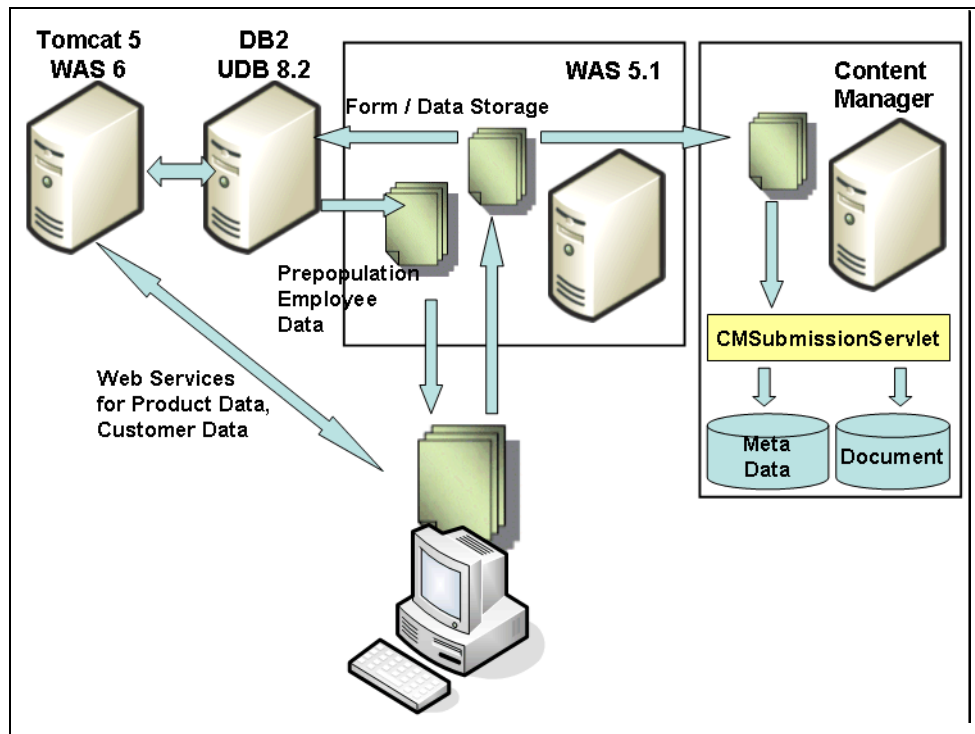


Figure 8-16 Integration scenario with form submission to Content Manager

We will add the following code to the WPFormsRedpaper servlet project:

- Create a new property CMSubmissionUrl to order.properties file. Here we will store the URL to the Content Manager servlet able to receive the submitted form. If this property is missing or empty, the SubmissionServlet should work as in Stage 2 (assuming there is no CM integration).

- Create a new attribute CMSubmissionUrl in SubmissionServlet class and adopt the init method to fill the attribute with the parameter value from the properties file.
- Add an extension to the doPost method that will submit a received form to Content Manager servlet, if a URL to the servlet was specified.
- Create an additional generic support function processing a post message used for form submission to CM.

Having available the stage 2 SubmissionServlet as prepared in Chapter 5, we can easily extend the existing code with the following lines. (Refer to Example 8-4 through Example 8-7.)

Example 8-4 Add a new property to order.properties file (adjust server name and servlet path

```
#path to Content Manager instance
CMSubmissionUrl=http://cmdemo.svl.ibm.com/formdemo/CMSubmissionServlet
```

Example 8-5 Add a new attribute in SubmissionServlet class

```
// *** CM Integration ***
//URL for Content manager integration
private static String CMSubmissionUrl;
// *** CM Integration ***
```

Example 8-6 Read the property in init method of SubmissionServlet class

```
//      *** CM Integration ***
      CMSubmissionUrl = orderProps.getProperty("CMSubmissionUrl");
      if (CMSubmissionUrl == null){CMSubmissionUrl = "";}
      System.out.println("CMSubmissionUrl: "+ CMSubmissionUrl);
//      *** CM Integration ***
```

Example 8-7 Add code submitting the form to CM in doPost event

```
//      *** CM Integration ***
      if (formState.equals("4") && (!CMSubmissionUrl.equals("")) {
        String url = "";
        if (previousFormState.equals("4")) {
          url = CMSubmissionUrl + "?action=update";
        } else {
          url = url = CMSubmissionUrl + "?action=store";
        }
        System.out.println("Submitting form to cmdemo: " + url);
        String respHTML = sendPost(url, strXFdl,"application/vnd.xfdl");
        System.out.println("CM sent");
        System.out.println(respHTML);
      }
//      *** CM Integration ***
```

The new supporting method submits a received form to CMSubmissionServlet installed in the Content Manager test environment. CMSubmissionServlet can send a response — we will not analyze it here. A print to System.out will log the successful transmission or any errors (Example 8-8).

Example 8-8 Supporting method processing a post request

```
public static String sendPost(String urlStrg, String content, String contentType)
    throws IOException {
    URL url;
    URLConnection urlConn;
    DataOutputStream printout;
```

```

BufferedReader in;
String result = "";
try {
    // create URL, open URL connection
    url = new URL(urlStrg);
    urlConn = url.openConnection();
    // activate input and output, no cache
    urlConn.setDoInput(true);
    urlConn.setDoOutput(true);
    urlConn.setUseCaches(false);
    // set content type
    urlConn.setRequestProperty("Content-Type", contentType);
    // Now sent POST message.
    printout = new DataOutputStream(urlConn.getOutputStream());
    printout.writeBytes(content);
    printout.flush();
    printout.close();
    // Get response
    String str;
    in= new BufferedReader(new InputStreamReader(urlConn.getInputStream()));
    while (null != ((str = in.readLine())))
    {
        result = result + str;
        System.out.println(str);
    }
    in.close();
} catch (MalformedURLException e) {
    e.printStackTrace();
}
return result;
}

```

Having these changes applied to the RAD project, we should create a new WAR file and deploy it to the application server. Since there is already an application WPFormsRedpaper, we will have to update the existing enterprise application with the new WAR file.

Note: We chose here the approach to control the submission to Content Manager based on a property stored in the application and the state of the submitted form. There are other valid scenarios as well (application based only, or alternatively, form based only).

8.3.4 Test the form integration with CM

To test the form integration, we will fill out a form and submit it to DB2 Content Manager. Then we use the eClient to search for our form and metadata. Follow these steps to test your form integration with DB2 Content Manager:

1. Be sure that all necessary servers are up and running:
 - a. WebSphere Application Server (WAS) server1 (where the servlet is deployed)
 - b. Content Manager Resource Manager server
 - c. eClient server
2. Fill in your form using the Workplace Forms Viewer.
 - a. Open the Forms Selection JSP with the URL:
<http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/>
 - b. Click **New Orders**.

- c. Select the forms template you deployed with design changes for the CM Integration.
- d. Fill out the form.
- e. On the toolbar of the tradition form page, click the button **Store a new Quote inCM**.
- f. You should the submission confirmation as shown in Figure 8-17.

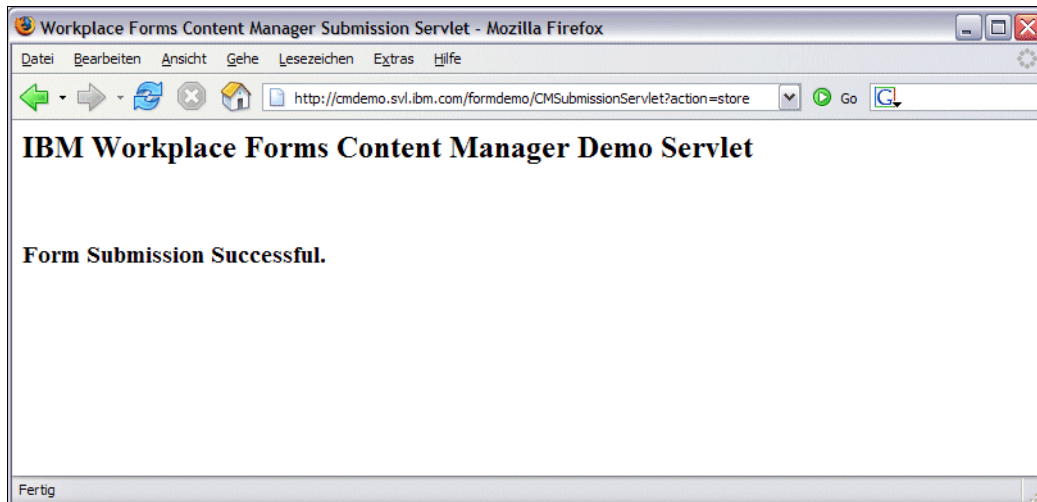


Figure 8-17 Form Submission Confirmation

3. Search and Display your submitted forms in the DB2 Content Manager eClient:
 - a. Start the eClient by clicking the desktop icon or use following URL:
<http://cmdemo.svl.ibm.com:9083/ec1ient/IDMLogon2.jsp>
 Then login to DB2 Content Manager with your valid credentials as shown in Figure 8-18.

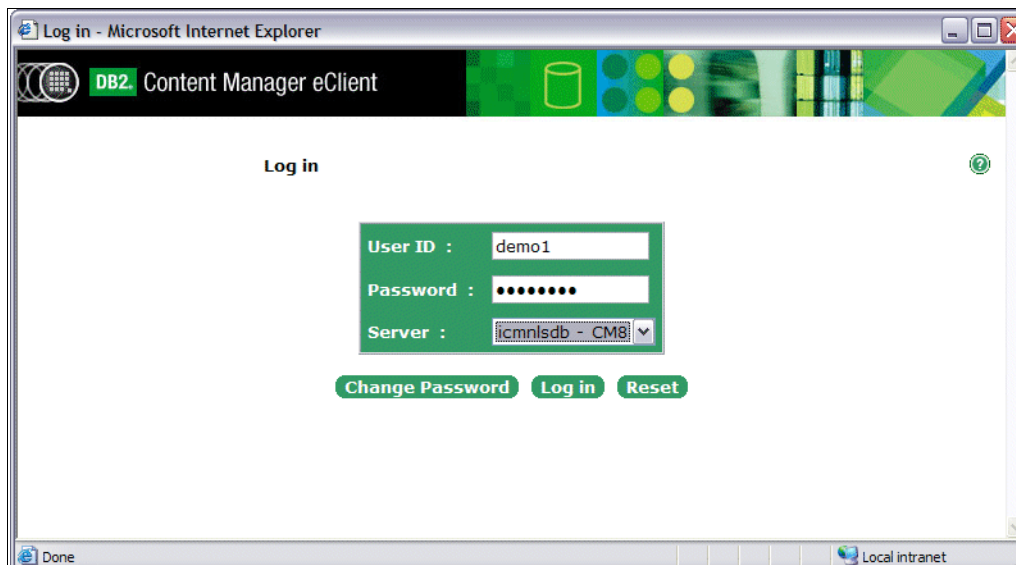


Figure 8-18 DB2 Content Manager eClient Login

- b. Click **Search** to open the Item Type List as shown in Figure 8-19.

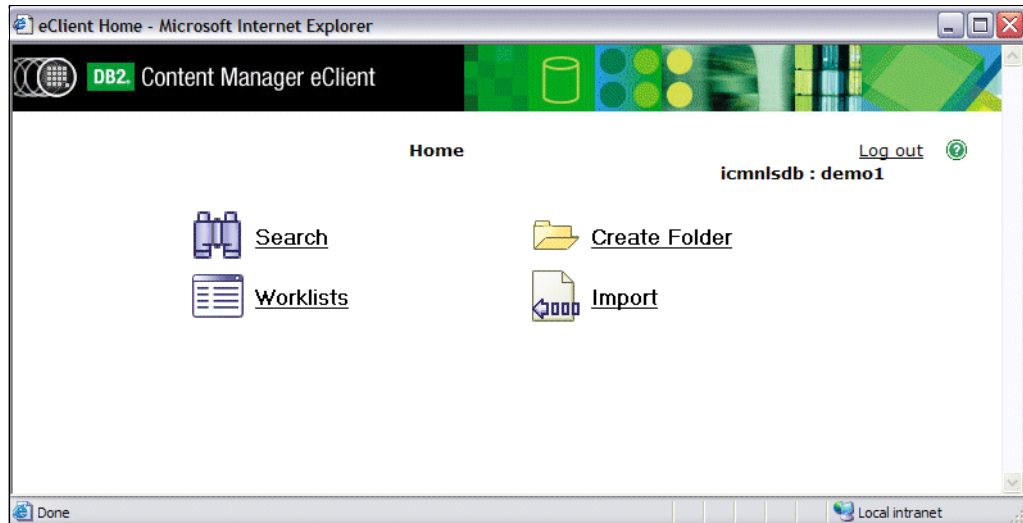


Figure 8-19 DB2 Content Manager eClient Home Page

- c. Scroll down and click **Sales Quote** as shown in Figure 8-20.

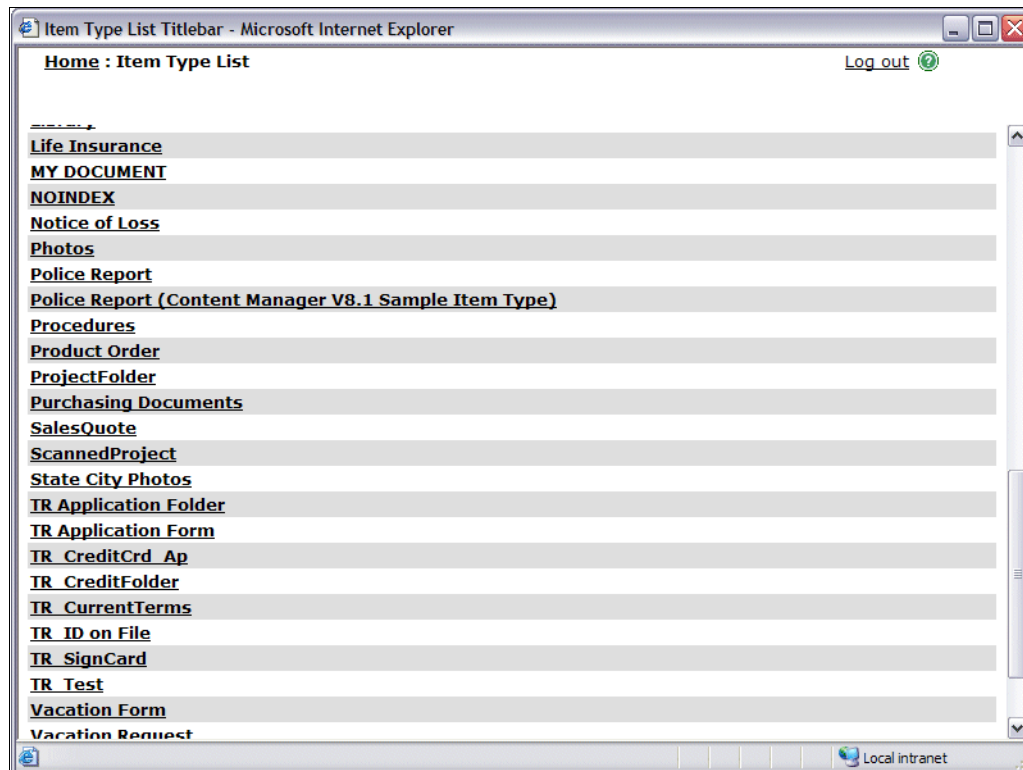


Figure 8-20 DB2 Content Manager eClient Item Type List

- d. Enter a * in field Order ID and click **Search** as shown in Figure 8-21.

Figure 8-21 DB2 Content Manager eClient Basic Search

- e. The Search Result page should open as shown in Figure 8-22.

	Order ID	Requestor Name	Submitter ID	Customer Name	Customer ID	Order Discount	Total Amount	Order State
<input type="checkbox"/>	100012414	Haas	1000	OnDemand Corporation	100000	0	24690.00	2
<input type="checkbox"/>	1000088	Thompson	1001	Portal Application Surfacing	100002	0	4230.00	4
<input type="checkbox"/>	1000089	Kwan	1002	Workplace Forms Redpapers Inc	100003	0.2	4103.92	4

Figure 8-22 DB2 Content Manager eClient Search results page

The Search results page shows you the submitted forms and the metadata that we specified for the Content Manager integration in a view perspective. To update an existing form, you can click the document icon at the beginning of each row. This will open the form in the Workplace Forms Viewer as a plug-in to your browser.



Domino integration

This chapter provides a new integration scenario based on a Domino infrastructure. It builds upon the same base sample application scenario described in Chapter 4, “Building the base scenario: Stage 1” on page 53, and Chapter 5, “Building the base scenario: Stage 2” on page 145, but now focuses on leveraging Domino for the back-end data.

Note: The code used for building this sample scenario application is available for download. For specific information about how to download the sample code, please refer to Appendix A, “Additional material” on page 333.

Note: All specific examples shown and used when building the sample scenario application are based on the codebase for IBM Workplace Forms Release 2.5.

9.1 Introduction to integration of Domino and Workplace Forms

Before proceeding directly into the technical details of this integration scenario, we provide a brief overview of the Domino Server and examine how each technology (namely, the Domino Server and Workplace Forms) can complement each other in the integration scenario.

IBM Lotus Domino server provides enterprise-grade collaboration capabilities that can be deployed as a core e-mail and enterprise scheduling infrastructure (IBM Lotus Domino Messaging Server), as a custom application platform (IBM Lotus Domino Utility Server), or both (IBM Lotus Domino Enterprise Server). An integral part of the IBM Workplace family, Lotus Domino server and its client software options deliver a reliable, security-rich messaging and collaboration environment that helps companies enhance the productivity of people, streamline business processes, and improve overall business responsiveness.

Besides messaging, calendaring, and scheduling, Domino integrates document management, workflow, collaboration, and database capabilities in one comprehensive product. It can handle both structured information and unstructured information as file attachments, and it adheres to Internet standards and protocols such as HTTP, XML, POP3, IMAP4, MIME, SMTP, DIIOP, and more. It can execute Java, JavaScript, LotusScript, and Notes @Formula language.

9.1.1 How can the two technologies complement each other?

Domino provides a great starting point to integrate with forms. We can use:

- ▶ File storage capabilities to store templates and filled forms
- ▶ Java to access Workplace Forms API
- ▶ Built-in HTTP service and servlet engine to communicate with a browser client
- ▶ Integrated development environment to build the application

How can Domino add value to Workplace Forms?

Domino can handle XFDL file attachments or mime types and store and exchange them with other systems in multiple ways (mail, Web services, connections to external data sources, access to file system, etc.) and access the content in XFDL files by Java API, built-in XML parsers, and text processing capabilities. This makes it easy to build an application handling forms, storing them with high sophisticated access rights to stored forms and extracted data, and attach collaboration and workflow to form and data handling.

How can Workplace Forms add value to Domino?

In Domino we can create forms with a visual editor (Domino Designer) for a fat client (Notes) or thin client (browser) as well. In a fat client, we can assign nested signatures. What makes it reasonable to use Workplace Forms having those built in capabilities?

Using Forms, we can exchange the templates and completed forms with other systems, including the layout, signatures, and internal processing rules of the form. A Domino Document can exchange only its values with other systems, not the processing logic, the layout, or signatures. Furthermore, with Workplace Forms we can integrate with mutual signature methods from authenticated password acceptance to biometric (retinal scans, fingerprint readers) and PKI certificates. And last but not least — we can print out the document in a pixel precise manner.

In the scenario we describe in this chapter, we use the Domino Enterprise Server to host the application, the Notes client for testing and maintenance for supporting data, and the Notes designer to create the code. Finally, we use Domino to work with the form layout and to provide the application navigation.

9.2 Overview and objective of this integration scenario

In comparison with the J2EE / DB2 based scenario described in the earlier chapters, no other systems are involved here. Domino will serve for the following purposes:

- ▶ Template storage
- ▶ Data storage for employee / product / customer data
- ▶ Data storage for submitted requests and extracted data
- ▶ Application server
- ▶ HTML server to communicate with the client plug-in
- ▶ Web service provider for the Web services used in the form

Recreation of the complete application including a high standard user interface is not the goal of this scenario. Instead, we show integration techniques with the Domino environment based on the same form used in the base J2EE integration scenario.

Accordingly, the main focus in this chapter is on the following topics:

- ▶ Show all features used in the J2EE environment, using a complete different application server and back-end storage (prepopulation, Web services, form storage, data extraction).
- ▶ Use the unchanged form created for J2EE scenario in stage 2 and provide necessary adaptations to the Domino environment during form instantiation.
- ▶ Show new techniques for data prepopulation and data retrieval.
- ▶ Make the scenario almost independent from the used XFDL form. The goal is to define form specific behavior as configuration parameters related to the provided form.

Second level features (in terms of Forms integration) as development of the implemented business logic in J2EE stage 2, high-end UI design, and full navigation facilities in the application are supported in a limited way only. These features depend highly on the real application to build, and have to be remade in each project using the available Domino developer skills.

Figure 9-1 illustrates the sample application within the context of a 3-tier architecture. The section outlined with a dashed line illustrates the focus of this integration scenario.

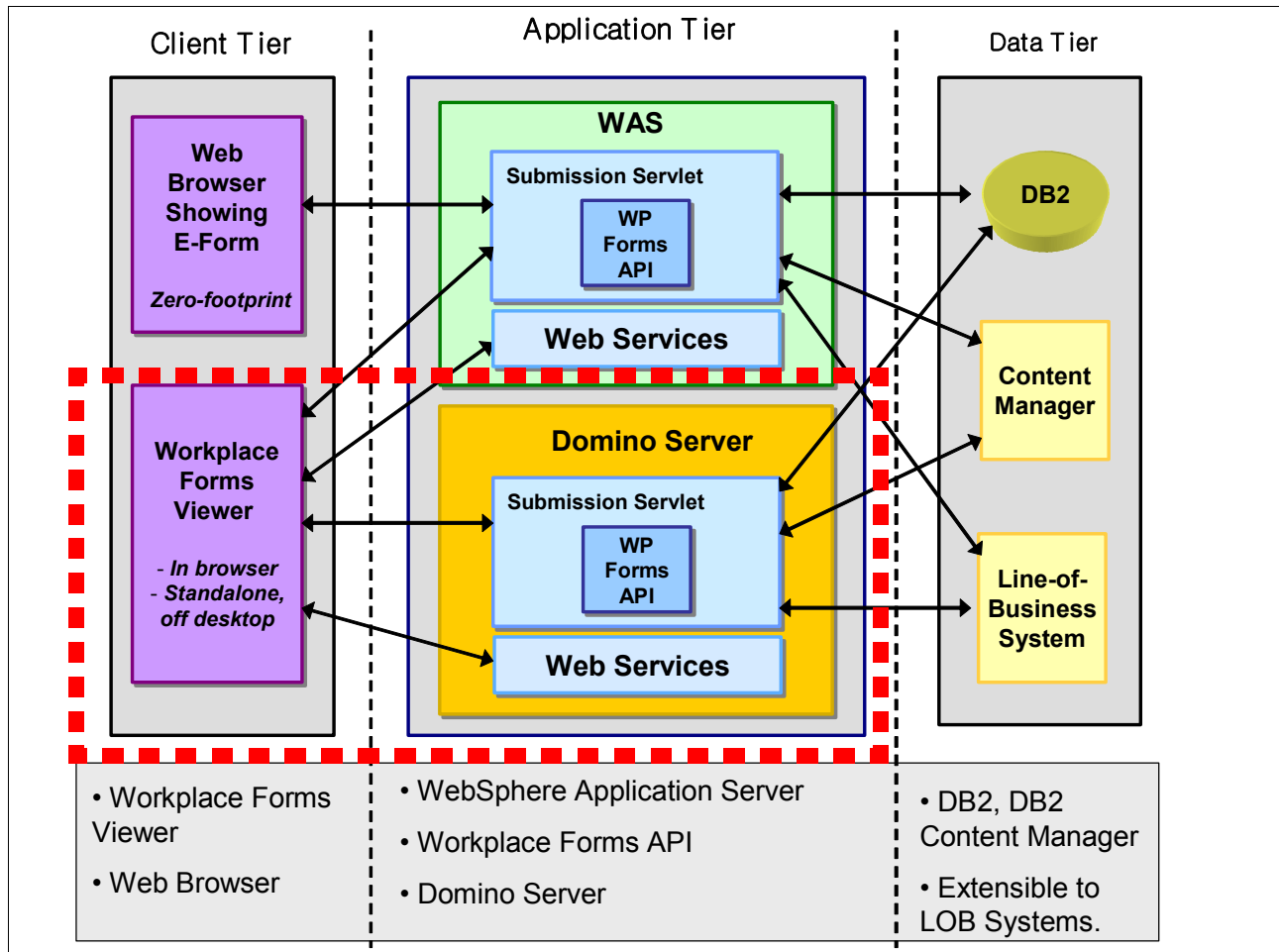


Figure 9-1 Illustrating the focus of this integration scenario

9.3 Environment overview

For Domino integration we will set up an application based on two Domino databases. One database will contain the Workplace Forms templates (Template Database) and store submitted forms and extracted metadata. This database contains the application UI, which the end user accesses with the browser. The other database (Repository Database) will serve as a back-end store for all other data such as customer list, employee data, and product catalog.

A diagram of this scenario is shown in Figure 9-2.

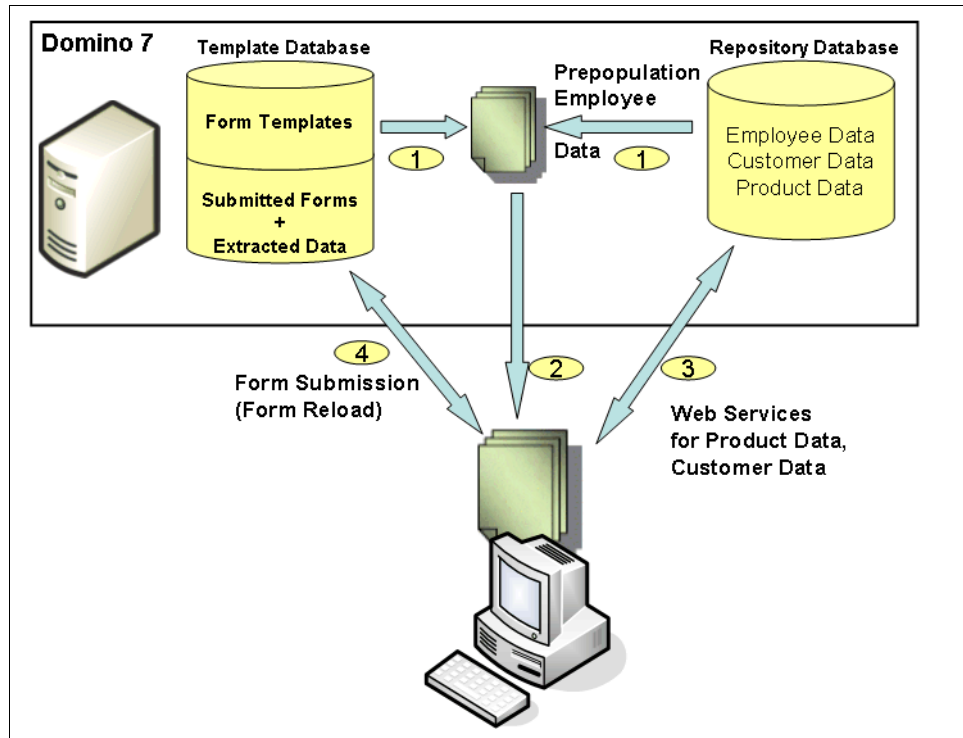


Figure 9-2 Basic usage scenario for Domino integration

The end user opens the Template Database with the browser and can navigate through available views showing links to different XFDL forms (templates, submitted forms for manager or director approval, approved forms, and canceled forms) similar to the J2EE stage 1 scenario, described in Chapter 4, “Building the base scenario: Stage 1” on page 53. Accessing this database, the user will have to authenticate against the Domino Directory.

Creating a new form from a chosen template, the Domino server will gather the employee data based on the user name and prepare this data along with the new order number and the template form for client download.

The client will — as in the J2EE scenario described in Chapter 5, “Building the base scenario: Stage 2” on page 145 — read product and customer data using a Web service. The service runs this time against a Domino based service provider.

The client submits the form to the Domino server. The server extracts the desired metadata and stores the form and Metadata in a Notes document. This document (metadata and stored form) will be updated, whenever the contained form is re-opened and re-submitted.

This chapter does not contain all detailed information necessary to recreate the integration environment. Rather, we refer to the available building blocks and discuss relevant topics.

Full application design and sample data is available in the redpaper repository.

You will find in this chapter the following new integration techniques:

- ▶ Embedding the XFDL form into an HTML page
- ▶ Prepopulation using a script in HTML page
- ▶ Prepopulation using text parsing
- ▶ Data extraction using API based on entire data instances, not single fields

Attention: Using text parsing to interact with the form is a very powerful method, because there are nearly no restrictions to the available operations like having appropriate navigation / data access methods offered by API or setting up right namespaces to access specific data. Nevertheless, there are some considerations to make before using text parsing:

- ▶ Be aware about the compression state of the form. You might have to uncompress the form before you can access it. XFDL uses base64-gzip encoded compression. The Forms API would do compressing/uncompressing in a transparent mode.
- ▶ It is quite easy to break a form using text parsing when inserting restricted characters.
- ▶ It is easy to break signatures written back to the form. Using the API would not allow a change to signed data or structures.
- ▶ It is really difficult to verify signatures when not using the Forms API.

Next, after setting up the system, we will discuss the development tasks.

9.4 Setting up the Domino environment

Domino environment is set up in a standard way. We will not describe these steps in detail. Make sure the following environment is available:

- ▶ Domino Server version 7.0 or later
- ▶ Servlet support enabled (Domino Servlet Manager) in Domino Directory/server document
- ▶ Tasks running: HTTP, Indexer
- ▶ Workplace Forms API installed
- ▶ Basic authentication enabled on the server (not session authentication)

Make sure that you have registered the deployed Workplace Forms API jar files in server notes.ini like this:

```
JavaUserClasses=[System32]\PureEdge\65\java\classes\pe_api.jar;[System32]\PureEdge\65\java\classes\ui_api.jar
```

Using complete databases from the redpaper repository, deploy both databases to the server in the redpaper directory. Sign both databases with an administration ID able to run unrestricted agents and Web services on the server. The ID should have manager rights in the ACL of both databases.

Building the application up, create two new databases in the path, redpaper/WPForms.nsf (Title WPForms Templates) and redpaper/WPFormsRep.nsf (Title Repository DB).

Apply the following ACL settings to the databases (Table 9-1).

Table 9-1 ACL settings for Domino integration

Database	User / Group	ACL-Settings
Template Database	Default	Author
	Anonymous	No Access (this will require user authentication)
Repository Database	Default	Reader
	Anonymous	Reader (necessary for Web service access)

To make user authentication match with the available employee data, open the repository database and edit / create some users applying in the Notes Username field the valid user name from the address book like this (Figure 9-3).

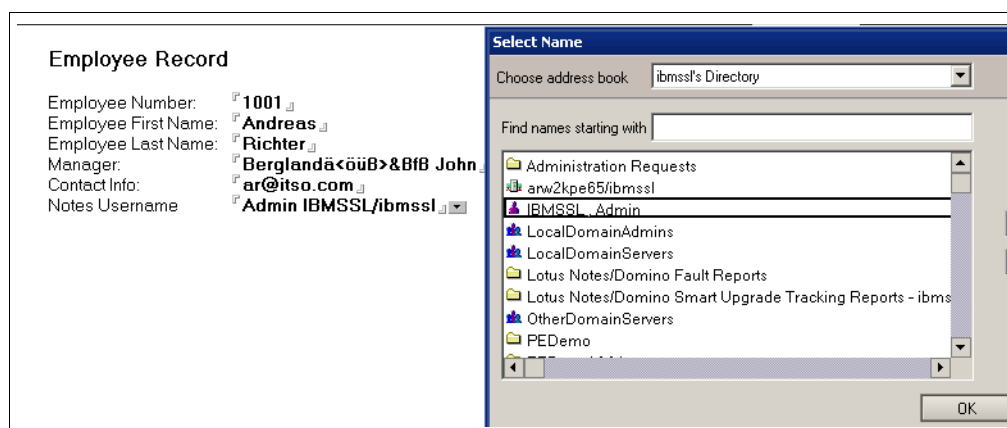


Figure 9-3 Assign valid Notes Usernames to the employees

Building the databases up from scratch, enter some sample data after finishing the form and view design in the Repository Database.

9.5 Domino development

There are several parts to create for Domino integration. We will use Notes @Formula, LotusScript, and Java to create all the necessary code. The development part will contain the following phases:

- ▶ Creating necessary static forms and views.
- ▶ Creating XFDLRendering form used to process data prepopulation
- ▶ Creating a servlet that will receive the submitted form (extract data and store form and data)
- ▶ Creating some dedicated views with links, creating new documents from the template, or opening existing XFDL forms
- ▶ Providing Web services needed to supply the product and customer data.

A main guideline for this chapter was not to write code related to a specific form. Instead we will see code processing a whole class of templates, which will then be parameterized to the needs of a specific form. This will be valid for the modules creating forms from templates (and prepopulating them with data) and receiving submitted forms for data retrieval and data storage.

9.5.1 Repository database

The repository database contains some external data, that we will access (read) while creating forms or working on forms. There is no sophisticated logic in the database. We have to compose just a few forms to store the data (Employees, customers and items) and some views to list them for lookup-purpose.

Forms

All Forms used in the repository database (employee data, customer data and product data) are quite simple; just store the necessary fields in the form. The only relevant field in the Notes Username is employee data. Make the field a Names field ready to pick up names from a directory. We need to store the names in full canonicalized form (such as CN=Andreas Richter/OU=DEP12/O=ACME) to match the user name available in the session when the user authenticates. Some samples for form design (make sure to match the the form name and the field names in your application) are shown in Figure 9-4, Figure 9-5, and Figure 9-6.

Customer Record	
ID	<input type="text" value="CUST_ID T"/>
Company Name	<input type="text" value="CUST_NAME T"/>
Contact Name	<input type="text" value="CUST_CONTACT_NAME T"/>
CRM Number	<input type="text" value="CUST_CRM_NO T"/>
Contact Position	<input type="text" value="CUST_CONTACT_POSITION T"/>
Contact Email	<input type="text" value="CUST_CONTACT_EMAIL T"/>
Contact Phone	<input type="text" value="CUST_CONTACT_PHONE T"/>
Related Account Manager	<input type="text" value="CUST_AMGR T"/>

Figure 9-4 Form Customer

Employee Record	
Employee Number:	<input type="text" value="Org_ID T"/>
Employee First Name:	<input type="text" value="Org_FirstName T"/>
Employee Last Name:	<input type="text" value="Org_LastName T"/>
Manager:	<input type="text" value="Org_MGR T"/>
Contact Info:	<input type="text" value="Org_ContactInfo T"/>
Notes Username	<input type="text" value="UserName T"/>

Figure 9-5 Form Employee

Item Record	
ID	<input type="text" value="IT_ID T"/>
Item	<input type="text" value="IT_Name T"/>
Price	<input type="text" value="IT_Price #"/>
Stock	<input type="text" value="IT_Stock #"/>

Figure 9-6 Form Item

Views

Create some maintenance views for each form sorted by the corresponding ID field. We will use these views for the Web services to look up detail data. In addition, we need one lookup view for employee data by name:

View (luEmployeesByNotesName) sorted by the Notes Username field, second column shows @Text(@DocumentUniqueID)

Now enter some sample data.

Web services

There are potentially three Web services to create (for employee data, customer data, and product catalog). We will show here only the creation of one Web service — CustomerInfo. The other Web services can be created the same way.

First we will take a bottom up approach to create a WSDL file based on a class. Next we will adopt the file and create the final Web service implementation in “Top Down” mode using the changed WSDL file.

Open the repository database in the Domino Designer, *Shared Code / WebServices* pane.

Click the **New WebService** button, name it CustomerInfo, and close the property box (there is no portType class entered for now). See Figure 9-7.

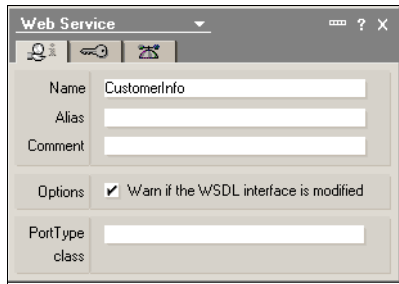


Figure 9-7 CustomerInfo Web service properties

Apply the Option Declare Statement in (Options) and create in (Declarations) the following classes representing the customer object and all its attributes we will maintain (Example 9-1).

Example 9-1 Code listing of declarations for customer object

```
Class CUSTOMER
    'this class will define the customer object and its attributes
    Public CUST_ID As String
    Public CUST_NAME As String
    Public CUST_AMGR As String
    Public CUST_CONTACT_NAME As String
    Public CUST_CONTACT_POSITION As String
    Public CUST_CONTACT_PHONE As String
    Public CUST_CONTACT_EMAIL As String
    Public CUST_CRM_NO As String

    Sub NEW
    End Sub

End Class

Class CustomerInfo
    'this class defines the web service methods (get List and get detail data)
    Sub NEW
    End Sub

    Function GETCUSTOMERLIST(FILTER As String) As String
        'this function will return the filtered customer list as string

    End Function

    Function GETCUSTINFO(CUST_ID As String) As CUSTOMER
        'this function will return customer detail data for one selected customer
```

End Function

End Class

Reopen the properties box for the Web service and name the created class CustomerInfo as the Web service definition class (Figure 9-8).

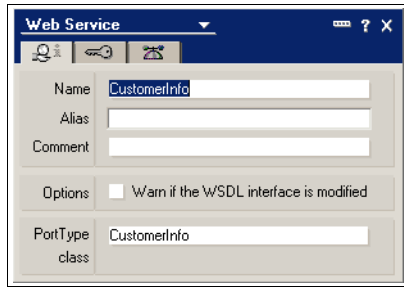


Figure 9-8 Assigning CustomerInfo as PortType class

Close the property box, save the Web service (CTRL-S), and export the WSDL file to the file system using the **Export WSDL** button (Example 9-2).

Example 9-2 Generated WSDL for CustomerInfo Web service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:DefaultNamespace"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="urn:DefaultNamespace" xmlns:intf="urn:DefaultNamespace"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="urn:DefaultNamespace" xmlns="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
      <complexType name="CUSTOMER">
        <sequence>
          <element name="CUST_ID" type="xsd:string"/>
          <element name="CUST_NAME" type="xsd:string"/>
          <element name="CUST_AMGR" type="xsd:string"/>
          <element name="CUST_CONTACT_NAME" type="xsd:string"/>
          <element name="CUST_CONTACT_POSITION" type="xsd:string"/>
          <element name="CUST_CONTACT_PHONE" type="xsd:string"/>
          <element name="CUST_CONTACT_EMAIL" type="xsd:string"/>
          <element name="CUST_CRM_NO" type="xsd:string"/>
        </sequence>
      </complexType>
    </schema>
  </wsdl:types>
  <wsdl:message name="GETCUSTINFORequest">
    <wsdl:part name="CUST_ID" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="GETCUSTINFOResponse">
    <wsdl:part name="GETCUSTINFOReturn" type="impl:CUSTOMER"/>
  </wsdl:message>
  <wsdl:message name="GETCUSTOMERLISTResponse">
    <wsdl:part name="GETCUSTOMERLISTReturn" type="xsd:string"/>
  </wsdl:message>
```

```

<wsdl:message name="GETCUSTOMERLISTRequest">
  <wsdl:part name="FILTER" type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="CustomerInfo">
  <wsdl:operation name="GETCUSTOMERLIST" parameterOrder="FILTER">
    <wsdl:input message="impl:GETCUSTOMERLISTRequest" name="GETCUSTOMERLISTRequest"/>
    <wsdl:output message="impl:GETCUSTOMERLISTResponse" name="GETCUSTOMERLISTResponse"/>
  </wsdl:operation>
  <wsdl:operation name="GETCUSTINFO" parameterOrder="CUST_ID">
    <wsdl:input message="impl:GETCUSTINFORequest" name="GETCUSTINFORequest"/>
    <wsdl:output message="impl:GETCUSTINFOResponse" name="GETCUSTINFOResponse"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="DominoSoapBinding" type="impl:CustomerInfo">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="GETCUSTOMERLIST">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="GETCUSTOMERLISTRequest">
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:DefaultNamespace" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="GETCUSTOMERLISTResponse">
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:DefaultNamespace" use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="GETCUSTINFO">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="GETCUSTINFORequest">
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:DefaultNamespace" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="GETCUSTINFOResponse">
      <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:DefaultNamespace" use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="CustomerInfoService">
  <wsdl:port binding="impl:DominoSoapBinding" name="Domino">
    <wsdlsoap:address location="http://localhost"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

This WSDL file contains some settings, that can cause problems in RAD6 Web service support and other settings, we would like to change to meet platform independent names.

In the WSDL file, change the following strings, wherever they occur (Table 9-2).

Table 9-2 Changed names in WSDL file

String in created WSDL file	New string
urn:DefaultNamespace	http://WPFormsRedpaper
Domino	WPFormsRedpaper

Save the file. This file can be used to create Web services for WebSphere Application Server (WAS) 6 / Tomcat 5.0 using RAD6 and Domino. Re-import it to the Web service.

Note: Compare the created WSDL with the project WSDL files provided in the redpaper repository. If the WSDL files are not identical, this will not break the Web service, but you will have to include the new WSDL file in the sample form in place of the currently contained WSDL to make the form fit to the created Web service. To avoid this, you can import in Domino Designer the WSDL from the redpaper repository.

The initial class definitions are now replaced with new class definitions (Example 9-3).

Example 9-3 Code listing of new class definitions

```
%INCLUDE "lsxsd.lss"
Class CUSTOMER_n0

    Public CUST_ID As String
    Public CUST_NAME As String
    Public CUST_AMGR As String
    Public CUST_CONTACT_NAME As String
    Public CUST_CONTACT_POSITION As String
    Public CUST_CONTACT_PHONE As String
    Public CUST_CONTACT_EMAIL As String
    Public CUST_CRM_NO As String

    Sub NEW
    End Sub

End Class

Const n0 = "WPFormsRedpaper"
Class CustomerInfo_n0

    Sub NEW
    End Sub

    Function GETCUSTOMERLIST(FILTER As String) As String
    End Function

    Function GETCUSTINFO(CUST_ID As String) As CUSTOMER_n0
    End Function

End Class
```

Now we can use this skeleton to enter the code for Web service implementation. The final Web service implementation will have these classes applied (Example 9-4).

Example 9-4 Code listing for Web service implementation

```
%INCLUDE "lsxsd.lss"
Class CUSTOMER_n0

    Public CUST_ID As String
    Public CUST_NAME As String
    Public CUST_AMGR As String
    Public CUST_CONTACT_NAME As String
    Public CUST_CONTACT_POSITION As String
    Public CUST_CONTACT_PHONE As String
    Public CUST_CONTACT_EMAIL As String
    Public CUST_CRM_NO As String

    Sub NEW
```

```

End Sub

End Class

Const n0 = "WPFormsRedpaper"
Class CustomerInfo_n0

Sub NEW
End Sub

Function GETCUSTOMERLIST(FILTER As String) As String
    'filter can contain any valid pattern for Like function
    'Example: A*
    'populate the products variable
    Dim session As New NotesSession
    Dim db As NotesDatabase
    Dim view As NotesView
    Dim ec As Long
    Dim entryCollection As NotesViewEntryCollection
    Dim entry As NotesViewEntry
    Dim customers As String

    Set db = session.currentDatabase

    Set view = db.GetView("Customers")

    Set entrycollection = view.allentries

    Set entry = entrycollection.GetFirstEntry

    'create a string like Name1 [ID1]~Name2 [ID2]~...~NameN [IDN]
    For ec=0 To (entrycollection.count -1)
        If (entry.ColumnValues(0) Like filter) Or filter="" Then
            If customers = "" Then
                customers = entry.ColumnValues(1)+" [" +entry.ColumnValues(0)+ "]"
            Else
                customers = customers + "~" + entry.ColumnValues(1)+" [" +
+entry.ColumnValues(0)+ "]"
            End If
        End If

        Set entry = entrycollection.GetNextEntry(entry)
    Next

    getCustomerList = customers
End Function

Function GETCUSTINFO(CUST_ID As String) As CUSTOMER_n0
    'CUST_ID should contain an ID or a string like Name [ID]
    On Error Goto errorHandler

    Dim session As New NotesSession
    Dim db As NotesDatabase
    Dim view As NotesView
    Dim entry As NotesDocument

    Dim cInfo As New Customer_n0

    'get target document
    Set db = session.CurrentDatabase

```

```

If Instr(cust_id, "[") > 0 And Instr(cust_id, "]") = Len(cust_id) Then
    'extract ID from input parameter
    cust_id = Strleft(Strrightback(cust_id,"[", "]")
End If
Set view = db.getView("Customers")
'get the document
Set entry = view.GetDocumentByKey(cust_id, True)
If Not entry Is Nothing Then
    'read the fields
    cInfo.cust_id= entry.getItemValue("cust_id")(0)
    cInfo.cust_name= entry.getItemValue("cust_name")(0)
    cInfo.cust_amgr= entry.getItemValue("cust_amgr")(0)
    cInfo.cust_contact_name= entry.getItemValue("cust_contact_name")(0)
    cInfo.cust_contact_position= entry.getItemValue("cust_contact_position")(0)
    cInfo.cust_contact_phone= entry.getItemValue("cust_contact_phone")(0)
    cInfo.cust_contact_email= entry.getItemValue("cust_contact_email")(0)
    cInfo.cust_crm_no= entry.getItemValue("cust_crm_no")(0)
Else
    MsgBox "entry not found: " & cust_id
End If

Set getCustInfo = cInfo

Exit Function
errorhandler:
    MsgBox "Error " & Err() & " in " & Lsi_info(2) & " line " & Erl() & ": " &
Error$
Resume Next

End Function

End Class

```

Save the service — it is ready to run.

The same procedure should be processed for the other Web services (EmployeeInfo and ProductInfo).

Tip: The EmployeeInfo Web service is created but not used in the redpaper demo XFDL form. You might not want to create it.

9.5.2 Template Database: components to create a new form from template

The Template Database has the role of the application database. It will contain several correlated design elements, which work together when creating new forms or opening existing forms in the browser.

For form creation, we will have to create the following components:

- ▶ A form to store templates
- ▶ A form to render the HTML page shown to the browser when creating a new request
- ▶ An initiation agent that will fill the rendering form
- ▶ A View listing all available forms along with suitable links to compose a new XFDL form from the template
- ▶ At least one stored template with prepopulation settings applied to test the created design elements.

- A parameter form and view storing necessary application parameters (order count, server URL, etc.)

The database will receive submitted forms as well — so we need an additional form to store submitted forms and at least one lookup view to find the related document on subsequent updates to the submitted XFDL documents, for example, those going through the approval workflow. We will create all these components in the next sections.

Parameter form and view

Here, we will create one simple form to store some application parameters. We will use these documents to store the DNS name of the server using it to create http links and we will store here the order number counter that increments on each new order creation. Give it an additional author field (Figure 9-9).

Figure 9-9 Parameter Form in Template Database

For lookup and maintenance, create one view sorted by field PAR_KEY like this (make sure that the fourth column presents the @DocumentUniqueID (Figure 9-10).

Key	NumValue	TextValue	UNID
OrderCounter	100093	Counter for next order number	37679CF96573D1C1C125712D007CA575
ServerUrl		http://arw2kwpl25.ibm.com	DEAB58525E62832FC12571360057A2B6
SubmissionUrl		/servlet/XFDLServletRedPaper?action	B4D23A78009383BEC125713D007FF474

Figure 9-10 Parameter lookup view

Template form

The template form will basically store the form templates as file attachments in a rich text field. Create a form named FormTemplate like this (Figure 9-11).

Figure 9-11 Basic fields for template storage

The name field should contain a short name for the template. This will be the link text that is presented to the end user when creating a new form from the template.

This form should not store only the form XFDL template, but contain also some additional functionality:

- ▶ It gets a set of fields used to define any prepopulation applied to the template when creating a new form. This makes it possible to run a generic prepopulation module, and prepopulate a different data set on form initiation.
- ▶ It needs to provide a converted version of the original XFDL template that can be easily accessed in the prepopulation moment.

To convert the attachment, open the PostSave event of the form and insert the following code (Example 9-5).

Example 9-5 Attachment converting in PostSave event

```
Sub Postsave(Source As NotesUIDocument)

    'detach the attachment to temp directory and store it back to bodyInline field as mime
    type
    Dim doc As NotesDocument
    Set doc = source.Document
    Dim session As New NotesSession
    Dim rtitem As NotesRichTextItem

    'get the body field
    Set body = doc.GetFirstItem("Body")
    Call body.update

    'get temp dir
    tmpdir = Environ("temp")
    If tmpdir = "" Then tmpdir = Environ("tmp")
    If tmpdir = "" Then tmpdir = "c:\temp"
    Dim sep As String
    sep = "\"
    If Instr(tmpdir, "/") > 0 Then sep = "/"
    Dim fileId As Variant

    'get attachment (should be only one!!)
    Forall att In body.EmbeddedObjects
        If att.Type = EMBED_ATTACHMENT Then
            filepath = tmpDir & sep & att.Source
            Print "filepath: " & filepath
            Call att.ExtractFile(filepath)
            found = True
        End If
    End Forall

    'Read the detached file into inputstream and append to Body richtext field on rendering
    form
    Dim stream As NotesStream
    Set stream = session.CreateStream
    If Not stream.Open(filepath, "ISO-8859-1") Then
        Print "Open failed"
        Exit Sub
    Else
        Print "Opened file " + filepath
    End If
    If stream.Bytes = 0 Then
```

```

        Print "File has no content"
    Exit Sub
End If

'clear up when updating
If doc.HasItem("BodyInline") Then
    Set rtitem=doc.getFirstItem("BodyInline")
    Call rtitem.remove
    Set rtitem=doc.CreateRichTextItem("BodyInline")
Else
    Set rtitem=doc.CreateRichTextItem("BodyInline")
End If
'store new form
Call rtitem.appendtext(stream.ReadText())
Call stream.Close
Call doc.Save(True, True)
'delete detached attachment from file system
Kill filepath
End Sub

```

This procedure will store the content of the attachment as plain in-line text to the field *BodyInline* as a mime type. Mime types can be accessed as streams without first storing the content as a file on the file system, when a new form should be created. Actually, the Notes client / Domino server can read only mime types and files as streams, not simple attachments. To prevent storing files on the file system on each new form on the server, we just do it after deploying a new form on the Notes client on document save. After each attachment update, the procedure will rewrite the content in the *BodyInline* field.

To prevent caching in the browser, insert the following code in the form's HTML header content (Example 9-6).

Example 9-6 HTML header code preventing IE6 from caching old content

```

"<META HTTP-EQUIV=\"Pragma\" CONTENT=\"no-cache\">"+
"<META HTTP-EQUIV=\"Expires\" CONTENT=\"-1\">"

```

To pre-configure prepopulation, we must define a framework that will fit our needs. In this redbook we assume that prepopulation will be based in most cases on some data stored in other Domino documents and that this data corresponds to data instances in the XFDL form.) For this purpose, the framework we create will do the job. If the scenario changes, the framework should be adjusted.

To address a Notes document, we need to define the database and the unique ID of this document. Other combinations are valid as well (such as DB ReplicaID/DocumentReplicaID or Database path/ View Name / sort key), but we will stay on the scenario with DB path and DocumentUniversalID.

Knowing the source document, we must define what data (fields) to use for prepopulation of additional information: The data instance name in the XFDL form to prepopulate, the field names to read from the document, and the corresponding element names for these fields in the data instance contained in XFDL form. This information should be entered in the same document as the related XFDL template. We will configure four fields to store the information (Table 9-3).

Table 9-3 Field description for preprocessing information

Field name	Description
prepopSourceDb	The path to the database storing the requested information. In our scenario this will be the path to the resource database (containing employee data).
prepopIDFormula	The most tricky functionality. Apply here a valid @Formula to compute the UNIQUE ID of the document in the target database, where we will read data for prepopulation.
prepopDataInstance	Instance name to prepopulate. In our scenario this will be mainly FormOrgData containing employee data.
prepopFields	List of field names and corresponding element names in the data instance. Multiple lines possible. A valid entry would be: ORG_FIRSTNAME#FirstName. Match exactly (case sensitive) order and naming in XFDL data instance!

In reality, we could have to prepopulate more than one data instance. Create four identical prepopulation sections (for example, in a tabbed table as shown in Figure 9-12).

Name:

Attached template:

Data Instance 1
Data Instance 2
Data Instance 3
Data Instance 4

Prepopulation Instance 1

Path to prepopulationsource database

@Formula for prepopulation document id

Data instance to prepopulate (tag name and id for datainstance):

Mapping (List of source fieldnames#target elements in data instance)


Figure 9-12 Fields defining a document for data prepopulation

The field names on tab 2, 3, and 4 will be the same as in tab 1, but contain the suffix `_1`, `_2` and `_3`. In runtime, a template document ready to process data instance prepopulation could look as shown in Figure 9-13.

Form Template

Name:

Attached template:

 Redpaper_Forms_Sample_v32.xfdl

Data Instance 1 | Data Instance 2 | Data Instance 3 | Data Instance 4 | Text Parsing

Prepopulation Instance 1

Path to prepopulation source database

@Formula for prepopulation document id

Data instance to prepopulate (tag name and id for data instance):

Mapping (List of source fieldnames#target elements in data instance)

Figure 9-13 Configured Template document with data prepopulation settings

Often the environment requires additional changes to the provided template, for example, adjusting some static entries in the form (such as URLs contained in the template or even the title or some other information) in the initiation process. To do so, we will introduce an additional technique for prepopulation — text parsing. To store search and replace values, we create an additional tab named TextParsing containing one multi value field (replacements), as shown in Figure 9-14.

Data Instance 1 | Data Instance 2 | Data Instance 3 | Data Instance 4 | Text Parsing

Text Parsing

Enter fix text replacements in xfdl file (file must not be compressed)
 Per line one replacements as text in the format *xfdl template#replace#text for new form*
 Use an empty line as separator

Figure 9-14 Replacements tab for text parsing

The information stored in this field will control a search/replace module that will run on the form created from template before it is submitted to the browser. Make sure to apply suitable separators for the field (use Empty line only). We will use this field mainly to exchange URLs pointing originally to the WebSphere Application Server (WAS) / J2EE environment and make them point to the corresponding Domino endpoint (Web service endpoints, submission URL), but we can exchange that way any text fragment contained in the template. A filled TextParsing tab could look like this (Figure 9-15).

Data Instance 1	Data Instance 2	Data Instance 3	Data Instance 4	Text Parsing
-----------------	-----------------	-----------------	-----------------	--------------

Text Parsing

Enter fix text replacements in xfdl file (file must not be compressed)
 Per line one replacements as text in the format *xfd! template#replace#text for new form*
 Use an empty line as separator
 http://vmforms1.cam.itso.ibm.com:8085/WpfWsCustomerT/services/WPFormsCust#replace#http://vmforms1.cam.itso.ibm.com/Redpaper/wpformsrep.nsf/CustomerInfo
 http://vmforms1.cam.itso.ibm.com:8085/WpfWsCatalogT/services/ProductCatalogPort#replace#http://vmforms1.cam.itso.ibm.com/Redpaper/wpformsrep.nsf/ProductInfo
 http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/SubmissionServlet?action=store#replace#http://vmforms1.cam.itso.ibm.com/servlet/XFDLServletRedPaper?action=store&url=http://vmforms1.cam.itso.ibm.com/redpaper/WPForms.nsf
 <value>Product Price Quotation</value>#replace#<value>Product Price Quotation Dominok</value> 』

Figure 9-15 Replacing target URLs and form title label

Now all prerequisites are created to write the code creating a new form from the template:

- ▶ Template storage installed with a template ready to read as a stream
- ▶ Configuration fields for data instance prepopulation setup
- ▶ Configuration for additional test parsing setup

Let us now define how a template will create a new form.

Creating a new form from template, embedding template in HTML form

As in the J2EE scenario, we will need a procedure reading the template and preparing the data for prepopulation. This can be easily done in Domino, calling an agent (LotusScript or Java) form by a URL fired from the browser client. We will take another approach here by triggering a URL that opens a new form in the Template Database.

This form will run upon opening an agent that collects the necessary information and then sends an HTML page to the browser containing the new form ready for viewing via the Viewer plug-in. We will call the agent the “initiation agent” here. Alternatively, we could create a Java agent, we could reuse the main code from the J2EE environment, but let us first show how to make things work using LotusScript.

LotusScript can easily access external libraries, if they are registered as OLE Automation classes. Unfortunately Workplace Forms offers a COM API, but the registration does not show them up as automation classes. So it would be hard to access these classes. Assuming another operation system for the server environment as Windows, the COM interface is not available at all — so we should look for other techniques to do the prepopulation.

Workplace Forms provides a way to embed an XFDL file into an HTTP page. This is done registering a Workplace Forms object to the HTML form and including a full XFDL file as a <SCRIPT> element activated by the object. The HTML page can contain additional initiation scripts, which will contain prepopulation instructions and data. When the browser opens the HTTP page, the object is initiated (the Viewer will pop up). The Viewer renders the contained form and executes all registered scripts from the HTML page, before the user can access the page.

The HTML page to create would look like this (Example 9-7).

Example 9-7 Sample for an embedded XFDL document

```
<BODY>
<HTML><BODY>
<OBJECT id="Object2" height="2000" width="980" border="1"
classid="CLSID:354913B2-7190-49C0-944B-1507C9125367">
<PARAM NAME="XFDLID" VALUE="XFDLData">
<PARAM NAME="instance_1" VALUE="ElementName InstanceId replace [0]">
```

```

</OBJECT>
<SCRIPT id=ElementName type="application/vnd.xfdl; wrapped=comment">
<!--
    <DataInstanceX>
        <FIELD1>val1</FIELD1>
        <FIELD2>val2</FIELD2>
    </DataInstanceX>
-->
</SCRIPT>
<SCRIPT language="XFDL" id="XFDLData" type="application/vnd.xfdl; wrapped=comment">
<!--
<?xml version="1.0" encoding="ISO-8859-1"?>
<XFDL xmlns="http://www.PureEdge.com/XFDL/6.5"
....
the full xfdl document is located here
....
</XFDL>
-->
</SCRIPT>
<BODY>

```

The page contains the following elements, as listed in Table 9-4.

Table 9-4 Element description for an embedded XFDL page

<OBJECT>	Description
<OBJECT>	Object element registering the Viewer class and setting up Viewer behavior with the contained attributes and child elements. ID - arbitrary but unique name in the page. width, height - Space allocated in browser by the open Viewer. classid - ID of Viewer activeX control in Windows registry.
<PARAM NAME="XFDLID" VALUE="XFDLData">	Defines the ID of the script element that contains the XFDL form. The value can be anything, as long as the XFDLID and the script ID match.
<PARAM NAME="instance_1" VALUE="ElementName InstanceId replace [0]">	Identifies XML instances inside an HTML page. This instances can be used to modify specified XML instances inside the XFDL form. This information includes: <ul style="list-style-type: none"> • The ID of the new instance to create • The ID of the form instance to locate Two additional values may be included: <ul style="list-style-type: none"> • Either replace or append, depending upon whether the new instance data replaces or adds to the original instance data. Note that replace is the default value. • The reference within the instance that indicates where the new data should be placed. Note that any namespaces listed in this value resolve relative to the document root. Multiple instance parameters must have sequentially numbered names, starting with 1. For example, instance_1, instance_2, and so on.

<OBJECT>	Description
<pre> <SCRIPT id=ElementName type="application/vnd.xfdl; wrapped=comment"> <!-- <DataInstanceX> <FIELD1>val1</FIELD1> <FIELD2>val2</FIELD2> </DataInstanceX> --> </pre>	Script containing the data for prepopulation. The data must contain a valid XML instance with the data to insert in the XFDL file. The ID must match the first part of the corresponding parameter with the name Instance_x
<pre> <?xml version="1.0" encoding="ISO-8859-1"?> <XFDL </XFDL> </pre>	Included full XFDL file.

To make the page generic for use with multiple forms, we will set up a Domino form containing an HTML skeleton for this data structure. All relevant (or almost all relevant) places containing variable names will be filled dynamically using Domino fields or computed text. The initiation agent will insert the necessary content in the document fields and submit the complete new document as HTML page to the browser (Figure 9-16).

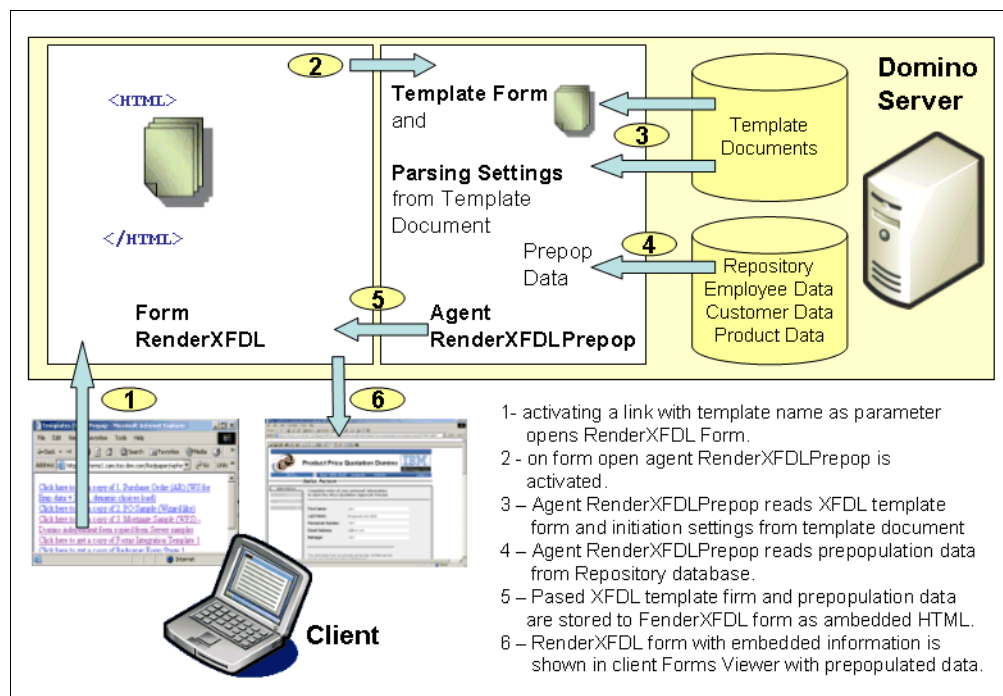


Figure 9-16 High level flow in Domino new document scenario

Initiation form RenderXFDL

First we will create the new document form for HTML rendering. Create a form named *RenderXFDL*. Set content type to HTML (Figure 9-17).

The screenshot shows the 'Form' properties dialog box. The 'Content type' is set to 'HTML'. Other options include 'On Create', 'On Open', 'On Close', 'On Web Access', and 'Data Source Options'.

Figure 9-17 Form properties - Content Type = HTML

Create the following elements on the form (Table 9-5).

Table 9-5 *RenderXFDL* form components

Component	Description
Field Query_String_Decoded	Editable, hidden from browser Contains the decoded query string This CGI variable contains all parameters from the calling URL (in this case the template to open, like <i>ReadForm&FormName=Redpaper Form&Ind=ARWE-6N2ULV</i>)
Static text as Pass-Thru HTML	<BODY> <HTML><BODY>
First part of XFDL object tag	<OBJECT id="Object2" height="2000" width="980" border="1" classid="CLSID:354913B2-7190-49C0-944B-1507C9125367"> <PARAM NAME="XFDLID" VALUE="XFDLData"> The parameters for height, width and border could be variable too - we will use static values here.

Component	Description
The script registration for the first Data instance to prepopulate	<pre><!-- red lines are hidden, when no prepopulation data available (PrepopulationDocUNID="") --> <PARAM NAME="instance_1" VALUE="<Computed Value> <Computed Value> replace [0]"></pre> <p>These lines must be hidden, when we do not have a prepopulation (Apply hide when Formula: PrepopulationDocUNID="") the computed Values will return the content of field prepopDataInstance. The initiation agent will create this field and fill it with the instance ID of the data instance to prepopulate. We will have here identical names for the new instance name and the instance name in HTML document containing new data.</p>
Registration for the other 3 potential prepopulations	<pre><PARAM NAME="instance_2" VALUE="<Computed Value> <Computed Value> replace [0]"> <PARAM NAME="instance_3" VALUE="<Computed Value> <Computed Value> replace [0]"> <PARAM NAME="instance_4" VALUE="<Computed Value> <Computed Value> replace [0]"></pre> <p>Hide each line, when the corresponding prepopulationUNID is empty (Apply hide when Formula: PrepopulationDocUNID_x="" where x = 1, 2 or 3) The computed Values will return the content of field prepopDataInstance_x. The initiation agent will create this field and fill it with the instance ID of the second, third or forth data instance to prepopulate.</p>
Closing tag for the object	</OBJECT>
SCRIPT tag containing the instance data for first data instance to prepopulate	<pre><SCRIPT id=<Computed Value> type="application/vnd.xfdl; wrapped=comment"> <!-- [RTF Field Prepop]--></SCRIPT></pre> <p>HTML Script tag surrounding the field Prepop that will contain the instance data for first prepopulation instance. The computed Values will return the content of field prepopDataInstance. Hide these 2 lines, when we do not have a prepopulation (Apply hide when Formula: PrepopulationDocUNID="")</p>
SCRIPT tag containing the instance data for the other 3 data instances to prepopulate	<pre><SCRIPT id=<Computed Value> type="application/vnd.xfdl; wrapped=comment"> <!-- [RTF Field Prepop_1] --></SCRIPT> <SCRIPT id=<Computed Value> type="application/vnd.xfdl; wrapped=comment"> <!-- [RTF Field Prepop_2] --></SCRIPT> <SCRIPT id=<Computed Value> type="application/vnd.xfdl; wrapped=comment"> <!-- [RTF Field Prepop_3] --></SCRIPT>></pre> <p>HTML Script tags surrounding the fields Prepop_x that will contain the instance data for other 3 prepopulation instances. The computed Values will return the content of the corresponding field prepopDataInstance_x. Hide lines, when we do not have a prepopulation (apply hide when Formula: PrepopulationDocUNID_x="" for each line with x 1, 2, or 3)</p>

Component	Description
Embedded XFDL form	<pre><SCRIPT language="XFDL" id="XFDLData" type="application/vnd.xfdl; wrapped=comment"> <!-- [RTF Field BodyInline] --> </SCRIPT></pre> <p>HTML Script tag surrounding BodyInline field that will contain the embedded XFDL form as plain text.</p>
End tag for Body	</BODY>

All content on this page - except the hidden first line containing the field Query_String_Decoded must be set to Pass-Thru HTML.

As the last action, we will assign a WebQueryOpen agent to the form. Open the WebQueryOpen event and assign the @Formula

```
@Command([ToolsRunMacro]; "RenderXFDLPrepop")
```

The completed form should look like this (Figure 9-18).

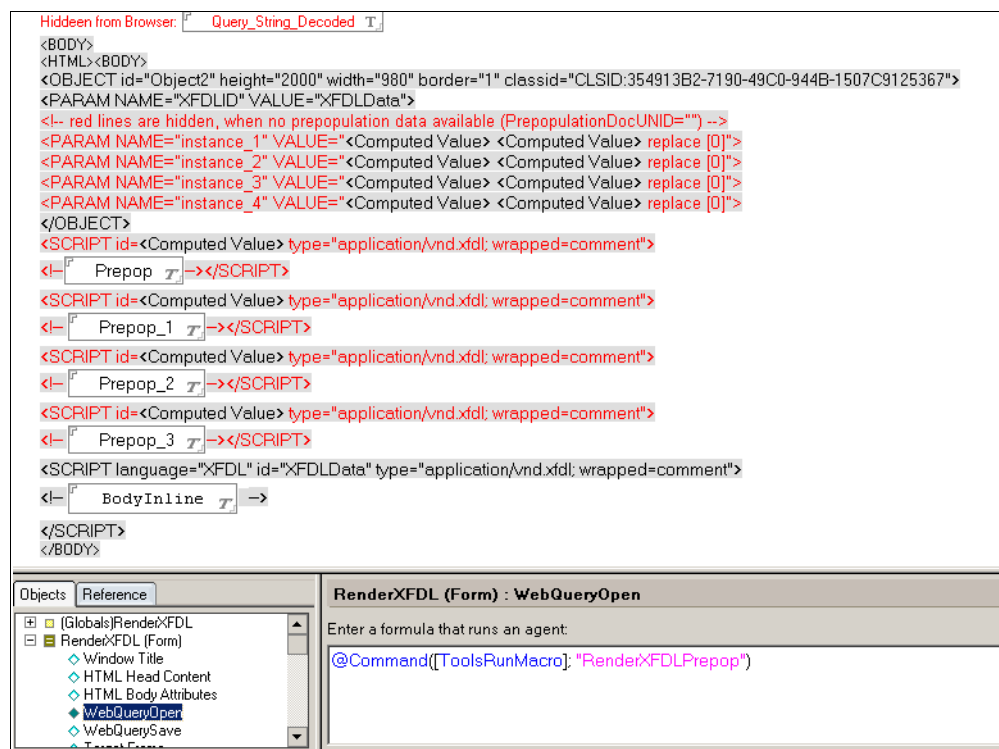


Figure 9-18 Form RenderXFDLPrepop preparing a embedded XFDL form with prepopulation

Now store the completed form.

Tip: The decisions are arbitrary as to what content to render in the form as static text, what content to render as computed text, and what content will be available as content of rich text fields. We could, for example, create the whole content as one large character stream containing the embedding information, the prepopulation data, and the XFDL file, and store it to one single rich text field as well. The structure given in this chapter should make the internal structure visible.

Note: Signing a form actually embedded in an HTML page can cause broken signatures when validating the signature without the embedding page being available at the signing moment. In these cases, you should take care of the options and item types signed by default. To correct this, open the “Advanced Group Options” tab at the detail page of a signing button in the Forms Designer and adjust the group signing options and item types.

Now create the agent filling data in the newly created form.

Initiation agent RenderXFDLPrepop

The initiation agent will run on form open and execute the following actions:

1. Locate the corresponding template document in the database.
2. Read configuration settings from the document (prepopulation settings, search/replace statements).
3. Compute all necessary fields in this document (Set data instance names, create XML structures for instance data).
4. Read the XFDL form template from the template document.
5. Execute text parsing operation (search/replace).
6. Insert the reworked XFDL template in the corresponding field of the new document form.
7. Create a new agent in the database and name it “RenderXFDLPrepop”. Make it a LotusScript Agent running on Target “None”, as a Web user, with no restricted operations.

Running as a Web user will enable us to read the user name from the session object (Figure 9-19).

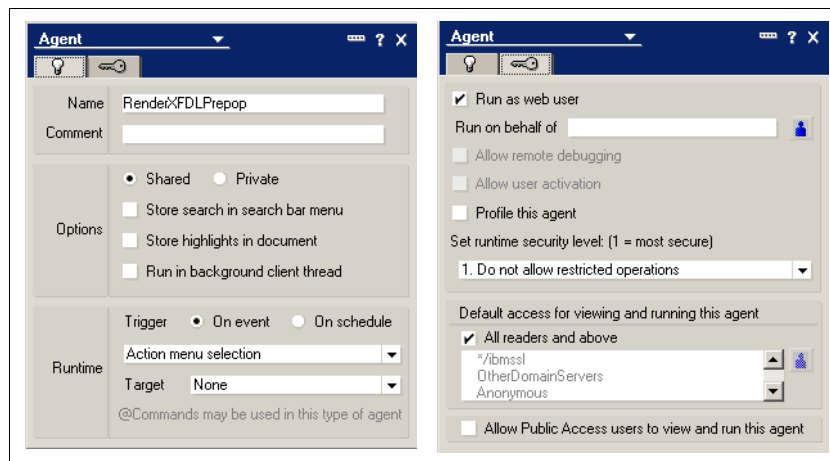


Figure 9-19 Initiation agent settings

Apply the following statements in (Options) and (Declarations) and supporting functions (Example 9-8).

Example 9-8 Options and helper routines for initiation agent

```
'RenderXFDLPrepop:
```

```
Option Public
Option Declare
```

```
Const CONSTLIBNAME = "Agent RenderXFDLPrepop"
```

```

Function encode(Byval strg As String) As String
    'simple encoder eliminating special html characters
    On Error Goto errorhandler
    strg = replaceSub(strg, "&", "&amp;")
    strg = replaceSub(strg, "\"", "&quot;")
    strg = replaceSub(strg, "<", "&lt;")
    strg = replaceSub(strg, ">", "&gt;")
    strg = replaceSub(strg, "'", "&apos;")
    encode = strg
    Exit Function
errorhandler:
    MessageBox "Error: " & Err() & " in " & CONSTLIBNAME & "." & Lsi_info(2) & " line " &
    Err()
    MessageBox Error$
End Function

Function replaceSub(str1 As String, str2 As String, str3 As String) As String
    'replaces str2 in str1 with str3 on each occurrence
    On Error Goto errorhandler
    If Instr(str1, str2) > 0 Then
        replaceSub = Join(Split(str1, str2), str3)
    Else
        replaceSub = str1
    End If
End Function
ex:
    Exit Function
errorhandler:
    MessageBox "Error: " & Err() & " in " & Lsi_info(2) & " line " & Err()
    MessageBox Error$
    Resume ex
End Function

```

Now fill in the Initialize event with the working code for the agent. The logic is straightforward — see in-line comments in the source code for details. Notice that the agent uses an additional helper creating the XML fragments for the prepopulated data instances (Call createXMLFragment(template, doc, prepop, i)) — this sub is missing for now (see Example 9-9).

Example 9-9 Working code for the agent

```

Sub Initialize
    On Error Goto errorhandler
    Dim session As New NotesSession
    Dim db As NotesDatabase      'template database
    Dim doc As NotesDocument     'new created document composing the HTML
                                'page for embedded XFDL form
    Dim template As NotesDocument 'template document containing XFDL template and processing
    parameters
    Dim View As NotesView        'Notes View for lookups
    Dim BodyXFDLtemplate As NotesRichTextItem 'RTF item with stored XFDL template as text
    Dim BodyXFDLform As NotesRichTextItem    'RTF item with stored XFDL template as text
    Dim username As String          'current username
    Dim formName As String 'name of the xfdl template to initiate
    Dim xfdl As String             'complete xfdl form as string for parsing
    Dim initialreplacements As Integer 'Indicator, if prepopulation is required or not
    Dim Prepop As NotesRichTextItem  'RTF receiving the instance data for
                                    'prepopulation (compose XML fragments)

    Dim params As Variant
    Dim CommandStr As String

```

```

'get handle to the new document (Form RenderXFDL)
Set doc = session.DocumentContext 'the new doc with form RenderXFDL
Set db = session.CurrentDatabase

CommandStr = session.DocumentContext.Query_String_Decoded(0)
'should return a param string like this: ReadForm&FormName=Redpaper Form&Ind=ARWE-6N2ULV
'Messagebox "params: " & CommandStr

params = Split(CommandStr , "&")
Forall p In params
    MsgBox "param: " & p
    If Instr(p, "FormName=") = 1 Then formName = Strright(p, "=")
End Forall

'get username from session
userName = session.EffectiveUserName

'Find the template document that contains the XFDL attachment based on the title in
eForm field
MsgBox "get Template: " + formName
Set View = db.GetView("Templates (Notes)")
Set template=View.GetDocumentByKey(formName)
If template Is Nothing Then
    MsgBox "got NO Template: " + formName
Else
    MsgBox "got Template: " + formName
End If

'compose the new xfdl from - first get the handle to form in template doc
Set BodyXFDLtemplate = template.GetFirstItem("BodyInline")

'here is the moment to do test parsing, if this is necessary.
Dim replacements As Variant 'read the replacements field
replacements = template.getitemvalue("replacements")
initialreplacements = False
If replacements(0) <> "" Then initialreplacements = True
MsgBox "Replacements: " & initialreplacements
If initialreplacements Then 'replacements -> read the xfdl form to string
    xfdl = BodyXFDLtemplate.GetUnformattedText

    Forall rep In replacements 'execute search/replace for each entry
        xfdl = replaceSub(xfdl, Strleft(rep, "#replace#"), Strright(rep, "#replace#"))
        MsgBox "Replace: " & rep
    End Forall

    Set BodyXFDLform = doc.getfirstitem("BodyInline" ) 'now apply the new xfdl form
    Call BodyXFDLform.AppendText(xfdl)
Else
    'OK - no replacements - we can copy the full xfdl into the new document
    Set BodyXFDLform = BodyXFDLtemplate.copyitemToDocument(doc, "BodyInline")
End If

'do prepopulation (compose the new data instances if there are valid parameters)
Dim i As Integer
For i = 0 To 3
    'This function we will create in next step
    Call createXMLFragment(template, doc , prepop, i)
Next

```

```

Exit Sub
errorhandler:
    MsgBox "Error: " & Err() & " in " & CONSTLIBNAME & "." & Lsi_info(2) & " line " &
    Err()
    MsgBox Error$
Exit Sub
End Sub

```

Now we create the missing sub, createXMLFragment (see Example 9-10). It composes the XML fragments for one prepopulation tab in the template document and stores the fragment, database instance ID, and unique ID for the source document in the corresponding fields on the RenderXFDL form. The unique ID will control the visibility of the lines containing the prepopulation scripts (see the hide formulas applied in form RenderXFDL).

Example 9-10 Creating the missing sub, createXMLFragment

```

Sub createXMLFragment(template As NotesDocument, doc As NotesDocument, Prepop As
NotesRichTextItem, no As Integer)
    On Error Goto errorhandler
    Dim fieldsuffix As String
    Dim prepopIDFormula As Variant
    Dim prepopID As Variant
    Dim prepopDataInstance As String
    Dim prepopFields As Variant
    Dim instance As String
    Dim ExportFields List As String

    Dim keydoc As NotesDocument
    fieldSuffix = ""
    If no <> 0 Then fieldSuffix = "_" & no

    MsgBox "PREPOP: " & no

    'get prepopulation settings (remove newline from source code)
    prepopIDFormula = Evaluate({@replacesubstring(prepopIDFormula} + fieldSuffix +
    {;@newline;" "}), template)
    'MsgBox prepopIDFormula(0)
    prepopIDFormula = prepopIDFormula(0)
    MsgBox "@Formula: " + prepopIDFormula

    If prepopIDFormula<>"" Then
        'OK - we will search for the source document
        'get doc id
        prepopID = Evaluate(prepopIDFormula, doc)
        prepopID = prepopID(0)
        MsgBox "ID: " & prepopID
        'get instance name to create
        prepopDataInstance = template.GetItemValue("prepopDataInstance" + fieldSuffix )(0)
        'get field attribute namens (mapping)
        prepopFields = template.GetItemValue("prepopFields" + fieldSuffix)
        Forall v In prepopFields
            'v contains a string as NotesFieldName#XMLElementName
            ExportFields(Strrightback("#"+v,"#")) = Strleft(v+"#", "#")
        End Forall

        'find source doc in target DB by id
        On Error Resume Next
        Dim sourcedb As NotesDatabase
        Dim path As String
    
```

```

path = template.GetItemValue("prepopSourceDb" + fieldSuffix )(0)
If path = "" Then
    'current database
    Set SourceDB = template.ParentDatabase
Else
    'other database
    Set sourcedb = New NotesDatabase("",path)
    If Not sourceDB.IsOpen Then Call sourcedb .Open("", "")
    If Not sourceDB.IsOpen Then Error 1000, "SourceDB not found: " & path
End If
MessageBox "sourceDB for prepop: " + sourceDB.Title

'get the target document by key
Set keyDoc = SourceDB.GetDocumentByUNID(prepopID)
On Error Goto errorHandler
If Not keyDoc Is Nothing Then
    MessageBox "got UNID: " + prepopID
    'get the prepop rich text field in the rendering form
    If Not doc.HasItem("Prepop" + fieldSuffix ) Then
        Set prepop = doc.CreateRichTextItem("Prepop" + fieldSuffix )
    Else
        Set Prepop=doc.GetFirstItem("Prepop" + fieldSuffix )
    End If

    'read source doc and create data instance
    instance = "<" + prepopDataInstance + ">" + Chr$(10)
    Forall it In ExportFields
        If keydoc.HasItem(it) Then
            Dim tmpv As Variant
            tmpv = keydoc.GetItemValue(it)(0)
            tmpv = encode(Cstr(tmpv))
            instance = instance + "<" + Listtag(it) + ">" + tmpv + "</" + Listtag(it) +
">" + Chr$(10)
            MessageBox "<" + Listtag(it) + ">" + tmpv + "</" + Listtag(it) + ">"
        End If
    End Forall
    instance = instance + "</" + prepopDataInstance + ">"
    'store xml fragment for data instance to the form field prepop_x
    Call Prepop.AppendText(instance)
    MessageBox instance
    Call doc.replaceitemvalue("prepopDataInstance" + fieldsuffix, prepopDataInstance)
End If
Else
    'MessageBox "NO repopIDFormula"
End If
Call doc.replaceItemValue("PrepopulationDocUNID" + fieldsuffix, prepopID) ' store id to
for (for hide formula use)
Exit Sub
errorHandler:
    MessageBox "Error: " & Err() & " in " & CONSTLIBNAME & "." & Lsi_info(2) & " line " &
Er1()
    MessageBox Error$
End Sub

```

The Agent is ready to run. Save it.

Note: The created document is never saved — it will exist only as an in-memory copy and disappear whenever the session ends. To store a document, we will wait for a submission from the browser. The related code is discussed in the next section.

Now we have to create two more components to make the new form creation possible:

- ▶ A template view for the browser showing available templates with specific URLs
- ▶ At least one template document in the database

First we will create the template view.

Template View (WEB)

Create a new view in Domino Designer available for browser only with two columns:

Name: Templates (WEB) prepop

Alias: TemplatesPrepop

Selection formula: `SELECT Form="FormTemplate"`

Column 1 : Field PFormName; plain sort

Column 2: @Formula

```
_db := @ReplaceSubstring(@Trim(@DbName);"\\";"");
```

```
"<A href=\"/" + _db + "/RenderXFDL" +  
?ReadForm&FormName="+@URLEncode("Domino";PFormName) +  
&Ind=" + @Unique + "\">Click here to get a copy of "+PFormName+"</A>"
```

The second column will create a link like this:

```
http://vmforms1.cam.itso.ibm.com/redpaper/WPForms.nsf/RenderXFDL?ReadForm&FormName=  
=MyFormTemplate&Ind=VMFS-6NDPF6
```

This will open in Template Database a new document using RenderXFDL form. This runs the RenderXFDLPrepop agent to fill the created document with the values. The parameters specify:

- ▶ The template to use is: (*FormName=MyFormTemplate*).
- ▶ The *Ind* parameter will change for each view entry, preventing possible caching on the server.

The initiation agent will extract the FormName parameter and create the appropriate field settings in the RenderXFDL form.

Create another view for this form to show the available templates for the Notes client. Hide this view for Web clients. There are no special settings for this view. Just make it convenient for you. It could be as simple as shown in Figure 9-20.

Form
1. Purchase Order (AR) (WS for Emp data + Items, dynamic choices load)
2. PO Sample (Wizard like)
3. Mortgage Sample (WFS) - Domino independent form copied from Server sample
Redpaper Form Stage 1
Redpaper Form Stage 2 V36

Figure 9-20 Template view in Notes client

Creating a template document

The last step before the first test is creating at least one template document.

Open the Template Database in the Notes client / Template view.

1. Create a new template document. Choose from the menu **Create - FormTemplate**.
2. Give it a name (such as **Forms Integration Template 1**).
3. Attach a form template used in J2EE stage 2 (or choose the appropriate form template from the redpaper resource zip).
4. Enter prepopulation settings for Employee data prepopulation as shown below.

Make sure that the assigned Names (Instance ID, Notes field names, element names in the data instance, match exactly the names used in the Notes / XFDL Template.

The assigned formula should find the employee data in the Repository database (Figure 9-21).

Form Template

Name:

Forms Integration Template 1

Attached template:

Redpaper_Forms_Sample_v39.xfdl

Data Instance 1 | Data Instance 2 | Data Instance 3 | Data Instance 4 | Text Parsing

Propopulation Instance 1

Path to prepopulation source database

redpaper\wpformsrep.nsf

@Formula for prepopulation document id

_id := @DbLookup(;;;"redpaper\wpformsrep.nsf";"luEmployeesByNotesName";@UserName;2);

@If(

@IsError(_id);"";

_id

Data instance to prepopulate (tag name and id for datainstance):

FormOrgData

Mapping (List of source fieldnames#target elements in data instance)

ORG_FIRSTNAME#FirstName

ORG_LASTNAME#LastName

ORG_ID#ID

ORG_CONTACTINFO#ContactInfo

ORG_MGR#Manager

Figure 9-21 Template name, attachment, and prepopulation for employee data

The prepopulation settings on this tab would make up an XML fragment for prepopulation like this (Example 9-11).

Example 9-11 The prepopulation fragment for data instance FormOrgData

```
<FormOrgData>
  <FirstName>John</FirstName>
  <LastName>Miller</LastName>
  <ID>1010</ID>
  <ContactInfo>jr@itso.com</ContactInfo>
  <Manager>1031</Manager>
</FormOrgData>
```

5. Switch to tab Data Instance 2 and enter the following settings. They will increment the order ID parameter and prepare the new order ID XML fragment (Figure 9-22).

Data Instance 1	Data Instance 2	Data Instance 3	Data Instance 4	Text Parsing
-----------------	-----------------	-----------------	-----------------	--------------

Propopulation Instance 2

Path to prepopulationsource database

@Formula for prepopulation document id
 _UNID := @dblookup["";"NoCache";@DBName;"Parameters";"OrderCounter".4];
 _ID := @TextToNumber("0"+@Text(@GetDocField[_UNID;"PAR_NumValue"])) + 1;
 @SetDocField[_UNID;"PAR_NumValue";_ID];
 _UNID

Data instance to prepopulate (tag name and id for datainstance):
OrderNumber

Mapping (List of source fieldnames#target elements in data instance)
PAR_NumValue#ORD_ID

Figure 9-22 Increment order number and create XML fragment for order number prepopulation

Settings on this tab would create an XML fragment for prepopulation like this (Example 9-12).

Example 9-12 XML fragment for order number prepopulation

```
<OrderNumber>
  <ORD_ID>100085</ORD_ID>
</OrderNumber>
```

Compare both XML structures for data instance prepopulation with the structure of the data instances in the XFDL template. We relate always to the first child element of the data instance (<FormOrgData> and <OrderNumber>). These names and the child element names must match case sensitive (Example 9-13).

Example 9-13 XML fragments representing the data instances to prepopulate in XFDL template

```
<xforms:instance xmlns="" id="FormOrgData">
  <FormOrgData>
    <FirstName></FirstName>
    <LastName></LastName>
    <ID></ID>
    <ContactInfo></ContactInfo>
    <Manager></Manager>
  </FormOrgData>
</xforms:instance>
.....
<xforms:instance xmlns="" id="OrderNumber">
  <OrderNumber>
    <ORD_ID></ORD_ID>
  </OrderNumber>
</xforms:instance>
```

In summary, the HTML page created for a new form with prepopulation should look like this (Example 9-14).

Example 9-14 Complete HTML page prepopulating an XFDL form

```
<BODY>
<HTML><BODY>
<OBJECT id="Object2" height="2000" width="980" border="1"
classid="CLSID:354913B2-7190-49C0-944B-1507C9125367">
<PARAM NAME="XFDLID" VALUE="XFDLData">
<!-- red lines are hidden, when no prepopulation data available (PrepopulationDocUNID="")
-->
<PARAM NAME="instance_1" VALUE="FormOrgData FormOrgData replace [0]">
```

```

<PARAM NAME="instance_2" VALUE="OrderNumber OrderNumber replace [0]">
</OBJECT>
<SCRIPT id=FormOrgData type="application/vnd.xfdl; wrapped=comment">
<!--<FormOrgData>
<FirstName>John</FirstName>
<LastName>Bergland&lt;öüß&gt;&amp;ßfß</LastName>
<ID>1010</ID>
<ContactInfo>jr@itso.com</ContactInfo>
<Manager>1031</Manager>
</FormOrgData--></SCRIPT>
<SCRIPT id=OrderNumber type="application/vnd.xfdl; wrapped=comment">
<!-- <OrderNumber>
<ORD_ID>100085</ORD_ID>
</OrderNumber--></SCRIPT>
<SCRIPT language="XFDL" id="XFDLData" type="application/vnd.xfdl; wrapped=comment">
<!-- <?xml version="1.0" encoding="ISO-8859-1"?>
<XFDL xmlns="http://www.PureEdge.com/XFDL/6.5"
xmlns:cm="http://www.PureEdge.com/idk/ibmcm/1.0"
xmlns:custom="http://www.PureEdge.com/XFDL/Custom"
xmlns:designer="http://www.PureEdge.com/Designer/6.1"
xmlns:pecs="http://www.PureEdge.com/PECustomerService"
xmlns:xfdl="http://www.PureEdge.com/XFDL/6.5"
xmlns:xforms="http://www.w3.org/2003/xforms">
  <globalpage sid="global">
    <global sid="global">
      <vfd_date>9/3/2006</vfd_date>
      <formid>
        <version>14.182.0</version>
      </formid>
      <custom:formid></custom:formid>
      <xmlmodel>
        <instances>
          <xforms:instance xmlns="" id="FormOrgData">
            <FormOrgData>
              <FirstName></FirstName>
              <LastName></LastName>
              <ID></ID>
              <ContactInfo></ContactInfo>
              <Manager></Manager>
            </FormOrgData>
          </xforms:instance>
          .....
        </instances>
      </xmlmodel>
      .....
    </global>
  </globalpage>
  <page sid="PAGE1">
    .....
  </page>
</XFDL> -->
</SCRIPT>
<BODY>

```

Switch the TextParsing tab and enter all necessary changes there (Figure 9-23). In our example we will change the following items:

- Change the end point URLs in stored WSDL files to make them running with our Domino server (initially they would point to the J2EE based Web service used in stage 2).

- Change the end point URL for the Submit button to hit our Domino server (initially it would point to the J2EE based servlet used in stage 2).
- Change the form title (to whatever you would like).

Figure 9-23 Text parsing settings

Here are some comments regarding the assigned new submission URL:

```
http://vmforms1.cam.itso.ibm.com/servlet/XFDLServletRedPaper?action=store&url=
http://vmforms1.cam.itso.ibm.com/redpaper/WPForms.nsf
```

The URL points to a servlet running on the Domino server. We will create this servlet in the next section. As a parameter (url=) we pass some additional information with the submission action. The servlet will use the URL parameter to determine the next page to display after a successful submission. In the example above we specified the Template Database default page as a Success URL.

Tip: To match exactly the text fragments stored in the form and look up the URLs stored currently in your XFDL form, look up the following places:

The location attributes in <wsdl:service> elements for all Web services:

```
<wsdl:service name="ProductCatalog">
  <wsdl:port binding="impl:ProductCatalogPortSoapBinding" name="ProductCatalogPort">
    <wsdlsoap:address
      location="http://vmforms1.cam.itso.ibm.com:8085/WpfWsCatalogT/services/ProductCatalog
      Port"></wsdlsoap:address>
    </wsdl:port>
  </wsdl:service>
  .....
<wsdl:service name="CustomerInfoService">
  <wsdl:port binding="impl:WPFormsCustSoapBinding" name="WPFormsCust">
    <wsdlsoap:address
      location="http://vmforms1.cam.itso.ibm.com:8085/WpfWsCustomerT/services/WPFormsCust">
    </wsdlsoap:address>
    </wsdl:port>
  </wsdl:service>
```

The URL stored for the Submit button:

```
<url>
  <ae>http://vmforms1.cam.itso.ibm.com:9080/WPFormsRedpaper/SubmissionServlet?action
  =store</ae>
</url>
```

Creating used parameter documents

In the Template Database, create the following parameter documents (Figure 9-24).

Parameter	Parameter
Parameter Name OrderCounter	Parameter Name ServerUrl
Num Value 100099	Num Value
Text Value Counter for next order number	Text Value http://vmforms1.cam.itso.ibm.com
Document Access: *	Document Access: Administrators

Figure 9-24 Parameter documents to create

Make sure that the assigned URL in the ServerUrl document will hit your Domino server when entered in the browser.

Allow all authenticated users to increment the OrderCounter parameter (field Document Access contains a wild card or any group for all authenticated users).

Now all prerequisites are ready to run the example.

Prepopulation test

Let us consider the first test. To get more familiar with the created scenario, have a second look at the overview picture (Figure 9-25).

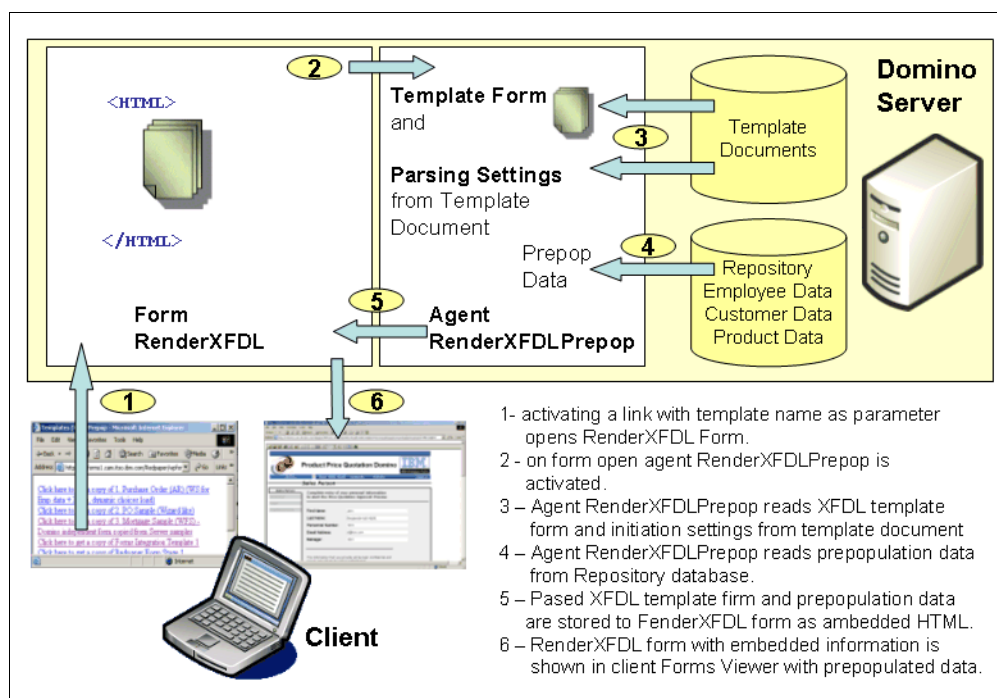


Figure 9-25 High level flow in Domino new document scenario

Figure 9-26 provides a more technical view showing all main components we have built for now.

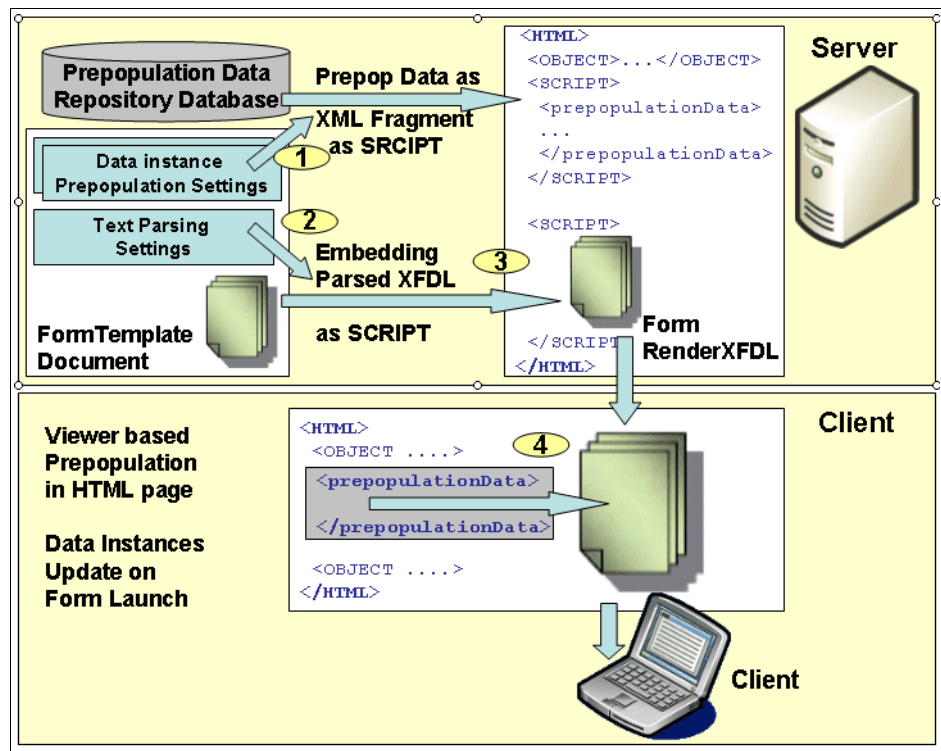


Figure 9-26 Domino prepopulation flow (component details)

To start the test, open the created templatesPrepop view in Template Database with MSIE 6 browser and authenticate (Figure 9-27) with one of the created test users in the Repository database:

<http://<yourservername>/redpaper/WPForms.nsf/TemplatesPrepop>

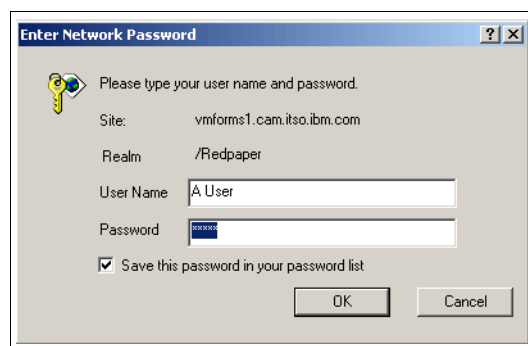


Figure 9-27 User authentication

The new template view should appear (Figure 9-28).

[Click here to get a copy of 1. Purchase Order \(AR\) \(WS for Emp data + Items, dynamic choices load\)](#)
[Click here to get a copy of 2. PO Sample \(Wizard like\)](#)
[Click here to get a copy of 3. Mortgage Sample \(WFS\) - Domino independent form copied from Server samples](#)
[Click here to get a copy of Forms Integration Template 1](#)
[Click here to get a copy of Redpaper Form Stage 1](#)
[Click here to get a copy of Redpaper Form Stage 2 V36](#)

Figure 9-28 View TemplatePrepop

Depending on the entered template documents, there can be other entries in the view. Check the generated links when moving the mouse over one of these links. They should look like:

`http://vmforms1.cam.itso.ibm.com/redpaper/WPForms.nsf/RenderXFDL?ReadForm&FormName=Redpaper%20Form%20Stage%201&Ind=VMFS-6NDPF5`

Make sure all required components are OK (Table 9-6).

Table 9-6 Components creating new form link

Component	Description
<code>http://vmforms1.cam.itso.ibm.com</code>	URL to the Domino server
<code>redpaper/WPForms.nsf</code>	Path to Template Database
<code>RenderXFDL?ReadForm</code>	Form name RenderXFDL and ReadForm URL command.
<code>FormName=Redpaper%20Form%20Stage%201</code>	Parameter FormName referencing the form name when opening RenderXFDL form
<code>Ind=VMFS-6NDPF5</code>	Arbitrary parameter to prevent caching changing for each link in the view

Click the link, **Click here to get a copy of Forms Integration Template 1**, for the document created in this chapter. A new page should pop up and show the Forms Viewer embedded in an HTML page (Figure 9-29).

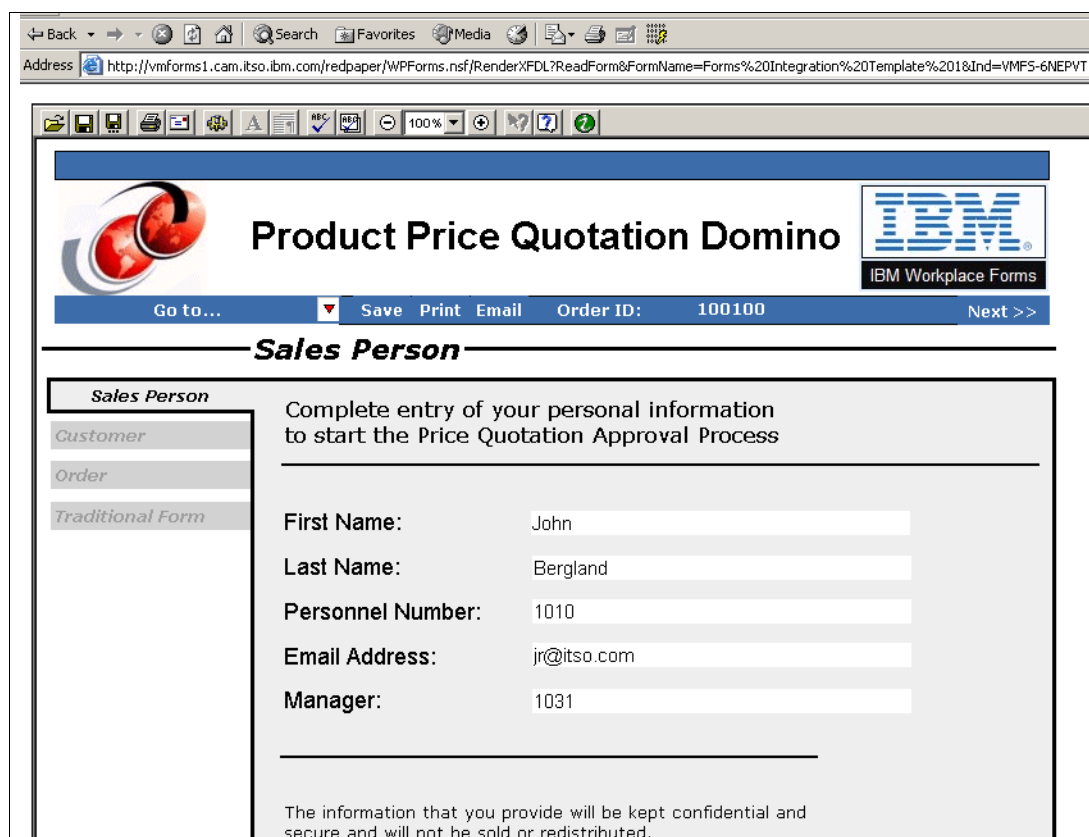


Figure 9-29 Viewer with prepopulated form embedded in an HTML page

You will see the small space between the browser's Address Bar and the Viewer. Right-click there and review the source code of the page. You should find all of the components created in this chapter, like this (see Example 9-15).

Example 9-15 Reviewing the source code of the page

```
<BODY>
<HTML><BODY>
<OBJECT id="Object2" height="2000" width="980" border="1"
classid="CLSID:354913B2-7190-49C0-944B-1507C9125367">
<PARAM NAME="XFIDLID" VALUE="XFIDLData">
<!-- red lines are hidden, when no prepopulation data available (PrepopulationDocUNID="")
-->
<PARAM NAME="instance_1" VALUE="FormOrgData FormOrgData replace [0]">
<PARAM NAME="instance_2" VALUE="OrderNumber OrderNumber replace [0]">
</OBJECT>
<SCRIPT id=FormOrgData type="application/vnd.xfdl; wrapped=comment">
<!--<FormOrgData>
<FirstName>John</FirstName>
<LastName>Bergland</LastName>
<ID>1010</ID>
<ContactInfo>jr@itso.com</ContactInfo>
<Manager>1031</Manager>
</FormOrgData>--></SCRIPT>
<SCRIPT id=OrderNumber type="application/vnd.xfdl; wrapped=comment">
```

```

<!-- <OrderNumber>
<ORD_ID>100100</ORD_ID>
</OrderNumber>--></SCRIPT>
<SCRIPT language="XFDL" id="XFDLData" type="application/vnd.xfdl; wrapped=comment">
<!-- <?xml version="1.0" encoding="ISO-8859-1"?>
<XFDL xmlns="http://www.PureEdge.com/XFDL/6.5"
xmlns:cm="http://www.PureEdge.com/idk/ibmcm/1.0"
xmlns:custom="http://www.PureEdge.com/XFDL/Custom"
xmlns:designer="http://www.PureEdge.com/Designer/6.1"
xmlns:pecs="http://www.PureEdge.com/PECustomerService"
xmlns:xfd="http://www.PureEdge.com/XFDL/6.5"
xmlns:xforms="http://www.w3.org/2003/xforms">>
  <globalpage sid="global">
    <global sid="global">
.....

```

Tip: If there are any errors at document opening, try the following suggestions:

1. Make a copy of the template document in the Notes client.
2. Give it a new name (such as TEST).
3. Remove all information from the prepopulation tabs (no data instances to prepopulate, no text parsing).
4. Save the form.
5. Open the TemplatesPrepop view in the browser as before.
6. Do a browser refresh, if the new form is not visible.
7. Try to make a new copy in the browser using this new test form.
8. This should work — but if not:
 - a. Inspect the rendered HTML.
 - b. Look for missing or additional fragments.
 - c. Review the RenderXFDL form design and RenderXFDLPrepop agent design to make things work well.
9. If this works, add the prepopulation information step-by-step, back to the TEST form document. After completing one prepopulation tab, save the form and try to open it in the browser. The following errors for prepopulation are common:
 - a. The @Formula reading the UNID from the source document is not OK or the view used for lookup does not match — repair this.
 - b. Data instance names do not match — double-check for this.
 - c. Field names or element names assigned in the Mapping field are misspelled — triple check for this.
 - d. Make sure the agent has sufficient execution rights and is running as a Web user (otherwise we could not get the username from the session).
10. Now enter text parsing settings step-by-step. Special characters in the text parsing tab will break the form:
 - a. Enter the available replacements one after another and do a test after each new entry.
 - b. After each new entry, save the form and do a new test in the browser.

Finally, after this troubleshooting, everything should work properly now.

9.5.3 Template Database: Receiving submitted forms in Domino

A submitted form will be passed to the server as content in a POST requests. Domino can receive POST requests running an agent and retrieve the content using CGI variables. Unfortunately there are limitations to the size of received information. Actually a CGI variable can store messages with up to 30.000 characters. For bigger submits other techniques must be engaged.

This is the point where a servlet comes into the game. Basically we will have two benefits from using a servlet to receive POST messages:

- ▶ There is an unlimited length for the contained data.
- ▶ Using Java, we can utilize all benefits coming with the Forms API.

Figure 9-28 shows the main components engaged in receiving submitted documents.

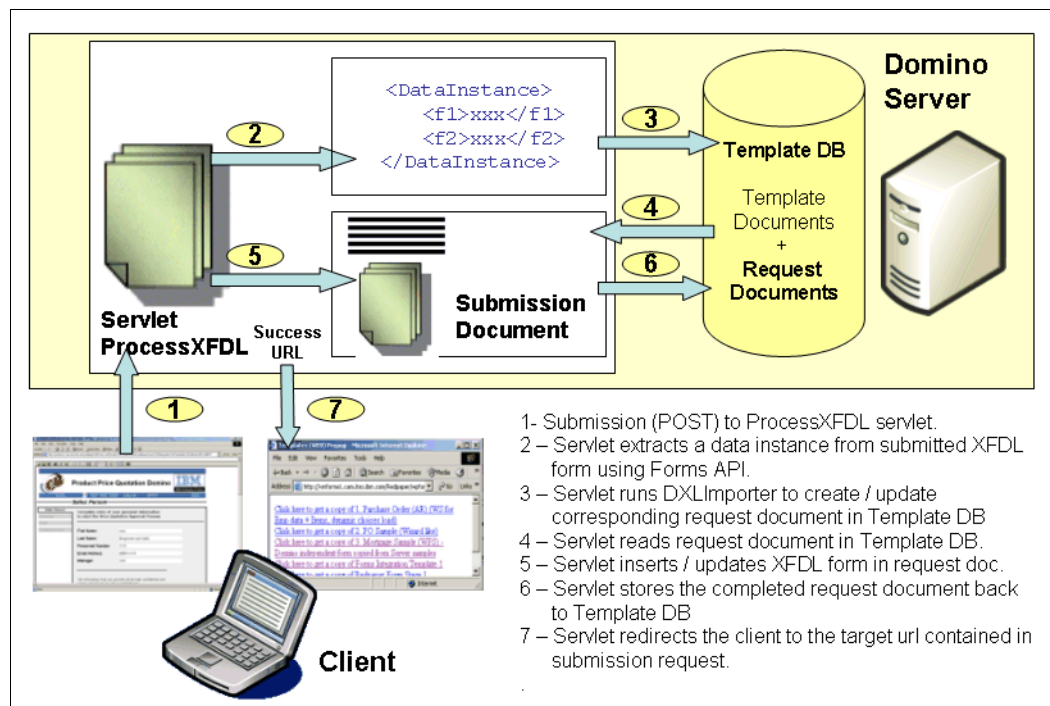


Figure 9-30 Flow on form submission to Domino

To build this scenario, we have to create the following components:

- ▶ The receiving servlet named ProcessXFDL and accordingly the deployment information (setvlets.properties file on Domino server)
- ▶ A form in the Template Database capable of storing the incoming documents along with the extracted metadata (QuotationRequest documents)
- ▶ A maintenance view for these documents
- ▶ A lookup view for the servlet to find the corresponding request document for the incoming XFDL form on the server (if there is any)

Now, let us build these components.

ProcessXFDL servlet receiving incoming XFDL documents

Building the servlet can be done using eclipse, RAD6, or any other Java development environment. The servlet is contained in a single Java file. There is some basic information needed to configure the servlet. This can be done in the servlet.properties file, or it can be read from the incoming XFDL document or from any other datasource). What portion of information comes from what data source will depend on the environment and design guides in place. In this redpaper we will read all information from servlet.properties file.

Note: The servlet.properties file can store multiple virtual addresses for the same servlet and assume, for each virtual address, different parameters. This makes it possible to use one generic servlet processing different XFDL forms.

The servlet should process the following tasks:

- ▶ Read setup information (init method). Since our setup information is static (stored in the servlet.properties file), we can read it in the init method.
 - Identify the database and form and field name to store the request document.
 - Connect to the database (username/password).
 - Detect the data instance ID to extract from the incoming XFDL form.
- ▶ Receive incoming POST messages (this will be done in the doPost method):
 - Extract the Success URL from the incoming message (URL parameter).
 - Read the incoming XFDL form.
 - Extract the assigned data instance from the XFDL file using the Forms API.
 - Read the custom ID from XFDL using the Forms API. This will be the key to search for a related request document in the target database.
 - Prepare the XML stream for the DXLImporter for document creation/update with the extracted data instance.
 - Run the DXLImporter to create/update the request document.
 - Store the received XFDL file to the just-created/updated request document as an attachment.
 - Submit the Success URL to the browser.
- ▶ Receive GET messages (for debug purposes only). The doGet method will submit the settings read from the servlet.properties file rendered in an HTML page.

The full servlet code is given in Example 9-16. For details, see the in-line comments.

Example 9-16 Listing of full servlet code

```
/**
 * @version 1.0
 * @author Cees Vandewoude, Andreas Richter
 * @comment Improved simple servlet version for IBM Workplace Forms Integration
 *         Redpaper
 */

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import lotus.domino.*;
import java.util.Vector;
```

```

import com.PureEdge.DTK;
import com.PureEdge.xfdl.FormNodeP;
import com.PureEdge.xfdl.XFDL;
import com.PureEdge.IFSUserData;
import com.PureEdge.IFSUserDataHolder;
import com.PureEdge.error.UWIException;
import com.PureEdge.IFSSingleton;

public class ProcessXFDL extends javax.servlet.http.HttpServlet implements
    javax.servlet.Servlet {

    // form name or alias to store the xfdl document and data
    String form;

    // relative path to target database to store the xfdl document and data
    String dbPath;

    // field to store the request
    String targetField;

    // file name for the created attachment containing the POST data
    String fileName;

    // debug mode (will print to console, if set to "on"
    static String debugMode;

    // content type for incoming message. Will be assigned to the mime entity
    String contentType;

    // username to create domino session and access target database
    String userName;

    String password;

    //Data instance to retrieve
    String instanceID;

    // servlet name
    static String sv;

    // initi params for Forms API
    private static FormNodeP theForm;

    static DxlImporter importer = null;

    public void init(ServletConfig config) {
        // Read servlet configuration parameters
        try {
            super.init(config);

            sv = this.getClass().getName();
            //read values from servlets.properties
            debugMode = config.getInitParameter("debugMode");
            if (debugMode == null)
                debugMode = "on";
            if (!(debugMode.equals("on")))
                debugMode = "off";
            debugOut(" debugMode: '" + debugMode + "'");

            dbPath = config.getInitParameter("dbPath");

```

```

        debugOut(" dbPath: '" + dbPath + "'");

        form = config.getInitParameter("form");
        if (form == null)
            form = "";
        debugOut(" form for new documents: '" + form + "'");

        targetField = config.getInitParameter("targetField");
        if (targetField == null)
            targetField = "Body";
        if (targetField.equals(""))
            targetField = "Body";
        debugOut(" field name for request: '" + targetField + "'");

        userName = config.getInitParameter("userName");
        if (userName == null)
            userName = "";
        debugOut(" userName: '" + userName + "'");

        password = config.getInitParameter("password");
        if (password == null)
            password = "";
        debugOut(" password: '" + password + "'");

        fileName = config.getInitParameter("fileName");
        if (fileName == null)
            fileName = "post#ID#.txt";
        if (fileName.equals(""))
            fileName = "post#ID#.txt";
        debugOut(" file name for POST attachment: '" + fileName + "'");

        contentType = config.getInitParameter("contentType");
        if (contentType == null)
            contentType = "text/plain";
        if (contentType.equals(""))
            contentType = "text/plain";
        debugOut(" contentType for created attachment: '" + contentType
            + "'");

        instanceID = config.getInitParameter("dataInstanceID");
        if (instanceID == null)
            instanceID = "";
        debugOut(" instanceId to retrieve: '" + instanceID + "'");

        System.out.println(sv + " initialized");

    } catch (javax.servlet.ServletException e) {
        e.printStackTrace();
    }
}

public void doGet(HttpServletRequest request, HttpServletResponse response) {
    //this method is optional for basic servlet functionality in this
    // context.
    //we will return only some state information - this option is useful
    // for debugging
    try {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.print("<H2>" + sv + "</H2><BR>");
    }
}

```

```

        + ": The url must be called with a POST action containing a valid
XFDL document <BR>");
        out.print("A GET request does not accept any parameters and returns the actual
servlet state only <BR>");
        out.print("<BR>");
        out.print("<BR>");
        out.print("<H2>Servlet Status and Statistics Report" + "</H2>");
        out.print("ServletClass: " + sv + "<BR>");
        out.print("<H3>Session parameters currently set for this web service proxy
servlet in servlets.properties</H3>");
        out.println(" dbPath: '" + dbPath + "'" + "<BR>");
        out.println(" userName: '" + userName + "'" + "<BR>");
        //out.println(" password: '" + password + "'" + "<BR>");
        out.println(" form for new documents: '" + form + "'" + "<BR>");
        out.println(" target field: '" + targetField + "'" + "<BR>");
        out.println(" attachment file name pattern: '" + fileName + "'" + "<BR>");
        out.println(" content type for attachment: '" + contentType + "'" + "<BR>");
        out.println(" dataInstanceID for value retrieval: '" + instanceID + "'" +
"<BR>");
        out.println(" debugMode: '" + debugMode + "'" + "<BR>");

    } catch (IOException e) {
        // Something went wrong.
        e.printStackTrace();
    }
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException {
    // values from the calling url (parameters)
    // url to process, when the transaction is finished
    String nextUrl;

    response.setContentType("text/html");
    ServletOutputStream out = response.getOutputStream();
    ServletInputStream theStream = request.getInputStream();
    boolean isNew = true;
    //try to get the next url from the request for client redirection
    nextUrl = request.getParameter("url");
    if (nextUrl == null)
        nextUrl = "";
    debugOut(" next url: '" + nextUrl + "'");

    out.println("<HTML><B>Submitting eform</B><BR>");

    try {
        //Initialize Forms API
        DTK.initialize("WPForms Integration Redpapper", "1.0.0", "6.5.0");

        XFDL theXFDL;
        theXFDL = IFSSingleton.getXFDL();
        //get the form in a String
        theForm = theXFDL.readForm(theStream, XFDL.UFL_SERVER_SPEED_FLAGS);
        debugOut(" theForm created");

        //get entire form as String
        String formString = getFormAsString(theForm);
        debugOut(" form stored to formString");

        //here we can read additional xfdl elements like this

```

```

// form=theForm.getLiteralByRefEx(null,"global.global.custom:dominoForm",
// 0, null, null);

//read xfdl form id to check, wether we already have the document
// in the database
String formID = theForm.getLiteralByRefEx(null,
    "global.global.custom:formid", 0, null, null);
if (formID == null)
    formID = ""; //ok no ID given - we can only insert a new doc
debugOut(" Form ID: " + formID);

//prepare piped streams to redirect xfdl data to inputstream
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream(po);
//extract data instance from xfdl as stream
theForm.extractInstance(instanceID, null, null, po, 0, null, "[0]",
    null);
debugOut(" data instance extracted");
po.flush();
//read extracted instance to String piStr
String piStr = getString(pi);
debugOut(" stream: " + piStr);

//destroy form to free memory
theForm.destroy();
debugOut(" form destroyed");

//OK - now we can analyse the extracted data and store the form in
// a document
try {
    //set up notes connection
    NotesThread.sinitThread();
    Session s = NotesFactory.createSession();
    debugOut(" Notes session created");
    Database db = s.getDatabase(null, dbPath);
    View vw = null;
    Document doc = null;
    Document docAtt = null;
    debugOut(" Notes db found");

    //get target db replica ID for DXLImporter
    String replID = db.getReplicaID();
    //search for a document with the assigned formID in the
    // database
    String UNID = "";
    if (!(formID.equals(""))){
        vw = db.getView("AllByXfdlId"); //special server view by
        // formID
        doc = vw.getDocumentByKey(formID, true);
        if (!(doc == null)) {
            //we have found a document -> update it
            UNID = doc.getUniversalID();
            isNew = false;
            //first update the attachment - update after importer
            // action will destroy the field update
            createAttachment(doc, fileName, targetField, s,
                theStream, contentType);
            doc.save(true);
        }
    }
}

```

```

// prepare DXLImporter for field update
Stream stream = s.createStream();
//createImporterXML composes an xml string for DXLImporter to
// insert/update a doc
debugOut(" import XML: "
    + createImporterXML(piStr, form, replID, UNID));
stream.write(createImporterXML(piStr, form, replID, UNID)
    .getBytes());
debugOut(" Stream filled");
importer = s.createDxlImporter();
//this will allow us to insert new documents, ib replica/UNID
// does not match
importer.setReplicaRequiredForReplaceOrUpdate(false);
importer

.setDocumentImportOption(DxlImporter.DXLIMPORTOPTION_UPDATE_ELSE_CREATE);
debugOut(" DXLImporter created");
//process import / update
importer.importDxl(stream, db);
debugOut(" document imported/updated");
stream.close();

if (isNew) {
    //New doc -> find the new document and attach the xfdl file
    String id = importer.getFirstImportedNoteID();
    debugOut(" first node requested: " + id);

    boolean found = false;
    docAtt = db.getDocumentByID(id);
    //may be we created multiple items
    //this can occur, if the extracted data instance was not OK
    while ((docAtt != null) && (found == false)) {
        if (docAtt.getItemValueString("Form").equals(form)) {
            found = true;
        } else {
            debugOut(" wrong form: "
                + docAtt.getItemValueString("Form"));
            id = importer.getNextImportedNoteID(id);
            debugOut(" next node requested: " + id);
            docAtt = null;
            docAtt = db.getDocumentByID(id);
        }
    }
}
if (docAtt != null) {
    //Now attach the xfdl file, if we found a new doc
    debugOut(" node accessed: " + docAtt.getUniversalID());
    createAttachment(docAtt, fileName, targetField, s,
        theStream, contentType);
    //store the formID to the document to find it for future updates
    docAtt.replaceItemValue("XFDLID", formID);
    //docAtt.computeWithForm(true, true);
    if (docAtt.save(true)) {
        //out.println("<br>New Document Saved");
        //out.println("<br>Thanks for submitting this eform!");
        out.println(getRespMessage(nextUrl) );
        //doc.recycle();
    } else {
        debugOut(" node not found.");
        out.println("<br>Unable to save document<br><br>");
    }
}

```

```

        out.println("<a href=\"" + nextUrl+"\">Return to application home
page</a>");
        out.println("</body></html>");
    }

    }else {
        //out.println("<br>Updated Document Saved");
        //out.println("<br>Thanks for submitting this eform!");
        out.println(getRespMessage(nextUrl) );
    }

    //docAtt.recycle();
    vw.recycle();
    db.recycle();
    s.recycle();

}

catch (Exception e) {
    e.printStackTrace();
}

}

catch (Exception ex) {
    ex.printStackTrace();
} finally {
    NotesThread.stermThread();
}

} // end of method Post

//support methods
//read xdf1 form from PormNodeP as String
private static String getFormAsString(FormNodeP theForm)
    throws UWException, IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    theForm.writeForm(baos, null, 0);
    baos.flush();
    return baos.toString();
}

//search and replace Strings
private static String replaceSubString(String inputString,
    String searchString, String replaceString) {

    int i = inputString.indexOf(searchString);
    if (i == -1) {
        return inputString;
    }

    String r = "";
    r += inputString.substring(0, i) + replaceString;
    if (i + searchString.length() < inputString.length()) {
        r += replaceSubString(inputString.substring(i
            + searchString.length()), searchString, replaceString);
    }

    return r;
}

```

```

    }

    //create an xml instance accepted for DXLImporter for insert and update
    private static String createImporterXML(String piStr, String form,
        String replID, String UNID) {
        //this is a quick and dirty code to transform a xfdl data instance
        //into an DXLImporter compatible xml fragment to insert/updat one dokument
        //string fragments not needed will set in comments.
        /*
        we get this (piStr):
        <FormOrderData xmlns="" xmlns:cm="http://www.PureEdge.com/idk/ibmcm/1.0"
xmlns:custom="http://www.PureEdge.com/XFDL/Custom"
xmlns:designer="http://www.PureEdge.com/Designer/6.1"
xmlns:pecs="http://www.PureEdge.com/PECustomerService"
xmlns:xfdl="http://www.PureEdge.com/XFDL/6.5" xmlns:xforms="http://www.w3.org/2003/xforms">
        <ID>100032</ID>
        <CustomerID></CustomerID>
        <Amount>2184.00</Amount>
        <Discount>0</Discount>
        </FormOrderData>

        we will return thomething like this:

        <database version="7.0">
        <document form='QuotationRequest' version='7.0' replicaid='C125712B00718095'>
        <noteinfo noteid='' unid='AFE0838101C7DB16C125713E00205F6F' sequence='' />
        <!-- -->
        <item name="ID"><text>100032</text></item><!--ID"><text> -->
        <item name="CustomerID"><text></text></item><!--CustomerID"><text> -->
        <item name="Amount"><text>2184.00</text></item><!--Amount"><text> -->
        <item name="Discount"><text>0</text></item><!--Discount"><text> -->
        </document>
        </database>

        A better method for this would be an XSLT transformation
        */

        //<database> tag
        String s1 = "<database version=\"7.0\">";
        //<document> tag - mandatory use ' - not " as string quotes
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        //assign the correct db replica id for document update!
        s1 = s1 + "<document form='" + form + "' version='7.0' replicaid='"
            + replID + "'>";
        //<noteInfo> tag - mandatory use ' - not " as string quotes
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        //assign the correctdocument UNID id for document update!
        s1 = s1 + "<noteinfo noteid='' unid='" + UNID + "' sequence='' />";
        //comment tag - necessary because in item elements we will begin with "-->"
        s1 = s1 + "<!--";
        //preconfigure end tags
        String s2 = "--></document></database>";
        String instanceXML = piStr;

        //first - stripe data instance tag / end tag
        instanceXML = instanceXML.substring(instanceXML.indexOf(">") + 1,
            instanceXML.length());
        instanceXML = instanceXML.substring(0, instanceXML.lastIndexOf("</>"));

```

```

//replace all element end tag brackets </ with "#endtag1#"
instanceXML = replaceSubString(instanceXML, "</", "#endtag1#");
//replace all element brackets > with "#endtag1#"
instanceXML = replaceSubString(instanceXML, ">", "#endtag1#");
//replace all begin brackets < with "--><item name=\"\"
instanceXML = replaceSubString(instanceXML, "<", "--><item name=\"");
//replace all former brackets > with "\"><text>"
instanceXML = replaceSubString(instanceXML, "#endtag2#", "\"><text>");
//replace all former end tags </ with "</text></item><!--"
instanceXML = replaceSubString(instanceXML, "#endtag1#",
    "</text></item><!--");

//now add begin and end tags for database / document
instanceXML = s1 + instanceXML + s2;
return instanceXML;
}

//read an input stream to a String
private static String getString(PipedInputStream pi) {
    String piStr = "";
    //this will work only for short string (< 2048 bytes)
    //longer strings must be read in a loop
    try {
        int rest = 0;
        byte[] b = new byte[2048];
        rest = pi.read(b, 0, 2048);
        piStr = new String(b);
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
    return piStr;
}

//create an attachment in a notes document as mime type
private static void createAttachment(Document docAtt, String fileName,
    String targetField, Session session, ServletInputStream servletin,
    String contentType) {
    try {
        servletin.reset(); //make sure, we read full post
        //Create mime entity
        // Do not convert MIME to rich text
        session.setConvertMIME(false);
        boolean isNew = true;
        Stream stream = session.createStream();
        MIMEEntity body = null;
        MIMEHeader header = null;

        // Create parent entity if new doc
        if (!(docAtt.hasItem(targetField))) {
            body = docAtt.createMIMEEntity(targetField);
            header = body.createHeader("Content-Type");
            header.setHeaderVal("multipart/mixed");
        } else {
            body = docAtt.getMIMEEntity(targetField);
            header = body.getNthHeader("Content-Type", 1);
            isNew = false;
        }

        //transfer xfdl document into a buffer document
        int readCounter = 1;
    }
}

```

```

byte[] b = new byte[2048];
byte[] bEnd;
int numberOfBytesRead = 0;
while ((numberOfBytesRead = servletin.read(b, 0, b.length)) != -1) {
    if (numberOfBytesRead != 2048) { //last part of the message -
        // truncate the buffe after end
        // of request
        byte[] bTrunc = new byte[numberOfBytesRead];
        for (int i = 0; i < numberOfBytesRead; i++)
            bTrunc[i] = b[i];
        stream.write(bTrunc);
    } else
        stream.write(b);
    readCounter++;
}
// we will close the stream later on, when all data is processed
servletin.reset();

//prepare stream to write into mime entity
stream.setPosition(0); //set pointer at the first character/byte
body.setContentFromBytes(stream, contentType, MIMEEntity.ENC_NONE);
body.decodeContent();

//now care about file name and other header values
String currFileName = fileName;

//create fie name
if (currFileName.indexOf("#ID#") > -1) {
    java.text.SimpleDateFormat formatter = new java.text.SimpleDateFormat(
        "yyyy-MM-dd-HH-mm-ss-SSS");
    java.util.Date currentTime = new java.util.Date();
    String dateString = formatter.format(currentTime);
    currFileName = currFileName.substring(0, currFileName
        .indexOf("#ID#"))
        + dateString
        + currFileName.substring(
            currFileName.indexOf("#ID#") + 4, currFileName
                .length());
    dateString = "";
}

//write headers
if (isNew) {
    //new -> create headers and set
    header = body.createHeader("Content-Disposition");
    header.setHeaderVal("attachment; filename=" + currFileName);
    header = body.createHeader("Content-ID");
    header.setHeaderVal(currFileName);
} else {
    //update -> get headers and set new file name
    header = body.getNthHeader("Content-Disposition", 1);
    header.setHeaderVal("attachment; filename=" + currFileName);
    header = body.getNthHeader("Content-ID");
    header.setHeaderVal(currFileName);
}

//close the request and save request document
docAtt.closeMIMEEntities(true, targetField);
stream.close();
docAtt.save(true, true);

```

```

        debugOut("attachment created: " + currFileName);

        //now clean up request objects and call the agent with the request
        // doc as context
        body.recycle();
        session.setConvertMIME(true);
    } catch (lotus.domino.NotesException en) {
        en.printStackTrace();
    } catch (java.io.IOException ei) {
        ei.printStackTrace();
    }
}

//write a trace line, if debug mode is on
static void debugOut(String str) {
    if (debugMode.equals("on")) {
        System.out.println(sv + ": " + str);
    }
}

//compose a response message, if there are no errors.
//if the call had an url= parameter, redirect to this url
static String getRespMessage(String nextUrl) {
    String respMessage = "";
    if (nextUrl.equals(""))
    {
        respMessage = "<html><h2>Document processed</h2><B></html>";
    }
    else
    {
        respMessage = "<html><head>";
        respMessage = respMessage + "<script language=\"JavaScript\">";
        respMessage = respMessage + "open(\"" + nextUrl+"\", \"\")";
        respMessage = respMessage + "self.focus();self.close();";
        respMessage = respMessage + "</script>";
        respMessage = respMessage + "</head><body></body></html>";

        respMessage = "<html><head></head><body>";
        respMessage = respMessage + "<form name=\"f\" action=\"" + nextUrl + "\""
method=get></form>";
        respMessage = respMessage + "<script language=\"JavaScript\">function
s(){document.f.submit();}window.setTimeout(\"s()\",10);</script>";
        respMessage = respMessage + "<a href=\"javascript:subm()\">please click here if
forwarding does not work in your browser</a>";
        respMessage = respMessage + "</body></html>";
    }
    return respMessage;
}

} // end of class

```

The servlet version above must run on a Domino server, since it does not connect remotely to the server.

For connecting remotely to a Domino server, such as using WebSphere Application Server (WAS) as a servlet engine, the Domino session could be initiated like this (Example 9-17).

Example 9-17 Initiating the Domino session

```
//open Domino session
//ORBServer is the name of a Domino Server running DIIOP task
//if ORBServer is empty we assume we are running on a Domino sever
//dbServer is the server name, whete the target db is located
//if have additional settings in servlet.properties file for
//ORBServer, dbServer, userName and password

try {
NotesThread.sinitThread();
    //very important - without this all concurrent running sessions will mix up
    //their domino objects ....
    if (ORBServer.equals("") || ORBServer.equals("localhost"))
    { //local session without DIIOP
        errorMessageDetail =
        "Error creating local Domino session (may be username or password in" +
        "servlets.properties not valid)";
        session = NotesFactory.createSession("",userName, password); }
    else
    { //remote session using DIIOP
        errorMessageDetail =
        "Error creating remote Domino session over DIIOP. ORBServer: '" + ORBServer +
        "' (may be ORBServer, username or password in servlets.properties not valid)";
        session = NotesFactory.createSession(ORBServer,userName, password);
    }

    if (debugMode.equals("on")) {System.out.println("SID: " + String.valueOf(sessionCallId)
    + " " + "2. Domino session created - read request message");}

    if (dbServer.equals("") || dbServer.equals("localhost"))
        { //get the target databas
        errorMessageDetail = "target database on localhost not found (insuffitient rights
or dbPath in servlets.properties not valid)";
        db = session.getDatabase(null, dbPath);}
    else
    {
        errorMessageDetail = "target database on server '" + dbServer +
        "' not opened (insuffitient rights or dbPath in servlets.properties not valid)";
        db = session.getDatabase(dbServer, dbPath);}

    ....
} catch ( lotus.domino.NotesException en){
    en.printStackTrace();
    errorMessageText = "ERROR: " + en.id + "ErrorMessage: " + en.text;
}
finally {
    NotesThread.stermThread();
}
}
```

To deploy the servlet (see Example 9-18), we have to complete three tasks:

1. Compile the servlet into a ProcessXFDL.class file.
2. Copy the class file to the <DominoDataRoot>/Domino/Servlets directory.
3. Copy the following servlets.properties file to the <DominoDataRoot> directory.

Example 9-18 The servlet.properties file in Domino server data root

```
##### beginning of servlets.properties #####
#assign an alias for each different servlet behavior:
servlet.XFDLServletRedPaper.code=ProcessXFDL

#set a behavior for each alias
# specify the following parameters
# paramexampledescription

# dbPathpath to the target database on the dbServer
# test/x.nsfspecify any valid path to the target database.
# for each incominh POST request the servlet will create a notes document in
this database
#
# form Form variable for documents to create
# memo this entry will set the form field in the created document
#
# targetFieldName of the target field to create the request file attachment.
# This attachment contains the complete POST request data
# body Create the attachment in Body field
# <empty>if no value is assigned, the field name is "body"
#
# fileName specify any valid file name for the created attachment
# place #ID# in the name where a unique ID should be added.
# request.txt
# incomingP0#ID#.xfd1
#
# contentTypecontent type for the stored attachment.
# Specify and valid content type for mime types
# text/plainPlain text
# text/xmlxml
# text/htmlhtml page
# application/xmlxml document
#
# responseFieldField to read the response attachment from.
# response field can be used, if a post processing agent is specified.
# In this case, the agent can create a response and store it in the specified
field
# responseThe response can be found in "response" field
# <empty>No response field specified => create a standard html response
#
# debugModewill cause the servlet to print debug messages in system console if set to 'on'
# on print trace messages
# off do not print trace messages
#
# respMessageTypedefines the type of the response message.
# in case of creating the response message from an attachment
# retrieved from responseField, make sure the attachment matches the defined
response type
# This parameter decides also about the type of any error messages (html or soap
style)
# html response message will be a valid html document to show up in a browser
# soap response document will be a valid soap response message
#

##### begin redpaper servlet settings #####

#create or update a WPForms related document containing the complete form and extracted for
data
```

```
#authenticated access to the database (username/password set)
#generate a standard html return message or open a specified url if url= parameter is set
in the post action (no response field declared)
servlet.XFDSLServletRedPaper.initArgs \
    dbPath=redpaper/WPForms.nsf,\
    fileName=Request#ID#.xfd1,\
    form=QuotationRequest,\
    targetField=Body,\
    contentType=application/vnd.xfd1,\
    userName=A User,\
    password=auser,\
    debugMode=on,\
    dataInstanceID=FormOrderData

##### end of servlets.properties #####
```

Tip: To add servlet support for a different form, add the following lines to the `servlet.properties` file:

At the beginning of the file, add this line:

```
servlet.XFDSLServletRedPaper2.code=ProcessXFDL
```

At the bottom of the file, add lines like this matching the new virtual address

```
servlet.XFDSLServletRedPaper2.initArgs \
    dbPath=redpaper/MyTargetDB.nsf,\
    fileName=MyRequest#ID#.xfd1,\
    form=MyRequestFormName,\
    targetField=MyBodyField,\
    contentType=application/vnd.xfd1,\
    userName=My Name,\
    password=myPassword,\
    debugMode=off,\
    dataInstanceID=myInstanceToExtract
```

After making changes to the `servlet.properties` file or the deployed `RenderXFDL.class` file, the Domino HTTP task must restart. On the server console, enter:

```
tell HTTP restart
```

Now we will describe the next component to build: the new `QuotationRequest` form.

QuotationRequest Form and corresponding views

This form should contain a body field to store the submitted XFDL form (named `Body` as defined in the `servlet.properties` file) and some fields containing the detail data extracted from the XFDL form. The field names must match exactly the element names in the extracted data instance (Example 9-19).

Example 9-19 Data instance `FormOrderData` to extract into the request document

```
</xforms:instance>
  <xforms:instance xmlns="" id="FormOrderData">
    <FormOrderData>
      <ID></ID>
      <CustomerID></CustomerID>
      <Amount></Amount>
      <Discount></Discount>
      <SubmitterID></SubmitterID>
      <State>1</State>
```

```

        <CreationDate></CreationDate>
        <CompletionDate></CompletionDate>
        <Owner></Owner>
        <Version></Version>
        <Approver1></Approver1>
        <ApprovalDate1></ApprovalDate1>
        <Approver1Comment></Approver1Comment>
        <Approver2></Approver2>
        <ApprovalDate2></ApprovalDate2>
        <Approver2Comment></Approver2Comment>
    </FormOrderData>
</xforms:instance>

```

DXLImporter will create all these fields in the request document. Nevertheless, we will only design some of them in the form.

Create a form in the Template DB named QuotationRequest (as defined in the servlets.properties file) and create the necessary fields. Finally, the form should look as shown in Figure 9-31.

Quotation Request Record
Request data:

Order Number	<input type="text" value="ID"/> T
Customer Number	<input type="text" value="CustomerID"/> T
Amount	<input type="text" value="AMOUNT"/> #
Discount	<input type="text" value="DISCOUNT"/> #
Creation Date	<input type="text" value="CreationDate"/> T
Completion Date	<input type="text" value="CompletionDate"/> T
Submitter ID	<input type="text" value="SubmitterID"/> T
State	<input type="text" value="State"/>

Request Form: T

Figure 9-31 Form QuotationRequest.

To prevent caching in the browser, insert the following code in the form's HTML header content (Example 9-20).

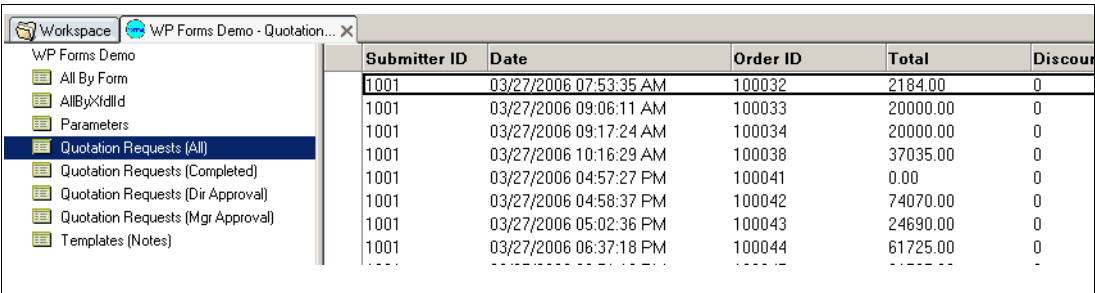
Example 9-20 HTML header code preventing IE6 from caching old content

```

"<META HTTP-EQUIV=\"Pragma\" CONTENT=\"no-cache\">"+
"<META HTTP-EQUIV=\"Expires\" CONTENT=\"-1\">"

```

Save the form. Create a maintenance view, Quotation Requests (All) for the forms as shown in Figure 9-32.



Workspace	WP Forms Demo - Quotation...
WP Forms Demo	
All By Form	
AllByXfdlId	
Parameters	
Quotation Requests (All)	
Quotation Requests (Completed)	
Quotation Requests (Dir Approval)	
Quotation Requests (Mgr Approval)	
Templates (Notes)	

Submitter ID	Date	Order ID	Total	Discount
1001	03/27/2006 07:53:35 AM	100032	2184.00	0
1001	03/27/2006 09:06:11 AM	100033	20000.00	0
1001	03/27/2006 09:17:24 AM	100034	20000.00	0
1001	03/27/2006 10:16:29 AM	100038	37035.00	0
1001	03/27/2006 04:57:27 PM	100041	0.00	0
1001	03/27/2006 04:58:37 PM	100042	74070.00	0
1001	03/27/2006 05:02:36 PM	100043	24690.00	0
1001	03/27/2006 06:37:18 PM	100044	61725.00	0
...

Figure 9-32 Maintenance view for all request documents

There are no special requirements for this view. Make one column appear as a link in the browser.

Assign the selection formula:

SELECT Form="QuotationRequest "

Save the view. Next make three copies of the view and change view title and selection formula as follows (Table 9-7).

Table 9-7 Additional quotation requests corresponding to workflow state

View title	Selection formula
Quotation requests (completed)	SELECT Form="QuotationRequest" & State = "4"
Quotation requests (manager's approval)	SELECT Form="QuotationRequest" & State = "2"
Quotation requests (director's approval)	SELECT Form="QuotationRequest" & State = "3"

Next we will create a lookup view to find the request documents related to the incoming XFDL form.

Form lookup view

To create a form independent relationship between an incoming XFDL form and related documents in a Notes database, we need corresponding unique keys in both the Notes document and the XFDL form.

It is a good idea to create in each form template a data item containing a unique identifier. This value could be set, for example, when creating new documents from template. In our scenario we have a unique order number — we will use this order number as this identifier. We could use any other unique identifier as well. Make sure that all incoming XFDL forms to the servlet will have only the same ID, when they are related to the same business object.

The view to create is simple:

View Name: AllByXfdlId (This name is hard coded in the servlet)
Selection Formula: SELECT @Trim(XFDLID) != ""
First Column: Field XFDLID, sorted case insensitive
Second Column: Formula @Text(@DocumentUniqueID)

Now save the view.

Compare the related XML element used in the XFDL form. This element is bound to the order number items and will receive the value when the order number is applied (Example 9-21).

Example 9-21 element <custom:formid> created in the XFDL form template

```
<!-- <?xml version="1.0" encoding="ISO-8859-1"?>
<XFDL xmlns="http://www.PureEdge.com/XFDL/6.5"
xmlns:cm="http://www.PureEdge.com/idx/ibmcm/1.0"
xmlns:custom="http://www.PureEdge.com/XFDL/Custom"
xmlns:designer="http://www.PureEdge.com/Designer/6.1"
xmlns:pecs="http://www.PureEdge.com/PECustomerService"
xmlns:xfdl="http://www.PureEdge.com/XFDL/6.5"
xmlns:xforms="http://www.w3.org/2003/xforms">>
  <globalpage sid="global">
    <global sid="global">
      <vfd_date>9/3/2006</vfd_date>
      <formid>
        <version>14.182.0</version>
      </formid>
      <custom:formid></custom:formid>

      .....
      <bindings>
        <bind>
          <instanceid>FormOrderData</instanceid>
          <ref>[null:FormOrderData][null:ID]</ref>
          <boundoption>global.global.custom:formid</boundoption>
        </bind>
      </bindings>
    .....
  </XFDL>
```

Having the form and views in place and the servlet / servlets.properties file deployed on server, just restart the HTTP task and see how it works.

Submitting XFDL form in the Domino environment

Perform these tests:

First test: Hit the servlet - Enter

<http://vmforms1.cam.itso.ibm.com/servlet/XFDLServletRedPaper>

The following panel should appear (Figure 9-33).

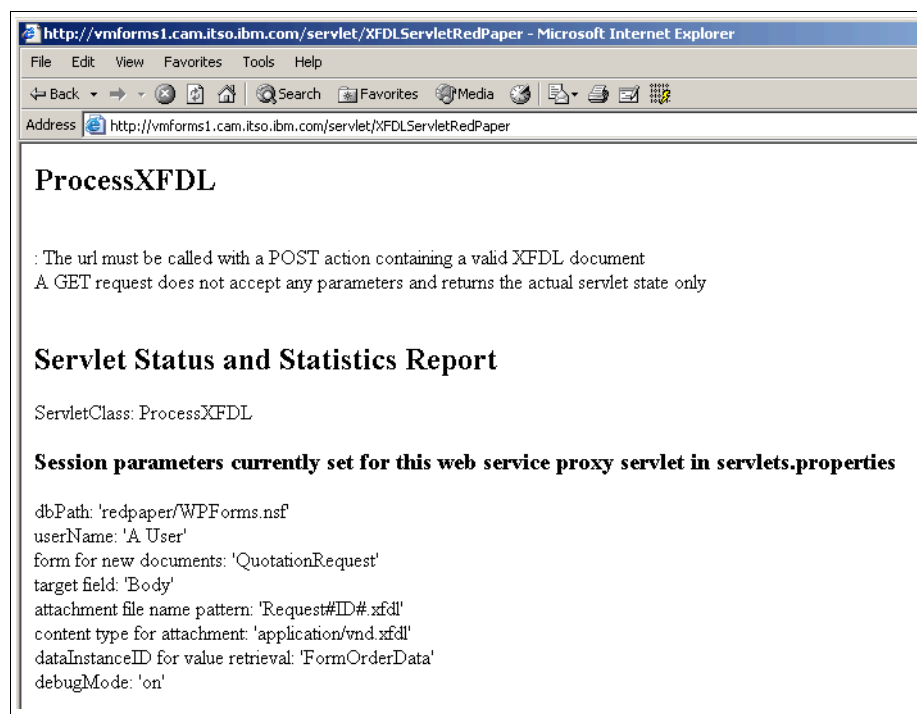


Figure 9-33 Servlet test with a GET request displays available settings

Next test: Create a new quotation request, enter some data, and submit it.

These are the check points to look for:

- ▶ The order number must be set, and employee data must be filled (prepopulation running).
- ▶ When clicking on customer choices or item selection there must be some choices available (Web services running, target URLs for the Web services successfully changed by text parsing).
- ▶ When selecting a customer and an item detail, data must appear (detail data Web services running and matching created data instances)
- ▶ Sign the document, and remember the order number (make a note of it for reference).
- ▶ Submit it — the Domino log should show a lot of trace lines coming up (Example 9-22).

Example 9-22 Trace lines in Domino log on XFDL form submit

```
03/31/2006 05:55:00 PM HTTP JVM: ProcessXFDL: next url:
'http://vmforms1.cam.itso.ibm.com/redpaper/WPForms.nsf'
03/31/2006 05:55:02 PM HTTP JVM: ProcessXFDL: theForm created
03/31/2006 05:55:02 PM HTTP JVM: ProcessXFDL: form stored to formString
03/31/2006 05:55:02 PM HTTP JVM: ProcessXFDL: Form ID: 100103
03/31/2006 05:55:02 PM HTTP JVM: ProcessXFDL: data instance extracted
03/31/2006 05:55:02 PM HTTP JVM: ProcessXFDL: stream: <FormOrderData xmlns=""
xmlns:cm="http://www.PureEdge.com/idk/ibmcm/1.0"
xmlns:custom="http://www.PureEdge.com/XFDL/Custom"
xmlns:designer="http://www.PureEdge.com/Designer/6.1"
xmlns:pecs="http://www.PureEdge.com/PECustomerService"
xmlns:xfd="http://www.PureEdge.com/XFDL/6.5" xmlns:xforms="http://www.w3.org/2003/xforms">
  <ID>100103</ID>
  <CustomerID>100003</CustomerID>
  <Amount>150.00</Amount>
```

```

    <Discount>0</Discount>
    <SubmitterID>1010</S
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: form destroyed
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: Notes session created
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: Notes db found
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: import XML: <database
version="7.0"><document form='QuotationRequest' version='7.0'
replicaId='8825714100620266'><noteinfo noteId='' unid='' sequence='' /><!--
--><item name="ID"><text>100103</text></item><!--ID"><text>
--><item name="CustomerID"><text>100003</text></item><!--CustomerID"><text>
--><item name="Amount"><text>150.00</text></item><!--Amount"><text>
--><item name="Discount"><text>0</text></item><!--Discount"><text>
--><item name="Su
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: Stream filled
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: DXLImporter created
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: document imported/updated
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: first node requested: 99A
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: node accessed:
45EA41465C22294588257143000A88A4
03/31/2006 05:55:03 PM HTTP JVM: ProcessXFDL: attachment created:
Request2006-03-31-17-55-03-641.xfdl

```

- ▶ The available views should be displayed as response pages.
- ▶ Navigate to Quotation Requests (All) view.
- ▶ Find the document with the order number, and open it (Figure 9-34).

Quotation Request Record

Request data:

Order Number	100103
Customer Number	100003
Amount	150.00
Discount	0
Creation Date	
Completion Date	
Submitter ID	1010
State	Completed

Request Form:

[Request2006-03-31-17-55-03-641.xfdl](#)

Type: application/vnd.xfdl

Name: Request2006-03-31-17-55-03-641.xfdl

Figure 9-34 New created request document



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/sg247279>

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, SG247279.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
Form_Page_scanned_image.jpg	Image to use as an initial starting point for building a form template.
CH4_J2EEStage1_DeployWAS5.zip	WAR file to deploy on the WebSphere Application Server (WAS) 5 containing the Web application for Stage 1 (stand-alone form, no DB2 integration) and deployment information.
CH4_J2EEStage1_Dev.zip	RAD6 project as Project Interchange file ready to re-import into RAD6 Workspace containing the Web application for Stage 1 (stand-alone form, no DB2 integration).

CH4_J2EEStage1_Form.zip	Sample form used in Web application for Stage 1 (stand-alone form, no DB2 integration).
CH5-8_DB2Basics_DB2_Setup.zip	Setup files creating DB2 tables and initial sample data along with the setup instructions. The created database is a prerequisite for all development and demonstration tasks in Chapters 5 through 8 (J2EE Stage 2, Portal integration, CM Integration).
CH5-8_DB2Basics_DeployTomcat5.zip	JAR files and WAR files to deploy on Web service provider (Tomcat 5 server). The deployed Web services are called in the sample forms used in Chapters 5 through 8 (J2EE Stage 2, Portal integration, CM Integration).
CH5-8_DB2Basics_Dev.zip	WSDL files used to create J2EE based Web services and RAD6 projects as zipped Project Interchange files ready to re-import into RAD6 Workspace containing the Web service projects and the DB2 connector project. The created Web services are called in the sample forms used in Chapters 5 through 8 (J2EE Stage 2, Portal integration, CM Integration), the DB2 connection application is a prerequisite for both the created Web services and the J2EE integration servlets / portlets created in Chapters 5 through 8.
CH5_J2EEStage2_DeployWAS5.zip	JAR files and WAR files to deploy on WebSphere Application Server (WAS) 5 with deployment instructions. The deployed JAR file provides database access to the SubmissionServlet for data prepopulation and data storage from/to DB2. The WAR file contains the Web application used in J2EE stage 2.
CH5_J2EEStage2_Dev.zip	RAD6 project as zipped Project Interchange file ready to re-import into RAD6 Workspace containing the Web application used in Chapter 5 (J2EE stage 2).
CH5_J2EEStage2_Form.zip	Workplace Forms templates used in Chapter 5 (J2EE stage 2) working with data prepopulation, Web services, and data extraction.
CH6_Portal_DeployPortal51.zip	Workplace Forms and Portal integration readme.txt used in Chapter 6, "Integrating with Portal".
CH6_Portal_Dev.zip	Workplace Forms WAR file used in Chapter 6, "Integrating with Portal".
CH6_Portal_Form.zip	Workplace Forms template used in Chapter 6, "Integrating with Portal".
CH8_CMIntegr_DeployWAS5.zip	WAR file to deploy on WebSphere Application Server (WAS) 5 with deployment instructions. The WAR file contains the extended Web application used in J2EE stage 2 able to submit received forms to Content Manager.

CH8_CMIntegr_Dev.zip

RAD6 project as zipped Project Interchange file ready to re-import into RAD6 Workspace containing the extended Web application used in J2EE stage 2 able to submit received forms to Content Manager.

CH8_CMIntegr_Form.zip

Workplace Forms templates used in Chapter 5 and 8 (J2EE stage 2 with CM integration) working with data prepopulation, Web services, and data extraction.

CH9_Domino_Deploy.zip

Domino databases and ProcessXFDL servlet to deploy on Domino server along with deployment instructions.

CH9_Domino_wsdl.zip

WSDL files used as a starting point to create Domino based and J2EE based Web services.

Related publications

The resources listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ IBM Workplace Forms Home Page:
<http://www-142.ibm.com/software/workplace/products/product5.nsf/wdocs/formshome>
- ▶ IBM Workplace Forms Resources on Lotus Developer Domain:
<http://www-128.ibm.com/developerworks/workplace/products/forms/>

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

A

- Absolute and Relative Positioning 79
- Access a Form through Workplace Forms API 118
- Access Control Server (ACS) 23–24
- Action name 217
- action value 134–135, 191, 195, 225, 227
- action.equaIgnoreCase 123–124, 185–186, 230
- actionPerformed method 217
- Adaptive handling of absolute and relative paths to the stored XFDL forms 139
- Add a Submit button 95
- Add Layout items 78
- Add Signature buttons 97
- Add the CM Integration in the Form 260
- Adding Input items 83
- Adding input items to the Traditional Form 84
- Adding input items to the Wizard Pages 86
- Adding the Data Model 102
- Adding the Portlet to a page 236
- Adding the xmlmodelValidate Function 107
- Adjustments to JSPs for Stage 2 190
- Advantages of using FormBridge 69
- applet 26
- Application Programmer Interface (API) 21–22, 118, 145, 164, 247, 274, 277
- Apply Formatting and Logic 87
- Approaches to Integrating Workplace Forms 29
- Approval workflow 205
- approval workflow 53, 168, 180, 190, 196, 287
- Approved Form Listing 143
- Aspects of Integrating Workplace Forms 33

B

- back-end system 1, 3, 107
- Banking and Regulated Industries 14
- base scenario 1, 53–54, 145–146, 168, 241
 - Building Stage 2 146, 168
 - complete description 241
 - Footprint functionality 241
- Basic Architecture 23, 26
- Basic Design of Content Manager Integration 253
- Basic servlet methods 108
- Build a toggle compute 91
- Building the base scenario - Stage 1 53
- Building the base scenario - Stage 2 145
- Building the layout 80
- Building the Servlet 108
- Building the Traditional Form Page 70
- business logic 1, 8, 48–49, 54–55, 127, 158, 205, 217
 - main part 127
- business process 3, 103, 274
 - critical part 3
 - starting point 3

C

- Calculation by decision 88
- Call doc.Save 289
- Calling the Web Services 199
- Calls to Process Server for Real-Time Process Interaction 49
- CGI variable 295, 313
- charset 132, 137, 193, 225–226
- Click-to-Action Sender
 - portlet 211–212
- Clickwrap signature 14, 25, 99, 247
- Client-Side Device / Hardware Integration 49
- CM Integration 253, 334–335
 - J2EE stage 2 335
- CM Submission servlet
 - sample implementation 252
- codebase 2, 53, 145, 249, 273
- COM interface 21, 118, 292
- Common Gateway Interfaces (CGI) 22
- Common scenario for Web services in Forms 170
- Common XML Data Model 12
- Compatible Technologies 22
- config.getInitParameter 315–316
- Considerations in advance - Best Practices for implementing the Traditional Forms and Wizard Pages 65
- Content Manager 42, 49, 59, 105, 249–250, 334
 - critical requirement 249
 - Demo Platform 250, 252
 - form items 253
 - instance 267
 - Integration 251, 253
 - Item Type 255
 - servlet 266–267
 - submission servlet 255
 - test environment 267
 - Update existing form items 253
 - Workflow 51
- Create a Custom Option 90
- Create a scanned template 71
- Create a template document 304
- Create Attributes and Item Type 255
- Create Multiple Signatures 98
- Create Tables 148
- Create template repository and form storage structure 116
- Create used parameter documents 308
- Create XML Bindings 105
- Create XML Data Model 104
- Created Links 139, 195
- Creating a Field Calculation 87
- Creating a JSP to view DB2 data 192
- Creating Clickwrap Signatures 99
- Creating JSPs 130
- Creating new form from template embedding template in

- HTML form 292
- Creating the JSP files to display in the Portal 224
- Creating the layout for the Wizard Pages 73
- custom option 79, 90, 199
- Customer data 56, 149, 162, 275, 277
- Customer Id 150, 165

D

- data element 33, 42, 105
- data fragment 43, 107
- Data Instance
 - Bindings tab Select FormOrderData 264
- data instance 32, 42, 103, 106, 162, 164, 182, 260, 262, 277, 296
 - Bindings tie one element 104
 - corresponding element names 290
 - entire first element 184
 - first child element 305
 - first node 184
 - FormMetaData 125
 - FormOrgData 304
 - Id 314
 - instance id 296
 - match 304
 - name 289
 - object 184
 - OrderNumber 183
 - prepopulation 167, 290, 292
 - response data 202
 - store xml fragment 302
 - update 183–184
- Data Integration 41
- Data Model 25, 31–32, 50, 66, 102, 157, 165
 - base data structures 103
 - individual elements 105
- data prepopulation 334–335
- Data Storage to DB2 146
- DB2 client 145, 157
- DB2 Content Manager 249–250
 - eClient 269–270
 - Integration 266
 - item types 255
 - System Administration Client 255
 - Workplace Forms 252
- DB2 Content Manager eClient
 - Basic Search 271
 - Home Page 270
 - Login 269
- DB2 Table
 - actual order metadata 166
- DB2 table 145, 154, 334
- Define the Purpose of the Form 67
- Deploying the Portlet 235
- Deployment Server
 - applet 26
 - servlet 26
- Description of the scenario 56
- Designing a form from scratch 67
- Designing the layout of the Traditional Form 70
- Determine Your Item Type Needs 67

- Developing Data Access Layer (DB2) 157
- Development Workstations 157
- Differences Between Webform Server and the Viewer 24
- Digital Signature 5, 9, 21, 23, 33, 38, 67, 97, 181, 242, 247
 - Standard Data Model 5
- Dim db 285
- director approval 62, 109, 150, 154, 277
- dirlisting1.jsp 136
- Display of Workplace Forms within a Portal Page 36
- Display of Workplace Forms within a Web Page 33
- Display of Workplace Forms Within Eclipse 41
- Display of Workplace Forms within Notes / Domino 38
- doGet method 108, 115, 127, 182, 188, 314
 - additional code 189
- Domino adds value to Workplace Forms 274
- Domino and Workplace Forms integration 274
- Domino and Workplace Forms technologies 274
- Domino Development 279
- Domino Integration 171–172, 180, 273, 276–277
 - Basic usage scenario 277
- doView method 217
- DXLImporter 314–315

E

- e.prin tStackTrace 159–160, 268, 316–317
- electronic form 1, 4, 18, 54–55
 - business value 3
- Electronic Forms - XML Intelligent documents 4
- element name 176, 198, 282
- E-mail Address 133, 149
- employee data 98, 146, 148, 276, 304
 - lookup view 280
 - prepopulation settings 304
- Enabling Smartfill 107
- Enterprise Content Management (ECM) 4
- entry.ColumnValues 285
- entry.getItemValue 286
- Environment overview 276
- ex.prin tStackTrace 120, 138, 231, 320
- extracted data instance
 - document creation/update 314
 - element names 327
- Extraction of form data 121
- Extraction of form data and storage of entire form 185

F

- Features and Functionality 17
- field name 158, 187, 280, 289
- file system 26–27, 109, 116, 145, 151, 212–213, 289
 - detached attachment 289
 - Form_Templates folder 142
 - stored template 182
- first test 175, 304, 308
 - Last step 304
- FONT size 135–136, 191, 227–228
- For Loops and nesting loops in XFDL 200
- Form as a front end to a business process 3
- Form design delta 243

- Form Document Model 18
- Form Lookup View 329
- FORM method 134–135, 191, 225, 227
- Form prepopulation 181
- Form Pre-Prepopulation using Web Services 196
- Form Storage 114, 116
- Form storage to local file system 126
- Form template listing 142
- Form-Based Data Collection as a Human-Task Within a Workflow 49
- FormBridge features 69
- FormNodeP theForm 119, 121, 182–183, 315, 320
- Forms
 - 280
- Forms Access Buttons 134
- forms API 108, 112, 174, 180, 278, 313
- Forms Marketplace 5
- Forms Viewer 8
- forms-based process 9
- Form-Template listing 142
- Formula dialogue 91–92
- Function Call
 - Dialogue 93
 - Interface 49, 118
- Function GETCUSTINFO 281, 284
- Function GETCUSTOMERLIST 281, 284

G

- given order id
 - order data details 166
- global sid 104, 125, 183, 245, 306, 312
- globalpage sid 104, 125, 183, 245, 306, 312
- Goal of integrating the application with WebSphere Portal 210
- Government Program Registration 14

H

- header.setHeaderVal 322–323
- href 132, 135, 178, 193, 303, 320
- HTML Form 11, 18, 23, 242, 292
- HTML page 234, 277, 286
 - registered scripts 292
- HttpServletRequest arg0 111, 115
- HttpServletRequest request 119, 122, 182, 185, 316–317
- HttpServletResponse arg1 111, 115
- HttpServletResponse response 119, 121, 182, 185, 316–317

I

- IBM DB2 Content Manager
 - Demo platform 252
 - Enterprise Edition 255, 260
- IBM Workplace
 - Form 19, 21, 32, 58, 65, 249, 253
 - Forms Integration 314
 - Forms Release 2.5 249
 - Forms Selection 137, 193

- Forms Server 118, 174, 241–242
- Forms Viewer 24
- Forms Webform Server 23
 - solution 10
- IBM Workplace Forms 1–2, 6
- IBM Workplace Forms application 1
- IBM Workplace Forms Designer 6, 8
- IBM Workplace Forms product family 1
- IBM Workplace Forms Release 2.5 2
- IBM Workplace Forms Server 6, 8
- IBM Workplace Forms suite 6
- IBM Workplace Forms tool 6, 14
- IBM Workplace Forms Viewer 8
- IFX 22, 38, 48
- IMG border 133, 137, 194
- import javax.servlet.Serv
 - letConfig 111, 113
 - letContext 113
 - letException 111, 113
- import namespace 176, 198, 282
- Importing the WSDL File 197
- index.jsp 226
- init method 108, 113, 267, 314
- initiation agent 286, 292
- Initiation Agent RenderXFDLPrepop 298
- Initiation form RenderXFDL 295
- Initiation of a task or workflow based on form submission or completion. 49
- Innovation Based on Standards - XForms and XFDL 11
- input item 67, 78
- INPUT type 134–135, 191, 195, 225, 227
- Installing DB2 Clients on Development Clients and Servers 157
- Installing DB2 Server 147
- instance data 293, 296
 - xml structures 298
- int i 138, 194, 231, 320, 323
- Integrating the Sales Quote Sample with DB2 Content Manager 255
- Integrating with Portal 209
- Integration - what does this mean within the context of Workplace Forms 30
- Integration of an XML Data Instance with a Line-Of-Business System 46
- Integration of Multiple XML Data Instances to Separate Systems 47
- Integration Points Summary 50
- Integration with IBM DB2 Content Manager 249
- Introduction to actual integration scenarios 51
- Introduction to the scenario used throughout this book 54
- Introduction to Workplace Forms 1
- item details (ID) 146, 148
- Item Properties 78
- Item Type 67, 250, 253

J

- java file 214, 217
 - file system 214
- Java Server Page 22, 130
- java.io.File.separator 137

java.io.File.separator 137
java.rmi.RemoteException 173–174
jsp value 138, 141, 191, 195
JSPs 53, 64, 145–146, 210, 214
 Home buttons 109

L

lastOrderState.equals 187–188
layout tool 67, 79
Layout Tools 79
Line-of- Business (LOB) 13
list.jsp 229
Logo Graphics 133

M

Manager Approval 62, 109, 150, 154
META http-equiv 132, 137, 193, 289, 328
metadata 251
Modifying the index.jsp 190

N

next section 70, 254, 287
next url 317, 331
NOT NULL Default 151

O

Objective of integration scenario 275
Order Id 150
order number 146, 167, 277
 new data instance 167
OrderNumber type 306, 311
orderState.equals 187–188
out.print 316–317
out.println 316–317
Overview 251
Overview of IBM Workplace Forms 6
Overview of Portal Integration 211
Overview of steps in Building Stage 1 Scenario 64
Overview of steps in Building Stage 2 of Base Scenario 146
Overview of the XML Data Model 103

P

Page Setup 70
Paper Form 68, 71
paper form 1, 8, 18, 67–68, 71
 scanned image 68
paper-based form to electronic forms-based application 56
PARAM Name 233–234, 292–293
Parameter form an view 287
Parking the Workplace Forms Viewer in the Portal 234
Partitioning of Features / Functionality 50
PID field 261–262
Populate Tables 153
pop-up list 65, 95
 multiple values 65

pop-up menu 199
portlet 23–24, 36, 42, 209–210
portlet session 218–219
 Bean name 218
portlet.xml 214
portletAPI
 URIAction name 225, 227
Possible starting points for creating a form 67
POST data 315
Preparing to Building the Form Template 64
prepopulation data 167, 296–297
prepopulation setting 286, 291
 stored template 286
prepopulation test 308
Preview of the Wizard Page to be built 74
PRIMARY Key 151–152
private static String
 APPROVED_FOLDER 114
 CANCELLED_FOLDER 114
 CMSubmissionUrl 267
 createImporterXML 321
 createTag 164
 DIRECTOR_FOLDER 114
 FORM_NAME_PREFIX 114
 getFormAsString 121, 320
 getJspExtension 224
 getString 322
 MANAGER_FOLDER 114
 replaceSubString 320
 SALES_REP_FOLDER 114
 TEMPLATE_FOLDER 114
Process Flow of Information in a Portlet 217
Process Integration 48
ProcessXFDL servlet receiving incoming XFDL documents 314
Product Positioning 10
Production Servers 157
props.getProperty 115, 126, 185
Proven eForm technology 15
Public CUST_AMGR 281, 284
Public CUST_CONTACT_EMAIL 281, 284
Public CUST_CONTACT_PHONE 281, 284
Public CUST_CONTACT_POSITION 281, 284
Public CUST_CRM_NO 281, 284
Public CUST_ID 281, 284
public static final String
 Cancel 218
 FORM_ACTION 218
 FORM_PROFILE_JSP 218
 INDEX_JSP 218
 LIST_APPROVED 218
 LIST_CANCEL 217
 LIST_JSP 218
 LIST_TEMPLATES 217
 LIST_WORKBASKET_DIRECTOR 217
 LIST_WORKBASKET_MANAGER 217
 LIST_WORKBASKET_SALES 217
 OPEN_FORM 218
 OPEN_FORM_JSP 218
 PROFILE_FORM 218

- SESSION_BEAN 218
- TEST_JSP 218
- Text 218
- VIEW_BEAN 218
- VIEW_JSP 218
- public void
 - actionPerformed 221
 - doGet 182, 316
 - doPost 119, 122, 185, 317
 - doView 219
 - encloseInstance 184
 - init 111, 114, 218, 315
 - messageReceived 223

Q

QuotationRequest Form and corresponding views 327

R

- RAD6 project 333–334
- Rational Application Developer (RAD) 130, 214
- Reading form data from DB2 188
- Real-Time Data Ingestion via Web Services 43
- Real-Time Data Integration 48
- Redbooks Web site 337
 - Contact us xi
- remote server 157
 - additional DB2 clients 157
- Repository Database 276, 278–279
 - employee data 304
 - view design 279
- request.getAttribute 133, 137, 194, 220, 226
- request.getInputStream 119, 123, 185, 317
- request.getParameter 123, 128, 182, 185, 222–223, 317
- request.getSession 127, 129
- request.setAttribute 219
- request.setAttribute 128–129, 219
- response.getOutputStream 119, 121, 182, 186, 317
- response.setHeader 121–122
- return r 320
- return respMessage 324
- Review of the specific Forms - End user perspective 59
- Reviewing the layout for the Traditional Form Page 80
- Reviewing the layout for the Wizard Page 83
- RTF Field 296
 - BodyInline 297
 - Prepop 296

S

- Same functionality 166
- sample application 1, 14–15, 20, 53, 58–59, 146, 205, 255, 273, 276
- Sample form 15, 67–68, 175, 197, 284, 334
- sample form
 - new wsdl file 284
- Sample Solutions 14
- SCRIPT id 293–294, 296
- Secured Communications 14
- Security Access Level Buttons 133

- Security Context Integration 49
- server side 22–23, 43, 175, 180
- Server-side prepopulation of form templates 42
- servlet 22–23, 42, 48, 53, 56, 108, 180–181, 247, 252–253, 274, 278
- Servlet Access to Form Data 179
- Servlet Access to form data (prepopulation / data retrieval) 180
- Servlet code skeleton 110
- Servlet doGet method for application navigation 127
- Servlet interaction 117
- Servlet interaction for forms processing 117
- Servlet to Servlet Communication 266
- servlet.properties 314
 - additional settings 325
- servlet.properties 314
- servlets.properties
 - username or password 325
 - Web service proxy servlet 317
- servlets.properties 315, 317
- Setting up Domino environment 278
- settings.getAttribute 219
- Setup the Toolbar 75
- signature button 69, 91, 97, 206–207
- Signature Filters 97
- SIGNATURE Validation 123, 125
- Signature validation 125
- sql query 158, 160
- Stage 1 53, 64, 108, 151, 180, 185, 277, 333–334
 - Form handler 112
 - Web application 333
- Stage 2 127, 145–146, 241, 247, 266–267, 275, 304, 306
 - first full orders 157
 - J2EE scenario 275
- Standard Architectures 22
- Starting to Build the Forms - Initial creation, design and layout 70
- Starting with a paper-based form 54
- state information 50–51, 316
- Steps to Build the Wizard Pages 75
- Storage of Form Templates and Completed Forms 42
- Straight-Through Integration 31, 33
- String action 122, 128, 222–223
- String clob 162
- String id_field 162
- String orderNumberXML 183–184
- String tagData 164
- StringReader sr 183–184
- STRONG >
 - Amount 194
 - Customer 194
 - ORDER Id 194
 - Owner 194
 - Status 194
- submission servlet 96, 252, 255
- Submitting XFDL Form in Domino Environment 330
- success URL 307, 314
- success1.jsp 140
- Sum Calculation 89
- Support for Arbitrary XML Instances 32

System.out.println 159
System.out.println 114–115, 159, 267–268, 316, 324

T

TABLE border 133, 137, 193–194, 225–226
TABLE width 133, 135, 195, 227–228
table WPF_ORDERS 150, 152
TABLE WPF_ORDNOCNT 151, 153
 order number 167
table WPF_ORG 150–151
taglib uri 225–226
target url 175–176
TD colspan 134, 225–226
Template Database 276–278
 created templatesPrepop view 309
 new form 292
 Parameter Form 287
 used parameter documents 287, 303
Template Database - Components to create a new form
 from template 286
Template Database - Receiving submitted forms in
 Domino 313
template document 290–291, 298
Template Form 287
Template View (WEB) 303
template.getitemvalue 300–301
Test the Form integration with CM 268
text parsing 174, 180, 277–278
 Replacements tab 291
The Workplace Forms Document Model 31
theForm.destination 120, 124, 185–186, 318
theForm.encloseInstance 164, 184
theForm.getLiteralByRefEx 318
theForm.getLiteralByRefEx 120, 187
theForm.writeForm 120, 122, 182, 320
theXFDL.readForm 119, 123, 182, 185, 317
This file is used to display the profile selection so that a
 user can set their role for the session. 225
toolbar 68, 72, 75, 261, 269
tr bgcolor 139, 141, 195, 226
traditional form 64–66, 84
 2-way data transfer 65
 business-appropriate order 64
 different fonts 65
 field reference 66
 Input Item 84
 intuitive manner 66
 item 65
 multiple locations 65
 necessary pages 70
 page 64, 66, 70–71, 260–261
 real estate 65
 respective fields 86
 single field 65
Typical API Uses 22

U

URL 96, 108, 159, 262, 266, 287, 291
used parameter documents 308

used XFDL form
 related XML element 330
User Interface (UI) 31, 33, 36, 130
User Interface (UI) Integration 33
userRole value 133–134
Using a sample form 68
Using scanned paper forms 68
Using Texcel FormBridge to convert an existing format
 69
Using Texcel FormBridge to create your form 69

V

value 54, 147
Value Proposition 8
Views
 280

W

W3C XForms
 standard 11–12
 Support 38
Web application 13, 108, 130, 173, 252, 333–334
 business tier 13
 chapter on programming Domino 173
Web browser 14, 23–24, 34–35, 95, 241–242, 253, 333
 form page 35
Web page 26, 33, 130, 234
 Workplace Forms 33
web page
 other elements 234
Web service 5, 8, 25, 43, 48, 62, 145–146, 241–242,
 274–275, 277, 334–335
 Common scenario 170
 data instance 202
 need to interact 43
 properties box 282
 Real-Time Data Ingestion 43
 target urls 331
Web Service Development 171
Web service runtime 175
Web Services 168
Web services 281
Web services integration 169
Web services moves the solution to Viewer only 247
Web services returning complex type objects 201
Web services returning simple type objects 199
Webform Server 23–24, 37, 241–242
 component 14
 documentation 24
 environment 24
 other differences 24
WebSphere Application Server (WAS) servlet
 actually working code fragment 187
WebSphere Portal 1, 15, 49, 51, 59, 209–210
 environment 214
 following file directories 214
 integration options 15
 same sample application 209
 server 211–213

- WebSphere Studio Application Developer (WSAD) 214
- What does XFDL add to XForms? 12
- What does XForms add to XFDL? 12
- What is a Form 3
- When to Use the XML Data Model 103
- Where we are in the process - Building Stage 1 Scenario 108, 130
- Where we are in the process of Building Stage 2 of Base Scenario 168, 180, 190, 196, 204
- wizard page 38, 64–66, 83
 - Input Items 87
 - multiple values 65
 - user input 65
- Wizard pages versus traditional form pages 64
- Work Basket 63, 117
- Workflow 204
- Workplace Form 17–18, 29–30, 33, 53–54, 93, 145, 168, 209–210, 249–250, 273, 334–335
 - Basic Design 253
 - basic structure 33
 - business value 54
 - common XForms model 12
 - Example screenshots 38
 - key integration points 50
 - open standards architecture 2
 - previously described benefits 37
 - Zero-Footprint Display 36
- Workplace Forms 1–2, 6, 274
- Workplace Forms adds value to Domino 274
- Workplace Forms and Domino integration 274
- Workplace Forms and Domino technologies 274
- Workplace Forms API 22–23, 42, 47, 108, 112, 114–115, 118, 180–181, 278
 - documentation 50
 - necessary external jars 112
- Workplace Forms API vs. XML Parsers 22
- Workplace Forms Component Technology 19
- Workplace Forms Deployment Server 26
- Workplace Forms Designer 19–20, 41, 46, 64, 68, 172, 174, 202, 260
 - formula wizard 88
 - function call wizard 202
 - Toolbar items 78
 - Workplace Form 174
- Workplace Forms Document Model and Straight-Through Integration 31
- Workplace Forms Server 21
- Workplace Forms Server API 21
- Workplace Forms Viewer 6, 8, 19–20, 30, 34, 85, 103, 170, 201, 232, 234, 241–242, 253, 260, 268
 - FCI 50
 - installation package 31
 - interface 21
 - only alternative 241
- Workplace Forms Webform Server 23
- Workplace Viewer 31, 170
 - Web services 170
- WPFormsRedPaperPortletC2ASenderView.jsp 225
- WPFormsRedPaperPortletView.jsp 232
- WPFRedpaper.war file 235

- Writing a Portlet 214
- wsdl
 - binding name 198, 283
 - input message 177, 179, 283
 - message name 176, 179, 282
 - operation name 177, 179, 283
 - output message 177, 179, 283
 - part name 176, 179, 282
- WSDL document 197
 - valid ports 197
- wsdl document 197
- wsdl file 171–172, 174, 197, 281–282, 284, 306, 334–335
 - Changed names 283
 - complex type objects 201
- wsdlsoap
 - address location 199, 283, 307
 - body encodingStyle 198–199, 283
 - operation soapAction 198–199, 283
- WTS Production 139, 141, 226, 228

X

- XFDL 12
- XFDL document 293, 315
- XFDL file 119–120, 146, 157, 274, 292
 - assigned data instance 314
- XFDL form 18–19, 87, 95, 150, 164, 234, 275, 277, 286, 289
 - change 178
 - corresponding data instances 177, 289
 - data 166
 - data instance name 289
 - DB2 189
 - following data instance 182
 - intercept requests 23
 - specified XML instances 293
 - template 286, 289, 298, 330
 - Web service invocation 172
- XFDL template 288, 299
 - data instances 305
- XFDL theXFDL 119, 122, 182, 185, 317
- XFDL xmlns 18, 125, 183, 293, 306
- XForms 2, 9, 12
- xforms
 - instance xmlns 104, 125, 177–178, 254, 263, 305–306
- XForms - Business Benefits & Customer Value 11
- XForms + XFDL in Alignment with SOA 12
- XML Data
 - dialogue 105
 - fragment 43
 - Instance 32, 46, 201
 - Model dialogue 105, 264
- XML data
 - model 12, 66, 102–104, 199, 262–263
 - schema 11
- XML data model 12
- XML document 42
- XML fragment 164, 183–184, 299, 301
- XML instance 103, 162, 167, 234, 293

- data element 105
- tag name 162
- XML model 66, 104, 201
- XML Parser 22, 181, 274
- XML Schema Validation 106
- XML version 18, 125, 177, 179, 214, 282, 293

Z

- Zero Footprint with WebForm Server 241
- Zero-Footprint Display of Workplace Forms 36
- Zero-Footprint Solution 242



IBM Workplace Forms: Guide to Building and Integrating a Sample Workplace Forms Application

(0.5" spine)

0.475" <-> 0.873"

250 <-> 459 pages



IBM Workplace Forms:

Guide to Building and Integrating a Sample Workplace Forms Application

Features and functionality

Designing forms

Integration topics

This IBM Redbook describes the features and functionality of Workplace Forms and each of its component products. After introducing the products and providing an overview of features and functionality, we discuss the underlying product architecture and address the concept of integration.

To help potential users, architects, and developers better understand how to develop and implement a forms application, we introduce a specific scenario based on a “Sales Quotation Approval” application. Using this base scenario as a foundation, we describe in detail how to build an application that captures data in a form, then applies specific business logic and workflow to gain approval for a specific product sales quotation.

Throughout the scenario, we build upon the complexity of the application and introduce increasing integration points with other data systems. Ultimately, we demonstrate how an IBM Workplace Forms application can integrate with WebSphere Portal, IBM DB2 Content Manager, and Lotus Domino.

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

SG24-7279-00

ISBN 0738495603