



Adobe

Overview

July 2006

Adobe® LiveCycle™ Forms

Version 7.2

© 2006 Adobe Systems Incorporated. All rights reserved.

Adobe® LiveCycle™ Forms 7.2 Overview for Microsoft® Windows®, UNIX®, and Linux®
Edition 3.0, July 2006

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names and company logos in sample material or in the sample forms included in this software are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group.

All other trademarks are the property of their respective owners.

This product includes code licensed from RSA Security, Inc.

Some portions licensed from IBM are available at <http://oss.software.ibm.com/icu4j/>

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Preface	4
1 About LiveCycle Forms.....	5
Working environments	5
Development environment	6
Run-time environment.....	6
End-user environment	7
How LiveCycle Forms processes a request	7
2 LiveCycle Forms Integration	9
Integrating with other Adobe products.....	9
Process management.....	9
LiveCycle Designer	9
LiveCycle Reader Extensions	10
LiveCycle Form Manager	11
LiveCycle Workflow.....	11
Document security and control.....	12
LiveCycle Document Security	12
LiveCycle Policy Server	13
Integrating with LiveCycle Forms.....	13
Glossary	15

Preface

This guide provides an overview of Adobe® LiveCycle™ Forms, part of the Adobe LiveCycle suite of products.

What's in this guide?

This guide contains specific information about LiveCycle Forms:

- Capabilities and features
- Working environments
- Integration with other Adobe products

Who should read this guide?

This guide is intended for system administrators and developers. However, the information is useful to anyone who needs an introduction to LiveCycle Forms and how it integrates with other Adobe products.

Related documentation

In addition to this overview, the following resources provide information about LiveCycle Forms:

For information about	See
Understanding how to use the LiveCycle Forms APIs to create custom applications	<i>Developing Custom Applications</i>
Installing, configuring, and administering in a development and run-time environment	<i>Installing and Configuring LiveCycle for JBoss</i> <i>Installing and Configuring LiveCycle for WebSphere</i> <i>Installing and Configuring LiveCycle for WebLogic</i>
The Form Server Module API, including a description and explanation of its classes and methods	<i>Form Server Module API Reference</i>
The XML Form Module API, including a description and explanation of its classes and methods	<i>XML Form Module API Reference</i>
The new features in this product release	<i>What's New</i>
The form objects and associated properties that are supported in each web browser.	<i>Transformation Reference</i>
Patch updates, technical notes, and additional information on this product version.	www.adobe.com/support/products/enterprise/index.html

1

About LiveCycle Forms

LiveCycle Forms enables organizations to extend their intelligent data capture processes by deploying electronic forms in PDF or HTML format over the Internet. With LiveCycle Forms, end users can easily access online forms without downloading any additional software, fill them online, or save them to fill offline.

Form data can be submitted to an organization's core systems via LiveCycle Forms, thereby increasing the quality of data gathered, improving customer service, and leveraging investment in core systems. By enabling end users to access and submit forms online, LiveCycle Forms enables businesses and government to reach all customers and citizens using their web browsers (on any platform or any device) without requiring them to download any proprietary software or plug-ins.

LiveCycle Forms offers many key features:

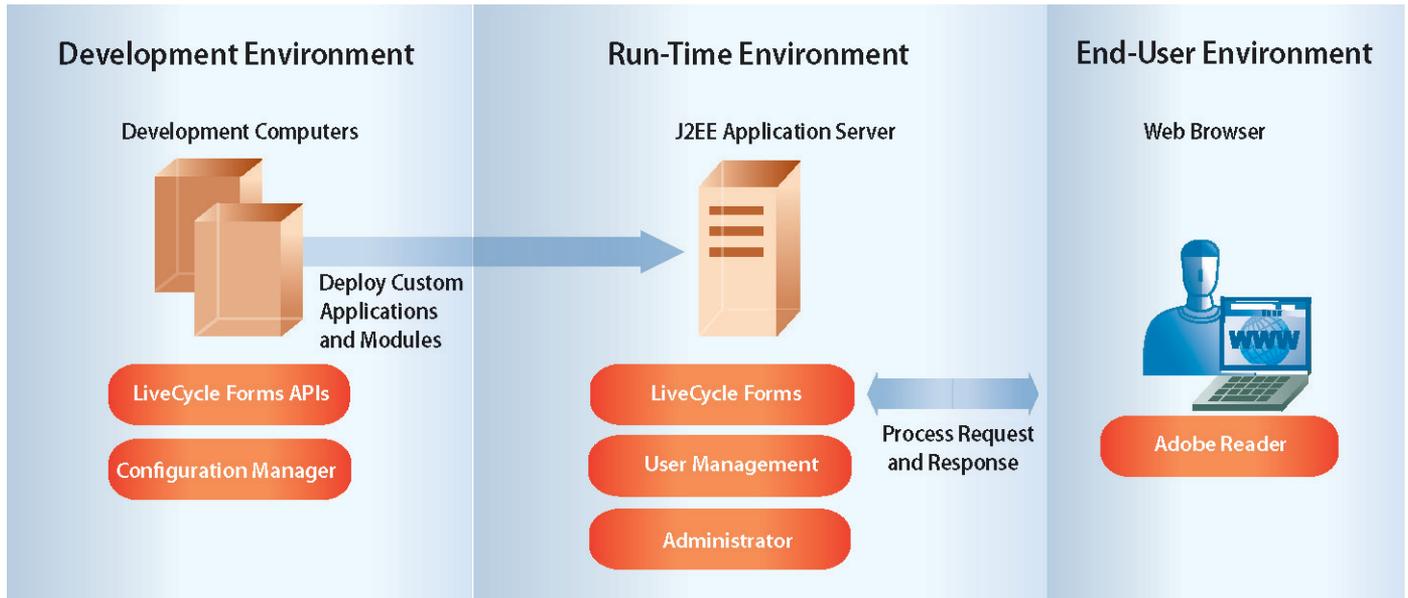
- Enables interactive access to documents using a web browser
- Automatically detects the browser type and platform, and then dynamically generates a HTML document that is based on a form design (typically created in Adobe LiveCycle Designer)
- For dynamic subforms created in Adobe LiveCycle Designer, adds extra fields and boilerplate as a result of merging the form design with data or as a result of scripting
- Detects whether form design scripts should run on the client or on the server
- Validates data entry by performing calculations, accessing databases, or enforcing business rules on field-level data, and then returns the resulting data to the browser
- Extracts submitted form data as XML
- Imports, exports, and links third-party XML schemas and data

Working environments

LiveCycle Forms consists of three working environments:

- Form-based applications are developed in the *development environment*.
- Requests are initiated in the *end-user environment*.
- The applications are processed in the *run-time environment*.

The following diagram illustrates these working environments:



Development environment

The development environment includes tools that enable developers and form authors to work together to create form-based applications tailored to any business requirements. LiveCycle Forms consists of modules, APIs, and configuration tools.

The modules are installed and configured in the development environment and deployed to the run-time environment. You deploy the modules to a Java 2 Enterprise Edition (J2EE) application server where they run as J2EE services.

The APIs included in the LiveCycle Forms installation help you to write the code that invokes particular services that are deployed to the application server.

Adobe Configuration Manager is installed along with LiveCycle Forms. You use Configuration Manager to configure and package most LiveCycle Forms modules into a single EAR file that you can deploy to a J2EE application server. For more information about using Configuration Manager, see the *Installing and Configuring* guide for your application server.

Run-time environment

The LiveCycle Forms run-time environment consists of modules that are deployed and executed on a J2EE application server. Depending on the steps required to process a request, LiveCycle Forms uses different modules. The services provide the functionality for client-side and server-side execution of documents that are rendered as PDF or HTML. Using configuration tools, administrators and developers can configure and administer the modules.

Adobe Administrator enables an administrator to configure optional run-time settings associated with LiveCycle Forms. For example, using Administrator, you can define the web context of a LiveCycle Forms custom application. For more information about Administrator, see the *Installing and Configuring* guide for your application server.

Adobe User Management allows administrators to maintain a database for all users and groups, synchronized with one or more third-party user directories. User Management provides authorization and user management for LiveCycle products, including Adobe LiveCycle Workflow, LiveCycle Forms, and Adobe LiveCycle Form Manager.

Note: Administrators access User Management from within Administrator.

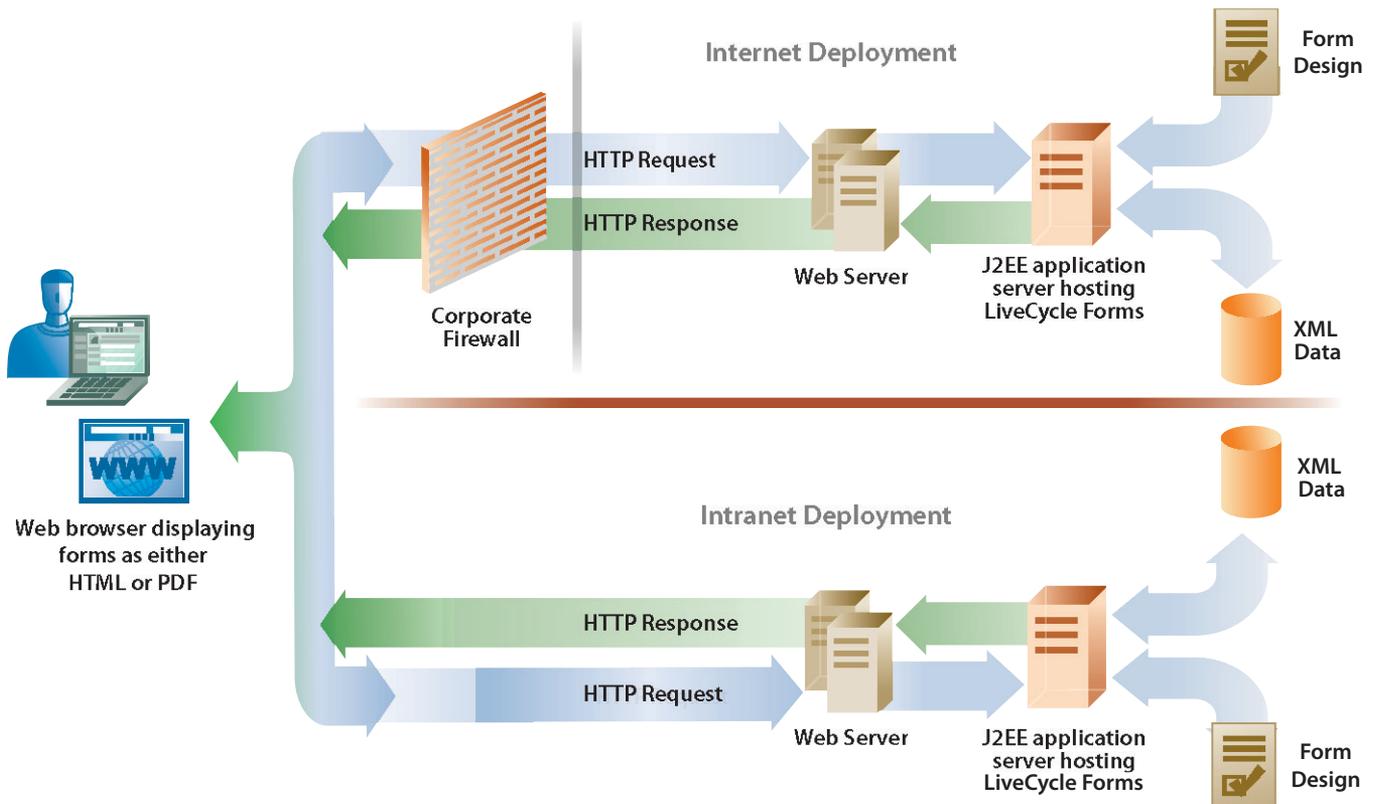
End-user environment

The LiveCycle Forms end user environment consists of a web browser (for HTML forms) together with Adobe Reader® (for PDF forms). LiveCycle Forms can detect the browser type and dynamically generate a PDF or HTML document of the form design created in LiveCycle Designer.

How LiveCycle Forms processes a request

When end users request a document from LiveCycle Forms (for example, by clicking a button or an image on a web page), the request initiates a series of specific processes and interactions among the web application, LiveCycle Forms, and the web browser. After receiving the form, end users can interact with it online. After end users are finished with the form, they submit it, along with form data, back to LiveCycle Forms.

The following diagram provides an example of how LiveCycle Forms processes a request from an end user:



These are the steps to initiate and process the request:

1. The end user accesses a web page and requests a form.
2. The web application invokes LiveCycle Forms and requests the form.
3. LiveCycle Forms retrieves the form design from a repository and data (possibly from an enterprise database), then merges the form design with the data to prepopulate parts of the form. The data can come from a variety of sources, such as an enterprise database, another form, or another application.
4. LiveCycle Forms determines the format in which to render the prepopulated form based on the browser information that is passed with the call. The format of a form can also be set programmatically by using the Form Server Module API. For information, see *Developing Custom Applications*.
5. LiveCycle Forms transforms the form design into PDF or HTML and then returns the prepopulated form to the end user.
6. The end user completes the form and then submits the form data back to LiveCycle Forms. Before form data is submitted back to LiveCycle Forms, applicable client-side scripts are executed. For example, a user may be prompted to provide a value for a mandatory form field.
7. The form data is submitted to LiveCycle Forms.
8. LiveCycle Forms extracts the submitted data, runs any server-side scripts associated with the button that was clicked, then executes the calculations and validations on the form. For information about calculating form data, see *Developing Custom Applications*.
9. LiveCycle Forms returns results. If validations fail, the result may be a form that is returned to the end user. However, if validations are successful, then the result may be XML data.

2

LiveCycle Forms Integration

LiveCycle Forms is part of the Adobe LiveCycle suite of products that can help your organization to efficiently and effectively deliver intelligent documents to end users and retrieve information from end users. Adobe LiveCycle products enable your organization to create and integrate intelligent documents (including the data) into your enterprise applications and business processes.

Intelligent documents are either non-interactive (end users cannot edit the document) or interactive (end users can interact with the document), depending on the intentions of the author. Adobe Reader or a web browser provide access to intelligent documents within and outside your organization.

Here are examples of intelligent documents:

- Expense report forms that automatically calculate sums
- Dynamic forms that contain information personalized for the end user
- Contractual documents that can track comments and approvals from suppliers and customers while protecting the original source document from unauthorized changes

Integrating with other Adobe products

LiveCycle Forms, combined with other LiveCycle products, enables your organization to integrate enterprise applications with document processes throughout the organization, improving document generation and process management. LiveCycle Forms can also integrate with other products, such as Adobe LiveCycle Document Security, to improve security within your organization.

Process management

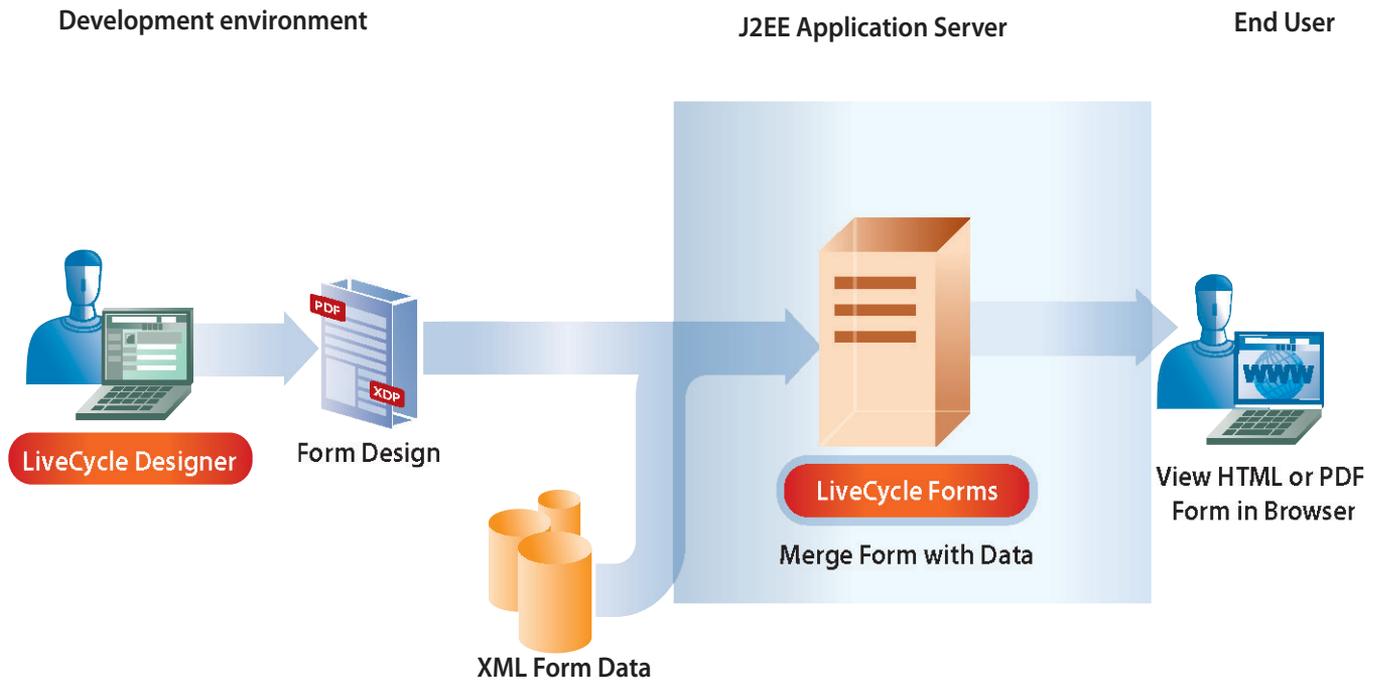
The four LiveCycle products that are included with process management and integrate with LiveCycle Forms are LiveCycle Designer, Adobe LiveCycle Reader Extensions, LiveCycle Form Manager, and LiveCycle Workflow. This section describes how these products integrate with LiveCycle Forms.

LiveCycle Designer

LiveCycle Designer is a graphical form design tool that simplifies the design and layout of forms for use with LiveCycle Forms and other Adobe products. It has an easy-to-use interface that enables form authors to quickly create and maintain form designs. The form author defines a form's business logic and can preview the form before it is deployed.

The form author can deploy the form designs for use with LiveCycle Forms either as XDP files or PDF files, depending on the requirements of the business process. LiveCycle Forms can render an XDP file either as an HTML form or PDF form. A PDF file is rendered as a PDF form.

The following diagram shows deploying form designs for use with LiveCycle Forms:



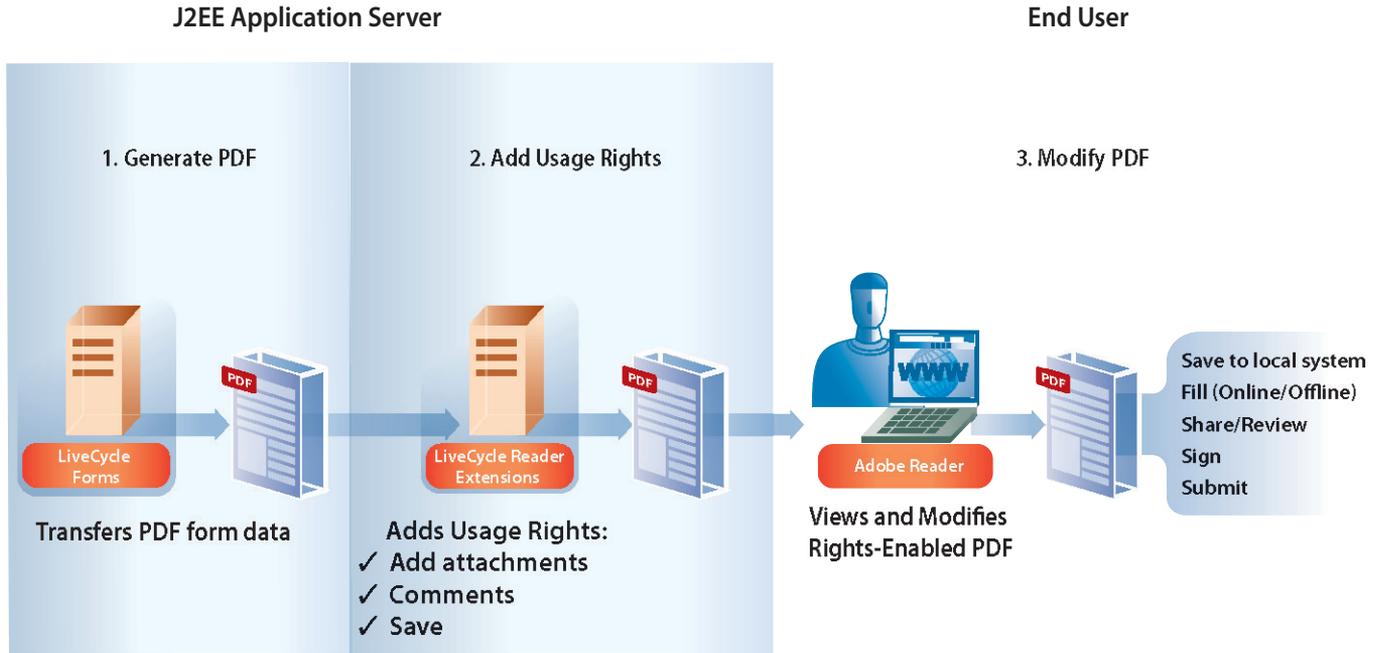
LiveCycle Reader Extensions

LiveCycle Reader Extensions enables organizations to extend the functionality of Adobe Reader by adding usage rights to PDF documents. Usage rights are permissions that enable the recipients of PDF documents within or outside organizations to access features in PDF documents that are not usually available in Adobe Reader.

With LiveCycle Reader Extensions, administrators can access a wizard-like web application to add usage rights to PDF documents. LiveCycle Reader Extensions also provides a set of Java APIs to programmatically add usage rights to PDF documents either in batches or in real-time.

By integrating a LiveCycle Forms custom application with LiveCycle Reader Extensions, end users can fill PDF forms offline, digitally sign PDF documents, save the data locally, and comment on PDF documents using Adobe Reader.

The following diagram shows LiveCycle Forms integrating with LiveCycle Reader Extensions:



LiveCycle Form Manager

LiveCycle Form Manager provides a central repository where administrators can organize and store form designs as well as control form design versions. Form authors can deploy their form designs (including images) to the LiveCycle Form Manager repository from LiveCycle Designer. Once deployed to the repository, the form designs are ready for use with LiveCycle Forms.

LiveCycle Form Manager enables end users to find, organize, and process the forms that they need from their web browser. Using LiveCycle Form Manager, end users can perform these tasks:

- Open and fill forms.
- Find forms quickly and easily.
- Reuse forms they have previously filled.
- Customize the presentation of forms.

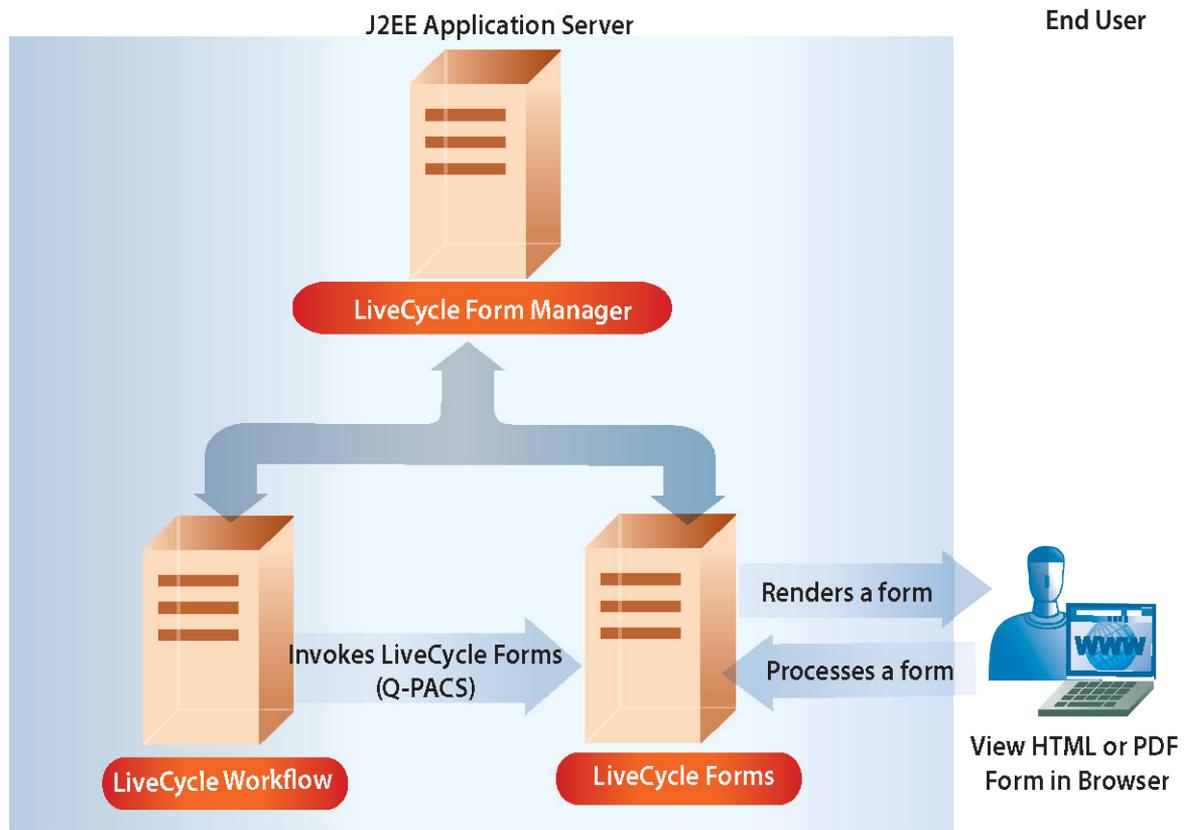
LiveCycle Workflow

Adobe LiveCycle Workflow enables an organization to automate business processes to improve organizational productivity. LiveCycle Workflow automatically routes forms to users, stores XML-based process information in a database, and integrates with legacy systems such as email servers, database servers, and LDAP servers, as well as with other LiveCycle products. LiveCycle Workflow provides tools for designing, deploying, and administering electronic processes, and provides business application monitoring (BAM) tools for monitoring process metrics using real-time information.

LiveCycle Workflow uses Quick Process Action Components (QPACs) to invoke LiveCycle Forms. QPACs are components that define a step of a business process. Using LiveCycle Workflow QPACs for LiveCycle Forms, you invoke the following functionality:

- Render a form to a client device, such as a web browser.
- Process a form that is submitted from a client device, such as a web browser.

The following diagram shows LiveCycle Workflow and LiveCycle Form Manager integrating with LiveCycle Forms:



Document security and control

The two LiveCycle products that are included with document security and control and integrate with LiveCycle Forms are LiveCycle Document Security and Adobe LiveCycle Policy Server. This section describes how these products integrate with LiveCycle Forms.

LiveCycle Document Security

LiveCycle Document Security ensures document authenticity, integrity, and confidentiality by providing all the digital signature and encryption capabilities of Adobe Acrobat® Professional or Acrobat Standard in a server environment. LiveCycle Document Security can also attach a policy that is created by LiveCycle Policy Server to a PDF document.

PDF documents can be signed and certified on the server. The technology used to digitally sign documents ensures that both the signer and recipients can be clear about what was signed, and that the document was not altered since it was signed.

PDF documents can be encrypted on the server for specific recipients and can also be decrypted on the server. When a document is encrypted, its contents become unreadable. Only an authorized user can decrypt the document to obtain access to the contents.

LiveCycle Document Security can also verify certified and encrypted PDF documents that originate from end users. The server-based certificate validation ensures that the certificate of the author is valid and that the document was not modified during transmission.

LiveCycle Policy Server

LiveCycle Policy Server is a web-based security system that enables users to dynamically apply confidentiality settings to their PDF documents, and maintain control over the documents, no matter how widely they are distributed.

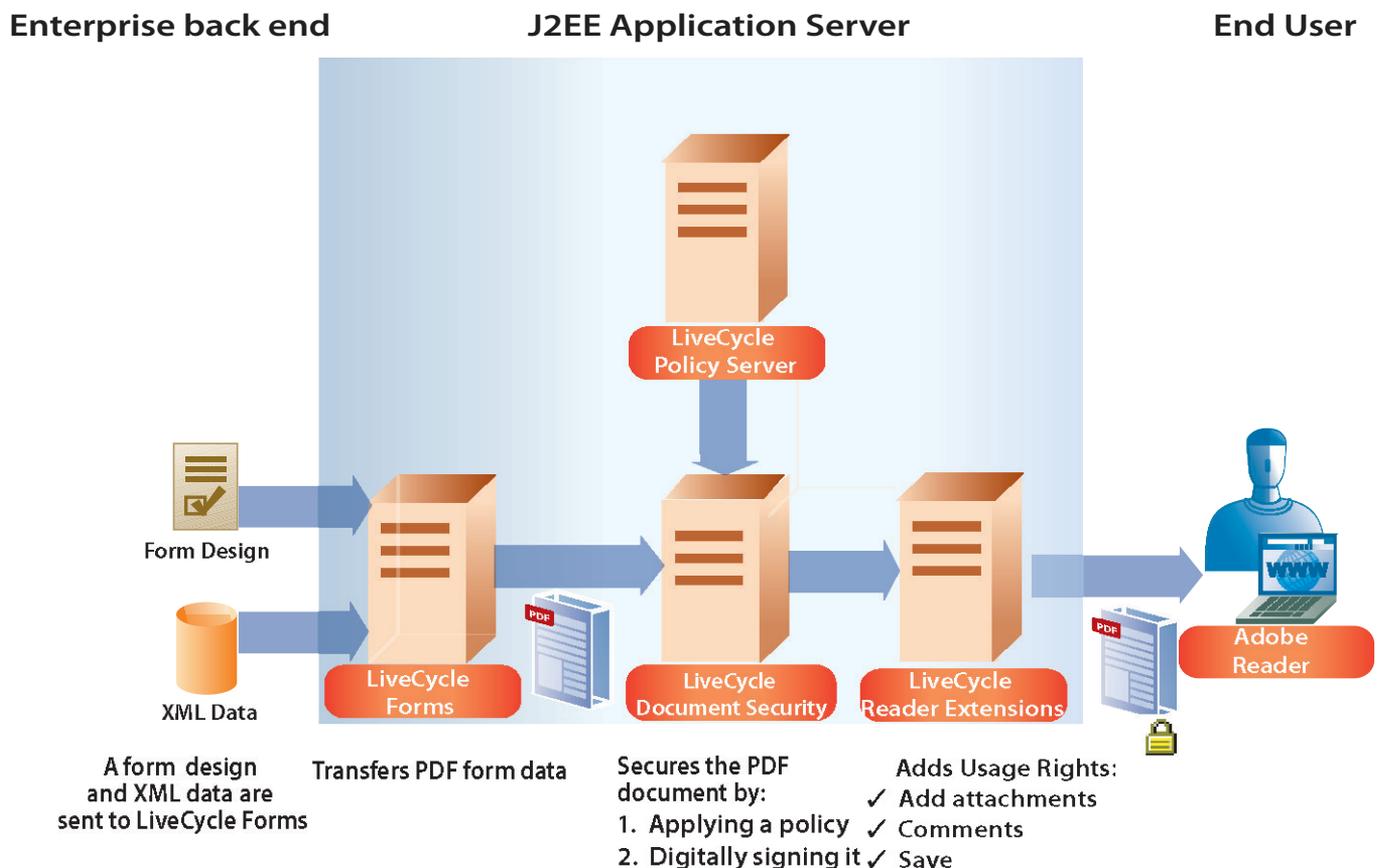
LiveCycle Policy Server prevents information from spreading beyond the users' reach by enabling them to maintain control over how recipients use the policy protected PDF document. Users can specify who can open a document and how they can use it, and monitor the document after they distribute it. They can also dynamically control access to a policy protected document, and even revoke access to the document if necessary.

Because PDF documents can contain many forms of information, such as text, audio, or video files, LiveCycle Policy Server enables users to safely distribute and maintain control of any information that is saved in the PDF document.

Using client applications (Acrobat), users are able to protect PDF documents by applying security policies (a collection of pre-defined user access and confidentiality settings). They can also use Acrobat or Adobe Reader to access policy-protected documents. Client applications also communicate with the server component to authenticate end users, provide event information to the server for auditing purposes, and determine if access to documents is authorized.

Integrating with LiveCycle Forms

The following diagram shows LiveCycle Forms integrating with LiveCycle Document Security and LiveCycle Policy Server:



In this scenario, LiveCycle Forms merges XML data with a form design (that was created by using LiveCycle Designer) and transfers the PDF form data to LiveCycle Document Security. LiveCycle Document Security saves the PDF form data as a PDF document and then applies a policy and digitally signs it. LiveCycle Reader Extensions then adds user rights to the PDF document. The secured PDF document is sent to users who can interact with it using Acrobat. For information about transferring PDF form data, see *Developing Custom Applications*.

Glossary

This glossary contains terminology definitions that are specific to documentation for the Adobe LiveCycle suite of products. These terms may have different meanings in other contexts, but they have restricted meanings in this documentation.

A

accessible forms

Forms that users with disabilities or vision impairment can view and fill using screen readers and other assistive technologies. See also *tagged Adobe PDF form*.

Acrobat form

A PDF document, created in Acrobat, that contains one or more form fields. The PDF document may also contain non-form content.

action

In a workflow, the representation of a step in a business process.

Adobe certified document

A document that is signed with a specific Adobe root certificate. An Adobe certified document provides a strong guarantee as to the authenticity and immutability of the document. See also *certificate*.

Adobe document services

Adobe document services extend the value of core enterprise systems to ensure more secure, reliable, and efficient use of business-critical information across the extended enterprise. Adobe document services include the Adobe LiveCycle suite of products and the Acrobat product line.

application

A set of generally interdependent files that make up a self-contained application that Adobe LiveCycle products can run. Applications may include files such as form designs, Java Server Pages, HTML pages, servlets, and images.

B

branch

A branch contains a set of actions interconnected by routes, representing a sequential path taken by a process at execution. The branch always determines the behavior of the workflow.

C

certificate

An electronic file that establishes your identity, by binding your identity to your public key, when doing business or other transactions on the web. A *certificate* (or sometimes called a *digital certificate*) is issued by a certificate authority (CA). See also *Adobe certified document*.

client

The requesting program in a client/server relationship. A web browser is an example of a client application.

credential

The file that contains a private key. (The corresponding public key is contained in a certificate.) A private key is what one principal presents to another used to establish identity in decryption and signing operations. Credentials are issued by an authentication agent or a certification authority. See also *certificate*.

D

deadline

The time by which a person must complete a work item. Deadlines are properties of workflows.

dynamic form

A form that can expand or contract to reflect the amount of incoming data. See also *interactive form*.

E

ebXML

Electronic Business using eXtensible Markup Language (ebXML). A modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. See also *registry*.

encryption

The conversion of data into a format (called a ciphertext) that cannot be easily understood by unauthorized persons. The conversion is done using an encryption algorithm.

F

form

An electronic document that captures and delivers data. A person may add data to an interactive form, or a server process may merge a form design with data to produce a form.

form authors

LiveCycle Designer users who are capable of creating fillable forms to be used in Acrobat or Adobe Reader, and simple non-interactive forms for deployment to LiveCycle Forms. See also *form developers*.

FormCalc

A calculation language similar to that used in common spreadsheet software that facilitates form design without requiring a knowledge of traditional scripting techniques or languages.

form design

The design-time version of a form that an author or developer creates in LiveCycle Designer.

form developers

LiveCycle Designer users who are capable of creating complex form-based applications for use in different environments. See also *form author*.

form object

A form element, such as a button or text field, that you can place on a form. An object has its own set of properties and events.

I

interactive form

A form that a person can interact with and complete electronically.

N

non-interactive form

A form that a person can view or print but cannot fill electronically. Non-interactive forms can be merged or prepopulated with data, but the data cannot be changed by a user. Non-interactive forms are designed for output.

P

PDF document

Portable Document Format. A file conforming to the PDF specification as published by Adobe Systems or a file conforming to the XDP specification, containing exactly one PDF packet and no more than one each XFA-Template, XFA-Configuration, XFA-SourceSet, and Annotations packets.

PDF form

A form that users can access in Acrobat and Adobe Reader. PDF forms are either interactive or non-interactive.

permissions

Security settings applied, for example, to restrict users from opening, editing, printing, or removing encryption from a PDF file. Permissions cannot be changed unless the user has the Permissions password. Permissions can be set in LiveCycle Designer, Acrobat, LiveCycle Document Security, and other products.

policy

Defines a set of security permissions and users who can access a PDF document to which the policy is applied. Policies are created using LiveCycle Policy Server and can be applied to documents using LiveCycle Policy Server, LiveCycle Document Security, or Acrobat 7.0 or later.

prepopulated form

A form that appears to the user with some or all fields automatically populated with data.

Q

QPAC

Quick Process Action Component. A JAR file that contains server-side code and client-side code for use with LiveCycle Workflow. In LiveCycle Workflow Designer, QPACs provide action components that can be added to workflows to represent a step in a process. LiveCycle Workflow Server interprets each action of the workflow and executes the server-side code of the corresponding QPACs. QPACs enable LiveCycle Workflow to interact with other Adobe LiveCycle products, such as LiveCycle Forms and LiveCycle Barcoded Forms.

R

reminder

A notification sent to people that reminds them to complete a work item. Reminders are properties of workflows.

render

An action whereby LiveCycle Forms merges a form design, possibly with data, to display a form in PDF or HTML format in a browser.

registry

An ebXML-compliant repository of shared information that provides services for the purpose of enabling business process integration between interested parties. See also *ebXML* and *repository*.

repository

The underlying storage area within a registry. See also *registry*.

restricted document

A PDF document with password security restrictions (permissions) that prevent the document from being opened, printed, or edited.

rights-enabled document

A PDF document that includes security extensions that enable Adobe Reader users to fill forms, add comments, and sign documents.

route

The path between actions on a workflow. Routes determine the order in which LiveCycle Workflow Server executes actions at run time.

run time

For form rendering, the time when an application or server process retrieves a form design, possibly merges it with data, and presents it to a user for viewing or filling.

S

split

A segment in a workflow that contains one or more branches. The branches in a split are executed in parallel.

static form

A form that remains exactly as it was designed. The layout does not change according to the amount of incoming data.

subform

An object that can act as a container for form objects and other subforms. A subform helps to position form objects relative to each other and provide structure in dynamic form designs. A subform can also provide a reference point, when binding data to a form, by restricting the scope for a field so that it matches that of the corresponding data node.

T

tagged Adobe PDF form

Includes a logical structure and a set of defined relationships and dependencies among the various elements, plus additional information that permits reflow. See also *accessible forms*.

turnkey

An installation option that automatically installs and configures the LiveCycle product files, JBoss application server, and MySQL database, and deploys the product files to JBoss. After you perform a turnkey installation, the LiveCycle product is ready to use.

U

usage rights

Rights that extend the functionality of Adobe Reader and enable users to save forms with data, add comments, and sign documents.

W

workflow

The electronic representation of a business process. Workflows are created using LiveCycle Workflow Designer.

X

XDP file

XML Data Package. LiveCycle Designer saves form designs as either XDP files or PDF files. LiveCycle Forms uses XDP files to render forms in PDF or HTML format.

XML Forms Architecture

Represents the underlying technology beneath the Adobe XML forms solution. It enables the construction of robust and flexible form-based applications for use on either the client or the server.

XML form

A PDF form that conforms to the Adobe PDF specification and the Adobe XML Forms Architecture. XML forms are typically created in LiveCycle Designer. XML forms can have the file name extension .xdp or .pdf.



Note:

The Form Server Module API is not described in this guide. You can locate reference information that describes the Form Server Module API by opening the `index.html` page in the `documentation/javadocs` directory.

XML Form Module API Reference

© 2006 Adobe Systems Incorporated. All rights reserved.

Adobe® LiveCycle™ Forms 7.2 XML Form Module API Reference for Microsoft® Windows®, UNIX®, and Linux®
Edition 3.0, July 2006

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names and company logos in sample material or in the sample forms included in this software are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, LiveCycle, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

IBM is a trademark of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group.

All other trademarks are the property of their respective owners.

This product includes code licensed from RSA Security, Inc.

Some portions licensed from IBM are available at <http://oss.software.ibm.com/icu4j/>

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Preface	5
What's in this guide?	5
Who should read this guide?	5
Related documentation	5
1 XML Form Module API	7
FormFactory interface	7
create	7
createDefault	8
Form interface	9
clearMessages	9
getMessages	9
exportXDP	9
exportXML	11
getConfigValue	12
getPacketList	13
getPageCount	13
importPackets	14
isPacketPresent	14
render	15
setConfigValue	16
ReturnStatus	17
XDP packets	17
Configuration options	18
Defaults	18
Scripting interface	19
Configuration options reference	19
Configuration options syntax	20
data.outputXSL.uri	20
data.range	21
data.record	21
data.startNode	22
data.xsl.debug.uri	22
data.xsl.uri	23
destination	23
locale	23
pdf.compression.compressLogicalStructure	23
pdf.compression.level	23
pdf.compression.type	24
pdf.encryption.encrypt	24
pdf.encryption.encryptionLevel	24
pdf.encryption.masterPassword	24
pdf.encryption.permissions.accessibleContent	24
pdf.encryption.permissions.contentCopy	25
pdf.encryption.permissions.documentAssembly	25
pdf.encryption.permissions.formFieldFilling	25

1 XML Form Module API (Continued)

Configuration options (Continued)

Configuration options syntax (Continued)

pdf.encrypted.permissions.modifyAnnots	25
pdf.encrypted.permissions.plaintextMetadata	26
pdf.encrypted.permissions.print	26
pdf.encrypted.permissions.printHighQuality	26
pdf.encrypted.permissions.change	26
pdf.encrypted.userPassword	27
pdf.fontInfo.embed	27
pdf.fontInfo.encodingSupport	27
pdf.fontInfo.map.equate	27
pdf.fontInfo.subsetBelow	28
pdf.interactive	28
pdf.openAction.destination	28
pdf.submitFormat	29
pdf.tagged	29
pdf.xdc.uri	29
temp.uri	29
template.base	29

2 Data Manager Module API 30

DataManager interface	30
createFileDataBuffer	30
createFileDataBufferFromUrl	31
getTempFileName	31
manageTempFile	32
DataBuffer interface	32
getBufLength	32
getBytes	32
getContentType	33
setContentType	33
FileDataBuffer interface	34
getFilepath	34
DMUtils class	34
getDataBuffer	34
getDataHandler	35
getInputStream	35

3 Connection API 36

ConnectionFactory interface	36
getConnection	36

Index 38

Preface

This guide is one of several resources available to help you learn about Adobe® LiveCycle™ Forms.

What's in this guide?

This guide provides a reference of the XML Form Module API and Data Manager Module API. Both of these APIs belong to LiveCycle Forms. The following table provides a brief description of when you would use these APIs in your custom applications.

Product module	When to use the API
XML Form Module	Use this API to create non-interactive applications that render PDF forms.
Data Manager Module	Use this API (with the Connection API) to invoke the XML Form Module.

Note: The Form Server Module API is not described in this guide. You can locate reference information that describes this API by viewing *Form Server Module API Reference*. To view *Form Server Module API Reference*, open the index.html page in the documentation/javadocs directory.

Who should read this guide?

This guide is for developers who want to develop applications that interact with the XML Form Module and is a companion guide to *Developing Custom Applications*.

Related documentation

The resources in this table can help you learn about LiveCycle Forms.

For information about	See
Understanding what LiveCycle Forms is and how it integrates with other Adobe products	<i>Overview</i>
Understanding how to use the LiveCycle Forms APIs to create custom applications	<i>Developing Custom Applications</i>
Installing, configuring, and administering LiveCycle Forms in a development and run-time environment	<i>Installing and Configuring LiveCycle for JBoss</i> <i>Installing and Configuring LiveCycle for WebSphere</i> <i>Installing and Configuring LiveCycle for WebLogic</i>
The Form Server Module API, including a description and explanation of its classes and methods	<i>Form Server Module API Reference</i> (in the documentation/javadocs directory)

For information about	See
New features in this product release	<i>What's New</i>
Form objects and associated properties that are supported in each web browser.	<i>Transformation Reference</i>
Other services and products that integrate with LiveCycle Forms	www.adobe.com
Patch updates, technical notes, and additional information on this product version	www.adobe.com/support/products/enterprise/index.html

1

XML Form Module API

The XML Form Module API lets you create custom applications that can process and work with non-interactive forms containing large data sets. Using the XML Form Module API, you can create applications that perform non-interactive, form-rendering operations such as these:

- Load XML data into an XML Data Package (XDP) file.
- Load XML data into an Adobe PDF file that contains XDP information.
- Control configuration and data-loading options.
- Render PDF documents.
- Extract XML data from an XDP file.

Caution: The XML Form Module API is deprecated. As a result, it is recommended that you use the Form Server Module API. The Form Server Module API is not described in this guide. You can locate reference information that describes the Form Server Module API by opening the `index.html` page in the `documentation/javadocs` directory.

The XML Form Module API consists of two interfaces: `FormFactory` and `Form`. The `FormFactory` interface has two methods: `create` and `createDefault`. Using either of these methods, you create a `Form` object. For information about creating a `FormFactory` object or a `Form` object, see the “Invoking LiveCycle Forms” chapter in *Developing Custom Applications*.

A `Form` object enables you to perform specific tasks, such as exporting data into a form (either an XDP file or a PDF file). When working with a `Form` object, you also work with objects that belong to the Data Manager API. For example, you call the `exportXDP` method to export data into a form. The `exportXDP` method returns a `FileDataBuffer` object that contains the data to export. The `FileDataBuffer` object belongs to the Data Manager API. For information, see [“Data Manager Module API” on page 30](#).

Knowledge of some Adobe XML Forms Specifications is a prerequisite to working with the XML Form Module API. For example, some XML Form Module API methods accept XDP packet names as parameters. XDP (one of the XML Forms Specifications) is an XML format that provides a mechanism for packaging units of content (known as XDP packets) within a surrounding XML container. These packets enable the XML Form Module API to determine how to process the input file. For a summary of each packet, see [“XDP packets” on page 17](#).

FormFactory interface

The `FormFactory` interface lets you create `Form` objects.

create

Creates a `Form` object and imports the packets listed by `packetNames` from the data specified by the `inputData` parameter.

Syntax

```
public Form create(FileDataBuffer inputData, PacketList packetNames)
    throws InvalidXDPException, InvalidPackageNameException
```

Parameters

<code>inputData</code>	Input data to load. These data formats are acceptable: <ul style="list-style-type: none">● PDF with XDP information● XDP● XML
<code>packetNames</code>	List of packets to load. These packet names are valid: <ul style="list-style-type: none">● <code>template</code>● <code>datasets</code>● <code>stylesheets</code>● <code>xfdf</code> (annotations)● <code>sourceSet</code>● <code>pdf</code>● <code>config</code> To include all packets, specify an asterisk (*). For a description of each packet, see “XDP packets” on page 17 .

Returns

A `Form` object.

Details

The typical usage of this method is to pass the configuration file as a `FileDataBuffer` object in the first argument and `config` as the packet list. For information, see the “Rendering Non-Interactive Forms” chapter in *Developing Custom Applications*.

Throws

`InvalidXDPEXception` if input data is not well-formed XML or if PDF does not contain XDP data.

`InvalidPacketNameException` if an invalid packet name is specified.

createDefault

Creates a `Form` object that uses default configuration values. For information about these configuration values, see [“Configuration options” on page 18](#).

Syntax

```
public Form createDefault()
```

Returns

A `Form` object.

Note: For information about using this method to create a `Form` object, see the “Invoking LiveCycle Forms” chapter in *Developing Custom Applications*.

Form interface

You use the `Form` interface to manipulate configuration values, render forms, and import or export data as an XDP or XML file.

clearMessages

Clears the messages that have accumulated since the `Form` object was created or the last time this method was called.

Syntax

```
void clearMessages()
```

getMessages

Returns the messages that have accumulated since the `Form` object was created or the last time this method was called.

Syntax

```
public FileDataBuffer getMessages()
```

Returns

A `FileDataBuffer` object that contains XML Forms Architecture messages.

exportXDP

Exports the packets specified by the `packetNames` parameter to a `FileDataBuffer` object in XDP format.

Syntax

```
public FileDataBuffer exportXDP(PacketList packetNames)  
    throws PacketNotFoundException, InvalidPackageNameException,  
    ExportException
```

Parameters

packetNames	List of packets to export. These packet names are valid: <ul style="list-style-type: none">● template● datasets● stylesheets● xfdf (annotations)● sourceSet● pdf● config To include all packets, specify an asterisk (*). For a description of each packet, see "XDP packets" on page 17 .
-------------	---

Returns

A `FileDataBuffer` object containing the exported data.

Throws

`PacketNotFoundException` if the specified packet is not found.

`InvalidPacketNameException` if an invalid packet name is specified.

`ExportException` if an error occurs during the export process.

Example

The following example exports all packets except datasets to an XDP file:

```
try {
String exportPackets[] = new String[packetList.length];
int nCountPL = 0;
int nCountEP = 0;
while (nCountPL < packetList.length)
{
    if (packetList[nCountPL] != "datasets")
    {
        exportPackets[nCountEP] = packetList[nCountPL];
        nCountEP++;
    }
    nCountPL++;
}

FileDataBuffer exportXDP = Form.exportXDP(exportPackets);

// Get the path for the created file
String xdpFile = exportXDP.getFilePath();
System.out.println("XDP data successfully exported to " + xdpFile);
}
```

```
catch (PacketNotFoundException e) {  
    System.out.println("PacketNotFoundException: " + e.packetName); }  
  
catch (InvalidPackageNameException e) {  
    System.out.println("InvalidPackageNameException: " + e.packetName); }  
  
catch (ExportException e) {  
    System.out.println("ExportException: " + e.reason); }
```

exportXML

Exports the packets specified by the `packetName` parameter to a `FileDataBuffer` object in XML format.

Syntax

```
public FileDataBuffer exportXML(PacketList packetName)  
    throws PacketNotFoundException, InvalidPackageNameException,  
    ExportException
```

Parameters

<code>packetNames</code>	List of packets to export. These packet names are valid: <ul style="list-style-type: none">● <code>template</code>● <code>datasets</code>● <code>stylesheets</code>● <code>xdfd (annotations)</code>● <code>sourceSet</code>● <code>pdf</code>● <code>config</code> For a description of each packet, see "XDP packets" on page 17 .
--------------------------	--

Returns

A `FileDataBuffer` that contains the exported data.

Throws

`PacketNotFoundException` if the specified packet is not found.

`InvalidPackageNameException` if an invalid packet name is specified.

`ExportException` if an error occurs during the export process.

Example

The following example exports all packets except datasets to an XDP file:

```
try {
String exportPackets[] = new String[packetList.length];
int nCountPL = 0;
int nCountEP = 0;
while (nCountPL < packetList.length)
{
    if (packetList[nCountPL] != "datasets")
    {
        exportPackets[nCountEP] = packetList[nCountPL];
        nCountEP++;
    }
    nCountPL++;
}

FileDataBuffer exportXDP = Form.exportXML(exportPackets);

// Get the path for the created file
String xdpFile = exportXDP.getFilePath();
System.out.println("XDP data successfully exported to " + xdpFile);
}

catch (PacketNotFoundException e) {
    System.out.println("PacketNotFoundException: " + e.packetName); }

catch (InvalidPacketNameException e) {
    System.out.println("InvalidPacketNameException: " + e.packetName); }

catch (ExportException e) {
    System.out.println("ExportException: " + e.reason); }
```

getConfigValue

Retrieves a value from a node in the Configuration Data Object Model (DOM). The node from which a value is retrieved is specified as an XML Scripting Object Model (SOM) expression. For information, see ["Configuration options" on page 18](#).

Syntax

```
public String getConfigValue(SOMExpression path)
    throws InvalidSOMExpressionException
```

Parameters

path	A SOM expression that specifies a node in the Configuration DOM from which a value is retrieved.
------	--

Returns

The value of the specific node.

Throws

`InvalidSOMExpressionException` if the path value is invalid.

getPacketList

Returns a list of all XDP packets that are currently imported. For a description of each packet, see [“XDP packets” on page 17](#).

Syntax

```
public PacketList getPacketList()
```

Returns

Returns a list of all loaded packets.

Example

The following example shows this method populating a string array, named `packetList`, with packet values:

```
// Get the packet list
String packetList [] = Form.getPacketList();
String packets = "The packets are ";

int nCount = 0;
while (nCount < packetList.length)
{
    packets = packets + packetList[nCount];
    nCount++;
    if (nCount != packetList.length)
        packets = packets + ", ";
}

System.out.println(packets);
```

getPageCount

Returns the number of pages that the latest render method produced.

Syntax

```
public long getPageCount()
```

Returns

The number of pages in the rendered document.

importPackets

Imports the XDP packets specified by `packetNames` from the data specified by the `inputData` parameter.

Syntax

```
void ImportPackets(FileDataBuffer inputData, PacketList packetNames)  
    throws InvalidXDPEException, InvalidPacketNameException
```

Parameters

<code>inputData</code>	Input data to load. Data can be imported from one of the following sources: <ul style="list-style-type: none">● PDF files with XDP information● XDP files● XML files
<code>packetNames</code>	List of packets to load. These packet names are valid: <ul style="list-style-type: none">● <code>template</code>● <code>datasets</code>● <code>stylesheets</code>● <code>xfdf (annotations)</code>● <code>sourceSet</code>● <code>pdf</code>● <code>config</code> For a description of each packet, see “XDP packets” on page 17 .

Throws

`InvalidXDPEException` if the input data is not structured XML or if PDF does not contain XDP data.

`InvalidPacketNameException` if an invalid packet name is specified.

isPacketPresent

Determines whether a specific packet was imported.

Syntax

```
public boolean isPacketPresent(String packetName)
```

Parameters

packetName	Name of packet. These packet names are valid: <ul style="list-style-type: none">● template● datasets● stylesheets● xfdf (annotations)● sourceSet● pdf● config For a description of each packet, see “XDP packets” on page 17 .
------------	--

Returns

True if packet is loaded; otherwise, False.

Example

The following example determines whether a packet named `datasets` exists:

```
// Determine whether a datasets packet exists
boolean datasetsPresent = Form.isPacketPresent("datasets");
if (datasetsPresent)
    System.out.println("datasets packet is present");
else
    System.out.println("datasets packet is not present");
```

render

Merges the imported form design with imported data and renders the resulting document. The `destination` configuration option specifies the rendered output format. You can set this value using the `setConfigValue` method. For information, see [“Configuration options” on page 18](#).

Syntax

```
public FileDataBuffer render(ReturnStatus renderStatus)
    throws NoTemplateException, InvalidTemplateException,
        InvalidXDCEException, RenderException
```

Parameters

renderStatus	The status of the render operation. For information, see “ReturnStatus” on page 17 .
--------------	--

Returns

A `FileDataBuffer` object that contains the rendered output.

Throws

`NoTemplateException` if no form design is imported.

`InvalidTemplateException` if the form design is invalid.

`InvalidXDCEXception` if no XML file is specified or the specified XDC file cannot be found. This file is named `acrobat7.xdc` and contains information such as the available fonts, and must be referenced to successfully render a PDF document. This file is placed on the Java 2 Enterprise Edition (J2EE) application server on which LiveCycle Forms is deployed. The file location is dependent on which J2EE application server and operating system you are using. For example, this file is placed in the following location when LiveCycle Forms is deployed on IBM® WebSphere running on Microsoft® Windows®.

```
C:\Program Files\WebSphere\AppServer\installedApps\adobe\server1\XMLFormService\bin
```

`RenderException` if an error occurs during the rendering process.

Note: For a complete discussion about rendering a form using the XML Form Module API, see the “Rendering Non-Interactive Forms” chapter in *Developing Custom Applications*.

setConfigValue

Assigns a configuration value to the Configuration DOM. The node for which the configuration value is set is specified by a SOM expression. For information, see [“Configuration options” on page 18](#).

Syntax

```
public void setConfigValue(SOMExpression path, String value)
    throws InvalidSOMExpressionException, InvalidConfigurationException
```

Parameters

<code>path</code>	The SOM expression that specifies a node for which a configuration value is set.
<code>value</code>	The configuration value to assign to the node.

Throws

`InvalidSOMExpressionException` if the node that is specified by the SOM expression does not exist.

`InvalidConfigurationException` if an invalid value is specified.

ReturnStatus

`ReturnStatus` is an enumeration specifying whether the `render` method was successful.

<code>XFA_RENDER_SUCCESS</code>	Successful.
<code>XFA_RENDER_FAILURE</code>	Unsuccessful.
<code>XFA_RENDER_SUCCESS_WITH_INFO</code>	Successful, with additional messages that can be viewed in the server log file.
<code>XFA_RENDER_SUCCESS_WITH_WARNINGS</code>	Successful but produced warnings, which can be viewed in the server log file.

Note: For information about the `render` method, see [“render” on page 15](#).

XDP packets

Using Adobe LiveCycle Designer, a form author can save a form design as an XDP file or a PDF file. XDP is an XML format that provides a mechanism for packaging units of content (known as XDP packets) within a surrounding XML container.

When the form author saves the form design, the resulting XDP or PDF file typically contains XDP packets that define the form design and related configuration information. The presence of XDP information in a PDF file enables users of Adobe Reader® 6.0.2 or later to fill in forms created in LiveCycle Designer. XDP files can also contain data packets; for example, when a user fills in a form and submits the data to LiveCycle Forms.

Several XML Form Module API methods accept XDP packet names as parameter values. The following table provides a summary of each XDP packet.

Packet name	Description
<code>template</code>	Encloses the XML form design created in LiveCycle Designer.
<code>datasets</code>	Encloses XML form data that originates from an XML form and/or may be intended for merging with an XML form.
<code>stylesheets</code>	Encloses a single Extensible StyleSheet Language Transformations (XSLT). The XDP file may enclose more than one <code>stylesheets</code> packet.
<code>xfdf</code>	Encloses collaboration annotations in a PDF document.
<code>pdf</code>	Encloses an encoded PDF form without the XML form data in a <code>datasets</code> packet.
<code>config</code>	Encloses configuration information that the XML Form Module uses to determine how to process the input file.

Example

The following XDP example contains two packets: `datasets` and `pdf`. The first packet is the `<xfa:datasets>` element, which encloses the XML form data subassembly of a PDF form. The second packet is the `<pdf>` element, which encloses an encoded PDF form without the XML form data contained in the first `datasets` packet:

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
  <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
    <xfa:data>
      <book>
        <ISBN>15536455</ISBN>
        <title>Introduction to XML</title>
        <author><firstname>Charles</firstname>
          <lastname>Porter</lastname>
        </author>
      </book>
    </xfa:data>
  </xfa:datasets>
  <pdf xmlns="http://ns.adobe.com/xdp/pdf/">
    <document>
      <chunk>
        JVBERi0xLjMKJeTjz9IKNSAwIG9iago8PC9MZW5...
        ZQo+PgpzdHJlYW0KeJylWetv3DYQvutX8FKgPZj...
        Z/iUBGstoTDg9cfVfPPgcPjJDxUnDH7wt3GCtPv...
      </chunk>
    </document>
  </pdf>
</xdp:xdp>
```

Configuration options

The XML Forms Architecture leverages XML for the representation of all information and incorporates XML architectural concepts such as DOMs. One such DOM is the Configuration DOM.

The Configuration DOM provides a mechanism for specifying configuration options for the XML Form Module. The XML Form Module provides a default configuration file (`default.xci`). This file is placed on the J2EE application server on which LiveCycle Forms is deployed. The file location is dependent on which J2EE application server and operating system you are using. For example, this file is placed in the following location when LiveCycle Forms is deployed on WebSphere running on Windows:

```
C:\Program Files\WebSphere\AppServer\installedApps\adobe\server1\XMLFormService
```

Defaults

Conceptually, the Configuration DOM exists before loading the XML Form Module default configuration file. The Configuration DOM initializes all option values to their default values at startup. Required options are initialized to their default values. Options that do not require keyword values are initially empty.

When the XML Form Module loads the Configuration DOM from a configuration file, if a particular element is not present in the configuration file, the associated option retains its pre-existing value.

The default behavior is described within the individual section for each option in the [“Configuration options syntax” on page 20](#).

Scripting interface

The Configuration DOM also supports a scripting interface that enables user-supplied scripts to examine and modify the configuration settings. The XML Form Module API uses the XML Forms Architecture SOM to reference nodes within the Configuration DOM.

A SOM expression consists of a sequence of node names separated by periods ("" characters). Starting from some point in the Configuration DOM tree, each successive name identifies which child of the current node to descend to, such as the following example:

```
pdf.xdc.uri
```

This particular SOM expression references the `uri` element that is a descendant of the `pdf` element in the Configuration DOM.

Configuration options reference

The following table provides a cross-reference of elements in the Configuration DOM, their corresponding SOM expression, and a link to information on usage:

Configuration option (SOM expression)	Configuration DOM element	See
<code>data.outputXSL.uri</code>	<code>uri</code> (child of <code>outputXSL</code> element)	page 20
<code>data.range</code>	<code>range</code>	page 21
<code>data.record</code>	<code>record</code>	page 21
<code>data.startNode</code>	<code>startNode</code>	page 22
<code>data.xsl.debug.uri</code>	<code>uri</code> (child of <code>debug</code> element; descendant of <code>data</code> element)	page 22
<code>data.xsl.uri</code>	<code>uri</code> (child of <code>xsl</code> element; descendant of <code>data</code> element)	page 23
<code>destination</code>	<code>destination</code>	page 23
<code>locale</code>	<code>locale</code>	page 23
<code>pdf.compression.compressLogicalStructure</code>	<code>compressLogicalStructure</code>	page 23
<code>pdf.compression.level</code>	<code>level</code>	page 23
<code>pdf.compression.type</code>	<code>type</code>	page 24
<code>pdf.encryption.encrypt</code>	<code>encrypt</code>	page 24
<code>pdf.encryption.encryptionLevel</code>	<code>encryptionLevel</code>	page 24
<code>pdf.encryption.masterPassword</code>	<code>masterPassword</code>	page 24
<code>pdf.encryption.permissions.accessibleContent</code>	<code>accessibleContent</code>	page 24
<code>pdf.encryption.permissions.contentCopy</code>	<code>contentCopy</code>	page 25

Configuration option (SOM expression)	Configuration DOM element	See
pdf.encrypted.permissions.documentAssembly	documentAssembly	page 25
pdf.encrypted.permissions.formFieldFilling	formFieldFilling	page 25
pdf.encrypted.permissions.modifyAnnots	modifyAnnots	page 25
pdf.encrypted.permissions.plaintextMetadata	plaintextMetadata	page 26
pdf.encrypted.permissions.print	print	page 26
pdf.encrypted.permissions.printHighQuality	printHighQuality	page 26
pdf.encrypted.permissions.change	change	page 26
pdf.encrypted.userPassword	userPassword	page 27
pdf.fontInfo.embed	embed	page 27
pdf.fontInfo.encodingSupport	encodingSupport	page 27
pdf.fontInfo.map.equate	equate	page 27
pdf.fontInfo.subsetBelow	subsetBelow	page 28
pdf.interactive	interactive	page 28
pdf.openAction.destination	destination	page 28
pdf.submitFormat	submitFormat	page 29
pdf.tagged	tagged	page 29
pdf.xdc.uri	uri (descendant of pdf element)	page 29
temp.uri	uri (descendant of temp element)	page 29
template.base	base	page 29

Configuration options syntax

This section provides a reference of each configuration option that you can specify as a SOM expression using the `getConfigValue` and `setConfigValue` methods of the XML Form Module API.

data.outputXSL.uri

Specifies the fully qualified path or URL of an XSLT script. When specified, this option invokes an XSLT interpreter to process the supplied data before loading it into the XML DOM. For example:

```
data.outputXSL.uri=http://www.mysite.com/xsl/data/loan.xsl
```

data.range

Controls which records are processed; that is, loads only those records indicated into the XML DOM.

The `range` option specifies the records that are to be processed.

The value of `range` is a comma-separated list of one or more record numbers and/or record number ranges. A record number is a non-negative decimal integer, where 0 (zero) indicates the first record. A record number range is a record number followed by a dash (-) character, followed by another record number that is numerically equal to or greater than the other record number. Record number ranges and record numbers can overlap, as in the following example:

```
data.range=3-5,9,4,5-6
```

This causes records 3, 4, 5, 6, and 9 to be processed and all other records to be ignored.

For more information on records, see ["data.record" on page 21](#).

data.record

Controls the division of the document into records. Records are processed sequentially as separate documents. The value of `record` is either an integer or a tag name.

If the value of `record` is an integer, it specifies the level in the tree at which the XML Form Module treats each node as the root of a record. 0 represents the root of the whole XML Data DOM. For example, if the value is 2, each element that is two levels in from the outermost element is considered as enclosing a record.

If the value of `record` is not an integer, it is interpreted as a tag name. The first element in the XML data file with a tag matching the value of `record` determines the level of a record within the tree. The XML Form Module processes nodes as records in the XML DOM that correspond to the same level and have a name matching the value of `record` as root nodes.

By default, the XML Form Module considers the document range to enclose one record of data represented by the first (top-level) data group within the document range.

For example, the following XML data file contains a single record, which is the element `<order>`, and represents the top-level data group within the document range:

```
<order>
  <number>1</number>
  <shipto>
    <customer>c001</customer>
  </shipto>
  <item>
    <book>
      <ISBN>15536455</ISBN>
      <title>Introduction to XML</title>
      <quantity>1</quantity>
      <unitprice>55.00</unitprice>
    </book>
  </item>
  <item>
    <book>
      <ISBN>15536456</ISBN>
      <title>Advanced XML</title>
```

```
<quantity>1</quantity>
<unitprice>75.00</unitprice>
</book>
</item>
</order>
```

To nominate the `item` elements as records, specify `data.record=item`. The XML Form Module processes each record in the XML data file as a separate document.

To nominate only the second `item` element as a record, specify `data.record=2`. The XML Form Module processes only the second `item` element in the XML data file.

data.startNode

Identifies the root of the subtree that is processed. The XML Form Module does not process any data outside the subtree. The expression in the `startNode` element is restricted to a fully qualified path of element types (tag names), starting with the root of the XML data document and referring to a single element. This option affects processing during every phase.

For example, consider this XML data file:

```
<order>
  <number>1</number>
  <shipto>
    <customer>c001</customer>
  </shipto>
  <item>
    <book>
      <ISBN>15536455</ISBN>
      <title>Introduction to XML</title>
      <quantity>1</quantity>
      <unitprice>55.00</unitprice>
    </book>
  </item>
  <item>
    <book>
      <ISBN>15536456</ISBN>
      <title>Advanced XML</title>
      <quantity>1</quantity>
      <unitprice>75.00</unitprice>
    </book>
  </item>
</order>
```

To process only the first item, specify `data.startNode="xfasom(order.item)"`. To process only the second item, specify `data.startNode="xfasom(order.item[2].book)"`.

data.xsl.debug.uri

Specifies the fully qualified path or URL where the XML Form Module saves a copy of the preprocessed XML data after the XSLT interpreter has created it. It is intended for debugging the XSLT script. This option takes effect during the data loading phase, as shown in the following example:

```
data.xsl.debug.uri=http://www.mysite.com/xsl/debug/loan.xdp
```

data.xsl.uri

Specifies the fully qualified path or URL of an XSLT script. When specified, this option invokes an XSLT interpreter to process the supplied XML data before loading it into the XML DOM. This option takes effect during the data loading phase, as shown in the following example:

```
data.xsl.uri=http://www.mysite.com/xsl/loan.xsl
```

destination

Specifies the output format when rendering documents. The only acceptable value is `pdf`.

locale

Specifies the locale to use when rendering documents. For a list of supported locales, see the *Developer's Guide*.

pdf.compression.compressLogicalStructure

Used to generate PDF documents with the logical structure compressed. Normally, the structure is not compressed and therefore increases the size of the document. This feature was introduced in Adobe Acrobat® 6.0 Professional and Acrobat 6.0 Standard. The default value is off (0) for the XML Form Module, but it is defaulted to on (1) when a form author saves a PDF from LiveCycle Designer. If you use this option to compress the logical structure, that logical structure cannot be used by Acrobat 5.0 or earlier versions. The PDF will still open in Acrobat 5.0; however, to users, it will look as if the document has no logical structure (used for accessibility and tabbing order).

These values are acceptable:

- 0 - Do not compress the logical structure.
- 1 - Compress the logical structure.

pdf.compression.level

When `destination` is `pdf`, specifies the degree of compression to use when generating a file. These values are acceptable:

- 0 - Disable compression.

positive integer - Enable compression. The *positive integer* is a value between 1 and 9. The value 1 represents the best speed, and the value 9 represents the best compression. The default compression level is 6.

pdf.compression.type

When `destination` is `pdf`, specifies the type of compression to apply to a file. These values are acceptable:

- `none` - Does not compress the output. This value is the default.
- `ccittfax3` - CCITT T.4 bi-level encoding. Only works for monochrome TIF images.
- `ccittfax4` - CCITT T.6 bi-level encoding. Offers better compression than `ccittfax3` and is only for monochrome TIF images.
- `ccittrle` - A run length compression that is used only for monochrome TIF image files.
- `packbit` - A run length compression scheme that works for monochrome or color image files.

pdf.encryption.encrypt

Determines whether the output document is encrypted. These values are acceptable:

- `0` - Do not encrypt. This value is the default.
- `1` - Encrypt.

pdf.encryption.encryptionLevel

Specifies the length of the encryption key to use. The default and only acceptable value is `40bit`.

pdf.encryption.masterPassword

Specifies the password (as a string) required to open an encrypted PDF document that has unlimited access rights.

pdf.encryption.permissions.accessibleContent

Controls the ability of accessibility aids to copy text or graphics from the document. Accessibility aids such as screen readers need to copy text from the document into their own buffers. This options sets a permission flag that controls the ability of programs to extract text or graphics from the document. When extracted, the data may be used for any purpose.

This permission flag applies only when an encrypted PDF document is opened using the user password. No restrictions apply if the document is not encrypted or if it is opened using the master password.

These values are acceptable:

- `0` - Disable copying to accessibility aids. This value is the default.
- `1` - Enable copying to accessibility aids.

pdf.encrypted.permissions.contentCopy

Enables or disables the user's ability to copy text or graphics from the document.

This option applies only when an encrypted PDF document is opened using the user password. No restrictions apply if the document is not encrypted or if it is opened using the master password.

These values are acceptable:

- 0 - Disable copying. This value is the default.
- 1 - Enable copying.

pdf.encrypted.permissions.documentAssembly

Controls the user's ability to reassemble the PDF document. This option controls whether the user has the right to insert, delete, or rotate pages, and create navigation elements such as bookmarks and thumbnail images. If permission is granted, the user has these rights, even if the document is encrypted or if it is opened using the master password.

These values are acceptable:

- 0 - Disable insertion, deletion, or rotation of pages, and creation of navigation elements.
- 1 - Enable insertion, deletion, or rotation of pages, and creation of navigation elements.

pdf.encrypted.permissions.formFieldFilling

Controls the user's ability to enter data into existing form fields. This option sets a permission flag that controls the user's ability to fill in form fields, including signature fields. If permission is granted, the user can fill in fields regardless of the content of the `pdf.encrypted.permissions.modifyAnnots` option.

This permission flag applies only when an encrypted PDF document is opened using the user password. No restrictions apply if the document is not encrypted or if it is opened using the master password.

These values are acceptable:

- 0 - Disable filling in form fields. This value is the default.
- 1 - Enable filling in form fields.

pdf.encrypted.permissions.modifyAnnots

Controls the user's ability to modify the annotation layer of the document. This option sets a permission flag that grants the user the ability to add or modify text annotations and, if the `change` option grants permission, to create or modify interactive form fields (including signature fields). This option can also grant the user the ability to fill in existing fields of a form, but that permission can also be granted independently by the `pdf.encrypted.permissions.formFieldFilling` option.

These values are acceptable:

- 0 - Disable modification of the annotation layer. This value is the default.
- 1 - Enable modification of the annotation layer.

pdf.encryption.permissions.plaintextMetadata

Decrypts the metadata in the output PDF document. Document metadata is represented by an XML stream contained within the output PDF document. However, if the document is encrypted, the metadata is also encrypted by default. This option decrypts the metadata stream even if the rest of the document is encrypted.

These values are acceptable:

- 0 - Encrypt the metadata if the rest of the document is encrypted. This value is the default.
- 1 - Do not encrypt the metadata even if the rest of the document is encrypted.

pdf.encryption.permissions.print

Controls the user's ability to print the PDF document. The printed copy may be degraded in appearance compared to the original, depending on the content of the `pdf.encryption.permissions.printHighQuality` option.

This permission flag applies only when an encrypted PDF document is opened using the user password. No restrictions apply if the document is not encrypted or if it is opened using the master password.

These values are acceptable:

- 0 - Disable printing. This value is the default.
- 1 - Enable printing.

pdf.encryption.permissions.printHighQuality

Controls the user's ability to print the PDF document with high fidelity and as much detail as the original document.

This permission flag applies only when an encrypted PDF document is opened using the user password. No restrictions apply if the document is not encrypted or if it is opened using the master password.

These values are acceptable:

- 0 - Disable high-fidelity printing. This value is the default.
- 1 - Enable high-fidelity printing.

pdf.encryption.permissions.change

Controls the user's ability to make changes to the PDF document. This option controls the permission flag that grants the user permission to modify the contents of the document by any means not controlled by the `pdf.encryption.permissions.modifyAnnots` option, the `pdf.encryption.permissions.formFieldFilling` option, or the `pdf.encryption.permissions.documentAssembly` option. For example, this option controls the user's ability to edit the boilerplate.

These values are acceptable:

- 0 - Disable changes. This value is the default.
- 1 - Enable changes.

pdf.encrypted.userPassword

Specifies the password (as a string) required to open an encrypted PDF document that has user access rights.

pdf.fontInfo.embed

Controls the embedding of fonts in the output document. If the fonts are not embedded, the client computer or printer may not be able to reproduce the text properly. Particular fonts may have license restrictions that prohibit embedding.

These values are acceptable:

- 0 - Do not embed the fonts in the output document. This value is the default.
- 1 - Embed whatever fonts can be embedded in the output document.

pdf.fontInfo.encodingSupport

Specifies a list of non-Unicode character encodings. Defining the list in advance may improve performance. When not specified, encoding tables are initialized when first encountering a non-Unicode encoding.

The value must be a whitespace separated list of encoding names. The names, which are shown in the following list, are case-sensitive:

- Big-Five - Traditional Chinese
- GB2312 - Simplified Chinese
- ISO-8859-1 - European Latin 1
- ISO-8859-2 - European Latin 2
- ISO-8859-7 - Greek
- KSC-5601 - Korean
- Shift-JIS - Japanese

pdf.fontInfo.map.equate

Supplies mappings from one specified typeface to another during the rendering of a document. Use this option to deal with typefaces specified in the XDC file that are not available on the printer or display device, as in the following example:

```
<equate from="Arial_normal_normal", to="Arial_bold_italic">
```

The example changes the font for text that is specified as plain Arial to bold, italic Arial in the generated output.

pdf.fontInfo.subsetBelow

Specifies a usage threshold below which an embedded font is reduced to the subset of symbols that are actually used.

A font is sometimes used only for a few characters in a document. It is not necessary to embed the entire font if only a portion of it is used. This option sets a usage threshold below which only the used subset is embedded. Above the threshold, the entire font is embedded, if possible.

This option does not apply to fonts that are not embedded. Embedded fonts are controlled by the `pdf.fontInfo.embed` option.

This option has no effect for fonts that are used in data entry fields. If the font is embedded, it is embedded in its entirety.

The value is a positive integer from 0 to 100, inclusive. The default value for this option is 100, which subsets all embedded fonts that are not used in data entry fields.

pdf.interactive

Specifies whether a PDF form should be generated as a flat form for printing or as an interactive form for online use.

One way to use a PDF document is to print it. Another way is to use client software with a user interface that enables interaction with the document. The content of this option indicates which way the output document is used.

These values are acceptable:

- 0 - The form is printed for filling in off-line. Interactive user interface objects such as fields and radio buttons are rendered as boilerplate, and the annotation layer is empty. This value is the default.
- 1 - The form is filled in online using a suitable client program. Interactive user interface objects are placed into the annotation layer.

pdf.openAction.destination

Specifies the action to be performed when opening the document in an interactive client.

These values are acceptable:

- `none` - No special action is performed. The document is displayed using default behavior. This value is the default.
- `pageFit` - The document is resized to fit the window.

pdf.submitFormat

Specifies the format in which the form data is sent to the server. This option applies only to interactive PDF documents. The content of the `submitFormat` element determines the settings for bits 3 (ExportFormat, 9 (SubmitPDF), and 6 (XFDF) in the submit-form action as described in the PDF 1.5 specification.

These values are acceptable:

`html` - The data is submitted in HTML.

`delegate` - The format is determined by the server and client at run time.

`fdf` - The data is submitted in XFDF.

`xml` - The data is submitted in XML.

`pdf` - The data is submitted in PDF.

pdf.tagged

Controls whether tags are included in the output PDF document. Tags, in the context of PDF, are additional information included in a document to expose the logical structure of the document. Tags assist accessibility aids and reformatting. For example, a page number may be tagged as an "artifact" so that a screen reader does not enunciate it in the middle of the text. Although tags make a document more useful, they also increase the size of the document and the processing time to create it.

These values are acceptable:

0 - Do not insert tags. This value is the default.

1 - Insert tags.

pdf.xdc.uri

Specifies the path of the XDC file, `acrobat7.xdc`, which is placed on the J2EE application server on which LiveCycle Forms is deployed. The file location is dependent on which J2EE application server and operating system you are using. The XDC file contains information such as the available fonts. The XML Form Module uses the device control information to render PDF documents, as in the following example:

```
form.setConfigValue("pdf.xdc.uri", "C:\\Program  
Files\\WebSphere\\AppServer\\installedApps\\adobe\\server1\\XMLFormService  
\\bin\\acrobat7.xdc")
```

temp.uri

Specifies the location of temporary files. When specified, overrides the system default location for temporary files.

template.base

Specifies the base location for URIs in the form design. When this option is not specified, the location of the form design is used as the base.

The value is a URI that specifies the fully qualified path pointing to the location of any files to include. This option can be set by a separate document (or even a script) external to the form design.

2

Data Manager Module API

The Data Manager Module API enables you to efficiently exchange `DataBuffer` tokens that represent the underlying data and is a mechanism to abstract the actual storage of data from the usage of the data. This API also contains a set of interfaces that represent the underlying data.

The XML Form Module can create, exchange, and manipulate references to `DataManager` objects. For example, you must create a `DataManager` object when you create or work with a `Form` object (this object belongs to the XML Form Module API). For information about creating a `Form` object, see the “Invoking LiveCycle Forms” chapter in *Developing Custom Applications*.

DataManager interface

The `DataManager` interface connects the Data Manager service to a transaction model on the J2EE application server hosting LiveCycle Forms. The `DataManager` service provides a factory for creating `DataBuffer` (and `FileDataBuffer`) objects. One `DataManager` servant is allocated (on request) per transaction.

Requests by a module for a `DataManager` resource results in the module getting the `DataManager` servant created in response to the first request. On instantiation, the `DataManager` servant creates a temporary directory that is specific to the current transaction. When the servant terminates, it deletes the temporary directory and all of its contents.

createFileDataBuffer

Creates a `FileDataBuffer` object and wraps it around the file specified as the parameter. If the file is outside the transaction-specific temporary directory, it is not automatically deleted unless the `DataManager` object is specifically requested to manage the file as part of the transaction by using the `manageTempFile` method. For information, see [“manageTempFile” on page 32](#).

Syntax

```
public FileDataBuffer createFileDataBuffer(String filePath)
    throws InvalidSourceException
```

Parameters

<code>filePath</code>	The fully qualified path to the file that wraps the <code>FileDataBuffer</code> object.
-----------------------	---

Returns

A `FileDataBuffer` object wrapped around the file specified as the `filePath` parameter.

Throws

`InvalidSourceException` if an invalid file is specified.

createFileDataBufferFromUrl

Creates a `FileDataBuffer` object from a URL that is passed as a string parameter. The `DataManager` object first creates a temporary file, copies the contents of the URL over to the file (including the content type), and creates a regular `FileDataBuffer` object wrapping the temporary file.

Syntax

```
public FileDataBuffer createFileDataBufferFromUrl (String url)
    throws InvalidSourceException
```

Parameters

<code>url</code>	The URL used to create a <code>FileDataBuffer</code> object.
------------------	--

Returns

A `FileDataBuffer` object created from a URL.

Throws

`InvalidSourceException` if an invalid URL is specified.

getTempFileName

Generates a temporary file name in the transaction-specific temporary directory and returns the full path to the file. If the `create` parameter is `true`, the file is created.

Syntax

```
public String getTempFileName (boolean create)
```

Parameters

<code>create</code>	Set this parameter to <code>true</code> to create the file.
---------------------	---

Returns

The full path to the temporary file name.

Details

If `create` is set to `false`, the file must be programmatically created. It is recommended that you set `create` to `true`.

manageTempFile

Requests that the `DataManager` object include the specified file as part of the transaction, even if the file is outside the transaction-specific temporary directory. The `DataManager` object maintains a list of managed files and deletes them at the end of the transaction.

Syntax

```
public void manageTempFile(String filePath)
    throws InvalidSourceException
```

Parameters

<code>filePath</code>	The file that is part of the transaction.
-----------------------	---

Throws

`InvalidSourceException` if an invalid `filePath` value is specified.

DataBuffer interface

The `DataBuffer` interface is the base class for the `FileDataBuffer` interface and provides a handle to the underlying data that may be exchanged between cooperating applications. The underlying data can originate from a file, a network socket, shared memory, or a variety of other sources.

getBufLength

Returns the size of the `DataBuffer` object in bytes.

Syntax

```
public long getBufLength()
```

Returns

The size of the `DataBuffer` object in bytes.

getBytes

Returns `nBytes` of data starting from position `pos` of the `DataBuffer`. If you specify an invalid value for the `pos` parameter, a run-time error occurs. Run-time exceptions are also thrown for out of memory conditions, as well as insufficient permission to read the underlying file (when using `FileDataBuffers` objects).

Syntax

```
public byte[] getBytes(long pos, long nBytes)
```

Parameters

<code>pos</code>	The starting position from which to return data. This value is zero-based.
<code>nBytes</code>	The number of bytes of data to return.

Returns

`nBytes` of data starting from position `pos`.

getContentType

Returns the content type (MIME-type) of the underlying data, if available.

Syntax

```
public String getContentType()
```

Returns

A string that identifies the content type (MIME-type). If the content type is not available, a string containing the value `unknown` is returned. This method returns the content type of the data that can be set using the `setContentType` method.

setContentType

Sets the content type for the underlying data in the `DataBuffer` object.

Syntax

```
public void setContentType(String contentType)  
    throws org.omg.CORBA.BAD_INV_ORDER
```

Parameters

<code>contentType</code>	The content type to set: <ul style="list-style-type: none">• <code>url-encoded</code>• <code>text/xml</code>• <code>application/xml</code>• <code>application/vnd.adobe.xdp</code>
--------------------------	---

Throws

This method is used once for a given `DataBuffer` object. After the content type is set, setting it again throws an `org.omg.CORBA.BAD_INV_ORDER` exception.

FileDataBuffer interface

The `FileDataBuffer` interface extends the `DataBuffer` interface. It provides a `DataBuffer` that is saved to a file located on a disk. The operations provided by this interface are primarily file related.

The actual implementation of the `FileDataBuffer` interface does not keep track of information about the saved file other than it exists and is readable. Furthermore, the implementation does not provide access to the actual data contained in the file.

getFilePath

Returns the full path to the file on which a `FileDataBuffer` object is based.

Syntax

```
public String getFilePath()
```

Returns

The full path to the file on which a `FileDataBuffer` object is based.

DMUtils class

The `DMUtils` class enables you to perform programmatical tasks involving data sets such as reading a data input stream into a temporary file and placing the file into a `DataBuffer` object. All `DMUtils` methods are static. For information about creating a `DMUtils` object, see the “Invoking LiveCycle Forms” chapter in *Developing Custom Applications*.

getDataBuffer

This method consists of two signatures. The first signature takes an `InputStream` and reads it into a temporary file. This method then creates a `DataBuffer` object and wraps it around the temporary file.

The second signature takes a `DataHandler` object, uses the associated `InputStream` to get the data into a temporary file, and creates a `DataBuffer` for the temporary file. This method also sets the content type value based on the `DataHandler` content type.

Syntax

```
public static DataBuffer getDataBuffer(DataManager dm, InputStream is)
public static DataBuffer getDataBuffer(DataManager dm, DataHandler dh)
```

Parameters

<code>dm</code>	A reference to a <code>DataManager</code> object.
<code>is</code>	The <code>InputStream</code> that is read into a temporary file.
<code>dh</code>	The <code>DataHandler</code> object that uses the associated <code>InputStream</code> to get data into a temporary file.

Returns

A `DataBuffer` object that wraps a temporary file.

getDataHandler

Takes a `DataBuffer` object and creates a `DataHandler` object for the data associated with the `DataBuffer`. If the `DataBuffer` object is a `FileDataBuffer` object, a `DataHandler` object is created that wraps the underlying file.

Syntax

```
public static DataHandler getDataHandler(DataManager dm, DataBuffer db)
```

Parameters

<code>dm</code>	A reference to a <code>DataManager</code> object.
<code>db</code>	The required <code>DataBuffer</code> object.

Returns

A `DataHandler` object created by this method.

getInputStream

Takes a `DataBuffer` object and creates an `InputStream` object associated with the data. If the underlying `DataBuffer` object is a `FileDataBuffer`, this method creates a `FileInputStream` object associated with the underlying file.

Syntax

```
public static InputStream getInputStream(DataManager dm, DataBuffer db)
```

Parameters

<code>dm</code>	A reference to a <code>DataManager</code> object.
<code>db</code>	The required <code>DataBuffer</code> object.

Returns

An `InputStream` associated with the data.

The Connection API enables client applications to use the Data Manager Module API. This API is functionality similar to a JDBC data source, a JCA connection factory, or an EJB home interface in that it is a registered factory for retrieving transaction-associated resources.

The Connection API consists of one interface named `ConnectionFactory`, which consists of a method named `getConnection`. To create connections to the modules, you must perform a JNDI look-up and then the `getConnection()` method. Although this chapter describes how to use this API to create a connection to the Data Manager Module, it does not contain all required information, such as which import statements to include and which JAR files to add to your project's build path. For complete information about creating connections to modules, including which Java import statements to include, see the "Invoking LiveCycle Forms" chapter in *Developing Custom Applications*.

ConnectionFactory interface

The `ConnectionFactory` interface lets you create a connection to a module. You cannot instantiate a `ConnectionFactory` object using a constructor. Instead, you create a `ConnectionFactory` object by performing a JNDI look-up using Java classes, such as the `Context` class and the `PortableRemoteObject` class.

After you create a `Context` object, perform a JNDI look-up by calling its `lookup` method and specify a module. Pass one of the following values to the `lookup` method:

- `DataManagerService` - To perform a look-up on the Data Manager Module
- `PDFManipulation` - To perform a look-up on the Invoking LiveCycle Forms
- `XMLFormService` - To perform a look-up on the Invoking LiveCycle Forms

After you call the `lookup` method, call the `PortableRemoteObject` object's `narrow` method to ensure that the object returned from the `lookup` method can be cast to a `ConnectionFactory` object. The following code example shows how to create a `ConnectionFactory` object while performing a look-up on the Invoking LiveCycle Forms service:

```
//Create a Context object
Context namingContext = new InitialContext();

// Lookup the Data Manager service
Object dmObject = namingContext.lookup("DataManagerService");
ConnectionFactory dmConnectionFactory = (ConnectionFactory)
    PortableRemoteObject.narrow(dmObject, ConnectionFactory.class);
```

getConnection

Opens a connection to the specified module.

Syntax

```
public Object getConnection()
    throws RemoteException
```

Details

This method returns a CORBA object representing a connection to the module. You must cast the return value to `org.omg.CORBA.Object`.

This method must be called from a valid JTA transaction. It returns a CORBA object representing the module that is allocated to the current transaction. Connections are allocated per transaction; multiple calls to this method from the same transaction returns the same object. Connections are automatically destroyed when the transaction is complete.

Returns

A connection to a module.

Throws

`RemoteExpression` if you attempt to look up an invalid module. For example, if you pass an invalid value to the `Context` object's `lookup` method, a `RemoteExpression` is thrown.

Example

The following example shows how to create a `DataManager` object using the `ConnectionFactory` interface's `getConnection` method:

```
//Declare a ConnectionFactory object
ConnectionFactory dmConnectionFactory = null;

// Lookup the Data Manager service
Object dmObject = namingContext.lookup("DataManagerService");
dmConnectionFactory = (ConnectionFactory)
    PortableRemoteObject.narrow(dmObject, ConnectionFactory.class);

//Start the transaction
transaction.begin();

//Get a DataManager object
DataManager mDataManager =
    DataManagerHelper.narrow((org.omg.CORBA.Object)dmConnectionFactory.
    getConnection());

//Perform tasks using the DataManager object
//Commit the transaction
transaction.commit();
```

Note: For a complete explanation of this code example, including the `UserTransaction.begin` method (transaction is `UserTransaction` object), see the “Invoking LiveCycle Forms” chapter in the *Developer's Guide*.

Index

C

clearMessages method 9
ConnectionFactory interface 36
create method 7
createDefault method 8
createFileDataBuffer method 30
createFileDataBufferFromUrl method 31

D

Data Manager Module API methods
 createFileDataBuffer 30
 createFileDataBufferFromUrl 31
 getBufLength 32
 getBytes 32
 getContentType 33
 getDataBuffer 34
 getDataHandler 35
 getFilePath 34
 getInputStream 35
 getTempFileName 31
 manageTempFile 32
 setContentType 33
DataBuffer interface 32
DataManager interface 30
DMUtils class 34

E

exportXDP method 9, 11

F

FileDataBuffer interface 34
Form interface 9
FormFactory interface 7

G

getBufLength method 32
getBytes method 32
getConfigValue method 12
getConnection method 36
getContentType method 33
getDataBuffer method 34
getDataHandler method 35
getFilePath method 34

getInputStream method 35
getMessages method 9
getPacketList method 13
getPageCount method 13
getTempFileName method 31

I

importPackets method 14
interfaces
 ConnectionFactory 36
 DataBuffer 32
 DataManager 30
 FileDataBuffer 34
 Form 9
 FormFactory 7
isPacketPresent method 14

M

manageTempFile method 32

R

render method 15
ReturnStatus enumeration 17

S

setConfigValue method 16
setContentType method 33

X

XML Form Module API methods
 clearMessages 9
 create 7
 createDefault 8
 exportXDP 9, 11
 getConfigValue 12
 getMessages 9
 getPacketList 13
 getPageCount 13
 importPackets 14
 isPacketPresent 14
 render 15
 setConfigValue 16



Adobe

Developing Custom Applications

July 2006

Adobe® LiveCycle™ Forms

Version 7.2

© 2006 Adobe Systems Incorporated. All rights reserved.

Adobe® LiveCycle™ Forms 7.2 Developing Custom Applications for Microsoft® Windows®, UNIX®, and Linux
Edition 3.0, July 2006

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names and company logos in sample material or in the sample forms included in this software are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, LiveCycle, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group.

All other trademarks are the property of their respective owners.

This product includes code licensed from RSA Security, Inc.

Some portions licensed from IBM are available at <http://oss.software.ibm.com/icu4j/>

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

List of Examples	7
Preface	8
1 Introduction	10
LiveCycle Forms APIs.....	11
Form Server Module API.....	11
XML Form Module API.....	12
Data Manager Module API.....	13
About form types	13
Interactive forms	13
Non-interactive forms	13
Dynamic forms.....	13
Server-side dynamic forms.....	14
Client-side dynamic forms.....	14
Static forms.....	14
Rendering different form types	15
Planning a LiveCycle Forms client application	16
Creating form designs for LiveCycle Forms.....	17
Designing form designs to render as HTML.....	18
HTML pages	18
Running scripts.....	18
Event timing	19
LiveCycle Designer buttons	20
HTML 4.0 web browser	20
Maintaining presentation changes	20
Caching forms.....	21
LiveCycle Forms processing requests.....	21
Requesting a form	21
Using Form Design buttons.....	23
Submit button	23
Calculate button	26
2 Invoking LiveCycle Forms	28
Including LiveCycle Forms library files	28
Invoking the Form Server Module	30
Locally invoking Form Server Module.....	31
Remotely invoking Form Server Module.....	32
Invoking Form Server Module using SOAP.....	34
Invoking Form Server Module using the Microsoft .NET client assembly	36
Creating Form Server Module objects using the FormServerFactory class.....	38
Creating a SOAPClient object using the FormServerFactory class	38
Creating an EJBClient object to locally invoke Form Server Module.....	39
Creating an EJBClient object to remotely invoke Form Server Module.....	39

2 Invoking LiveCycle Forms (Continued)

Invoking the Data Manager Module	40
Creating a DataManager object.....	41
Using the DMUtils object	42
Working with the Document object	43
Creating a Document object using a remote file.....	43
Creating a Document object using a data stream.....	44
Creating a Document object using a local file	44
Returning the content of a Document object to a file	45
Writing the content of a Document object to a data stream	45
Invoking the XML Form service.....	45
Creating a Form object	45

3 Rendering Interactive Forms as PDF 48

About rendering PDF forms	48
Rendering a form using an EJBClient object	52
Specifying the form design to render.....	53
Passing a zero-length byte array	53
Setting preference options to render the form as PDF	53
Specifying the web context of a client application	53
Specifying the target URL	54
Specifying the PDF version.....	54
Caching PDF forms	54
Caching PDF forms in the client web browser	55
Accessing LiveCycle Form Manager application store	55
Setting the Standalone option.....	55
Setting XCI run-time options	56
Creating application logic to render a form as PDF	57
Rendering a Form using a SOAPClient object	58
Retrieving submitted form data.....	60
Form design considerations.....	60
Relationship between form fields and XML data	61
Creating application logic to retrieve submitted data	62
Saving submitted data as XML.....	65
Converting the content type of form data	66
Rendering prepopulated forms	68
Creating application logic to render a prepopulated form	68
Converting an XML document to a byte stream	69
Converting an XML string to a byte stream	71
Prepopulating a form using a Document object.....	72
Rendering a form at the client.....	74
Passing a form design by value.....	77

4 Rendering Dynamic Forms..... 79

About dynamic forms	79
Form design considerations.....	80
XML data source.....	80
Rendering prepopulated dynamic forms	82
Creating an in-memory XML data source	83
Converting the XML data source to a byte array	87
Rendering a prepopulated dynamic form	89

5	Rendering Forms as HTML	91
	Client applications rendering HTML forms	91
	Form considerations	92
	Rendering a form as HTML	92
	Setting preference options to render the form as HTML	92
	Specifying the client applications web context	93
	Caching HTML forms	93
	Creating application logic to render a form as HTML	93
6	Calculating Form Data	95
	About form design scripts	95
	Handling a form containing a script	96
	Rendering a form that contains a script	97
	Creating application logic to handle a form containing a calculation script	98
7	Working with PDF Form Fields	100
	Importing form data	100
	Exporting form data	101
	Flattening form fields	102
8	Transferring PDF Data	103
	About transferring data	103
	Form design considerations	104
	Retrieving submitted PDF data	105
	Creating a PDF document	106
	Saving a PDF document	108
9	Authenticating Users	110
	About user authentication	110
	Performing user authentication	111
	Programmatically authenticating a user	111
	Setting the LiveCycle Forms invocation context	112
	Creating application logic to authenticate users	112
10	Rendering Forms from .NET	114
	Client applications rendering PDF forms	114
	Rendering a form using the Microsoft .NET client assembly	115
	Retrieving submitted data	117
	Rendering prepopulated forms	119
	Creating application logic to render a prepopulated form	119
11	Rendering Forms using the XML Form Module API	121
	Creating a Form object	121
	Importing packets	124
	Importing a template packet	124
	Importing a datasets packet	124
	Determining if a packet exists	125
	Setting configuration values	125
	Setting the destination configuration value	125
	Setting the pdf.xdc.uri configuration value	125
	Determining a configuration value	126
	Rendering PDF documents	126

A	Character Sets and Unicode Encodings	129
B	Language and Locale Combinations.....	130
	Glossary	133
	Index	137

List of Examples

Example 2.1	Remotely invoking Form Server Module using the EJBClient object	34
Example 2.2	Remotely invoking Form Server Module using the SOAPClient class	36
Example 2.3	Creating a SOAPClient object using the FormServerFactory class	38
Example 2.4	Creating a local EJBClient object using the FormServerFactory class	39
Example 2.5	Creating a remote EJBClient object using the FormServerFactory class.....	40
Example 2.6	Creating a DataManager object	42
Example 2.7	Creating a Document object using a remote file	43
Example 2.8	Creating a Document object using a data stream	44
Example 2.9	Creating a Document object using a local file	44
Example 2.10	Returning the content of a Document object to a file	45
Example 2.11	Writing the content of a Document object to a data stream	45
Example 2.12	Creating a Form object using createDefault.....	47
Example 3.1	Rendering a form to a client web browser using an EJBClient object.....	57
Example 3.2	Rendering a form to a client web browser using a SOAPClient object.....	58
Example 3.3	Retrieving submitted form data	64
Example 3.4	Saving submitted data as XML	66
Example 3.5	Prepopulating a form by converting an XML document to a byte stream	70
Example 3.6	Prepopulating a form by converting a string variable to a byte stream	71
Example 3.7	Prepopulating a form using a Document object	72
Example 3.8	Rendering a form at the client	75
Example 4.1	Creating an in-memory XML data source	84
Example 4.2	Converting an in-memory XML data source to a byte array	88
Example 4.3	Rendering a prepopulated dynamic form.....	89
Example 5.1	Rendering a form as HTML to a client web browser	94
Example 6.1	Handling a form containing a calculation script	98
Example 8.1	Retrieving submitted PDF data	105
Example 8.2	Creating a PDFDocument object using a temporary file containing PDF data	107
Example 8.3	Saving a PDF document	109
Example 9.1	Authenticating a user with User Manager.....	113
Example 10.1	Rendering a form to a client web browser	116
Example 10.2	Retrieving data from a form.....	118
Example 11.1	Creating a Form object by using the FormFactory object's create method	123
Example 11.2	Rendering a PDF document by using the Form object's render method.....	127

Preface

This guide provides information about Adobe® LiveCycle™ Forms, one of the many products provided by Adobe document services.

What's in this guide?

This guide describes the development environment, the architecture, and the required activities from planning to deployment. This guide explains how to use the APIs to develop client applications. It is a companion guide to the *Form Server Module API Reference* and the *XML Form Module API Reference*.

Who should read this guide?

This guide is intended for developers who are responsible for developing client applications for LiveCycle Forms.

Related documentation

In addition to this guide, the resources in the table provide information about LiveCycle Forms.

For information about	See
Understanding what LiveCycle Forms is and how it integrates with other Adobe products	<i>Overview</i>
Installing, configuring, and administering LiveCycle Forms in a development and run-time environment	<i>Installing and Configuring LiveCycle for JBoss</i> <i>Installing and Configuring LiveCycle for WebSphere</i> <i>Installing and Configuring LiveCycle for WebLogic</i>
The Form Server Module API, including a description and explanation of its classes and methods	<i>Form Server Module API Reference</i> Note: <i>This is in HTML format.</i>
The XML Form Module API, including a description and explanation of its classes and methods	<i>XML Form Module API Reference</i>
The Adobe User Management SPI, including a description and explanation of its classes and methods	<i>Adobe User Management SPI Reference</i>
Using the User Management SPI to develop custom service providers	<i>Developing User Management Service Providers</i>
The new features in this product release	<i>What's New</i>
The form objects and associated properties that are supported in each web browser.	<i>Transformation Reference</i>

For information about	See
Other services and products that integrate with LiveCycle Forms	www.adobe.com
Patch updates, technical notes, and additional information on this product version.	www.adobe.com/support/products/enterprise/index.html

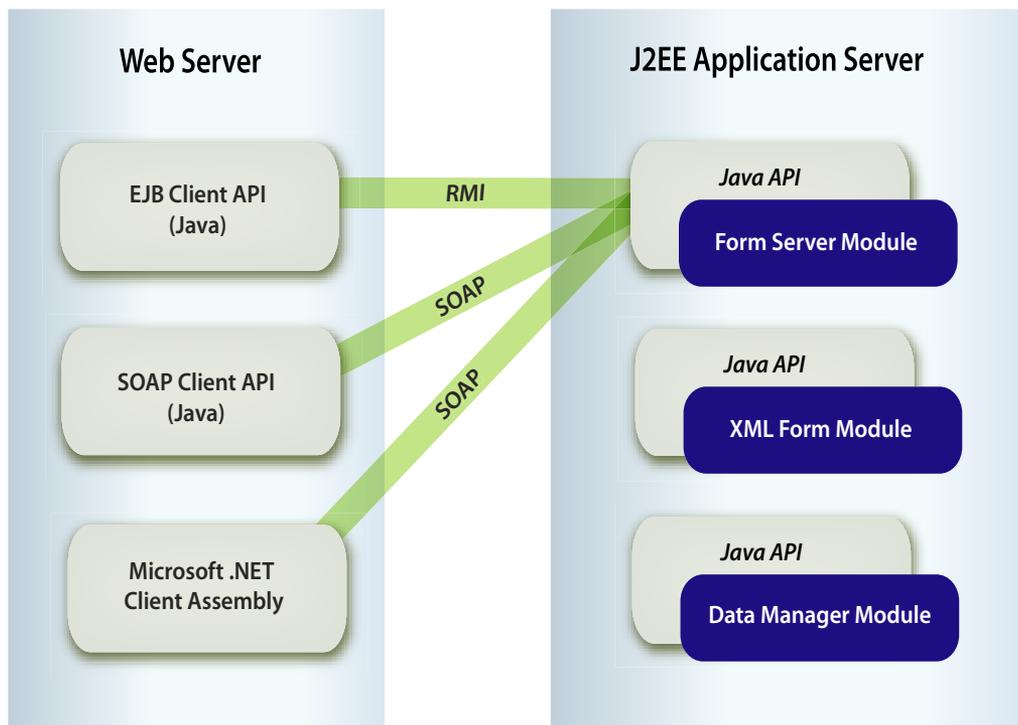
1

Introduction

This chapter provides an overview of Adobe LiveCycle Forms architecture, a description of LiveCycle Forms APIs, a summary of each module, and a planning section.

LiveCycle Forms APIs are used to create applications that access the services of specific modules that make up LiveCycle Forms. All LiveCycle Forms modules are stateless and are only accessible through the corresponding APIs. For example, you can invoke the Form Server Module using the Form Server Module API. The following diagram shows each LiveCycle Forms API accessing its corresponding service.

LiveCycle Forms Architecture



LiveCycle Forms APIs provide public Java interfaces for different modules. Each module runs as a Java 2 Enterprise Edition (J2EE) service on your J2EE application server. You use a Java development environment to create applications, such as Java servlets, that interact with specific modules. For example, you can create a Java servlet that invokes the Form Server Module in response to an end user clicking a link that is displayed within a web browser.

The Form Server Module API contains a managed client assembly for Microsoft® .NET. Using this client assembly, you can create web service client applications in the Microsoft Visual Studio .NET 2003 development environment that interact with the Form Server Module. For information, see [“Rendering Forms from .NET” on page 114](#).

LiveCycle Forms integrates SOAP. Using both the Java-based and .NET-based client libraries, you can create applications that access the LiveCycle Forms remotely from any platform.

Note: This guide does not discuss how to install and deploy LiveCycle Forms to a J2EE application server. For information, see the *Installing and Configuring* guide for your application server.

LiveCycle Forms APIs

LiveCycle Forms consists of modules that perform specific operations. All modules, with the exception of the Font Manager Module, provide a public Java API. The following table identifies each module that has a public API and provides a brief description of when you would use the API in your client applications.

Product module	When to use each API	See summary on
Form Server Module	Use this API to create interactive data capture applications that render XML forms displayed as either HTML or PDF to a client device, typically web browsers, and process form submissions. Processing a form submission means handling data that a user enters into a form using a client device, such as a web browser.	page 11
XML Form Module	Use this API to create non-interactive applications for rendering forms. For example, you can use the XML Form Module API to produce a large, multi-document, non-interactive output stream. Note: The XML Form Module API is deprecated.	page 12
Data Manager Module	Use this API to invoke the XML Form Module. For information, see “Invoking the XML Form service” on page 45. An important component of this API is the <code>Document</code> object, which is used by the Form Server Module API. For information, see “Working with the Document object” on page 43. This API is also used to transfer data from one LiveCycle product to another. For information, see “Transferring PDF Data” on page 103.	page 13

For information on how to invoke each module, see [“Invoking LiveCycle Forms” on page 28.](#)

Form Server Module API

You use the Form Server Module API to create interactive data capture applications. The Form Server Module validates, processes, transforms, and delivers forms typically created in Adobe LiveCycle Designer. Form authors can develop a single form design that the Form Server Module can render in PDF or HTML format in a variety of browser environments.

When an end user requests a form, a client application, such as a Java servlet, sends the request to the Form Server Module, which returns the form in an appropriate format to the end user. When the Form Server Module receives a request for a form, it uses a set of transformations to merge data with a form design and then delivers the form in a format that best matches the presentation and form filling capabilities of the target browser.

The Form Server Module performs the following functions:

- Provides server-side execution of the intelligence that is in the form design. The Form Server Module executes the validations and calculations included in the form design and returns the resulting data to the browser. For information, see [“Calculating Form Data” on page 95](#).
- Detects whether form design scripts should run on the client or the server. For clients that support client-side scripting such as Internet Explorer 5.0 and later, an appropriate scripting model is loaded into the device so that the scripts can run directly on the client computer. For information about the properties and methods supported in each transformation, see the *LiveCycle Designer Help*.
- Dynamically generates a PDF or an HTML document of the form design with or without data. An HTML form can deliver multipage forms page by page. In contrast, a PDF form delivers all the pages at once. In LiveCycle Designer, the form author can script the current page number in the form design. The Form Server Module can merge one page of data submitted at a time or merge only the single page into the form design.
- Supports dynamic subforms created in LiveCycle Designer. Form Server Module adds extra fields and boilerplate as a result of merging the form design with data or as a result of scripting. In the case of HTML, the added subforms can grow to unlimited page lengths. In the case of PDF, the added subforms paginate at the page lengths specified in the form design.
- Validates data entry by performing calculations, accessing databases, or enforcing business rules on field-level data.
- Displays validation errors in different ways (split frame left, top, right, bottom; no frame left, top, right, bottom; or no UI). This is all done without maintaining any state on the server. The validation errors are also made available in the XML-based validation error document.
- Maintains the state of any pass-through data that has been passed in by the application. Pass-through data is data that does not have corresponding fields on the form design being processed. The pass-through data is passed back to the calling application after the target device submits the data.
- Enables a non-technical user to amend a form design by using LiveCycle Designer to meet ongoing business requirements. In contrast, a web application that displays HTML pages may require a user to modify HTML or XML source code to make changes to a web page.

XML Form Module API

You typically use the XML Form Module API to create applications that can process and work with non-interactive forms and large data sets. Using this API, you can create applications that perform non-interactive form rendering operations such as these:

- Loading XML data into an XML Data Package (XDP) file
- Loading XML data into a PDF file that contains XDP information
- Controlling configuration and data-loading options
- Rendering PDF documents
- Extracting XML data from an XDP file

For information on how to use the XML Form Module API, see [“Rendering Forms using the XML Form Module API” on page 121](#).

Data Manager Module API

You use the Data Manager Module API to invoke the XML Form Module and to transfer data as a stream of bytes. For information on how to use the Data Manager Module API, see [“Invoking the Data Manager Module” on page 40](#).

About form types

Before you start using the LiveCycle Forms API to create an application that interacts with LiveCycle Forms, you should have a solid understanding of the different form types that are used by LiveCycle Forms. This section describes these form types.

Interactive forms

An interactive form contains one or more fields for collecting information interactively from a user. An interactive form design produces a form that can be filled online or (in the case of PDF forms) offline. Users can open the form in Acrobat, Adobe Reader, or an HTML browser and enter information into the form’s fields. An interactive form can include buttons or commands for common tasks, such as saving data to a file or printing. It can also include drop-down lists, calculations, and validations.

Note: An interactive form can be dynamic or static.

Non-interactive forms

A non-interactive form does not respond to user interaction. This form type is often a PDF form that is downloaded by a user, printed, and filled manually. A non-interactive dynamic form can be prepopulated with data, and then made available to the users. For example, billing statements are an example of a non-interactive form.

Note: A non-interactive form can be dynamic or static.

Dynamic forms

A dynamic form has a dynamic layout that changes based on data prepopulation or through user interaction. A dynamic form may be interactive or non-interactive. A non-interactive dynamic form is prepopulated with data, then made available to a user without interactive features. An interactive dynamic form may or may not be prepopulated with data, but contains interactive fields or other interactive features that enables a user to interact with it.

A dynamic form design specifies a set of layout, presentation, and data capture rules, including the ability to calculate values based on user input. The rules are applied when a user enters data into the form or when a server merges data into a form. Dynamic forms are usually rendered by LiveCycle Forms or Acrobat 7.0 and Adobe Reader 7.0. Dynamic forms are particularly useful when displaying an undetermined amount of data to users. You do not need to predetermine a fixed layout or number of pages for the form, as is required by a static form. When rendered as a PDF form, intelligent page breaks are generated.

Two types of dynamic forms exist: server-side and client-side dynamic forms. Both server-side and client-side dynamic forms are based on form designs that are created in LiveCycle Designer.

Server-side dynamic forms

A server-side dynamic form can be a data-driven dynamic form; that is, the form is populated with data during rendering. The amount of data determines the form's layout. Multiple data value instances can be provided for a given field, causing the field to dynamically replicate so that each data value is displayed within the form.

Fields that are dynamically added to a dynamic form are contained in structures called subforms, which are located within the form design. An example of a server-side dynamic form is one that is part of a client application that queries a database and retrieves an unknown number of records. After retrieving records from a database, the application calls the LiveCycle Forms API to merge the data into the form. After the data is merged into the form, the application renders the form to a user.

Client-side dynamic forms

A client-side dynamic form is typically used to collect data from end users by enabling them to click a button (or another control) that produces a new field in which data is entered. The new field appears on the form immediately and does not require a round trip to the server. That is, the form is not sent to the J2EE application server hosting LiveCycle Forms and then rendered back to the client web browser with the new field. An example of a client-side dynamic form is one that contains fields that enable a user to enter items to purchase and a button that enables the user to add new fields. Each time the user clicks the button, a new subform is added to the form (a subform can contain a set of related fields).

Static forms

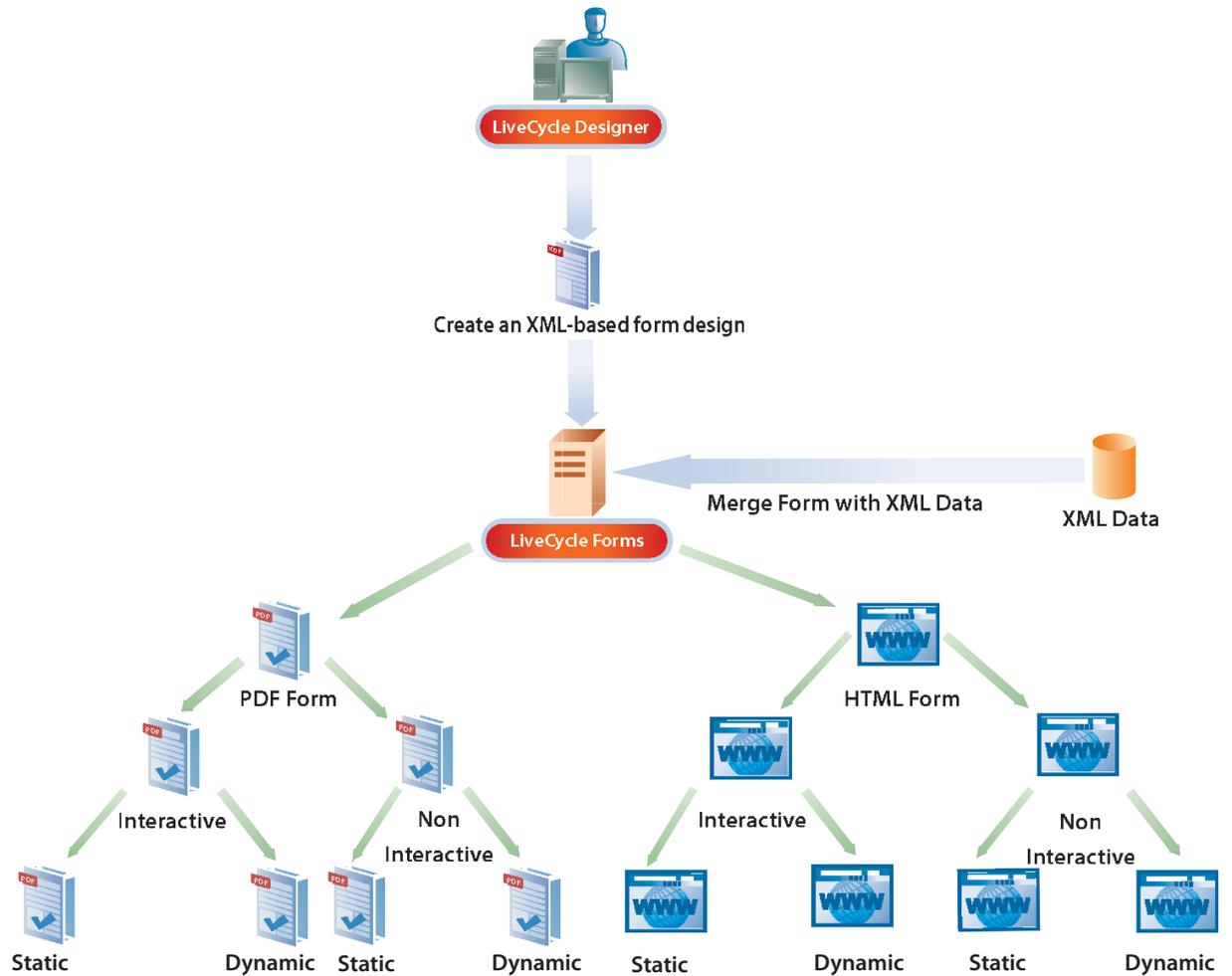
A static form has a fixed layout that does not change regardless of how much data is placed into the fields. A static form can be interactive, in which case a user fills the form, or non-interactive, in which case a server may prepopulate the form with data. Any fields left unfilled are present in the form but empty. Conversely, if there is more data than the form can hold, the form cannot expand to accommodate the excess data.

In the case of an interactive form, the end user cannot enter extra information beyond what the form fields can hold. Similarly, excess data merged by LiveCycle Forms overruns the area bounded by the object and the excess data is not displayed. As a result, when creating a static form, form authors position and size the objects in such a way that the objects can accommodate the largest expected set of data.

Static PDF forms are created by LiveCycle Designer when a form design is saved as a PDF file, or they can be rendered on the server by passing an XDP file to LiveCycle Forms. A static form can be generated from a dynamic form design, but once rendered as PDF, the background is locked. Static forms are easily cacheable on the server, so are quickly accessed when requested by a user.

Rendering different form types

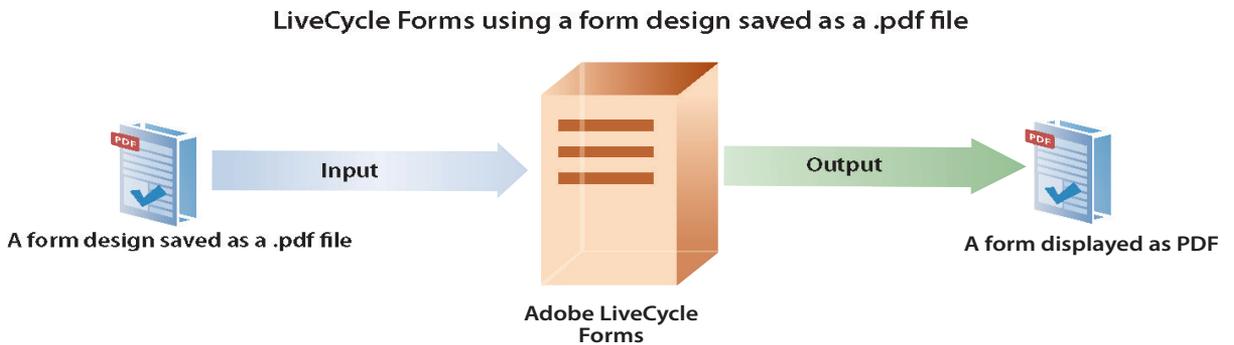
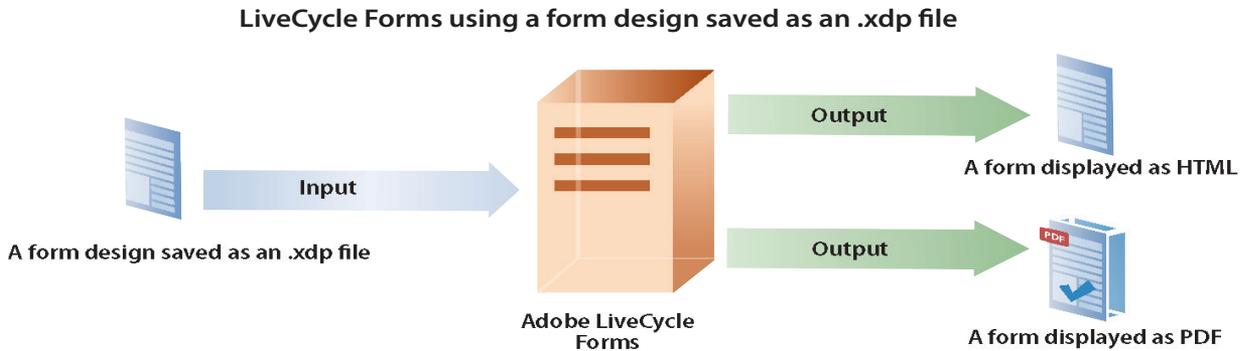
LiveCycle Forms is capable of rendering the form types that are described in this section as either PDF or HTML. For example, LiveCycle Forms can render an interactive form as PDF or a non-interactive form as HTML. The following diagram shows the different form types that LiveCycle Forms can render.



Planning a LiveCycle Forms client application

Before you use the Form Server Module API to create a client application that interacts with LiveCycle Forms, it is recommended that you plan your application. Creating application logic using the Form Server Module API represents only one aspect of creating a LiveCycle Forms application.

LiveCycle Forms requires form designs created using LiveCycle Designer. Form designs are XML templates that are saved as either .xdp or .pdf files. LiveCycle Forms outputs forms that are displayed as either HTML or PDF. The following diagram shows the valid input and output of LiveCycle Forms.



As shown in this diagram, if the form design is saved as an .xdp file, LiveCycle Forms can output a form that is displayed as either HTML or PDF. However, if the form design is saved as a .pdf file, LiveCycle Forms can only output a form that is displayed as PDF. That is, LiveCycle Forms cannot output a form that is displayed as HTML if the form design is saved as a .pdf file.

The first step in planning your application is to determine the output format of the forms. If you want LiveCycle Forms to output forms that are displayed as HTML, then save your form designs as .xdp files. For information about creating form designs to output as HTML, see [“Designing form designs to render as HTML” on page 18](#).

If you want LiveCycle Forms to output forms that are displayed as either HTML or PDF, save your form designs as .xdp files. If you want LiveCycle Forms to only output forms as PDF, save your form designs as .pdf or .xdp files. You specify the form’s output format when creating application logic. For information, see [“Setting preference options to render the form as PDF” on page 53](#).

Next, plan the content of your form designs. Form design content varies from simplistic form designs that contain text and text box fields to complex form designs that contain multiple pages, different controls (such as radio buttons and drop-down lists), and scripts. Even though this guide does not discuss how to use LiveCycle Designer to create form designs, it contains sections that discuss issues related to form designs. For information about using LiveCycle Designer to create form designs, see the *LiveCycle Designer Help*.

After you create your form designs, you are ready to use the Form Server Module API to start developing a LiveCycle Forms client application. The first step is to invoke LiveCycle Forms. For information, see [“Invoking LiveCycle Forms” on page 28](#).

Creating form designs for LiveCycle Forms

Behavioral differences exist between form designs that are used to render PDF and HTML. Because form designs that are rendered as PDF are viewed using Adobe Acrobat® Professional or Acrobat Standard or Adobe Reader®, the form supports a full range of object properties that you define in the form design.

When you create a form design that uses a client-side subform control (using the InstanceManager), you can successfully view the PDF form in LiveCycle Designer using the PDF Preview option. A user can dynamically add rows, without first submitting the form to LiveCycle Forms and having the form rendered back with the additional rows. However, to use the form design in LiveCycle Forms, you must render the form at the client. For information, see [“Rendering a form at the client” on page 74](#).

Static text placed in subforms that flow content left to right and contain rectangle fills does not display when rendered in Internet Explorer. To ensure that static text always appears, you need to place static text in subforms that position content left to right, rather than flow content left to right.

If you are rendering a form as HTML, some client devices (for example, older web browsers) do not provide the same level of support for individual object properties. To create a single form design that reduces these limitations, follow this process:

1. Consult the *Transformation Reference* to determine how objects behave in a particular client device.
2. If you are designing a static form and want to output the form as HTML, you must enable transformation caching. For information, see the *LiveCycle Designer Help*.
3. When creating the form design, try to work around any limitations in the client applications by finding ways to implement the form without relying on unsupported object properties.
4. If required, include a layout that works for both PDF and HTML formats.
5. Read the section in LiveCycle Designer Help that discusses creating accessible forms and use the guidelines to build accessibility into your form design.
6. Ask your form developer where scripts should run. By default, scripts run on the client. If the scripts that you include in a form design should run on the server, or both the client and server, you may have to change the default setting. For example, a form design may contain a script that extracts data from a database that is only available on the server. In this situation, the default setting must be modified so that the script runs at the server.
7. Periodically preview the form using LiveCycle Designer or the client device (for example, a web browser) to troubleshoot problems early in the design process.
8. If LiveCycle Forms will be merging forms with data, use test data to thoroughly test your form designs.

Designing form designs to render as HTML

This section discusses issues related to creating form designs that will be rendered as HTML.

HTML pages

When a form design is rendered as HTML, each second-level subform is rendered as an HTML page (panel). You can view a subform's hierarchy in LiveCycle Designer. Child subforms that belong to the root subform (the default name of a root subform is form1) are the panel subforms. The following example is of a form design's subforms.

```
form1
  Master Pages
  PanelSubform1
    NestedDynamicSubform
      TextEdit1
  PanelSubform2
    TextEdit1
  PanelSubform3
    TextEdit1
  PanelSubform4
    TextEdit1
```

When form designs are rendered as HTML, the panels are not constrained to any particular page size. If you have dynamic subforms, they should be nested within the panel subform. Dynamic subforms are able to expand to an infinite number of HTML pages.

When a form is rendered as HTML, page sizes (required for paginating forms rendered as PDF) have no meaning. Because a dynamic form can expand to an infinite number of HTML pages, it is important to avoid footers on the master page. A footer beneath the content area on a master page can overwrite HTML content that flows past a page boundary.

You must explicitly move from panel to panel using the `xfa.host.pageUp` and `xfa.host.pageDown` methods. You change pages by sending a form to LiveCycle Forms and having LiveCycle Forms render the form back to the client device, typically a web browser. For information about these methods, see the *XML Form Object Model Reference*. For information about this document, go to http://partners.adobe.com/public/developer/livecycle/topic_designer.html.

Note: The process of sending a form to LiveCycle Forms and then having LiveCycle Forms render the form back to the client device is referred to as round tripping data to the server.

Running scripts

A form author specifies whether a script executes on the server or the client. LiveCycle Forms creates a distributed, event processing environment for execution of form intelligence that can be distributed between the client and the server by using the `runAt` attribute. For information about this attribute, see the *LiveCycle Designer Help*.

LiveCycle Forms can execute scripts while the form is being rendered. As a result, you can prepopulate a form with data by connecting to a database or web services that may not be available on the client. You can also set a button's `Click` event to run on the server so that the client will round trip data to the server. This allows the client to run scripts that may require server resources, such as an enterprise database, while a user is interacting with a form. For information, see ["Calculating Form Data" on page 95](#).

You can design forms that move between pages (panels) by calling `xfa.host.pageUp` and `xfa.host.pageDown` methods. This script is placed in a button's `Click` event and the `runAt` attribute is set to `Both`. The reason you choose `Both` is so that Acrobat or Adobe Reader (for forms that are rendered as PDF) can change pages without going to the server and HTML forms can change pages by round tripping data to the server. That is, a form is sent to LiveCycle Forms, and a form is rendered back as HTML with the new page displayed.

For information about using the `xfa.host.pageUp` and `xfa.host.pageDown` methods, see the *XML Form Object Model Reference*. For information about this document, go to http://partners.adobe.com/public/developer/livecycle/topic_designer.html.

If a form design containing the `resetData` script method is rendered as an HTML form, the `resetData` method is not invoked within the form. However, if the form design is rendered as a PDF form, the `resetData` method is invoked within the form. This issue applies to LiveCycle Forms 6.0.1, 7.0, 7.0.1, 7.1, and 7.2.

The `resetData` method resets field values that are located within a form. For more information about this method, see the *Adobe XML Form Object Model 2.2 Reference* guide.

XFA subsets

When creating form designs to render as HTML, you must restrict your scripting of the XFA subset. The XFA subset supports JavaScript™.

Scripts that run on the client or run on both the client and the server must be written within the XFA subset. Scripts that run on the server can use the full XFA scripting model and also use FormCalc. For information about using JavaScript, see the *LiveCycle Designer Help*.

When running scripts on the client, only the current panel being displayed can use script. For example, you cannot script against fields that are located in panel A when panel B is displayed. When running scripts on the server, all panels can be accessed.

You must also be careful when using Scripting Object Model (SOM) expressions within scripts that run on the client. Only a simplified subset of SOM expressions are supported by scripts that run on the client.

Event timing

The XFA subset defines what XFA events are mapped to HTML events. There is a slight difference in behavior on the timing of calculate and validate events. In a web browser, a full calculate event is executed when you exit a field. Calculate events are not automatically executed when you make a change to a field value. You can force a calculate event by calling the `xfa.form.execCalculate` method.

In a web browser, validate events are only executed when exiting a field or submitting a form. You can force a validate event by using the `xfa.form.execValidate` method.

Forms displayed in a web browser (as opposed to Adobe Reader or Acrobat) conform to the XFA null test (errors or warnings) for mandatory fields. If the null test produces an error and you exit a field without specifying a value, a message box is displayed and you are repositioned to the field after clicking OK. If a null test produces a warning and you exit a field without specifying a value, you are prompted to click either OK or Cancel, giving you the option of proceeding without specifying a value or returning to the field to enter a value.

For more information about a null test, see *LiveCycle Designer Help*.

LiveCycle Designer buttons

Clicking a submit button sends form data to LiveCycle Forms and represents the end of form processing. Data is submitted to LiveCycle Forms and, if no validation errors occur, the data is sent to a client application. For an example of a client application, see [“Sample loan application” on page 49](#).

The `preSubmit` event can be set to run on the client or server. The `preSubmit` event runs prior to the form submission if it is configured to run on the client. Otherwise, the `preSubmit` event runs on the server during the form submission. For more information about the `preSubmit` event, see the *LiveCycle Designer Help*.

If validation errors occur, they are displayed according to the validation UI options that are passed to the `renderForm` method's `sOption` parameter. For information about this parameter, see the *Form Server Module API Reference*.

If a button has no client-side script associated with it, data is submitted to the server, calculations are performed on the server, and the HTML form is regenerated. If a button contains a client-side script, data is not sent to the server and the client-side script is executed in the web browser.

HTML 4.0 web browser

A web browser that supports HTML 4.0 does not support the XFA subset nor client-side scripting. When creating a form design to work in both HTML 4.0 and MSDHTML or CSS2HTML, a script that runs at the client will actually run on the server. For example, assume a user clicks a button that is located on a form displayed in an HTML 4.0 web browser. In this situation, the form is sent to the server where the client-side script is executed.

It is recommended that you place your form logic in calculate events, which runs at the server in HTML 4.0 and on the client for MSDHTML or CSS2HTML.

Maintaining presentation changes

As you move between HTML pages (panels), only the state of the data is maintained. Settings such as background color or mandatory field settings are not maintained (if different than the initial settings). To maintain the presentation state, you must create fields (usually hidden) that represent the presentation state of fields. If you add script to a field's `Calculate` event that changes the presentation based on hidden field values, you are able to preserve the presentation state as you move back and forth between HTML pages (panels).

The following script maintains the `fillColor` of a field based on the value of `hiddenField`. Assume this script is located in a field's `Calculate` event.

```
If (hiddenField.rawValue == 1)
    this.fillColor = "255,0,0"
else
    this.fillColor = "0,255,0"
```

Caching forms

LiveCycle Forms caches a form for performance reasons. A form is cached only if the form designer enables a form design to be cached. For information, see the *LiveCycle Designer Help*.

Only form designs that have a static presentation should be cached. LiveCycle Forms caches forms if caching is enabled regardless whether the form design is suitable for caching.

For information about caching a form that is rendered as a PDF, see [“Caching PDF forms” on page 54](#).

LiveCycle Forms processing requests

Before using the Form Server Module API to create a client application, it is important to understand how LiveCycle Forms processes a request. This section describes how LiveCycle Forms processes requests such as a form request, and specifies the order in which events and scripts execute.

Requesting a form

When a user requests a form from LiveCycle Forms (for example, by clicking a button located on an HTML page), the request initiates a series of specific processes and interactions among the client application, LiveCycle Forms, and the client device, typically a web browser. The client application invokes the Form Server Module API, which is used to render the form to the client device. For information, see [“Invoking LiveCycle Forms” on page 28](#).

The following table summarizes the interaction among a client device (for example, a web browser), a client application (created by using the Form Server Module API), and LiveCycle Forms when a user requests a form.

User action	Client application action	LiveCycle Forms action
A user invokes LiveCycle Forms from a web page.	Creates an object that inherits from the <code>IFormServer</code> interface (for example, an <code>EJBClient</code> object) and calls the <code>renderForm</code> method. For information, see “Invoking the Form Server Module” on page 30 .	Opens the form design. The form design is specified by the <code>renderForm</code> method's <code>sFormQuery</code> parameter.
		If data is passed to LiveCycle Forms (in the <code>renderForm</code> method's <code>cData</code> parameter), LiveCycle Forms prepopulates the form with the data.

User action	Client application action	LiveCycle Forms action
		Executes all form-wide field initialize events.
		Executes all form-wide page initialize events.
		Executes all form-wide field calculate events.
		Executes all form-wide page calculate events.
		Executes a page enter event.
		Executes a form ready event.
		Executes a page enter or exit event.
		Transforms the form design into PDF or HTML format. This change is defined by the <code>renderForm</code> method's <code>sFormPreference</code> parameter.
		Returns the form to the client application.
	Verifies that an error was not returned.	
	Creates a binary stream and sends it to the client web browser.	

Internet Explorer 5.0, Netscape Navigator 6.0, and Opera 5.0 browsers perform these actions:

- Runs each field initialization marked Run script on client.
- Runs the page initialization marked Run script on client.
- Runs each field calculation marked Run script on client.
- Runs the page calculations marked Run script on client.

Note: These actions only occur if the form is rendered as HTML.

Views the form as either PDF or HTML.

Using Form Design buttons

For LiveCycle Forms to retrieve form data, perform calculations, or validate field data, the form must provide the mechanism to initiate the request. This is typically accomplished through the use of buttons that are located on the form design. The caption displayed on a command button label indicates to the end user the function of the button. When a user clicks a button, the form-related processing is prompted by the script associated with the button. Typically, a button initiates either a submit or a calculate operation.

Buttons are the most common way to initiate logic contained in form design scripts. Placing a button on a form design in LiveCycle Designer and configuring its submit option implies a submit operation. The intent of a submit button is to complete the form and submit data to LiveCycle Forms. However, validation operations may interrupt this process. For example, if a user enters a wrong value into a field, the user may have to correct the value before the form data can be submitted to LiveCycle Forms. Placing other button types on the form implies a calculate operation. The intent of a calculate operation is to run calculations and update the form prior to a submit operation.

Submit button

A button can submit form data as either XML or PDF data to LiveCycle Forms. For example, assume a user fills an interactive form and then clicks a submit button. This action results in the form data being submitted to LiveCycle Forms. A client application, such as a Java servlet that is created by using the Form Server Module API, can retrieve the data. For information, see [“Retrieving submitted form data” on page 60](#).

A PDF form can submit up to four different variations (XDP, XML, PDF, and URL encoded data). An HTML form only submits URL encoded name-value pairs. By default, when the submission format is PDF, LiveCycle Forms captures the PDF data and returns it back out without performing any calculations. You set the submit type in LiveCycle Designer. For information, see the *LiveCycle Designer Help*.

The content type of submitted PDF data is `application/pdf`. In contrast, the content type of submitted XML data is `text/xml`. You can transfer submitted PDF data to other LiveCycle products. For information, see [“Transferring PDF Data” on page 103](#).

The following table summarizes the interaction among a client device (such as a web browser), a client application, and LiveCycle Forms when a user clicks a button that initiates a submit operation.

User actions	Client application actions	LiveCycle Forms actions
A user enters data into form fields and clicks a submit button. This initiates a submit operation.	Client validations marked run on client are executed.	
		Browser performs an HTTP post to the target URL (this value is defined either in LiveCycle Designer or by the <code>renderForm</code> method's <code>sTargetURL</code> parameter).

User actions	Client application actions	LiveCycle Forms actions
	<p>Creates an object that inherits from the <code>IFormServer</code> interface (for example, an <code>EJBClient</code> object) and calls the <code>processFormSubmission</code> method. For information, see “Retrieving submitted form data” on page 60.</p>	
		<p>LiveCycle Forms merges posted data back into the form. (if applicable).</p>
		<p>Executes the field click event.</p>
		<p>Executes the form-wide field calculate events.</p>
		<p>Executes the form-wide page calculate events.</p>
		<p>Executes the form-wide field validation events.</p>
		<p>Executes the page validation events (which include <code>validate</code>, <code>formatTest</code>, and <code>nullTest</code>).</p>
		<p>Executes the Form <code>Close</code> event.</p>
		<p>If this validation process fails, it indicates that at least one error exists. The returned <code>FSAction</code> code is set to <code>FSValidate</code>. For information, see “Retrieving submitted form data” on page 60.</p>
	<p>Verifies that LiveCycle Forms returned an <code>FSAction</code> code of <code>FSValidate</code>. In this situation, the result is sent back to the client browser so that the user can correct the mistake.</p>	
	<p>Performs a binary write operation to the browser with the form.</p>	

User actions	Client application actions	LiveCycle Forms actions
<p>For forms that are displayed as HTML, the end user sees the form containing the same data, calculations, and list of errors to correct before resubmitting.</p> <p>For forms that are displayed as PDF, a user interface is not defined. Validation errors can be retrieved by using the <code>OutputContext</code> interface's <code>getValidationErrorsList</code> method.</p>	<p>Verifies that LiveCycle Forms returns an <code>FSAction</code> value of <code>FSSubmit (0)</code>.</p> <p>Acknowledges that all form processing is complete and that LiveCycle Forms has returned XML data. For information about retrieving the XML data, see “Creating application logic to retrieve submitted data” on page 62.</p> <p>Any additional processing is application specific. For example, a wizard-style application can request the next form panel, do additional data investigations, update the database, or initiate a new workflow process.</p>	<p>If this validation process succeeds, the returned <code>FSAction</code> code is set to <code>FSSubmit</code>.</p>
<p>The view is application specific. For example, a new form can be displayed.</p>		

Calculate button

A button can make a request to LiveCycle Forms to execute a calculation operation. When a user clicks a button, a `ProcessHttpRequest` call is issued to LiveCycle Forms. Then LiveCycle Forms executes the calculation operation and returns calculation results within the form. For information, see [“Calculating Form Data” on page 95](#).

The following table summarizes the interaction among a client device (such as a web browser), a client application, and LiveCycle Forms when a user clicks a button that initiates a calculation operation.

User actions	Client application actions	LiveCycle Forms actions
<p>A user clicks a button that is located on a form.</p> <p>If the button’s <code>Click</code> event is marked run on client, the form is not submitted to LiveCycle Forms. The script is executed in a web browser, Acrobat, or Adobe Reader.</p> <p>If the button’s <code>Click</code> event is marked run on server, the form is submitted to LiveCycle Forms.</p>	<p>Creates an object that inherits from the <code>IFormServer</code> interface (for example, an <code>EJBClient</code> object) and calls the <code>processFormSubmission</code> method.</p> <p>For information, see “Handling a form containing a script” on page 96.</p>	<p>LiveCycle Forms merges new data into the form design (if applicable).</p> <p>Executes the field click event.</p> <p>Executes the form-wide field calculate events.</p> <p>Executes the form-wide page calculate events.</p> <p>Executes a page enter or exit event.</p> <p>Executes the form-wide field validation events.</p>

User actions	Client application actions	LiveCycle Forms actions
		Executes the page validation event.
		Executes the page exit event
		Returns the form to the client application that invoked LiveCycle Forms. The form's format does not change. If the form is submitted in PDF, it is sent back to the client browser in PDF. Sets the value of <code>FSAction</code> to calculate (1). You can retrieve this value by calling the <code>IOutputContent</code> interface's <code>getFSAction</code> method.
	Verifies that LiveCycle Forms did not return an error.	
	Creates a binary stream and sends it to the client web browser.	
Views calculation results that are displayed in the form.		

2

Invoking LiveCycle Forms

Creating client applications that interact with LiveCycle Forms requires invoking modules by using different APIs. Each API lets you invoke a different LiveCycle Forms module. For example, you can create an application that renders a form that is displayed in PDF to a client web browser by invoking the Form Server Module.

The LiveCycle Forms APIs are implemented in Java and have public methods that enable you to invoke LiveCycle Forms services installed on a J2EE application server. As a result, you need to use a Java development environment to create applications. The only exception is a managed client assembly for Microsoft .NET. Using this client assembly, you can create client applications in the Microsoft Visual Studio .NET 2003 development environment that interact with the Form Server Module. The Form Server Module API is the only API that has a Microsoft .NET client assembly.

This chapter contains the following information.

Topic	Description	See
Including LiveCycle Forms library files	Describes how the different API modules are packaged.	page 28
Invoking the Form Server Module	Describes how to create Form Server Module API objects, which are necessary to invoke the Form Server Module.	page 30
Invoking the Data Manager Module	Describes how to create Data Manager Module API objects, which are necessary to invoke the Data Manager service.	page 40
Invoking the XML Form Module	Describes how to create XML Form Module API objects, which are necessary to invoke the XML Form Module.	page 45

Including LiveCycle Forms library files

You need to include the following JAR files in your Java project to successfully invoke LiveCycle Forms:

formserver-client.jar: This JAR file that contains the Form Server Module API and must be used when invoking the Form Server Module.

adobe-common.jar: This file is necessary to reference in your application's class path when you are using the Form Server Module API. For example, this JAR file contains the `com.adobe.idp.Document` class, which the Form Server Module API methods use. For information, see ["Working with the Document object" on page 43](#).

AdobeCSAUtils.jar: This file is necessary to use the Form Server Module.

datamanager-client.jar: This file lets you invoke the Data Manager Module using the Data Manager Module API and has to be referenced in your application's class path.

DocumentServicesLibrary.jar: This file is necessary to use the Data Manager Module API. For example, this JAR file defines `com.adobe.service.ConnectionFactory`.

um-client.jar: This file must be referenced in your application's class path to authenticate users with User Manager. For information, see ["Authenticating Users" on page 110](#).

xmlform-client.jar: This file lets you invoke the XML Form Module using the XML Form Module API. You do not have to include this JAR file unless you want to invoke the XML Form Module.

axis.jar: This file must be referenced in your application's class path to invoke LiveCycle Forms using SOAP.

commons-discovery.jar: This file must be referenced in your application's class path to invoke LiveCycle Forms using SOAP.

commons-logging.jar: This file must be referenced in your application's class path to invoke LiveCycle Forms using SOAP.

saaj.jar: This file must be referenced in your application's class path to invoke LiveCycle Forms using SOAP.

log4j-1.2.8.jar: This file must be referenced in your application's class path to invoke LiveCycle Forms using SOAP (or the appropriate JAR file for your specific logging implementation)

jaxrpc.jar: This file must be referenced in your application's class path to invoke LiveCycle Forms using SOAP.

j2ee.jar: This file must be referenced in your application's class path to invoke LiveCycle Forms using SOAP and if you are outside of a J2EE application server.

Obtaining Axis library files

You need to obtain the JAR files specified in this section in order to invoke LiveCycle Forms using SOAP (that is, using the `SOAPClient` class). You can obtain the latest JAR files from the <http://ws.apache.org/axis/java/install.html> website.

Upgrading LiveCycle Forms JAR files

When upgrading from a previous version of LiveCycle Forms to LiveCycle Forms 7.2, it is strongly recommended that you update the JAR files in your client application. You also may have to modify your application logic if you are currently passing null to the `renderForm` method's `cData` parameter. For information, see ["Passing a zero-length byte array" on page 53](#).

The following table lists the installation location of JAR files that are installed with LiveCycle Forms. The JAR files not listed, such as `axis.jar`, are in the install directory of the J2EE application server on which LiveCycle Forms is deployed.

File	Location
formserv-client.jar	<ul style="list-style-type: none">● C:\Adobe\LiveCycle\components\forms\common\lib\adobe (Microsoft Windows®)● /opt/adobe/livecycle/components/forms/common/lib/adobe (UNIX®)
adobe-common.jar	<ul style="list-style-type: none">● C:\Adobe\LiveCycle\components\csa\common\lib\adobe (Windows)● /opt/adobe/livecycle/components/csa/common/lib/adobe (UNIX)
AdobeCSAUtils.jar	<ul style="list-style-type: none">● C:\Adobe\LiveCycle\components\csa\<app_server>\lib\adobe (Windows)● /opt/adobe/livecycle/components/csa/<app_server>/lib/adobe (UNIX)
datamanager-client.jar	<ul style="list-style-type: none">● C:\Adobe\LiveCycle\components\csa\common\lib\adobe (Windows)● /opt/adobe/livecycle/components/csa/common/lib/adobe (UNIX)

File	Location
DocumentServices Library.jar	<ul style="list-style-type: none"> • C:\Adobe\LiveCycle\components\csa\<app_server>\lib\adobe (Windows)</app_server> • /opt/adobe/livecycle/components/csa/<app_server>/lib/adobe (UNIX)
um-client.jar	<ul style="list-style-type: none"> • C:\Adobe\LiveCycle\components\um\<app_server>\lib\adobe (Window)</app_server> • /opt/adobe/livecycle/components/um/<app_server>/lib/adobe (UNIX)
xmlform-client.jar	<ul style="list-style-type: none"> • C:\Adobe\LiveCycle\components\xmlform\common\lib\adobe (Windows) • /opt/adobe/livecycle/components/xmlform/common/lib/adobe (UNIX)

Note: C:\ or /opt is the drive on which LiveCycle Forms is installed. For information about installing LiveCycle Forms, see the *Installing and Configuring* guide for your application server.

Invoking services

After you add the library files to your Java project, you can invoke LiveCycle Forms by creating the objects specified in this table.

Module	API	Invoking object	See
Form Server Module	Form Server Module API	<ul style="list-style-type: none"> • EJBClient • SOAPClient 	page 30 page 34
Data Manager	Data Manager Module API	<ul style="list-style-type: none"> • DataManager • Document • FileDataBuffer 	page 40
XML Form	XML Form Module API	<ul style="list-style-type: none"> • Form 	page 45

Note: The remaining sections explain how to invoke each module.

Invoking the Form Server Module

The Form Server Module API consists of two Java classes that you can use to instantiate objects used to invoke the Form Server Module. The first client class is named `EJBClient` and enables you to create client applications, such as Java servlets, within the J2EE development environment. The second class is named `SOAPClient` and enables you to create client applications that use SOAP to invoke the Form Server Module.

Both the `EJBClient` class and `SOAPClient` class inherit from the `IFormServer` interface. As a result, both classes support rendering forms displayed as either PDF or HTML to a client web browser. For information, see [“Rendering Interactive Forms as PDF” on page 48](#).

Both classes also support handling submitted forms. For information, see [“Retrieving submitted form data” on page 60](#).

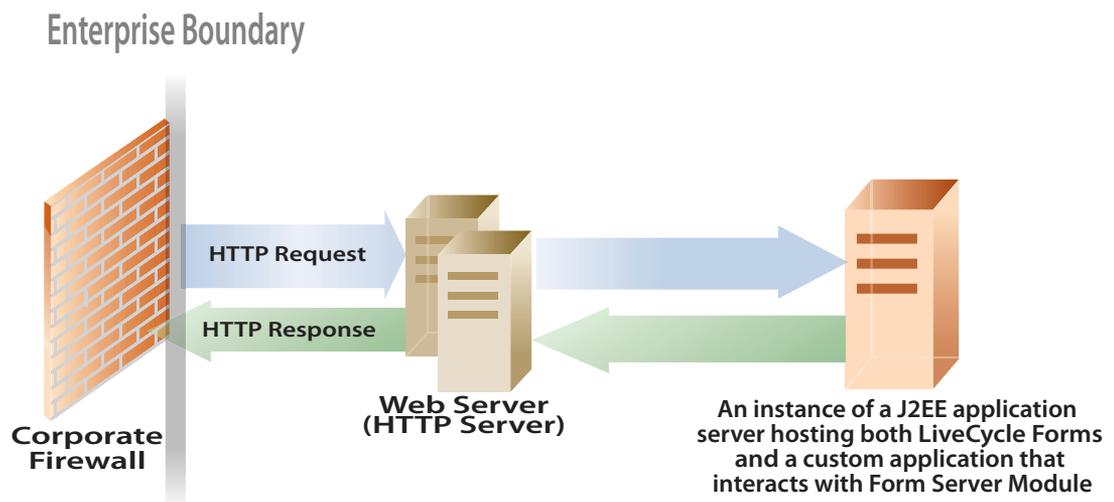
You can invoke the Form Server Module by using one of these methods:

- Locally using an `EJBClient` object
- Remotely using an `EJBClient` object
- Using a `SOAPClient` object
- Using the Microsoft .NET client assembly
- Using the `FormServerFactory` class

Note: All of these methods create either an `EJBClient` or a `SOAPClient` object, both of which are implementations of the `IFormServer` interface.

Locally invoking Form Server Module

You can locally invoke the Form Server Module using the `EJBClient` class. In this situation, the client application that contains the invoking `EJBClient` object is located on the same J2EE application server hosting Form Server Module.



Invoking the Form Server Module locally by using the `EJBClient` class is a three-step process:

1. Add the necessary JAR files to your Java project's build path. For information, see ["Including LiveCycle Forms library files" on page 28](#).

2. Add the following import statements to your Java project:

```
import com.adobe.formServer.client.EJBClient;  
import com.adobe.formServer.interfaces.*;
```

3. Use the `EJBClient` constructor to create an `EJBClient` object:

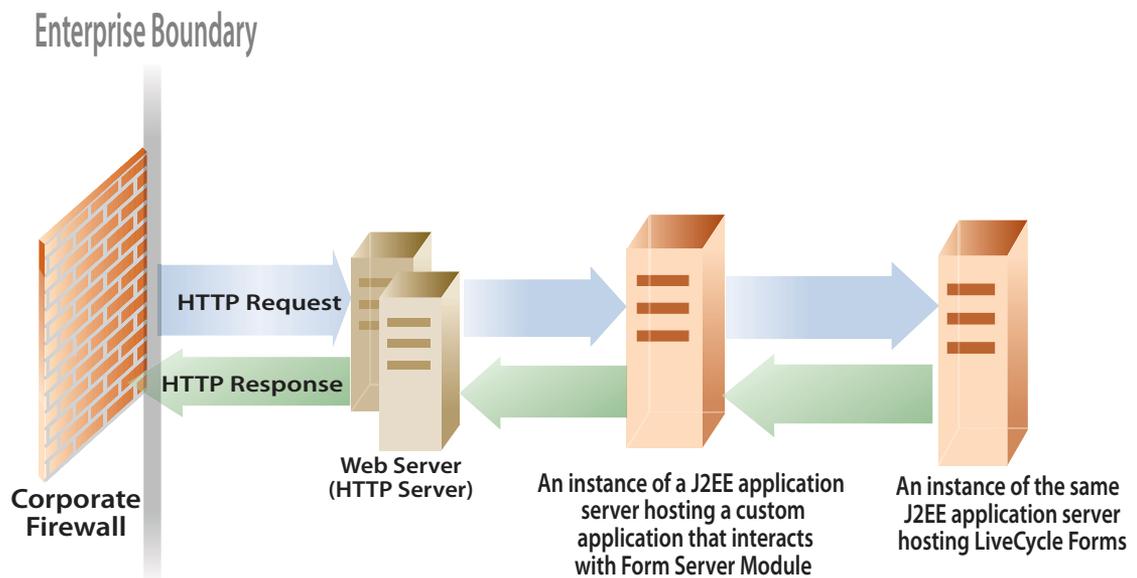
```
EJBClient formServer = new EJBClient();
```

After you create an `EJBClient` object, you can perform tasks such as rendering a form. For information, see ["Rendering a form using an EJBClient object" on page 52](#).

Note: Throughout this guide, the Form Server Module is invoked locally.

Remotely invoking Form Server Module

You can remotely invoke the Form Server Module using an `EJBClient` object. In this situation, the client application that contains the invoking `EJBClient` object and the Form Server Module are located on separate J2EE application servers.



The Form Server Module API is the only LiveCycle Forms API that can be used in a client application that is located on a separate J2EE application server. If you deploy your client application to a separate application server, then you cannot use the XML Form Module API or the Data Manager API. To use these APIs, you must deploy your client application to the J2EE application server on which LiveCycle Forms is deployed.

Caution: Remotely invoking the Form Server Module may not have the same performance as locally invoking the Form Server Module.

Creating application logic to remotely invoke Form Server Module

Invoking the Form Server Module remotely by using an `EJBClient` object involves performing a Java Naming and Directory Interface (JNDI) look-up operation. The J2EE application servers on which the client application and the Form Server Module are deployed must be the same and support a JNDI look-up operation.

The client application performs a JNDI look-up on the J2EE application server to get an EJB reference to the Form Server Module. To perform a JNDI look-up, a client application needs to pass the following information to the initial context:

- A provider URL
- Server-specific naming context factory

You can pass this information in two different ways:

- Create a JNDI properties file that specifies a provider URL and a naming context factory. Create the client application to read the information and programmatically create a Java `InitialContext` object. You set this information during the initial context look-up.
- Create a `jndi.properties` file in the client application's class path. When the client application creates a Java `InitialContext` object, it automatically gets the values located in the `jndi.properties` file.

WebSphere

The following example shows the contents of a `jndi.properties` file that is used to invoke the Form Server Module that is deployed on WebSphere.

```
java.naming.factory.initial=com.ibm.websphere.naming.  
WsnInitialContextFactory  
java.naming.provider.url=iiop://<AppServer>:<AppPort>
```

JBoss

The following example shows the contents of a `jndi.properties` file that is used to invoke the Form Server Module that is deployed on JBoss.

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory  
java.naming.provider.url=<AppServer>:<AppPort>  
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

WebLogic

The following example shows the contents of a `jndi.properties` file used to invoke the Form Server Module that is deployed on WebLogic.

```
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory  
java.naming.provider.url=t3://<AppServer>:<AppPort>
```

Invoking the Form Server Module

Invoking the Form Server Module remotely is a seven-step process:

1. Add the necessary JAR files to your Java project's build path. For information, see ["Including LiveCycle Forms library files" on page 28](#).
2. Add the following import statements to your Java project:

```
import com.adobe.formServer.client.EJBClient;  
import com.adobe.formServer.interfaces.*;
```
3. Use the `EJBClient` constructor to create an `EJBClient` object.
4. Create a `java.util.Hashtable` object to store JNDI environment values.
5. Populate the `java.util.Hashtable` object with JNDI environment name-value pairs by calling its `put` method (an example of this step is shown in the code listing that follows this list). Among the values that you must specify is the URL of the J2EE application server hosting LiveCycle Forms.

6. Create a `javax.naming.InitialContext` object by using its constructor and pass the `java.util.Hashtable` object.
7. Call the `EJBClient` object's `setInitialContext` method and pass the `javax.naming.InitialContext` object.

The following code example shows the application logic to remotely invoke the Form Server Module. Both the client application and the Form Server Module are running on their own instances of JBoss.

Example 2.1 Remotely invoking Form Server Module using the EJBClient object

```
//Create an EJBClient object
EJBClient formServer = new EJBClient();

// Create a Hashtable object
Hashtable propsJNDI = new Hashtable();

//Populate the Hashtable object with JNDI environment values
propsJNDI.put("java.naming.factory.initial", "org.jnp.interfaces.Naming
ContextFactory");
propsJNDI.put("java.naming.provider.url", "jnp://myJBossServer:1099");
propsJNDI.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.
interfaces");

try {
    //Create an InitialContext object
    InitialContext initialContextOb = new InitialContext(propsJNDI);

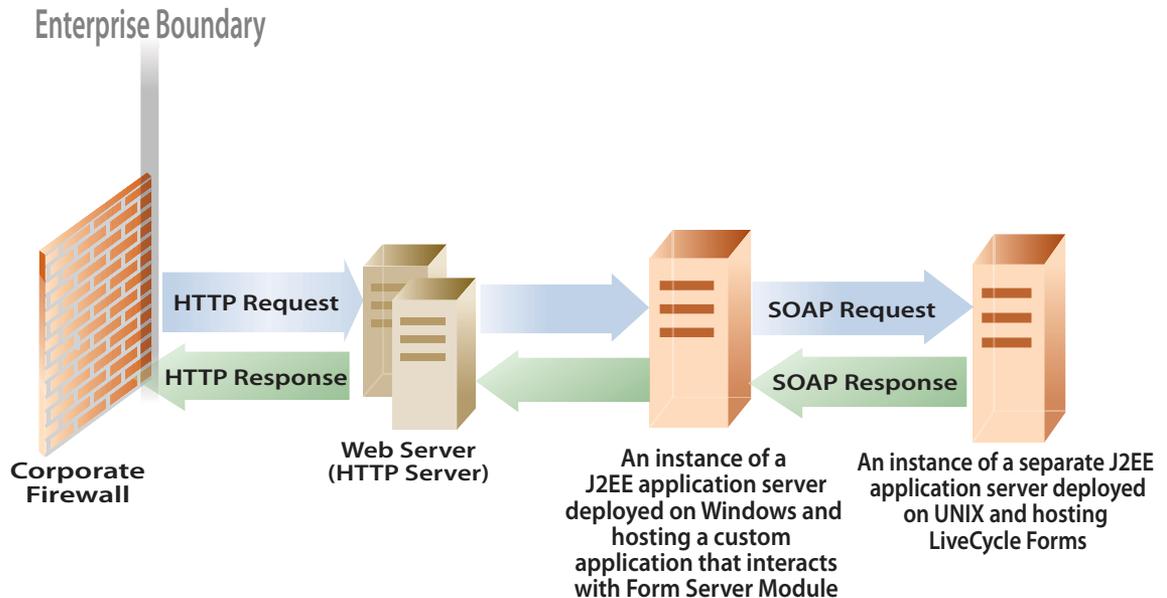
    //Call the EJBClient object's setInitialContext object
    formServer.setInitialContext(initialContextOb);
}

//Catch an exception
catch (Exception ex)
{
    System.out.println("LiveCycle Forms exception is "+ ex.getMessage());
}
```

Invoking Form Server Module using SOAP

You can remotely invoke the Form Server Module using a `SOAPClient` object. The client application that contains the invoking `SOAPClient` object is usually installed on a separate J2EE application server from the J2EE application server hosting the Form Server Module.

The two J2EE application servers do not have to be the same and can also be running on different operating systems. For example, the J2EE application server hosting the client application may be running on Windows, and the J2EE application server hosting the Form Server Module may be running on UNIX.



SOAP works through firewalls and can be load balanced using HTTP load-balancing tools. In addition, you can also use the `SOAPClient` object to locally invoke the Form Server Module.

Creating application logic to remotely invoke the Form Server Module

You instantiate the `SOAPClient` class to invoke the Form Server Module. Set the SOAP endpoint by invoking the `SOAPClient` object's `setSoapEndPoint` method and passing a valid SOAP endpoint as an argument. The following table specifies valid SOAP endpoints for specific J2EE application servers.

J2EE application server	SOAP endpoint
JBoss	<code>http://<AppServerURL>:8080/jboss_net/services/AdobeFSService</code>
WebLogic	<code>http://<AppServerURL>:7001/FormServerWS/services/AdobeFSService</code>
WebSphere	<code>http://<AppServer>:9080/FormServerWS/services/AdobeFSService</code>

`AppServerURL` represents the host name of the computer on which LiveCycle Forms is deployed.

Invoking the Form Server Module by using the `SOAPClient` class is a four-step process:

1. Add the necessary JAR files to your Java project's build path. For information, see ["Including LiveCycle Forms library files" on page 28](#).

2. Add the following import statements to your Java project:

```
import com.adobe.formServer.client.SOAPClient;
import com.adobe.formServer.interfaces.*;
```

3. Use the `SOAPClient` constructor to create a `SOAPClient` object:

```
SOAPClient formServer = new SOAPClient();
```

4. Set the `SOAPClient` object's endpoint by calling the `setSoapEndPoint` method. This method specifies the location of the J2EE application server that hosts LiveCycle Forms.

The following code example shows the application logic to remotely invoke the Form Server Module by using a `SOAPClient` object.

Example 2.2 Remotely invoking Form Server Module using the `SOAPClient` class

```
//Create a SOAPClient object
SOAPClient formServer = new SOAPClient();

//Set the SOAPClient object's SOAP end point
formServer.setSoapEndPoint("http://<AppServerURL>:8080/jboss_net/services/
AdobeFSService");
```

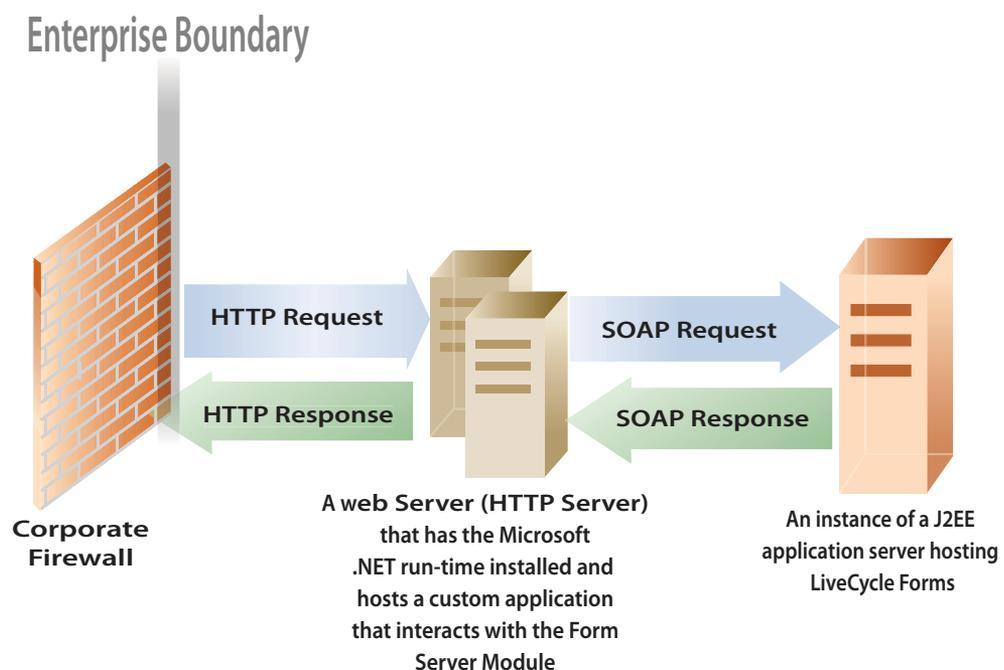
Note: After you create a `SOAPClient` object, you can perform tasks such as rendering a form to a client web browser. For information, see [“Rendering a Form using a SOAPClient object” on page 58](#).

Invoking Form Server Module using the Microsoft .NET client assembly

You use the Microsoft .NET client assembly to create client applications in the Microsoft .NET development environment. This client assembly consists of a file named `SoapClient.dll` and is located in the LiveCycle Forms library directory. For information about this directory, see [“Including LiveCycle Forms library files” on page 28](#).

The Microsoft .NET client assembly contains two main classes that enable you to create client applications in the Microsoft .NET development environment. The first class is `FSSoapClient` and is equivalent to the `SOAPClient` class used in the Java implementation. The second class is named `IOutputContent` and is equivalent to the `IOutputContext` interface used in the Java implementation.

The client application that contains the Microsoft .NET client assembly must reside on an application server that has the Microsoft .NET run-time installed. LiveCycle Forms is installed on a J2EE application server.



Creating application logic to invoke the Form Server Module

Invoking the Form Server Module by using the Microsoft .NET client assembly is a four-step process:

1. From the Microsoft .NET Visual Studio development environment, reference the SoapClient.dll file.
2. Add the following import statement to your Microsoft .NET project:

```
using SoapClient ; // C# syntax
imports SoapClient ' VB.NET syntax
```

3. Use the FSSoapClient constructor to create a FSSoapClient object:

```
FSSoapClient formServer = new FSSoapClient(); // C# syntax
Dim formServer As New FSSoapClient ' VB.NET syntax
```

4. Set the SOAPClient object's SOAP endpoint by calling the setSoapEndPoint method:

```
formServer.setSoapEndPoint ("http://<AppServer>:<AppPort>/FormServerWS/
services/AdobeFSService"); //C# syntax
formServer.setSoapEndPoint ("http://<AppServer>:<AppPort>/FormServerWS/
services/AdobeFSService") ' VB.NET syntax
```

Note: After you create a FSSoapClient object, you can perform tasks such as rendering a form. For information, see [“Rendering Forms from .NET” on page 114](#).

Referencing the Microsoft .NET client assembly

You create a reference to the SoapClient.dll assembly within the Microsoft .NET Visual Studio development environment. Before doing so, ensure that LiveCycle Forms is installed on your development computer.

Getting the ICSharp utility

The Form Server SOAP client is dependent on an ICSharp utility, which consists of a single file named ICSharpCode.SharpZipLib.dll. You must reference this file from your Microsoft .NET Visual Studio project. You can download this utility from the following website:

<http://prdownloads.sourceforge.net/sharpdevelop/050SharpZipLib.zip?download>.

► To reference the LiveCycle Forms SOAP client within Microsoft .NET Visual Studio

1. From the **Project** menu, click **Add Reference**.
2. Click the **.NET** tab.
3. Click **Search** and navigate to the LiveCycle Forms library directory located at C:\Adobe\LiveCycle\components\forms\common\lib\adobe.
4. Select **SoapClient.dll**.

Creating Form Server Module objects using the FormServerFactory class

You can use the `FormServerFactory` class to invoke the Form Server Module. This class has a static method named `create` that returns an `IFormServer` interface that is based on a `SOAPClient` object or an `EJBClient` object. You pass a `java.util.Properties` object to the `create` method as an argument. You must invoke the `create` method within a `try` statement.

Using the `java.util.Properties` object's `setProperty` method, specify a value for the following properties:

- `FormServerFactory.CLASSNAME_PROP`—Specifies whether to create a `SOAPClient` or an `EJBClient` object.
- `FormServerFactory.INITIALCONTEXT_PROP`—Specifies the `EJBClient` object's initial context. You can specify the name of the `javax.naming.InitialContext` object. This property is set only if you are remotely invoking the Form Server Module. For information, see ["Creating an EJBClient object to remotely invoke Form Server Module" on page 39](#).
- `FormServerFactory.ENDPOINT_PROP`—Specifies the SOAP endpoint. This property is set only if you are creating a `SOAPClient` object. For information, see ["Invoking Form Server Module using SOAP" on page 34](#).

The difference between using a `new` operator to instantiate a `SOAPClient` or `EJBClient` object (as shown throughout this chapter) or using the `FormServerFactory` class is a matter of which Java syntax you prefer using. There is no performance advantage in using one method over the other.

Note: It is possible to omit specifying a `CLASSNAME_PROP` value. By default, the `create` method returns an `EJBClient` object. If you specify an `ENDPOINT_PROP` value, the `create` returns a `SOAPClient` object. If you specify an `INITIALCONTEXT_PROP` value, the `create` method returns an `EJBClient` object.

Creating a SOAPClient object using the FormServerFactory class

The following code example creates a `SOAPClient` object by using the `FormServerFactory` class.

Example 2.3 Creating a SOAPClient object using the FormServerFactory class

```
//Create an IFormServer interface based on an EJBClient object
Properties props = new Properties();

//Define the properties
props.setProperty(FormServerFactory.CLASSNAME_PROP,
"com.adobe.formserver.client.SOAPClient");
props.setProperty(FormServerFactory.ENDPOINT_PROP,
"http://<AppServerURL>:8080/jboss_net/services/AdobeFSService");

try{
    //Call create
    IFormServer formServer = FormServerFactory.create(props);
}

//Catch an exception
catch (Exception ex)
{
    System.out.println("LiveCycle Forms exception is " + ex.getMessage());
}
```

Creating an EJBClient object to locally invoke Form Server Module

The following code example creates an `EJBClient` object by using the `FormServerFactory` class. This object is used to locally invoke the Form Server Module because the `FormServerFactory.INITIALCONTEXT_PROP` property is not defined. Notice that the return value is cast to `EJBClient`.

Example 2.4 Creating a local EJBClient object using the FormServerFactory class

```
// Create an EJBClient object
Properties props = new Properties ();

//Define the properties
props.setProperty(FormServerFactory.CLASSNAME_PROP,
"com.adobe.formServer.client.EJBClient");

try
{
    //Call create
    IFormServer formServer = FormServerFactory.create(props);
}
//Catch an exception
catch (Exception ex)
{
    System.out.println("LiveCycle Forms exception is "+ ex.getMessage());
}
```

Creating an EJBClient object to remotely invoke Form Server Module

To remotely invoke LiveCycle Forms by using EJB, create an `EJBClient` object by invoking the `FormServerFactory` object's `create` method.

Before you invoke the `create` method, you must specify JNDI environment values to enable your client application to perform a Java look-up operation. The J2EE application servers on which the client application and LiveCycle Forms are deployed must be the same and must support a JNDI look-up operation.

Create a `java.util.Hashtable` object that is used to store the JNDI environment values. You must ensure that you use JNDI environment values that are compatible with the J2EE application server on which LiveCycle Forms is deployed. For example, assuming that LiveCycle Forms is deployed on JBoss, you can use the following JNDI environment values:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://<AppServer>:<AppPort>
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

After you populate a `java.util.Hashtable` object with JNDI environment values, create a `javax.naming.InitialContext` object by using its constructor and pass the `java.util.Hashtable` object that stores the JNDI environment values.

Next, create a `java.util.Properties` object by using its constructor. Invoke the `java.util.Properties` object's `put` method and pass the following arguments:

- `FormServerFactory.INITCONTEXT_PROP` static property
- `javax.naming.InitialContext` object

You can omit the `FormServerFactory` object's `CLASSNAME_PROP` property. By default, the `create` method returns an `EJBClient` object.

Invoke the `FormServerFactory` object's static `create` method and pass the `java.util.Properties` object. This method returns an `EJBClient` object that is used to remotely invoke LiveCycle Forms. The following code example shows how to create an `EJBClient` object to remotely invoke LiveCycle Forms (this instance of LiveCycle Forms is deployed on JBoss).

Example 2.5 Creating a remote EJBClient object using the FormServerFactory class

```
// Create a Hashtable object
Hashtable propsJNDI = new Hashtable();

// Populate the Hashtable object with JNDI environment values
propsJNDI.put("java.naming.factory.initial", "org.jnp.interfaces.
NamingContextFactory");
propsJNDI.put("java.naming.provider.url", "jnp://myJBossServer:1099");
propsJNDI.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.
interfaces");

try{

    //Create an InitialContext object
    InitialContext initialContextOb = new InitialContext(propsJNDI);

    //Create an Properties object
    Properties props = new Properties ();

    //Define the properties
    props.put(FormServerFactory.INITCONTEXT_PROP, initialContextOb);

    //Call create to create an EJBClient object
    IFormServer formServer = FormServerFactory.create(props);
}

catch (Exception ex)
{
    System.out.println(ex.getMessage());
}
```

Note: To use the `FormServerFactory` class in your Java project, add the following import statement:
`import com.adobe.formServer.client.FormServerFactory.`

Invoking the Data Manager Module

You must create a `DataManager` object before you can create applications using the XML Form Module API. This API is dependent on the Data Manager Module API. The Data Manager Module API does not have a constructor that lets you instantiate a `DataManager` object. Instead, you create this object by performing a Java JNDI look-up by using the Connection API.

The Connection API lets you create a connection to the Data Manager service running on a J2EE application server. This API consists of a single class named `ConnectionFactory`. For information about the Data Manager Module API or the Connection API, see the *XML Form Module API Reference*.

In addition to using the Connection API, you also use two Java classes, `javax.naming.InitialContext` and `javax.rmi.PortableRemoteObject`, to perform a Java JNDI look-up. Using these classes, you create a CORBA object representing a connection to the Data Manager service.

The Data Manager Module API is a transaction-based API, which means that a `DataManager` object must be created within a transaction. Using the `javax.transaction.UserTransaction` interface, you start a transaction by invoking its `begin` method and complete the transaction by invoking its `commit` method.

Note: A client application that uses the Data Manager Module API must be deployed to the J2EE application server hosting LiveCycle Forms. You cannot remotely invoke the Data Manager service.

Creating a DataManager object

You create a `DataManager` object by performing the following programmatic tasks within a Java project:

1. Add the necessary JAR files to your Java project's build path. For information, see ["Including LiveCycle Forms library files" on page 28](#).

2. Add the following import statements to your Java project:

```
import com.adobe.service.DataManager;           //Data Manager Module API
import com.adobe.service.DataManagerHelper;    //Data Manager Module API
import com.adobe.service.ConnectionFactory;    //Connection API
import javax.naming.InitialContext;           //InitialContext class
import javax.rmi.PortableRemoteObject;        //PortableRemoteObject
import javax.transaction.UserTransaction;     //UserTransaction class
```

3. Create an `javax.naming.InitialContext` object by using the `InitialContext` constructor:

```
InitialContext namingContext = new InitialContext();
```

4. Perform a JNDI look-up by invoking the `javax.naming.InitialContext` object's `lookup` method and pass the string `DataManagerService` as an argument. Store the return value in an `Object` variable. The following line of code shows this application logic:

```
Object dmObject = namingContext.lookup("DataManagerService");
```

5. Create a `ConnectionFactory` object by invoking the `javax.rmi.PortableRemoteObject` object's `narrow` method. This method determines if the return value of the `lookup` method can be cast to a `ConnectionFactory` object. Cast the return value to `ConnectionFactory`. The following line of code shows this application logic:

```
ConnectionFactory dmConnectionFactory = (ConnectionFactory)
    PortableRemoteObject.narrow(dmObject, ConnectionFactory.class);
```

6. Create a `javax.transaction.UserTransaction` object and invoke its `begin` method. To create this object, invoke the `javax.naming.InitialContext` object's `lookup` method and pass the string `java:comp/UserTransaction` as an argument. Cast the return value to `UserTransaction`. The following lines of code show this application logic:

```
UserTransaction transaction = (UserTransaction)
    namingContext.lookup("java:comp/UserTransaction");
transaction.begin();
```

7. Create a `DataManager` object by calling the `DataManagerHelper` object's `narrow` method (it is unnecessary to instantiate a `DataManagerHelper` object). Pass the `ConnectionFactory` object to this method and call its `getConnection` method. Cast the return value to `org.omg.CORBA.Object`. The following line of code shows this application logic:

```
DataManager mDataManager =  
    DataManagerHelper.narrow((org.omg.CORBA.Object) dmConnectionFactory.  
        getConnection());
```

The following example creates a `DataManager` object.

Example 2.6 Creating a `DataManager` object

```
//Declare a ConnectionFactory object  
ConnectionFactory dmConnectionFactory = null;  
  
//Create an InitialContext object  
InitialContext namingContext = new InitialContext();  
  
// Lookup the Data Manager service  
Object dmObject = namingContext.lookup("DataManagerService");  
dmConnectionFactory = (ConnectionFactory)  
    PortableRemoteObject.narrow(dmObject, ConnectionFactory.class);  
  
//Begin a transaction  
UserTransaction transaction = (UserTransaction)  
    namingContext.lookup("java:comp/UserTransaction");  
transaction.begin();  
  
//Create a DataManager object  
DataManager mDataManager =  
    DataManagerHelper.narrow((org.omg.CORBA.Object) dmConnectionFactory.  
        getConnection());  
  
//Perform tasks using the DataManager object  
  
//Complete the transaction  
transaction.commit();
```

Note: If you create a `DataManager` object outside of a transaction, a Java exception occurs.

Using the `DMUtils` object

The `DMUtils` object enables you to perform programmatic tasks involving data sets, such as reading a data input stream into a temporary file and placing the file into a `DataBuffer` object. All `DMUtils` methods are static.

Before you can use the `DMUtils` object, perform tasks required to create a `DataManager` object, such as adding the Data Manager Jar files to your Java build path and including the necessary import statements. Then, include the following import statement in your Java project:

```
import com.adobe.util.DMUtils;
```

After you perform these tasks, you can use the `DMUtils` object in your Java project. For information about the `DMUtils` class, see the *API Reference*.

Working with the Document object

A `Document` object enables you to pass data between different LiveCycle services and components, and supports remote RMI calls. This object belongs to the `com.adobe.idp` package and is part of the Data Manager Module API. The following two Form Server Module API methods accept a `Document` object as an argument:

- `renderForm`
- `ProcessFormSubmission`

You can, for example, pass a `Document` object to the `renderForm` method that results in the data that is stored in the `Document` object being merged with the form. Also, the `IOutputContext` interface that belongs to the Form Server Module API has a method named `saveOutputContent` that returns a `Document` object.

The following list describes the main features of the `Document` class:

- The `Document` object's content is defined by using `java.io.InputStream`, `com.adobe.service.DataBuffer`, a byte array, a file, or any in-memory object that provides read and write methods.
- The `Document` object is a Java serializable type; therefore, it can be passed over an RMI call (as opposed to `java.io.InputStream`).
- Data is either stored in memory as part of the `Document` object or saved as a file on a disk.
- The `Document` object is cached in memory for the duration of the current J2EE transaction. Therefore, if both the sender and the receiver of the `Document` share the same J2EE transaction, the in-memory object is passed directly with no extra serialization.
- Any temporary storage resources occupied by the `Document` object's content are removed automatically upon the `Document` disposal.

This section discusses how to create a `Document` object and the tasks you can perform by using a `Document` object.

Creating a Document object using a remote file

You can create a `Document` object that is based in a file located on a remote computer. For example, assume that you want to prepopulate a form with data located within an XML file. Next, assume that the XML file is located on a remote server. You create a `Document` object that is based on the remote XML file by using the `Document` constructor that accepts a `java.net.URL` object. For information about populating a form, see ["Rendering prepopulated forms" on page 68](#).

The following code example creates a `Document` object that is based on a remote XML file.

Example 2.7 *Creating a Document object using a remote file*

```
//Create an URL object
URL myURL = new URL("http://<AppServer>:<AppPort>/DataFile/GetForms.xml");

//Create a Document object
Document myRemoteDocument = new Document(myURL);
```

Note: To create a `Document` object, you must specify the following import statement in your Java project

```
import com.adobe.idp.Document.
```

Creating a Document object using a data stream

You can create a `Document` object that is based on a data stream, such as a byte array that is obtained by invoking a `java.io.InputStream` object's `read` method. The following code example creates a `Document` object by using a `Document` constructor that accepts a byte array (for the purpose of this code example, assume that a `java.io.InputStream` object named `myInputStream` exists).

Example 2.8 Creating a Document object using a data stream

```
//Get the size of the InputStream buffer
int avail = myInputStream.available();

//Allocate the size of the InputStream to a byte array
byte[] myBytes = new byte[avail];

//Populate the byte array with the contents of the InputStream
myInputStream.read(myBytes);

//Create a Document object using the byte array
Document myDataStreamDocument = new Document(myBytes);
```

Note: You can also create a `Document` object by passing a `java.io.InputStream` object to the `Document` constructor.

Creating a Document object using a local file

You can create a `Document` object that is based on a local file by using the `Document` constructor that accepts a `java.io.File` object. When you create a `Document` object, it keeps a reference to the `java.io.File` object. The local file should remain available until the contents of the `Document` object is saved to a permanent storage.

A `Document` object's contents can be saved to a permanent storage when the `Document` object's `passivate` method is invoked or when the `Document` object's content is read by invoking an access method such as `getInputStream` or `getFile`.

The `Document` constructor that accepts a `java.io.File` object also requires a boolean value that specifies whether the `Document` object controls the file when the contents are saved to a permanent storage.

The following code example creates a `Document` object that is based on a local file named `FormData.xml`. The boolean value is set to `false`, which means that the `Document` object does not control the file.

Example 2.9 Creating a Document object using a local file

```
//Create a File object based on a local file named FormData.xml
File myFile = new File("C:\\\\FormData.xml");

//Create a Document object based on a local file
Document myLocalDocument = new Document(myFile, false);
```

Returning the content of a Document object to a file

You can return the contents of a `Document` object to a file by invoking the `Document` object's `getFile` method. The following code example returns the content of the `Document` object to a `java.io.File` object.

Example 2.10 Returning the content of a Document object to a file

```
File docContentFile = myDocument.getFile();
```

Writing the content of a Document object to a data stream

You can write the contents of a `Document` object to a `java.io.InputStream` object by invoking the `Document` object's `getInputStream` method. The following code example writes the content of the `Document` object to a `java.io.InputStream` object.

Example 2.11 Writing the content of a Document object to a data stream

```
InputStream myDataStream = myDoc.getInputStream();
```

Invoking the XML Form service

You use the XML Form Module API to create a `Form` object. Using this object, you can perform tasks such as rendering forms. For information, see ["Rendering Forms using the XML Form Module API" on page 121](#).

Caution: The XML Form Module API is deprecated; it is recommended that you use the Form Server Module API.

In addition to using the Connection API, you also use two standard Java classes, `InitialContext` and `PortableRemoteObject`, to perform a Java JNDI look-up. Using these classes, you create a CORBA object representing a connection to the XML Form service.

The XML Form Module API is a transaction-based API, which means a `Form` object must be created within a transaction. Using the `UserTransaction` class, you can create a `UserTransaction` object. Call the `UserTransaction` object's `begin` method to start the transaction and its `commit` method to complete the transaction.

Note: A client application that uses the XML Form Module API must be deployed to the J2EE application server hosting LiveCycle Forms.

Creating a Form object

You create a `Form` object by performing the following programmatic tasks within a Java project:

1. Add the `xmlform-client.jar` file to your Java project's build path. For information about the location of this file, see ["Including LiveCycle Forms library files" on page 28](#).
2. Add the following import statements to your Java project:

```
import com.adobe.document.xmlform.Form;  
import com.adobe.document.xmlform.FormFactory;  
import com.adobe.document.xmlform.FormFactoryHelper;
```

3. Create a `DataManager` object (the import statements required to create this object are required to create a Form object). For information, see [“Invoking the Data Manager Module” on page 40](#).

4. Create an `InitialContext` object by using the `InitialContext` constructor:

```
InitialContext xmlNamingContext = new InitialContext();
```

5. Perform a JNDI look-up by calling the `InitialContext` object's `lookup` method and pass the string `XMLFormService` as an argument. Store the return value in an `Object` variable. The following line of code shows this application logic:

```
Object xmlObject = xmlNamingContext.lookup("XMLFormService");
```

6. Create a `ConnectionFactory` object by calling the `PortableRemoteObject` object's `narrow` method. This method determines if the return value of the `lookup` method can be cast to a `ConnectionFactory` object. Cast the return value to `ConnectionFactory`. The following line of code shows this application logic:

```
ConnectionFactory xmlConnectionFactory = (ConnectionFactory)  
PortableRemoteObject.narrow(xmlObject, ConnectionFactory.class);
```

7. Create a `UserTransaction` object and call its `begin` method. To create a `UserTransaction` object, call the `InitialContext` object's `lookup` method and pass the string `java:comp/UserTransaction` as an argument. Cast the return value to `UserTransaction`. The following lines of code show this application logic:

```
UserTransaction transaction = (UserTransaction)  
namingContext.lookup("java:comp/UserTransaction");  
transaction.begin();
```

8. Create a `FormFactory` object by calling the `FormFactoryHelper` object's `narrow` method (it is unnecessary to instantiate a `FormFactoryHelper` object). Pass the `ConnectionFactory` object to this method and call its `getConnection` method. Cast the return value to `org.omg.CORBA.Object`. The following line of code shows this application logic:

```
FormFactory mFormFactory =  
FormFactoryHelper.narrow((org.omg.CORBA.Object)xmlConnection  
Factory.getConnection());
```

9. Create a Form object by calling the `FormFactory` object's `createDefault` method. This method creates a Form object by using default configuration values. The following line of code show this application logic:

```
Form myForm = mFormFactory.createDefault();
```

The following code example shows how to create a Form object.

Example 2.12 Creating a Form object using createDefault

```
//Declare a ConnectionFactory object
ConnectionFactory xmlConnectionFactory = null;

// Lookup the XMLForm service
Object xmlObject = namingContext.lookup("XMLFormService");

//Create a ConnectionFactory object
xmlConnectionFactory = (ConnectionFactory)
PortableRemoteObject.narrow(xmlObject,ConnectionFactory.class);

//Begin a transaction
UserTransaction transaction = (UserTransaction)
    namingContext.lookup("java:comp/UserTransaction");
transaction.begin();

//Use the xmlConnectionFactory object to create a FormFactory object
FormFactory mFormFactory =
    FormFactoryHelper.narrow((org.omg.CORBA.Object)xmlConnection
    Factory.getConnection());

// Create a Form object by calling createDefault
Form myForm = mFormFactory.createDefault();

//Perform tasks using the Form object

//Complete the transaction
transaction.commit();
```

Note: This example shows calling the FormFactory object's createDefault method to create a Form object that uses default configuration settings. For information about creating a Form object by calling the FormFactory object's create method, setting configuration options, and then using the Form object to render a form, see ["Rendering Forms using the XML Form Module API" on page 121](#).

3

Rendering Interactive Forms as PDF

You use the Form Server Module API to develop client applications that interact with the Form Server Module. The Form Server Module renders forms, such as an interactive form, that are displayed as either PDF or HTML across the Internet or an intranet to client devices, typically web browsers. An interactive form contains one or more fields for collecting information interactively from a user. This chapter explains how to render interactive forms displayed as PDF. For information about rendering forms as HTML, see [“Rendering Forms as HTML” on page 91](#).

A form is rendered in response to a request made by a client device. For example, a client application can instruct the Form Server Module to render a form in response to an HTTP request that is initialized from a client web browser. When the form is rendered back to the client web browser, the user can fill in the form and click a Submit button to send information back to the Form Server Module.

The code examples displayed in this chapter show how to locally invoke the Form Server Module using an `EJBClient` object. There are other ways in which to invoke the Form Server Module. For information, see [“Invoking the Form Server Module” on page 30](#).

This chapter contains the following information.

Topic	Description	See
About rendering PDF forms	Describes the characteristics of a form-based application.	page 48
Rendering PDF forms	Describes the methods you can use to render a form to a client web browser.	page 52
Retrieving submitted data	Describes the methods you can use to retrieve data submitted from a form.	page 60
Rendering prepopulated forms	Describes the methods you can use to prepopulate a form prior to rendering it.	page 68
Rendering a form at the client	Describes how to render a form at the client.	page 74
Passing a form design by value	Describes how to pass a form design by value.	page 77

About rendering PDF forms

The Form Server Module is stateless and runs on a J2EE application server and is accessible through the Form Server Module API. A client application that uses the Form Server Module API is able to invoke the Form Server Module and instruct it to perform tasks such as rendering forms, processing submitted data, and prepopulating forms with data.

The Form Server Module sends forms across a network and renders them to client devices, such as web browsers. Although a form can be rendered on different types of client devices, the examples in this chapter assume that forms are rendered on web browsers.

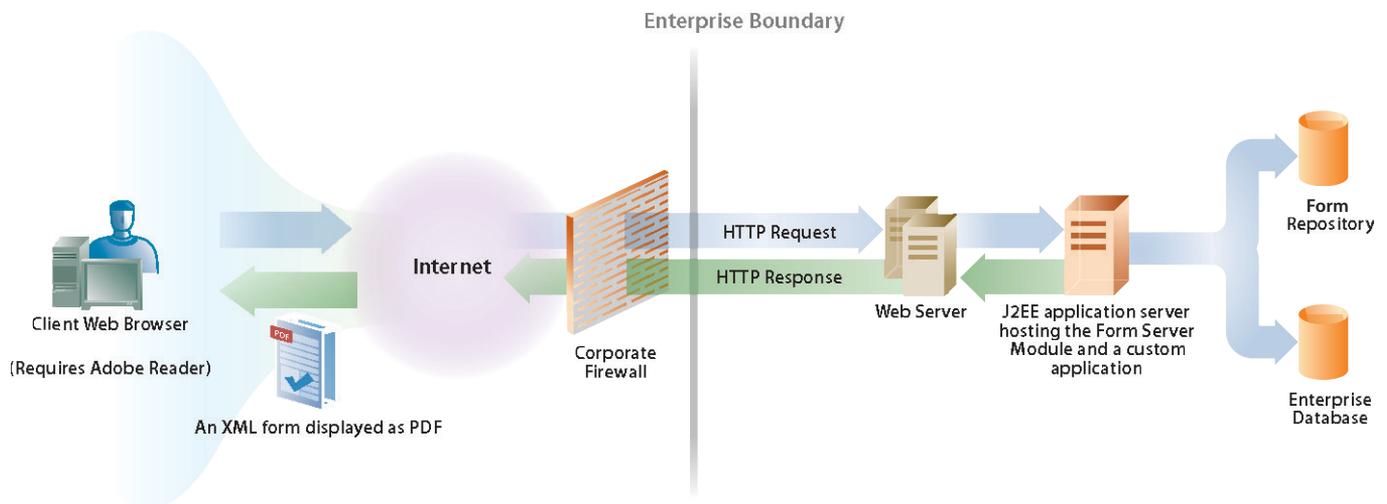
A client application that uses the Form Server Module API is able to retrieve the data submitted with a form. For example, when a user fills in a form and submits it, a client application can retrieve the data that

the user entered in the form's fields. The client application can then process the data in a variety of ways, such as performing calculations, storing it in an enterprise database, or sending it to another application, such as an application that authorizes credit cards.

The Form Server Module can prepopulate a form prior to rendering it. Prepopulating a form involves inserting data into a form. For example, a client application can query data from a database and instruct the Form Server Module to insert the data into a form and then render the form. Once the form is rendered to a client web browser, the user is able to view the data in the displayed form.

Using the Form Server Module API, you can create different types of client applications that interact with the Form Server Module, such as Java servlets or JSPs. This chapter discusses creating Java servlets that can be deployed on the J2EE application server on which the Form Server Module is deployed. For information about deploying the Form Server Module on a J2EE application server, see the *Installing and Configuring* guide for your application server.

Consider a client web browser sending an HTTP request to a client application requesting a form that is displayed as PDF. When the client application receives the HTTP request, it sends the request to the Form Server Module, which then renders the form to the client web browser within an HTTP response. Adobe Reader (or Acrobat) must be installed on the computer hosting the client web browser for a form that is displayed as PDF to be visible to a user, as shown in the following diagram.



Note: This diagram shows a Form Server Module API client application and the Form Server Module existing on the same J2EE application server. You can deploy a client application to a separate J2EE application server and remotely invoke the Form Server Module. For information, see [“Remotely invoking Form Server Module” on page 32](#).

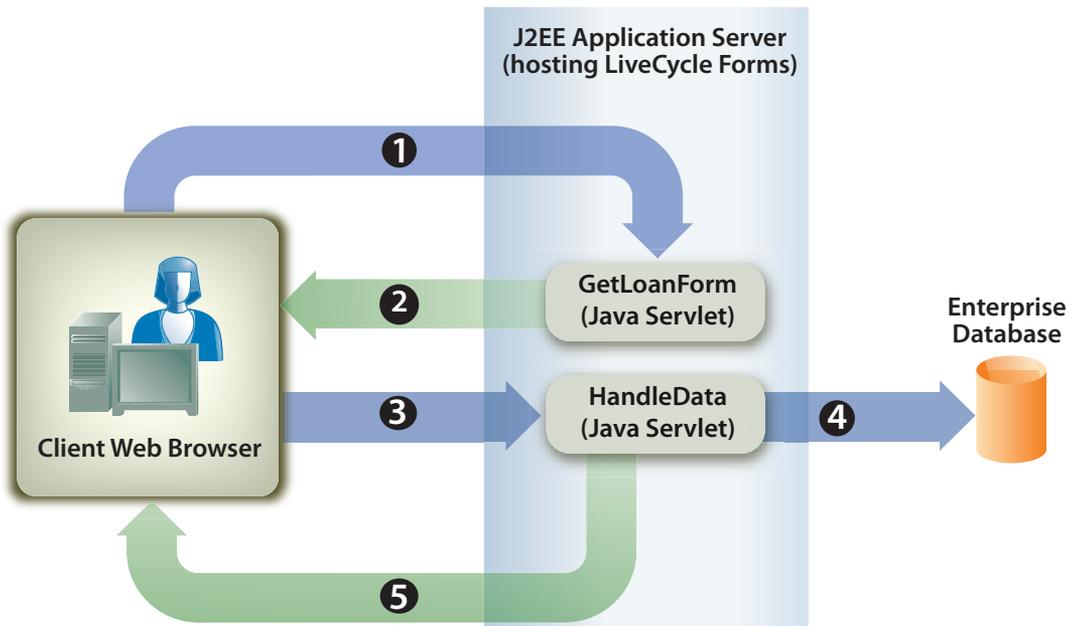
Sample loan application

A sample loan application is introduced to enhance your understanding of the Form Server Module. This application lets a user fill in a form with data required to secure a loan. After the user fills in the form, the data is submitted to the Form Server Module by clicking a Submit button. This sample application consists of the following components:

- An HTML page named StartLoan.html that functions as the application's start page. This page invokes a Java servlet named GetLoanForm.
- A Java servlet named GetLoanForm (contains the Form Server Module API) that renders a loan form that is saved as Loan.xdp.

- A loan form that lets a user fill in data required to secure a loan. This form contains a Submit button that the user clicks to submit data to the HandleData Java servlet.
- A Java servlet named HandleData (contains the Form Server Module API) that retrieves the submitted data.
- A confirmation form that displays a confirmation message.

The following diagram shows the loan application's logic flow.



The following table describes the steps in this diagram.

- | | |
|---|---|
| 1 | The <code>GetLoanForm</code> Java servlet is invoked from the <code>StartLoan.html</code> page. This page contains a link that invokes the <code>GetLoanForm</code> Java servlet. |
| 2 | The <code>GetLoanForm</code> Java servlet uses the Form Server Module API to render the loan form to the client web browser. For information, see “Rendering a form using an EJBClient object” on page 52. |
| 3 | After the user fills in the loan form and clicks the Submit button, data is submitted to the <code>HandleData</code> Java servlet. For information about the loan form, see “Loan form” on page 51. |
| 4 | The <code>HandleData</code> Java servlet uses the Form Server Module API to process the form submission and retrieve field data. The data is then stored in an enterprise database. For information, see “Retrieving submitted form data” on page 60. |
| 5 | A confirmation form is rendered back to the web browser. Data such as the user’s first and last name is merged with the form before it is rendered. For information about merging data with a form, see “Rendering prepopulated forms” on page 68. |

Loan form

This interactive loan form is rendered by the sample loan application's GetLoanForm Java servlet:

Loan Application	
Amount requested	
Last Name	Mailing Address
First Name	Address _____
Social Security Number	City _____
Position Title	State/Province _____
Phone Number (Area code, number and extension)	Zip/Postal Code _____
	Email Address
	Fax Number (Area code, number)
Project Description	
State the project objectives and specific methods for achieving these goals.	

Submit

Note: For information about an interactive form, see ["Interactive forms" on page 13](#).

Confirmation form

This static confirmation form is rendered by the sample loan application's HandleData Java servlet:

Loan Confirmation	
First Name	<input type="text"/>
Last Name	<input type="text"/>
Amount	<input type="text"/>
Thank you for your loan application. You will be contacted shortly by phone to inform you whether your loan application was approved. We look forward to doing business with you.	

The `HandleData` Java servlet prepopulates this form with the user's first and last name as well as the amount. After the form is prepopulated, it is sent to the client web browser. For information, see ["Rendering prepopulated forms" on page 68](#).

Note: For information about a static form, see ["Static forms" on page 14](#).

Java servlets

The sample loan application is an example of a Form Server application that exists as Java servlets. A Java servlet is a Java program running on a J2EE application server, such as WebSphere, and contains Form Server Module API code.

The following code shows the syntax of a Java servlet named `GetLoanForm`:

```
public class GetLoanForm extends HttpServlet implements Servlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp
        throws ServletException, IOException {
    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp
        throws ServletException, IOException {
        //Throughout this chapter Form Server code is placed here
    }
}
```

Normally, you would not place Form Server Module API code within a Java servlet's `doGet` or `doPost` methods. It is better programming practice to place Form Server Module API code within a separate class, instantiate the new class from within the `doPost` method (or `doGet` method), and call the appropriate methods. However, for code brevity, Form Server Module API code is placed in the `doPost` method.

Rendering a form using an EJBClient object

Using the Form Server Module API, you create application logic to render a form that is displayed as PDF to a client web browser as part of an HTTP response. A form must be rendered before a user can interact with it. Consider the loan application described earlier in this chapter. Before a user can fill in the loan form, it must be rendered and displayed in a web browser. For information about this form, see ["Loan form" on page 51](#).

To render a form, create an `EJBClient` object (or a `SOAPClient` object) and call its `renderForm` method. The `renderForm` method returns an `IOutputContext` interface. You use this interface to populate a data stream to send to the client web browser with the form.

Note: Before you render a form, it is recommended that you are familiar with invoking the Form Server Module. For information, see ["Invoking the Form Server Module" on page 30](#).

Specifying the form design to render

You set the `renderForm` method's `sFormQuery` parameter to specify the form design to render. Assign the file name of the form design to this parameter, as shown in the following example:

```
String sFormQuery = "Loan.xdp";
```

You set the `renderForm` method's `sContentRootURI` parameter to specify the network location of the form design. Typically, this is a URI value, as shown in the following example:

```
String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
```

AppServer represents the host name of the computer on which your client application is deployed. *AppPort* represents the port that the application uses. For example, an application running on the WebSphere J2EE application server uses port 9080.

Note: In this code example, a form design is accessed using HTTP. However, a form can be referenced using a URI referencing a network path.

Passing a zero-length byte array

You must ensure that you do not pass a null value to the `renderForm` method's `cData` parameter. When using 7.0 library files (or previous library files), you could pass null to this parameter. However, when using the Form Server Module 7.2 API, a `com.adobe.idp.Document` object can be passed to the `cData` parameter in addition to a byte array. As a result, a null value can no longer be passed to `cData`.

If you do not want to merge data with a form prior to rendering it, then instead of passing a null value to `cData`, you can pass a zero-length byte array:

```
byte[] cFormData = new byte[0];
```

The `cData` parameter is used to prepopulate forms. For information, see ["Rendering prepopulated forms" on page 68](#).

Setting preference options to render the form as PDF

You can render a form design that was created in LiveCycle Designer and saved as a PDF or an XDP file. Set the `renderForm` method's `sFormPreference` parameter to one of the following values:

PDFForm: Renders the form as an interactive PDF document.

PDF: Renders the forms as a non-interactive PDF document.

PDFMerge: Merges data into a prerendered PDF form.

These options are used to render the form as PDF. To view the `renderForm` method's `sFormPreference` values that are used to render the form as HTML, see ["Setting preference options to render the form as HTML" on page 92](#).

Specifying the web context of a client application

Set the `renderForm` method's `sApplicationWebRoot` with a string value that specifies the web context of the client application that uses the Form Server Module API. This value is combined with the `sTargetURL` parameter to construct an absolute URL to access application-specific web content. Typically, this is an URI value, as shown in the following example:

```
String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
```

Specifying the target URL

Set the `renderForm` method's `sTargetURL` parameter to specify the target URL to where form data is posted. Typically, this parameter references a Java servlet located on the J2EE application server on which the client application is deployed. For example, in the sample loan application, data is posted to the `HandleData` Java servlet. Usually, this value is an URL value, as shown in the following example:

```
String sTargetURL = "http://<AppServer>:<AppPort>/LoanApp/HandleData"
```

When the form is rendered to a client web browser, the user fills in the form and clicks the Submit button. Data is posted to the URL specified by the `sTargetURL` parameter. If you specify an invalid value, a `RenderFormException` is thrown. For more information about this parameter, see the *Form Server Module API Reference*.

A Submit button on a form design may have a URL that specifies the location to where data is posted (the URL value is specified in the Submit to URL text box). If these two values conflict, the URL in the form design overrides the value of `sTargetURL`.

If you have a form that contains a Submit button and a calculate button (with a corresponding script that runs at the server), you must use `sTargetURL` to specify the location where the form is sent to calculate the script. Use the Submit button on a form design to specify the URL to where data is posted. For information, see ["Calculating Form Data" on page 95](#).

Specifying the PDF version

You can use the Form Server Module API to specify the version of the PDF file to render. You can render a form as PDF version 1.5 or 1.6. By default, a form that is rendered as PDF is version 1.6. To change the PDF version to 1.5, assign the `renderForm` method's `sOptions` parameter with the following string value:

```
PDFVersion=1.5
```

Caching PDF forms

You can use the Form Server Module API to cache a form that is rendered as PDF in the server cache. Each form is cached after it is generated for the first time. On a subsequent render, if the cached form is newer than the form design's timestamp, the form is retrieved from the cache. By caching forms, you improve the performance of the Form Server Module because it does not have to retrieve the form design from a repository and re-render the form.

To cache a form in the server cache, assign the `renderForm` method's `sOptions` parameter with the following string value:

```
CacheEnabled=True
```

Form designs that have the same `FormQuery` (an argument for the `renderForm` method) but are stored in different `ContentRootURI` (another argument for the `renderForm` method) locations are not cached in the same cache location. Each `ContentRootURI` location has a unique cache location so that form designs from different URI locations are not overwritten.

In addition to using the Form Server Module API, you must also enable form caching in LiveCycle Designer. For information see, the *LiveCycle Designer Help*.

Caching PDF forms in the client web browser

You can use the Form Server Module API to cache a PDF form in the client web browser cache. Only forms that are rendered as interactive forms can be stored in the client web browser cache. That is, the value of the `renderForm` method's `sFormPreference` parameter must be `PDFForm`. For information, see [“Setting preference options to render the form as PDF” on page 53](#).

When a client web browser makes a request to the Form Server Module for a PDF, the Form Server Module generates the PDF and stores it in the server cache. The PDF form is also cached by the browser after it is sent to the client web browser.

When subsequent requests for the PDF form are made, a time stamp located in the PDF form that is stored in the client web browser cache is compared with the time stamp of the PDF form that is generated by the Form Server Module and stored in the server cache. If they are the same, the PDF form is retrieved from the client cache. This results in reduced bandwidth usage and improved performance because the Form Server Module does not have to redeliver the same content to the client web browser.

To cache a form in the client web browser cache, assign the `renderForm` method's `sOptions` parameter with the following string value:

```
clientCache=True
```

Note: The default value for `clientCache` is `false`.

Accessing LiveCycle Form Manager application store

If your deployment of LiveCycle Forms requires the Form Server Module to access the application store that accompanies LiveCycle Form Manager to retrieve resources, such as a form design, then you must create a Java `Context` object that is required to access the application store. Without the Java `Context` object, the Form Server Module will not be able to access the application store and retrieve the necessary resources.

After you create the necessary `Context` object, you must call the `IFormServer` interface's `setInvocationContext` method and pass the `Context` object. You must call the `setInvocationContext` method before you call the `renderForm` method. For more information about the `setInvocationContext` method, see the *Form Server Module API Reference*.

Note: For information about authenticating users with User Manager, see [“Authenticating Users” on page 110](#).

Setting the Standalone option

If your business requirements do not require the Form Server Module to perform server-side calculations, you can set the Standalone option to `true`, which results in forms being rendered without state information. This setting improves the performance of the Form Server Module.

State information is necessary if you want to render an interactive form to an end user who then enters information into the form and submits the form back to the Form Server Module. The Form Server Module then performs a calculation operation and renders the form back to the user with the results displayed in the form.

If a form without state information is submitted back to the Form Server Module, only the XML data is available. That is, you can invoke the `processFormSubmission` method to retrieve XML data from the form; however, server-side calculations are not performed. For information, see [“Retrieving submitted form data” on page 60](#).

To render a form without state information, assign the `renderForm` method's `sOptions` parameter with the following string value:

```
standAlone=true
```

Note: The default value for this option is `false`.

Setting XCI run-time options

An XCI file is a configuration file that the Form Server Module uses. By default, Form Server Module uses a XCI file named `pa.xci` which is located in the `formServerEJB.jar` file (this file is located within the `LiveCycle.ear` file).

You can use the Form Server Module API to set XCI run-time options that are executed when the `renderForm` or `processFormSubmission` methods are invoked. By setting XCI run-time options, you can change configuration values. For example, you can change the font that is displayed within a rendered form. The ability to set XCI run-time options is optional.

The Form Server Module currently uses a default XCI configuration file named `pa.xci`. This default XCI is parsed as an XML DOM on each request (each time `processFormSubmission` or `renderForm` is invoked), and the XCI is updated based on the Form Server Module API options that are specified.

You define XCI run-time options by assigning the `renderForm` (or `processFormSubmission`) method's `sOptions` parameter with the following string value:

```
XCI=<path-expression>=<value>;<path-expression>=<value>;..<path-expression>=<value>
```

where `<path-expression>` is an x-path expression that is relative to the configuration root element representing the element to update. `<value>` is the value to assign to the path expression. If the value is omitted, the element is cleared.

The following example changes the default typeface to `MyriadPro`:

```
XCI=present/pdf/fontInfo/defaultTypeface=MyriadPro
```

An XCI option consists of a series of XCI updates that are delimited by a semicolon. Each update specifies an XCI path that is updated and followed by a value to be set. If the path location does not exist, it is created. If the path location does exist, it is updated.

Only one XCI option is supported in the `sOption` parameter. If more than one XCI option is specified, the first one is considered and additional ones are ignored. However, as stated, a single XCI option can consist of multiple XCI updates. For a complete list of all XCI options that can be set by using the Form Server Module API, see the *Form Server Module API Reference*.

Note: An `@` character can be used to set an XCI attribute. Support for x-path is limited and only `/` and `@` characters are supported.

Creating application logic to render a form as PDF

The following process describes how to render a form to a client web browser by locally invoking the Form Server Module:

1. Create an `EJBClient` object by calling the `EJBClient` constructor. For information, see [“Locally invoking Form Server Module” on page 31](#).
2. Call the `EJBClient` object’s `renderForm` method and set its parameters (this is shown in the example that follows this numbered list). This method returns an `IOutputContext` interface. (You are able to get the `IOutputContext` interface’s content type and its byte stream. A typical usage involves using Java classes to send the byte stream to the client web browser. This is shown in the example that follows this list).
3. Create a Java `ServletOutputStream` object used to send a byte stream to the client web browser.
4. Set the Java `HttpServletResponse` object’s content type to match the `IOutputContext` object’s content value. You can achieve this by calling the `HttpServletResponse` object’s `setContentType` method and passing the `IOutputContext` object’s `getContentType` return value.
5. Create a byte array and populate it by calling the `IOutputContext` object’s `getOutputContent` method. This task assigns the content of the `IOutputContext` object to a byte array.
6. Call the `HttpServletResponse` object’s `write` method to send a data stream to the client web browser. Pass the byte array to the `write` method.

The following code example renders a form named `Loan.xdp` to a client web browser.

Example 3.1 *Rendering a form to a client web browser using an EJBClient object*

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {

    //Create a EJBClient object
    EJBClient formServer = new EJBClient();

    //Declare and populate local variables to pass to renderForm
    String sFormQuery = "Loan.xdp";           //Defines the form design to render
    String sFormPreference = "PDFForm";      //Defines the preference option
    String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
    String sTargetURL = "http://<AppServer>:<AppPort>/LoanApp/HandleData";
    String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
    byte[] cData = new byte[0]; //cData

    try{
        //Call renderForm
        IOutputContext myOutputContext = formServer.renderForm(
            sFormQuery,           //sFormQuery
            sFormPreference,     //sFormPreference
            cData,               //cData,
            "CacheEnabled=False", //sOptions
            null,                //sUserAgent,
            sApplicationWebRoot, //sApplicationWebRoot
            sTargetURL,          //sTargetURL
            sContentRootURI,     //sContentRootURI
```

```
        null                //sBaseURL
    );

    // Create a ServletOutputStream object
    ServletOutputStream oOutput = resp.getOutputStream();

    //Set the HTTPResponse object's content type
    resp.setContentType(myOutputContext.getContentType());

    // Get the length of the output stream
    int outLength = myOutputContext.getOutputContent().length;

    //Create a byte array and allocate outLength bytes
    byte[] cContent = new byte[outLength];

    //Populate the byte array by invoking getOutputContext
    cContent = myOutputContext.getOutputContent();

    //Write a byte stream back to the web browser. Pass the byte array
    oOutput.write(cContent);
}
//Catch a thrown exception
catch (Exception ioEx)
{
    System.out.println("Exception error is: " +ioEx.getMessage());
}
}
```

Note: This example shows how to render a form by locally invoking the Form Server Module. You can use an `EJBClient` object to remotely invoke the Form Server Module. For information, see [“Remotely invoking Form Server Module” on page 32](#).

Rendering a Form using a SOAPClient object

You can use a `SOAPClient` object to render a form as PDF (or as HTML) to a client web browser as part of an HTTP response. You must set the soap endpoint to successfully render a form. For information, see [“Invoking Form Server Module using SOAP” on page 34](#).

Other than setting a soap endpoint, the application logic to render a form is the same as using an `EJBClient` object. For information, see [“Rendering a form using an EJBClient object” on page 52](#).

Example 3.2 Rendering a form to a client web browser using a SOAPClient object

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {

    //Create a SOAPClient object
    SOAPClient formServer = new SOAPClient();

    //Set the soap end point
    formServer.setSoapEndPoint("http://<AppServerURL>:8080/jboss_net/services/
AdobeFSService");

    //Declare and populate local variables to pass to renderForm
```

```
String sFormQuery = "Loan.xdp";           //Defines the form design to render
String sFormPreference = "PDFForm";       //Defines the preference option
String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
String sTargetURL = "http://<AppServer>:<AppPort>/LoanApp/HandleData";
String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
byte[] cData = new byte[0]; //cData

try{
    //Call renderForm
    IOutputContext myOutputContext = formServer.renderForm(
        sFormQuery,           //sFormQuery
        sFormPreference,     //sFormPreference
        cData,               //cData
        "CacheEnabled=False", //sOptions
        null,                //sUserAgent,
        sApplicationWebRoot, //sApplicationWebRoot
        sTargetURL,         //sTargetURL
        sContentRootURI,   //sContentRootURI
        null,               //sBaseUrl
    );

    // Create a ServletOutputStream object
    ServletOutputStream oOutput = resp.getOutputStream();

    //Set the HTTPResponse object's content type
    resp.setContentType(myOutputContext.getContentType());

    // Get the length of the output stream
    int outLength = myOutputContext.getOutputContent().length;

    //Create a byte array and allocate outLength bytes
    byte[] cContent = new byte[outLength];

    //Populate the byte array by invoking getOutputContext
    cContent = myOutputContext.getOutputContent();

    //Write a byte stream back to the web browser. Pass the byte array
    oOutput.write(cContent);
} //end of try
//Catch a thrown exception
catch (Exception ioEx)
{
    System.out.println("Exception error is: " +ioEx.getMessage());
}
}
```

Note: To successfully invoke the Form Server Module using a SOAPClient object, you must place additional JAR files into your application's class path. For information, see ["Including LiveCycle Forms library files" on page 28.](#)

Retrieving submitted form data

Most web applications require a user to fill in an online form and submit the data. After the client application retrieves data, it can process it in a variety of ways, such as storing it in a database, sending it to another application, merging it with another form, or displaying it in a web browser. What you do with the data depends on your business requirements.

Consider the loan application introduced earlier in this chapter. After the user fills in the loan form and clicks the Submit button, data is submitted to the HandleData Java servlet. For information about this application, see [“Sample loan application” on page 49](#).

The following diagram shows data being submitted to the HandleData Java servlet from an interactive form displayed in a web browser.



The following table explains the steps in the diagram.

- | | |
|---|--|
| 1 | A user fills in an interactive form and clicks the form’s Submit button. |
| 2 | Data is submitted to the HandleData Java servlet as XML data. |
| 3 | The HandleData Java servlet contains application logic to retrieve the data. |

Form design considerations

When data is submitted from a client web browser to LiveCycle Forms, it can be submitted as either XML or PDF data. To retrieve the data that is entered into form fields, the data must be submitted as XML data. If data is submitted as PDF, you cannot retrieve individual field values. The content type of data that is submitted as XML is `text/xml`. In contrast, the content type of data that is submitted as PDF is `application/pdf`. The content type of data submitted as XDP is `application/vnd.adobe.xdp+xml`.

The form design must be configured correctly in LiveCycle Designer for data to be submitted as XML data. To properly configure the form design to submit XML data, ensure that the Submit button that is located on the form design is set to submit XML data. For information about setting the Submit button to submit XML data, see the *LiveCycle Designer Help*.

Note: There are use cases to submit data from a client web browser to LiveCycle Forms as PDF data. For information, see [“Transferring PDF Data” on page 103](#).

Relationship between form fields and XML data

When form data is submitted as XML, you can use the Form Server Module API to retrieve XML data that was submitted. All form fields appear as nodes in an XML document. The node values correspond to the values that the user filled in. Consider the loan form introduced earlier in this chapter. Each field in this form appears as a node within the XML data. The value of each node corresponds to the value that a user fills in. Assume a user fills in the loan form with data shown in the following form:

Loan Application	
Amount requested 250000	
Last Name Johnson	Mailing Address
First Name Jerry	Address <u>120 NoWhere Dr.</u>
Social Security Number 999-999-999	City <u>New York</u>
Position Title Computer Engineer	State/Province <u>NY</u>
	Zip/Postal Code <u>5555</u>
Phone Number (Area code, number and extension) (555) 555-5959	Email Address JJohnson@NoMailServer.com
	Fax Number (Area code, number) (555) 555-5960
Project Description State the project objectives and specific methods for achieving these goals.	
This loan will be used to purchase my first home.	

The following example shows a section of XML data that is retrieved by using the Form Server Module API.

The fields in the loan form. These values can be retrieved using Java XML classes, such as those located in the org.w3c.dom package.

```

- <form1>
  - <grantApplication>
    - <page1>
      <Amount>250000</Amount>
      <LastName>Johnson</LastName>
      <FirstName>Jerry</FirstName>
      <SSN>999-999-999</SSN>
      <PositionTitle>Computer Engineer</PositionTitle>
      <Address>120 NoWhere Dr.</Address>
      <City>New York</City>
      <StateProv>NY</StateProv>
      <ZipCode>5555</ZipCode>
      <Email>FJones@NoMailServer.com</Email>
      <PhoneNum>(555) 555-5959</PhoneNum>
      <FaxNum>(555) 555-5960</FaxNum>
      <Description>This loan will be used to purchase my first home.</Description>
    </page1>
  </grantApplication>
</form1>

```

Creating application logic to retrieve submitted data

The Form Server Module API contains methods that you can use to retrieve submitted form data. The `processFormSubmission` method must be invoked to retrieve submitted data. The `processFormSubmission` method is overloaded. The first version of `processFormSubmission` requires the following arguments:

- A `javax.servlet.http.HttpServletRequest` object.
- A string value that represents run-time options. For information, see the *Form Server Module API Reference*.

This version of `ProcessFormSubmission` is used in the code example located in this section. For information, see [“Retrieving submitted form data” on page 64](#).

The other version of `ProcessFormSubmission` requires the following arguments:

- A byte array or a `Document` object that contains submitted data. You can create a `Document` object by invoking the `javax.servlet.http.HttpServletRequest` object’s `getInputStream` method within the `Document` constructor:

```
Document myDocument = new Document (req.getInputStream());
```

For information about a `Document` object, see [“Working with the Document object” on page 43](#).

- A string value that specifies the CGI environment variables.
- A string value that specifies the HTTP header `User-Agent` that provides information about the target device.
- A string value that represents run-time options.

The `ProcessFormSubmission` method returns an `IOutputContext` interface that you can use to retrieve the submitted data. Before you retrieve submitted data, it is recommended that you determine whether the Form Server Module is finished processing the data.

When a client web browser submits a form, this does not necessarily mean that the Form Server Module is finished processing the data. With each submission, the data must be passed to the `processFormSubmission` method, which returns an `IOutputContext` interface.

To determine if the Form Server Module is finished processing the data, call the `IOutputContext` interface’s `getFSAction` method. This method returns one of the following values:

- 0 (Submit)—Validated XML data is ready to be processed.
- 1 (Calculate)—Calculation results must be written to the client application. For information, see [“Calculating Form Data” on page 95](#).
- 2 (Validate)—Calculations and validations must be written to the client application.
- 3 (Next)—The current page has changed with results that must be written to the client application.
- 4 (Previous)—The current page has changed with results that must be written to the client application.

If the `getFSAction` method returns the value 0, you can retrieve the submitted data. To retrieve submitted data, you convert the `IOutputContext` interface's content to an XML data source. After you perform this task, you can retrieve data from the XML data source by using the following Java Document Object Model (DOM) classes:

- `DocumentBuilderFactory`: Used to create a `DocumentBuilder` object.
- `DocumentBuilder`: Used to create a `Document` object.
- `Document`: Used to create an object that represents an entire XML document.
- `NodeList`: Used to create an object that represents a collection of nodes within an XML document.
- `Node`: Used to create an object that represents a single node within an XML document.

To use these classes in your Java project, add the following import statements:

- `import org.w3c.dom.Document;`
- `import org.w3c.dom.NodeList;`
- `import org.w3c.dom.Node;`
- `import org.w3c.dom.Element;`
- `import javax.xml.parsers.*;`

The following process describes how to create application logic to retrieve data from a form:

1. Create an `EJBClient` object. For information, see ["Invoking the Form Server Module" on page 30](#).
2. Call the `EJBClient` object's `processFormSubmission` method. This method returns an `IOutputContext` interface that contains the data submitted from the form. For information about the `processFormSubmission` method, see the *Form Server Module API Reference*.
3. Create a byte array by calling the `IOutputContext` interface's `getOutputContent` method.
4. Create an `InputStream` object by calling the `ByteArrayInputStream` constructor and passing the byte array.
5. Create a `DocumentBuilderFactory` object by calling the static `DocumentBuilderFactory` object's `newInstance` method.
6. Create a `DocumentBuilder` object by calling the `DocumentBuilderFactory` object's `newDocumentBuilder` method.
7. Create a `Document` object by calling the `DocumentBuilder` object's `parse` method and passing the `InputStream` object.
8. Retrieve the value of each node within the XML document. One way to accomplish this is to create a custom method that accepts two parameters: the `Document` object and the name of the node whose value you want to retrieve. This method returns a string value representing the value of the node. In the code example that follows this process, this custom method is called `getNodeText`. The body of this method is shown.
9. Calls the `getNodeText` method for each field from which to retrieve a value. For example, to retrieve all fields in the loan form, you must call `getNodeText` 13 times.

The following code example shows application logic that retrieves data submitted from a form.

Example 3.3 Retrieving submitted form data

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {

    // Create an EJBClient object and an IOutputContext interface
    EJBClient formServer = new EJBClient();
    IOutputContext myOutputContext = null;

    try{

        // Call processFormSubmission to handle the submitted data. Pass the
        // HttpServletRequest object
        myOutputContext = formServer.processFormSubmission(req, "OutputType=0");

        //Determine the processing state associated with the submitted form
        short fsAction = myOutputContext.getFSAction();
        if (fsAction ==0)
        {

            // Get the length of the output stream
            int outLength = myOutputContext.getOutputContent().length;

            //Create a byte array and allocate outLength bytes
            byte[] formOutput = new byte[outLength];

            //Populate the byte array by invoking getOutputContext
            formOutput = myOutputContext.getOutputContent();

            //Create an InputStream object
            InputStream formInputStream = new ByteArrayInputStream(formOutput);

            // Create a DocumentBuilder object
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();

            // Create a Document object by calling the DocumentBuilder object's
            // parse method and pass the InputStream object
            Document myDOM = builder.parse(formInputStream);

            // Call getNodeText for each field in the form
            String myAmount = getNodeText("Amount", myDOM);
            String myLastName = getNodeText("LastName", myDOM);
            String myFirstName = getNodeText("FirstName", myDOM);
            String mySSN = getNodeText("SSN", myDOM);
            String myTitle = getNodeText("PositionTitle", myDOM);
            String myAddress = getNodeText("Address", myDOM);
            String myCity = getNodeText("City", myDOM);
            String myStateProv = getNodeText("StateProv", myDOM);
            String myZipCode = getNodeText("ZipCode", myDOM);
            String myEmail = getNodeText("Email", myDOM);
            String myPhoneNum = getNodeText("PhoneNum", myDOM);
            String myFaxNum = getNodeText("FaxNum", myDOM);
```

```
        String myDescription= getNodeText("Description", myDOM);
    } //End of If statement
} //End of try statement

catch (Exception ioEx)
{
    System.out.println("Exception error is: " +ioEx);
}
} // End of doPost

// Create the getNodeText custom method
private String getNodeText(String nodeName, Document myDOM)
{
    //Get the node by name. NodeName is the name of the
    //node passed to this method
    NodeList oList = myDOM.getElementsByTagName(nodeName);
    Node myNode = oList.item(0);
    NodeList oChildNodes = myNode.getChildNodes();

    String sText = "";
    for (int i = 0; i < oChildNodes.getLength(); i++)
    {
        Node oItem = oChildNodes.item(i);
        if (oItem.getNodeType() == Node.TEXT_NODE)
        {
            sText = sText.concat(oItem.getNodeValue());
        }
    }
    return sText;
} //End of getNodeText
```

Note: This code example retrieves data from the loan form. For information, see [“Relationship between form fields and XML data” on page 61](#).

Saving submitted data as XML

You can save submitted form data as an XML file by using the `IOutputContext` interface that is returned by the `ProcessFormSubmission` method. To save submitted form data as an XML file, make sure the data is submitted as XML data. That is, ensure that the content type of the submitted data is `text/xml` or `application/vnd.adobe.xdp+xml`. For information, see [“Form design considerations” on page 60](#).

To save data that is submitted from a client device, write the data to a `java.io.File` object. Create a Java `java.io.File` object by using its public constructor and ensure that you specify XML as the file name extension. Populate this object with the submitted XML data by invoking the `IOutputContext` interface’s `getOutputContent` method. This method returns a byte array that contains XML data.

Write the byte array contents to a `java.io.File` object by using a `java.io.FileOutputStream` object. Invoke this object’s `write` method and pass the byte array.

Example 3.4 Saving submitted data as XML

```
//Create an EJBClient object, which implements IFormServer
EJBClient formServer = new EJBClient();

// Call processFormSubmission to handle the submitted XML data. Pass the
// HttpServletRequest object
IOException outputContext =
formServer.processFormSubmission(req, "OutputType=0");

//Determine the content type -- make sure it is text/xml
String ct = outputContext.getContentType();

if ((ct.equals("text/xml")) || (ct.equals("application/vnd.adobe.xdp+xml")))
{
    // Get the length of the output stream
    int outLength = outputContext.getOutputStream().length;

    //Create a byte array and allocate outLength bytes
    byte[] formOutput = new byte[outLength];

    //Populate the byte array by invoking getOutputStream
    formOutput = outputContext.getOutputStream();

    //Create a XML File object
    File tempFile = new File("C:\\myXML.xml");

    //Create a Java FileOutputStream object
    FileOutputStream myOutput = new FileOutputStream(tempFile);

    //Write the byte array contents to the file
    myOutput.write(formOutput);
    myOutput.close();
}
```

Converting the content type of form data

Typically, the Form Server Module returns data to a client application as XDP data. However, you can use the Form Server Module API to convert the content type of data that the Form Server Module returns to a client application.

Consider, for example, a situation where a form that was created by using Acrobat is submitted. However, the client application that handles the submitted form contains business logic to parse XFA data (the format of forms created by using LiveCycle Designer), not XFDF data (the format of forms created by using Acrobat). In this situation, the XFDF data can be converted to XFA data so that the client application can parse the data and retrieve the values that a user entered.

You can convert the content type of data by setting a run-time option named `exportDataFormat` that you pass to the `processFormSubmission` method. To pass the `exportDataFormat` to the `processFormSubmission` method, set the `sOptions` parameter with the following string value:

```
exportDataFormat=<value>
```

The `exportDataFormat` run-time option supports the following values:

XDP: Returns submitted data as XDP (default).

XDP.toXFAData: Returns submitted data as XDP with applicable conversions to XFA data (for example, XFDF data is converted and returned as XFA data in the datasets packet).

XDP.DataOnly: Returns only the data packets of the XDP data (exclude the PDF base64 encoded packet). Applicable conversions to XFA data are performed (for example, XFDF data is converted to XFA data if an *Acrobat form* is submitted).

XMLData: Returns XML data without any XFA data. This is useful when you do not want to parse through elements such as XDP and datasets. Applicable conversions to XFA data are performed. For example, XFDF data is converted to XFA data if an *Acrobat form* is submitted.

Auto: Determines the content type depending on what format is submitted. If XDP is submitted, XDP is returned. If XML data is submitted, XML data is returned. If Url-encoded data is submitted, XML data is returned. If PDF is submitted, XDP is returned. Any applicable conversions to XFA data are done. XFDF data is converted to XFA data if an *Acrobat form* is submitted.

The following table lists the different ways in which a form's content type can be converted.

ExportDataFormat value	Submitted content type	Returned content type
XDP	<ul style="list-style-type: none"> text/xml application/vnd.adobe.xdp+xml application/pdf, application/x-www-form-urlencoded 	application/vnd.adobe.xdp+xml
XDP.toXFAData	<ul style="list-style-type: none"> text/xml application/vnd.adobe.xdp+xml application/pdf application/x-www-form-urlencoded 	application/vnd.adobe.xdp+xml
XDP.DataOnly	<ul style="list-style-type: none"> text/xml application/vnd.adobe.xdp+xml application/pdf application/x-www-form-urlencoded 	application/vnd.adobe.xdp+xml
XMLData	<ul style="list-style-type: none"> text/xml application/vnd.adobe.xdp+xml application/pdf application/x-www-form-urlencoded 	text/xml
Auto	<ul style="list-style-type: none"> application/vnd.adobe.xdp+xml application/pdf 	application/vnd.adobe.xdp+xml
Auto	<ul style="list-style-type: none"> text/xml application/x-www-form-urlencoded 	text/xml

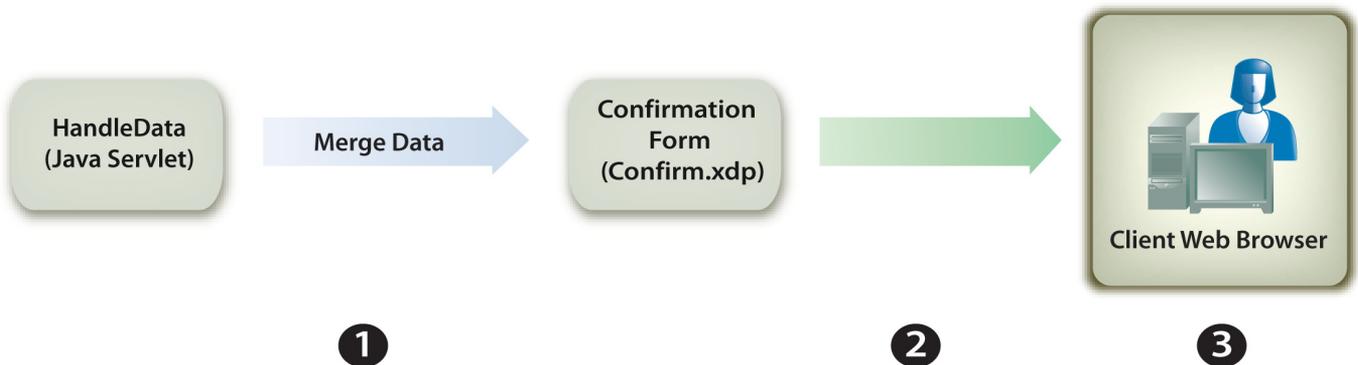
Rendering prepopulated forms

A client application can prepopulate a form with data prior to rendering it. The data can come from a variety of sources, such as an enterprise database, another form, or another application. Prepopulating a form has several advantages:

- Enables the user to view custom data in a form
- Reduces the amount of typing the user does to fill in a form
- Ensures data integrity by having control over where data is placed

Using the Form Server Module API, you can create a client application capable of prepopulating a form. Consider the loan sample application introduced earlier in the chapter. After data is submitted to the HandleData Java servlet, a confirmation form is rendered back to the web browser. This form contains data that the user entered into the loan application. For information about this form, see [“Confirmation form” on page 51](#).

The following diagram shows the HandleData Java servlet prepopulating the confirmation form and rendering it to the client web browser.



The following table explains the steps in the diagram.

- | | |
|---|---|
| 1 | The HandleData Java servlet prepopulates the confirmation form with data. |
| 2 | The confirmation form is rendered to the client web browser. |
| 3 | The confirmation form is displayed in the client web browser. |

Note: Prepopulating a form is also known as merging data with a form.

Creating application logic to render a prepopulated form

You must assign a byte array to the `renderForm` method's `cData` parameter to prepopulate a form prior to rendering it. This byte array represents an XML data source containing fields located in the form. For each field that you want to prepopulate, you must specify a value. It is not necessary to match the exact structure of the XML document. For example, to prepopulate the confirmation form, specify a value for the `LastName`, `FirstName`, and `Amount` fields.

Assume that a form containing 10 fields has data in 4 of the fields. Next, assume that you want to prepopulate the remaining 6 fields. In this situation, you must specify 10 XML elements in the XML data source used to prepopulate the form. If you specify only 6 elements, the original 4 fields will be empty.

Here are three ways in which you can create a byte array to assign to the `cData` parameter:

- Convert an existing XML document containing data to merge to a byte stream.
- Assign a string representing an XML document to a byte array. To convert a string to a byte array, you call the `String` object's `getBytes` method. If the XML document is encoded as UTF-8, it must remain that way. Ensure that the `getBytes` method does not change UTF-8 to unicode encoding.
- Create an in-memory XML data source using Java DOM classes. For information, see [“Creating an in-memory XML data source” on page 83](#).

The method you choose depends on your preference. However, you would typically use the third method when prepopulating a form containing many fields or a dynamic form. For information about prepopulating a dynamic form, see [“Rendering prepopulated dynamic forms” on page 82](#).

Note: Instead of assigning a byte array to the `renderForm` method's `cData` parameter, you can assign a `Document` object. For information, see [“Prepopulating a form using a Document object” on page 72](#).

Converting an XML document to a byte stream

Before you convert an XML document to a byte stream, ensure its schema is valid and it contains data values that you want to merge with the form. Assume you want to merge the data entered into the loan form shown earlier in this chapter with the confirmation form. The first step is to ensure the XML schema matches the form, as shown in this example.

```
- <Untitled>
  <FirstName /> ← Corresponds to the FirstName field
  <LastName /> ← Corresponds to the LastName field
  <Amount /> ← Corresponds to the Amount field
</Untitled>
```

The next step is to ensure that the XML document contains data values, as shown in this example.

```
<FirstName>Jerry</FirstName>
<LastName>Johnson</LastName>
<Amount>250000</Amount>
```

Once you ensure that the XML document's schema is valid and it contains data that you want to merge with the form, you convert the XML document to a byte stream. However, you must first reference it. Using a Java `URL` object, you can reference an XML document located on any computer accessible over the Internet or an intranet.

You can use a `BufferedInputStream` Java object to convert an XML document to a byte stream. After you create a byte stream, you can assign it to the `renderForm` method's `cData` parameter. For information about this method, see the *Form Server Module API Reference*.

The following process describes the application logic to prepopulate a form by converting an XML document to a byte stream:

1. Create a Java `URL` object that references an existing XML document.
2. Create a `BufferedInputStream` object by using its constructor. Within the constructor, pass the `URL` object and call its `openStream` method, as the following example shows:

```
BufferedInputStream oStream = new
BufferedInputStream (URLob . openStream ( ) ) ;
```

3. Create a user-defined method that takes a `BufferedInputStream` object as a parameter and returns a byte stream. This method, named `convertXML`, is shown in the example following this process.
4. Assign the byte stream to the `renderForm` method's `cData` parameter. When `renderForm` is called, the byte stream is merged into the form specified by the `sFormQuery` parameter.

The following code example prepopulates the confirmation form by assigning a byte stream to the `renderForm` method's `cData` parameter. The byte stream is created from an XML document named `ConfirmData.xml` and resembles the XML document shown in the previous diagram.

Example 3.5 Prepopulating a form by converting an XML document to a byte stream

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
    //Create an EJBClient object
    EJBClient formServer = new EJBClient();

    // Declare local variables to pass to renderForm
    String sFormName = "Confirm.xdp"; //Defines the rendered form
    String sFormPreference = "PDF";
    String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
    String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
    try{
        // Create an URL object that references an XML document named
        // ConfirmData.xml
        URL URLOb = new
            URL("http://<AppServer>:<AppPort>/LoanApp/forms/ConfirmData.xml");

        // Create a BufferedInputStream object. Pass the URL object to
        // its constructor and call openStream
        BufferedInputStream oStream = new BufferedInputStream
            (URLOb.openStream());

        //Populate the xXMLData parameter by calling convertXML. Pass the
        //BufferendInputStream object
        byte[] cData = ConvertXML(oStream);

        IOutputContext myOutputContext = formServer.renderForm(
            sFormQuery,           //sFormQuery
            sFormPreference,     //sFormPreference
            cData,               //cData,
            "CacheEnabled=False", //sOptions
            null,                //sUserAgent,
            sApplicationWebRoot, //sApplicationWebRoot
            sTargetURL,         //sTargetURL
            sApplicationWebRoot, //sApplicationWebRoot
            null                 //sBaseURL
        );

        // Create a ServletOutputStream object
        ServletOutputStream oOutput = resp.getOutputStream();

        //Set the HTTPResponse object's content type
        resp.setContentType(myOutputContext.getContentType());
    }
```

```

// Get the length of the output stream
int outLength = myOutputContext.getOutputStream().length;

//Create a byte array and allocate outLength bytes
byte[] cContent = new byte[outLength];

//Populate the byte array by invoking getOutputStream
cContent = myOutputContext.getOutputStream();

//Write a byte stream back to the web browser. Pass the byte array
oOutput.write(cContent);
} //end of try
//Catch a thrown exception
catch (Exception ioEx)
{
    System.out.println("Exception error is: " +ioEx.getMessage());
}
}

```

Converting an XML string to a byte stream

You can assign a byte array a string variable representing the XML document as opposed to converting an entire XML document to a byte stream. The `String` object's `getBytes` method is called, which converts the string to an array of bytes. You must specify the field names and values within the string.

The following code example prepopulates the confirmation form by converting a string variable to a byte stream.

Example 3.6 Prepopulating a form by converting a string variable to a byte stream

```

public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {

    //Create an EJBClient object
    EJBClient formServer = new EJBClient();

    // Declare local variables to pass to renderForm
    String sFormName = "Confirm.xdp"; //Defines the rendered form
    String sFormPreference = "PDF";
    String sApplicationWebRoot = "http://AppServer:<AppPort>/LoanApp";
    String sContentRootURI = "http://AppServer:<AppPort>/LoanApp/forms";

    try{
        //Assign a stream byte to the cData parameter
        String confirmData =
            "<root><FirstName>Jerry</FirstName><LastName>Johnson
            </LastName><Amount>250000</Amount></root>";
        byte [] cData = confirmData.getBytes("UTF-8");

        //Call renderForm and pass cData to prepopulate the confirmation form
        IOOutputContext myOutputContext = formServer.renderForm(
            sFormQuery, //sFormQuery
            sFormPreference, //sFormPreference
            cData, //cData,
            "CacheEnabled=False", //sOptions

```

```
        null,                //sUserAgent,
        sApplicationWebRoot, //sApplicationWebRoot
        sTargetURL,         //sTargetURL
        sApplicationWebRoot, //sApplicationWebRoot
        null                //sBaseURL
    );

    // Create a ServletOutputStream object
    ServletOutputStream oOutput = resp.getOutputStream();

    //Set the HTTPResponse object's content type
    resp.setContentType(myOutputContext.getContentType());

    // Get the length of the output stream
    int outLength = myOutputContext.getOutputContent().length;

    //Create a byte array and allocate outLength bytes
    byte[] cContent = new byte[outLength];

    //Populate the byte array by invoking getOutputContext
    cContent = myOutputContext.getOutputContent();

    //Write a byte stream back to the web browser. Pass the byte array
    oOutput.write(cContent);
} //end of try
//Catch a thrown exception
catch (Exception ioEx)
{
    System.out.println("Exception error is: " +ioEx.getMessage());
}
} //end of doPost
```

Prepopulating a form using a Document object

You can prepopulate a form by assigning the `renderForm` method's `cData` parameter a `Document` object. The `Document` object must store valid XML containing data values that you want to merge with the form. For information, see [“Converting an XML document to a byte stream” on page 69](#).

After you create a `Document` object, invoke the `renderForm` method and pass the `Document` object. For information about creating a `Document` object, see [“Working with the Document object” on page 43](#).

The following code example prepopulates a form by using a `Document` object.

Example 3.7 Prepopulating a form using a Document object

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
    //Create an EJBClient object
    EJBClient formServer = new EJBClient();

    // Declare local variables to pass to renderForm
    String sFormQuery = "Confirm.xdp"; //Defines the rendered form
    String sFormPreference = "PDF";
    String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
    String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
```

```
try{
    // Create a Document object that references an XML document named
    // ConfirmData.xml

    //Create an URL object
    URL myURL = new
URL ("http://<AppServer>:<AppPort>/DataFile/ConfirmData.xml");

    //Create a Document object
    Document myXMLDocument = new Document (myURL);

    //Invoke renderForm
    IOutputContext myOutputContext = formServer.renderForm(
    sFormQuery,           //sFormQuery
    sFormPreference,     //sFormPreference
    myXMLDocument,       //pass the Document object,
    "CacheEnabled=False", //sOptions
    null,                 //sUserAgent,
    sApplicationWebRoot, //sApplicationWebRoot
    null,                 //sTargetURL
    sApplicationWebRoot, //sApplicationWebRoot
    null                  //sBaseURL
    );

    // Create a ServletOutputStream object
    ServletOutputStream oOutput = resp.getOutputStream();

    //Set the HTTPResponse object's content type
    resp.setContentType(myOutputContext.getContentType());

    // Get the length of the output stream
    int outLength = myOutputContext.getOutputContent().length;

    //Create a byte array and allocate outLength bytes
    byte[] cContent = new byte[outLength];

    //Populate the byte array by invoking getOutputContext
    cContent = myOutputContext.getOutputContent();

    //Write a byte stream back to the web browser. Pass the byte array
    oOutput.write(cContent);
} //end of try
//Catch a thrown exception
catch (Exception ioEx)
{
    System.out.println("Exception error is: " +ioEx.getMessage());
}
} //end of doPost
```

Rendering a form at the client

You can optimize the delivery of PDF content and improve LiveCycle Form's ability to handle network load by using the client-side rendering capability of Acrobat 7.0 or Adobe Reader 7.0 or higher.

This process is known as rendering a form at the client. To render a form at the client, the client device (typically a web browser) must use Acrobat 7.0 or Adobe Reader 7.0 or higher.

To render a form at the client, set the following run-time options that belong to the `renderForm` method's `sOptions` parameter:

- `RenderAtClient`
- `SeedPDF` (this is conditional depending on the transformation)

First, you must assign a value to the `RenderAtClient` run-time option. If `RenderAtClient` is set to `true`, the form is delivered to the client device where it is rendered.

If `RenderAtClient` is `auto` (the default value), the form design determines whether the form is rendered at the client. The form design must be a dynamic form design. For information about a dynamic form design, see the *LiveCycle Designer Help*.

The `SeedPDF` run-time option is used when the transformation is `PDF` or `PDFForm` (the `renderForm` method's `sFormPreference` parameter to `PDF` or `PDFForm`). For information about setting the `renderForm` method's `sFormPreference` parameter, see ["Setting preference options to render the form as PDF" on page 53](#).

The `SeedPDF` run-time option is a PDF container that acts as the seed from which a dynamically rendered PDF is displayed. It may contain additional fonts required by the form being rendered that are not provided with the form itself.

When you use the `SeedPDF` option to optimize the performance of the Form Server Module, the transformation is responsible for combining the PDF container with the form design and the xml data. Both the form design and the xml data are delivered to Acrobat or Adobe Reader, where the form is rendered.

If the transformation is `PDFMerge` (the `renderForm` method's `sFormPreference` parameter to `PDFMerge`), then it is not necessary to use the `SeedPDF` run-time option. Instead, assign the `renderForm` method's `sFormQuery` parameter with the file name of the seed.pdf. You must also convert the form design and the xml data to a byte array and assign the byte array to the `renderForm` method's `cData` parameter. A possible usage for passing in the form design with the data is in cases where the form design is perhaps not available on disk, or it is being dynamically generated by the application.

If you want to define the `seedPDF` run-time option, it is recommended that you use LiveCycle Designer to create a simple dynamic PDF file for use as a seed PDF file. The following steps are required to perform this task:

1. Determine whether you need to embed any fonts within the seed PDF file. Open a new file. The seed PDF file will need to contain additional fonts required by the form being rendered, if any. When embedding fonts into the seed PDF file, make sure you are not violating any font licensing agreements. In LiveCycle Designer, you can determine whether you can legally embed fonts: save the seed PDF file as a dynamic PDF form. Upon saving, if there are fonts you cannot embed into the form, LiveCycle Designer displays a message listing the fonts you cannot embed. This message is not displayed in LiveCycle Designer for static PDFs.

2. If you are creating the seed PDF file in LiveCycle Designer, Adobe recommends that, at a minimum, you add a text field that contains a message. The message should be directed at users of earlier versions of Adobe Reader stating that they need Acrobat 7.0 or Adobe Reader 7.0 to view the document.
3. Save the seed PDF file as a dynamic PDF file with the PDF file extension.

The following process describes how to render a form at the client (in this example, the `seedPDF` run-time option is not defined):

1. Create an `EJBClient` object by calling the `EJBClient` constructor. For information, see [“Locally invoking Form Server Module” on page 31](#).
2. Convert a form design (an XDP file that contains both data and a form design) into an array of bytes. A form design is based on XFA architecture and can contain both data and a form design. For information about XFA architecture, go to http://partners.adobe.com/public/developer/xml/index_arch.html.
3. Set the `RenderAtClient` run-time option to `true` (this run-time option belongs to the `renderForm` method's `sOptions` parameter).
4. Invoke the `renderForm` method and set its parameters (this is shown in the example that follows this numbered list). Set the `sFormPreference` parameter to `PDFForm`. Also, assign the byte array (created in step two) to the `cData` parameter.
5. The `renderForm` method returns an `IOutputContext` interface. You are able to get the `IOutputContext` interface's content type and its byte stream. A typical usage involves using Java classes to send the byte stream to the client web browser. For information, see [“Rendering a form using an EJBClient object” on page 52](#).

The following code example renders a form at the client.

Example 3.8 *Rendering a form at the client*

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
    //Create an EJBClient object
    EJBClient formServer = new EJBClient();

    // Reference an XDP that contains data and a form design
    File myXDP = new File("C:\\LoanForm.xdp");

    // Create a FileInputStream object
    FileInputStream fInput = new FileInputStream(myXDP);

    // Get the size of the buffer
    int mSize = fInput.available();
    byte [] myBytes = new byte[mSize];

    //Populate the byte array
    fInput.read(myBytes);
```

```
// Declare local variables to pass to renderForm
String sFormQuery = "Seed.pdf"; //Defines the file name of the Seed PDF
String sFormPreference = "PDFMerge";
String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
String sBaseURL = null;

// Define the run-time option
String options = "RenderAtClient=true";
try{
    //Call renderForm()
    IOutputContext myOutputContext = formServer.renderForm(
        sFormQuery, //sFormQuery
        sFormPreference, //sFormPreference
        myBytes, //cData,
        options, //sOptions
        null, //sUserAgent,
        sApplicationWebRoot, //sApplicationWebRoot
        null, //sTargetURL
        sContentRootURI, //sContentRootURI
        sBaseURL //sBaseURL
    );
    // Create a ServletOutputStream object
    ServletOutputStream oOutput = resp.getOutputStream();

    //Set the HTTPResponse object's content type
    resp.setContentType(myOutputContext.getContentType());

    // Get the length of the output stream
    int outLength = myOutputContext.getOutputContent().length;

    //Create a byte array and allocate outLength bytes
    byte[] cContent = new byte[outLength];

    //Populate the byte array by invoking getOutputContext
    cContent = myOutputContext.getOutputContent();

    //Write a byte stream back to the web browser. Pass the byte array
    oOutput.write(cContent);
} //end of try
//Catch a thrown exception
catch (Exception ioEx)
{
    System.out.println("Exception error is: " +ioEx.getMessage());
}
}
```

Passing a form design by value

Typically, a form design is created in LiveCycle Designer and is passed by reference to the Form Server Module. This is achieved by specifying a value for the `renderForm` method's `sFormQuery` and `sContentRootURI` parameters. For information, see [“Specifying the form design to render” on page 53](#).

The following list describes the advantages of passing a form design by reference:

- Form designs can be large and, as a result, it is more efficient to pass them by reference to avoid having to marshal form design bytes by value. The Form Server Module can also cache the form design so that when cached, it does not have to continually read the form design from a repository.
- By using both the `FormQuery` parameter and the `ContentRootURI` parameter, the Form Server Module can also resolve the location of linked content within the form design. For example, linked images that are referenced from within the form design are relative URLs. Linked content is always assumed to be relative to the form design location. Therefore, resolving linked content is a matter of determining its location by applying the relative path to the absolute location of the form design.
- After forms are rendered, the Form Server Module can perform server-side calculations. To perform these calculations, the Form Server Module requires a location to where form designs can be reread. The Form Server Module obtains a location by embedding the `FormQuery` and `ContentRootURI` values as pass-through data in the rendered form. When the client device makes subsequent requests for calculations, it submits these parameters along with the data. The Form Server Module uses these parameters to reread the form design from the referenced location.

Instead of passing a form design by reference, you can pass a form design by value. Passing a form design by value is efficient when a form design is dynamically created; that is, when a client application generates the XML that creates a form design. In this situation a form design is not stored in a physical repository because it is stored in memory.

The following limitations apply when a form design is passed by value:

- No relative linked content can be within the form design. All images and fragments must be embedded inside the form design or referred to absolutely.
- Server-side calculations cannot be performed after the form is rendered. If the form is submitted back to the Form Server Module, the data is extracted and returned without any server-side calculations.
- Because HTML can only use linked images at run time, it is not possible to generate HTML with embedded images. This is because the Form Server Module supports embedded images with HTML by retrieving the images from a referenced form design. Because a form design that is passed by value does not have a referenced location, embedded images cannot be extracted when the HTML page is displayed. Therefore, image references must be absolute paths to be rendered in HTML.
- Data entered into a rendered HTML form that is passed by value is not submitted back to LiveCycle Forms. That is, assume a user enters data into an HTML form that was passed by value and then clicks a submit button (that is located on the form). Because the data is not submitted back, you cannot invoke the `processFormSubmission` method and retrieve the data. If you want to pass an interactive form by value in order to retrieve data from the user, then use the `PDFForm` transformation type. Otherwise, pass an interactive HTML form by reference. For more information, see [“Setting preference options to render the form as PDF” on page 53](#).

You must dynamically create a form design programmatically in order to pass a form design by value. If you want to display data within the form that will be rendered as a PDF form, the data must be specified within the `xfa:datasets` element. For information about XFA architecture, go to http://partners.adobe.com/public/developer/xml/index_arch.html.

To pass a form design by value, create an `EJBClient` object (or a `SOAPClient` object) and call its `renderForm` method. Assign the following values to the `renderForm` method's parameters:

- `sFormQuery`: This value is empty when passing a form design by value.
- `sFormPreference`: This value can be set to a valid PDF or HTML transformation value. For information about supported values, see the *Form Server Module API Reference*.
- `cData`: This value must be valid XFA XML specified as a byte array or a `Document` object.
- `sOptions`: All run-time options are supported.
- `sUserAgent`: This value is support but not required.
- `sApplicationWebRoot`: This value is supported.
- `sTargetURL`: This value is the target for the form submission. For information, see ["Specifying the target URL" on page 54](#).
- `sContentRootURI`: This value is ignored because the form design is passed by value and the form design contains no relative linked content.
- `sBaseURL`: This value is ignored because the form design is passed by value and the form design contains no relative linked content.

4 Rendering Dynamic Forms

This chapter explains how you can use the Form Server Module API to develop client applications capable of rendering dynamic forms across the Internet or an intranet to client devices, typically web browsers. A dynamic form can display an undetermined amount of data. An example of a dynamic form is a purchase order form that displays purchased items. The number of items appearing in a purchase order form differs from customer to customer.

This chapter contains the following information.

Topic	Description	See
About dynamic forms	Describes a dynamic form's characteristics.	page 79
Rendering prepopulated dynamic forms	Describes how you can use the Form Server Module API to prepopulate and render a dynamic form to a client web browser.	page 82

Note: Before reading this chapter, it is recommended that you are familiar with using the Form Server Module API to render forms. For information, see ["Rendering Interactive Forms as PDF" on page 48](#).

About dynamic forms

Dynamic forms are useful to display an undetermined amount of data to users. Because the layout of a dynamic form adjusts automatically to the amount of data that is merged, you do not need to predetermine a fixed layout or number of pages for the form as you need to do with a static form.

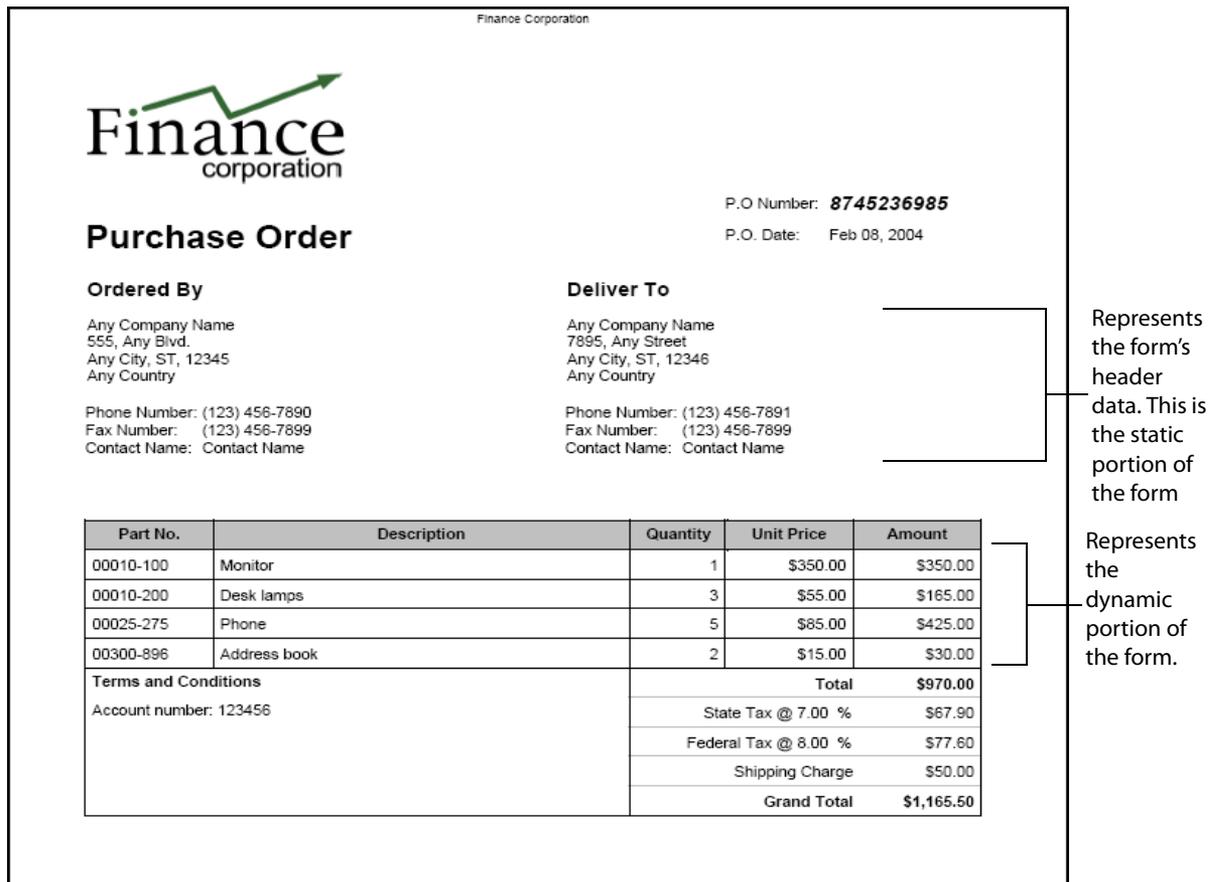
Two types of dynamic forms exist: client-side and server-side dynamic forms. A client-side dynamic form is typically used to collect data from end users by enabling them to click a button (or another control) that produces a new field in which data is entered. The new field appears on the form immediately and does not require a round trip to the server. That is, the form is not sent to the J2EE application server hosting LiveCycle Forms and then rendered back to the client web browser with the new field.

Assume, for example, a client-side dynamic form contains fields that enable a user to enter items to purchase and a button that enables the user to add new fields. Each time the user clicks the button, a new field is added to the form. You can create a client-side dynamic form by using LiveCycle Designer. For information, see the *LiveCycle Designer Help*.

In contrast, a server-side dynamic form is populated on the server. For example, a client application that uses the Form Server Module API can query a database and retrieve an unknown number of records. The client application then prepopulates a dynamic form with the data and then renders the form to a client web browser. This chapter explains how to use the Form Server Module API to prepopulate and render server-side dynamic forms. It does not explain how to create or use client-side dynamic forms.

Consider, for example, a web-based online store. As a user selects items to purchase, the items are added to a virtual shopping cart. When the user has finished purchasing items, a dynamic purchase order form is displayed that the user can print and keep as proof of the transaction. The dynamic form contains all the purchased items, as well as other information, such as a description of the items, the subtotal, taxes, and total.

The following diagram shows an example of a dynamic purchase order form.



Note: Dynamic forms can be prepopulated with data from other sources such as an enterprise database or external applications. A web-based application represents only one example of using dynamic forms.

Form design considerations

Both server-side and client-side dynamic forms are based on form designs that are created in LiveCycle Designer. A form design specifies a set of layout, presentation, and data capture rules, including calculating values based on user input. The rules are applied when data is entered into a form. Fields that are dynamically added to a dynamic form are subforms that are within the form design. For example, in the purchase order form shown in the previous diagram, each line is a subform. For information about creating a form design that contains subforms, see the *LiveCycle Designer Help*.

XML data source

It is important that you understand the relationship between a form design on which a server-side dynamic form is based and the XML data source that is used to prepopulate it. A server-side dynamic form is prepopulated with data when a form design is merged with an XML data source. The following two XML data sources can prepopulate a dynamic form:

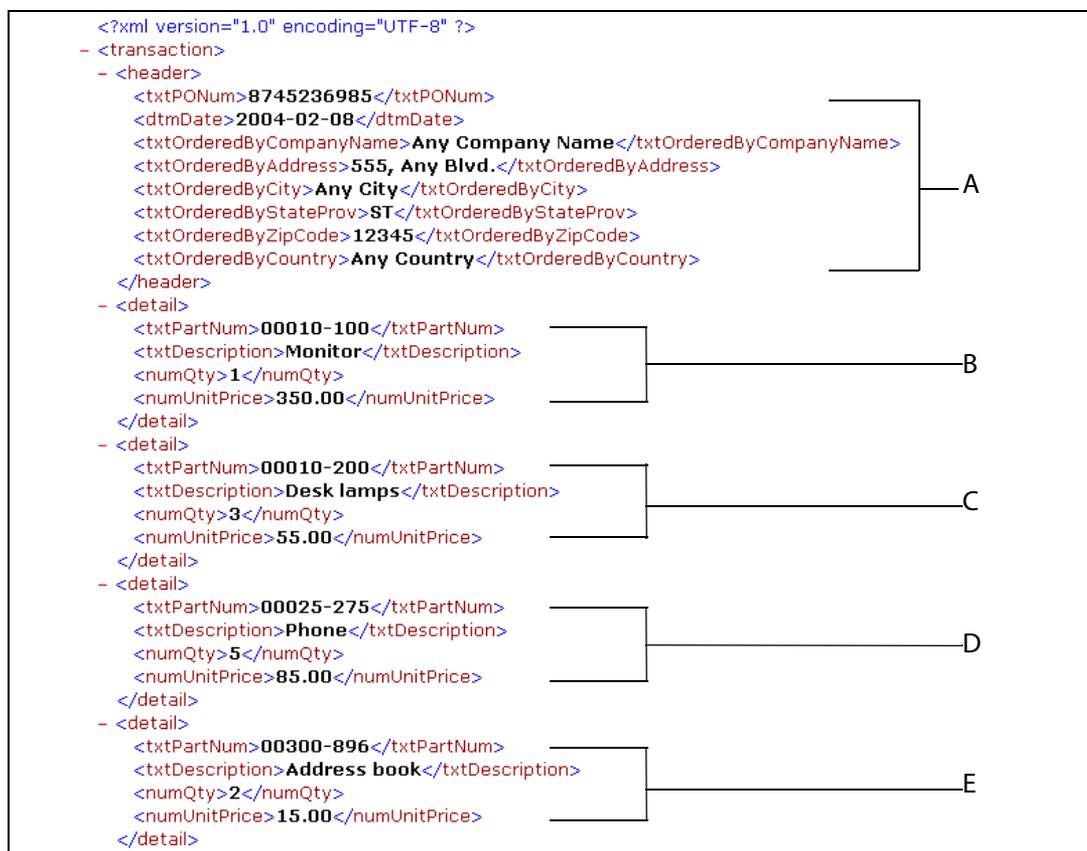
- An XDP data source, which is XML that conforms to XFA syntax.
- An arbitrary XML data source that contains name/value pairs matching the form's field names (the examples in this chapter use an arbitrary XML data source).

An XML element must exist for every form field that you want to prepopulate. The XML element name must match the field name. An XML element is ignored if it does not correspond to a form field or if the XML element name does not match the field name.

An XML data source is used to prepopulate both static and dynamic forms. However, the difference is that an XML data source that prepopulates a dynamic form contains repeating XML elements that are used to prepopulate subforms that are repeated within the form. These repeating XML elements are called data subgroups.

The XML data source that is used to prepopulate the dynamic purchase order form shown in the previous diagram contains four repeating data subgroups. Each data subgroup corresponds to a purchased item. The purchased items are a monitor, a desk lamp, a phone, and an address book.

The following diagram shows the arbitrary XML data source that is used to prepopulate the dynamic purchase order form.



The following table explains the letters in the previous diagram.

A	XML elements used to prepopulate non-repeating fields such as address and city. This is the static portion of the form.
B	A data subgroup that contains information about the monitor.
C	A data subgroup that contains information about the desk lamp.
D	A data subgroup that contains information about the phone.
E	A data subgroup that contains information about the address book.

Notice that each data subgroup contains four XML elements that correspond to this information:

- Items part number
- Items description
- Quantity of items
- Unit price

The name of a data subgroup's parent XML element must match the name of the subform that is located in the form design. For example, in the previous diagram, notice that the name of the data subgroup's parent XML element is `detail`. This corresponds to the name of the subform that is located in the form design on which the purchase order form is based. If the name of the data subgroup's parent XML element and the subform do not match, a server-side dynamic form is not prepopulated.

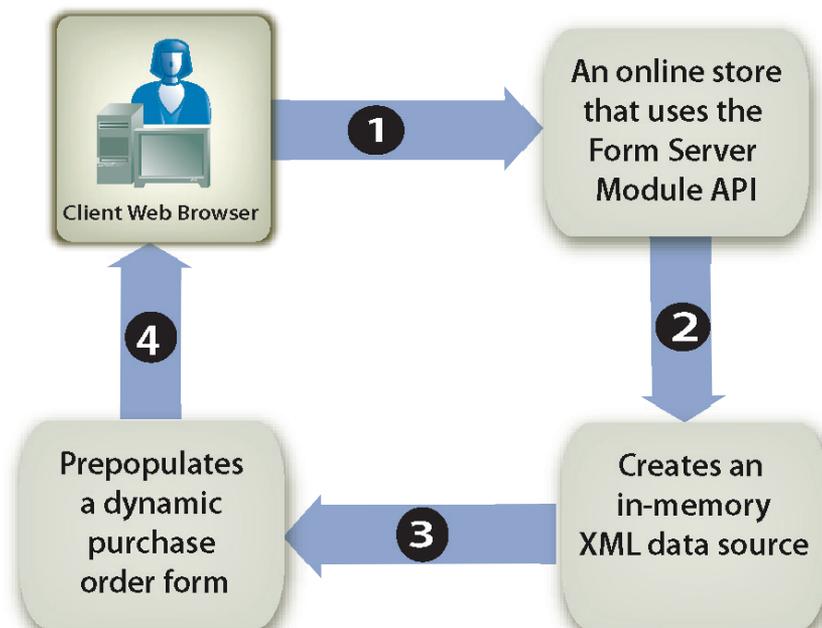
Each data subgroup must contain XML elements that match the field names in the subform. The `detail` subform located in the form design contains the following fields:

- `txtPartNum`
- `txtDescription`
- `numQty`
- `numUnitPrice`

Rendering prepopulated dynamic forms

You use the Form Server Module API to create applications that render server-side dynamic forms. Because the data that is placed into a server-side dynamic form is obtained at run-time, you must prepopulate a dynamic form by creating an in-memory XML data source and placing the data directly into the in-memory XML data source.

Consider the example of a web-based online store. After a user is finished purchasing items, all the purchased items are placed into an in-memory XML data source that is used to prepopulate the dynamic form. The following diagram shows this process, which is explained in the table following the diagram.



The following table describes the steps in this diagram.

-
- | | |
|---|--|
| 1 | A user purchases items from a web-based online store. The online store is implemented as a Java servlet that uses the Form Server Module API. |
| 2 | After the user finishes purchasing items and clicks the Submit button, an in-memory XML data source is created. Purchased data items and user information are placed into the in-memory XML data source. |
| 3 | The XML data source is used to prepopulate a dynamic purchase order form. |
| 4 | The dynamic purchase order form is rendered to the client web browser. For an example of a dynamic purchase order form, see "About dynamic forms" on page 79 . |
-

Creating an in-memory XML data source

You can use Java DOM classes to create an in-memory XML data source. This task is necessary if you are creating an application, such as a web-based online store, that receives data at run-time and uses the data to prepopulate a dynamic form. The data must be placed into an XML data source that conforms to the form. For information about the relationship between a dynamic form and the XML data source, see ["XML data source" on page 80](#).

The following process describes one way to create an in-memory XML data source:

1. Create a Java `DocumentBuilderFactory` object by calling the `DocumentBuilderFactory` class' `newInstance` method.
2. Create a Java `DocumentBuilder` object by calling the `DocumentBuilderFactory` object's `newDocumentBuilder` method.
3. Call the `DocumentBuilder` object's `newDocument` method to instantiate a Java `Document` object.
4. Create the XML data source's root element by calling the `Document` object's `createElement` method. This creates an `Element` object that represents the root element. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the root element to the document by calling the `Document` object's `appendChild` method, and pass the root element object as an argument. The following lines of code shows this application logic:

```
Element root = (Element)document.createElement("transaction");  
document.appendChild(root);
```

5. Create the XML data source's header element by calling the `Document` object's `createElement` method. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the header element to the root element by calling the the `root` object's `appendChild` method, and pass the header element object as an argument. The XML elements that are appended to the header element correspond to the static portion of the form. The following lines of code shows this application logic:

```
Element header = (Element)document.createElement("header");  
root.appendChild(header);
```

6. Create a child element that belongs to the header element by calling the `Document` object's `createElement` method, and pass a string value that represents the element's name. Cast the return value to `Element`. Next, set a value for the child element by calling its `appendChild` method, and pass the `Document` object's `createTextNode` method as an argument. Specify a string value that

appears as the child element's value. Finally, append the child element to the header element by calling the header element's `appendChild` method, and pass the child element object as an argument. The following lines of code shows this application logic:

```
Element poNum= (Element)document.createElement("txtPONum");
poNum.appendChild(document.createTextNode("8745236985"));
header.appendChild(poNum);
```

7. Add all remaining elements to the header element by repeating step 6 for each field appearing in the static portion of the form (in the XML data source diagram, these fields are shown in section A).
8. Create the XML data source's detail element by calling the `Document` object's `createElement` method. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`. Next, append the detail element to the root element by calling the `root` object's `appendChild` method, and pass the detail element object as an argument. The XML elements that are appended to the detail element correspond to the dynamic portion of the form. The following lines of code shows this application logic:

```
Element detail = (Element)document.createElement("detail");
root.appendChild(detail);
```

9. Create a child element that belongs to the detail element by calling the `Document` object's `createElement` method, and pass a string value that represents the element's name. Cast the return value to `Element`. Next, set a value for the child element by calling its `appendChild` method, and pass the `Document` object's `createTextNode` method as an argument. Specify a string value that appears as the child element's value. Finally, append the child element to the detail element by calling the detail element's `appendChild` method, and pass the child element object as an argument. The following lines of code shows this application logic:
- ```
Element txtPartNum = (Element)document.createElement("txtPartNum");
txtPartNum.appendChild(document.createTextNode("00010-100"));
detail.appendChild(txtPartNum);
```
10. Repeat step 9 for all XML elements to append to the detail element. To properly create the XML data source used to populate the purchase order form, you must append the following XML elements to the detail element: `txtDescription`, `numQty`, and `numUnitPrice`.
  11. Repeat steps 9 and 10 for all dynamic data items that you want to populate the form with.

The following code example shows a custom method named `GetXMLDataSource` that creates an in-memory XML data source that can prepopulate a dynamic form. The method returns a `Document` object.

#### **Example 4.1** *Creating an in-memory XML data source*

```
public Document GetXMLDataSource()
{
 Document document = null;
 try {
 //Create DocumentBuilderFactory and DocumentBuilder objects
 DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
 DocumentBuilder builder = factory.newDocumentBuilder();

 //Create a new Document object
 Document document = builder.newDocument();

 //Create the root element and append it to the XML DOM
 Element root = (Element)document.createElement("transaction");
```

```
document.appendChild(root);

//Create the header element
Element header = (Element)document.createElement("header");
root.appendChild(header);

//Create the txtPONum element and append it to the header element
Element txtPONum = (Element)document.createElement("txtPONum");
txtPONum.appendChild(document.createTextNode("8745236985"));
header.appendChild(txtPONum);

//Create the dtmDate element and append it to the header element
Element dtmDate = (Element)document.createElement("dtmDate");
dtmDate.appendChild(document.createTextNode("2004-02-08"));
header.appendChild(dtmDate);

//Create the txtOrderedByCompanyName element and append it to
//the header element
Element txtOrderedByCompanyName =
(Element)document.createElement("txtOrderedByCompanyName");
txtOrderedByCompanyName.appendChild(document.createTextNode("Any Company
Name"));
header.appendChild(txtOrderedByCompanyName);

//Create the txtOrderedByAddress element and append it to the header element
Element txtOrderedByAddress =
(Element)document.createElement("txtOrderedByAddress");
txtOrderedByAddress.appendChild(document.createTextNode("555, Any Blvd"));
header.appendChild(txtOrderedByAddress);

//Create the txtOrderedByCity element and append it to the header element
Element txtOrderedByCity =
(Element)document.createElement("txtOrderedByCity");
txtOrderedByCity.appendChild(document.createTextNode("Any City"));
header.appendChild(txtOrderedByCity);

//Create the txtOrderedByStateProv element and append it to the header element
Element txtOrderedByStateProv =
(Element)document.createElement("txtOrderedByStateProv");
txtOrderedByStateProv.appendChild(document.createTextNode("ST"));
header.appendChild(txtOrderedByStateProv);

//Create the txtOrderedByZipCode element and append it to the header element
Element txtOrderedByZipCode =
(Element)document.createElement("txtOrderedByZipCode");
txtOrderedByZipCode.appendChild(document.createTextNode("12345"));
header.appendChild(txtOrderedByZipCode);

//Create the txtOrderedByCountry element and append it to the header element
Element txtOrderedByCountry =
(Element)document.createElement("txtOrderedByCountry");
txtOrderedByCountry.appendChild(document.createTextNode("Any Country"));
header.appendChild(txtOrderedByCountry);

//Create the detail element and append it to the root
```

```
Element detail = (Element)document.createElement("detail");
root.appendChild(detail);

//Create the txtPartNum element and append it to the detail element
Element txtPartNum = (Element)document.createElement("txtPartNum");
txtPartNum.appendChild(document.createTextNode("00010-100"));
detail.appendChild(txtPartNum);

//Create the txtDescription element and append it to the detail element
Element txtDescription = (Element)document.createElement("txtDescription");
txtDescription.appendChild(document.createTextNode("Monitor"));
detail.appendChild(txtDescription);

//Create the numQty element and append it to the detail element
Element numQty = (Element)document.createElement("numQty");
numQty.appendChild(document.createTextNode("1"));
detail.appendChild(numQty);

//Create the numUnitPrice element and append it to the detail element
Element numUnitPrice = (Element)document.createElement("numUnitPrice");
numUnitPrice.appendChild(document.createTextNode("350.00"));
detail.appendChild(numUnitPrice);

//Create another detail element named detail2 and append it to root
Element detail2 = (Element)document.createElement("detail");
root.appendChild(detail2);

//Create the txtPartNum element and append it to the detail2 element
Element txtPartNum2 = (Element)document.createElement("txtPartNum");
txtPartNum2.appendChild(document.createTextNode("00010-200"));
detail2.appendChild(txtPartNum2);

//Create the txtDescription element and append it to the detail2 element
Element txtDescription2 = (Element)document.createElement("txtDescription");
txtDescription2.appendChild(document.createTextNode("Desk lamps"));
detail2.appendChild(txtDescription2);

//Create the numQty element and append it to the detail2 element
Element numQty2 = (Element)document.createElement("numQty");
numQty2.appendChild(document.createTextNode("3"));
detail2.appendChild(numQty2);

//Create the NUMUNITPRICE element
Element numUnitPrice2 = (Element)document.createElement("numUnitPrice");
numUnitPrice2.appendChild(document.createTextNode("55.00"));
detail2.appendChild(numUnitPrice2);
// end of in-memory XML data source
}

catch (Exception ee)
{
 System.out.println(ee);
}
return document;
}
```

## Java import statements

The following Java import statements must be added to your Java project to successfully compile this code example:

```
import org.w3c.dom.Document ;
import org.w3c.dom.NodeList ;
import org.w3c.dom.Node ;
import org.w3c.dom.Element ;
import javax.xml.parsers.* ;
```

## XML data source

The previous code example creates a subset of the XML data source shown earlier in the chapter. The element values are hard-coded for simplicity. Typically, these values are stored in variables and are then assigned to XML elements. The following diagram shows the XML data source that is created by this code example.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <transaction>
 - <header>
 <txtPONum>8745236985</txtPONum>
 <dtmDate>2004-02-08</dtmDate>
 <txtOrderedByCompanyName>Any Company Name</txtOrderedByCompanyName>
 <txtOrderedByAddress>555, Any Blvd</txtOrderedByAddress>
 <txtOrderedByCity>Any City</txtOrderedByCity>
 <txtOrderedByStateProv>ST</txtOrderedByStateProv>
 <txtOrderedByZipCode>12345</txtOrderedByZipCode>
 <txtOrderedByCountry>Any Country</txtOrderedByCountry>
 </header>
 - <detail>
 <txtPartNum>00010-100</txtPartNum>
 <txtDescription>Monitor</txtDescription>
 <numQty>1</numQty>
 <numUnitPrice>350.00</numUnitPrice>
 </detail>
 - <detail>
 <txtPartNum>00010-200</txtPartNum>
 <txtDescription>Desk lamps</txtDescription>
 <numQty>3</numQty>
 <numUnitPrice>55.00</numUnitPrice>
 </detail>
</transaction>
```

To see the purchase order form prepopulated with this XML data source, see [“Dynamic purchase order form” on page 90](#).

## Converting the XML data source to a byte array

An in-memory XML data source that is created by using Java DOM classes must be converted to a byte array before it can be used to prepopulate a form. The byte array is assigned to the `renderForm` method's `cData` parameter. An in-memory XML data source can be converted to a byte array by using Java XML transform objects.

The following code example shows a custom method named `ConvertDOM` that requires a `Document` object as an argument, uses Java XML transform classes to convert the `Document` object to a byte array, and returns the byte array.

### Example 4.2 Converting an in-memory XML data source to a byte array

```
public byte[] ConvertDOM(Document doc)
{
 //Declare a byte array
 byte[] mybytes = null ;
 try
 {
 // Create a Java Transformer object
 TransformerFactory transFact = TransformerFactory.newInstance();
 Transformer transForm = transFact.newTransformer();

 // Create a Java ByteArrayOutputStream object
 ByteArrayOutputStream myOutputStream = new ByteArrayOutputStream();

 //Create a Java Source object
 Source myInput = new DOMSource(doc);

 //Create a Java Result object
 Result myOutput = new StreamResult(myOutputStream);

 //Populate the Java ByteArrayOutputStream object
 transForm.transform(myInput, myOutput);

 // Get the size of the ByteArrayOutputStream buffer
 int myByteSize = myOutputStream.size();

 //Allocate myByteSize to the byte array
 mybytes = new byte[myByteSize];

 // Copy the content to the byte array
 mybytes = myOutputStream.toByteArray();
 }
 catch (Exception e)
 {
 System.out.println(e);
 }
 return mybytes ;
}
```

### Java import statements

The following Java import statements must be added to your Java project to successfully compile this code example:

```
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
```

## Rendering a prepopulated dynamic form

You can use the Form Server Module API to render a prepopulated dynamic form. The form design that is rendered in this section is named PurchaseOrder.xdp. The difference between rendering a static prepopulated form and rendering a dynamic prepopulated form is creating the XML data source that is used to prepopulate the dynamic form. For information about rendering a prepopulated form, see [“Rendering prepopulated forms” on page 68](#).

The following code example prepopulates the purchase order dynamic form by creating an in-memory XML data source, converting the XML data source to a byte array, and assigning the byte array to the `renderForm` method's `cData` parameter.

### Example 4.3 *Rendering a prepopulated dynamic form*

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
 ServletException, IOException {

 //Create an EJBClient object
 EJBClient formServer = new EJBClient();

 // Declare local variables to pass to renderForm
 String sFormName = "PurchaseOrder.xdp"; //Defines the form design
 String sFormPreference = "PDF";
 String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
 String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
 try{
 // Create an in-memory XML data source
 Document mydoc = GetXMLDataSource();

 //Populate the cData parameter by calling ConvertDOM. Pass the
 //Document object
 byte[] cData = ConvertDOM(mydoc);

 IOOutputContext myOutputContext = formServer.renderForm(
 sFormQuery, //sFormQuery
 sFormPreference, //sFormPreference
 cData, //cData,
 "CacheEnabled=False", //sOptions
 null, //sUserAgent,
 sApplicationWebRoot, //sApplicationWebRoot
 sTargetURL, //sTargetURL
 sApplicationWebRoot, //sApplicationWebRoot
 null //sBaseURL
);

 // Create a ServletOutputStream object
 ServletOutputStream oOutput = resp.getOutputStream();

 //Set the HTTPResponse object's content type
 resp.setContentType(myOutputContext.getContentType());

 // Get the length of the output stream
 int outLength = myOutputContext.getOutputContent().length;

 //Create a byte array and allocate outLength bytes
```

```
byte[] cContent = new byte[outLength];

//Populate the byte array by invoking getOutputContext
cContent = myOutputContext.getOutputContent();

//Write a byte stream back to the web browser. Pass the byte array
oOutput.write(cContent);
}
//Catch a thrown exception
catch (Exception ioEx)
{
 System.out.println("Exception error is: " +ioEx);
}
} //end of doPost
```

### Dynamic purchase order form

This code example calls the two custom methods that are created in this chapter. GetXMLDataSource is called to create an in-memory XML data source that is used to prepopulate the purchase order dynamic form. For information about this XML data source, see [“Creating an in-memory XML data source” on page 83](#).

The custom method, convertDOM, is called to transform the in-memory XML data source to an array of bytes. The array of bytes is assigned to the renderForm method’s cData parameter. The following diagram shows the purchase order form after it is prepopulated with the XML data source created in this chapter.

Finance Corporation



P.O Number: **8745236985**

P.O. Date: Feb 08, 2004

## Purchase Order

**Ordered By**

Any Company Name  
 555, Any Blvd  
 Any City, ST, 12345  
 Any Country

Phone Number:  
 Fax Number:  
 Contact Name:

**Deliver To**

Phone Number:  
 Fax Number:  
 Contact Name:

Part No.	Description	Quantity	Unit Price	Amount
00010-100	Monitor	1	\$350.00	\$350.00
00010-200	Desk lamps	3	\$55.00	\$165.00
Terms and Conditions			<b>Total</b>	<b>\$515.00</b>
				\$0.00
				\$0.00
			Shipping Charge	
			<b>Grand Total</b>	<b>\$515.00</b>

**Note:** Because the in-memory XML data source that is created in this chapter contains limited data, not all of the fields in this form are prepopulated. To see this form completely filled in, see [“About dynamic forms” on page 79](#).

# 5

## Rendering Forms as HTML

This chapter explains how you can use the Form Server Module API to render forms that are displayed as HTML to client devices, typically web browsers. A form is rendered in response to a request made by a client device. For example, the Form Server Module can render a form that is displayed as HTML in response to an HTTP request that is initialized from a web browser.

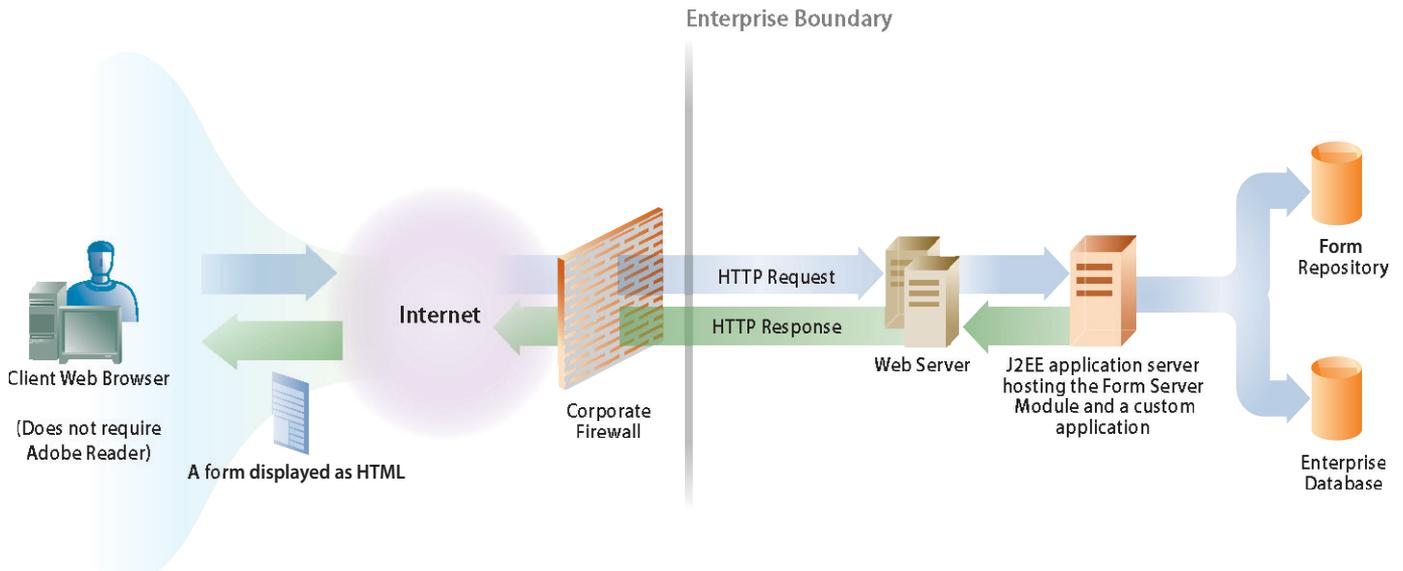
This chapter contains the following information.

Topic	Description	See
client applications rendering HTML forms	Describes the characteristics of a client application that renders HTML forms.	<a href="#">page 91</a>
Rendering a form as HTML	Covers the methods you can use to render a form to a client web browser.	<a href="#">page 92</a>

### Client applications rendering HTML forms

The Form Server Module can render forms in HTML format as opposed to PDF. A benefit of rendering a form as HTML is that the computer on which the client web browser is located does not require Adobe Reader or Acrobat. For information about LiveCycle Forms rendering forms as PDF, see [“Rendering Interactive Forms as PDF” on page 48](#).

Consider a client web browser sending an HTTP request to a client application requesting a form that is displayed as HTML. When the application receives the HTTP request, it sends the request to the Form Server Module, which then renders the form to the client web browser within an HTTP response. This process is shown in the following diagram.



**Note:** This diagram shows a client application and the LiveCycle Forms existing on the same J2EE application server. You can deploy a client application to a separate J2EE application server and remotely invoke the Form Server Module. For information, see [“Remotely invoking Form Server Module” on page 32](#).

## Form considerations

For the Form Server Module to render a form as HTML, you must save a form design as an .xdp file. If you attempt to render a form as HTML that is based on a form design saved as a .pdf file, you will cause a `RenderFormException`.

When developing a form design in LiveCycle Designer that will be rendered as HTML, consider the following criteria:

- Do not use an object's border properties to draw lines, boxes, or grids on your form. Some browsers may not line up borders exactly as they appear in a LiveCycle Designer preview. Objects may appear layered or may push other objects off their expected position.
- You can use lines, rectangles, and circles to define the background.
- Draw text slightly larger than what seems to be required to accommodate the text. Some web browsers do not display the text legibly.

For more information about creating form designs to render as HTML, see [“Designing form designs to render as HTML” on page 18](#).

## Rendering a form as HTML

Using the Form Server Module API, you create application logic to render a form that is displayed as HTML to a client web browser as part of an HTTP response. To render a form that is displayed as HTML, create an `EJBClient` object (or a `SOAPClient` object) and call its `renderForm` method. The `renderForm` method returns an `IOOutputContext` interface. You use this interface to populate a data stream with the form and then send the data stream to the client web browser within an HTTP response.

In addition to using the Form Server Module API, you also use standard Java classes. These classes enable you to perform necessary tasks, such as creating a data stream to send to the client web browser.

You specify the form design to render by setting the `renderForm` method's `sFormQuery` parameter. For information, see [“Specifying the form design to render” on page 53](#).

**Note:** Before you render a form, it is recommended that you are familiar with invoking the Form Server Module. For information, see [“Invoking the Form Server Module” on page 30](#).

## Setting preference options to render the form as HTML

You can render a form as HTML by using a form design that was created in LiveCycle Designer and saved as an .xdp file. Set the `renderForm` method's `sFormPreference` parameter to one of the following values:

- `MSDHTML`—Renders the form as dynamic HTML for Internet Explorer 5.0 or later
- `HTML4`—Renders the form as HTML compatible with older browsers that do not support absolute positioning of HTML elements.
- `AHTML`—Compatible with accessibility enhanced web browsers (currently Internet Explorer 5.0 or later).
- `StaticHTML`—Renders the form as read-only HTML. This option was added to LiveCycle Forms 7.1.
- `CSS2HTML`—Compatible with CSS2 specification.

These options are used to render the form as HTML. To view the `renderForm` method's `sFormPreference` values that are used to render the form as PDF, see ["Setting preference options to render the form as PDF" on page 53](#).

**Note:** AHTML and CSS2HTML are compliant with XHTML 1.0. For information, go to [www.w3.org/TR/xhtml1/](http://www.w3.org/TR/xhtml1/).

## Specifying the client applications web context

You set the `renderForm` method's `sApplicationWebRoot` with a string value that specifies the web context of the client application that is deployed to a J2EE application server. This value is combined with `sTargetURL` to construct an absolute URL to access application-specific web content. Typically, this value is a URL value, as shown in the following example:

```
String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp";
```

## Caching HTML forms

To cache an HTML form, assign the `renderForm` method's `sOptions` parameter with the following string value:

```
CacheEnabled=True
```

For more information, see ["Caching PDF forms" on page 54](#).

In addition to using the Form Server Module API, you must also enable form caching in LiveCycle Designer. For information see, *LiveCycle Designer Help*.

## Creating application logic to render a form as HTML

The following process describes how to render a form as HTML to a client web browser:

1. Create an `EJBClient` object by calling the `EJBClient` constructor. For information, see ["Locally invoking Form Server Module" on page 31](#).
2. Call the `EJBClient` object's `renderForm` method and set its parameters (this is shown in the example that follows this process. This method returns an `IOOutputContext` interface.
3. Create a Java `ServletOutputStream` object used to send a byte stream to the client web browser.
4. Set the Java `HttpServletResponse` object's content type to match the `IOOutputContext` object's content value. You can achieve this by calling the `HttpServletResponse` object's `setContentType` method and passing the `IOOutputContext` object's `getContentType` return value:

```
HttpServletResponseOb.setContentType(OutputContextOb.getContentType());
```

5. Create a byte array and populate it by calling the `IOOutputContext` object's `getOutputContent` method. This task assigns the content of the `IOOutputContext` object to a byte array. The following line of code shows this application logic:

```
byte[] cContent = OutputContextOb.getOutputContent();
```

6. Call the `HttpServletResponse` object's `write` method to send a data stream to the client web browser. Pass the byte array to the `write` method.

The following code example renders a form named Loan.xdp to a client web browser.

**Example 5.1 Rendering a form as HTML to a client web browser**

```
//Create a EJBClient object
EJBClient formServer = new EJBClient();

//Declare and populate local variables to pass to renderForm
String sFormQuery = "Loan.xdp"; //Defines the form design to render
String sFormPreference = "MSDHTML"; //Defines the preference option
String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
String sTargetURL = "http://<AppServer>:<AppPort>/LoanApp/HandleData";
String sApplicationWebRoot = "http://<AppServer>:<AppPort>/LoanApp";
byte[] cData = new byte[0]; //cData

try{
 //Call renderForm
 IOOutputContext myOutputContext = formServer.renderForm(
 sFormQuery, //sFormQuery
 sFormPreference, //sFormPreference
 cData, //cData,
 "CacheEnabled=true", //Enable caching
 null, //sUserAgent,
 sApplicationWebRoot, //sApplicationWebRoot
 sTargetURL, //sTargetURL
 sContentRootURI, //sContentRootURI
 null //sBaseURL
);

 // Create a ServletOutputStream object
 ServletOutputStream oOutput = resp.getOutputStream();

 //Set the HTTPResponse object's content type
 resp.setContentType(myOutputContext.getContentType());

 // Get the length of the output stream
 int outLength = myOutputContext.getOutputContent().length;

 //Create a byte array and allocate outLength bytes
 byte[] cContent = new byte[outLength];

 //Populate the byte array by invoking getOutputContext
 cContent = myOutputContext.getOutputContent();

 //Write a byte stream back to the web browser. Pass the byte array
 oOutput.write(cContent);
}
//Catch a thrown exception
catch (Exception ioEx)
{
 System.out.println("Exception error is: " +ioEx.getMessage());
}
}
```

**Note:** This example shows how to render a form by locally invoking the Form Server Module. You can use an EJBClient object to remotely invoke the Form Server Module. For information, see [“Remotely invoking Form Server Module” on page 32.](#)

# 6

## Calculating Form Data

This chapter explains how you can calculate data that is located in a form. Assume, for example, that a user enters values into an interactive form and clicks a calculate button. The Form Server Module can calculate the values and display the result in the form. To calculate form data, you must perform two tasks. First, you create a form design script that calculates form data. A form design supports three types of scripts. One script type runs on the client, another runs on the server, and the third type runs on both the server and the client. The script type discussed in this chapter runs on the server.

Second, you use the Form Server Module API to create a client application that handles a form design that contains an embedded script. The fact that a form design contains calculations does not affect how you use the Form Server Module API. It is important to note that the Form Server Module API does not actually calculate or manipulate data. It simply handles and returns a form that contains a script that performs calculations.

This chapter contains the following information.

Topic	Description	See
About form design scripts	Describes a form design that contains a script that runs on the server.	<a href="#">page 95</a>
Handling a form containing a script	Describes how to use the Form Server Module API to create application logic that handles a form containing a script.	<a href="#">page 96</a>

**Note:** It is recommended that you are familiar with using the Form Server Module API to render forms before you read this chapter. For information, see [“Rendering Interactive Forms as PDF” on page 48](#)

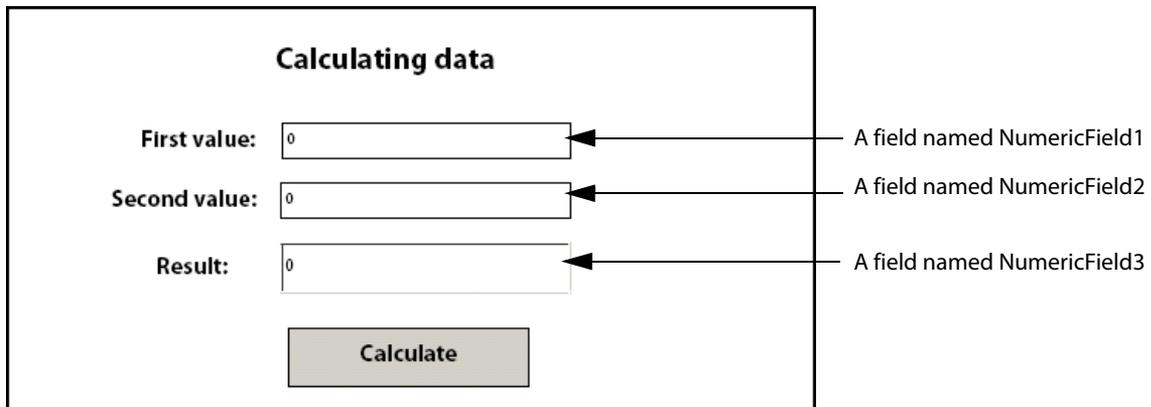
### About form design scripts

As part of the form design process, you can make use of calculations and scripts to provide a richer user experience. Calculations and scripts can be added to most form fields and objects. You must create a form design script to perform calculation operations on data that a user enters into an interactive form.

When data is submitted from a client web browser to LiveCycle Forms, it can be submitted as XML data or PDF data. The content type of data that is submitted as XML is `text/xml`. In contrast, the content type of data that is submitted as PDF is `application/pdf`. You use LiveCycle Designer to determine if data is submitted as PDF or XML. For information, see *LiveCycle Designer Help*.

Typically, a form that is submitted as PDF will contain scripts that are executed on the client. However, server-side calculations can also be executed without a problem. A Submit button cannot be used to calculate scripts. In this situation, calculations are not executed because the Form Server Module considers the interaction to be complete. For information, see [“Using Form Design buttons” on page 23](#).

To illustrate the usage of a form design script, this section examines a simple interactive form that contains a script that is configured to run on the server. The following diagram shows a form design containing a script that adds values that a user enters into the first two fields and displays the result in the third field.



The syntax of the script located in this form design is as follows:

```
NumericField3 = NumericField2 + NumericField1
```

In this form design, the Calculate button is a command button, and the script is located in this button's `Click` event. When a user enters values into the first two fields (NumericField1 and NumericField2) and clicks the Calculate button, the form is sent to the Form Server Module, where the script is executed. The Form Server Module renders the form back to the client device with the results of the calculation displayed in the NumericField3 field. For information about how the Form Server Module executes scripts, see ["Using Form Design buttons" on page 23](#).

For information about creating a form design script, see the *LiveCycle Designer Help*.

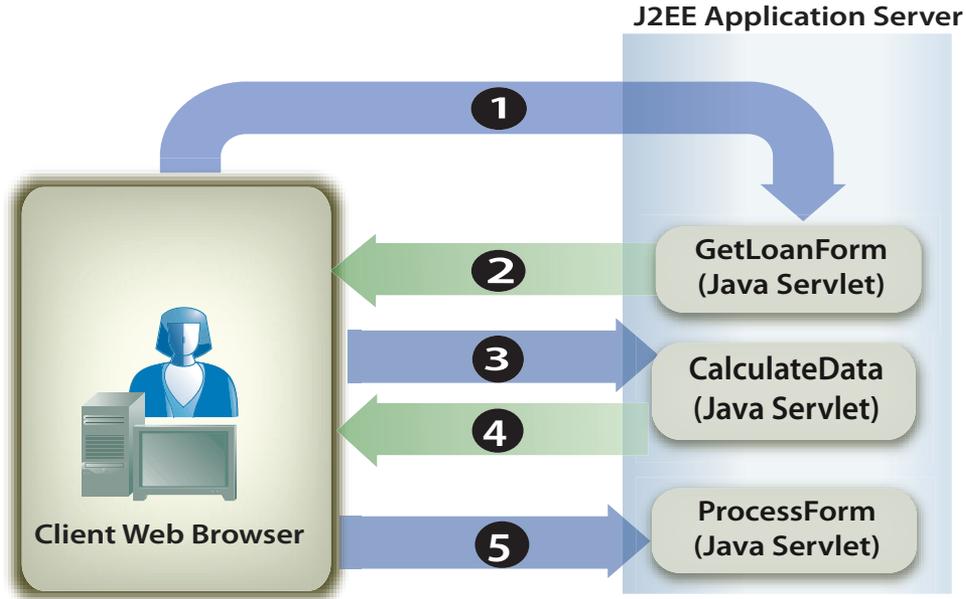
## Handling a form containing a script

You use the Form Server Module API to create application logic that handles a form design containing a script configured to run on the server. Consider a client application that lets a user fill in a form with data required to secure a loan. To assist the user, assume that the loan form contains a calculate button and fields that enable a user to enter data, such as an interest rate value and the number of months for which loan payments are made.

The user enters values into the form and clicks the Calculate button to view the results. The following process describes the application:

- The user accesses an HTML page named `StartLoan.html` that functions as the web application's start page. This page invokes a Java servlet named `GetLoanForm`.
- The `GetLoanForm` servlet renders a loan form. This form contains a script, interactive fields, a calculate button, and a submit button.
- The user enters values into the form's fields and clicks the Calculate button. The form is sent to the `CalculateData` Java servlet where the script is executed. The form is sent back to the user with the calculation results displayed in the form.
- The user continues entering and calculating values until a satisfactory result is displayed. When satisfied, the user clicks the Submit button to process the form. The form is sent to another Java servlet named `ProcessForm` that is responsible for retrieving submitted data. For information, see ["Retrieving submitted form data" on page 60](#).

The following diagram shows the application's logic flow.



The following table describes the steps in this diagram.

1	The GetLoanForm Java servlet is invoked from the StartLoan.html page. This page contains a link that invokes the GetLoanForm Java servlet.
2	The GetLoanForm Java servlet uses the Form Server Module API to render the loan form to the client web browser. For information, see <a href="#">"Rendering a form using an EJBClient object" on page 52.</a>
3	The user enters data into interactive fields and clicks the Calculate button. The form is sent to the CalculateData Java servlet, where the script is executed.
4	The form is rendered back to the web browser with the calculation results displayed in the form.
5	The user clicks the Submit button when the values are satisfactory. The form is sent to another Java servlet named ProcessForm.

## Rendering a form that contains a script

The difference between rendering a form that contains a script configured to run on the server and rendering a form that does not contain a script is that you must specify the target location used to execute the script. If a target location is not specified, a script that is configured to run on the server is not executed.

For example, consider the application introduced in this section. The CalculateData Java servlet is the target location where the script is executed.

To specify a target location, assign the `renderForm` method's `sTargetURL` parameter a value that specifies the target location. For more information about this parameter, see ["Specifying the target URL" on page 54.](#)

For information about rendering a form, see ["Creating application logic to render a form as PDF" on page 57.](#)

## Creating application logic to handle a form containing a calculation script

You use the Form Server Module API to create application logic that handles a form that contains a script configured to run on the server. Call the `processFormSubmission` method to retrieve the form. This method returns an `IOutputContext` interface that you use to retrieve the submitted form that contains a script.

You call the `IOutputContext` interface's `getFSAction` method to determine the processing state associated with a submitted form. When a form is submitted for a script to be calculated, the `getFSAction` method returns the value 1. In this situation, a script configured to run on the server is automatically executed.

After you verify that `getFSAction` returns 1, you can create application logic to render the form back to the client web browser. When the form is displayed, the calculated value will appear in the appropriate field(s).

### Example 6.1 Handling a form containing a calculation script

```
//Create an EJBClient object and an IOutputContext interface
EJBClient formServer = new EJBClient();
IOutputContext myOutputContext = null;

try{
 // Call processFormSubmission to handle the submitted data.Pass the
 // HttpServletRequest object
 myOutputContext = formServer.processFormSubmission(req, "OutputType=0");

 //Determine the processing state associated with the submitted form
 short fsAction = myOutputContext.getFSAction();

 if (fsAction == 1)
 {
 //Calculation results must be returned to the client web browser
 //Create a ServletOutputStream object
 ServletOutputStream oOutput = resp.getOutputStream();

 //Get the IOutputContext object's character set
 //Use this value to set the HttpServletResponse
 //object's content type
 String charset = myOutputContext.getCharSet();

 if (charset == null || charset.length() == 0)
 charset = "ISO88591";

 if (myOutputContext.getContentType().compareToIgnoreCase("text/html") ==
0)
 resp.setContentType (myOutputContext.getContentType() + "; charset=" +
charset);
 else
 resp.setContentType (myOutputContext.getContentType());
 }
}
```

```
//Create a byte array. Call the IOutputContext object's
//getOutputContext method
byte[] cContent = myOutputContext.getOutputContent();

//Write a byte stream back to the Web browser
oOutput.write(cContent);
}
}

catch (Exception e)
{
 e.printStackTrace();
}
```

This chapter explains how you can use the Form Server Module API to import and export data from fields located in an interactive form. This chapter also discusses how to manipulate form fields by converting them into page contents. This process is called *flattening*. After you flatten a form field, users cannot manipulate the field in Acrobat or Adobe Reader. That is, a user cannot enter data into a field after it is flattened. No operation is available to reverse this process.

Two types of PDF forms exist:

- An *Acrobat form* (created in Acrobat) is a PDF document that contains one or more form fields. The PDF document may also contain non-form content. An Acrobat form does not contain forms that conform to the XML Forms Architecture.
- An *XML form* (typically created in LiveCycle Designer) is a PDF document that conforms to the PDF 1.5 or 1.6 specification and contains form field information and possibly data that conforms to the XML Forms Architecture.

This chapter contains the following information.

Topic	Description	See
Importing form data	Explains how to import data into a form.	<a href="#">page 100</a>
Exporting form data	Explains how to export data from a form.	<a href="#">page 101</a>
Flattening form fields	Explains how to manipulate form fields by converting them into page contents.	<a href="#">page 102</a>

## Importing form data

You can use the Form Server Module API to import data into an Acrobat form. This data can be in one of these formats:

- FDF is the normal Acrobat form data format.
- XFDF is an XML version of FDF.

The following table shows which formats you can use for each type of form.

Import formats	Result	Form type	Export formats	Result
FDF & XFDF	Successful	Acrobat form	FDF & XFDF	Successful
FDF & XFDF	InvalidFormDataFormat exception	XML form	FDF & XFDF	InvalidFormFormat exception
FDF & XFDF	InvalidFormFormat exception	Not a form	FDF & XFDF	InvalidFormFormat exception

You can import FDF or XFDF data into an Acrobat form by creating an `EJBClient` object (or a `SOAPClient` object) and calling its `renderForm` method. Assign the following values to the `renderForm` method's parameters:

- `sFormQuery`—References the Acrobat form for which data is imported.
- `sFormPreference`—Must be set to `PDFMerge`.
- `cData`—Must be valid XFDF or FDF data as a byte array. For information, see [“Rendering prepopulated forms” on page 68](#).
- `sOptions`—No specific options are required.
- `sUserAgent`—This value is not required.
- `sApplicationWebRoot`—This value is not required.
- `sTargetURL`—The target for the form submission. For information, see [“Specifying the target URL” on page 54](#).
- `sContentRootURI`—The root URI where the source PDF is located.
- `sBaseURL`—This value is not required.

The `renderForm` method returns an `IOutputContent` interface that you use to retrieve the PDF that contains FDF or XFDF data. For example, you can use the `IOutputContent` interface to save the PDF as a PDF file. For information, see [“Saving a PDF document” on page 108](#).

## Exporting form data

You can export form data from an Acrobat or an XML form by using the Form Server Module API. You can export XML data from an Acrobat form (designed using Acrobat) or an XML form (designed using LiveCycle Designer) by calling the `EJBClient` (or `SOAPClient`) object's `processFormSubmission` method.

When a PDF is passed as the request buffer to the `processFormSubmission` method, with the option `PDFtoXDP=1`, the Form Server Module converts the PDF to its XML representation as XDP.

If the PDF is an Acrobat form, the XDP will contain an `xfdf` packet that will include the `xfdf` field data (fields) and annotations (annots). If the PDF is an XML form, the XDP will contain the data in the datasets packets and any annotations are contained in the `xfdf` packet.

To export form data, assign the following values to the `processFormSubmission` method's parameters:

- `cRequestBuffer`—The PDF that contains data to export. This PDF can be an Acrobat form or an XML form.
- `sEnvironmentBuffer`—Must include the header `CONTENT_TYPE=application/pdf`.
- `sUserAgent`—This value is not required and can be null.
- `sOptions`—The value `PDFtoXDP` must be specified.

The `processFormSubmission` method returns an `IOutputContent` interface that you use to retrieve data that was exported from the form. Call the `IOutputContent` interface's `getOutputContent` method to retrieve the form data. For information about retrieving data by using the `processFormSubmission` method, see [“Creating application logic to retrieve submitted data” on page 62](#).

## Flattening form fields

Using the Form Server Module API, you can flatten form fields. You can only flatten form fields; you cannot flatten other types of annotations. To flatten form fields, create an `EJBClient` object (or a `SOAPClient` object) and call its `renderForm` method. Assign the following values to the `renderForm` method's parameters:

- `sFormQuery`—The PDF that contains form fields to flatten.
- `sFormPreference`—Must be set to PDF.
- `cData`—Must be valid FDF or XFDF XML specified as a byte array or a `Document` object..
- `sOptions`—No specific options required.
- `sUserAgent`—This value is not required.
- `sApplicationWebRoot`—This value is not required.
- `sTargetURL`— This value is not required.
- `sContentRootURI`—The root URI where the source PDF is located.
- `sBaseUrl`—This value is not required.

The `renderForm` method returns an `IOutputContent` interface that you use to retrieve the PDF that contains flattened fields. The content type of PDF is `application/pdf`. Call the `IOutputContent` interface's `getContentType` method to retrieve the content type.

For information about using the `renderForm` method to render a form as PDF, see [“Rendering a form using an EJBClient object” on page 52](#).

This chapter explains how you can programmatically transfer PDF data between LiveCycle Forms and other Adobe LiveCycle products that use the PDF Manipulation Module API, such as Adobe LiveCycle Document Security or Adobe LiveCycle Reader Extensions. To transfer PDF data, you use the Form Server Module API, the PDF Manipulation Module API, and the Data Manager Module API. For information about LiveCycle Document Security, see the *LiveCycle Document Security Developer's Guide*.

A client application that uses the Form Server Module API, the Data Manager Module API and the PDF Manipulation Module API must be deployed on the J2EE application server hosting LiveCycle Forms and the other LiveCycle products. The PDF Manipulation Module cannot be invoked remotely.

This chapter also explains how to save a form that is submitted to LiveCycle Forms as a PDF file. To save a PDF file, you only need to use the Form Server Module API. Neither the Data Manager Module API nor the PDF Manipulation Module API are required.

This chapter contains the following information.

Topic	Description	See
About transferring data	Describes the process of transferring data from one LiveCycle module to another.	<a href="#">page 103</a>
Retrieving submitted PDF data	Describes how to retrieve PDF data that is submitted from a client web browser and transfer the PDF data from LiveCycle Forms to LiveCycle Document Security.	<a href="#">page 105</a>
Creating a PDF document	Describes how to create a PDF document from data that is transferred from the Form Server Module.	<a href="#">page 106</a>

It is recommended that you are familiar with using the Form Server Module API to render forms before you read this chapter. For information, see ["Rendering Interactive Forms as PDF" on page 48](#)

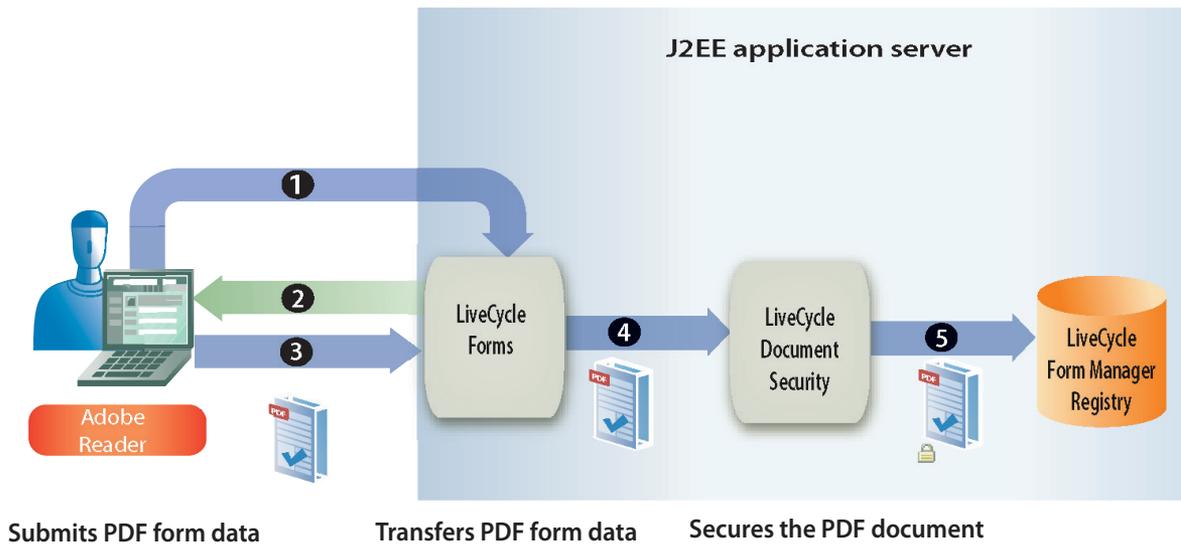
## About transferring data

A client application can consist of two or more Adobe LiveCycle APIs. Consider a client application that uses the Form Server Module API to invoke the Form Server Module and the PDF Manipulation Module API to invoke LiveCycle Document Security. Using these APIs, you can transfer PDF data from the LiveCycle Forms to LiveCycle Document Security.

Consider a web application that invokes the Form Server Module. After the Form Server Module renders an interactive form to a client web browser, the user fills in an interactive form and submits it back to the Form Server Module as PDF data. The option of submitting PDF data to LiveCycle Forms is specified in LiveCycle Designer. For information about rendering a form, see ["Rendering a form using an EJBClient object" on page 52](#).

When the LiveCycle Forms receives the PDF data, it sends the data to LiveCycle Document Security, which uses the PDF Manipulation Module API. LiveCycle Document Security can encrypt the PDF data, save it as a PDF document, and store the document in an enterprise data source or a content management repository.

The following diagram shows the application's logic flow.



The following table describes the steps in this diagram.

- |   |                                                                                                                                                                                                                 |
|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | A web page contains a link that accesses a Java servlet that invokes LiveCycle Forms.                                                                                                                           |
| 2 | LiveCycle Forms renders a form to the client web browser.                                                                                                                                                       |
| 3 | The user fills in an interactive form and clicks a submit button. The form is submitted back to LiveCycle Forms as PDF data. This option is set in LiveCycle Designer.                                          |
| 4 | LiveCycle Forms transfers the PDF data to LiveCycle Document Security, which uses the PDF Manipulation Module API.                                                                                              |
| 5 | The PDF Manipulation Module API is used to encrypt the PDF data and save it as a PDF document. The client application uses a separate API to store the secured PDF document in a content management repository. |

## Form design considerations

When data is submitted from a client web browser to LiveCycle Forms, it can be submitted as XML or PDF data. For LiveCycle Forms to transfer data to LiveCycle Document Security, the data must be submitted as PDF data. If data is submitted as XML, it cannot be transferred to the PDF Manipulation Module. The content type of submitted PDF data is `application/pdf`. In contrast, the content type of submitted XML data is `text/xml`.

The form design must be configured correctly in LiveCycle Designer in order to submit PDF data. To properly configure the form design to submit PDF data, ensure that the Submit button that is located on the form design is configured to submit PDF data. For information about configuring the Submit button, see the *LiveCycle Designer Help*.

**Note:** A form design can be saved as a PDF file or an XDP file. The important factor is that the Submit button is configured to submit PDF data, not whether the form design is saved as a PDF file or an XDP file.

## Retrieving submitted PDF data

After a user fills in an online form and submits PDF data, your client application must retrieve the PDF data before it can transfer it to LiveCycle Document Security.

The Form Server Module API contains methods that enable you to retrieve PDF data submitted from a form. The `IFormServer` interface's `processFormSubmission` method must be called to retrieve submitted data. This method returns an `IOutputContext` interface that you can use to retrieve the submitted data.

To transfer PDF data from LiveCycle Forms to LiveCycle Document Security, you must write the PDF data to a temporary file. Before you populate the temporary file with the PDF data, ensure that the submitted data is PDF data. To perform this task, call the `IOutputContext` interface's `getContentType` method. This method returns a string value that specifies the content type of the submitted data. If the submitted data is PDF data, the return value is `application/pdf`.

Create a temporary file by instantiating a Java `File` object by using its public constructor. Populate the `File` object with the submitted PDF data. To perform this task, call the `IOutputContext` interface's `getOutputContent` method. This method returns a byte array that contains the PDF data.

Write the byte array contents to the temporary file by using a Java `FileOutputStream` object. Call this object's `write` method and pass the byte array. The PDF Manipulation API references this temporary file to create a `PDFDocument` object. For information, see ["Creating a PDF document" on page 106](#).

### Example 8.1 Retrieving submitted PDF data

```
//Create an EJBClient object, which implements IFormServer
EJBClient formServer = new EJBClient();

// Call processFormSubmission to handle the submitted PDF data. Pass the
// HttpServletRequest object
IOutputContext outputContext =
formServer.processFormSubmission(req, "OutputType=0");

//Determine the content type -- make sure it is application/pdf
String ct = outputContext.getContentType();

if (ct.equals("application/pdf"))
{
 //Get the binary PDF data
 byte [] formOutput = outputContext.getOutputContent();

 //Create a temporary File object
 File tempFile = new File("C:\\tempData.dat");

 //Create a Java FileOutputStream object
 FileOutputStream myOutput = new FileOutputStream(tempFile) ;

 //Write the byte array contents to the file
 myOutput.write(formOutput);
 myOutput.close();
}
```

**Note:** There are a few points to consider about the previous code example. First, assume that this code example is located in a Java servlet method that has an `HttpServletRequest` object as an argument. This object is passed to the `processFormSubmission` method. Second, this code example is located within a `try` statement and has corresponding `catch` statements. For simplicity, these statements are removed. Third, the temporary file is named `tempData.dat` and is stored in the root of `C:\`.

## Creating a PDF document

You can create a `PDFDocument` object from a temporary file containing PDF data by using the Data Manager Module API and the PDF Manipulation Module API. After you create a `PDFDocument` object, you can perform PDF manipulation tasks, such as encrypting the PDF document. For information, see the *LiveCycle Document Security Developer's Guide*.

To create a `PDFDocument` object, it is necessary to create a `FileDataBuffer` object. The `FileDataBuffer` interface belongs to the Data Manager Module API. To create a `FileDataBuffer` object, call the `DataManager` object's `createFileDataBuffer` method (or `createFileDataBufferFromUrl`). For information about the `FileDataBuffer` interface, see the *Form Server Module API Reference*.

In addition to using a `DataManager` object, you use two standard Java classes, `InitialContext` and `PortableRemoteObject`, to perform a Java JNDI look-up operation. Using these classes, you create a CORBA object representing a connection to the PDF Manipulation service (part of LiveCycle Document Security).

You can create a `PDFDocument` object by performing the following programmatic tasks within a Java project:

1. Create a `DataManager` object (the import statements required to create this object are required to create a `PDFManipulation` object). The `DataManager` object used in this example is named `mDataManager`. For information, see ["Invoking the Data Manager Module" on page 40](#).

2. Create an `InitialContext` object by using the `InitialContext` constructor:

```
InitialContext pdfnamingContext = new InitialContext();
```

3. Perform a JNDI look-up by calling the `InitialContext` object's `lookup` method and pass the string `PDFManipulation` as an argument. Store the return value in an `Object` variable. The following line of code shows this application logic:

```
Object pdfObject = namingContext.lookup("PDFManipulation");
```

4. Create a `ConnectionFactory` object by calling the `PortableRemoteObject` object's static `narrow` method. This method determines if the return value of the look-up method can be cast to a `ConnectionFactory` object. Cast the return value to `ConnectionFactory`. The following line of code shows this application logic:

```
ConnectionFactory pdfConnectionFactory = (ConnectionFactory)
PortableRemoteObject.narrow(pdfObject, ConnectionFactory.class);
```

5. Create a PDFFactory object by calling the PDFFactoryHelper object's static narrow method (it is unnecessary to instantiate a PDFFactoryHelper object). Pass the ConnectionFactory object to this method and call its getConnection method. Cast the return value to org.omg.CORBA.Object. The following line of code shows this application logic:

```
PDFFactory mPDFFactory =
 PDFFactoryHelper.narrow((org.omg.CORBA.Object)pdfConnectionFactory.
 getConnection());
```

6. Create a FileDataBuffer object by calling the DataManager object's createFileDataBuffer method. Reference the temporary data file that contains PDF data (this file was created in the previous section). The following line of code shows this application logic:

```
FileDataBuffer pdfFile =
 mDataManager.createFileDataBuffer("C:\\tempData.dat");
```

7. Create a PDFDocument object by calling the PDFFactory object's openPDF method and pass the FileDataBuffer object:

```
PDFDocument pdf = mPDFFactory.openPDF(pdfFile);
```

The following code example shows how to create a PDFDocument object by using a DataManager object named mDataManager.

### **Example 8.2** *Creating a PDFDocument object using a temporary file containing PDF data*

```
//Declare a ConnectionFactory object
ConnectionFactory pdfConnectionFactory = null;

//Lookup the PDF Manipulation service
Object pdfObject = namingContext.lookup("PDFManipulation");
pdfConnectionFactory = (ConnectionFactory)
 PortableRemoteObject.narrow(pdfObject, ConnectionFactory.class);

//Use the pdfConnectionFactory object to create a PDFFactory object
PDFFactory mPDFFactory =
 PDFFactoryHelper.narrow((org.omg.CORBA.Object)pdfConnectionFactory.
 getConnection());

//Create a FileDataBuffer object to store an existing PDF document by
//calling the DataManager object's createFileDataBuffer method.
//Reference the tempData.dat file
FileDataBuffer pdfFile =
 mDataManager.createFileDataBuffer("C:\\tempData.dat");

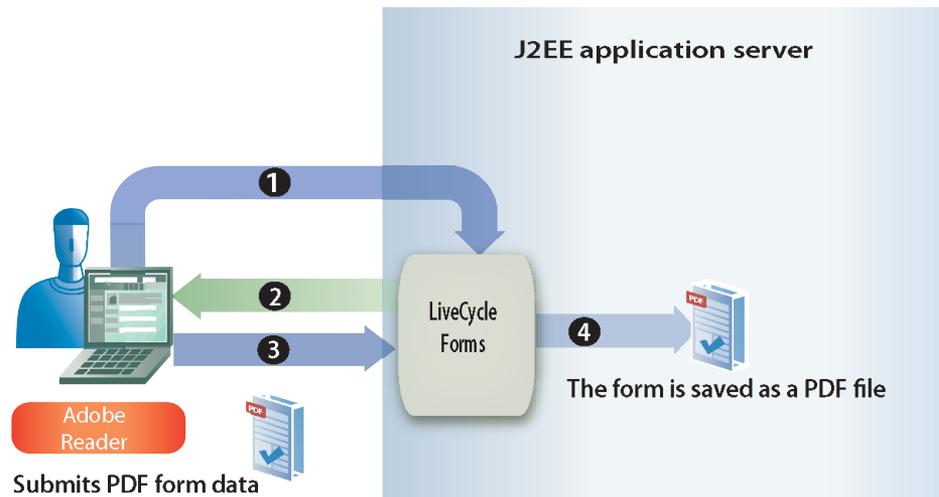
//Create a PDFDocument object by using the PDFFactory object's
//openPDF method and passing the FileDataBuffer object
PDFDocument pdf = mPDFFactory.openPDF(pdfFile);
```

For more information about creating a FileDataBuffer object, see ["Invoking the Data Manager Module" on page 40](#).

## Saving a PDF document

Instead of transferring PDF data to another LiveCycle product, you can save PDF data as a PDF file. For example, consider the application introduced earlier in this chapter. When LiveCycle Forms receives a form that is submitted as PDF data, the form can be saved as a PDF file. You can then view the form using Acrobat or Adobe Reader.

The following diagram shows the application's logic flow.



The following table describes the steps in this diagram.

- |   |                                                                                                                                                                        |
|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | A web page contains a link that accesses a Java servlet that invokes LiveCycle Forms.                                                                                  |
| 2 | LiveCycle Forms renders a form to the client web browser.                                                                                                              |
| 3 | The user fills in an interactive form and clicks a submit button. The form is submitted back to LiveCycle Forms as PDF data. This option is set in LiveCycle Designer. |
| 4 | LiveCycle Forms saves the PDF data as a PDF file.                                                                                                                      |

For LiveCycle Forms to save the PDF data as a PDF file, the data must be submitted as PDF data. If a form is submitted as XML, it cannot be saved as a PDF file. The content type of submitted PDF data is `application/pdf`. In contrast, the content type of submitted XML data is `text/xml`.

The Form Server Module API contains methods that enable you to retrieve PDF data submitted from a form. The `IFormServer` interface's `processFormSubmission` method must be called. This method returns an `IOutputContext` interface that you can use to retrieve the submitted data.

To save PDF data that is submitted to LiveCycle Forms, you write the PDF data to a Java `File` object. Before you write PDF data to the Java `File` object, ensure that the submitted data is PDF data. To perform this task, call the `IOutputContext` interface's `getContentType` method. This method returns a string value that specifies the content type of the submitted data. If the submitted data is PDF data, the return value is `application/pdf`.

Create a Java `File` object by using its public constructor. Be sure to specify `.pdf` as the file name extension. Populate the `File` object with the submitted PDF data. To perform this task, call the `IOutputContext` interface's `getOutputContent` method. This method returns a byte array that contains the PDF data.

Write the byte array contents to the Java File by using a Java FileOutputStream object. Call this object's write method and pass the byte array.

### Example 8.3 Saving a PDF document

```
//Create an EJBClient object, which implements IFormServer
EJBClient formServer = new EJBClient();

// Call processFormSubmission to handle the submitted PDF data. Pass the
// HttpServletRequest object
IOutputContext outputContext =
formServer.processFormSubmission(req, "OutputType=0");

//Determine the content type -- make sure it is application/pdf
String ct = outputContext.getContentType();

if (ct.equals("application/pdf"))
{
 //Get the binary PDF data
 byte [] formOutput = outputContext.getOutputContent();

 //Create a pdf File object
 File tempFile = new File("C:\\myPDF.pdf");

 //Create a Java FileOutputStream object
 FileOutputStream myOutput = new FileOutputStream(tempFile) ;

 //Write the byte array contents to the file
 myOutput.write(formOutput);
 myOutput.close();
}
```

**Note:** There are a couple of points to consider about the previous code example. First, assume that this code example is located in a Java servlet method that has an HttpServletRequest object as an argument. This object is passed to the processFormSubmission method. Second, this code example is located within a try statement and has corresponding catch statements. For simplicity, these statements are removed. Third, the PDF file is named my PDF.pdf and is stored in the root of C:\.

# 9

## Authenticating Users

This chapter explains how you can use the User Manager API to programmatically authenticate users with Adobe User Management. User Manager allows administrators to maintain a database for all users and groups, synchronized with one or more third-party user directories. User Management provides authorization and user management for LiveCycle Forms.

This chapter contains the following information.

Topic	Description	See
About user authentication	Explains user authentication	<a href="#">page 110</a>
Performing user authentication	Explains how to programmatically authenticate users	<a href="#">page 111</a>

**Note:** It is also possible to create custom service providers for User Management. For information about creating custom service providers, see the *Developing User Management Service Providers* guide.

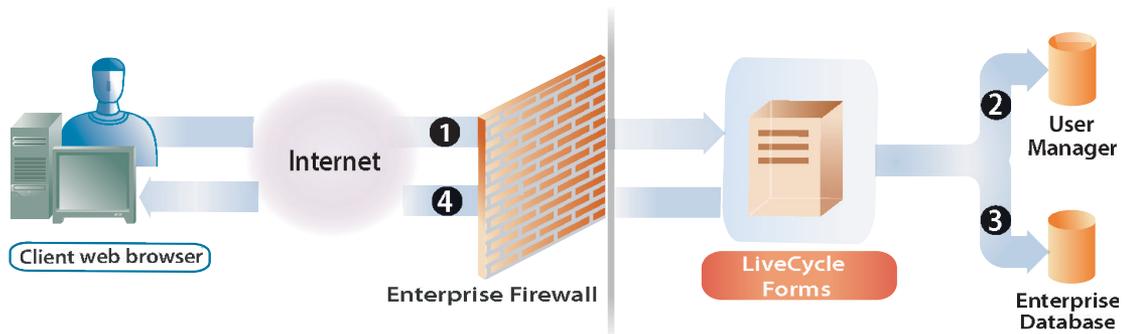
### About user authentication

Using the User Manager API, you can enable your LiveCycle Forms client application to authenticate users with User Manager. User authentication may be required to interact with an enterprise database or other enterprise repositories that store secure data.

Consider, for example, a scenario where a user enters a user name and password into a web page and submits the values to a J2EE application server hosting both LiveCycle Forms and a client application. A LiveCycle Forms client application can authenticate the user with User Manager.

If the authentication is successful, the application accesses a secured enterprise database. Otherwise, a message is sent to the user stating that the user is not an authorized user.

The following diagram shows the application's logic flow.



The following table describes the steps in this diagram:

---

1	The user accesses a web site and specifies a user name and password. This information is submitted to a J2EE application server hosting LiveCycle Forms.
2	The user credentials are authenticated with User Manager. If the user credentials are valid, the workflow proceeds to step 3. Otherwise, a message is sent to the user stating that the user is not an authorized user.
3	User information and a form design are retrieved from a secured enterprise database.
4	User information is merged with a form design and the form is rendered to the user.

---

## Performing user authentication

You must use the User Manager API to authenticate a user with User Manager.

To use the User Manager API, import the `um-client.jar` file into your Java project. The location of this JAR file is dependant upon the J2EE application server on which LiveCycle Forms is installed. For example, if LiveCycle Forms is installed on JBoss, you must use the `um-client.jar` file that is located in the `jboss` folder.

The `um-client.jar` file can be located in one of the following folders:

- `C:\Adobe\LiveCycle\components\um\jboss\lib\adobe\um-client.jar`
- `C:\Adobe\LiveCycle\components\um\websphere\lib\adobe\um-client.jar`
- `C:\Adobe\LiveCycle\components\um\weblogic\lib\adobe\um-client.jar`

where `C:\` is the drive on which you installed LiveCycle Forms. For information, see the *Installing and Configuring* guide for your application server.

## Programmatically authenticating a user

To authenticate a user, you must create a `UserManager` object and invoke its `authenticate` method. This method requires the following arguments:

- A string value that represents the user name
- A byte array that represents the password

The `authenticate` method returns an `AuthResult` object if the authentication is successful. The fully qualified name of this object is `com.adobe.idp.um.api.infomodel.AuthResult`. If the authentication is unsuccessful, this method throws an exception.

You must also create a `com.adobe.idp.Context` object by using its constructor. Do not confuse this `Context` object with the `javax.naming.Context` object.

The `com.adobe.idp.Context` object is required because you must invoke its `initPrincipal` method and pass the `AuthResult` object that was returned by the `authenticate` method. The `initPrincipal` method sets the `com.adobe.idp.Context` object with the authentication results that are stored in the `AuthResult` object.

**Note:** You cannot include both the `com.adobe.idp.Context` import statement and the `javax.naming.Context` import statement in your Java project. The `javax.naming.Context` object does not contain an `initPrincipal` method.

## Setting the LiveCycle Forms invocation context

You can set the invocation content that is used by the Form Server Module by calling the `IFormServer` interface's `setInvocationContext` method and passing the `com.adobe.idp.Context` object. The Form Server Module will forward this context as necessary when making requests for content from other repositories.

For example, assume a secured enterprise database requires a user context before letting a client application have access to it. In this scenario, the Form Server Module forwards the context that is set by invoking the `setInvocationContext` method.

## Creating application logic to authenticate users

You can programmatically authenticate users by performing the following tasks in a Java project:

1. Add the `um-client.jar` to your Java project's class path.
2. Add the following import statements to your Java project:

```
// UserManager API import statements
import com.adobe.idp.Context;
import com.adobe.idp.um.*;
import com.adobe.idp.um.api.infomodel.AuthResult;
```
3. Create an `InitialContext` object by using the `InitialContext` constructor.
4. Perform a JNDI look-up by invoking the `InitialContext` object's `lookup` method and pass the string `com.adobe.idp.um.UserManagerHome` as an argument. Store the return value in a `UserManagerHome` object variable and ensure that you cast the return value to `UserManagerHome`.
5. Create a `UserManager` object by invoking the `UserManagerHome` object's `create` method.
6. Authenticate a user by invoking the `UserManager` object's `authenticate` method. Pass a user name and password to this method and store the return value in an `AuthResult` object.
7. Create a `com.adobe.idp.Context` object by using its constructor.
8. Invoke the `Context` object's `initPrincipal` method and pass the `AuthResult` object that was returned by the `authenticate` method.

The following example authenticates a user with User Manager.

**Example 9.1 Authenticating a user with User Manager**

```
// Declare objects
Context ctx = null;
UserManagerHome userManagerHome = null;
UserManager userManager = null;
InitialContext initialContext = null ;

try
{
 // Allocate memory to an InitialContext object
 initialContext = new InitialContext();

 //Perform a lookup to User Manager
 userManagerHome =
(UserManagerHome) initialContext.lookup("com.adobe.idp.um.UserManagerHome") ;

 // Allocate memory to the UserManager object by invoking the UserManagerHome
 //object's create method
 userManager = userManagerHome.create();

 // Call the authenticate method
 AuthResult res = userManager.authenticate("<username>",
"<Password>".getBytes());

 // Allocate memory to the com.adobe.idp.Context object
 ctx = new Context();

 // Invoke initPrincipal
 ctx.initPrincipal(res);

 // Set the Form Server Module's invocation context
 IFormServerOb.setInvocationContext(cxt);
}

catch (Exception e)
{
 System.out.println("The exception is "+ e.getMessage());
}
```

**Note:** For information about invoking the Form Server Module, see ["Invoking the Form Server Module" on page 30](#).

# 10 Rendering Forms from .NET

This chapter explains the programming details required to render PDF forms using Microsoft Visual Studio .NET 2003. The discussion is based on a nearly identical process of rendering interactive forms to the one discussed and illustrated in [“Rendering Interactive Forms as PDF” on page 48](#).

The code examples displayed in this chapter show how to locally invoke the Form Server Module using a `FSSoapClient` object referenced in an ASP .NET project using Visual C# .NET. You will need to add references to `SOAPClient.dll` and `ICSharpCode.SharpZipLib.dll`, and your source file should contain the following statement referencing the `SoapClient` namespace:

```
using SoapClient;
```

For information, see [“Invoking Form Server Module using the Microsoft .NET client assembly” on page 36](#).

This chapter contains the following information.

Topic	Description	See
Form-based applications	Describes the characteristics of a form-based application.	<a href="#">page 114</a>
Rendering PDF forms	Covers the methods you can use to render a form to a client web browser.	<a href="#">page 115</a>
Retrieving submitted data	Covers the methods you can use to retrieve data submitted from a form.	<a href="#">page 117</a>
Rendering prepopulated forms	Covers the methods you can use to prepopulate a form prior to rendering it.	<a href="#">page 119</a>

## Client applications rendering PDF forms

The Form Server Module is stateless and runs on a J2EE application server and is only accessible through the Form Server Module API. A client application that uses the Form Server Module API is able to invoke the Form Server Module and instruct it to perform tasks such as rendering forms, processing submitted data, and prepopulating forms with data.

A client application that uses the Form Server Module API is able to retrieve the data that is submitted with a form. For example, when a user fills in a form and submits it, a client application can retrieve the data that was entered in the form's fields by the user. The client application can then process the data in a variety of ways, such as performing calculations, storing it in an enterprise database, or sending it to another application, such as an application that authorizes credit cards.

The Form Server Module can prepopulate a form prior to rendering it. Prepopulating a form involves inserting data into a form. For example, a client application can query data from a database and instruct the Form Server Module to insert the data into a form and then render the form. When the form is rendered to a web browser, the user is able to view the data in the displayed form.

Using the Form Server Module API, you can create different types of client applications that interact with the Form Server Module, such as ASP .NET pages. This chapter discusses creating ASP .NET web applications that can be deployed on an IIS server.

Consider a client web browser sending an HTTP request to a client application on an IIS-based web server requesting a form. When the web application receives the HTTP request, it sends the request to the Form Server Module, which then renders the form back to the client web browser within an HTTP response.

## Rendering a form using the Microsoft .NET client assembly

Using the Microsoft .NET client assembly, you create application logic to render a form to a client web browser as part of an HTTP response. A form must be rendered before a user can interact with it.

To render a form, you create a `FSSoapClient` object and call its `renderForm` method. The `renderForm` method returns an `IOutputContext` interface. You use this interface to populate a data stream to send to the client web browser with the form.

In addition to using the Microsoft .NET client assembly, you also use standard .NET Framework Class Library (FCL) classes. These classes enable you to perform necessary tasks, such as creating a data stream to send to the client web browser.

The following process describes how to render a form to a client web browser by locally invoking the Form Server Module:

1. Create an `FSSoapClient` object by calling the `FSSoapClient` constructor:

```
FSSoapClient formServer = new FSSoapClient();
```

2. Set the SOAP endpoint by calling the `FSSoapClient` object's `setSoapEndPoint` method:

```
formServer.setSoapEndPoint("http://<AppServerURL>:8080/jboss_net/
services/AdobeFSService");
```

3. Call the `FSSoapClient` object's `renderForm` method and sets its parameters (this is shown in the example that follows this process). This method returns an `IOutputContext` interface. (You can get the `IOutputContext` interface's content type and its byte stream. A typical usage involves using Visual C# .NET classes to send the byte stream to the client web browser, which is shown in the example that follows this process).

4. Create a byte array and populate it by calling the `IOutputContent` object's `getOutputContent` method. This task assigns the content of the `IOutputContent` object to a byte array. The following line of code shows this application logic:

```
byte[] cContent = myOutputContext.getOutputContent();
```

5. Call the `Response` object's `BinaryWrite` method to send a data stream to the client web browser. Pass the byte array to the `BinaryWrite` method:

```
Response.BinaryWrite(cContent);
```

The following code example renders a form named Loan.xdp to a client web browser:

**Example 10.1 Rendering a form to a client web browser**

```
public void PageLoad(object sender, System.EventArgs e){

 //Create an FSSoapClient object
 FSSoapClient formServer = new FSSoapClient();
 formServer.setSoapEndPoint("http://<AppServerURL>:8080/jboss_net/services/
AdobeFSService");

 //Declare and populate local variables to pass to renderForm
 String sFormQuery = "Loan.xdp"; //Defines the form design to render
 String sFormPreference = "PDFForm"; //Defines the preference option
 String sContentRootURI = "http://<AppServer>:<AppPort>/LoanApp/forms";
 String sTargetURL = "http://<AppServer>:<AppPort>/LoanApp/HandleData"
 String sApplicationWebRoot = "http://<AppServer:<AppPort>/LoanApp";

 try{
 //Call renderForm
 IOutputContext myOutputContext = formServer.renderForm(
 sFormQuery, //sFormQuery
 sFormPreference, //sFormPreference
 null, //cData,
 "CacheEnabled=False", //sOptions
 null, //sUserAgent,
 sApplicationWebRoot, //sApplicationWebRoot
 sTargetURL, //sTargetURL
 sContentRootURI, //sContentRootURI
 null //sBaseURL
);

 //Create a byte array. Call the IOutputContext interface's
 //getOutputContext method
 byte[] cContent = myOutputContext.getOutputContent();

 //Write a byte stream back to the web browser. Pass the byte array
 Response.BinaryWrite(cContent);
 }
 //Catch a thrown exception
 catch (Exception ex)
 {
 byte[] bArray = (new System.Text.ASCIIEncoding()).GetBytes(ex.Message);
 Response.BinaryWrite(bArray);
 }
}
```

## Retrieving submitted data

The Form Server Module API contains methods that enable you to retrieve data submitted from a form. The `processFormSubmission` method must be called to retrieve submitted data. This method returns an `IOutputContext` interface that you can use to retrieve the submitted data.

To retrieve data from the `IOutputContext` interface, you convert the `IOutputContext` interface's content to an XML data source. After you perform this task, you can retrieve data from the XML data source by using the following FCL DOM classes (available within the `System.Xml` namespace):

- `XmlDocument`—Used to create an object that represents an entire XML document.
- `XmlElement`—Used to create an object that represents a node within an XML document.
- `XmlAttributeCollection`—Used to iterate through the attributes of an `XmlElement`.
- `XmlNode`—Used to create an object that represents a single node within an XML document.
- `MemoryStream`—Used to transfer the byte array to the DOM.

To use these classes in your project, add the following `using` statements:

- `using System.Xml;`
- `using System.IO;`

The following process describes how to create application logic to retrieve data from a form:

1. Create an `FSSoapClient` object by calling the `FSSoapClient` constructor.
2. Call the `FSSoapClient` object's `processFormSubmission` method. This method returns an `IOutputContext` interface that contains the data submitted from the form.
3. Create a byte array by calling the `IOutputContext` interface's `getOutputContent` method.
4. Create a `MemoryStream` object by passing the byte array into its constructor.
5. Create an `XmlDocument` object and call its `Load` method using the `MemoryStream` object as a parameter. You now have the DOM.
6. Retrieve the value of each node within the XML document. One way to accomplish this is to create a custom method that accepts two parameters: the `XmlDocument` object and the name of the node whose value you want to retrieve. This method returns a string value representing the value of the node. In the code example that follows this process, this custom method is called `getNodeText`. The body of this method is shown.
7. Calls the `getNodeText` method for each field from which to retrieve a value. For example, to retrieve all fields in the loan form, you must call `getNodeText` 13 times.

The following code example shows the application logic that retrieves data submitted from a form.

**Example 10.2 Retrieving data from a form**

```
public void PageLoad(object sender, System.EventArgs e){

 try{

 // Call processFormSubmission to handle the submitted data. Pass the
 // Request object
 myOutputContext =
 formServer.processFormSubmission(Request, "OutputType=0");

 //Populate a byte array by calling IOutputContext object's
 //getOutContent method
 byte [] formOutput = myOutputContext.getOutputContent();

 //Create a MemoryStream object
 MemoryStream myMemoryStream = new MemoryStream(formOutput);

 // Create an XmlDocument object
 XmlDocument myDOM = new XmlDocument();

 // Load the XML data into the XmlDocument object:
 myDOM.Load(myMemoryStream);

 // Call getNodeText for each field in the form
 String myAmount = getNodeText("Amount", myDOM);
 String myLastName = getNodeText("LastName", myDOM);
 String myFirstName = getNodeText("FirstName", myDOM);
 String mySSN = getNodeText("SSN", myDOM);
 String myTitle = getNodeText("PositionTitle", myDOM);
 String myAddress = getNodeText("Address", myDOM);
 String myCity = getNodeText("City", myDOM);
 String myStateProv = getNodeText("StateProv", myDOM);
 String myZipCode = getNodeText("ZipCode", myDOM);
 String myEmail = getNodeText("Email", myDOM);
 String myPhoneNum = getNodeText("PhoneNum", myDOM);
 String myFaxNum = getNodeText("FaxNum", myDOM);
 String myDescription= getNodeText("Description", myDOM);
 }//End of try statement

 catch (processFormSubmissionException e)
 {
 System.out.println("The process form submission error is: " +e);
 }

 catch (Exception ioEx)
 {
 System.out.println("Exception error is: " +ioEx);
 }
}
```

```
// Create the getNodeText custom method
private String getNodeText(String nodeName, XmlDocument myDOM)
{
 //Get the node by name. nodeName is the name of the
 //node passed to this method
 XmlNodeList nl = myDOM.GetElementsByTagName(nodeName);
 XmlNode myNode = nl.Item(0);

 return myNode.InnerText;
} //End of getNodeText
```

## Rendering prepopulated forms

A client application can prepopulate a form with data prior to rendering it. The data can come from a variety of sources, such as an enterprise database, another form, or another application. Prepopulating a form has several advantages:

- Enables the user to view custom data in a form
- Reduces the amount of typing the user does to fill in a form
- Ensures data integrity by having control over where data is placed

Using the Form Server Module API, you can create a client application capable of prepopulating a form. Consider the loan sample application introduced in a previous chapter. For information, see [“Sample loan application” on page 49](#).

After data is submitted to the HandleData web application, a confirmation form is rendered back to the web browser. This form contains data that the user entered into the loan application.

The following table explains the steps in this diagram.

1	The HandleData web application prepopulates the confirmation form with data.
2	The confirmation form is rendered to the client web browser.
3	The confirmation form is displayed in the client web browser.

**Note:** Prepopulating a form is also known as merging data with a form.

## Creating application logic to render a prepopulated form

You must assign a byte array to the `renderForm` method's `cData` parameter to prepopulate a form prior to rendering it. This byte array represents an XML data source containing fields located in the form. For each field that you want to prepopulate, you must specify a value. It is not necessary to match the exact structure of the XML document. For example, to prepopulate the confirmation form, specify a value for the `LastName`, `FirstName`, and `Amount` fields.

Assume that a form containing 10 fields has data in 4 of the fields. Next, assume that you want to prepopulate the remaining 6 fields. In this situation, you must specify 10 XML elements in the XML data source used to prepopulate the form. If you specify only 6 elements, the original 4 fields will be empty.

Here are three ways in which you can create a byte array to assign to the `cData` parameter:

- Convert an existing XML document containing data to merge to a byte stream.
- Assign a string representing an XML document to a byte array. To convert a string to a byte array, create a `System.Text.ASCIIEncoding` object. Then invoke its `GetBytes` method, which receives the string as a parameter.
- Create an `XmlDocument` object and use its `InnerXml` property to retrieve a string representation, which can be converted to a byte array. For information, see [“Converting the XML data source to a byte array” on page 87](#).

The method you choose depends on your preference. However, you would typically use the third method when prepopulating either a multipage form containing many fields or a dynamic form. For information about prepopulating a dynamic form, see [“Rendering Dynamic Forms” on page 79](#).

# 11

## Rendering Forms using the XML Form Module API

You use the XML Form Module API to develop applications capable of rendering forms as PDF documents. For example, you can create an application that retrieves data from an enterprise database, merges it with a form design that is created in LiveCycle Designer, and renders the form as a PDF document. The application can render a separate PDF document for each enterprise database record.

You cannot use the XML Form Module to render forms to a client web browser. To perform this task, you must use the Form Server Module API. After you render a form using the XML Form Module, you can send it to a printer. However, to print forms, it is recommended that you use LiveCycle Print.

**Caution:** The XML Form Module API is deprecated. As a result, it is recommended that you use the Form Server Module API.

This chapter contains the following information.

Topic	Description	See
Creating a Form object	Describes how to create a Form object using the <code>create</code> method.	<a href="#">page 121</a>
Importing packets	Describes how to import data packets.	<a href="#">page 124</a>
Setting configuration values	Describes how to set configuration values.	<a href="#">page 125</a>
Rendering PDF documents	Explains how to render PDF documents.	<a href="#">page 126</a>

### Creating a Form object

When creating a `Form` object, you use the Connection API and two standard Java classes, `InitialContext` and `PortableRemoteObject`, to perform a JNDI look-up and return a CORBA object that represents a connection to the XML Form service. For information about the Connection API, see the “Connection API” chapter in the *XML Form Module API Reference*.

The XML Form Module API is a transaction-based API, which means a `Form` object must be created within a transaction. Using the Java `UserTransaction` class, you can create a `UserTransaction` object. Call the `UserTransaction` object’s `begin` method to start the transaction and its `commit` method to complete the transaction.

You can use one of two ways to create a `Form` object. Both ways involve creating a `FormFactory` object. This object has a `create` method and a `createDefault` method; you can use either one to create a `Form` object. This chapter describes creating a `Form` object by using the `create` method.

#### Default.xci file

When calling the `create` method, you reference an XML configuration file named `default.xci`. This file is placed on the J2EE application server on which LiveCycle Forms is deployed. The file location is dependent

on which J2EE application server and operating system you are using. This file is placed in the following location when LiveCycle Forms is deployed on WebSphere that is located on Windows:

```
C:\Program Files\WebSphere\AppServer\installedApps\adobe\server1\XMLFormService
```

By referencing this configuration file, you can use the `Form` object to set and change configuration values. For information, see [“Setting configuration values” on page 125](#).

When calling the `FormFactory` object’s `createDefault` method, you use the default values specified in the configuration file. For information about creating a `Form` object by calling the `createDefault` method, see [“Invoking the XML Form service” on page 45](#).

## Creating application logic to create a Form object

Specify these values when calling the `create` method:

- The default configuration file.
- The `config` packet. For information about packets, see [“Importing packets” on page 124](#).

You create a `Form` object by performing the following programmatic tasks:

1. Add the `xmlform-client.jar` file to your Java project’s build path. For information about the location of this file, see [“Including LiveCycle Forms library files” on page 28](#).

2. Add the following import statements to your Java project:

```
import com.adobe.document.xmlform.Form;
import com.adobe.document.xmlform.FormFactory;
import com.adobe.document.xmlform.FormFactoryHelper;
```

3. Create a `DataManager` object. The import statements required to create this object are required to create a `Form` object. For information, see [“Invoking the Data Manager Module” on page 40](#).

4. Create an `InitialContext` object by using the `InitialContext` constructor:

```
InitialContext xmlNamingContext = new InitialContext();
```

5. Perform a JNDI look-up by calling the `InitialContext` object’s `lookup` method and passing the string `XMLFormService` as an argument. Store the return value in an `Object` variable:

```
Object xmlObject = xmlNamingContext.lookup("XMLFormService");
```

6. Create a `ConnectionFactory` object by calling the `PortableRemoteObject` object’s `narrow` method. This method determines if the return value of the `lookup` method can be cast to a `ConnectionFactory` object. Cast the return value to `ConnectionFactory`:

```
ConnectionFactory xmlConnectionFactory = (ConnectionFactory)
PortableRemoteObject.narrow(xmlObject, ConnectionFactory.class);
```

7. Create a `FormFactory` object by calling the `FormFactoryHelper` object’s `narrow` method (it is unnecessary to instantiate a `FormFactoryHelper` object). Pass the `ConnectionFactory` object to this method and call its `getConnection` method. Cast the return value to

`org.omg.CORBA.Object`:

```
FormFactory mFormFactory =
FormFactoryHelper.narrow((org.omg.CORBA.Object)xmlConnection
Factory.getConnection());
```

8. Create a `FileDataBuffer` object that stores the default.xci configuration file. Call the `DataManager` object's `createFileDataBuffer` method and pass the location of the configuration file:

```
FileDataBuffer configFile =
mDataManager.createFileDataBuffer("C:\\Program
Files\\WebSphere\\AppServer\\installedApps\\adobe\\server1\\XMLForm
Service\\default.xci");
```

9. Create a string array with one element. This string array stores the names of packets that are imported. Because you need to specify the config packet when calling the `create` method, assign the value `config` to the string array:

```
String pList[] = new String[1];
List[0] = "config";
```

10. Create a `Form` object by calling the `FormFactory` object's `create` method. Pass the `FileDataBuffer` object representing the configuration file and the string array storing the packet name:

```
Form myForm = mFormFactory.create(configFile,pList);
```

The following example shows how to create a `Form` object.

### **Example 11.1 Creating a Form object by using the FormFactory object's create method**

```
//Declare a ConnectionFactory object
ConnectionFactory xmlConnectionFactory = null;

// Lookup the XMLForm service
Object xmlObject = namingContext.lookup("XMLFormService");

//Create a ConnectionFactory object
xmlConnectionFactory = (ConnectionFactory)
PortableRemoteObject.narrow(xmlObject,ConnectionFactory.class);

//Use the xmlConnectionFactory object to create a FormFactory object
FormFactory mFormFactory =
FormFactoryHelper.narrow((org.omg.CORBA.Object)xmlConnection
Factory.getConnection());

// Create a FileDataBuffer object by referencing the default.xci file
FileDataBuffer configFile = mDataManager.createFileDataBuffer("C:\\Program
Files\\WebSphere\\AppServer\\installedApps\\adobe\\server1\\XMLForm
Service\\default.xci");

//Create a string array to store all packet names
String pList[] = new String[1];
pList[0] = "config";

// Create a Form object by calling create
Form myForm = mFormFactory.create(configFile,pList);
```

## Importing packets

In LiveCycle Designer, a form author typically saves the form design for use with LiveCycle Forms as an XDP file. XDP is an XML format that provides a mechanism for packaging units of content (known as XDP packets) within a surrounding XML container.

When the form author saves the form design, the resulting XDP file typically contains XDP packets that define the form design and related configuration information. Using the XML Form Module API, you can import packets. Typically, you import these packets:

- `template`—Encloses the XML form design definition created in LiveCycle Designer. This packet specifies the form that is rendered as a PDF document.
- `datasets`—Encloses XML form data that originates from an XML form and may be intended for merging with an XML form design definition.

To import these packets, you call the `Form` object's `importPackets` method, which requires the following parameters:

- The name of the packet to import
- A `FileDataBuffer` object representing the data or form design to load

### Importing a template packet

To import a `template` packet, you specify `template` as the name of the packet and the path to the form design file. You need to create a `FileDataBuffer` object, which belongs to the Data Manager Module API, to reference the form design file. To populate this object, call the `DataManager` object's `createFileDataBuffer` (or `createFileDataBufferFormURL`) method and pass the path to the form design file.

The following example shows how to import a `template` packet:

```
FileDataBuffer formDefinitionFile =
mDataManager.createFileDataBuffer("C:\\po.xdp");
pList[0] = "template";
myForm.importPackets(formDefinitionFile, pList);
```

### Importing a datasets packet

To import a `datasets` packet, you specify `datasets` as the name of the packet and the XML file that contains the data to merge with the form design file. You need to create a `FileDataBuffer` object to reference the XML file. To populate the `FileDataBuffer`, call the `DataManager` object's `createFileDataBuffer` method and pass the path to the XML file.

The following example shows how to import a `datasets` packet:

```
FileDataBuffer xmlFile = mDataManager.createFileDataBuffer("C:\\po.xml");
pList[0] = "datasets";
myForm.importPackets(xmlFile, pList);
```

## Determining if a packet exists

Using the XML Form Module API, you can determine if a specific packet exists by calling the `Form` object's `isPacketPresent` method. This method returns a boolean value indicating whether the packet exists.

The following example shows how to determine whether the `template` packet exists:

```
boolean templatePresent = myForm.isPacketPresent("template");
if (templatePresent)
 pw.println("The template packet is present");
else
 pw.println("The template packet is not present");
```

## Setting configuration values

The XML Forms Architecture leverages XML for the representation of all information and incorporates XML architectural concepts such as Document Object Models (DOMs). One such DOM is the Configuration DOM.

The Configuration DOM provides a mechanism for specifying configuration options for the XML Form Module. The XML Form Module provides a default configuration file (`default.xci`). This file is loaded when the `Form` object is instantiated by using the `create` method. For information, see [“Creating a Form object” on page 121](#).

Using the XML Form Module API, you can set the `destination` and `pdf.xdc.uri` configuration values, which are nodes in the Configuration DOM. To set a configuration value, you call the `Form` object's `setConfigValue` method and specify the following parameters:

- The expression that specifies a Configuration DOM node for which a value is set
- The value to assign to the node

### Setting the destination configuration value

The `destination` configuration value specifies the output format when rendering documents. The only acceptable value is `pdf`. The following example shows how to set the `destination` configuration value:

```
myForm.setConfigValue("destination", "pdf");
```

### Setting the pdf.xdc.uri configuration value

The `pdf.xdc.uri` configuration value specifies a path to an XDC file named `acrobat7.xdc`. This file contains information such as the available fonts and must be referenced to successfully render a PDF document. This file is placed on the J2EE application server on which LiveCycle Forms is deployed. The file location is dependent on which J2EE application server and operating system you are using. This file is placed in the following location when LiveCycle Forms is deployed on WebSphere that is located on Windows:

```
C:\Program Files\WebSphere\AppServer\installedApps\adobe\server1\XMLFormService\bin
```

The following example shows how to set the `pdf.xdc.uri` configuration value:

```
myForm.setConfigValue("pdf.xdc.uri", "C:\\Program
Files\\WebSphere\\AppServer\\installedApps\\adobe\\server1\\XMLFormService
\\bin\\acrobat7.xdc");
```

For information about deploying the XML Form Module, see the *Installing and Configuring guide* for your application server.

For a complete list of the configuration values that you can set, see the “XML Form Module API” chapter in the XML Form Module *API Reference*.

## Determining a configuration value

Using the XML Form Module API, you can determine the value of a Configuration DOM node by calling the `Form` object’s `getConfigValue` method. This method requires an expression that specifies a node in the Configuration DOM from which a value is retrieved. The value of the node is returned as a `String` value. The following example shows how to get the value of `pdf.xdc.uri`:

```
String myConfigValue = myForm.getConfigValue("pdf.xdc.uri");
```

## Rendering PDF documents

After you create a `Form` object, import data packets, and set configuration values, you can render a PDF document. The data contained in the XML document that is defined by the `datasets` packet is merged with the form design that is specified by the `template` packet.

To render a PDF document, call the `Form` object’s `render` method. This method returns a `FileDataBuffer` object representing the PDF document. You can save the content of this object as a PDF document. For information about the `FileDataBuffer` object, see the “Data Manager Module API” chapter in the XML Form Module *API Reference*.

You can render a PDF document by performing the following programmatic tasks:

1. Create a `Form` object. For information, see [page 121](#).
2. Import data packets. For information, see [page 124](#).
3. Set configuration values. For information, see [page 125](#).
4. Create a `ReturnStatusHolder` object to pass to the `render` method.
5. Populate a `FileDataBuffer` object by calling the `Form` object’s `render` method. Pass the `ReturnStatusHolder` object as an argument.
6. Get the byte size of the `FileDataBuffer` object by calling its `getBufLength` method. This method returns a long value representing the size, in bytes, of the `FileDataBuffer` object.
7. Get the data contents of the `FileDataBuffer` object by calling its `getBytes` method. This method returns an array of bytes representing the data within the `FileDataBuffer` object. Store this method’s return value in a byte array variable.
8. Create a Java `File` object by using the `File` constructor. Because this file represents the saved PDF document, ensure that the file name extension is `.pdf`.

9. Create a `FileOutputStream` object by using its constructor. This object is used to write the byte array to the PDF file.
10. Call the `FileOutputStream` object's `write` method and pass the byte array returned from the `FileDataBuffer` object's `getBytes` method.
11. Close the `FileOutputStream` object by calling its `write` method.

The following example shows how to render and save a PDF document.

**Example 11.2 Rendering a PDF document by using the Form object's render method**

```
//Declare a ConnectionFactory object
ConnectionFactory xmlConnectionFactory = null;

// Lookup the XMLForm service
Object xmlObject = namingContext.lookup("XMLFormService");

//Create a ConnectionFactory object
xmlConnectionFactory = (ConnectionFactory)
 PortableRemoteObject.narrow(xmlObject, ConnectionFactory.class);

//Use the xmlConnectionFactory object to create a FormFactory object
FormFactory mFormFactory =
 FormFactoryHelper.narrow((org.omg.CORBA.Object)xmlConnection
 Factory.getConnection());

// Create a FileDataBuffer object and reference the default.xci file
FileDataBuffer configFile = mDataManager.createFileDataBuffer("C:\\Program
Files\\WebSphere\\AppServer\\installedApps\\adobe\\server1\\XMLForm
Service\\default.xci");

//Create a string array to store packet names
String pList[] = new String[1];
pList[0] = "config";

// Create a Form object by calling create
Form myForm = mFormFactory.create(configFile,pList);

//Import a template packet
FileDataBuffer formDefinitionFile =
 mDataManager.createFileDataBuffer("C:\\po.xdp");
pList[0] = "template";
myForm.importPackets(formDefinitionFile,pList);

//Import a datasets packet
FileDataBuffer xmlFile = mDataManager.createFileDataBuffer("C:\\po.xml");
pList[0] = "datasets";
myForm.importPackets(xmlFile,pList);

//Set the destination configuration value
myForm.setConfigValue("destination", "pdf");
```

```
//Set the pdf.xdc.uri configuration value
myForm.setConfigValue("pdf.xdc.uri", "C:\\Program
Files\\WebSphere\\AppServer\\installedApps\\adobe\\server1\\XMLFormService\\
bin\\acrobat7.xdc");

//Create a ReturnStatusHolder object
ReturnStatusHolder rs = new
 ReturnStatusHolder(ReturnStatus.XFA_RENDER_FAILURE);

//Call the render method
FileDataBuffer myRenderedPDF = myForm.render(rs);

//Get the byte size of the FileDataBuffer object
long pdfSize = myRenderedPDF.getBufLength();

//Get the DataBuffer object's data contents. Specify 0
// as the starting position and pdfSize as the length
byte [] pdfData = myRenderedPDF.getBytes(0, pdfSize);

//Create a PDF file named Test.pdf and place it in the root of C:\
File myPDFFile = new File("C:\\Test.pdf");

//Create a FileOutputStream object
OutputStream myFileW = new FileOutputStream(myPDFFile);

//Call the FileOutputStream object's write method and pass the pdf data
myFileW.write(pdfData);

//Close the FileOutputStream object
myFileW.close();
```

**Note:** This example assumes that a `DataManager` object exists and that this entire code fragment is located within a transaction block. For information about this object, see [“Invoking the DataManager Module” on page 40](#).

# A

## Character Sets and Unicode Encodings

Each module supports data, forms, and other content that use specific character sets and encodings of those character sets.

A character set defines a set of characters or glyphs. An encoding describes how those characters are represented using a series of bits. A character is encoded either through its original encoding (single or multibyte encoding) or a Unicode encoding.

The following Unicode encodings are supported:

- UTF-8
- UTF-16BE (Big Endian)
- UTF-16LE (Little Endian)

This table lists the character sets supported in original encoding and through Unicode encoding.

Original and Unicode encoding support	Unicode encoding support
Big5 (Traditional Chinese)	ISO-2022-KR (Korean)
Big5-HKSCS (Traditional Chinese Big5 with Hong Kong SCS)	ISO-8859-5 (Cyrillic)
GBK (Simplified Chinese)	ISO-8859-9 (Turkish)
ISO-2022 (Japanese)	windows-949 (Korean)
ISO-2022-JP-2 (Japanese)	windows-1251 (Cyrillic)
ISO-8859-1 (West European)	windows-1254 (Turkish)
ISO-8859-2 (East European)	
ISO-8859-7 (Greek)	
Shift_JIS (Japanese)	
windows-932 (Japanese Shift-JIS)	
windows-936 (Simplified Chinese GBK)	
windows-950 (Traditional Chinese Big5)	
windows-1250 (Central Europe)	
windows-1252 (Latin I)	
windows-1253 (Greek)	
JIS X 0212	

For a reference of the characters included in the Microsoft Windows character sets, go to [www.microsoft.com/globaldev/reference/WinCP.msp](http://www.microsoft.com/globaldev/reference/WinCP.msp). For a reference of the characters included in the remaining character sets, go to <http://oss.software.ibm.com/cgi-bin/icu/convexp>.

# B

## Language and Locale Combinations

Each module handles form fields and data that are localized to a wide variety of languages and locales.

This table lists the specific language and locale combinations.

<b>Language</b>	<b>Locale ID</b>
Bulgarian (Bulgaria)	bg_BG
Chinese, Simplified (P.R.C.)	zh_CN
Chinese, Traditional (Taiwan)	zh_TW
Chinese, Traditional with HKSCS-2001 Extensions (Hong Kong)	zh_HK
Croatian (Republic of Croatia)	hr_HR
Czech (Czech Republic)	cs_CZ
Danish (Denmark)	da_DK
Dutch (Belgium)	nl_BE
Dutch (Netherlands)	nl_NL
English (Australia)	en_AU
English (Canada)	en_CA
English (India)	en_IN
English (Ireland)	en_IE
English (New Zealand)	en_NZ
English (South Africa)	en_ZA
English (United Kingdom)	en_GB
English (United Kingdom, Euro Currency)	en_GB@Euro
English (United States)	en_US
Finnish (Finland)	fi_FI
French (Belgium)	fr_BE
French (Canada)	fr_CA
French (France)	fr_FR

<b>Language</b>	<b>Locale ID</b>
French (Luxembourg)	fr_LU
French (Switzerland)	fr_CH
German (Austria)	de_AT
German (Germany)	de_DE
German (Luxembourg)	de_LU
German (Switzerland)	de_CH
Greek (Greece)	el_GR
Hungarian (Hungary)	hu_HU
Italian (Italy)	it_IT
Italian (Switzerland)	it_CH
Japanese (Japan)	ja_JP
Korean (Korea)	ko_KR
Norwegian (Norway)	no_NO
Norwegian (Norway, Nynorsk)	no_NO_NY
Polish (Poland)	pl_PL
Portuguese (Brazil)	pt_BR
Portuguese (Portugal)	pt_PT
Romanian (Romania)	ro_RO
Russian (Russia)	ru_RU
Serbo-Croatian (Bosnia and Herzegovina)	sh_BA
Serbo-Croatian (Croatia)	sh_HR
Serbo-Croatian (Republic of Serbia and Montenegro)	sh_CS
Slovak (Slovak Republic)	sk_SK
Slovenian (Republic of Slovenia)	sl_SL
Spanish (Argentina)	es_AR
Spanish (Bolivia)	es_BO
Spanish (Chile)	es_CL

---

<b>Language</b>	<b>Locale ID</b>
Spanish (Colombia)	es_CO
Spanish (Costa Rica)	es_CR
Spanish (Dominican Republic)	es_DO
Spanish (Ecuador)	es_EC
Spanish (El Salvador)	es_SV
Spanish (Guatemala)	es_GT
Spanish (Honduras)	es_HN
Spanish (Mexico)	es_MX
Spanish (Nicaragua)	es_NI
Spanish (Panama)	es_PA
Spanish (Paraguay)	es_PY
Spanish (Peru)	es_PE
Spanish (Puerto Rico)	es_PR
Spanish (Spain)	es_ES
Spanish (Uruguay)	es_UY
Spanish (Venezuela)	es_VE
Swedish (Sweden)	sv_SE
Turkish (Turkey)	tr_TR

---

# Glossary

---

This glossary contains terminology definitions that are specific to documentation for the Adobe LiveCycle suite of products. These terms may have different meanings in other contexts, but they have restricted meanings in this documentation.

## A

### accessible forms

Forms that users with disabilities or vision impairment can view and fill using screen readers and other assistive technologies. See also *tagged Adobe PDF form*.

### Acrobat form

A PDF document, created in Acrobat, that contains one or more form fields. The PDF document may also contain non-form content.

### action

In a workflow, the representation of a step in a business process.

### Adobe certified document

A document that is signed with a specific Adobe root certificate. An Adobe certified document provides a strong guarantee as to the authenticity and immutability of the document. See also *certificate*.

### Adobe document services

Adobe document services extend the value of core enterprise systems to ensure more secure, reliable, and efficient use of business-critical information across the extended enterprise. Adobe document services include the Adobe LiveCycle suite of products and the Acrobat product line.

### application

A set of generally interdependent files that make up a self-contained application that Adobe LiveCycle products can run. Applications may include files such as form designs, Java Server Pages, HTML pages, servlets, and images.

## B

### branch

A branch contains a set of actions interconnected by routes, representing a sequential path taken by a process at execution. The branch always determines the behavior of the workflow.

## C

### certificate

An electronic file that establishes your identity, by binding your identity to your public key, when doing business or other transactions on the web. A *certificate* (or sometimes called a *digital certificate*) is issued by a certificate authority (CA). See also *Adobe certified document*.

### client

The requesting program in a client/server relationship. A web browser is an example of a client application.

### credential

The file that contains a private key. (The corresponding public key is contained in a certificate.) A private key is what one principal presents to another used to establish identity in decryption and signing operations. Credentials are issued by an authentication agent or a certification authority. See also *certificate*.

## D

### deadline

The time by which a person must complete a work item. Deadlines are properties of workflows.

### dynamic form

A form that can expand or contract to reflect the amount of incoming data. See also *interactive form*.

## E

### ebXML

Electronic Business using eXtensible Markup Language (ebXML). A modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. See also *registry*.

### encryption

The conversion of data into a format (called a ciphertext) that cannot be easily understood by unauthorized persons. The conversion is done using an encryption algorithm.

## F

### form

An electronic document that captures and delivers data. A person may add data to an interactive form, or a server process may merge a form design with data to produce a form.

### form authors

LiveCycle Designer users who are capable of creating fillable forms to be used in Acrobat or Adobe Reader, and simple non-interactive forms for deployment to LiveCycle Forms. See also *form developers*.

### FormCalc

A calculation language similar to that used in common spreadsheet software that facilitates form design without requiring a knowledge of traditional scripting techniques or languages.

### form design

The design-time version of a form that an author or developer creates in LiveCycle Designer.

### form developers

LiveCycle Designer users who are capable of creating complex form-based applications for use in different environments. See also *form author*.

### form object

A form element, such as a button or text field, that you can place on a form. An object has its own set of properties and events.

## I

### interactive form

A form that a person can interact with and complete electronically.

## N

### non-interactive form

A form that a person can view or print but cannot fill electronically. Non-interactive forms can be merged or prepopulated with data, but the data cannot be changed by a user. Non-interactive forms are designed for output.

## P

### PDF document

Portable Document Format. A file conforming to the PDF specification as published by Adobe Systems or a file conforming to the XDP specification, containing exactly one PDF packet and no more than one each XFA-Template, XFA-Configuration, XFA-SourceSet, and Annotations packets.

### PDF form

A form that users can access in Acrobat and Adobe Reader. PDF forms are either interactive or non-interactive.

### permissions

Security settings applied, for example, to restrict users from opening, editing, printing, or removing encryption from a PDF file. Permissions cannot be changed unless the user has the Permissions password. Permissions can be set in LiveCycle Designer, Acrobat, LiveCycle Document Security, and other products.

## policy

Defines a set of security permissions and users who can access a PDF document to which the policy is applied. Policies are created using LiveCycle Policy Server and can be applied to documents using LiveCycle Policy Server, LiveCycle Document Security, or Acrobat 7.0 or later.

## prepopulated form

A form that appears to the user with some or all fields automatically populated with data.

## Q

### QPAC

Quick Process Action Component. A JAR file that contains server-side code and client-side code for use with LiveCycle Workflow. In LiveCycle Workflow Designer, QPACs provide action components that can be added to workflows to represent a step in a process. LiveCycle Workflow Server interprets each action of the workflow and executes the server-side code of the corresponding QPACs. QPACs enable LiveCycle Workflow to interact with other Adobe LiveCycle products, such as LiveCycle Forms and LiveCycle Barcoded Forms.

## R

### reminder

A notification sent to people that reminds them to complete a work item. Reminders are properties of workflows.

### render

An action whereby LiveCycle Forms merges a form design, possibly with data, to display a form in PDF or HTML format in a browser.

### registry

An ebXML-compliant repository of shared information that provides services for the purpose of enabling business process integration between interested parties. See also *ebXML* and *repository*.

## repository

The underlying storage area within a registry. See also *registry*.

### restricted document

A PDF document with password security restrictions (permissions) that prevent the document from being opened, printed, or edited.

### rights-enabled document

A PDF document that includes security extensions that enable Adobe Reader users to fill forms, add comments, and sign documents.

### route

The path between actions on a workflow. Routes determine the order in which LiveCycle Workflow Server executes actions at run time.

### run time

For form rendering, the time when an application or server process retrieves a form design, possibly merges it with data, and presents it to a user for viewing or filling.

## S

### split

A segment in a workflow that contains one or more branches. The branches in a split are executed in parallel.

### static form

A form that remains exactly as it was designed. The layout does not change according to the amount of incoming data.

### subform

An object that can act as a container for form objects and other subforms. A subform helps to position form objects relative to each other and provide structure in dynamic form designs. A subform can also provide a reference point, when binding data to a form, by restricting the scope for a field so that it matches that of the corresponding data node.

## T

### **tagged Adobe PDF form**

Includes a logical structure and a set of defined relationships and dependencies among the various elements, plus additional information that permits reflow. See also *accessible forms*.

### **turnkey**

An installation option that automatically installs and configures the LiveCycle product files, JBoss application server, and MySQL database, and deploys the product files to JBoss. After you perform a turnkey installation, the LiveCycle product is ready to use.

## U

### **usage rights**

Rights that extend the functionality of Adobe Reader and enable users to save forms with data, add comments, and sign documents.

## W

### **workflow**

The electronic representation of a business process. Workflows are created using LiveCycle Workflow Designer.

## X

### **XDP file**

XML Data Package. LiveCycle Designer saves form designs as either XDP files or PDF files. LiveCycle Forms uses XDP files to render forms in PDF or HTML format.

### **XML Forms Architecture**

Represents the underlying technology beneath the Adobe XML forms solution. It enables the construction of robust and flexible form-based applications for use on either the client or the server.

### **XML form**

A PDF form that conforms to the Adobe PDF specification and the Adobe XML Forms Architecture. XML forms are typically created in LiveCycle Designer. XML forms can have the file name extension *.xdp* or *.pdf*.

# Index

## A

Acrobat forms 100  
APIs  
    Form Server Module 30  
    XML Form Module 45  
authenticating users 110  
axis.jar file 29

## B

begin method 40

## C

caching forms  
    client-side caching 55  
    server-side caching 54  
calculating data 48, 92, 114  
client-side caching 55  
client-side subform controls 17  
commons-discovery.jar file 29  
commons-logging.jar file 29  
configuration values, setting 125  
Connection API 40  
content type  
    converting 66  
    form considerations 60  
converting  
    form fields to text 102  
    XML document to byte array 69  
    XML string to byte array 71  
converting content type 66  
create method 38  
createDefault method 45  
creating  
    application logic to render forms 57, 83, 93  
    application logic to retrieve submitted data 62  
    byte array to render prepopulated form 68, 119  
    DataManager object 40  
    EJBClient object (local) 31  
    EJBClient object (remote) 32  
    form designs 17  
    Form object 45  
    Form objects 121  
    Microsoft.NET client object 36  
    SOAPClient object 34

## D

Data Manager Module 11  
Data Manager Module API 40  
datamanager-client.jar file 28  
datasets packet, importing 124  
default.xci file 121  
DMUtils object 42

Document object 43, 62  
DocumentBuilder object 62  
DocumentBuilderFactory object 62  
dynamic forms 79

## E

EJBClient class 30  
EJBClient object  
    using to invoke the Form Server Module 34  
    using to render forms 57  
event timing 19  
examples  
    converting an XML data source to a byte array 88, 89  
    converting an XML document to a byte array 70  
    converting an XML string to a byte array 71  
    creating a DataManager object 42, 113  
    creating a Form object using createDefault 47  
    creating a Form object using the create method 123  
    creating a PDFDocument using a temporary file 107  
    creating a SOAPClient object using FormServerFactory class 38  
    creating an EJBClient object using FormServerFactory class 39,  
    40  
    dynamically creating an in-memory XML data source 84  
    handling a form containing a calculation script. 98  
    invoking Form Server Module using EJBClient object 34  
    invoking Form Server Module using SOAPClient class 36  
    prepopulating a dynamic form 84  
    prepopulating a form 70, 71  
    rendering a form to a client web browser 94, 116  
    rendering a form using a SOAPClient object 58  
    rendering a form using an EJBClient object 57  
    rendering a PDF document 127  
    rendering a prepopulated form 89  
    retrieving submitted data from a form 64, 118  
    retrieving submitted PDF data 105  
    saving a PDF document 109  
exportDataFormat run-time option 66  
exporting form data 101

## F

files  
    datamanager-client.jar 28  
    formserver-client.jar 28, 32  
    SoapClient.dll 36  
    xmlform-client.jar 28  
flattening form fields 102  
form design buttons 20  
form design scripts 18  
form designs  
    about creating 16  
    creating 17  
    passing by value 77

- form fields
  - See also* forms
  - about 100, 110
  - flattening 102
  - importing and exporting data 100, 110
- form rendering options 53, 92
- form server 92
- Form Server Module
  - API. *See* APIs
  - invoking 30
  - invoking locally 31
  - invoking remotely 32
  - invoking through SOAP 34
- form-based applications 48, 92, 114
- forms
  - about rendering 48, 92, 114
  - design to render 53
  - rendering interactive 52, 92, 115
  - rendering prepopulated 68, 119
  - sample loan application 49
  - setting preferences 53, 92
  - submitted data 60, 117
- formserver-client.jar file 28
- FormServerFactory class, create method 38

**G**  
getOutputContent method 57, 93

**H**  
HTML

- creating form designs 18
- output 16
- rendering 91

HTTP request and response 48, 92, 114

**I**  
ICSharp utility 37  
import statements 87, 88  
importing

- form data 100, 110
- XDP packets 124

InitialContext object 40  
invoking

- Data Manager service 40
- Form Server Module 30
- Form Server Module locally 31
- Form Server Module remotely 32
- Form Server Module using SOAP 34
- XML Form service 45

IOutputContext interface 57, 93

**J**  
j2ee.jar file 29  
JAR files 28  
Java import statements. *See* import statements  
Java JNDI look-up 40

Java servlets

- about 52
- creating to render forms 57, 83, 93
- creating to retrieve submitted data 62

jaxrpc.jar file 29

**L**  
library files 28

**M**  
merging data 68, 119  
methods

- begin 40
- create 38
- createDefault 45
- getOutputContent 57, 93
- newInstance 62
- parse 62
- processFormSubmission 62
- renderForm 57, 93
- setContentType 57, 93
- setInitialContext 32
- setSOAPEndPoint 35

**N**  
newInstance method 62  
Node object 62  
NodeList object 62

**O**  
objects

- creating a Form object 121
- DataManager 40
- Document 62
- DocumentBuilder 62
- DocumentBuilderFactory 62
- EJBClient (local) 31
- EJBClient (remote) 32
- Form 45
- Form Server client (Microsoft .NET client object) 36
- Node 62
- NodeList 62
- SOAPClient 34

**P**  
parse method 62  
passing a form design by value 77  
PDF documents, rendering 126  
PDFManipulationAPI.jar file 28  
planning applications 16  
PortableRemoteObject object 40  
posting data

- merging data 68, 119
- retrieving 60, 117
- specifying target URL 54

prepopulating forms 68, 119  
processFormSubmission method 62  
processing a submitted form 60, 117

## R

- renderForm method 57, 93
- rendering forms
  - about 48, 92, 114
  - HTML 92
  - interactive 52, 92, 115
  - non-interactive 126
  - PDF 53
  - prepopulated 68, 119
- retrieving submitted data 60, 62, 117
- running scripts 18
- run-time options
  - converting content type (exportDataFormat) 66
  - standalone option 55
  - XCI options 56

## S

- saving a PDF document 108
- saving XML data 65
- sContentRootURI parameter 53
- server-side caching 54
- ServletOutputStream object 57, 93
- servlets. *See* Java servlets
- setContentTypes method 57, 93
- setInitialContext method 32
- setSOAPEndPoint method 35
- setting
  - a SOAP endpoint 35
  - configuration values for XML Form Module 125
  - form preference options 53, 92
- sFormQuery parameter 53
- SOAP endpoint 35
- SOAPClient class
  - about 30
  - creating 35
  - setSOAPEndPoint method 35

- SOAPClient object
  - using to invoke Form Server Module 34
  - using to render forms 58
- sPreference parameter 53, 92
- standalone option 55
- submitted data, retrieving 62
- submitting data as XML 60, 117

## T

- target URL 54
- template packet, importing 124
- transactions 40
- transferring PDF data 103

## U

- URL
  - object 69
  - specifying for posting data 54
- User Manager 110
- User Manager API 110
- UserTransaction object 40

## X

- XCI run-time options 56
- XDP packets 124
- XFA subsets 19
- XML classes 62
- XML documents, converting 69
- XML Form Module API 45
- XML forms 100
- XML string variable, converting 71
- xmlform-client.jar file 29



## Adobe® LiveCycle™ Forms

Version 7.2

- What's New
- Overview
- Installing and Configuring LiveCycle for JBoss
- Installing and Configuring LiveCycle for WebSphere
- Installing and Configuring LiveCycle for WebLogic
- Developing Custom Applications
- Developing User Management Service Providers
- Form Server Module API Reference
- XML Form Module API Reference
- Adobe User Management SPI Reference
- Transformation Reference
- Adobe LiveCycle Forms Readme

July 2006

# Installing and Configuring LiveCycle

The Installing and Configuring LiveCycle™ documentation provides information about installing, configuring, and deploying LiveCycle products. To ensure that customers have access to the most recent and updated information, the documentation is located at the following website:

[www.adobe.com/support/documentation/en/livecycle/](http://www.adobe.com/support/documentation/en/livecycle/)

## Installing and Configuring guides

The Installing and Configuring LiveCycle documentation is provided in a set of six guides based on the LiveCycle product and application servers.

The *Installing and Configuring LiveCycle for JBoss*, *Installing and Configuring LiveCycle for WebLogic*, and *Installing and Configuring LiveCycle for WebSphere* guides describe how to install and configure the following LiveCycle products and deploy the products to the specific application server:

- Adobe® LiveCycle Assembler 7.2.1
- Adobe LiveCycle Forms 7.2
- Adobe LiveCycle Form Manager 7.2
- Adobe LiveCycle PDF Generator 7.2 Professional, LiveCycle PDF Generator 7.2 Elements, and LiveCycle PDF Generator 7.2 for PostScript®
- Adobe LiveCycle Print 7.2
- Adobe LiveCycle Workflow 7.2.1
- Watched Folder

The *Installing and Configuring LiveCycle Security Products for JBoss*, *Installing and Configuring LiveCycle Security Products for WebLogic*, and *Installing and Configuring LiveCycle Security Products for WebSphere* guides describe how to install and configure the following LiveCycle products and deploy the products to the specific application server:

- Adobe LiveCycle Document Security 7.2
- Adobe LiveCycle Policy Server 7.2
- Adobe LiveCycle Reader® Extensions 7.2

## Updated product information

A Knowledge Center article has also been posted to communicate any updated LiveCycle product information. You can access the article from the following website:

[www.adobe.com/support/products/enterprise/knowledgecenter/c4811.pdf](http://www.adobe.com/support/products/enterprise/knowledgecenter/c4811.pdf)



# Transformation Reference

July 2006

**Adobe® LiveCycle™ Forms**

Version 7.2

© 2006 Adobe Systems Incorporated. All rights reserved.

Adobe® LiveCycle™ Forms 7.2 Transformation Reference for Microsoft® Windows®, UNIX®, and Linux®  
Edition 3.0, July 2006

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, LiveCycle, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Apple, Mac, and Mac OS are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Microsoft and Windows are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

UNIX is a registered trademark of The Open Group.

All other trademarks are the property of their respective owners.

This product includes code licensed from RSA Security, Inc.

Some portions licensed from IBM are available at <http://oss.software.ibm.com/icu4j/>.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

# Contents

---

Preface .....	4
1 About LiveCycle Forms Transformations .....	6
2 Button Object.....	7
3 Check Box Object .....	11
4 Circle Object .....	15
5 Content Area Object .....	16
6 Date/Time Field Object .....	17
7 Decimal Field Object.....	22
8 Drop-down List Object .....	27
9 Email Submit Button Object .....	32
10 HTTP Submit Button Object.....	36
11 Image Object .....	40
12 Image Field Object.....	42
13 Line Object .....	46
14 List Box Object .....	47
15 Numeric Field Object .....	52
16 Page Object .....	57
17 Password Field Object .....	58
18 Radio Button Object .....	63
19 Rectangle Object.....	67
20 Reset Button Object.....	69
21 Subform Object.....	73
22 Text Object .....	76
23 Text Field Object .....	78
24 Endnotes.....	83

# Preface

---

This *Transformation Reference* identifies which Adobe® LiveCycle™ Designer 7.0 properties are supported by various web browsers. Use this reference for forms developed in LiveCycle Designer and rendered by Adobe LiveCycle Forms into HTML format.

## What's in this reference?

This reference includes a table for each object that is available in the Standard library in LiveCycle Designer. The table lists the LiveCycle Designer properties and the LiveCycle Forms browser transformations.

## Who should read this guide?

This reference is intended for form developers who create forms that are rendered in HTML format and who need to understand which LiveCycle Designer properties are supported by different browsers.

## Related documentation

In addition to this reference, the following resources provide information about LiveCycle Forms.

<b>For information about</b>	<b>See</b>
Understanding how to use the LiveCycle Forms APIs to create custom applications	<i>Developing Custom Applications</i>
Installing, configuring, and administering in a development and run-time environment	<i>Installing and Configuring LiveCycle for JBoss</i> <i>Installing and Configuring LiveCycle for WebSphere</i> <i>Installing and Configuring LiveCycle for WebLogic</i>
The Form Server Module API, including a description and explanation of its classes and methods	<i>Form Server Module API Reference</i>
The XML Form Module API, including a description and explanation of its classes and methods	<i>XML Form Module API Reference</i>
The new features in this product release	<i>What's New</i>
The LiveCycle Designer form objects and properties, as well as scripting in LiveCycle Designer	<i>LiveCycle Designer Help</i>

---

<b>For information about</b>	<b>See</b>
The Adobe XML Form Object Model, which includes the LiveCycle Designer scripting objects, properties, and methods	<i>Adobe XML Form Object Model Reference</i> <a href="http://www.adobe.com/devnet/livecycle/designing_forms.html">www.adobe.com/devnet/livecycle/designing_forms.html</a>
Patch updates, technical notes, and additional information on this product version	<a href="http://www.adobe.com/support/products/enterprise/index.html">www.adobe.com/support/products/enterprise/index.html</a>

---

# 1

## About LiveCycle Forms Transformations

This *Transformation Reference* is intended to be used for forms developed in LiveCycle Designer 7.0 or later. Although forms designed for a browser environment support the majority of the objects available in LiveCycle Designer, they do not support all of their properties in every type of browser. This reference lists all of the objects and their properties and indicates which properties are supported. The reference also lists the scripting properties, methods, and events that are supported for some of the objects.

LiveCycle Forms should support any browser that follows the CSS2 specification. Since browsers vary widely in their support of CSS2 and older browsers provide no support at all, several browsers and generic user agents have been targeted with their own specific transformation types. A special accessible transformation is also available for Microsoft® Internet Explorer 5.0 and later browsers. Internet Explorer browsers on the Apple® Mac OS® platform render as XHTML.

**Note:** LiveCycle Forms 7.2 does not support tables in form designs rendered as HTML forms.

### How to use this reference

Each table is devoted to a single object that is available in the Standard library in LiveCycle Designer. The properties for each object are listed down the left column, organized by the palette in which they appear. For some objects, the supported scripting methods, properties, and events are also listed down the left column. The LiveCycle Forms browser transformations are listed across the top. The browser support for each property, method, and event is identified as described in the legend.

This guide includes only those scripting properties, methods, and events that are available in HTML clients. For a complete list of the Adobe XML Form Object Model scripting objects, properties, and methods, see the *Adobe XML Form Object Model Reference* guide on the LiveCycle Developer Center at [www.adobe.com/devnet/livecycle/designing\\_forms.html](http://www.adobe.com/devnet/livecycle/designing_forms.html). For a complete list of the scripting events, including descriptions and examples, see *LiveCycle Designer Help*.

Formats supported	
HTML 4 (Low end)	
MSDHTML	Microsoft Internet Explorer 5.0 and 6.0
XHTML	Apple Safari 1.2, Mozilla FireFox 1.0, and Netscape Navigator 7.2 <b>Note:</b> Netscape 7.2 is not supported on the Mac OS platform.
AHTML	Microsoft Internet Explorer 5.0 and 6.0

Legend	
Y	Implemented
N	Not supported by browser
(123)	<a href="#">Endnotes</a> available
(property)	Only supports the mentioned property. For example: height, width, protected.
(Not property)	Does not support the mentioned property. For example: Not Dotted Stroke.

# 2

## Button Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b>						
Edges Style	N	Y	Y	N	Y	Y
Edges Thickness	N	Y	Y	N	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	N	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y (Solid)	Y (Solid)	Y (Solid)	N	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	N	N	N
<b>Font palette</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	N	Y	Y
Height	Y (5)	Y	Y	N	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y	Y	Y	Y	Y
Control Type	Y	Y	Y	Y	Y	Y
Presence	Y	Y	Y	Y	Y	Y
Locale	N	N	N	N	N	N
<b>Object &gt; Execute</b>						
Connection	Y	Y	Y	Y	Y	Y
Run At	Y (Server)	Y (Server)	Y (Server)	Y (Server)	Y (Server)	Y (Server)
Re-merge Form Data	Y (Server)	Y (Server)	Y (Server)	Y (Server)	Y (Server)	Y (Server)
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 3

## Check Box Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b> (6)						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	N	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	N	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	N

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	N	N	N	N	N
States	N	N	N	N	N	N
Size	N	Y	Y	Y	Y	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	N	N	N	N	N
<b>Object &gt; Value</b>						
Type	N	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N
Validation Script Message	Y	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding (Open, Save, Submit)	Y	Y	Y	Y	Y	Y
On Value	Y	Y	Y	Y	Y	Y
Off Value	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 4

## Circle Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
N/A						
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	N	Y	N	N	N	N
Y	N	Y	N	N	N	N
Width	N	Y	N	N	N	N
Height	N	Y	N	N	N	N
Anchor	N	Y	N	N	N	N
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	N	N	N	N
Top	N	Y	N	N	N	N
Right	N	Y	N	N	N	N
Bottom	N	Y	N	N	N	N
<b>Object &gt; Draw</b>						
Type	N	Y	N	N	N	N
Appearance	N	Y	N	N	N	N
Start	N	Y	N	N	N	N
Sweep	N	Y	N	N	N	N
Line Style	N	Y	N	N	N	N
Fill	N	Y (Not Radial)	N	N	N	N
Presence	N	Y	N	N	N	N
<b>Paragraph palette</b>						
N/A						

# 5

## Content Area Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
N/A						
<b>Font palette</b>						
N/A						
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	N	Y	Y	Y	Y	Y
Y	N	Y	Y	Y	Y	Y
Width	N	Y	Y	Y	Y	Y
Height	N	Y	Y	Y	Y	Y
<b>Object &gt; Content Area</b>						
Name	Y	Y	Y	Y	Y	Y
Flow Direction	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
N/A						

# 6

## Date/Time Field Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b> (6)						
Edges Style	Y	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y (Not None) (2)	Y (Not None) (2)	Y	Y	Y (Not None) (2)
Control Type	Y	Y	Y	Y	Y	Y
Display Pattern	N	N	N	N	N	N
Edit Pattern	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y
<b>Object &gt; Value</b>						
Type	N	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Empty Message	Y	Y	Y	Y	Y	Y
Calculation Script	Y	Y	Y	Y	Y	Y
Runtime Property	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N
Validation Pattern	Y	Y	Y	Y	Y	Y
Validation Pattern Message	Y	Y	Y	Y	Y	Y
Validation Script Message	Y	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Data Pattern	Y	Y	Y	Y	Y	Y
Data Format	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

## 7

## Decimal Field Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b> (6)						
Edges Style	Y	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y (Not None) (2)	Y (Not None) (2)	Y	Y	Y (Not None) (2)
Display Pattern	N	N	N	N	N	N
Edit Pattern	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y
<b>Object &gt; Value</b>						
Type	N	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y
Empty Message	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Validation Pattern	Y	Y	Y	Y	Y	Y
Validation Pattern Message	Y	Y	Y	Y	Y	Y
Validation Script Message	Y	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Data Pattern	Y	Y	Y	Y	Y	Y
Data Format	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 8

## Drop-down List Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b> (6)						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y (Not None)	Y (Not None)	Y	Y	Y (Not None)
List Items	Y	Y	Y	Y	Y	Y
Allow Custom Text Entry	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y
<b>Object &gt; Value</b>						
Type	N	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Empty Message	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N
Validation Pattern	Y	Y	Y	Y	Y	Y
Validation Pattern Message	Y	Y	Y	Y	Y	Y
Validation Script Message	Y	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Specify Item Values	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	N (Left Only)	Y (Not Justify)	N (Left Only)	Y (Not Justify)	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
addItem	N	Y	Y	Y	Y	Y
clearItems	N	Y	Y	Y	Y	Y
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
change	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 9

## Email Submit Button Object

The Email Submit Button object is rendered as a submit button in HTML.

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b>						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y	Y	Y	Y	Y
Presence	Y	Y	Y	Y	Y	Y
Locale	N	N	N	N	N	N
Email Address	N	N	N	N	N	N
Email Subject	N	N	N	N	N	N
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
preSubmit	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 10

## HTTP Submit Button Object

The HTTP Submit Button object is rendered as a submit button in HTML.

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b>						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y	Y	Y	Y	Y
Presence	Y	Y	Y	Y	Y	Y
Locale	N	N	N	N	N	N
URL	N	N	N	N	N	N
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
preSubmit	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 11 | Image Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
<b>Borders (6)</b>						
Edges Style	N	N	N	N	N	N
Edges Thickness	N	N	N	N	N	N
Corners	N	N	N	N	N	N
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	N	Y (Solid)	Y (Solid)	Y	Y	N
Color (FillStart)	N	N	N	Y	Y	N
Color (FillEnd)	N	Y	Y	Y	Y	Y
<b>Font palette</b>						
N/A						
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Object &gt; Draw</b>						
Type	N	Y	Y	Y	Y	N
URL	Y	Y	Y	Y	Y	N
Embed Image Data	N	N	N	N	N	N
Sizing	Y	Y	Y	Y	Y	N
Presence	N	Y	Y	Y	Y	N
<b>Paragraph palette</b>						
N/A						

# 12 Image Field Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b> (6)						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	No	No	No	No	No	No
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	N	Y	Y	Y	Y	Y
URL	Y	Y	Y	Y	Y	Y
Embed Image Data	N	N	N	N	N	N
Sizing	Y	Y	Y	Y	Y	Y
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Import Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 13 | Line Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
N/A						
<b>Font palette</b>						
N/A						
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	N	Y	Y	Y	Y	N
Y	N	Y	Y	Y	Y	N
Width	N	Y	Y	Y	Y	Y
Height	N	Y	Y	Y	Y	Y
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Object &gt; Draw</b>						
Type	N	Y	Y	Y	Y	N
Appearance	N	Y	Y (Horizontal, Vertical)	Y (Horizontal, Vertical)	Y (Horizontal, Vertical)	N
Line Style	N	Y	Y	Y	Y	N
Presence	N	Y	Y	Y	Y	N
<b>Paragraph palette</b>						
N/A						

# 14 List Box Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders (6)</b>						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color only)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y (Not None)	Y (Not None)	Y	Y	Y (Not None)
List Items	Y	Y	Y	Y	Y	Y
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y
<b>Object &gt; Value</b>						
Type	Y	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y
Empty Message	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Validation Pattern	Y	Y	Y	Y	Y	Y
Validation Pattern Message	Y	Y	Y	Y	Y	Y
Validation Script Message	Y	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Specify Item Values	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	N (Left Only)	Y (Not Justify)	N (Left Only)	Y (Not Justify)	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
addItem	N	Y	Y	Y	Y	Y
clearItems	N	Y	Y	Y	Y	Y
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
change	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 15 | Numeric Field Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b> (6)						
Edges Style	Y	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)				
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color)	N				

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y (Not None) (2)	Y (Not None) (2)	Y	Y	Y (Not None) (2)
Display Pattern	N	N	N	N	N	N
Edit Pattern	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y
<b>Object &gt; Value</b>						
Type	N	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y
Empty Message	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Validation Pattern	Y	Y	Y	Y	Y	Y
Validation Pattern Message	Y	Y	Y	Y	Y	Y
Validation Script Message	Y	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Data Pattern	Y	Y	Y	Y	Y	Y
Data Format	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 16 Page Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
N/A						
<b>Font palette</b>						
N/A						
<b>Layout palette</b>						
N/A						
<b>Object &gt; Binding</b>						
N/A						
<b>Page palette</b>						
Name	Y	Y	Y	Y	Y	Y
Size	Y	Y	Y	Y	Y	Y
Orientation	Y	Y	Y	Y	Y	Y
<b>Landscape</b>						
Restrict Page Occurrence	Y	Y	Y	Y	Y	Y
Include Page in Numbering	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
N/A						

# 17

## Password Field Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b> (6)						
Edges Style	Y	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)				
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color)	N				

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y (Not None) (2)	Y (Not None) (2)	Y	Y	Y (Not None) (2)
Password Display Character	N	N	N	N	N	N
Edit Pattern	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Object &gt; Value</b>						
Type	N	Y	Y	Y	Y	Y
Empty Message	Y	Y	Y	Y	Y	Y
Validation Pattern	Y	Y	Y	Y	Y	Y
Validation Pattern Message	Y	Y	Y	Y	Y	Y
Validation Script Message	Y	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Data Pattern	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 18 Radio Button Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders (6)</b>						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	N	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	N	N	N	Y	N	N
<b>Font palette</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Size	Y	Y	Y	Y	Y	Y
Group	Y	Y	Y	Y	Y	Y
On Value	Y	Y	Y	Y	Y	Y
Appearance (4)	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	Y	Y	Y	Y	Y
<b>Object &gt; Group Value</b>						
Type	N	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y
Empty Message	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N
Validation Pattern	N	N	N	N	N	N
Validation Pattern Message	N	N	N	N	N	N
Validation Script Message	Y	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Object &gt; Group Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 19 Rectangle Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
N/A						
<b>Font palette</b>						
N/A						
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	N	Y	Y	Y	Y	N
Y	N	Y	Y	Y	Y	N
Width	N	Y	Y	Y	Y	Y
Height	N	Y	Y	Y	Y	Y
Anchor	N	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Object &gt; Draw</b>						
Type	N	Y	Y	Y	Y	N
Line Style	N	Y	Y	Y	Y	N
Corners	N	Y	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	N
Fill	N	Y (Not Radial)	Y (Not Radial or Linear)	Y (Not Radial or Linear)	Y (Not Radial or Linear)	N
Presence	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Paragraph palette</b>						
N/A						

# 20 Reset Button Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders</b>						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y	Y	Y	Y	Y
Presence	Y	Y	Y	Y	Y	Y
Locale	N	N	N	N	N	N
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 21 Subform Object

LiveCycle Designer properties, methods, and events	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
<b>Borders</b>						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	N	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	N
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	N	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	N
Fill Start	N	Y	Y	Y	Y	N
Fill End	N	Y	Y	Y	Y	Y
<b>Font palette</b>						
N/A						
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Import/Export Bindings (Execute)	N	N	N	N	N	N
Repeat Subform for Each Data Item	Y	Y	Y	Y	Y	Y
Min Count	Y	Y	Y	Y	Y	Y
Max	Y	Y	Y	Y	Y	Y
Overflow Leader	N	N	N	N	N	N
Overflow Trailer	N	N	N	N	N	N
<b>Object &gt; Subform</b>						
Type	Y	Y	Y	Y	Y	Y
Flow Direction	Y	Y	Y	Y	Y	Y
Allow Page Breaks within Content	N	N	N	N	N	N
Place	N	N	N	N	N	N
Keep w/ Previous	N	N	N	N	N	N
Keep w/ Next	N	N	N	N	N	N
After	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	N	N	N	N	N
<b>Paragraph palette</b>						
N/A						
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
borderWidth	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
instanceManager.occure.max	N	Y	Y	Y	Y	Y
instanceManager.occure.min	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties, methods, and events</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
<b>Methods</b>						
instanceManager.addInstance	N	Y	Y	Y	Y	Y
execCalculate	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
instanceManager.insertInstance	N	Y	Y	Y	Y	Y
instanceManager.moveInstance	N	Y	Y	Y	Y	Y
instanceManager.removeInstance	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y
instanceManager.setInstances	N	Y	Y	Y	Y	Y
<b>Events</b>						
initialize	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
calculate	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 22

## Text Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
N/A						
<b>Border palette</b>						
<b>Borders (6)</b>						
Edges Style	N	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	N	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	N
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	N	Y	Y	Y	Y	N
Color (FillStart)	N	Y	Y	Y	Y	N
Color (FillEnd)	N	Y	Y	Y	Y	N
<b>Font palette</b>						
Font	Y	Y	Y	Y	Y	Y
Size	Y	Y	Y	Y	Y	Y
Baseline Shift	N	N	N	N	N	N
Style (4)	Y	Y	Y	Y	Y	Y
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Object &gt; Draw</b>						
Type	N	Y	Y	Y	Y	N
Presence	N	Y	Y	Y	Y	N
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y (Not Justify)	Y (Not Justify)	Y (Not Justify)	Y (Not Justify)	Y (Not Justify)
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	Y
Right	N	Y	Y	Y	Y	Y
First	Y	Y	Y	Y	Y	Y
By	Y	Y	Y	Y	Y	Y
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	Y

## 23

## Text Field Object

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Accessibility palette</b>						
Tooltip	N	Y	Y	Y	Y	Y
Screen Reader Precedence	N	N	N	N	N	N
Custom Screen Reader Text	N	N	N	N	N	Y
<b>Border palette</b>						
<b>Borders (6)</b>						
Edges Style	Y	Y	Y	Y	Y	Y
Edges Thickness	N	Y	Y	Y	Y	Y
Corners	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)	Y (Rectangle)
Radius	N	N	N	N	N	N
<b>Background Fill</b>						
Style	Y	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)	Y (Solid)
Color (FillStart)	N	Y	Y	N	Y	Y
Color (FillEnd)	Y	N	N	Y	N	N
<b>Font palette</b>						
<b>Caption</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y	Y	Y	Y	Y	N
<b>Value</b>						
Font	Y	Y	Y	Y	Y	N
Size	Y	Y	Y	Y	Y	N
Baseline Shift	N	N	N	N	N	N
Style (3)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	Y (Color)	N

LiveCycle Designer properties	HTML 4 (Low End)	MSDHTML IE 5 & 6	XHTML Netscape 7	XHTML Safari	XHTML FireFox	Accessibility HTML
<b>Layout palette</b>						
<b>Size &amp; Position</b>						
X	Y (5)	Y	Y	Y	Y	N (1)
Y	Y (5)	Y	Y	Y	Y	N (1)
Width	Y (5)	Y	Y	Y	Y	Y
Height	Y (5)	Y	Y	Y	Y	Y
Width Expand to fit	N	N	N	N	N	N
Height Expand to fit	N	N	N	N	N	N
Anchor	Y	Y	Y	Y	Y	Y
Rotate	N	N	N	N	N	N
<b>Margins</b>						
Left	N	Y	Y	Y	Y	N
Top	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
Bottom	N	Y	Y	Y	Y	N
<b>Caption</b>						
Position	Y	Y	Y	Y	Y	N
Reserve	N	Y	Y	Y	Y	N
<b>Object &gt; Field</b>						
Type	Y	Y	Y	Y	Y	Y
Appearance (4)	N	Y (Not None) (2)	Y (Not None) (2)	Y	Y	Y (Not None) (2)
Allow Multiple Lines	Y	Y	Y	Y	Y	Y
Allow Plain Text Only	N	N	N	N	N	N
Limit Length	Y	Y	Y	Y	Y	Y
Max Chars	Y	Y	Y	Y	Y	Y
Display Pattern	N	N	N	N	N	N
Edit Pattern	N	N	N	N	N	N
Presence	Y	Y	Y	Y	Y	Y
Locale	N	N	N	N	N	N

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Object &gt; Value</b>						
Type	N	Y	Y	Y	Y	Y
Default	Y	Y	Y	Y	Y	Y
Empty Message	Y	Y	Y	Y	Y	Y
Override Message	N	N	N	N	N	N
Validation Pattern	Y	Y	Y	Y	Y	Y
Validation Pattern Message	Y	Y	Y	Y	Y	Y
Validation Script Message	Y	Y	Y	Y	Y	Y
<b>Object &gt; Binding</b>						
Name	Y	Y	Y	Y	Y	Y
Default Binding	Y	Y	Y	Y	Y	Y
Data Pattern	Y	Y	Y	Y	Y	Y
Data Format	Y	Y	Y	Y	Y	Y
Import/Export Bindings	Y	Y	Y	Y	Y	Y
<b>Paragraph palette</b>						
<b>Caption</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	Y	Y	Y	Y	N
<b>Value</b>						
HorizontalAlign (left, right, center, justify)	N	Y	Y	Y	Y	N
VerticalAlign (top, middle, bottom)	N	N	N	N	N	N
<b>Indents</b>						
Left	N	Y	Y	Y	Y	N
Right	N	Y	Y	Y	Y	N
First	Y	Y	Y	Y	Y	N
By	Y	Y	Y	Y	Y	N

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Spacing</b>						
Above	N	N	N	N	N	N
Below	N	N	N	N	N	N
Line Spacing	N	Y	Y	Y	Y	N
<b>Client-side scripting supported for HTML</b>						
<b>Properties</b>						
access	N	Y	Y	Y	Y	Y
borderColor	N	Y	Y	Y	Y	Y
borderWidth	N	Y	Y	Y	Y	Y
fillColor	N	Y	Y	Y	Y	Y
fontColor	N	Y	Y	Y	Y	Y
formattedValue	N	Y	Y	Y	Y	Y
h	N	Y	Y	Y	Y	Y
index	N	Y	Y	Y	Y	Y
mandatory	N	Y	Y	Y	Y	Y
name	N	Y	Y	Y	Y	Y
parent	N	Y	Y	Y	Y	Y
presence	N	Y	Y	Y	Y	Y
rawValue	N	Y	Y	Y	Y	Y
validationMessage	N	Y	Y	Y	Y	Y
w	N	Y	Y	Y	Y	Y
x	N	Y	Y	Y	Y	Y
y	N	Y	Y	Y	Y	Y
<b>Methods</b>						
execCalculate	N	Y	Y	Y	Y	Y
execEvent	N	Y	Y	Y	Y	Y
execInitialize	N	Y	Y	Y	Y	Y
execValidate	N	Y	Y	Y	Y	Y
resolveNode	N	Y	Y	Y	Y	Y
resolveNodes	N	Y	Y	Y	Y	Y

<b>LiveCycle Designer properties</b>	<b>HTML 4 (Low End)</b>	<b>MSDHTML IE 5 &amp; 6</b>	<b>XHTML Netscape 7</b>	<b>XHTML Safari</b>	<b>XHTML FireFox</b>	<b>Accessibility HTML</b>
<b>Events</b>						
calculate	N	Y	Y	Y	Y	Y
click	N	Y	Y	Y	Y	Y
enter	N	Y	Y	Y	Y	Y
exit	N	Y	Y	Y	Y	Y
initialize	N	Y	Y	Y	Y	Y
mouseDown	N	Y	Y	Y	Y	Y
mouseUp	N	Y	Y	Y	Y	Y
validate	N	Y	Y	Y	Y	Y

# 24 Endnotes

#	Transformation/ Browser	Objects	Properties	Notes
1	AHTML	All except Page	Layout/Size & Position: <ul style="list-style-type: none"> <li>• X</li> <li>• Y</li> </ul>	Objects are laid out in one column.
2	IE, Netscape	Date-Time Field, Decimal Field, Numeric Field, Password Field	Appearance	None option appears as sunken.
3	All	All except: <ul style="list-style-type: none"> <li>• Circle</li> <li>• Content Area</li> <li>• Line</li> <li>• Page</li> <li>• Rectangle</li> <li>• Image</li> <li>• Subform</li> </ul>	Font/Style/Underline	Only single underlines are supported in HTML.
4	All	All except: <ul style="list-style-type: none"> <li>• Circle</li> <li>• Content Area</li> <li>• Line, Page</li> <li>• Rectangle</li> <li>• Image</li> <li>• Subform</li> </ul>	Object/Field/ Appearance	Custom appearances may not be supported in HTML.
5	HTML 4	All except Page	Layout/Size & Position: <ul style="list-style-type: none"> <li>• X</li> <li>• Y</li> <li>• Width</li> <li>• Height</li> </ul>	Size and position approximated using HTML tables.
6	MSDHTML, XHTML	All fields and Text object	Border	Text in HTML may appear to be shifted to the left because the border extends inward. This behavior occurs when you use the default even-handed borders. To ensure you have the minimum amount of margin, adjust the margins to the size of the border width.



**Adobe**

# Developing User Management Service Providers

**Adobe® User Management**

July 2006

Version 1.23

© 2006 Adobe Systems Incorporated. All rights reserved.

Adobe® User Management 1.23 Developing User Management Service Providers for Microsoft® Windows®, Linux®, and UNIX®  
Edition 1.2, July 2006

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, and LiveCycle are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

IBM and WebSphere are trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

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

Microsoft and Windows are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are the property of their respective owners.

This product contains either BISAFE and/or TIPEM software by RSA Data Security, Inc.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

Portions Copyright (C) 1991, 1999 Free Software Foundation, Inc. The JBOSS, OmniORB, JacORB, and SwarmCache libraries are licensed under the GNU Library General Public License, a copy of which is included with this software.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

# Contents

<b>Preface</b> .....	<b>5</b>
What's in this guide? .....	5
Who should read this guide? .....	5
Related documentation .....	5
<b>1 Introduction</b> .....	<b>6</b>
Understanding the authentication process.....	6
Creating custom service providers .....	7
Including the User Management SPI JAR file .....	7
<b>2 Creating Custom Authentication Providers</b> .....	<b>8</b>
About custom authentication providers .....	8
Examining the authentication process .....	9
User Management SPI interfaces .....	10
Sample authentication provider .....	10
Retrieving authentication values.....	10
Retrieving configuration information.....	11
Performing the authentication operation.....	12
Sending authentication results to User Management.....	14
<b>3 Creating Custom Directory Service Providers</b> .....	<b>15</b>
Sample directory service provider .....	15
Directory service provider interfaces .....	16
Implementing the directory service provider interfaces .....	16
DirectoryPrincipalProvider interface.....	17
DirectoryUserProvider interface .....	21
DirectoryGroupProvider interface .....	21
Connecting to the directory .....	23
Getting the directory properties .....	24
GroupConfigBO interface .....	24
UserConfigBO interface .....	24
Testing the connection.....	24
Configuring User Management for custom directory service providers .....	25
<b>4 Registering Custom Service Providers</b> .....	<b>26</b>
User Management configuration settings .....	26
XML configuration file.....	27
XML element types in the configuration file.....	28
Defining domains for custom service providers .....	28
Configuring User Management to use custom authentication providers .....	29
Identifying authentication providers.....	29
Configuring domains for authentication providers .....	31
Configuring User Management to use directory service providers .....	33
<b>5 Deploying Custom Service Providers</b> .....	<b>35</b>
Packaging your custom service provider .....	35
Repackaging the LiveCycle EAR file .....	35
Deploying custom service providers .....	36
<b>Index</b> .....	<b>38</b>

## List of Examples

---

Example 2.1	Retrieving authentication values .....	11
Example 2.2	Retrieving User Management configuration information .....	12
Example 2.3	Performing an authentication operation .....	13
Example 2.4	Returning authentication results in an AuthResponseImpl object .....	14
Example 3.1	Entry point for user services.....	17
Example 3.2	Representing the state information.....	17
Example 3.3	Determining the state .....	19
Example 3.4	Collecting records.....	19
Example 3.5	Setting the principal information .....	20
Example 3.6	Implementing the DirectoryUserProvider interface .....	21
Example 3.7	Group state information .....	21
Example 3.8	Retrieving group members.....	22
Example 4.1	XML configuration file.....	27
Example 4.2	Defining a domain.....	28
Example 4.3	Identifying an authentication provider .....	30
Example 4.4	Configuring a domain for an authentication provider .....	31
Example 4.5	Configuring a domain .....	32
Example 4.6	Configuring the domain to use custom directory service providers .....	33
Example 4.7	Configuring the custom group provider.....	34
Example 4.8	Configuring the custom user provider .....	34

# Preface

---

This guide provides information about the use of Adobe® User Management SPI, which provides a means by which you can create custom service providers for User Management.

## What's in this guide?

This document provides information about the programmatic interfaces and classes that are used to create custom service providers for User Management. In addition, this guide discusses how to register and deploy custom service providers. This guide is a companion guide to *User Management SPI Reference*.

## Who should read this guide?

This guide is intended for Java 2 Enterprise Edition (J2EE) developers who are responsible for developing custom service providers for User Management.

## Related documentation

In addition to this guide, the *User Management SPI Reference* describes the interfaces and classes that are located in the User Management SPI. You can learn more about other Adobe services and products at [www.adobe.com](http://www.adobe.com) and <http://partners.adobe.com/public/developer/main.html>.

# 1

## Introduction

The Adobe User Management SPI is a Java API that enables you to create custom service providers for Adobe User Management. User Management enables administrators to maintain a database for all users and groups, synchronized with one or more third-party user directories. User Management provides authentication, authorization, and user management for LiveCycle products, including Adobe LiveCycle™ Workflow, Adobe LiveCycle Forms, and Adobe LiveCycle Form Manager. For more information about User Management, see *User Management Help*.

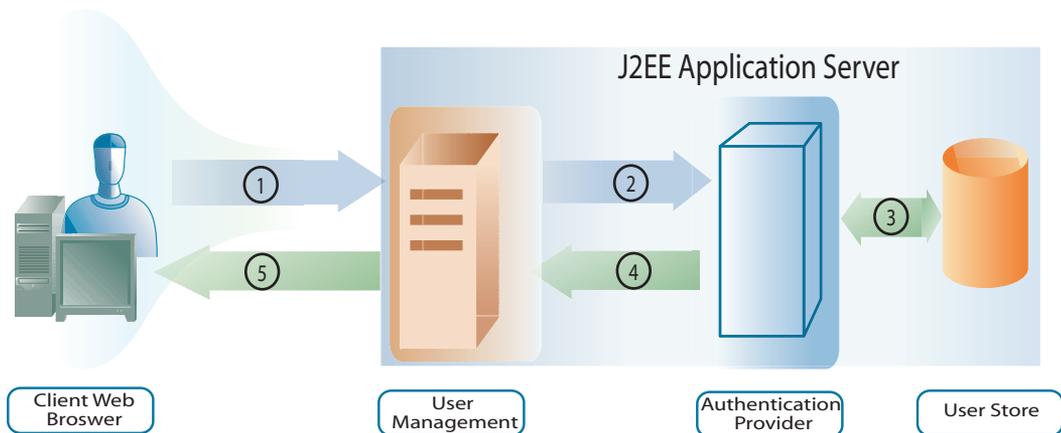
A User Management service provider consists of a custom authentication provider and a custom directory provider. A custom authentication provider authenticates users, and a directory provider stores user information. Using the User Management SPI, you can extend the User Management default functionality that is related to authenticating users and storing user information.

### Understanding the authentication process

Authentication providers receive authentication requests from User Management and provide responses to it.

When User Management receives an authentication request (for example, a user attempts to log in), it passes user information to the authentication provider to authenticate. User Management receives the results from the authentication provider after it authenticates the user.

The following diagram shows the interaction among an end user attempting to log in, User Management, and an authentication provider.



The following table describes each step of the process and indicates the SPI members involved.

Step	Description
1	A user attempts to log into a LiveCycle product that invokes User Management. The user specifies a user name and password.
2	User Management sends the user name and password, as well as configuration information, to the authentication provider.

Step	Description
3	The authentication provider connects to the user store and authenticates the user.
4	The authentication provider returns the results to User Management.
5	User Management either lets the user log in or denies access to the product.

## Creating custom service providers

You use the User Management SPI to create a custom service provider by performing the following tasks:

1. Create a custom authentication provider. For information, see [“Creating Custom Authentication Providers” on page 8](#).
2. Create a custom directory provider. For information, see [“Creating Custom Directory Service Providers” on page 15](#).
3. Configure the User Management XML configuration file to recognize the new service provider. For information, see [“Registering Custom Service Providers” on page 26](#).
4. Deploy the custom service provider. For information, see [“Deploying Custom Service Providers” on page 35](#).

## Including the User Management SPI JAR file

User Management SPI consists of two Java packages:

- com.adobe.idp.um.spi.authentication
- com.adobe.idp.um.spi.directoryservices

You use classes and interfaces located within these packages to create custom service providers. For information about these packages and their contents, see the *API Reference*.

The User Management SPI is packaged in a JAR file named um-spi.jar. You must copy the JAR file into your application’s class path in order to use the User Management SPI in your Java project.

The um-spi.jar file is installed in the following directory when User Management is installed:

```
C:\Adobe\LiveCycle\components\um\<app_server>\lib\adobe
```

where C:\ is the drive on which User Management is installed, and app\_server is the J2EE application on which User Management is deployed. For example, assume that User Management is deployed on JBoss. In this situation, the um-spi.jar file is in the following directory:

```
C:\Adobe\LiveCycle\components\um\jboss\lib\adobe
```

To access the interfaces and classes in the um-spi.jar file, add the following import statements to your Java project:

```
import com.adobe.idp.um.spi.authentication.*;
import com.adobe.idp.um.spi.directoryservices.*;
```

# 2

## Creating Custom Authentication Providers

This chapter explains how to use the User Management SPI to develop custom authentication providers that are based on user name and password authentication.

By default, User Management supports JAAS and LDAP authentication. However, by using the User Management SPI, you can create a custom authentication provider and then configure User Management to use the custom authentication provider to replace its default authentication provider. You can also configure User Management to use the custom authentication provider in addition to the default authentication provider.

A custom authentication provider is dependent on a custom directory service provider; therefore, you must create a custom directory service provider when you create a custom authentication provider. User information that is authenticated by a custom provider is placed in a data store that is accessed by a customer directory service provider. For information, see [“Creating Custom Directory Service Providers” on page 15](#).

This chapter provides the following information.

Topic	Description	See
About custom authentication providers	Explains how to use the User Management SPI to develop a custom authentication provider.	<a href="#">page 8</a>
Retrieving authentication values	Explains how to retrieve user values that are passed from User Management to the custom authentication provider.	<a href="#">page 10</a>
Retrieving configuration information	Explains how to retrieve User Management configuration information that is used in the authentication process.	<a href="#">page 11</a>
Performing the authentication operation	Explains how to perform an authentication operation. This section also provides an example of an authentication operation by using a tab-delimited file.	<a href="#">page 12</a>
Sending authentication results to User Management	Explains how to send authentication results to User Management.	<a href="#">page 14</a>

### About custom authentication providers

The User Management SPI provides a Java API for developing custom authentication providers. To develop a custom authentication provider, create a Java class that implements the `AuthProvider` interface that belongs to the `com.adobe.idp.um.spi.authentication` package. This class must contain a method named `authenticate`, which is called by User Management to authenticate users.

User Management passes user and configuration values to the `authenticate` method when a user attempts to log in. A user name and the corresponding password are passed within a `java.util.Map` object. Configuration information is passed within a `java.util.List` object. The following code fragment shows the method signature of the `authenticate` method:

```
public AuthResponse authenticate(Map credential, List authConfigs)
```

This method returns an `AuthResponse` object that specifies whether the user was authenticated. For information, see [“Sending authentication results to User Management” on page 14](#).

## Examining the authentication process

This section examines in more detail the authentication process that was introduced earlier in this guide. For information, see [“Understanding the authentication process” on page 6](#).

During most authentication steps, User Management invokes methods of your custom authentication provider. For example, User Management invokes the `authenticate` method after a user attempts to log in. The following table explains the relationship between the authentication process and the User Management SPI, and specifies the SPI methods involved in each step of the authentication process.

Step	Description	SPI member used
1	A user attempts to log into a LiveCycle product that invokes User Management. The user specifies a user name and password.	No SPI methods are invoked.
2	User Management sends the user name and password, as well as configuration information, to the authentication provider.	User Management invokes the <code>authenticate</code> method that is located in an authentication provider. The <code>authenticate</code> method requires a <code>java.util.Map</code> object that contains user information and a <code>java.util.List</code> object that contains configuration information as arguments. For information, see <a href="#">“Retrieving authentication values” on page 10</a> .  The configuration information contained in the <code>java.util.List</code> object is a collection of one or more <code>AuthConfigBO</code> objects. For information, see <a href="#">“Retrieving configuration information” on page 11</a> .
3	The authentication provider connects to the user store and authenticates the user.	You are required to develop Java code that performs the authentication. For example, you can authenticate a user by using a third-party API, such as the Java JDBC API, that interacts with a specified user store.  This chapter authenticates a user by searching a tab-delimited file that contains user information. For information, see <a href="#">“Performing the authentication operation” on page 12</a> .
4	The authentication provider returns the results to User Management.	The authentication provider stores the results in an <code>AuthResponse</code> object. The <code>authenticate</code> method returns the <code>AuthResponse</code> object to User Management. For information, see <a href="#">“Sending authentication results to User Management” on page 14</a> .
5	User Management either lets the user log in or denies the user access to the product.	No SPI methods are invoked.

## User Management SPI interfaces

The following table lists and describes the User Management SPI interfaces and classes that are used to create a custom authentication provider.

Interface	Description
<code>AuthProvider</code>	This interface is the primary interface that your custom authentication provider must implement.
<code>AuthConfigBO</code>	This interface defines a container for storing configuration information about an authentication provider.
<code>AuthResponse</code>	This interface is used to communicate authentication results to User Management. If the authentication provider successfully authenticates the user, it communicates the success to User Management along with the authenticated user name, user domain, and authentication type.  Likewise, this interface is used to inform User Management that authentication was unsuccessful.
<code>AuthResponseImpl</code>	This class is used to create an object instance of the <code>AuthResponse</code> interface.

## Sample authentication provider

User Management provides a sample implementation of the `AuthProvider` interface. The sample demonstrates how service providers interact with User Management. The sample Java code is provided in the file named `FBAuthenticationProviderImpl.java`. You can retrieve the User Management sample from the Adobe LiveCycle Developer Center website at [http://partners.adobe.com/public/developer/livecycle/index\\_samples.html](http://partners.adobe.com/public/developer/livecycle/index_samples.html).

**Tip:** You may find it useful to print the sample code and refer to it as you read this chapter.

## Retrieving authentication values

User Management passes user and configuration information to your authentication provider's `authenticate` method. User information is passed in by using a `java.util.Map` object, and configuration information is passed in by using a `java.util.List` object.

You retrieve the user information by invoking the `java.util.Map` object's `get` method. Pass the following keys to the `get` method to retrieve user information.

Key name	Description
<code>AuthProvider.USER_NAME</code>	The user name to authenticate.
<code>AuthProvider.PASSWORD</code>	The password that corresponds to the user name.
<code>AuthProvider.AUTH_TYPE</code>	The type of authentication to perform.  The type must be <code>AUTHTYPE_USERNAME_PWD</code> . If no type is specified, the type <code>AUTHTYPE_USERNAME_PWD</code> is assumed. If a different type is specified, the authentication provider must respond with an indication that the parameters were not understood.

The following example retrieves authentication values from the `java.util.Map` object that is passed to the authentication provider. The `java.util.Map` object's `get` method is used to retrieve each value.

### Example 2.1 Retrieving authentication values

```
public AuthResponse authenticate(Map credential, List authConfigs) {

 //Declare string variables to store user information
 String userName = null;
 String password = null;
 String authType = null;

 // Retrieve the user values passed from User Management
 if (credential.containsKey(AuthProvider.USER_NAME))
 userName = (String) credential.get(AuthProvider.USER_NAME);
 if (credential.containsKey(AuthProvider.PASSWORD))
 password = (String) credential.get(AuthProvider.PASSWORD);
 if (credential.containsKey(AuthProvider.AUTH_TYPE))
 authType = (String) credential.get(AuthProvider.AUTH_TYPE);
 //More logic
}
```

**Note:** The `java.util.Map` object may also include the keys named `AuthProvider.CONTEXT` and `AuthProvider.ENCODED_KERBEROS_TICKET`. Your authentication provider can ignore these keys.

## Retrieving configuration information

A custom authentication provider can retrieve configuration information that is passed by User Management. This information represents domain information specified in the User Management configuration settings. For information about domain information, see [“Defining domains for custom service providers” on page 28](#).

User Management configuration settings can include required connection values for the data store as well as other information specific to your authentication provider. The domain information includes the name of the domain in which you are authenticating. You must retrieve the domain name for each authentication operation that is performed.

Multiple domains can exist for one authentication provider. It is the authentication provider's responsibility to try them all and select the one that succeeds. The domain name is specified in the response that is sent to User Management. For information, see [“Sending authentication results to User Management” on page 14](#).

The `java.util.List` object that was passed to the authentication provider's `authenticate` method contains an `AuthConfigBO` object for each domain that is configured for a specific authentication provider. An `AuthConfigBO` object is a container for domain configuration information that applies to a specific authentication provider.

Iterate through the `java.util.List` object to retrieve all `AuthConfigBO` objects by creating an `Iterator` object. To create an `Iterator` object, call the `java.util.List` object's `iterator` method, as shown in the following code fragment:

```
Iterator it = authConfigs.iterator();
```

After you create an `Iterator` object, reference the individual `AuthConfigBO` objects by invoking the `Iterator` object's `next` method and casting the return value, as shown in the following code fragment:

```
AuthConfigBO conf = (AuthConfigBO)it.next();
```

Call the `AuthConfigBO` object's `getCustomConfiguration` method to get a `java.util.Map` object that contains key-value pairs of string objects representing the first-level custom configuration settings. You cannot use the `java.util.Map` object to modify configuration settings.

You get the value of a specific key by invoking the `java.util.Map` object's `get` method and passing a string value that represents the key name. However, to retrieve a key value, you must know the key name that is in the User Management XML configuration file (an example of getting the value of an XML key named `FILENAME` is shown in the following code example). For information about the User Management XML configuration file, see ["User Management configuration settings" on page 26](#).

The following example retrieves configuration information from the `java.util.List` object named `authConfigs` that was passed to the authentication provider's `authenticate` method. Notice that a file name value is retrieved. This value is stored in a string variable named `filename` and is used in the implementation of this authentication provider.

### **Example 2.2 Retrieving User Management configuration information**

```
//AuthConfigs is an authenticate parameter
//Create an Iterator object
Iterator it = authConfigs.iterator();

while(it.hasNext()){
 //Get an AuthConfigBO object
 AuthConfigBO conf = (AuthConfigBO)it.next();

 //Retrieve a java.util.Map object that stores User Management
 //configuration information
 java.util.Map configSettings = conf.getCustomConfiguration();

 //Get a value of the FILENAME key
 String filename = (String) configSettings.get("FILENAME");

 //Get the domain name
 String domainName = conf.getDomainName();

 //Do something with filename and domainName
}
```

## Performing the authentication operation

After you retrieve user information and required configuration values, you can perform an authentication operation.

You can use a third-party Java API to interact with the authentication server provider you are using. For example, you can use the Java Simple Authentication and Security Layer (SASL) API to perform authentication operations.

For the purposes of this chapter, an authentication operation is performed by using a 0-based, tab-delimited file. If the specified user name, password, and domain values are located in the file, the authentication operation is successful. Otherwise, the authentication operation is unsuccessful.

The following table describes the tab-delimited file.

Column	Description	Example
0	The domain of the user	MyDomain2
1	The login identifier of the user	s_user10
2	The password of the user	password10

In the following code example, a user-defined method named `checkValues` searches a tab-delimited file for the specified user name, password, and domain values. All three values are passed to this method as arguments. This method returns `true` if the authentication operation is successful; otherwise, it returns `false`.

### Example 2.3 *Performing an authentication operation*

```
private boolean checkValues(
 String userid,
 String password,
 String domain,
 String filename) {
 BufferedReader br = null;
 try {
 //create a BufferedReader object
 br = new BufferedReader(new FileReader(filename));
 String line = null;

 //Read each line in the file and search for the values
 while ((line = br.readLine()) != null) {
 String[] parts = line.split("\t");
 if (userid.equals(parts[1]) && domain.equals(parts[0]) &&
password.equals(parts[2]))
 return true;
 }
 } catch (IOException e) {
 System.err.println(e);
 } finally {
 if (br != null) {
 try {
 br.close();
 } catch (IOException e) {}
 }
 }
 return false;
}
```

The file name was retrieved from the configuration settings that were passed by User Management. For information, see ["Retrieving configuration information" on page 11](#).

The user name and password values were retrieved from the `java.util.Map` object that was passed to the `authenticate` method. For information, see ["Retrieving authentication values" on page 10](#).

## Sending authentication results to User Management

After the authentication provider performs the authentication, it must return the results to User Management. The authentication provider must specify the user name, the domain name, and whether the authentication was successful.

The User Management SPI provides the `AuthResponseImpl` class, which implements the `AuthResponse` interface. You create an `AuthResponse` object and populate it with the appropriate information. The following code fragment creates an `AuthResponse` object:

```
AuthResponse response = new AuthResponseImpl();
```

You can inform User Management that the authentication was successful by invoking the `AuthResponse` object's `setAuthStatus` method and passing `AuthResponse.AUTH_SUCCESS`. You can also inform User Management that the authentication was unsuccessful by invoking the `AuthResponse` object's `setAuthStatus` method and passing `AuthResponse.AUTH_FAILED`.

Use the `return` statement in the `authenticate` method to send the `AuthResponse` object to User Management. For information about the `authenticate` method, see [“About custom authentication providers” on page 8](#).

The following example populates an `AuthResponse` object after a user is successfully authenticated.

### **Example 2.4** *Returning authentication results in an `AuthResponseImpl` object*

```
//create the AuthResponse object
AuthResponse response = new AuthResponseImpl();
response.setAuthStatus(AuthResponse.AUTH_SUCCESS);
response.setUsername(userName);
response.setDomain(conf.getDomain());
return response;
```

## Trapping errors

The `AuthProvider` interface does not throw exceptions, but errors should be included in authentication results. Your authentication provider should collect exceptions and add them to the `AuthResponse` object by using the `setExceptions` method. You can also provide an error description by using the `setErrorMessage` method.

# 3

## Creating Custom Directory Service Providers

This chapter explains how to use the User Management SPI to develop custom directory service providers that you may integrate with User Management.

User Management is packaged with a directory service provider that supports connections to LDAP directories. If your organization uses a non-LDAP repository to store user records, you can create your own directory service provider that works with your repository.

Directory service providers retrieve records from a user store at the request of User Management. User Management regularly caches user and group records in the database to improve performance.

To implement a custom directory service provider, create a user provider and a group provider:

- User providers retrieve all required user records from the repository.
- Group providers retrieve all required user and group records within a specified group, and retrieve all required group records in the repository.

This chapter contains the following information.

Topic	Description	See
Sample directory service provider	Provides information on sample implementations of the directory service provider interfaces	<a href="#">page 15</a>
Directory service provider interface	Describes the directory service provider interfaces to be implemented	<a href="#">page 16</a>
Implementing the directory service provider interfaces	Explains how to develop custom directory service provider interfaces	<a href="#">page 16</a>
Connecting to the directory	Describes how to retrieve directory configuration information and test the connection to User Management	<a href="#">page 23</a>
Configuring User Management to use custom providers	Describes the steps required to update the XML-based configuration file to enable the usage of your custom providers within User Management	<a href="#">page 25</a>

### Sample directory service provider

User Management provides a sample implementation of the `DirectoryPrincipalProvider`, `DirectoryGroupProvider`, and `DirectoryUserProvider` interfaces. The sample demonstrates how directory service providers may interact with User Management. You can retrieve the User Management sample from the Adobe LiveCycle Developer Center website at [http://partners.adobe.com/public/developer/livecycle/index\\_samples.html](http://partners.adobe.com/public/developer/livecycle/index_samples.html).

**Tip:** You may find it useful to print the sample code and reference it as you read this chapter.

## Directory service provider interfaces

You must implement the following three interfaces when using a custom repository to store user and group information.

Interface	Purpose
<code>DirectoryPrincipalProvider</code>	The base interface for the user provider ( <code>DirectoryUserProvider</code> ) and group provider ( <code>DirectoryGroupProvider</code> ) interfaces. Declares methods for retrieving user and group records and for testing the configuration settings and constants for reporting exceptions to User Management.
<code>DirectoryUserProvider</code>	The user provider interface. You must implement the methods and exceptions needed for implementing a custom user provider that retrieves user records and for testing the configuration settings. User Management requires an implementation of this interface to synchronize its database with the user store.
<code>DirectoryGroupProvider</code>	The group provider interface. You must implement the methods and exceptions needed for implementing a custom group provider that retrieves user records that belong to a specific group in the repository and for testing the configuration settings. User Management requires an implementation of this interface when any action is performed on a group, such as adding a group to a policy.

All three interfaces require the implementation of a `getPrincipals` method and a `testConfiguration` method. In all three cases, the entry point for user services is the `getPrincipals` method.

**Note:** User Management calls the `getPrincipals` method only for the implementations of the `DirectoryUserProvider` and `DirectoryGroupProvider` interfaces.

## Implementing the directory service provider interfaces

To retrieve user and group records, you must implement the `DirectoryUserProvider` and `DirectoryGroupProvider` interfaces described in [“Directory service provider interfaces” on page 16](#), and configure User Management for your custom repository by modifying the configuration file as described in [“Configuring User Management for custom directory service providers” on page 25](#).

In this section, implementation guidelines are provided for the interfaces and are based on these samples provided in the User Management installation: `FBDirectoryPrincipalProviderImpl.java`, `FBDirectoryUserProviderImpl.java`, and `FBDirectoryGroupProviderImpl.java`.

**Note:** For information about the location of these samples, see [“Sample directory service provider” on page 15](#).

## DirectoryPrincipalProvider interface

The `DirectoryPrincipalProvider` interface is the base interface for `DirectoryUserProvider` and `DirectoryGroupProvider`. This interface requires that you implement methods that provide the entry point for user services (`getPrincipals`) and a test of the configuration for User Management (`testConfiguration`).

The `getPrincipals` method receives two parameters:

- `config`—A `DirectoryProviderConfig` object that contains information about the user records to be retrieved and User Management configuration information.
- `state`—An implementor-defined object that indicates whether this is the first time the method has been called or the state of the previous call. This object typically contains fields originating from the User Management configuration settings; however, it can also be used to store any information required between `getPrincipals` calls.

The `getPrincipals` method is repeatedly called by User Management until all records (both user and group) are retrieved. The record collection is returned within a `DSPrincipalCollection` object, which also contains the state information to be used in the next call.

The following example shows the `getPrincipals` method, which is the entry point for user services.

### Example 3.1 *Entry point for user services*

```
public DSPrincipalCollection getPrincipals (
 DirectoryProviderConfig config,
 Object state) throws IDPEException {
 return grabBatchPrincipals (config, state);
}
```

In the example, processing is handled in the `grabBatchPrincipals` method. A description of the method's algorithm follows.

Declare an object of the `DSPrincipalCollection` class, which will store the user or group records to be retrieved from the custom repository. This object will be returned when the `getPrincipals` method terminates and will contain both the records and the state information used in the next call.

The state information should include the following data:

- Principal type
- Batch size
- Principal's file handle
- Configuration information
- Domain

The following example shows implementor-defined classes used to represent the state information.

### Example 3.2 *Representing the state information*

```
protected class FBConfig {
 private String principalType;
 private int batchSize;
 private String principalsFileName;
}
```

```
private Map customConfig;
private String domain;
private static final String RECORD_USER = "USER";
private static final String RECORD_GROUP = "GROUP";

public int getBatchSize() {
 return batchSize;
}
public Map getCustomConfig() {
 return customConfig;
}
public String getDomain() {
 return domain;
}
public String getPrincipalsFileName() {
 return principalsFileName;
}
public String getPrincipalType() {
 return principalType;
}
public void setBatchSize(int batchSize) {
 this.batchSize = batchSize;
}
public void setCustomConfig(Map customConfig) {
 this.customConfig = customConfig;
}
public void setDomain(String domain) {
 this.domain = domain;
}
public void setPrincipalsFileName(String principalsFileName) {
 this.principalsFileName = principalsFileName;
}
public void setPrincipalType(String principalType) {
 this.principalType = principalType;
}
}
protected class FBPrincipalState{
 public FBConfig config;
 public File principalsFile;
 public BufferedReader brPrincipals;

 public void finalize() {
 try {
 brPrincipals.close();
 } catch (Exception e) {
 System.err.println(e);
 }
 }
}
}
```

If this is the first time the `getPrincipals` method is called (state parameter is null), perform the initializations required to begin collecting the information. Otherwise, copy the state into the `DSPrincipalCollection` object to propagate it to the next call.

In the following example, `dpc` is the `DirectoryProviderConfig` object, which is used to read the configuration preferences. The implementor-defined `prepEnumeration` method gathers the state information.

### Example 3.3 Determining the state

```
FBPrincipalState fbState = null;
if (state == null)
 fbState = prepEnumeration(dpc);
else if (state instanceof FBPrincipalState)
 fbState = (FBPrincipalState) state;
else {
 // throw exception
}

ret.setState(fbState); //carry state forward
```

Collect the records, which are objects of the `DSPrincipalRecord` class. If there are no objects to collect, `null` is returned. It is recommended that implementations try to return fewer than 1000 records at a time for optimal performance.

After you obtain a new record, add it to the collection. In the following example, this task is done by invoking the `DSPrincipalCollection` object's `addDSPrincipalRecord` method.

### Example 3.4 Collecting records

```
int batchcount = 0;
int maxbatch = fbState.config.getBatchSize();
String line = null;
while (batchcount < maxbatch &&
 (line = fbState.brPrincipals.readLine()) != null) {
 ++batchcount;
 DSPrincipalRecord rec = grabPrincipal(fbState, line);
 ret.addDSPrincipalRecord(rec);
}

if (batchcount == 0 && line == null)
 return null;
```

For each record, set the principal type (user or group) according to the configuration information. It is recommended that you consider the option of setting the organization name of the principal.

For all principals, you must set the following information:

- Domain name
- Canonical name
- Primary email address
- Full name

For users, you must set the User identification. For groups, you must set the type of group record.

In the following example, the `grabPrincipal` method performs these tasks. Each setting is used and is labeled as either required or optional within the example.

### Example 3.5 *Setting the principal information*

```
private DSPrincipalRecord grabPrincipal(
 FBPrincipalState state,
 String line) {

 //Create a DSPrincipalRecordImpl object
 DSPrincipalRecord rec = new DSPrincipalRecord();

 rec.setPrincipalType(state.config.getPrincipalType());
 String[] parts = line.split("\t");

 // These settings apply to all principals:
 rec.setDomainName(parts[0]); // REQUIRED
 rec.setCanonicalName(parts[1]); // REQUIRED
 rec.setDescription(parts[2]); // OPTIONAL
 rec.setEmail(parts[3]); // REQUIRED
 rec.setCommonName(parts[4]); // REQUIRED
 rec.setOrg(parts[5]); // OPTIONAL

 //These settings apply only to users:
 if (state.config.getPrincipalType().equals(FBConfig.RECORD_USER)) {
 rec.setUserid(parts[6]); // REQUIRED
 //password is parts[7] (useful only for authentication providers)
 rec.setFamilyName(parts[8]); // OPTIONAL
 rec.setGivenName(parts[9]); // OPTIONAL
 rec.setInitials(parts[10]); // OPTIONAL
 rec.setTelephoneNumber(parts[11]); // OPTIONAL
 rec.setPostalAddress(parts[12]); // OPTIONAL
 rec.setLocale(parts[13]); // OPTIONAL
 rec.setTimezone(parts[14]); // OPTIONAL
 }

 // These settings apply only to groups:
 if (state.config.getPrincipalType().equals(FBConfig.RECORD_GROUP)) {
 rec.setGroupType(DSPrincipalRecord.GROUPTYPE_PRINCIPALS);
 }

 return rec;
}
```

The implementation of the `DirectoryUserProvider` interface has the same required methods as the `DirectoryPrincipalProvider` interface.

You may optionally extend the methods defined in the base class. The example reuses the base class method implementations.

## DirectoryUserProvider interface

The `DirectoryUserProvider` interface inherits from the `DirectoryPrincipalProvider` interface and must be implemented for your custom user provider service.

You may optionally override the methods defined in the base class, as shown in the following example.

### Example 3.6 *Implementing the DirectoryUserProvider interface*

```
public class FBDirectoryUserProviderImpl
 extends FBDirectoryPrincipalProviderImpl
 implements DirectoryUserProvider {

 public FBDirectoryUserProviderImpl() {
 super();
 }
}
```

## DirectoryGroupProvider interface

The `DirectoryGroupProvider` interface inherits from the `DirectoryPrincipalProvider` interface. You may optionally override the methods defined in the base class, but you must also implement one more method—the `getGroupMembers` method.

The `getGroupMembers` method accepts two parameters:

- A `DirectoryProviderConfig` object containing information used to connect to the repository
- A `DSPrincipalIdRecord` object that identifies the group to be retrieved

The method returns a `DSGroupContainmentRecord` object, which is a container for the user and group records belonging to a group.

The `getGroupMembers` method depends on an implementor-defined object that contains the state information as described in [“Determining the state” on page 19](#), as well as the group containment file name. An implementation is shown in the following example.

### Example 3.7 *Group state information*

```
private class FBGroupConfig {
 private FBConfig commonConfig;
 private String groupContainmentFileName;
 public FBConfig getCommonConfig() {
 return commonConfig;
 }

 public String getGroupContainmentFileName() {
 return groupContainmentFileName;
 }

 public void setCommonConfig(FBConfig config) {
 commonConfig = config;
 }
}
```

```
 public void setGroupContainmentFileName(String string) {
 groupContainmentFileName = string;
 }
 }

private FBGroupConfig createFBGroupConfig(DirectoryProviderConfig dpc){
 FBGroupConfig ret = new FBGroupConfig();
 ret.setCommonConfig(createFBConfig(dpc));

 ret.setGroupContainmentFileName(
 (String)ret.getCommonConfig().getCustomConfig().get(
 "GROUPCONTAINMENT_FILE_NAME"
)
);
 return ret;
}
```

To retrieve the user records, implement the `getGroupMembers` method by performing these tasks:

1. Use the User Management configuration information stored in the `DirectoryProviderConfig` object (in this case by invoking the `createFBGroupConfig` method).
2. Set up a `DSGroupContainmentRecord` object.
3. Read in the user records and collect each one (in this case, it is accomplished by calling a method named `grabContainment`). Only those user or group records that belong to the group should be collected. It is also possible to collect nested groups. To do so, compare the group name contained in each record with the group name of the principal. When a group is found, store its domain and group name.
4. Collect all the principals belonging to that group.

### **Example 3.8 Retrieving group members**

```
public DSGroupContainmentRecord getGroupMembers(
 DirectoryProviderConfig dpc,
 DSPrincipalIdRecord group
) throws IDPException {
 // Check to make sure the group domain is known:
 if (group == null) {
 // throw IDPException
 }

 FBGroupConfig config = createFBGroupConfig(dpc);
 DSGroupContainmentRecord ret = null;
 String groupName = group.getCanonicalName();

 // open file and search for this groupname.
 try {
 BufferedReader brContainment = new BufferedReader(
 new FileReader(config.getGroupContainmentFileName())
);
 }
```

```
String line = null;
while ((line = brContainment.readLine()) != null) {

 ret = grabContainment(config, groupName, line);

 if (ret != null)
 break;
}
brContainment.close();
} catch (Exception e) {
 // throw IDPException
}
return ret;
}

private DSGroupContainmentRecord grabContainment(
 FBGroupConfig config,
 String groupName,
 String line
) {
 String[] parts = line.split("\t");
 if (! parts[0].equals(groupName))
 return null;

 String domain = config.getCommonConfig().getDomain();

 DSGroupContainmentRecord ret = new DSGroupContainmentRecord();
 ret.setDomainName(domain);
 ret.setCanonicalName(groupName);

 for (int i=1;i<parts.length;i++) {
 DSPrincipalIdRecord memprin = new DSPrincipalIdRecord();
 memprin.setDomainName(domain);
 memprin.setCanonicalName(parts[i]);
 ret.addPrincipalMember(memprin);
 }

 return ret;
}
```

## Connecting to the directory

This section provides information on how to retrieve directory properties from the directory service provider configuration.

## Getting the directory properties

The directory service provider interface implementations all rely on the directory service provider configuration. To obtain this information, use the `DirectoryProviderConfig` object, which provides these methods:

- `getDomain`—Retrieves the domain name
- `getGroupConfig`—Retrieves the group provider configuration information by returning an instance of a class that implements the `GroupConfigBO` interface (see [GroupConfigBO interface](#))
- `getUserConfig`—Retrieves the user provider configuration information by returning an instance of a class that implements the `UserConfigBO` interface (see [UserConfigBO interface](#))

**Note:** The `getGroupConfig` method and `getUserConfig` method both return implementations of interfaces that define a `getCustomConfiguration` method, which makes it possible to access the configuration settings within a `java.util.Map` object that contain key-value pairs of strings that represent the first-level configuration settings. These strings may be parsed as needed.

See [“Implementing the directory service provider interfaces” on page 16](#) for examples of the `DirectoryProviderConfig` object’s usage.

### GroupConfigBO interface

Classes implementing the `GroupConfigBO` interface store configuration information about the group provider. This information is obtained from the XML-based User Management configuration file (see [“Configuring User Management for custom directory service providers” on page 25.](#))

For your convenience, the User Management SPI provides a generic implementation called `GenericGroupConfigBO`.

### UserConfigBO interface

Classes that implement the `UserConfigBO` interface store configuration information about the user provider. This information is obtained from the XML-based User Management configuration file (see [“Configuring User Management for custom directory service providers” on page 25.](#))

For your convenience, the User Management SPI provides a generic implementation called `GenericUserConfigBO`.

## Testing the connection

This section describes how you can test the connection to the directory.

► **To test the connection to the directory:**

1. Log into the Adobe LiveCycle Administration Console and click **Settings > User Management > Domain Management**.
2. Under **Synchronize All Directories Manually**, click **OK**.

**Note:** This test will not call your `testConfiguration` method; it calls the `getPrincipals` method.

## Configuring User Management for custom directory service providers

When configuring User Management for custom directory service providers, you must also modify its configuration file.

► **To configure User Management for custom directory service providers:**

1. Log into the Adobe LiveCycle Administration Console and click **Settings > User Management > Configuration > Import and export configuration files**.
2. Click **Export** and, in the File Download dialog box, click **Open**. The config.xml file is downloaded.
3. Edit the config.xml file according to the directions provided in [“Configuring User Management to use directory service providers” on page 33](#).
4. Click **Browse** and then click **Import**. The config.xml file is uploaded to User Management.

# 4

## Registering Custom Service Providers

You must register custom service providers with User Management after you have developed them using the User Management SPI.

Most of the custom service providers that you can develop for User Management require you to configure User Management appropriately. For example, User Management requires some service providers to be associated with a domain. You will need to create a new User Management domain that includes settings for your custom service providers. This is because you cannot edit domains from the user interface that contain custom service providers. Otherwise, you will be unable to use the user interface to modify the LDAP settings in that domain.

This chapter contains the following information.

Topic	Description	See
User Management configuration settings	Describes how User Management configuration settings are implemented and explains the structure of the configuration file	<a href="#">page 26</a>
Defining domains for custom service providers	Describes the settings you need to configure to create a new User Management domain	<a href="#">page 28</a>
Configuring User Management to use custom authentication providers	Describes the settings you need to configure to integrate your custom authentication service provider with User Management	<a href="#">page 29</a>
Configuring User Management to use directory service providers	Describes the settings you need to configure to integrate your custom directory service provider with User Management	<a href="#">page 33</a>

### User Management configuration settings

User Management configuration settings include information about the service providers that User Management employ.

User Management can export the configuration settings to a file on your local file system. The settings are expressed in XML format. XML elements represent the different product components and their configuration settings.

Initially, XML includes settings only for the service providers packaged with User Management. You need to add settings for your custom service providers.

To retrieve the XML file (config.xml), log into the User Management web pages by using an administrator account and export the User Management configuration. After you edit the file, import it to apply the changes.

For more information about exporting and importing configuration settings, see the *User Management Help*.

**Caution:** When you are editing the config.xml file, change only the configuration settings that affect your custom service providers. When you import the file, User Management applies all the settings that the file defines.

## XML configuration file

The configuration file that User Management exports is structured so that each configurable aspect of the product is represented by a different XML element.

The following code shows the top levels of the config.xml file. The lowest level shown includes elements that represent the configurable components.

### Example 4.1 XML configuration file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE preferences (View Source for full doctype...)>
- <preferences EXTERNAL_XML_VERSION="1.0">
- <root type="system">
 <map />
- <node name="Adobe">
 <map />
- <node name="LiveCycle">
 <map />
- <node name="Config">
 <map />
- <node name="UM">
 <map />
 + <node name="AuthProviders">
 + <node name="AuthSchemes">
 + <node name="DirectoryServices">
 + <node name="DisplaySettings">
 + <node name="Domains">
 + <node name="GroupProviders">
 + <node name="SAML">
 + <node name="StorageProviders">
 + <node name="Synchronization">
 + <node name="UserProviders">
 </node>
 </node>
 </node>
 </node>
 </root>
</preferences>
```

The `AuthProviders`, `Domains`, `GroupProviders`, and `UserProviders` elements apply to custom service providers. For example, the `node` element that has the `name` attribute value of `Domains` contains an XML element for each User Management domain.

## XML element types in the configuration file

In general, the configuration file uses three element types to structure the configuration information:

- `node` elements represent configurable components in the environment. For example, a `node` element exists for each authentication provider that User Management uses. `node` elements each have a `name` attribute and contain other `node` elements, and must include at least one `map` element.
- `map` elements contain the elements that hold the property-value pairs that constitute configuration settings. The configuration settings are associated with the `map` element's parent `node` element. `map` elements can contain `entry` elements.
- `entry` elements use attributes to specify property-value pairs. `entry` elements have no content, but they have the attributes `key` and `value` that represent property names and values, respectively. An `entry` element can define only one property-value pair.

## Defining domains for custom service providers

Domains define different user bases. The boundary of a domain is usually defined according to the way your organization is structured or how your user store is set up.

User Management domains provide configuration settings that authentication providers and directory service providers use.

In the configuration XML that User Management exports, the `root` node that has the `name` attribute value of `Domains` contains an XML element for each domain defined for User Management. Each of these elements contain other elements that define aspects of the domain associated with specific service providers.

It is strongly recommended that you define a new domain that provides configuration settings for your custom service providers. For example, if you are implementing a custom authentication provider and a custom directory service provider that connect to the same user store, you should create a new domain.

The following example shows the structure of the XML element that represents a domain in the User Management configuration settings (the attribute values in italics are the values you can customize). Your domains should never be local (see the `map` entry whose `key` is `isLocal`).

### Example 4.2 Defining a domain

```
<node name="your_domain_name">
 <map>
 <entry key="name" value="your_domain_name"/>
 <entry key="description" value="a_description_for_your_domain"/>
 <entry key="isLocal" value="false"/>
 </map>
 <node name="AuthConfigs"> ... </node>
 <node name="DirectoryConfigs"> ... </node>
</node>
```

To create a new domain, insert similar XML tags into the exported configuration file. Place the tags within the `node` element that has the `name` attribute value of `Domains`.

**Note:** The `node` element that has the `name` attribute value of `AuthConfigs` contains configuration information that authentication providers need. The `node` element that has the `name` attribute value of `DirectoryConfigs` contains configuration information that directory providers need.

## Configuring User Management to use custom authentication providers

User Management configuration settings are used to register authentication providers, specify domains used for authentication, and associate authentication providers with domains.

You can also include domain information that your authentication provider requires at run time, such as parameters for connecting to the user store.

For information about User Management configuration settings, see [“User Management configuration settings” on page 26](#).

### Identifying authentication providers

The User Management configuration settings must identify each authentication provider that User Management uses.

Each authentication provider is identified by a corresponding element in the User Management XML configuration file. It is necessary to add an element to the first-level `AuthProviders` node that indicates which authentication providers should be globally loaded. These providers are associated with the specific domains listed in each of the child nodes.

Each domain element includes a `node` element that has the `name` attribute of the `AuthProviders` node. The `AuthProviders` node contains a separate `node` element that describes each authentication provider.

These nodes include elements that specify all the information that User Management requires to use the authentication provider, such as the name of the class that User Management should use to initiate the authentication provider and a pointer to the domain settings associated with the authentication provider.

The following example shows an XML `node` element within `AuthProviders`, arbitrarily named `SPIFB` in this case, that corresponds to an authentication provider. As shown by the highlighted portions, several relationships apply within both the `AuthProviders` node and the `Domains` node:

- The path specified in the `AuthProviders` node's entry named `configInstance` contains `SPIDom`, which is arbitrarily named and corresponds to an identically named node specified within `Domains`. It also corresponds to part of the path specified in the `AuthProviders` node's entry named `configInstance`.
- The `AuthProviders` node's `SPIFB` element corresponds to both the last part of the path specified in its entry named `configInstance` and an identically named node within the `SPIDom` node specified within `Domains`.
- The `SPIDom` node specified within `Domains` must be identical to the `value` used in its `map` node's `name` entry.

Because these relationships apply, be careful when renaming nodes to update the corresponding nodes.

### Example 4.3 Identifying an authentication provider

```
- <node name="AuthProviders">
 <map />
 + <node name="JAAS">
 + <node name="Kerberos">
 + <node name="LDAP">
 - <node name="SPIFB">
 - <map>
 <entry key="configured" value="true" />
 <entry key="enabled" value="true" />
 <entry key="visibleInUI" value="false" />
 <entry key="allowMultipleConfigs" value="false" />
 <entry key="className" value="com.adobe.idp.samples.spi.authentication.FBAuthenticationProviderImpl" />
 <entry key="configInstance" value="/Adobe/LiveCycle/Config/UM/Domains/SPIDom/AuthConfigs/SPIFB" />
 <entry key="order" value="2" />
 </map>
 + <node name="DefaultConfig">
 </node>
<node name="Domains">
 <map />
 + <node name="DefaultDom">
 - <node name="SPIDom">
 - <map>
 <entry key="description" value="SPI Testing Domain" />
 <entry key="name" value="SPIDom" />
 <entry key="isLocal" value="false" />
 </map>
 - <node name="AuthConfigs">
 <map />
 - <node name="SPIFB">
 - <map>
 <entry key="authProviderNode" value="/Adobe/LiveCycle/Config/UM/AuthProviders/SPIFB" />
 <entry key="FILENAME" value="c:\users500" />
 </map>
 </node>
 </node>
</node>
```

Therefore, to identify an authentication provider, insert similar tags into the exported configuration file. Place the tags within the `node` element that has the `name` attribute value of `AuthProviders`, and be aware of related names used in `Domains`.

Specify a name for the authentication provider in the `node` element you insert. The name must be unique to all authentication providers in the context of the configuration settings.

The following table describes the attribute values you need to provide for the `entry` elements inside the first `map` element.

Key name	Value
enabled	true or false Specifies whether User Management should use this provider. User Management uses this provider if <code>value</code> is <code>true</code> .
configured	true or false Specifies whether the domain has been configured for this authentication provider. If the domain has been configured, <code>value</code> should be <code>true</code> .

Key name	Value
<code>visibleInUI</code>	<code>true</code> or <code>false</code> Specifies whether this authentication provider should appear as a selectable option in the User Management web pages. For your authentication provider, <code>value</code> should be <code>false</code> .
<code>allowMultipleConfigs</code>	<code>true</code> or <code>false</code> Specifies whether more than one domain configuration instance exists for this authentication provider. If multiple configuration instances exist, <code>value</code> should be <code>true</code> .
<code>order</code>	Any integer, beginning at 1. This integer is reserved for future use.
<code>classname</code>	The fully qualified name of the class used to initiate this authentication provider. User Management calls the <code>authenticate</code> method of this class to request authentication.
<code>configInstance</code>	The value of the <code>name</code> attribute of the <code>node</code> element that contains the domain configurations for this authentication provider. You must provide cross-references to all the domains that use this authentication provider. This enables you to indicate which domain to use for this authentication provider. Each node element inside <code>AuthConfigs</code> represents a configuration instance.

## Configuring domains for authentication providers

The configuration settings for User Management domains include information to support each associated authentication provider. You must add configuration settings to the domain with which your authentication service provider is associated.

If you have not already set up a domain, you need to set one up immediately. For more information, see [“Defining domains for custom service providers” on page 28](#).

The following example shows a node that provides the domain configuration for an authentication provider and provides a reference from inside your domain up to the authentication provider entry. For a more complete example, see [“Configuring User Management to use directory service providers” on page 33](#).

### Example 4.4 Configuring a domain for an authentication provider

```
<node name="your_authentication_provider">
 <map>
 <entry key="authProviderNode"
 value="/Adobe/LiveCycle/Config/UM/AuthProviders/name"/>
 <entry key="additional_key_name" value="corresponding_value"/>
 </map>
</node>
```

To configure your domain, you must add a similar XML tag in the exported configuration file. Place the tag in the `node` element that has the `name` attribute value of `AuthConfigs`. This `AuthConfigs` node is in the domain that the authentication provider uses, as shown in [“Defining a domain” on page 28](#).

The `node` element for your authentication provider can include any number of properties that the provider requires. The `node` element must include `entry` elements that contain the two property-value pairs described in the following table.

Key name	Value
<code>authProviderNode</code>	The path to the node in the XML configuration file that represents your authentication provider.
<i>additional_key_name</i>	Any additional information that the authentication service provider needs at run time. Use this key name to create as many custom keys as you need to store the configuration information for your authentication provider to connect to the authentication system.

**Note:** Ensure that your domain for custom providers is uniquely named. Assume that the name will be treated in a case-insensitive manner.

In the following example, the two property-value pairs are `authProviderNode` and `FILENAME`. The `FILENAME` entry is an arbitrary example of a key that you may specify.

#### Example 4.5 Configuring a domain

```
<node name="Domains">
 <map />
+ <node name="DefaultDom">
- <node name="SPIDom">
 + <map>
 - <node name="AuthConfigs">
 <map />
 - <node name="SPIFB">
 - <map>
 <entry key="authProviderNode" value="/Adobe/LiveCycle/Config/UM/AuthProviders/SPIFB" />
 <entry key="FILENAME" value="c:\users500" />
 </map>
 </node>
 </node>
</node>
```

**Note:** Do not add children nodes; only first-level keys can be used.

## Configuring User Management to use directory service providers

After you download the XML file, edit the file according to the following example. Look for a node named `Domains` and add the `DirectoryConfigs` subnode after `</map>`, as shown in the example that follows.

**Note:** The highlighted entries must match corresponding nodes within the `GroupProviders` and `UserProviders` nodes, which are discussed in [“Configuring the custom group provider” on page 34](#) and [“Configuring the custom user provider” on page 34](#).

### Example 4.6 Configuring the domain to use custom directory service providers

```
<node name="Domains">
 <map />
+ <node name="DefaultDom">
- <node name="SPIDom">
 - <map>
 <entry key="description" value="SPI Testing Domain" />
 <entry key="name" value="SPIDom" />
 <entry key="isLocal" value="false" />
 </map>
+ <node name="AuthConfigs">
- <node name="DirectoryConfigs">
 <map />
 - <node name="sdkspi_DirectoryConfig01">
 <map />
 - <node name="NameDoesntMatterGroupConfig">
 - <map>
 <entry key="BATCH_SIZE" value="150" />
 <entry key="PRINCIPAL_FILE_NAME" value="c:\groups500" />
 <entry key="GROUPCONTAINMENT_FILE_NAME" value="c:\containment500" />
 <entry key="groupProviderNode" value="/Adobe/LiveCycle/Config/UM/GroupProviders/GroupSPIFB" />
 </map>
 </node>
 - <node name="NameDoesntMatterUserConfig">
 - <map>
 <entry key="BATCH_SIZE" value="150" />
 <entry key="PRINCIPAL_FILE_NAME" value="c:\users500" />
 <entry key="userProviderNode" value="/Adobe/LiveCycle/Config/UM/UserProviders/UserSPIFB" />
 </map>
 </node>
</node>
</node>
</node>
</node>
</node>
```

**Note:** The names of the nodes within `DirectoryConfigs` may be arbitrarily chosen. Therefore, `sdkspi_DirectoryConfig01`, `NameDoesntMatterGroupConfig`, and `NameDoesntMatterUserConfig` are arbitrary names. The only requirement is that the names within the same node depth must be distinct. In addition, the `GROUPCONTAINMENT_FILE_NAME` and `PRINCIPAL_FILE_NAME` entries are arbitrary examples of keys you may specify. In such cases, do not add children nodes; only first-level keys may be used.

To configure your custom group provider, specify the name of the class that implements the `DirectoryGroupProvider` interface. In the following example, this class is `FBDirectoryGroupProviderImpl`. Look for a node named `GroupProviders` and add the following subnode after `</map>`. In this case, it has been arbitrarily named `GroupSPIFB`. You may give it a name you prefer, but it must match the name in the path that is provided in the `groupProviderNode` entry within the `Domains` node, as shown by the yellow highlighted text in [Configuring the domain to use custom directory service providers](#) and in the following example.

#### Example 4.7 Configuring the custom group provider

```
<node name="GroupProviders">
 <map />
+ <node name="LDAP">
- <node name="GroupSPIFB">
 - <map>
 <entry key="className" value="com.adobe.idp.samples.spi.directoryservices.FBDirectoryGroupProviderImpl" />
 <entry key="enabled" value="true" />
 <entry key="configured" value="true" />
 <entry key="allowMultipleConfigs" value="true" />
 </map>
</node>
</node>
```

To configure your custom user provider, specify the name of the class that implements the `DirectoryUserProvider` interface. In the following example, this class is `FBDirectoryUserProviderImpl`. Look for a node named `UserProviders` and add the following subnode after `</map>`. In this case, it has been arbitrarily named `UserSPIFB`. You may give it a name you prefer, but it must match the name in the path that is provided in the `userProviderNode` entry within the `Domains` node, as shown by the blue highlighted text in [Configuring the domain to use custom directory service providers](#) and in the following example.

#### Example 4.8 Configuring the custom user provider

```
<node name="UserProviders">
 <map />
+ <node name="LDAP">
- <node name="UserSPIFB">
 - <map>
 <entry key="className" value="com.adobe.idp.samples.spi.directoryservices.FBDirectoryUserProviderImpl" />
 <entry key="enabled" value="true" />
 <entry key="configured" value="true" />
 <entry key="allowMultipleConfigs" value="true" />
 </map>
</node>
</node>
```

# 5

## Deploying Custom Service Providers

This chapter describes how to deploy a custom service provider to the J2EE application server on which User Management is deployed. You must perform the tasks described in this chapter in order for User Management to use the custom service provider that is created by using the User Management SPI.

To deploy your custom service provider, package it into a JAR file. After you create the JAR file, you must place the file into the LiveCycle.ear file by running Adobe Configuration Manager. After you run Configuration Manager, the custom service provider is deployed and User Management can use it to authenticate users.

This chapter provides the following information.

Topic	Description	See
Packaging your custom service provider	Explains packaging your Java application into a JAR file	<a href="#">page 35</a>
Repackaging the LiveCycle EAR file	Explains how to use Configuration Manager to repackage the LiveCycle.ear file to include the JAR file	<a href="#">page 35</a>

### Packaging your custom service provider

After you finish developing a custom service provider, package your Java project into a JAR file. Include all CLASS files that are located in your Java application in the JAR file. For information on creating a JAR file, go to <http://java.sun.com/>.

If your custom service provider is dependent on external JAR files, you must package these JAR files along with your custom service provider.

### Repackaging the LiveCycle EAR file

You must repackage the LiveCycle.ear file to include the JAR file(s) that contain your custom service provider and external JAR files that your custom service provider is dependent on. To repackage the LiveCycle.ear file, you run Configuration Manager, which is a wizard-like tool that automates the product configuration and assembly process. For more information about Configuration Manager see the *Installing and Configuring* guide that accompanies your LiveCycle product.

Before you run Configuration Manager, you must place your JAR file in the following directory:

```
C:\Adobe\LiveCycle\components\um\<app_server>\ext
```

where C:\ is the drive on which User Management is installed and *app\_server* is the J2EE application on which User Management is deployed. Assume that User Management is deployed on JBoss. In this situation, place your JAR file in the following directory:

```
C:\Adobe\LiveCycle\components\um\jboss\ext
```

**Note:** After you place the JAR in the appropriate directory, you can run Configuration Manager. It is recommended that you configure your LiveCycle product the same as you did when you originally installed it by selecting the same options that you selected when you first ran Configuration Manager.

## Deploying custom service providers

If User Management is deployed on JBoss, you must update the application.xml file to include the name of the JAR file(s) that make up your custom service provider. Configuration Manager merges the changes that you make to the application.xml file to the application.xml file that is located in the repackaged LiveCycle.ear file.

If User Management is deployed on IBM® WebSphere or BEA Weblogic, you must also perform an additional step. You must open the um.jar file and modify the manifest file to include references that specify all the JAR files that you placed in the LiveCycle.ear file. This step is not necessary if User Management is deployed on JBoss.

### ► To deploy a custom service provider on JBoss:

1. Copy the custom JAR file(s) to the C:\Adobe\LiveCycle\components\um\jboss\ext directory.
2. Update the application.xml file.
3. Run Configuration Manager to repackage the LiveCycle.ear file. For information, see [To repackage the LiveCycle.ear file by running Configuration Manager:](#)

### ► To deploy a custom service provider on WebSphere:

1. Copy custom JAR file(s) to the C:\Adobe\LiveCycle\components\um\websphere\ext directory.
2. Open the C:\Adobe\LiveCycle\components\um\websphere\ejb\um.jar file and edit the manifest to include the custom JAR file(s) in the **Class-Path:** entry.
3. Run Configuration Manager to repackage the LiveCycle.ear file. For information, see [To repackage the LiveCycle.ear file by running Configuration Manager:](#)

### ► To deploy a custom service provider on Weblogic:

1. Copy custom JAR file(s) to the C:\Adobe\LiveCycle\components\um\weblogic\ext directory.
2. Open the C:\Adobe\LiveCycle\components\um\weblogic\ejb\um.jar file and edit the manifest to include the custom JAR file(s) in the **Class-Path:** entry.
3. Run Configuration Manager to repackage the LiveCycle.ear file. For information, see [To repackage the LiveCycle.ear file by running Configuration Manager:](#)

### ► To repackage the LiveCycle.ear file by running Configuration Manager:

1. Navigate to the *LiveCycle root*/ConfigurationManager directory and start Configuration Manager:
  - (Microsoft® Windows®) Double-click **ConfigurationManager.exe**.
  - (Linux®) Run the configurationmanager.sh file.
2. On the Configuration Manager Welcome screen, click **Next**.
3. Select **Manual Configuration**, select one of the options below it, and then click **Next**.
4. Select the product(s) you want to configure and click **Next**.

For information about configuring your LiveCycle product, see the *Installing and Configuring* guide that accompanies your LiveCycle product.

5. Choose the application server that you are deploying to and click **Next**:
  - WebSphere (supported on Linux)
  - JBoss (supported on Windows)

6. On the Application Deployment Configuration screen, accept the following default information for the application that is being assembled, or type custom descriptions:
  - File name:** The default is LiveCycle.ear, or type another name for the final EAR file that you deploy.
  - Application context:** The default is LiveCycle, or use a descriptive name of your choice.
  - Application description:** The default is LiveCycle, or use a brief description of the application.
7. Select **User Management** in the product assembly. (You must select User Management because you are repackaging the LiveCycle.ear file with a JAR that contains a custom service provider for User Management.)

**Note:** This configuration dialog box appears only if LiveCycle Forms is selected.
8. On the Data Manager Module Configuration screen, select **Enable SSL** if you are using SSL security on your application server.
9. Type the SSL credential password. (If you have not yet set up your SSL credential, you can type a password here and use it when you create an SSL credential.)
10. (Optional) Enter a directory to use for Adobe LiveCycle products temp file.
11. Click **Next** and, on the Data Manager Module Configuration Continued screen, accept the default values for the following properties or type new values:
  - localDocumentStorageSweepInterval
  - globalDocumentStorageSweepInterval
  - defaultDocumentMaxInlineSize
  - globalDocumentStorageRootDir
  - defaultDocumentDisposalTimeout
12. (Optional) Select **globalDocumentStorageForceNFS** to set this property to `true`.
13. Click **Next** and, on the Font Manager Module Configuration screen, click **Next**.

The Configuration and Assembly Progress screen displays the progress of the configuration process. The Configuration and Assembly Details screen displays information about the product configuration and assembly.
14. Click **Next**, and then click **Finish** to exit Configuration Manager.

The deployable files are located in the following directory:

  - (Windows) *LiveCycle root*\ConfigurationManager\export
  - (Linux) *LiveCycle root*/lifecycle/ConfigurationManager/export
15. Redeploy the LiveCycle.ear file to the J2EE application server that is hosting your LiveCycle product. For information, see the *Installing and Configuring* guide that accompanies your LiveCycle product.
16. Restart your J2EE application server.

**Note:** If you originally used the JBoss turnkey option, you can use it again to repackage the LiveCycle.ear file. For information, see the chapter that discusses how to install your LiveCycle product by using the turnkey in the product's *Installing and Configuring* guide.

# Index

---

## A

- adding
  - exceptions to authentication response 14
  - user and group records to collection 19
- Adobe Configuration Manager 35
- AuthConfigBO interface 10
- AuthConfigBO objects 11
- authenticate method 8
- authentication
  - performing operations 12
  - process 6, 9
  - result errors 14
  - retrieving configuration information 11
  - retrieving values 10
  - sending results to User Management 14
- authentication providers
  - configuring domains for 31
  - configuring User Management to use 29
  - creating 8
  - identifying 29
- AuthProvider interface 10
- AuthResponse interface 10
- AuthResponse object, creating 14
- AuthResponseImpl class 10
- AuthScheme interface 10

## C

- checkValues method 12
- collecting user and group records 19
- config object 17
- ConfigFactory class 10
- configuration file
  - importing and exporting settings 26
  - modifying 25
  - structure of 27
  - XML element types 28
- Configuration Manager. *See* Adobe Configuration Manager
- connection, directory, testing 24
- creating
  - authentication providers 8
  - AuthResponse object 14
  - directory service providers 15
  - domains for custom service providers 28
  - Iterator object 11
  - JAR files 35
  - service providers 7
- custom authentication providers. *See* authentication providers
- custom service providers. *See* service providers

## D

- defining domains 28
- deploying custom service providers 36
- directory service providers
  - about 15
  - configuring User Management for 25
  - configuring User Management to use 33
  - implementing interfaces 16
  - interfaces 16
  - retrieving directory properties from configuration 23
  - sample implementation 15
- DirectoryGroupProvider interface 16, 21
- DirectoryPrincipalProvider interface 17
- DirectoryProviderConfig object 17, 21, 24
- DirectoryUserProvider interface 16, 21
- domains
  - configuring for authentication providers 31
  - configuring to use directory service providers 33
  - defining for custom service providers 28
- DSGroupContainmentRecord object 21
- DSPrincipalCollection class 17
- DSPrincipalIdRecord object 21

## E

- element types, configuration file 28
- entry elements 28

## G

- get method 10
- getCustomConfiguration method 24
- getDomain method 24
- getGroupConfig method 24
- getGroupMembers method 21
- getPrincipals method 17
- getUserConfig method 24
- grabBatchPrincipals method 17
- group providers
  - about 15
  - configuring 34
  - retrieving configuration information 24
- GroupConfigBO interface 24

## I

- Iterator object, creating 11

## J

- JAR file 7, 35
- Java API 8
- JBoss, deploying custom service providers on 36

## L

- LDAP directories 15
- library files 7
- LiveCycle.ear file, repackaging 35

## M

- map elements 28
- methods
  - authenticate 8
  - checkValues 12
  - get 10
  - getCustomConfiguration 24
  - getDomain 24
  - getGroupConfig 24
  - getGroupMembers 21
  - getPrincipals 17
  - getUserConfig 24
  - grabBatchPrincipals 17
  - setExceptions 14

## N

- node elements 28

## P

- packaging custom service providers 35

## R

- repackaging LiveCycle.ear file 35
- retrieving
  - authentication values 10
  - configuration information 11
  - group members 22
  - user and group provider configuration information 24
  - user and group records 16

## S

- service providers
  - about deploying 35
  - about registering 26
  - creating 7
  - defining domains for 28
  - deploying 36
  - packaging Java project into JAR file 35
- setExceptions method 14
- setting principal information 20
- SPI classes 10
- SPI interfaces 10
- state object 17

## T

- tab-delimited file, searching 12
- testing directory connection 24

## U

- um-spi.jar file 7
- user providers
  - about 15
  - configuring 34
  - retrieving configuration information 24
- UserConfigBO interface 24

## W

- WebLogic, deploying custom service providers on 36
- WebSphere, deploying custom service providers on 36

## X

- XML configuration file. *See* configuration file
- XML element types, configuration file 28



**Adobe**

## What's New

# Adobe® LiveCycle™ Forms

Version 7.2

- Platform Support [More ...](#)
- Improved Installation and Configuration Documentation [More ...](#)
- Improved Configuration Manager [More ...](#)

July 2006

## Platform Support

Adobe® LiveCycle™ Forms is now supported on a standard set of operating systems, application servers, and databases.

For a complete list of the supported application servers, operating systems, and databases, see the *Installing and Configuring LiveCycle for JBoss*, *Installing and Configuring LiveCycle for WebSphere*, or *Installing and Configuring LiveCycle for WebLogic* guide.

[Back to top](#)

## Improved Installation and Configuration Documentation

To improve the interoperability experience of LiveCycle products, the installation and configuration instructions for LiveCycle Forms have been combined with the instructions for Adobe LiveCycle Assembler, Adobe LiveCycle Form Manager, Adobe LiveCycle PDF Generator, Adobe LiveCycle Print, and Adobe LiveCycle Workflow.

The installation and configuration guides are specific to each application server:

- *Installing and Configuring LiveCycle for JBoss*
- *Installing and Configuring LiveCycle for WebSphere*
- *Installing and Configuring LiveCycle for WebLogic*

[Back to top](#)

## Improved Configuration Manager

LiveCycle Forms now includes an enhanced Adobe Configuration Manager. Using Configuration Manager, you can perform the following tasks:

- Configure and assemble LiveCycle products for deployment
- Configure your IBM® WebSphere® application server or BEA WebLogic Server® for LiveCycle products
- Validate application server settings for LiveCycle products
- Automatically deploy LiveCycle products to an application server
- Initialize database schemas for deployed LiveCycle products
- Verify deployed LiveCycle products are available and operational

[Back to top](#)

Adobe, the Adobe logo, and LiveCycle are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

BEA WebLogic Server is a registered trademark of BEA Systems, Inc.

IBM and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries, or both.

All other trademarks are the property of their respective owners.