IBM PRESS

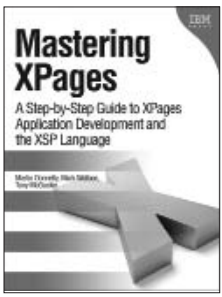
XPages

Portable Command Guide

A Compact Resource to XPages Application Development and the XSP Language

Martin Donnelly, Maire Kehoe,
Tony McGuckin, Dan O'Connor

Related Books of Interest



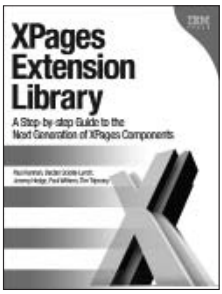
Mastering XPages

A Step-by-Step Guide to XPages Application Development and the XSP Language

By Martin Donnelly, Mark Wallace, Tony McGuckin

ISBN: 0-13-248631-8

The first complete, practical guide to XPages development—direct from members of the XPages development team at IBM Lotus. Martin Donnelly, Mark Wallace, and Tony McGuckin have written the definitive programmer's guide to utilizing this breakthrough technology. Packed with tips, tricks, and best practices from IBM's own XPages developers, *Mastering XPages* brings together all the information developers need to become experts—whether you're experienced with Notes/Domino development or not. The authors start from the very beginning, helping developers steadily build your expertise through practical code examples and clear, complete explanations. Readers will work through scores of real-world XPages examples, learning cutting-edge XPages and XSP language skills and gaining deep insight into the entire development process. Drawing on their own experience working directly with XPages users and customers, the authors illuminate both the technology and how it can be applied to solving real business problems.



XPages Extension Library

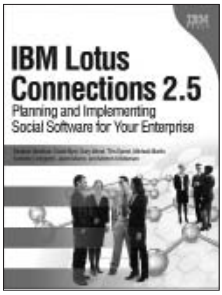
A Step-by-Step Guide to the Next Generation of XPages Components

By Paul Hannan, Declan Sciolla-Lynch, Jeremy Hodge, Paul Withers, Tim Tripcony

ISBN: 0-13-290181-1

The XPages Extensibility Framework is one of the most powerful application development features found in IBM Lotus Notes Domino. It enables developers to build their own artifacts and move far beyond XPages' out-of-the-box features. The XPages Extension Library is the greatest manifestation of this framework. A team of all-star XPages experts from inside and outside IBM show developers how to take full advantage of the XPages Extensibility Library and the growing portfolio of components built with them. The authors walk through installing and configuring the XPages Extension Library, integrating it with Domino Designer, and using new XPages components to quickly build state-of-the-art applications for web, the Notes client and mobile devices.

Related Books of Interest



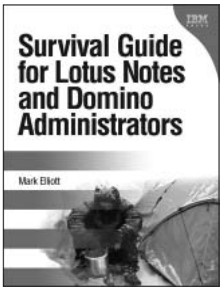
IBM Lotus Connections 2.5

Planning and Implementing Social Software for Your Enterprise

By Stephen Hardison, David Byrd, Gary Wood, Tim Speed, Michael Martin, Suzanne Livingston, Jason Moore, and Morten Kristiansen

ISBN: 0-13-700053-7

In *IBM Lotus Connections 2.5*, a team of IBM Lotus Connections 2.5 experts thoroughly introduces the newest product and covers every facet of planning, deploying, and using it successfully. The authors cover business and technical issues and present IBM's proven, best-practices methodology for successful implementation. The authors begin by helping managers and technical professionals identify opportunities to use social networking for competitive advantage–and by explaining how Lotus Connections 2.5 places full-fledged social networking tools at their fingertips. *IBM Lotus Connections 2.5* carefully describes each component of the product–including profiles, activities, blogs, communities, easy social bookmarking, personal home pages, and more.

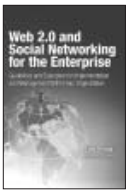


Survival Guide for Lotus Notes and Domino Administrators

By Mark Elliott

ISBN: 0-13-715331-7

Mark Elliott has created a true encyclopedia of proven resolutions to common problems and has streamlined processes for infrastructure support. Elliott systematically addresses support solutions for all recent Lotus Notes and Domino environments.



Web 2.0 and Social Networking for the Enterprise

Guidelines and Examples for Implementation and Management Within Your Organization

Bernal

ISBN: 0-13-700489-3

Related Books of Interest

The Social Factor

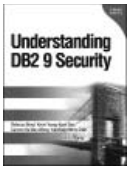
Innovate, Ignite, and Win through Mass Collaboration and Social Networking

By Maria Azua

ISBN: 0-13-701890-8

Business leaders and strategists can drive immense value from social networking "inside the firewall." Drawing on her unsurpassed experience deploying innovative social networking systems within IBM and for customers, Maria Azua demonstrates how to establish social networking communities, and then leverage those communities to drive extraordinary levels of innovation. *The Social Factor* offers specific techniques for promoting mass collaboration in the enterprise and strategies to monetize social networking to generate new business opportunities. Whatever your industry, *The Social Factor* will help you learn how to choose and implement the right social networking solutions for your unique challenges...how to avoid false starts and wasted time...and how to evaluate and make the most of today's most promising social technologies—from wikis and blogs to knowledge clouds.

Listen to the author's podcast at:
ibmpressbooks.com/podcasts



Understanding DB2 9 Security

Bond, See, Wong, Chan

ISBN: 0-13-134590-7

DB2 9 for Linux, UNIX, and Windows

DBA Guide, Reference, and Exam Prep, 6th Edition

Baklarz, Zikopoulos

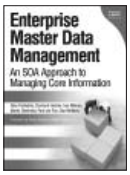
ISBN: 0-13-185514-X

The Art of Enterprise Information Architecture

A Systems-Based Approach for Unlocking Business Insight

Godinez, Hechler, Koening, Lockwood, Oberhofer, Schroeck

ISBN: 0-13-703571-3

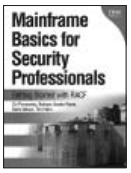


Enterprise Master Data Management

An SOA Approach to Managing Core Information

Dreibelbis, Hechler, Milman, Oberhofer, van Run, Wolfson

ISBN: 0-13-236625-8



Mainframe Basics for Security Professionals

Getting Started with RACF

Pomerantz, Vander Weele, Nelson, Hahn

ISBN: 0-13-173856-9

This page intentionally left blank

XPages Portable Command Guide

This page intentionally left blank

XPages Portable Command Guide

A Compact Resource to XPages Application Development and the XSP Language

Martin Donnelly, Maire Kehoe, Tony McGuckin, Dan O'Connor

IBM Press, Pearson plc

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City
ibmpressbooks.com

IBM Press Program Managers: Steven M. Stansel, Ellice Uffer

Cover design: IBM Corporation

Published by Pearson plc

Publishing as IBM Press

IBM Press offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside the U.S., please contact

International Sales
international@pearson.com

Library of Congress Cataloging-in-Publication Data

XPages portable command guide : a compact resource to XPages application development and the XSP language / Martin Donnelly ... [et al.].
p. cm.
Includes bibliographical references.
ISBN 978-0-13-294305-5 (pbk.)
1. XPages. 2. Application software--Development. 3. Web site development.
I. Donnelly, Martin, 1963-
QA76.625.X63 2012
006.7'6--dc23

2011047429

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing February 2012

ISBN-13: 978-0-13-294305-5
ISBN-10: 0-13-294305-0

Associate Publisher
Dave Dusthimer

Marketing Manager
Stephane Nakib

Executive Editor
Mary Beth Ray

Publicist
Heather Fox

Development Editor
Eleanor Bru

Managing Editor
Kristy Hart

Designer
Alan Clements

Project Editor
Anne Goebel

Copy Editor
Krista Hansing
Editorial Services, Inc.

Indexer
Lisa Stumpf

Compositor
Nonie Ratcliff

Proofreader
Debbie Williams

Manufacturing Buyer
Dan Uhrig

Dedications

To the memory of my parents, Betty and Paddy, whose love and support I will always cherish.

—Martin

To my parents and my husband, Nelius, for all their support.

—Maire

To Martin: Once again, you pulled us over the line! You deserve a medal.

To my parents and family: I love you all and hope you enjoy reading another great book about XPages!

For "my two girls," Paula and Anna-Rose: a beautiful wife and special daughter who mean absolutely everything to me!

—Tony

Dedicated to the memory of my parents, Peter and Rita—I miss you both.

To my family—in particular, my wife, Anne Marie, and daughter, Aileen—my contribution to this book would not have been possible without your support and encouragement. I love you both.

Finally, to my coauthors—thank you for putting faith in a "Designer developer" to contribute to this fine book!

—Dan

This page intentionally left blank

Contents

Introduction

Welcome to the XPages Portable Command Guide! This book is designed, for the most part, as a quick information guide for XPages developers with some real-world experience under their belts. It focuses on the road less traveled—**xsp.properties** parameters, **notes.ini** settings, XSP JS object functions, and such. In other words, it covers the little-known magic bullets that are not well documented but invariably help get you out of a programming bind. In that sense, it is an ideal companion for more holistic tomes such as *Mastering XPages,* which is designed to give broad coverage to the runtime and application development experience in general. Having said that, this book does dive into detail, when appropriate—after all, the authors are developers, so we just can't help ourselves!

XPages is a rich and powerful application development framework for Notes/Domino, first introduced in version 8.5 at Lotusphere 2009. Since that time, XPages has gone from strength to strength, with three further release updates, an open source XPages Extension Library, a dedicated IBM XWork server, a best-selling IBM Press book, and many other initiatives and innovations. We hope this Portable Command Guide helps add to the general success of XPages by bringing new information to the community and making application development a little bit easier for all concerned.

Reading Audience

This book is for XPages developers with some practical experience. Neophytes are advised to start with a more general book, such as *Mastering XPages,* or perhaps to use this book as its companion guide.

Conventions

Any programming code, markup, or XSP keywords are illustrated in numbered listings using a fixed width font.

User interface elements (menus, links, buttons, and so on) of the Notes client, Domino Designer, or any sample applications are referenced using a **bold** font. So too are file system paths, locations, and artifacts, such as the **notes.ini** and **xsp.properties** files.

Important words and phrases are emphasized using an *italic* font.

Visual representations of the design-time experience or runtime features are typically captured as screen shots and written up as numbered figures, using super-imposed callouts where appropriate.

In general, chapters feature a summary table of XPages commands, parameters, or properties near the beginning and seek to explain these in brief, concise terms. These items, or important subsets thereof, are typically then given more expansive

treatment in the rest of the chapter. Most chapters also have an accompanying NSF sample application containing practical examples that can be perused using Domino Designer and run in preview mode for the web or Notes client. These samples are available online for download at the following website: www.ibmpressbooks.com/title/0132943050

The samples are based on the latest release of XPages available at the time of writing (version 8.5.3), although many examples work with earlier releases. Visit this website to download the no-charge version of Domino Designer 8.5.3: www.ibm.com/developerworks/downloads/ls/dominodesigner/

How This Book Is Organized

This book is divided into six chapters, to separately address the many different aspects of XPages software development in as logical a manner as possible.

- **Chapter 1, "Working with XSP Properties,"** gives you all the details you need to locate, edit, and load the **xsp.properties** file, and thus configure the XPages runtime for your own specific requirements. An XSP property is a simple parameter definition that can modify the behavior of the XPages runtime in "magical" ways.
- **Chapter 2, "Working with Notes/Domino Configuration Files,"** concerns itself with the practical business of identifying the **notes.ini** settings that have particular relevance to XPages and explains their usage in detail.
- **Chapter 3, "Working with the Console,"** gives an overview of the many ways you can interact with the XPages runtime at the console level for runtime analysis, troubleshooting, or application debugging.
- **Chapter 4, "Working with the XSP Client Side JavaScript Object,"** examines the XSP Client Side JavaScript Object and lists simple examples of all the publically exposed functions that that can be used in an XPage. It also provides a general overview of Client Side JavaScript scripting techniques and other miscellaneous features relevant to XPages development.
- **Chapter 5, "Server-Side Scripting,"** gives an overview of Server Side JavaScript scripting objects and supporting libraries. This chapter also examines ways to integrate custom Java classes and create Managed Beans.
- **Chapter 6, "Server-Side Debugging Techniques,"** provides detail on setting up a debug and logging environment for your XPages applications. It also explains the details of stack traces and how you can analyze and decipher such information when troubleshooting an application.
- **Appendix A, "Definitive Resources,"** points to a collection of definitive reference sources that describe all the details of the XSP tags and Java and JavaScript classes. It also points to specification documents that define the technologies that XPages consumes or extends.

- **Appendix B, "Useful Online Resources,"** gives a snapshot of the authors' favorite XPages websites at the time of writing. This list of sites should help you find whatever you need to know about XPages that you cannot find in this book.
- **Appendix C, "Make Your Own Journal,"** provides blank pages for you to add your own specific notes on settings, markup, code fragments, or whatever else you need that might not be listed in this book.

Acknowledgments

We would like to start by thanking our two very thorough and knowledgeable technical reviewers, Mark Wallace and David Taieb. Thanks to you both for keeping us honest and for providing invaluable feedback—*most* of which we included here. ;-)

A big and sincere thank you to all those in the Notes/Domino application development leadership team for supporting this project—especially to Eamon Muldoon, Pete Janzen, Maureen Leland, Peter Rubinstein, and Philippe Riand.

Behind us are some very special teams of people—particularly the XPages runtime team in IBM Ireland and the Domino Designer team in Littleton, Massachusetts. Each member of these teams has unique strengths and skills, which we have completely exploited over the course of writing this book. The user experience and documentation teams also worked closely with us and helped bring clarity and objectivity to all we do. Our thanks to all: Andrejus Chaliapinas, Brian Gleeson, Darin Egan, Edel Gleeson, Graham O'Keeffe, Greg Grunwald, Jim Cooper, Jim Quill, Kathy Howard, Lisa Henry, Lorcan McDonald, Mark Vincenzes, Michael Blout, Mike Kerrigan, Padraic Edwards, Paul Hannan, Robert Harwood, Robert Perron, Simon McLoughlin, Teresa Monahan, and Vin Manduca.

It was once again a tremendous privilege for us to work with our friends at IBM Press, particularly Mary Beth Ray, Ellie Bru, Anne Goebel, Vanessa Evans, and Chris Cleveland. On the IBM side, Steven Stansel and Ellice Uffer worked tirelessly on getting the message out there for the *Mastering XPages* book and are already beating the drum for this one! Thanks for the help and the fun along the way.

Finally a great big thank you as always to our customers and business partners for continuing to explore new ground with XPages and driving further adoption of this most truly wonderful technology. Viva XPages!

About the Authors

The authors of this book have a few things in common. All four hail from Ireland, work for the IBM software group, and have made significant contributions to the development of both XPages and Domino Designer.

Martin Donnelly is a software architect and tech lead for the XPages runtime team in IBM Ireland. He graduated with a Bachelor of Commerce degree from University College Cork in 1984 and later completed a Master's degree in Computer Science at Boston University (2000). Martin has worked on all XPages releases, from Notes/Domino 8.5 through 8.5.3, and also worked on a precursor technology: XFaces for Lotus Component Designer. In the 1990s, while living and working in Massachusetts, he was a lead developer on Domino Designer. Now based once again in Ireland, Martin lives in Cork with his wife, three daughters, and two greyhounds. Despite the fact that he should have hung up his boots years ago, he still persists in playing soccer on a weekly basis and enjoys salmon angling during the summer when the opportunity presents itself.

Maire Kehoe is a senior software engineer in the IBM Ireland software lab. She completed an Honors Bachelor of Science degree in Computer Applications in Dublin City University (DCU) and began working for IBM in 2003. She worked on the Lotus Component Designer product from 2004 to 2007 and moved to IBM Lotus Domino to help develop the XPages runtime for the Domino server. Maire lives in Dublin with her husband and enjoys travel and musicals (and tea).

Tony McGuckin is a senior software engineer in the IBM Ireland software lab. After studying Software Engineering at the University of Ulster, he began his career with IBM in 2006, working in software product development on the Lotus Component Designer runtime. He then transitioned into the XPages core runtime team when XPages was born. When not directly contributing to the core runtime, Tony is kept busy with research and development of the next generation of IBM software development tools, as well as middleware, conferencing, and consultancy. Outside the lab, Tony enjoys food, wine, and cooking; recently acquired a curious taste for classical music; and likes to get off the beaten track to take in Irish scenery and wildlife.

Dan O'Connor is a senior software engineer in the Littleton, Massachusetts, software lab. He graduated with a Bachelor of Engineering degree in Computer Engineering from the University of Limerick, Ireland in 2000. He joined IBM through Lotus Software in Cambridge, Massachusetts, in 2000. Since then, Dan has worked on different projects, but most have focused on Eclipse and JavaServer Faces. In 2002, he joined the Rational Application Developer team to work on a "new" technology called JSF. In 2006, he rejoined the Lotus division to work on Lotus Component Designer and moved to Domino Designer in 2008 as the UI team lead. Dan lives in Milton, Massachusetts, with his wife and daughter. In his spare time, he spends too many hours following Gaelic football and occasionally dabbles in "home improvement," much to the profit of the local plumber!

CHAPTER 1

Working with XSP Properties

For most XPages developers, the process of building knowledge and expertise begins inside Domino® Designer. Typically, a neophyte developer loads Domino Designer, learns to create a new XPage, and then quickly figures out how to drag and drop controls from palette to page and assign values to these controls via the various property panels. From there, the natural progression is to discover how to program the controls dynamically, using JavaScript or one of the other languages available within the XPages framework. Pretty soon, a developer can be laying out fully fledged pages that link and combine in clever ways to form an impressive application. At that stage, it is common for a developer to feel that the XPages learning curve is complete and all that remains is to roll out applications and await the plaudits of the user community. Unfortunately, the feel-good factor is often short lived and a swift reality check is delivered by those tasked with appraising the initial versions of applications. New and unforeseen factors always come to the fore when an application is unveiled to users, and at this point the less experienced developer becomes acutely aware of the need for a whole new set of tools that can help tune and adapt an application to the many and varied nuances of real-world usage. This is when a collection of XPages framework parameters, known as **xsp.properties**, become the new best friend of the XPages developer!

XPages is a rich and extensive application framework that supports applications on a number of different runtime platforms, such as web browsers, the Notes® client, and mobile devices (smartphones, iPads, and so forth). The best applications are inevitably those that can deliver the required core functionality across all platforms, while at the same time leveraging the best unique features of each individual platform at runtime. Often it is possible to deliver such smart behavior programmatically, such as by dynamically detecting the runtime environment and adapting the application markup appropriately; other times, it is more effective to simply have parameters that dictate the appropriate behavior in a particular context. Diversity of runtime platforms is just one example of the need for runtime adaptability. An application designed to work a particular way out of the box might require different adaptations to satisfy varying customer requirements, whether they are driven by divergent performance and scalability metrics or simply by differing consumer expectations of application runtime behavior. Whatever the driving force is, XPages requires a means of modifying its behavior to flexibly adapt to different well-known use cases. The collection of parameters defined in its **xsp.properties** file is a primary tool for doing just that.

To get more concrete about what **xsp.properties** is and what it can do for you, Table 1.1 provides a high-level summary definition of all such parameters available within the XPages framework as of Notes/Domino V8.5.3. This chapter explores all 47 of these parameters in detail, along with some practical examples of how and why they can be applied to solve common problems. First, however, you should download PCGCH01.nsf

and open it in Domino Designer so that you can be ready to explore some of the hands-on sample XPages this chapter covers. All the sample NSFs are available at this website: www.ibmpressbooks.com/title/0132943050

Table 1.1 xsp.properties

Category/Name	Description
Timeout	
xsp.application.timeout	Defines when an application is discarded from memory after a period of user inactivity. Takes a numeric value representing time expressed in minutes.
xsp.application.forcefullrefresh	Boolean property that, when set to true, requests a full application refresh when the design of a class changes. The default value is false.
xsp.session.timeout	Defines when a user session is discarded from memory after a period of user inactivity. Takes a numeric value representing time expressed in minutes.
xsp.session.transient	Boolean property. `true` means that the sessions, and thus the pages, are not persisted between requests.
Theme	
xsp.theme	The name of an XPages theme that an application uses by default. The default theme is named **webstandard**.
xsp.theme.web	The name of an XPages theme to use when running on the web. If not assigned an explicit value, the `xsp.theme` value is used.
xsp.theme.notes	The name of an XPages theme to use when running on the Notes client. The `xsp.theme` value is used when a value is not explicitly assigned.
Resource Aggregation	
xsp.resources.aggregate	Boolean value indicating whether resources defined on an XPage should be automatically aggregated wherever possible to optimize download time.
File Upload	
xsp.upload.maximumsize	Controls the maximum size of any file that can be uploaded as a document attachment. Takes a numeric value representing the file size expressed in kilobytes, with the default value being 1024 (that is, 1024KB, equating to 1MB).

Category/Name	Description
xsp.upload.directory	Accepts a string value that identifies a folder to be used as a temporary storage location when uploading files. Deprecated in favor of `xsp.persistence.dir.xspupload`.
JSF Persistence	
xsp.persistence.discardjs	Boolean property indicating whether to discard the JavaScript context for a page after the page is processed. Set to `true` by default.
xsp.persistence.mode	Accepts a string value that defines how XPages are persisted according to how a user uses an application: on disk, in memory, or a hybrid model that persists pages to disk except for the latest page, which is stored in memory. The string values that represent these modes are `file`, `basic`, and `fileex` respectively.
xsp.persistence.tree.maxviews	Takes a numeric value defining the maximum number of pages to be persisted when the in-memory mode is used. The default value is 4 pages.
xsp.persistence.file.maxviews	Takes a numeric value defining the maximum number of pages to be saved when the on-disk or hybrid persistence mode is used. The default value is 16 pages.
xsp.persistence.viewstate	Defines what information is persisted for a given XPage, namely the full component tree, nothing at all, or just the changes to the page. The string values that represent these modes are `fulltree`, `nostate`, `delta`, and `deltaex`, respectively.
xsp.persistence.file.gzip	Boolean property indicating whether the persisted files should be compressed to GZIP format on disk (less disk space, more CPU processing). Defaults to `false`.
xsp.persistence.file.async	Boolean property indicating whether file persistence should be done asynchronously. Defaults to `true`.
xsp.persistence.file.threshold	Number property indicating whether pages less than the given size should be saved in memory instead of on disk. The threshold is in bytes. Defaults to `0`.
xsp.persistence.dir.xspstate	String value that defines the directory where the JSF pages are persisted.

Category/Name	Description
xsp.persistence.dir.xspupload	String value that defines the directory where the temporary uploaded files are stored. Supersedes the `xsp.upload.directory` property.
xsp.persistence.dir.xsppers	String value that defines the directory where the document attachments are temporarily persisted.
Client Side JavaScript	
xsp.client.script.dojo.version	Notes/Domino can have multiple versions of the Dojo toolkit installed. This property accepts a string value that identifies a specific version to use; if no string value is given, XPages uses the latest version.
xsp.client.script.dojo.djConfig	String value used to add parameters to the `djConfig` attribute of Dojo. For example, to load Dojo in debug mode, apply this setting: `xsp.client.script.dojo.djConfig=isDebug:true`.
HTML Page Generation	
xsp.html.doctype	String value that defines the document doctype XPages generates. Defaults to HTML 4.01 transitional.
xsp.html.meta.contenttype	Boolean property. A value of `true` requests that XPages generate a `<meta>` tag in the HTML header defining the content type and the optional character set. This meta tag is the first tag that appears in the `<head>` section of the rendered HTML page.
xsp.html.preferredcontenttypexhtml	Boolean property indicating whether to force the content type to be `application/xhtml+xml`, if the user agent supports it. Defaults to `false`.
xsp.html.page.encoding	String value defining the character set returned for the page. The default is `utf-8`.
xsp.compress.mode	String value defining the compression mode used when a page is rendered. Valid values are `none`, `gzip` (default), and `gzip-nolength`.
xsp.client.validation	Boolean property indicating whether to enable client-side error validation. Client-side validation is performed before the page is submitted. Defaults to `true`.

Category/Name	Description
xsp.redirect	Boolean property indicating whether the XPages runtime should update the user agent when redirecting to a new page—for example, when performing a page navigation. Defaults to `true`.
Error Management	
xsp.error.page	String value used to identify the name of a custom error page to display any runtime errors. If blank, XPages provides a default error page.
xsp.error.page.default	Boolean property that defines whether the XSP layer should display the default error page. Defaults to `false`.
User Prefences	
xsp.user.timezone	Boolean property. A value of `true` means that the user's time zone, as detected in the client browser, is used when date objects are displayed in text form on an XPage. A `false` or undefined value means that the server time zone is used.
xsp.user.timezone.roundtrip	Boolean property indicating that the user time zone is not used in the initial display of any XPages in the application, allowing an optimization to avoid a round-trip to the server for computing the time zone.
AJAX Options	
xsp.ajax.renderwholetree	Boolean property indicating whether the JSF tree should be completely processed during the render phase. Defaults to `true`.
Script Caches Size	
ibm.jscript.cachesize	Controls the number of compiled JavaScript expressions. Accepts a numeric value. The default value is `400`.
ibm.xpath.cachesize	Numeric value used to control the number of compiled XPath expressions.
Active Content Filtering	
xsp.richtext.default.htmlfilter	String value that defines which filter to use by default for controls that output user-entered HTML values, such as the Computed Field control.
xsp.richtext.default.htmlfilterin	String value that defines which filter to use by default for controls that allow editing HTML values, such as the rich text control.

Category/Name	Description
xsp.htmlfilter.acf.config	String value that identifies the Active Content Filtering (ACF) library configuration file to use.
Resources Servlet	
xsp.expires.global	Defines the default expiration duration for global resources. Accepts a numeric value expressing the period in days. Defaults to 10 days.
Repeating Controls	
xsp.repeat.allowZeroRowsPerPage	Boolean property that controls whether empty pages are allowed in repeating controls—for example, a View panel, a Repeat, or a Data table.
Partial Update Timeout	
xsp.partial.update.timeout	Accepts a numeric value that defines the time-out for partial update operations. Defaults to 20 seconds.
Link Management	
xsp.save.links	String value that defines whether native links are saved in Notes (client) or Domino (web) format. Valid values are `UseNotes` and `UseWeb`, respectively. These are case sensitive.
xsp.default.link.target	Accepts a string value that can be used as a default link target value for all links in the application. Valid values are `_self` (open target in same window) and `_blank` (open target in new window).
Control Libraries	
xsp.library.depends	A comma-separated and trimmed list of library IDs (strings) that an application depends on to run.
Composite Data	
xsp.theme.preventComposite DataStyles	Boolean property controlling how theme property values are applied to Custom Control `style` and `styleClass` properties.

Table 1.1 groups all these magic runtime switches and levers into various categories. This chapter explores each of these categories later. A cursory skimming of the summary definitions indicates that these properties do not all fall solely within the exclusive domain of the Domino application developer, but many are appropriate to Notes/Domino administrators also. For instance, many properties offer a simple means of executing administrative tasks, such as fine-tuning performance, applying security rules, limiting

document size, and so forth. First and foremost, however, you need to find this **xsp.properties** file!

Locating and Updating xsp.properties

One interesting point about the **xsp.properties** file is that there is potentially more than one of them. Every XPages NSF application contains an **xsp.properties** file, and you likely will find an **xsp.properties** file on your Domino server and/or Notes client installation also. Where exactly are these files located? You can start with the properties file that is embedded directly in every XPages application.

XPages applications are standard web applications based on the J2EE specification, and J2EE-compliant web applications typically place configuration files inside a standard private **WEB-INF** folder. XPages adheres to this by locating its **xsp.properties** file inside the **WEB-INF** folder inside the NSF. The Domino Designer perspective does not expose the raw Java™ project structure that underlies your XPages application, but you can view it by switching perspective or by including other Eclipse views in your Domino Designer perspective. To do the latter, select the **Window > Show Eclipse Views > Other** menu and then choose Package Explorer view from the Java category. In Domino Designer V8.5.3, this adds a new tab adjacent to the Application Navigator, from which you can explore *all* project elements. After you have opened the Package Explorer, find your current NSF, expand it, and then peruse its contents. Open the **Web Content\WEB-INF** folder to locate the **xsp.properties** file, and then double-click to open the file. Figure 1.1 shows a simple **xsp.properties** file in a plain text editor. You can view and/or modify the file contents directly from this editor.

Luckily, viewing and updating the embedded **xsp.properties** file is not always as cumbersome as just described. Domino Designer makes it easy to work with most of the more commonly used settings. You might have already been setting and modifying property values within the local **xsp.properties** file and not been aware of this process. Many of these XPages properties, along with other Notes/Domino properties, are surfaced in a single general-purpose Application Properties editor. Look for an outline entry of the same name in the Application Navigator and double-click it to activate the multitab editor. Selecting the XPages tab in the bottom editor pane gives access to the properties. Figure 1.2 shows a sample, with some property mappings highlighted.

The embedded **xsp.properties** file specifies parameter values that apply to the application that contains it. The Domino server also has an **xsp.properties** file so that parameter values can be applied to all applications loaded on the server. A sample file, appropriately named **xsp.properties.sample**, is installed automatically as part of the Domino installation process; it is located in the **data\properties** folder under the Domino root directory. (Note that this is the location on MS Windows® platforms; it can vary on other systems.) You can edit this file with a standard text editor. It contains all the properties listed in Table 1.1. All property assignments are commented out—that is, the line is prefixed with a # character, as follows:

```
#xsp.application.timeout=30
```

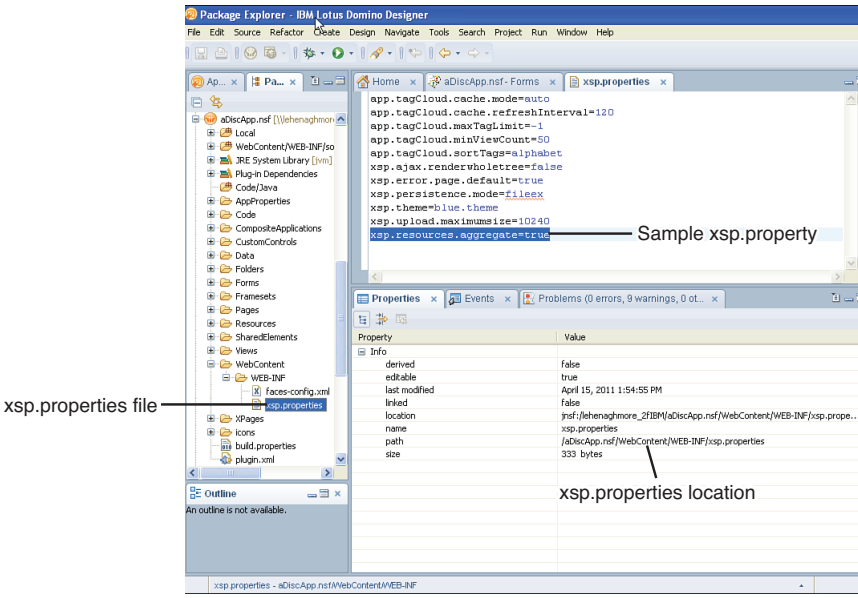


Figure 1.1 Accessing xsp.properties from the Eclipse Package Explorer

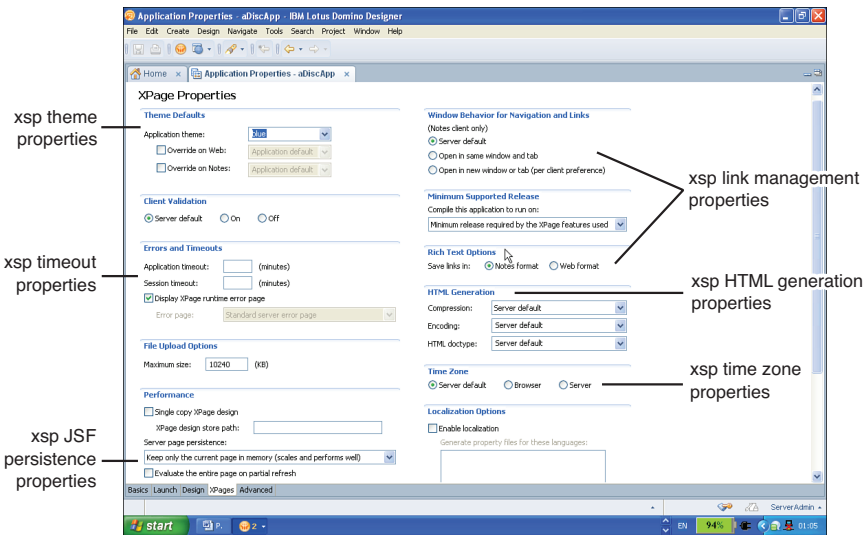


Figure 1.2 Accessing xsp.properties from the Domino Designer Applications Editor Explorer

The default value is typically demonstrated as a default assignment. To uncomment a line, simply delete the # character prefix. To enforce a property, do the following:

1. Rename the file from **xsp.properties.sample** to **xsp.properties**, if not already done.
2. Edit the **xsp.properties** file, uncomment the line, and apply the desired value.
3. Restart the server (or just the http server task), depending on the chosen property.

To restart just the http task on the server, you can enter `restart task http` in the Domino console window. Alternatively, just type `tell http q`, followed by a `load http` command.

> **TIP** If you are using clusters, you must repeat the operation for all servers in the cluster. Server **xsp.properties** file changes are not automatically replicated between servers of the same cluster.

Similarly, the Notes client has an **xsp.properties** file where you can apply settings on a client-wide basis. If the **xsp.properties** file has not been used, it is named **xsp.properties.sample**; in this case, you must rename it to **xsp.properties** before you can apply any properties. You can find this file in the same **data\properties** subfolder structure under the Notes root directory; and the same rules and principles apply when it comes to enforcing settings.

As described in Step 3, you might or might not have to restart your Domino server or Notes client for a given property value change to take effect. This depends on the individual nature of a given property: Some are static and require a restart when changed, whereas others are dynamic and are reread when next executed.

The Timeout Properties

The XPages runtime creates and manages an application session whenever a user initially requests an application. Thereafter, subsequent requests from any user, regardless of how many, cause that application to use the initial application session object for storing application-scoped objects. Furthermore, the XPages runtime creates and manages a user session for each user of any given application. Each user is associated with a unique user session object within the context of the current application.

This category of **xsp.properties** facilitates the management of application and user session timeout durations, as well as determines the way in which the XPages runtime maintains the user session and application object. This maintenance relates to how a user session is serialized between page requests and also to how the objects representing an application are recycled and reinstantiated between page requests based on design element changes.

xsp.application.timeout

If you look for the `xsp.application.timeout` setting in the **xsp.properties** file, you will find the snippet shown in Listing 1.1.

Listing 1.1 xsp.properties Snippet for the xsp.application.timeout Property

```
# Application timeout management defines when an application is
# discarded from memory after a period of inactivity expressed
# in minutes
#xsp.application.timeout=30
```

By default, a Domino server boots up without any XPages applications in memory. You can learn more on how to preload XPages applications in Chapter 2, "Working with Notes/Domino Configuration Files." Only when the XPages runtime processes a user request for a given XPages application is an application loaded into memory and processed to serve the incoming request. Thereafter, that application resides in memory within the XPages runtime until all user sessions associated with that application have been discarded from the XPages runtime and the `xsp.application.timeout` duration has been exceeded. As shown in Listing 1.1, the default timeout value is 30 minutes. This property can be set at the global server level or, alternatively, within an application. Figure 1.3 shows where this can be set within Designer for an application.

xsp.session.timeout

If you look for the `xsp.session.timeout` setting in the **xsp.properties** file, you will find the snippet shown in Listing 1.2.

Listing 1.2 xsp.properties Snippet for the xsp.session.timeout Property

```
# Session timeout management defines when a user session is
# discarded from memory after a period of inactivity expressed
# in minutes
#xsp.session.timeout=30
```

When a user first issues a request for an XPages application, following successful authentication, a user session object is instantiated and associated with that user. This user session object is used to store temporary data created during that user's usage of the application. By default, no explicit mechanism exists for discarding a user session object when a user closes a browser or otherwise exits the system. Therefore, this `xsp.session.timeout` property defines a timeout period for discarding a user session object from server memory, based on a period of inactivity. This is expressed in minutes, with 30 minutes being the default period.

This property can be set at the global server level or, alternatively, within an application. Figure 1.4 shows where this can be set within Designer for an application.

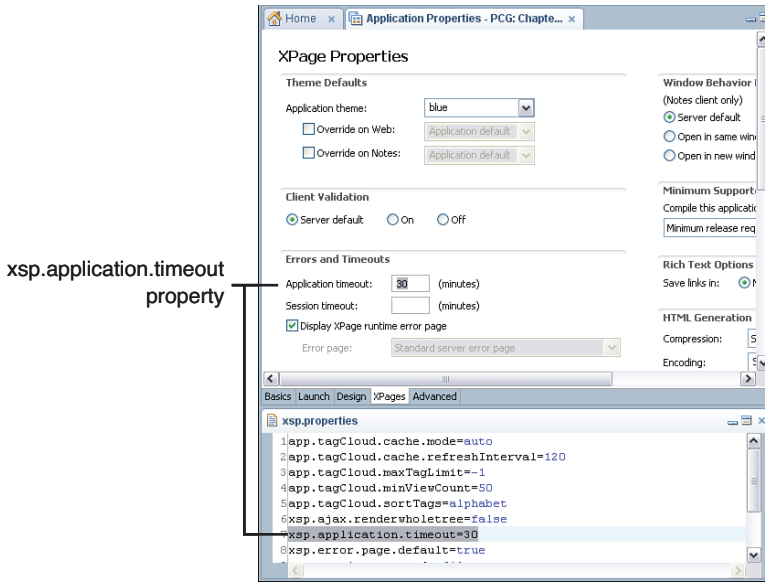


Figure 1.3 The xsp.application.timeout property exposed in the application properties editor in Designer

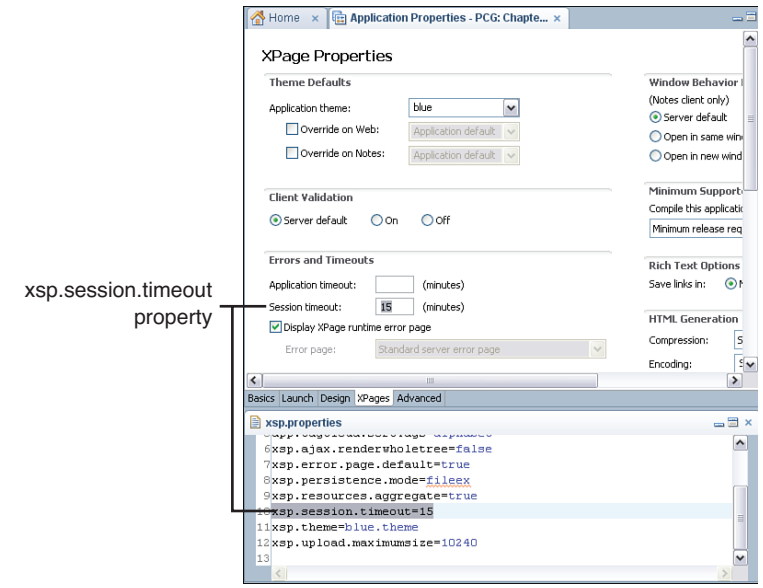


Figure 1.4 The xsp.session.timeout property exposed in the application properties editor in Designer

xsp.session.transient

This property is new in Notes/Domino 8.5.3.

By default, the XPages runtime is a stateful web application framework. A request for an XPage results in a degree of server-side processing that begins with creation or retrieval of a user session and ultimately ends with a rendering process that builds up the content for a response. During this server-side processing, a user session configuration object, along with all the controls on a requested XPage, have their respective properties and values serialized to disk and/or deserialized from disk. This is due to the inbuilt serialization mechanism of XPages that manages and provides the stateful characteristics of the XPages runtime. Based on application requirements, it might be beneficial from a performance and scalability perspective for an application not to participate in this serialization process, to optimize its level of participation. This aim of the `xsp.session.transient` property is to provide a way to control how user session objects are serialized between requests.

If you look for the `xsp.session.transient` setting in the **xsp.properties** file, you will find the snippet in Listing 1.3.

Listing 1.3 xsp.properties Snippet for the xsp.session.transient Property

```
# Transient sessions means that the sessions, and thus the pages,
# are not persisted between requests
#xsp.session.transient=false
```

By default, the XPages runtime sets this property to `false`. Therefore, the serialization process includes all user session objects, but not the sessionScope object. This means that any XPages a given user requests are serialized/deserialized in association with the user session object over the life of that user session object. They are discarded along with the user session object when the overall user session timeout duration passes.

Alternatively, if this property is set to `true`, the XPages runtime automatically avoids serializing user session objects between XPage requests. It is important to note that a user session object still is instantiated for a request, but it simply is not serialized between requests. This also means that properties and values of controls within requested XPages still participate in the serialization process—this ensures that an XPage can still provide a rich user experience for the scoped variables and partial execution of actions, for example. However, when a user navigates to another XPage, the associated stateful data for that XPage is discarded because the user session object is not serialized between requests. This feature is made available for use cases that require an extremely optimal level of performance tuning where server memory must be finely managed. Note that such use cases are those in which partial updates are applied against only the current page; full page refreshes cause the state to be discarded between requests. Therefore, the design and intent of the page require careful consideration to benefit from this feature.

xsp.application.forcefullrefresh

This property was introduced in Notes/Domino 8.5.3. It is set to `false` by default and is particularly useful during the development phase of an XPages application. Listing 1.4 shows the relevant section of the **xsp.properties** file.

Listing 1.4 xsp.properties Snippet for the xsp.application.forcefullrefresh Property

```
# Application refresh when this property is set to true, then a
# full application refresh is requested when the design of a
# class changes (means that all the data is discarded in scopes)
#xsp.application.forcefullrefresh=false
```

The property ensures a full refresh of all data and objects stored within the scoped variables and context whenever an XPage is refreshed in a browser *while* the application design is open within Domino Designer and design changes are being made. Note that "refresh" in this context means resetting to empty values. This specifically ensures that view-, session-, and application-scoped variables, objects, and even Managed Beans are forcefully refreshed as a consequence of any application design changes. Request-scoped variables, objects, and Managed Beans, on the other hand, automatically are refreshed anyway because of the rules governing the request scope.

When this `xsp.application.forcefullrefresh` property is set to `true`, an application design refresh from a template causes this same behavior. Therefore, after the design refresh task has executed, subsequent requests to a given application with this property set to `true` cause a complete refresh of the scoped variables, objects, and context for the first incoming request after the design refresh. Ideally, for a production environment, this setting should be set to `false`; it is intended as a development-time aid.

The Theme Properties

This category of **xsp.properties** provides a way to set the application theme in three different ways. This accommodates the possibility of an application running in different platforms—namely, a Notes client or Domino server—and requiring different themes for each. You can also use this group of settings to specify a single theme for use in all platforms, which is the most common case. This is done by not specifying any **Override on Web** or **Override on Notes** settings, therefore allowing the theme specified for the **Application Theme** setting to be the one used regardless of running environment.

xsp.theme

If the `xsp.theme.web` or `xsp.theme.notes` properties are not specified, the `xsp.theme` property sets the default theme for all platforms. Therefore, you can configure the default theme for a Notes client or Domino server using this setting, as shown in Listing 1.5.

Listing 1.5 xsp.properties Snippet for the xsp.theme Property

```
# Name of the XSP theme to use
#xsp.theme=webstandard
```

This property is applied at platform level and affects all new or existing applications that do not specify their own theme. However, an application can override this setting using its own `xsp.properties` setting, therefore providing an Application Level setting. Figure 1.5 shows where this can be set within Designer for an application.

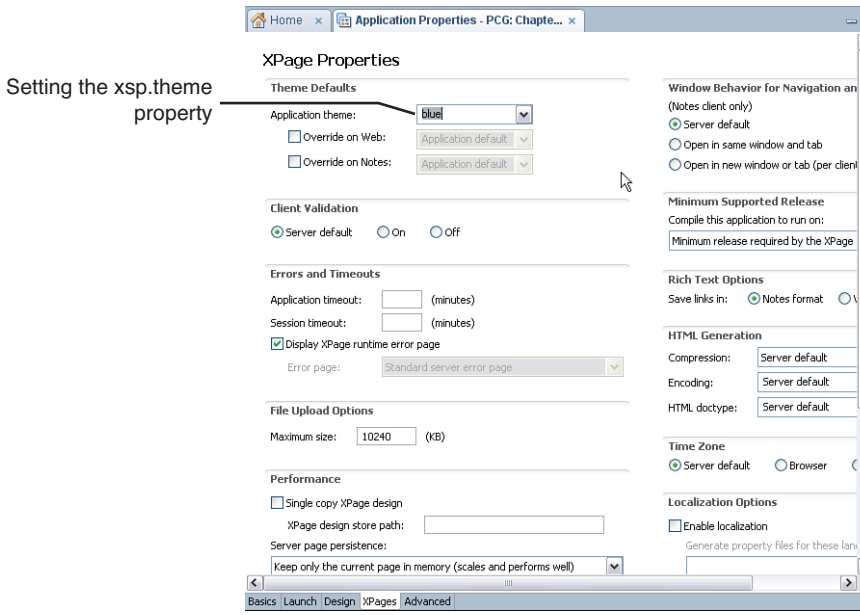


Figure 1.5 The xsp.theme property exposed in the application properties editor in Designer

TIP Refer to the *Mastering XPages* book (ISBN: 0132486318) from IBM® Press: www.ibmpressbooks.com/bookstore/product.asp?isbn=0132486318 where you will find extensive details on creating and applying XPages Themes in Chapter 14, "XPages Theming" (pages 543–620).

xsp.theme.web

Set this property to define the theme to display for web applications. Simply assign the name of any existing theme to the property declaration shown in Listing 1.6.

Listing 1.6 xsp.properties Snippet for the xsp.theme.web Property

```
# Name of the XSP theme to use when running on the web, if this
# property is not defined, the xsp.theme is used
#xsp.theme.web=
```

Regardless of whether the **xsp.theme** property is specified, the **xsp.theme.web** property takes precedence when specified for applications accessed using a browser. Figure 1.6 shows where this can be set within Designer for an application.

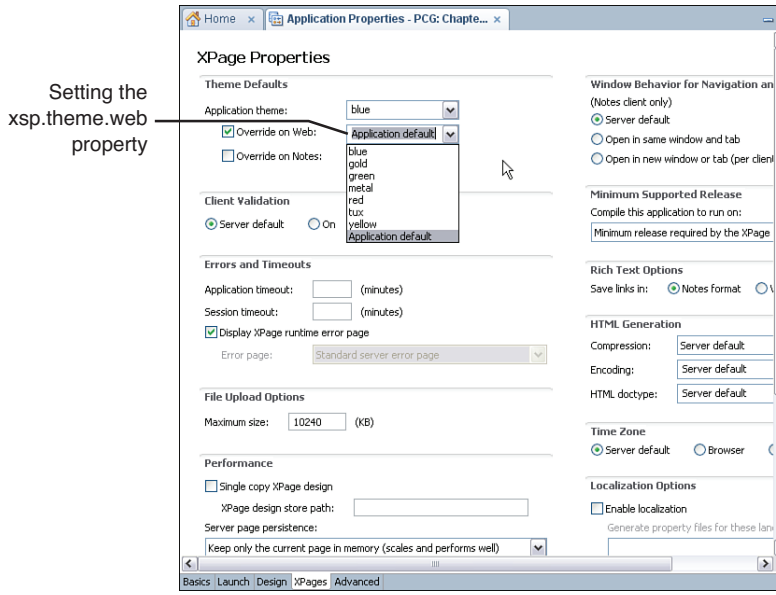


Figure 1.6 The xsp.theme.web property exposed in the application properties editor in Designer

xsp.theme.notes

Set this property to define the theme to display for XPages applications running in the Notes client. Listing 1.7 shows the relevant part of the **xsp.properties** file.

Listing 1.7 xsp.properties Snippet for the xsp.theme.notes Property

```
# Name of the XSP theme to use when running on the Notes client, if
# this property is not defined, the xsp.theme is used
#xsp.theme.notes
```

If the **xsp.theme** property is specified, this property takes precedence for applications accessed using a Notes client.

Set this property to define the theme to display for XPages applications running in the Notes client. Regardless of whether the **xsp.theme** property is specified, the `xsp.theme.notes` property takes precedence when specified for applications accessed using the Notes client. Figure 1.7 shows where this can be set within Designer for an application.

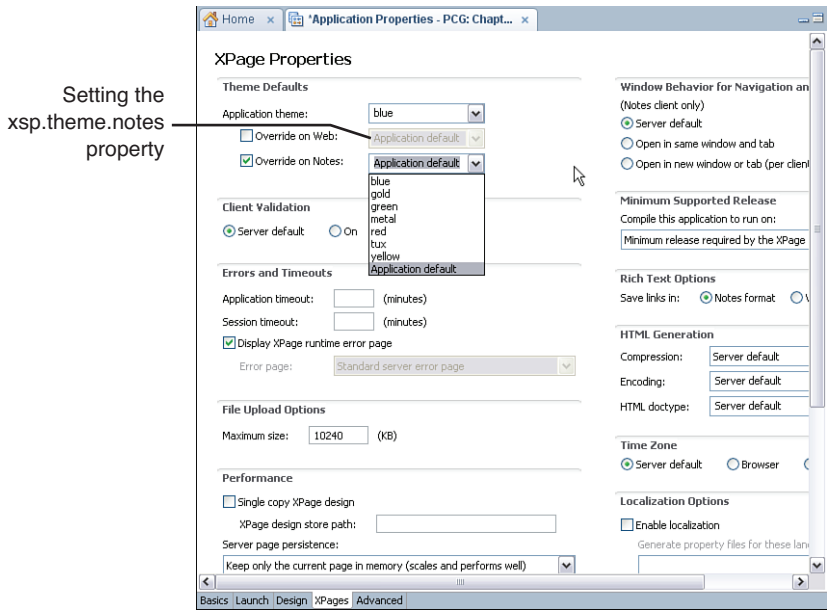


Figure 1.7 The xsp.theme.notes property exposed in the application properties editor in Designer

When you specify any of these theme properties from the Application Properties editor for any given application, they are written into the **xsp.properties** file for that application. Figure 1.8 shows that all three theme properties have been specified using the Application Properties editor.

Figure 1.9 shows these theme property values. They have been written into the application's underlying **xsp.properties** file.

Incidentally, you can use the **propertiesInspector** XPage found within the supporting **PCGCH01.nsf** application to inspect the values of any application-level property. When you open this XPage in a browser or on the Notes client, you see a type-ahead edit box that enables you to select from the available **xsp.properties**, as shown in Figure 1.10.

After you select any of the **xsp.properties** and then click the **Inspect** button, you see the currently specified value of that property based on the application-level setting. Figure 1.11 shows an example for the `xsp.theme` property that has been specified within the Application Properties editor for the theme **blue**.

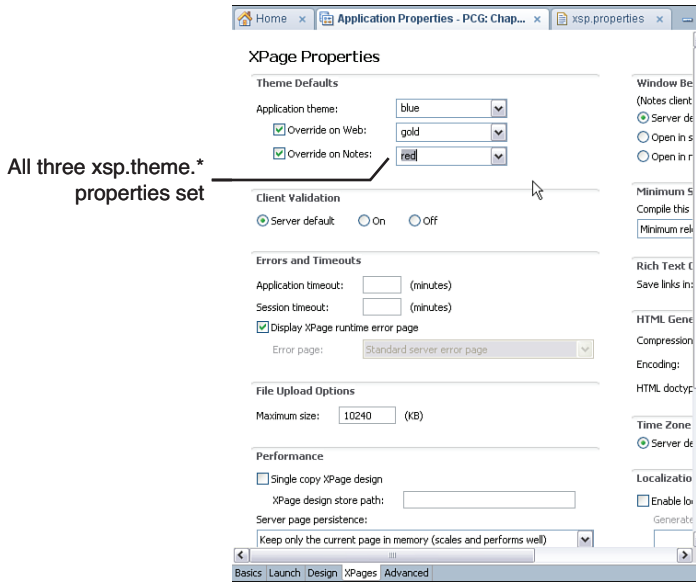


Figure 1.8 All three xsp.theme.* properties specified in the application properties editor in Designer

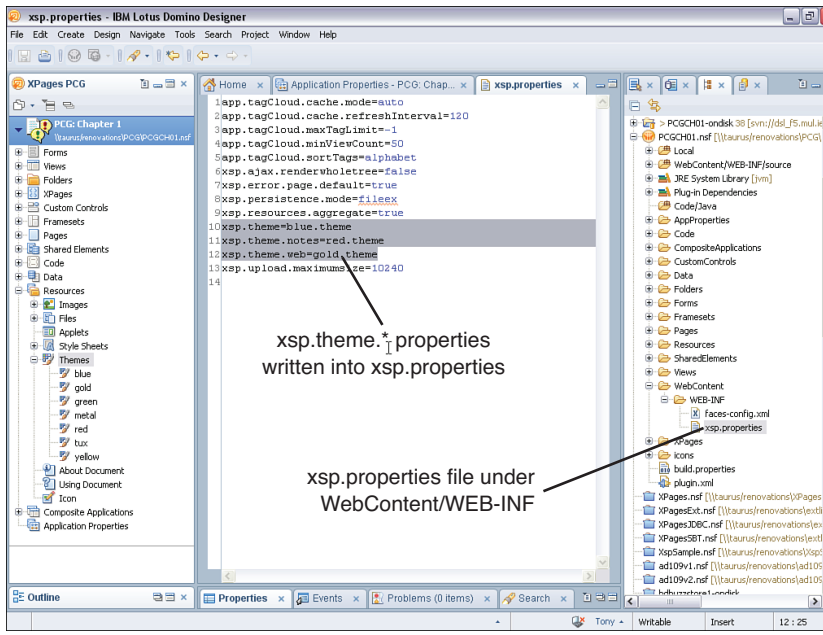


Figure 1.9 All three xsp.theme.* properties written into the application's underlying xsp.properties file

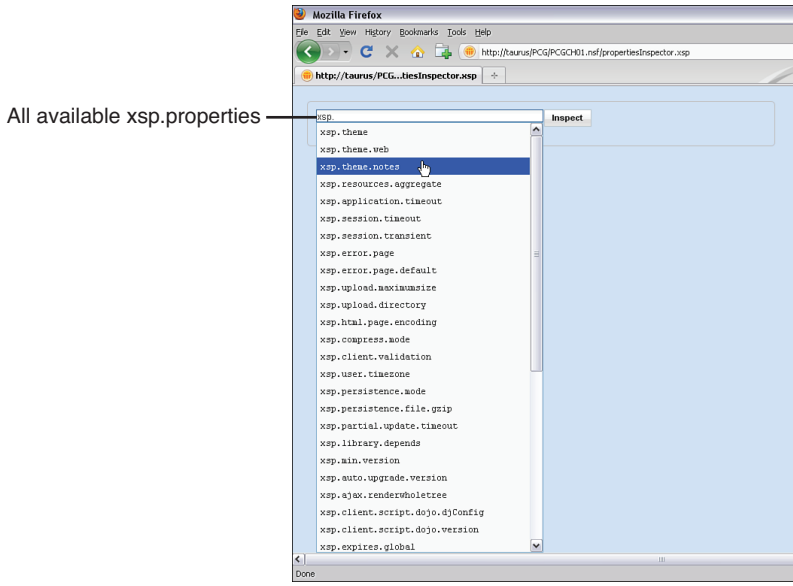


Figure 1.10 The type-ahead list of available xsp.properties displayed in the propertiesInspector XPage

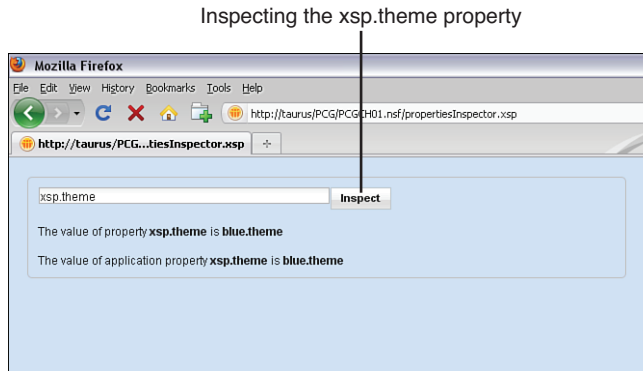


Figure 1.11 The propertiesInspector XPage showing the currently set value of the xsp.theme property

The Resources Properties

This category is new in Notes/Domino 8.5.3 and contains only one published entry.

xsp.resources.aggregate

The purpose of the `xsp.resources.aggregate` property is to enhance performance by reducing the number of requests made for resources like JavaScript files and

cascading style sheets, when an XPage is loaded by an end-user at runtime. The relevant part of the **xsp.properties** file is shown in Listing 1.8.

Listing 1.8 xsp.properties Snippet for the xsp.resources.aggregate Property

```
# Defines if the resources served to a page should be aggregated.
# This option should be used to provide the best download time
# experience. The option defaults to false when not set, but new
# applications created in Designer 8.5.3 or later will contain
# an xsp.properties file with the option value set to true.
#xsp.resources.aggregate=false
```

The more sophisticated the XPage is, the more likely it is to require many Dojo modules and CSS resources. Each resource request necessitates a network round-trip before the XPage rendering can be completed. Each round-trip has a performance impact, especially on slow or busy networks.

The number of resource requests required for a typical XPage can ramp up quickly—and in a manner that is often not immediately transparent to the application developer. This mostly occurs as a result of large and opaque resource-dependency trees generated by rich Dojo controls and complex UI themes. For instance, many of the sample pages in the standard XPages Extension Library demo application routinely generated more than 80 resource requests, although often the XSP markup of the XPages themselves do not explicitly declare resources. Clever analysis of the resource dependencies by the XPages runtime indicates that Dojo modules and CSS files can be aggregated (joined in the right order in a single file) so that fewer large-payload requests replace many small-payload requests. The former is far more efficient in terms of performance.

If you are inspecting the resource requests made on a typical XPage, such as when using Firebug on a Firefox browser, you will see resource requests of the form shown in Listing 1.9, when aggregation is *not* in effect.

Listing 1.9 Nonaggregated Resource Requests

```
...
GET http://server/domjs/dojo-x.y.z/dojo/../dijit/_base.js
GET http://server/domjs/dojo-x.y.z/dojo/../dijit/_base/focus.js
GET http://server/domjs/dojo-x.y.z/dojo/../dijit/_base/manager.js
GET http://server/domjs/dojo-x.y.z/dojo/../dijit/_base/popus.js
...
```

The simple resource requests shown here are replaced by something less humanly digestible when aggregation is applied. For instance, the `GET` request may point to an aggregated file named something like `@Wc&@Ei&@ESb.js`, an example of an aggregated Dojo module resource. When aggregation is in effect, you will not see these aggregated names in the `GET` request stack and you will also see fewer requests for the XPage.

The standard 8.5.3 help documentation summarizes the performance benefits of this feature quite well. It short, it provides the following:

- A decrease in requests sent from the browser to the server
- An increase in user performance, particularly in the context of networks with high latency
- An increase in the speed of JS/CSS parsing from the browser
- The freeing of server connections to fulfill other requests

The `xsp.resources.aggregate` property is exposed directly in Domino Designer by the **Application Properties > XPages > Use runtime optimized JavaScript and CSS resources** check box, so it does not require direct editing of the **xsp.properties** file to apply the setting for a given application. As indicated in the property comments, any new applications created with Domino Designer 8.5.3 automatically insert this property in the local **xsp.properties** file with a value of `true`. Earlier releases of the XPages core runtime ignore this property, so it is safe to apply in an environment that contains a mix of Notes/Domino 8.5.*x* releases, in which applications replicate with each other.

Some unpublished properties in this category can also give you more granular control over the operation of the feature. Table 1.2 summarize these properties. Their behavior is self-explanatory, although unpublished properties are not guaranteed to be supported in future releases.

Table 1.2 Other Aggregation Properties

Name	Description
xsp.resources.aggregate.dojo	Boolean value indicating whether XPages Dojo modules should be aggregated. Defaults to `true`.
xsp.resources.aggregate.css	Boolean value indicating whether XPages CSS resource should be aggregated. Defaults to `true`.
xsp.resources.aggregate.appjs	Boolean value indicating whether JavaScript resources located in the NSF itself should be aggregated. Defaults to `true`.
xsp.resources.aggregate.appcss	Boolean value indicating whether CSS resources located in the NSF itself should be aggregated. Defaults to `true`.

It is also worth mentioning that this feature works equally well on the Notes client and Domino server, although network round-trips for resources may not be such a worry if you are running in the Notes clients.

Finally, aggregated resources are also packaged in compressed gzip form, as long the browser accepts gzip content (refer to the `xsp.compress.mode` section for more information on compression). The browser caches aggregated resources just like any other

such resource, and the cache header field in the response is set to 10 days, by default. You can change this value by using the `xsp.expires.mini` property, to assign it an alternative number of days (`0` means no cache).

In a production environment, it is beneficial to enable resource aggregation. Otherwise, during development, you can turn off this setting if you are interested in viewing the individual resource links within the generated HTML markup.

The File Upload Properties

This category of properties enables you to override the default settings for handling a file upload attachment using the XPages core File Upload control. In particular, two properties control the maximum attachment size and the target upload directory on the receiving server.

xsp.upload.maximumsize

The `xsp.upload.maximumsize` property gives you the capability to set a specified maximum attachment size for file uploads processed by an XPage—in particular, file uploads handled by the XPages core File Upload control. The default setting is declared in the placeholder location in xsp.properties, as shown in Listing 1.10.

Listing 1.10 xsp.properties Snippet for the xsp.upload.maximumsize Property

```
# This controls the maximum size, in kilobytes, of a file being
# uploaded as an attachment
#xsp.upload.maximumsize=1024
```

The Domino server web engine enables you to specify a maximum attachment size for file uploads using the Domino Server Configuration document from within Domino Administrator. Ultimately, that setting takes precedence over any other setting. Figure 1.12 shows where to find this property within an application's Application Properties editor.

You can set this property at the server level within the **xsp.properties** file of a Domino server or Notes client. You should set it within the upper limits of the overall Domino Server Configuration document maximum file attachment size setting. Of course, you can increase the Domino Server Configuration document setting accordingly.

xsp.upload.directory

By default, the XPages runtime receives a File Upload attachment or Rich Text Editor embedded image file and temporarily persists it to disk on the receiving server before further processing the attachment or image. (See the `xsp.persistence.dir.xspupload` property for the default behavior.) Listing 1.11 shows this property as it appears in the **xsp.properties** file.

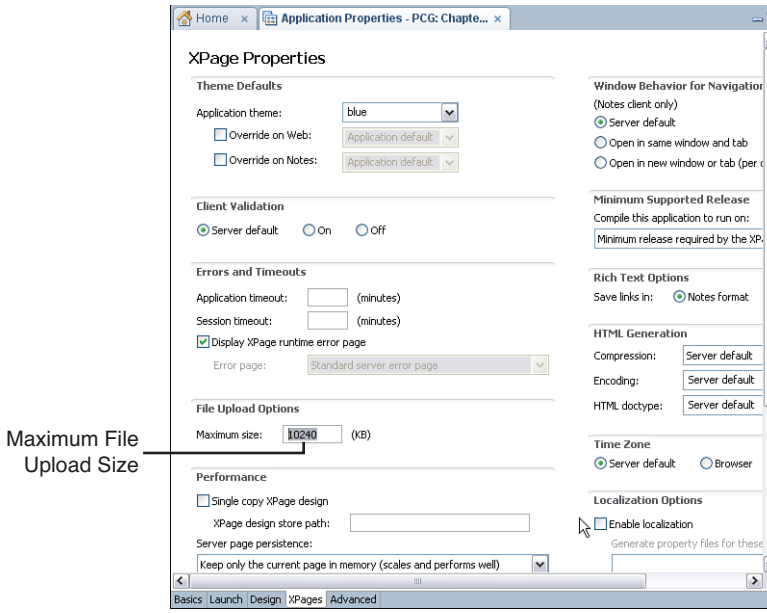


Figure 1.12 The xsp.upload.maximumsize property exposed in the Application Properties in Designer

Listing 1.11 xsp.properties Snippet for the xsp.upload.directory Property

```
# Directory used to temporarily store the uploaded attachment
# Default to a temporary directory returned by the OS
#xsp.upload.directory=
```

This property is a global setting; therefore, it must be specified only within a Domino server or Notes client **xsp.properties** file. Setting the `xsp.upload.directory` property within an application's **xsp.properties** file has no effect because the global setting always takes precedence.

This setting has been available since version 8.5, but it is now deprecated and provided only for legacy compatibility purposes. You should use the `xsp.persistence.dir.xspupload` property instead. If specified, the `xsp.upload.directory` property value is used instead of the `xsp.persistence.dir.xspupload` property value. The `xsp.upload.directory` property enables you to specify a different temporary upload directory. This can be useful if disk space is an issue and you need to point at another disk space directory on another server- or network-assigned storage device.

The JSF Persistence Properties

This category mostly relates to how a user's page state is saved on the server between requests interacting with that page. Options relate to how the page state is saved on

the server file system. This category also includes some options for how uploaded and attached files are saved on the server file system. But to start, an option controls how JavaScript variables are saved between requests, which impacts on the size of the page state.

xsp.persistence.discardjs

This property controls how data stored as global JavaScript variables are handled when a page request is complete. Listing 1.12 shows the relevant section of **xsp.properties**.

Listing 1.12 xsp.properties Snippet for the xsp.persistence.discardjs Property

```
# Discard the JavaScript context for a page after the page is processed
# This is a runtime optimization that is set to true by default
# but might be reverted to avoid compatibility issues
# (although it is *not* advised).
#xsp.persistence.discardjs=true
```

Version 8.5.1 included a change in how global Server Side JavaScript variables are handled between page requests. You can set this option to `false` to revert to the older behavior from version 8.5.0, although it is better to change your application to work with the new behavior. The new behavior gives better performance by helping your application to use less memory.

The behavior that the option controls is related to global variables defined in Server Side JavaScript code, such as the `valueComputed` variable shown in Listing 1.13.

Listing 1.13 Global Server Side JavaScript Variable

```
valueComputed = false;
// update valueComputed
```

When such a global variable is defined, it can be referenced in any Server Side JavaScript in the XPage that executes after the global variable has been defined, until the end of the current server request.

In version 8.5.0, global variables were available to be referenced for a longer duration, from when the global variable was defined until a different XPage instance was loaded. That is, when multiple redisplays of the same XPage used repeated requests to the server, global variables continued to be available during subsequent server requests. The new behavior, post–version 8.5.0, ensures that global variables are discarded between page requests.

When attempting to redesign your application to handle the shorter global variable availability, you can use `viewScope` variables for values that you need to access in subsequent redisplays of the same XPage. Listing 1.14 shows the revised code.

Listing 1.14 Scoped Server Side JavaScript Variable

```
viewScope.valueComputed = false;
// update viewScope.valueComputed
```

Similarly, you can change any other code in the application that used to reference `valueComputed` (or your variable name) to reference `viewScope.valueComputed`.

The `viewScope` is a namespace where you can store simple values while the same page is being repeatedly redisplayed. It is not possible to store a document in `viewScope`, but it is often sufficient to store the document ID and retrieve the document using the ID. It is okay to save Strings, Booleans, Numbers, and many other values in the `viewScope`, but if nonserializable objects are placed in the `viewScope`, problems can arise—see the information on the `xsp.persistence.mode` option for more details. Other scopes can be useful as well, such as the `sessionScope`, which stores values for the duration of the current user's login session. Chapter 5, "Server-Side Scripting," explains the scopes in the section "Scope Objects."

xsp.persistence.mode

This option determines how the server-side tree of controls is saved between requests while the user is interacting with the XPage. Listing 1.15 shows the help information provided in the **xsp.properties** file.

Listing 1.15 xsp.properties Snippet for the xsp.persistence.mode Property

```
# Defines the persistence mode for the JSF pages (a.k.a. Views)
#   file:     All the pages are persisted on disk
#   fileex:  All the pages are persisted on disk except
#             the current one, which stays in memory
#   basic:  All the pages stay in memory, the default.
#xsp.persistence.mode=
```

When a user first opens an XPage in a web browser, a server-side tree representing the controls in the XPage is built up and used to output the XPage to the browser. If the user then interacts with the same page, such as by clicking a `Section` control to make its contents visible, the same control tree is used to respond and interact with the user. As the user interacts with the page, the control tree maintains state so that, for example, values typed into fields are still present between requests and a document can be edited across multiple requests without saving on each request.

To allow such interaction, the tree of controls for an XPage must be saved on the server between requests from the user. A specific control tree instance that one user requesting an XPage has created is known as a page or a JSF view. The options controlling this saving are known as the persistence options, meaning that they control how the pages manage to persist (or, continue to be present) when attempting to redisplay them.

Broadly speaking, the page can be either saved on the server's file system or kept in memory (RAM) so that no explicit save operation takes place.

When the pages are saved in memory, the response time for individual users is quick. Most servers can handle low levels of infrequent users, but as the number of users increases, eventually the server might have not enough memory. When the server runs out of memory, it reports the serious `java.lang.OutOfMemoryError` problem, which prevents XPages from being displayed to any users. This problem likely has negative consequences for non-XPages use of the server.

When pages are saved on the file system, they do not cause the server to run out of memory. The application will be less likely to fail as it scales up to larger numbers of users and a greater request load. However, when the pages are saved to the server's file system, it may be necessary to configure how that saving occurs. (Refer to the section entitled "No Space Left on Device Problems," for suggested solutions when the file system runs out of space.) Saving pages to the file system also introduces the possibility of serialization problems that prevent XPages from displaying. (This chapter discusses how to fix such problems in an upcoming section titled "XPages Problems When Storing Pages on the File System.")

It is also possible to choose a model that acts as a hybrid between in-memory and file system schemes. The most commonly used option, `fileex`, saves the most recent page the user has touched in memory and saves the previous pages in the usage history on the file system. This is sensible because users are most likely to continue interacting with the most recent page. Users generally interact with previous pages only if they press the browser back button or if they are using multitab browsing, in which the web browser has multiple pages open in the same application.

The `xsp.persistence.file.threshold` option enables another compromise by saving pages that are smaller than a certain size in memory and writing only larger pages to disc. However, the process of saving in memory is different when using the threshold option. A later section, "xsp.persistence.file.threshold Property," is dedicated to this option.

Choosing the Persistence Mode in Designer

You can change the main setting that controls persistence in Application Properties, in Designer. The **Application Properties > XPages > Performance > Server page persistence** combo box offers the following options:

- **Server default.**
- **Keep pages in memory (best performance).** This saves the mode as `basic`, meaning that the pages are always saved in memory.
- **Keep pages on disk (best scalability).** This saves the mode as `file`, meaning that the pages are always saved on disk unless the threshold option is set to some value.

- **Keep only the current page in memory (scales and performs well).** This saves the mode as `fileex`, meaning that the most recently accessed page for each user is saved in memory and older pages for the users are saved on disk.

In applications created in a Designer version 8.5.2 or later, this setting defaults to `fileex`. That is, the actual **xsp.properties** file in the application has the literal mode *`fileex`* specified. This means that editing the server **xsp.properties** file does not have an effect on the persistence of such applications. For applications created in version 8.5.0 or 8.5.1, this setting defaults to `Server default`, but it is advisable to edit such applications to change the setting to `fileex`. However, when making such a change, it is best to verify that the application continues to work. Refer to the section "Serialization Problems Giving an Error Page with NotSerializableException" if issues arise.

Cache Size Limits and XPages Behavior When Limits Are Encountered

Whether saving on disk or in memory, limits govern the number of pages in a user's browser session that are maintained on the server. You can configure the limits for the maximum number of views for an application. When a user in a given session navigates through more pages than the maximum limit, older pages are discarded. Selecting which page to discard is accomplished using an MRU (most recently used) algorithm, to preserve the most recently accessed pages. The pages discarded are the pages whose last *redisplay time* was furthest in the past, not the pages that were *created* furthest in the past. The discarding happens when the user navigates to a new page, so continued redisplay of the same page does not cause previously cached pages to be discarded.

When a user attempts to access a page that was discarded from the cache, the page is displayed in its initial state, as if this were the first time the user opened the page. Users can access old pages either by redisplaying them using the browser back button or by clicking a button or link on an open page in a browser tab that is not the most recent tab the user was interacting with. When a user accesses a discarded page, the previous state of the page is not available; if the user had been editing values in a document, those values would not be present in the page. If the user clicked a Save button and expected the page to be no longer editable, the user might not realize that the updated values were lost. Certain applications and populations of users are more likely to encounter discarded page problems; examples are users who commonly use tabbed browsing and edit multiple documents at once. If users of your application likely will attempt to access discarded pages, you might want to either increase the maximum number of pages saved per user or provide warnings about the use of tabbed browsing with this application.

XPages Problems When Storing Pages on the File System

An application that saves the pages in files on disk might encounter some problems that it would not if it saved the pages only in memory.

Serialization Problems Giving an Error Page with NotSerializableException

The most common type of problem with saving to files is a `java.io.NotSerializableException`. Serialization is the process of converting from

in-memory objects to a representation of those objects that can be saved to a file. When the configuration options indicate that pages are to be saved to the file system and a page is about to be saved, this serialization is performed and any problems in the serialization process give a `java.io.NotSerializableException`. The exception prevents the XPage from displaying and an error page appears with a stack trace, as shown in Figure 1.13. One of the messages at the top of the error page is **java.io.NotSerializableException: *classname***, where *classname* is the name of a class that could not be serialized.

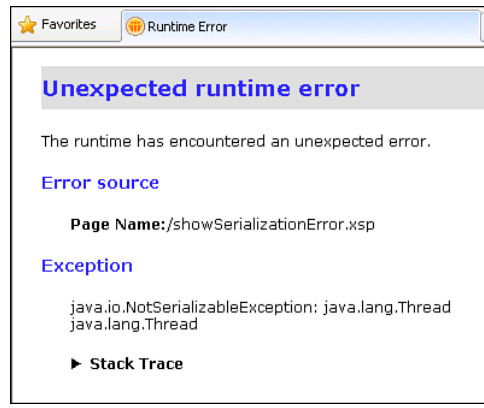


Figure 1.13 Example of an error page showing serialization problems

When a `NotSerializableException` occurs, it indicates a problem in the design of the application where a nonserializable object is being used but cannot be persisted as part of the page state. Commonly, the application designer buffered a nonserializable object in the `viewScope` map. Alternatively, the application designer might have used a `Compute On Page Load` expression to generate the value of a control property, but the computed value is not serializable. Many types of objects are serializable and okay to save in the `viewScope` and other serialized scopes. For example, Strings, Numbers, and Java Dates are serializable, as are Arrays, Lists, and Maps containing such values. When saving other objects to the `viewScope`, it is recommended to verify that they are serializable. Java developers can check whether the object's class implements the `java.io.Serializable` interface and that all the contents of the object are also serializable. It may be more convenient to verify that the `NotSerializableError` does not occur, by changing the application to save the pages on disk, opening the page that is saving the value to the `viewScope`, and clicking a submit button to redisplay the page a few times.

To fix your serialization problem, you should redesign your application not to save the nonserializable object. Usually, it is possible to save some String value that can reconstruct any nonserializable object during subsequent page requests. If it is not immediately apparent where the problem object is arising, you can take some common steps to find the cause of the problem. In the `afterRenderResponse` event on your page, you

should print the names and values of the `viewScope` contents to the server console and do a `(value instanceof java.io.Serializable)` check on each to determine which `viewScope` object is causing the problem. When you have the name used to save the object in the `viewScope`, you can search your application for references to that name and put further debugging code where values are saved in the `viewScope`.

Functional Problems Caused by Control State Saving Issues

Besides serialization problems, the other kind of problem you may see when saving pages on the file system occurs when using controls and data sources built using the XPages Extensibility APIs rather than the controls the XPages runtime provides. If problems arise in the control implementations, you might see functional issues in the controls after the pages are redisplayed. The values set on the control in the XPage might be missing after the page is restored from disk. For instance, the control style might be gone, or if it was configured to do a partial update, it might have reverted to doing a full update. Such issues can be subtle and hard to notice, especially because the page appears correctly on the initial page display. Those problems occur when controls do not correctly implement the `javax.faces.component.StateHolder` interface. The solution is usually to ask the control developer to fix the problem in the control. Work around the issue might be possible by computing the values instead of directly setting them in the XPage. For instance, instead of setting the style as:

```
style="background-color: blue"
```

You might compute it as this:

```
style="#{javascript: 'background-color: blue'}"
```

No Space Left on Device Problems When the Disk Is Full

When saving page states to the file system has been enabled by setting `xsp.persistence.mode` to `file` or `fileex`, the possibility exists that the server file system might run out of space. This is determined by the amount of space used in proportion to how many users are accessing XPages in various applications, and up to a limit, to how many XPages they view during their session. A more likely scenario than the file system running out of space is that the server will run out of memory, so the solution here is not to change the mode to `basic` (saving the state in memory). Instead, you can use options to reduce the number of page files saved, to decrease their size, and possibly to avoid saving pages.

- The solution with fewest negative effects is to change the location where the page files are saved so that it points to a different, larger drive that is less likely to run out of space. The location where the files are saved is configurable through the option `xsp.persistence.dir.xspstate`; see the section dedicated to that option for more details.
- The option `xsp.persistence.file.gzip` ensures that files are compressed using GZIP format before they are saved to the file system. This results in smaller files, but at a cost of both more processing load on the server when compressing

the files and longer response time to the users on the client side when decompressing the files to restore the pages.

- The option `xsp.persistence.tree.maxviews` can be used to reduce the number of files saved per user. The capability to use that option depends on the usage patterns of the application users. Reducing this limit might make users lose data entered into the browser when the cache limit is reached.
- The `xsp.session.timeout` option can be reduced to decrease the time page files remain on disk after the user stops interacting with the application. The usefulness of this option depends on the user application usage patterns. Possible negative effects can arise, including loss of data typed into a browser. It can be useful if many intermittent users are each accessing a few pages and then closing the browser or navigating away from the site.
- The `xsp.persistence.file.threshold` option can reduce the number of page files, at the expense of using more memory. It can be useful if the application pages are often small.
- The `xsp.persistence.viewstate` and `xsp.session.transient` options ensure that no pages are saved, but they are useful for only certain types of applications. Generally, they are useful when the application does not allow editing or creating documents and mostly uses HTTP `GET` requests instead of HTTP `POST` requests.

xsp.persistence.tree.maxviews

This option applies when using the option `xsp.persistence.mode` with the value `basic`, indicating that a user's pages should be stored in memory. The default value is `4` (as shown in Listing 1.16), which means that the server stores four pages per user in memory.

Listing 1.16 xsp.properties Snippet for the xsp.persistence.tree.maxviews Property

```
# Defines the number of pages persisted when in memory (MRU algorithm)
#xsp.persistence.tree.maxviews=4
```

Refer to the section "Cache Size Limits and XPages Behavior When Limits Are Encountered" for details on what happens when users access a fifth page and the possible negative effects if a user attempts to access a page that is no longer in the cache. Those negative effects include the possibility of data loss, where values that the user has typed into the browser are no longer available and the user then must reenter values. If users of this application encounter such problems, it may be necessary to increase this in-memory cache size limit. If increasing this limit causes the server to run out of memory, it may be necessary to change the `xsp.persistence.mode` option to `file` or `fileex`; fewer problems arise with large cache sizes when dealing with pages persisted on the file system.

By changing your application design, you might reduce this limit without negative effects to your users. Generally, your application must be designed to mostly use HTTP

GET requests instead of POST requests, to have few or no forms where users need to edit data, and few or no server-side actions in response to events. Reducing the value means that less memory is used per user. Thus, a greater number of users can access the server without incurring the response time issues associated with setting the modes file and fileex, which are usually used when attempting to support a greater number of users.

xsp.persistence.file.maxviews

This option applies when using the value file or fileex with the option xsp.persistence.mode, meaning that some or all of a user's saved page states are written to the server's file system. This option controls the number of previous pages the user viewed that will be saved as files on the file system before further opened pages cause older pages to be discarded. The default value is 16 pages, as shown in Listing 1.17.

Listing 1.17 xsp.properties Snippet for the xsp.persistence.file.maxviews Property

```
# Defines the number of pages persisted on disk,
# when "file" or "fileex" is defined (MRU algorithm)
#xsp.persistence.file.maxviews=16
```

Again, refer to the section "Cache Size Limits and XPages Behavior When Limits Are Encountered" for details on the behavior when users access the next page after the limit is reached. In theory, it might be necessary to increase this limit if users are using multiple page tabs in the web browser or if they use the browser's back history.

If problems with insufficient disk space on the server arise, a possible solution is to reduce this limit so that fewer pages are saved per user. The value should not be so low that it causes the cache limit negative effects, however. The previous section, "No Space Left on Device Problems," covers part of the xsp.persistence.mode topic.

xsp.persistence.viewstate

This setting controls how much of the page control tree state is saved between requests. It has four possible values: fulltree, nostate, delta, and deltaex, as shown in Listing 1.18.

Listing 1.18 xsp.properties Snippet for the xsp.persistence.viewstate Property

```
# Defines the persistence mode for the JSF pages (a.k.a. Views)
#   fulltree:      Persists the full page content. Default mode.
#   nostate:      The page is *not* persisted at all. Useful
#                  for pure read-only pages.
# The following options are valid only when the page
# is persisted in memory
#   delta:           Only persists the changes since the
#                   page was constructed
```

```
#   deltaex:           Persists the full state for the current page,
#                 and delta's for all other pages #xsp.persistence.
viewstate=
```

The default value for this property is `fulltree`, meaning that the full control tree state is saved between requests. This behavior can also be configured through the XPage root control's `viewState` property, which you can set to the same possible values. The value set in an XPage overrides the application setting in the **xsp.properties** file. The behavior when this option is set to `nostate` is similar to when the option `xsp.session.transient` is enabled.

The value `nostate` means that none of the control tree is saved between requests. Instead, when attempting to restore a control tree state for a previously displayed page, a new control tree is created and used to process the incoming request. Some of the behavior here is similar to the behavior when the cache size limits are reached. For example, if the user has navigated to a different tab in a Tabbed Panel or has toggled a Section area open or closed, those controls appear at the initial state, either displaying the wrong tab or appearing closed instead of open.

However, submitted values from the browser are not ignored. Values in edit boxes can be saved to a document and simple actions can occur in response to events (although only if they are in an area that is initially visible in the first page display). When designing a page intended to be used with the `nostate` value, values that would normally be saved in a `viewScope` variable can be saved in an Input Hidden control included in page submissions and thus can maintain their value in the browser across multiple requests. In general, though, this option is usually used when building applications that are intended only to display data, not to edit existing documents or to create new documents. That way, the application developer doesn't have to worry about maintaining data across pages and possibly losing user-entered values.

As a refinement of the `nostate` value, the value `delta` attempts to address some of the limitations of the `nostate` value by saving certain values between requests. However, it still does not save the control tree and it still re-creates a new control tree for every request. Not all XPages controls support the delta option, and not all control values are saved. When using the `delta` strategy for saving the control tree state, certain parts of the control tree are saved. Examples include the `viewScope` map; the Tabbed Panel control's current tab state; the Section control's open or closed state; and the View Panel control's `first` property value, which corresponds to which page of data is being displayed. An example of a value that is not saved is the View Panel control's `rows` value, which indicates how many documents will be displayed in each page of data. It is common to make the `rows` configurable so that an end user can choose to display more rows in each page. With the `nostate` and `delta` options, the View Panel control reverted to displaying the initial number of rows on the next redisplay of the XPage, so the user's change to the `rows` value was lost.

The `deltaex` value gives a similar behavior to the `delta` option, in that most of the previous pages in a user's session history are saved using the `delta` option. Thus, only

some of the state in the control tree is saved. The difference with the `deltaex` value is that the current page the user is viewing is entirely saved. The entire control tree state for the current page, not just the minimal state used with the `delta` value, is serialized and saved in memory. The behavior where it serializes the state before saving it in memory is different than the normal in-memory saving of the control tree explained for the `xsp.persistence.mode` value `basic`. That serialization strategy is also used with the `xsp.persistence.file.threshold` option. It has the disadvantage that serialization errors may occur, such as those explained in the previous section on serialization problems. However, this option has an advantage over `delta` view state saving: The user is less likely to encounter problems where page state seems to have reverted. The page state is the same for most use cases, and only when users use the Back button in the browser will they encounter the behavior explained for the `delta` value.

xsp.persistence.file.gzip

This option applies when the `xsp.persistence.mode` option is set to `file` or `fileex`, meaning that the page state is stored on the server as files on the file system. Listing 1.19 shows the relevant part of the **xsp.properties** file.

Listing 1.19 xsp.properties Snippet for the xsp.persistence.file.gzip Property

```
# Defines if the persisted files should be GZIP'ed on disk
# (less disk space, more CPU processing)
#xsp.persistence.file.gzip=false
```

You can set this property to `true` to cause the files to be zipped or compressed before they are saved. (GZIP is a particular compression utility.) This means that the file size is smaller, so there is less possibility of running out of disk space on the server. However, this comes at the cost of running the compression algorithm, which uses up server computation time and may cause decreased response time for users, depending on how busy the server is. The section entitled “No Space Left on Device Problems” discusses other solutions to low disk space problems, as part of the xsp.persistence.mode property discussion.

xsp.persistence.file.async

This option applies when the `xsp.persistence.mode` option is set to `file` or `fileex`, meaning that the page state is stored on the server as files on the file system. It defaults to `true`, as shown in Listing 1.20.

Listing 1.20 xsp.properties Snippet for the xsp.persistence.file.async Property

```
# Defines if the pages persistence to a file
# should be done asynchronously (best response time,
# creates extra threads on the server)
#xsp.persistence.file.async=true
```

The default value (`true`) means that, after the page has been used to generate the HTML response, it is not immediately saved to the file system during that user's browser request. Instead, a helper thread later handles the operation of writing the file to the file system; the response to the browser is not delayed by interaction with the server file system. This leads to better response times and better user experience in the browser.

The most common use case for setting this option to `false` is for debugging issues encountered when writing the page to the file system.

Before the page content is passed to the helper thread, it is serialized to a buffer. This can result in serialization problems with an error page displayed to the user, as mentioned in the section on the `xsp.persistence.mode` option. Later, the helper thread attempts to save the buffer to the file system. Any problems during the file system save do not appear in the browser and are noted only in the server log files. Problems saving to the file system can occur if the server disk is full, if the server process does not have permission to write values to the cache location, or if any other types of `java.io.IOException` problems occur. If a problem saving the page arises, the behavior for the user is the same as if the page was discarded because the cache was full. This can lead to data loss issues. For more information, refer to the section "Cache Size Limits and XPages Behavior When Limits Are Encountered."

xsp.persistence.file.threshold

This option applies when the `xsp.persistence.mode` option is set to `file or fileex`, meaning that the page state is stored on the server as files on the file system. Listing 1.21 shows the relevant property and help text as contained in the **xsp.properties** file.

Listing 1.21 xsp.properties Snippet for the xsp.persistence.file.threshold Property

```
# Defines if the pages should be serialized in memory,
# instead of in files, when their size is less
# than a specific amount of bytes. 0, which is default,
# means that it is always serialized to disk
#xsp.persistence.file.threshold=0
```

When set to `0`, the user's pages on the server are serialized (converted from control tree objects to a representation of objects that can be saved to a file), and the serialized pages are saved to a file on the server's file system. This option provides a size threshold that depends on the size of the serialized page, measured in bytes. If the serialized page size is less than this threshold, the serialized page is stored in memory instead of on the file system. Note that this is different than the behavior when the `xsp.persistence.mode` option is set to `basic`. In that case, the actual page control tree objects are saved in memory, whereas, in this `file` or `fileex` case, the serialized representation of that page is saved instead. Because these smaller pages are saved in memory instead of on disk, response times are improved for browser requests for these pages. The possibility of running out of server memory does arise, along with the consequent issues discussed at

length elsewhere in this chapter. If such problems are encountered, it may be necessary to reduce this threshold again. This threshold option was added in version 8.5.3.

xsp.persistence.dir.xspstate

This option controls the location where the state of pages previously opened by a user are saved when saving to the file system. Listing 1.22 shows a snippet from the relevant section in the xsp.properties file.

Listing 1.22 xsp.properties Snippet for the xsp.persistence.dir.xspstate Property

```
# Define the directory where the JSF pages are persisted
# defaults to <tempdir>/<notesSessionID>/xspstate
#xsp.persistence.dir.xspstate=
```

Pages are saved to the file system when the `xsp.persistence.mode` option is `file` or `fileex`. The generally recommended mode is `fileex`, so this location is likely to contain page files.

The page state files are saved to a location such as this: **C:\Documents and Settings*userName*\Local Settings\Temp\notes74483D\xspstate\2\CXXTXFVP7C\cxzkvftr56.ser**

The folder **C:\Documents and Settings*userName*\Local Settings\Temp** is the default Windows temporary folder. That folder location varies, depending on your operating system and how your OS is configured. In this example, the folder **notes74483D** is a Notes/Domino instance temporary folder; the number changes every time Notes or Domino is restarted. In version 8.5.0, this folder didn't exist and the XPages persistence used different subfolders under the temporary folder. The folder **xspstate** is a container folder for the page files; other container folders are present at that level. The **2** is a folder corresponding to the application. Those numbers are lazily assigned based on the order applications are accessed using XPages, starting at 1 for the first application opened after the server starts. The **CXXTXFVP7C** is a user session identifier. Different users, or the same user logged in using a different web browser, have different session IDs. The sessions also have a timeout, so a user reaccessing the application after the session has expired triggers a new session. The **cxzkvftr56.ser** is the `viewId` identifying this XPage control tree instance. The `viewId` is accessible through Server Side JavaScript via the `view.getViewId()` API call and is present in the HTML source produced by an XPage. Files saved here are discarded when the page cache for that user session is full, when the user session has timed out due to inactivity, and when the server is restarted. It may be useful to change this setting to point to a different location, if the folder is taking up too much space on the main server drive and an alternate drive has more available space. Other options also can be set to reduce the space used by the page saving. For more information, refer to the section "No Space Left on Device Problems," as part of the `xsp.persistence.mode` property discussion. It may also be useful to change this if faster drives are available, to give better turn-around time to individual web users.

This option is server-wide, so it should be set in the server **xsp.properties** file. Values set in a particular application's **xsp.properties** file are ignored.

xsp.persistence.dir.xspupload

As described in the help text in Listing 1.23, this option defines the location where uploaded files are temporarily stored on the server file system.

Listing 1.23 xsp.properties Snippet for the xsp.persistence.dir.xspupload Property

```
# Define the directory where the temporary uploaded files are stored
# defaults to <tempdir>/<notesSessionID>/xspupload
#xsp.persistence.dir.xspupload=
```

If the File Upload control is bound to a document field so that it will be saved as an attachment, the file will not be in this temporary upload folder for long and will be moved to the folder referenced by the option `xsp.persistence.dir.xsppers`.

The default location of this folder is like so: **C:\Documents and Settings\userName\Local Settings\Temp\notes74483D\xspupload**

Refer to the option `xsp.persistence.dir.xspstate` for a discussion of the folders up to `xspupload`. Within the `xspupload` folder, the file is saved with a temporary filename, unrelated to the name before upload or to the name used when it is attached to the document. Note that there is a limit to the size of files that can be uploaded, configurable through the option `xsp.upload.maximumsize`.

It may be useful to change this setting to point to a different location if the folder is taking up too much space on the main server drive and another drive has more available space.

This option is server-wide, so it should be set in the server **xsp.properties** file. Values set in a particular application's **xsp.properties** file are ignored.

The `xsp.persistence.dir.xspupload` property is related to the `xsp.upload.directory` property but should be used as the preferred option because the `xsp.upload.directory` is deprecated.

xsp.persistence.dir.xsppers

This option controls the location where document attachments are temporarily stored on the server file system after the files have been uploaded and associated with a document, but before the document has been saved. An intermediary step comes before the attachment is associated with a document: At that point, the file is saved in the location indicated by the `xsp.persistence.dir.upload` option. Listing 1.24 displays the relevant snippet from **xsp.properties**.

Listing 1.24 xsp.properties Snippet for the xsp.persistence.dir.xsppers Property

```
# Define the directory where the document attachments
# are temporarily persisted (stored)
# defaults to <tempdir>/xsppers
#xsp.persistence.dir.xsppers=
```

While the file is in this **xsppers** folder, the browser URL to download the file is like: **http://serverName/appName.nsf/xsp/.ibmmodres/persistence/DominoDoc-3-Body/red.GIF**

With the default settings, the actual file location on the server file system is like: **C:\Documents and Settings\userName\Local Settings\Temp\notes74483D\xsppers\2\CXXTXFVP7C\DominoDoc-3-Body\red.GIF**

See the option `xsp.persistence.dir.xspstate` for a discussion of the folders up to `DominoDoc-3-Body`, except that these attachment files are saved in an `xsppers` folder instead of the `xspstate` folder described in that option.

DominoDoc-3-Body/ indicates that this is the third document to which files are being attached in the application. That folder can contain multiple files as more attachments are added to the document.

red.GIF is usually the name of the file before it was uploaded, although there is an option on the file upload control to assign a different name to the uploaded file.

After the document has been saved, the file no longer is accessed through a persistence URL. Instead, it is accessed through a Domino document attachment URL: **http://serverName.example.com/appName.nsf/xsp/.ibmmodres/domino/OpenAttachment/appName.nsf/91AB1F5555CF7E06802578FA005F8DFC/Body/red.GIF**

Some different variants of that URL syntax exist—for example, the document might be from an application on a different server.

The files remain in the temporary persistence location until the user session expires. The file is not removed after the document is saved, although it is no longer referenced by URLs.

Note that there is a limit to the size of files that can be uploaded, configurable through the option `xsp.upload.maximumsize`.

It may be useful to change this setting to point to a different location if the folder is taking up too much space on the main server drive and another drive has more available space.

This option is server-wide, so it should be set in the server **xsp.properties** file. Values set in a particular application's **xsp.properties** file are ignored.

The Client Side JavaScript Properties

These options relate to the JavaScript framework used in the browser. "Client side" here means in the browser, as opposed to Server Side JavaScript, which executes in the XPages runtime as part of the web server. The Client Side JavaScript framework used by XPages is the Dojo Toolkit, so these options relate to how Dojo is used in XPages.

xsp.client.script.dojo.version

This option relates to the Dojo Toolkit, which comes installed on the Domino server and in the Notes client. Pay close attention to the help text shown in Listing 1.25.

Listing 1.25 xsp.properties Snippet for the xsp.client.script.dojo.version Property

```
# The version of the Dojo Toolkit to use.
# By default the Dojo version is detected by examining the folder
# Data/domino/js/ for subfolders with names like dojo-<version>,
# and using the latest version available.
# Change this setting if you are installing different versions of Dojo
# in that folder and you need XPages to use a specific version.
# Note, using XPages with a Dojo version other than the default
# is unsupported; if you do so you will need to test for
# compatibility problems.
#xsp.client.script.dojo.version=
```

Dojo is used in XPages as a client (browser) JavaScript framework and to provide the browser behavior of some controls. People familiar with Dojo can also use the dojo utilities in their own application scripts. In addition, it is possible to use the full set of Dojo controls directly in XPages pages. However, most of those controls have not been tested with XPages so it is up to the application developer to debug any problems encountered.

The Dojo Toolkit resources (JavaScript files, icons, style sheets, and so on) are installed on the Domino server, where web URLs can access them. The HTML for individual XPages refers to the main **dojo.js** file, which provides the Dojo infrastructure, and to other Dojo files as needed.

Different versions of Dojo exist, each with a different version number. Whenever you upgrade a Domino server, such as from version 8.5.2 to version 8.5.3, a new, later version of Dojo is installed. If you do an upgrade install, the older version of Dojo for the previous Domino server version remains present in the server's `Domino\data` folder. (From version 8.5.3, now that the supported Dojo is packaged as an OSGi plug-in, the older Dojo version no longer is available on upgrade.) The XPages runtime is verified to run with only a single version of Dojo, the XPages-supported Dojo version for that release, so usually the older version left after an upgrade is unused. The Domino server versions 8.5.2 and 8.5.3 contain two installs of Dojo. Lotus iNotes uses the installed version with the earlier number, and the XPages runtime uses the later Dojo version. The

URLs to Dojo resource locations contain the Dojo version number, so to refer to a Dojo file, your application should use paths like this:

```
src="/.ibmxspres/dojoroot/dijit/themes/tundra/tundra.css"
```

The XPages runtime preprocesses the **/.ibmxspres/dojoroot/** portion of that path. Thus, the HTML output for the page contains a URL that points into the Dojo resource location for the current Dojo version.

You can use this `xsp.client.script.dojo.version` option to choose a Dojo version to use in XPages, from among the Dojo version installs available on the server. Use of any Dojo version other than the version associated with XPages for that Domino release is an unsupported configuration, so you must test to verify that the XPages infrastructure and the XPages controls used in your application work with the version of Dojo you have chosen. The version format is like 1.6.1—that is, three integers separated by dots, read as "major.minor.micro." A text value after the third number is allowed, though not included in version comparisons, so it can be like "1.6.1.xxx," with the text referred to as a qualifier.

The option is usually set in an application's **xsp.properties** file when your application needs to use an unsupported Dojo version. You might need to set the option in the server-wide **xsp.properties** file if you have installed an unsupported version for use in a specific application but it is being detected as the default version and is being used in all XPages applications on the server. In that case, the server-wide **xsp.properties** file should be explicitly configured to use the supported version. When setting the option in the server file, keep in mind that the server **xsp.properties** file is not overwritten on upgrade, so after upgrade, you must edit the file to change to the version supported by the upgraded Domino version.

> **TIP** Note that when setting this option in an application **xsp.properties** file, you also need to disable the resource aggregator by setting the option `xsp.resources.aggregate` to `false`. The aggregator incorrectly uses the files for the server-wide Dojo version, yet the reference to the main **dojo.js** file uses the application Dojo version. Thus, you end up with a mix of Dojo versions in use on the page, giving Client Side JavaScript errors. That issue is still open in version 8.5.3.

Table 1.3 shows the Dojo versions that XPages runtime supports.

Table 1.3 XPages Runtime–Supported Dojo Versions

XPages Version	Dojo Version
Domino 8.5.0 (server only)	Dojo 1.1.1
Notes/Domino 8.5.1	Dojo 1.3.2
Notes/Domino 8.5.2	Dojo 1.4.1
Notes/Domino 8.5.3	Dojo 1.6.1

Reasons to Use Different Dojo Versions

You might want to use different Dojo versions in your application, for a few reasons. As mentioned before, none of these is officially supported, so testing is required.

You might want to upgrade to a later point release of Dojo when that point release has Dojo fixes or new features that you need for Dojo controls used in your application. Point releases are instances when the third part of the version number changes, such as from 1.6.0 to 1.6.1. Different point releases are usually broadly compatible and less likely to result in breaking functionality, although you should read the release notes to be aware of any possible issues.

You might want to downgrade to the version of Dojo used in the previous version of the Domino server after a server upgrade. This is sometimes useful as a quick fix when your application is using a specific Dojo version, such as when the paths to the Dojo resources are explicitly using the version number, instead of using the **/.ibmxspres/dojoroot/** paths mentioned earlier. Alternatively, your application might have been using the `dojox` experimental Dojo controls, and they might have changed significantly in the upgraded Dojo version so that your pages are now broken. As with all unsupported Dojo versions, you must retest your applications with this older Dojo version because the server-side behavior of the XPages runtime controls will have been tested with only the supported Dojo version. It is better to upgrade your application design to work with the new supported version.

You might want to install a source copy of the supported Dojo version for debugging. The version of Dojo installed in the Domino server is compressed to remove whitespace, to use shorter variable names, and to implement other changes to make the files smaller and the code run faster. That makes it more difficult to read the JavaScript code when you need to debug, to investigate some problem. You can install the more verbose source copy of Dojo from the **dojotoolkit.org** website. You will need to give your source copy a version, such as **1.6.1.source** instead of **1.6.1**, because the versions need to be unique. Then in the application, set this `xsp.client.script.dojo.version` option to the version **1.6.1.source** so that the uncompressed source is available for debugging.

As another option, you might want to use an entirely different version of Dojo than that supported by the XPages runtime. Perhaps it has some feature you want to use in your controls, or maybe you are using some third-party Dojo-based controls that work with only a specific Dojo version. This is the riskiest option and the one most likely to introduce compatibility problems in your applications. You will likely need to fully test that your application and the XPages controls it uses work with the proposed Dojo version in all the browsers your application users will be using. Areas to test in particular are the Partial Update behavior, the event handling (for actions configured in the Designer Events view), the Rich Text Editor control, the Date Time Picker, any type-ahead controls, and anywhere you're setting a **dojoType** property on an XPage control or in pass-through HTML.

Installing Multiple Dojo Versions and Determining the Version Used

The Domino server detects the Dojo-installed versions available in various ways. Among the detected versions, an algorithm selects the default Dojo version for the server. In addition, within an application are various factors that determine the Dojo version used for that application.

Since version 8.5.3, there is an XPages Dojo contribution extension point so that the Dojo resources can be provided within a plug-in, not just as a folder under the **Domino\data** directory.

The supported Dojo version in version 8.5.3 is packaged as a plug-in (as mentioned, another version on the server is not supported for XPages, just for Lotus iNotes). The plug-in is installed as a zipped **.jar** file here:

```
{dominobin}\osgi\shared\eclipse\plugins\
com.ibm.xsp.dojo_8.5.3.20110824-1655.jar
```

That Dojo plug-in, shipped with the Domino server, lists the Dojo version in an inner text file:

```
/resource/dojo.properties
```

In version 8.5.3, it lists the version as follows:

```
DojoVersion.versionStr=1.6.1
```

Your server might contain other plug-ins contributing Dojo versions to the XPages runtime. This is most likely to occur if you've installed some XPages Extensibility library of controls that requires and provides some other version of Dojo. Additionally, when you want to install different Dojo versions, you might find it useful to package the alternate Dojo versions as plug-ins.

The other way of installing Dojo versions is as **domjs** subfolders, as in this folder present in Domino 8.5.3 servers and used only by Lotus iNotes:

```
{dominodata}\domino\js\dojo-1.5.1\
```

When providing Dojo versions as folders, they must be present in that **Domino\data\domino\js** parent folder. That parent folder is known as **domjs** because, on the Domino server, the URL used to access the subfolder will be like http://server.example.com/domjs/dojo-1.5.1.

The name of the Dojo folder under **domjs** must match that format, with **dojo-** and the **major.minor.micro** version number. A folder name such as **dojo-1.5.1.xxx** is also possible, but the extra text at the end is not included in version comparisons. If the folder name does not match either of those formats, it will not be recognized as an installed version of Dojo. Because the XPages Dojo extension point became available only in version 8.5.3, in the older servers, the only way to provide Dojo versions is through **domjs** folders.

The contents of the Dojo resource folder should be like this:

```
dijit/
dojo/
dojox/
ibm/
```

The first three folders, **dijit**, **dojo**, and **dojox**, are standard folders that are available in the download zips from the Dojo Toolkit website.

The **ibm/** folder contains extra JavaScript files that the XPages runtime requires. When installing a new Dojo version, you need to copy the **ibm** folder from the supported Dojo location into your new location so that those files are available to the XPages runtime. (Anyone using the version 8.5.3 XPages Dojo extension point will notice that it was designed so that you don't need to copy the **ibm** folder, but that feature is unavailable at this time.)

At a server level there is a detected default Dojo version, usually used when individual applications do not specify an explicit version in the **xsp.properties** option.

If the server-wide **xsp.properties** file contains the option `xsp.client.script.dojo.version` and it matches one of the installed Dojo versions, that will be the default dojo version for the server.

Otherwise, the default version is chosen from among the installed versions. Dojo installs provided as domjs folders are considered candidates for default, unless the version is like **1.1.1.xxx**. The text at the end after the "major.minor.micro" numbers disqualifies them as candidates. Dojo resource locations provided through the XPages Dojo extension point are considered candidates when their implementation of the method `DojoLibrary.isDefaultLibrary()` returns `true`. The candidate location with the highest version number becomes the default server dojo version.

Within an application, the version of Dojo used depends on the value of the `xsp.client.script.dojo.version` option and also on the maximum dependency Dojo version.

The max dependency Dojo version applies only when your application uses any XPages Extensibility library of controls. As explained in the `xsp.library.depends` option, an application can be configured to use multiple libraries of controls other than those that the XPages runtime provides. A library can declare a required version of Dojo, indicating that the library will not work if the Dojo version the application uses is less than the required version. When an application depends on multiple libraries requiring different minimum versions of Dojo, the application must use a Dojo version the same or greater than the maximum Dojo version required by any of the libraries. That maximum is known as the max dependency Dojo version.

In an application, when the `xsp.client.script.dojo.version` option is explicitly configured, that version of Dojo is used. Note that the version can have text after the version numbers, as in **1.6.1.source**. If the application-configured Dojo version

(including any text after the numbers) does not exactly match one of the installed Dojo versions, an error will prevent the application from running. If the application is using some control libraries, the option value is validated against the max dependency Dojo version. If the version in the option is less than the max dependency version, an error will prevent the application from running. That can happen if you upgrade some library on your server so that the dependency Dojo versions change and your application-specified version is no longer viable.

When the option is not configured in the application, either the max dependency version or the server default Dojo version is used. The one with the highest version number applies.

xsp.client.script.dojo.djConfig

This option can add extra values to the `djConfig` object in the page header besides values that XPages automatically outputs. Listing 1.26 includes the relevant section of the **xsp.properties** file.

Listing 1.26 xsp.properties Snippet for the xsp.client.script.dojo.djConfig Property

```
# Add parameters to the djConfig attribute of Dojo.
# Useful to switch Dojo to debug, using for example:
#   xsp.client.script.dojo.djConfig=isDebug:true
#xsp.client.script.dojo.djConfig=
```

The full description of parameters to the `djConfig` object is part of the Dojo Toolkit documentation at **dojotoolkit.org**. The value format is the contents of a Client Side JavaScript object, so it can have multiple name and value pairs. A colon (`:`) separates the name and value, and commas (`,`) separate the different pairs. The value is written in the HTML page as an attribute, so it should not contain double quotes (`"`) or newline characters. Consider two examples of the format:

```
someBoolean:true, someString: 'text', someNumber: 20
someArray: ['text1','text2'], someObj: {name1:'value1', name2:'value2'}
```

This option can be set per XPage, in the XPage root control property named `properties`. Unlike in some other options, the behavior is not solely determined by the value in the application and server **xsp.properties** files.

The option is most commonly used to set the parameter `isDebug:true`. That parameter is used when your page has some Client Side JavaScript error, to find out more information about the cause. The option is useful when writing complicated Client Side JavaScript code or when attempting to use Dojo controls instead of the predefined XPages controls by setting a `dojoType` property on an XPage control or in an HTML snippet in the XPage source. Since version 8.5.3, by default, the XPages resource aggregator appends all the Dojo **.js** files into a single file. For debugging purposes, then, it is usually necessary to disable the resource aggregator by setting the option `xsp.resources.aggregate` to `false`. Also, the **.js** files on the server are compressed,

with formatting whitespace removed. To make the files more readable for debugging purposes, it may be useful to replace the server's dojo resources with a source-uncompressed version of the same, downloaded from the Dojo Toolkit website. Uncompressed versions of the XPages runtime files are included in the Domino server, but under names such as **file.js.uncompressed.js**; you'll need to do some file renaming to use the uncompressed versions. For more on debugging Client Side JavaScript, see Chapter 4, "Working with the XSP Client Side JavaScript Object."

Besides the parameters explicitly configured in this option, the XPages runtime outputs other parameters to the `djConfig` object, some by default and others depending on property settings in the XPage.

Every XPage outputs the locale parameter, as in **locale: 'en-us'**. That usually corresponds to the language and region of the user's web browser, although if the application is translated but not to the user's language, the application's default language may be used instead. For more on how the XPages locale is chosen, see the Internet document "Locale Use in XPages."

Another `djConfig` locale parameter, **extraLocale**, may be useful to configure in this option when your application has pages containing multiple languages. If you need to specify that option, keep in mind that the XPages locales are Java locales, as with **en_US**, which need to be explicitly converted to a Dojo locale, such as **en-us**. Some locale conversions are complicated, as with the language Indonesian, which is `in` in Java but `id` in Dojo. For more details, in the Internet document "Locale Use in XPages," see the section "Norwegian and the Deprecated Locale Codes."

The `djConfig` parameter `parseOnLoad:true` is usually set by the XPages runtime instead of through this option. It is not always present on an XPage, but the presence of certain XPage controls causes the option to be set to `true`. In addition, the XPages root control has a Boolean property `dojoParseOnLoad` that can be set to output this parameter. The `parseOnLoad` parameter means that the Dojo infrastructure detects any `dojoType` attributes in the HTML source and converts those elements to Dojo controls or modules.

A `djConfig` parameter named `modulePaths` provides locations where Dojo will search for Dojo module JavaScript files. To output a module path, an XPage resource can be set in an individual XPage. Actually, it outputs the module path as a separate line in the header, but the effect is similar to when the value is set in the `djConfig` parameter, so the `djConfig modulePaths` parameter is not generally used in XPages.

The module path resource can be used when providing Dojo-based controls or modules as JavaScript files in your application. Listing 1.27 provides the XPage source for such a page.

Listing 1.27 Setting the Dojo Module Path in XSP Markup

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="www.ibm.com/xsp/core"
     dojoParseOnLoad="true">
```

```
  <xp:this.resources>
    <xp:dojoModulePath prefix="inapp" url="/" />
    <xp:dojoModule name="inapp.CustomButton" />
  </xp:this.resources>
  <xp:button value="Label" id="button1"
    dojoType="inapp.CustomButton" />
</xp:view>
```

There is also an XPages extension point for module paths, used in XPages Extensibility control libraries. For example, the extension point is used by the XPages Extension Library project on OpenNTF.org. When that library is installed, any Dojo module names beginning with `extlib.` are not searched for in the Domino server's Dojo folder. Instead, they are registered as being available from a URL folder pointing into the Extension Library plug-in. The URL to an individual module JavaScript file is in this form: **http://server.example.com/xsp/.ibmxspres/.extlib/dijit/Tooltip.js**

The HTML Page-Generation Properties

This category of properties provides you with several properties that can be used to configure the emitted HTML markup from the XPages runtime. This includes document and content type, encoding, compression mode, client-side validation, and client or server redirection.

xsp.html.doctype

This property gives you control over the `<!DOCTYPE>` tag that is emitted as the first line of all your XPages at runtime. This is not an HTML tag, but a declaration to the browser for the document type definition (DTD) used to create the page that is being served up—essentially, the type and version of the markup language. Some good summary information is provided in the **xsp.properties** file, in Listing 1.28.

Listing 1.28 xsp.properties Snippet for the xsp.html.doctype Property

```
# Defines the document doctype generated by the engine
# Defaults to HTML 4.01 transitional, but XHTML is available with:
#   html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN
#   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
# Note that when using XHTML, the content type is still text/html as
➥IE,
# as well as Dojo, don't support application/xhtml+xml
# xsp.html.doctype=HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
➥http://www.w3.org/TR/html4/loose.dtd"
```

You can specify the literal `<!DOCTYPE>` option directly in the **xsp.properties** file or via Domino Designer using **Application Properties > XPages > HTML doctype:** combo

box menu. With the latter, you can make your selections using the logical doc type names, as follows:

- HTML Strict
- HTML Transitional
- XHTML Strict
- XHTML Transitional
- HTML5

These are transformed into the following `<!DOCTYPE>` declarations at runtime, respectively:

```
<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\"
\"http://www.w3.org/TR/html4/strict.dtd">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
\"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\"
\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\"
\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!DOCTYPE html>
```

HTML Transitional is the runtime default. Even though you can change this setting using this property, you must consider certain issues. As pointed out via the comments in the **xsp.properties** file, not all browsers fully support XHTML mode. Using a strict HTML DTD can cause problems with Dojo controls within XPages—for instance, the `dojoType` attribute is used by the XPages Date Time Picker renderer when generating the markup for the Dojo control. Because `dojoType` is not part of the HTML specification, the emitted markup fails a strict validator.

Similarly, you can have HTML content on an XPage that does not emanate from the XSP design markup at all, but rather from the document data itself. Suppose that your XPage contains a Rich Text control instance and you are working with one originally created on the native Notes client. The data in the Notes rich-text field has been transformed into HTML by an internal CD-MIME engine. The resulting HTML might not be well formed or might contain deprecated HTML tags, which would also result in validation failures.

These are issues to bear in mind if you are considering a `<!DOCTYPE>` change.

xsp.html.meta.contenttype

This property is really a convenient means of inserting an HTML `<meta>` tag into the emitted page markup as the first line in the `<head>` section. Simply set the property to the desired Boolean value, as shown in Listing 1.29.

Listing 1.29 xsp.properties Snippet for the xsp.html.meta.contenttype Property

```
# Ask the XPages runtime to generate a <meta> tag in the HTML header
# defining the content type, and the optional  character set.
# This meta tag is the first tag appearing after the <head> one.
# For example, it generates something like:
# <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
#xsp.html.meta.contenttype=false
```

The tag includes `http-equiv` and `content` attributes, with the values reflecting the settings applied by the runtime. For example, if such a metadata declaration is required by policy for all web pages in an application or organization, setting this property makes the implementation simple.

The alternative is to stipulate the metadata manually, by inserting `<xp:metadata>` tags with the required attribute values into the `<xp:resources>` of each XPage, which is cumbersome. Extra metadata, over and above that automatically specified by this property, can be added to any page using the `<xp:metadata>` tag. You will end up with multiple `<meta>` tags in the `<head>` section of the emitted markup, but that is okay.

xsp.html.preferredcontenttypexhtml

Listing 1.30 summarizes this property in the **xsp.properties** file.

Listing 1.30 xsp.properties Snippet for the xsp.html.preferredcontenttypexhtml Property

```
# Force the content type to be application/xhtml+xml if the user agent
# supports it
# WARN: this has to be used very carefully as some features won't work
# with an XML content type. For example, the innerHTML JavaScript
# property is read only and then breaks Dojo or XPages partial refresh.
# Moreover the RichText fields converted to MIME are not XHTM
➥compatible
# This option should only be used in very particular cases
# xsp.html.preferredcontenttypexhtml=false
```

Although you can use the `xsp.html.doctype` property to set the `<!DOCTYPE>` of the emitted page to XHTML, the `Content-Type` response field is still set to `text/html` to guard against the potential pitfalls described for that property, such as non-XHTML-enabled browsers and noncompliant XHMTL markup generated in certain use cases. If your application is not affected by such issues and you want to coerce the `Content-Type` field in the response header to explicitly specify the `application/xhtml+xml` type (which makes it a real XML file), simply set this `xsp.html.preferredcontenttypexhtml` property to `true`. It will be applied as long as the end user's browser includes `application/xhtml+xml` as an `Accept` parameter value.

xsp.html.page.encoding

This property is used to set the preferred character encoding for the web pages generated by your XPages. The XPages default encoding in utf-8, as shown in Listing 1.31.

Listing 1.31 xsp.properties Snippet for the xsp.html.page.encoding Property

```
# Defines the character set returned for the page
#xsp.html.page.encoding=utf-8
```

UTF-8 is part of the Unicode standard and is the preferred character encoding for web pages. However, you can specify alternative character encoding standards, although this is not a common occurrence in practice, at least from this author's experience. A comprehensive selection of character encodings is made available in Domino Designer via the Application Properties editor, as shown in Figure 1.14.

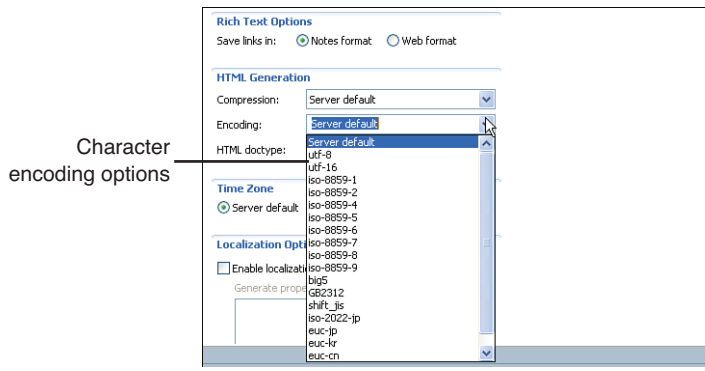


Figure 1.14 Application Properties, HTML Generation > Encoding Options

Picking an entry from the combo box writes the value as the `xsp.html.page.encoding` setting in the local **xsp.properties** file, while the Domino server/Notes client file provides a convenient global override mechanism.

xsp.compress.mode

XPages supports a number of compression options that are designed to give maximum control over the manner in which XPages content is sent from the server to the client browser. Good summary information is provided in the xsp.properties file, as shown in Listing 1.32.

Listing 1.32 xsp.properties Snippet for the xsp.compress.mode Property

```
# XSP compression
# This defines the compression mode used when a page is rendered to
# the client. The compression is effectively enabled when the client
```

```
# supports it, as specified in the HTTP header of the request.
# The possible values are
# none
#  no compression at all
# gzip
#  the response contains the content-length header, which forces
#  the data to be buffered.
#  this is the prefered mode and is required to support HTTP 1.1
#  persistent connections.
# gzip-nolength
#  The content is compressed but it doesn't compute the content-length,
#  Thus it doesn't need to buffer the result
#xsp.compress.mode=gzip
```

Compressing text resources such as HTML, CSS, and JavaScript can boost runtime performance by drastically reducing the physical size of the data that is sent from the server to the browser when rendering an XPage. The default mode specifies an encoding format called gzip, short for GNU zip. XPages content is compressed using this format by default, as long as the browser client can handle this type of encoding. The compression mode negotiation occurs between browser and server using the request/response headers. If the browser specifies an `Accept-Encoding` of `gzip` in the request, XPages responds with gzipped content and sets the `Content-Encoding` field in the response header to `gzip`. The size of the compressed content is also computed and included in the `Content-Length` response header field, unless `gzip-nolength` is specified as the preferred compression mode. When the `Content-Length` field is set, all the content is buffered before being sent to the browser. When not set, the response is sent to the browser immediately, which can be advantageous in low-bandwidth situations as the page will incrementally be rendered by the browser as it receives content from the server.

You can access this feature directly through Domino Designer using the **Application Properties > XPages > HTML Generation > Compression** combo box options.

xsp.client.validation

This property specifies whether to apply client-side validation. The default setting is `true`, as shown in Listing 1.33.

Listing 1.33 xsp.properties Snippet for the xsp.client.validation Property

```
# Enable the client validation - default to true
#xsp.client.validation=true
```

User input validation is a fundamental aspect of any application. Any input control included on an XPage can have validation conditions attached so that the data ultimately submitted by the end user at runtime can be checked against particular criteria. The

validation criteria might be that a required field is not empty or that a number in another field falls within a certain range. Any stipulated validation criteria are *always* executed on the server side by the XPages runtime as part of the validation phase of the request lifecycle. In other words, validator code is always executed on the Domino server after the XPage is submitted from the browser agent. However, XPages also provides the option of having validation performed on the client side—that is, it provides the capability to apply the validation criteria in the browser container *before* the page is submitted to the server. In this scenario, if the data on the page does not pass the validation conditions, the page is not submitted to the server at all and the user is notified with one or more error boxes in the browser. This can result in an improved user experience—because of quicker validation feedback, for example—and thus client-side validation is enabled by default in the XPages runtime.

If client-side validation is not really required for a given application—remember, server-side validation is performed regardless—you can simply turn it off. For a particular application, you can do this via Domino Designer by selecting the **Application Properties > XPages > Client Validation > Off** radio button. This action results in `xsp.client.validation=false` being written to the local **xsp.properties** file. If the **Server Default** option is chosen, the preference is taken from the **xsp.properties** file on the Domino server (or from the **xsp.properties** file in the Notes client if running on that platform, which makes the UI name somewhat confusing).

When client validation is turned on, client validation code is rendered at runtime in the emitted HTML for any input control on the XPage that has validators attached. Note that individual controls can also opt out of the client-side validation either by setting their own `disableClientSideValidation` Boolean property to `true` or by setting the `disableValidators` Boolean property on event handlers belonging to the control to `true`. Finally, be aware that not all standard validators have a client-side implementation.

xsp.redirect

The `xsp.redirect` comment included verbatim from the **xsp.properties** file, in Listing 1.34, summarizes this feature concisely.

Listing 1.34 xsp.properties Snippet for the xsp.redirect Property

```
# xsp Page redirect mode - This happens when the runtime redirects
# to a new page (navigation rules, API, simple action)
# When this property is true, then the runtime emits an HTTP 302 code
# to the ask client to redirect to the new page.
# This ensures that the client has the right URL in its address bar,
# at the cost of an extra client/server roundtrip.
# Else, the redirection is purely done on the server, without any
# notification to the browser (the URL doesn't change)
#xsp.redirect=true
```

When something happens within the runtime on the server side that changes the current page, as with the execution of a navigation rule, simple action, or particular set of API calls, XPages can simply render the new page to the browser client or instruct the client to request the new page. The latter means that the browser always shows the correct URL in the address bar. This is critical if your application needs to support page bookmarking, enabling users to return to a particular part of your application at some future point via a browser bookmark. The `xsp.redirect` property defaults to `true`, but you can set it to `false` to gain some performance optimization if bookmarking is not important and you generally have no requirement to have the browser address bar accurately reflect the active page.

The Error-Management Properties

These options control the application's response to the browser when an unanticipated problem occurs. In those situations, instead of displaying the current application page or the next page triggered by some action, an error page displays and reports that some problem occurred. These errors are different from the anticipated validation errors, which are displayed in the Display Errors controls, such as messages indicating that a required field has not been supplied. Such validation errors are expected to occur as users fill out forms, and the message indicating the problem is displayed within the current page so that users can continue editing their form and not lose the values they entered so far.

The default behavior for an XPages application is that the underlying web server handles any errors encountered and returns a simple page with the HTTP error code, along with possibly a one-line description of the problem. The most common such error is probably the `Error 500 - Command Not Handled` error page, which indicates a problem in the application code (as opposed an authentication problem, unknown page, or other server-level problem).

These options allow different pages to be displayed instead of the simple web server error pages.

xsp.error.page.default

This option, if set to `true`, displays an error page provided by the XPages runtime as the default in-built page for debugging problems in an XPages application. Listing 1.35 shows the relevant section from the **xsp.properties** file.

Listing 1.35 xsp.properties Snippet for the xsp.error.page.default Property

```
# Defines if the default error page should be displayed by the XSP
# layer this is very useful in development as it displays extra
# information on the error
#xsp.error.page.default=false
```

The setting can be set for the entire server by changing the server **xsp.properties** file, but it is more normal to set the option for a specific application. To set the option for an application, in **Application Properties**, choose the **XPages** tab and check the check box **Display XPages runtime error page,** as shown in Figure 1.15. (In earlier versions, that option was named **Display default error page**—the behavior is unchanged.)

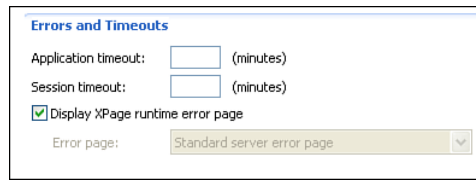


Figure 1.15 Application Properties, Display XPages runtime error page

You normally do not use this option in a production environment because it is intended only for debugging. It is considered less scary for an end user to see an Error 500 page than to see a page with a stack trace. However, the application designer can choose what works best for the application.

The default built-in error page has the text **Unexpected runtime error**, displays the messages from the exception that caused the problem and from any ancestors that cause exceptions, and enables the user to toggle open an area to see the full stack trace of the problem. In addition, if the error occurred within some JavaScript code in the XPage, the default error page displays the snippet of JavaScript code and highlights in red the line where the problem occurred, as shown in Figure 1.16.

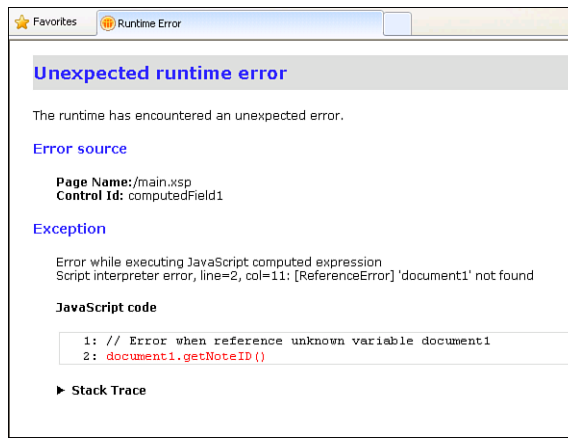


Figure 1.16 Default error page with JavaScript snippet

Unlike the user-provided error page, limitations govern the circumstances under which the XPages default error page will appear. That error page is displayed only if the encountered problem is a `java.lang.Exception`, not when the problem is the more serious `java.lang.Error` type. When a `java.lang.Error` is encountered, the problem is written to a log file and the default web server Error 500 simple error page is displayed. It does not attempt to display the default error page for such serious problems, which can occur when the JVM has run out of memory or required class dependencies cannot be resolved, because the problems are so serious that attempting to display the default error page will probably fail, too. In that situation, it is best to log the information currently available rather than risk losing that information or complicating the effort to debug the problem by filling the error logs with problems caused by failing to output the default error page.

When building your own user error-provided error page, it may be best to treat such `java.lang.Error` problems in a similar manner. However, such errors are passed to the custom error page in case the page designer wants to handle certain types of `java.lang.Error` problems differently, perhaps partially recovering from the problem.

When we refer to the default error page, we generally mean that page with the main message `Unexpected Runtime Error`. Some other, less frequently occurring error pages provided by the XPages runtime are also displayed only when the default error page option is enabled. The other runtime error pages are used at a lower level of code, closer to the web server engine. An example is a page that says `Page Not Found` instead of `Unexpected Runtime Error` but otherwise looks the same, including the exception messages and the area where the user can toggle open the stack trace.

The HTTP response code for the error page was incorrect in releases before version 8.5.3. Previously, it returned `HTTP status 200 OK`, indicating that the XPage had displayed without any problem. Now, since version 8.5.3, it returns `HTTP status 500 Internal Server Error`, indicating that some error occurred in the XPage it was attempting to display.

Chapter 6, “Server-Side Debugging Techniques,” details how the XPages runtime logs error and trace information.

xsp.error.page

This option can be set to an XPage name to display an XPages error page provided by application developer, instead of the usual web server error page or the XPages runtime default error page. This is done in the snippet of **xsp.properties**, shown in Listing 1.36.

Listing 1.36 xsp.properties Snippet for the xsp.error.page Property

```
# Defines an XSP specific error page
# When not defined, it displays a default error page
#xsp.error.page=
```

To set the option, create a new XPage that will be used for your error handling. Then in the **Application Properties**, **XPages** tab, under **Errors and Timeouts**, in the **Error page:** drop-down list, choose your XPage.

To quickly test that your custom error page is used, in some other XPage of your application, set `loaded="true"` on the XPage root control and then open that XPage in your browser. Your custom error page is displayed instead of the XPage that you opened in the browser, because an error returns when the root XPage control is not loaded.

To refer to the problem that caused the custom error page to be displayed, you can reference the **requestScope.error** value. That object will be a `java.lang.Throwable`, which has the information about where the problem occurred in your application. The commonly used methods there are `getMessage()`, which gives the text of the problem; `getCause()`, which may give another `Throwable` that caused this problem (and possibly a chain of `Throwable` causes); and the methods for printing and accessing the Java stack trace that can be read by a Java developer to understand how the problem occurred.

In addition, if the problem in the application occurred during the execution of some Server Side JavaScript code, it is possible to find which code was executing and the line number and column number where the problem occurred. That information is available by checking whether the error object and its cause implement `com.ibm.xsp.exception.XSPExceptionInfo` and `com.ibm.jscript.InterpretException` (information about those interfaces is part of the XPages Extensibility JavaDocs available on the Internet). For example, Listing 1.37 provides a simple error page.

Listing 1.37 XSP Markup for a Sample Error Page

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="www.ibm.com/xsp/core">
  <b>An error occurred</b>
  <pre>
    <xp:text escape="true" id="computedField1">
      <xp:this.value><![CDATA[#{javascript:
        var output = requestScope.error.toString()+"\n";
        output+="\n";
        if( requestScope.error instanceof
                 com.ibm.xsp.exception.XSPExceptionInfo ){
          var codeSnippet = requestScope.error.getErrorText();
          var control = requestScope.error.getErrorComponentId();
          var cause = requestScope.error.getCause();
          output+="In the control : "+control+"\n";
          output+="\n";

          if( cause instanceof com.ibm.jscript.InterpretException ){
```

```
            var errorLine = cause.getErrorLine();
            var errorColumn = cause.getErrorCol();

            output+="At line "+errorLine;
            output+=", column "+errorColumn+" of:\n";
          }else{
            output+="In the script:\n";
          }
          output+=codeSnippet;
        }
        return output;
      }]]></xp:this.value>
    </xp:text>
  </pre>
</xp:view>
```

That page displays in a browser, as follows in Figure 1.17.

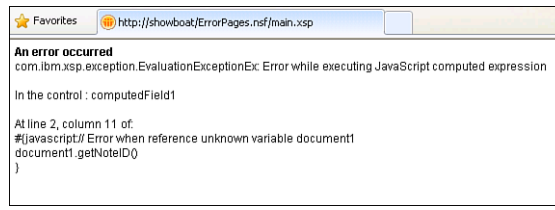


Figure 1.17 Custom error page with JavaScript snippet

In general, you design your error page so that its appearance matches the design of the rest of your application.

However, when developing your error page, keep in mind that you cannot use any XPages server-side simple actions or any server-side Events. This is because the URL used for the custom error is the same as the URL for the page that was opened when the problem was encountered, so any attempt to communicate with the server-side representation of the page will contact the wrong XPage.

Note that the HTTP response code for the error page defaults to `HTTP 500: Internal Server Error`. You might want to update that response code if the encountered error more closely matches one of the other HTTP response codes. To update the status code, use the API shown in Listing 1.38.

Listing 1.38 Server-Side JavaScript Snippet to Update HTTP Response Code

```
var responseObj = facesContext.getExternalContext().getResponse();
responseObj.setStatus(404/*Page Not Found*/);
```

Note that, even when using this option, some problems might not display in your custom error page, so you might still want to set the `xsp.error.page.default` option to make it easier to debug such issues. That might happen for some problems that occur at a lower level in the web server, where the facility for loading an XPage might not be available and it is not possible to load the custom error XPage.

It is also worth noting that if a problem arises in the display of the custom error page itself, that problem is noted in the log file and the default web server error page is displayed to report the original problem. (The default page is usually a simple Error 500 page.)

The most common occurrence where the custom error page cannot be displayed arises when the original problem was severe. An example might be a Java `OutOfMemoryError`, in which no further objects can be created because all the memory available to the server has been consumed. In general, problems that implement `java.lang.Error` are likely to be unrecoverable. If users of your application are likely to encounter such problems, you should probably redesign your application to prevent them from occurring.

The User Preferences Properties

These options control the detection of the end user's browser settings and preferences.

xsp.user.timezone

XPages supports different time zones, so dates can be displayed in the user's local time, using the server's time zone, or using a programmatically configured time zone. Listing 1.39 shows the relevant section of the **xsp.properties** file.

Listing 1.39 xsp.properties Snippet for the xsp.user.timezone Property

```
# Defines the timezone to use
# When not specified, it uses the server timezone.
#xsp.user.timezone=false
```

The time zone is used when displaying date objects as text. The date objects contain the universal region-independent time (for example, 1pm in London today, saved in milliseconds), and the current time zone is used to display the time. For example, it knows to display the date as 8am EST when the web server is located in the eastern USA. By default, in XPages, the server time zone is used. So if you have users in different time zones, the exact same time is shown to each user instead of using the local time for that user. This option can be set to `true` to use the user's browser time zone instead of the server's, so that users see their local time.

You can change the time zone that an individual application uses in **Designer**, in the **Application Properties**, **XPages** tab, as shown in Figure 1.18. The **Time Zone** section enables you to choose from three values. The default value is **Server Default**, meaning that the option is not set in the application, so the value in the server-wide **xsp.properties** file is used. The other values are **Browser**, meaning the user's web browser's time

zone, and **Server,** meaning the time zone on the web server where the XPages runtime is running.

Figure 1.18 Application Properties, Time Zone option

It is also possible to set Time Zone programmatically instead of detecting the browser or server time zones. The current Time Zone for an XPage can be accessed through the `context` object, through Server Side JavaScript. You can create a Java `TimeZone` object and set it as the current time zone. The value set then is used for the rest of the current user session. Unlike other values that can be set onto the context, it is not necessary to reload the page after changing the time zone. The time zones values are either `TimeZone` objects or text IDs defined by the `java.util.TimeZone` class, as in **America/Los_Angeles**. Not all common names for time zones are supported, so you might need to refer to the documentation for that class. That documentation also describes the custom time zone syntax; for example, **GMT+5** means 5 hours ahead of GMT.

Listing 1.40 provides the time zone methods on the context object.

Listing 1.40 TimeZone API Method Calls Available on Context Object

```
context.getTimeZone() : TimeZone
context.getTimeZoneString() : string
context.setTimeZone(timeZone:TimeZone)
context.setTimeZoneString(timeZone:string)
```

The server time zone is always available through the I18n library, like so:

```
I18n.getServerTimeZone(): TimeZone
```

In addition, it is possible to set the time zone for an individual Edit Box or Computed Field. Verify that the control display type is set to **Date/Time** in the Properties view, **Data** tab. Then in the **All Properties** tab, **Data** category, either configure the converter's `timeZone` property to a value such as **America/New_York** or compute the property to programmatically determine the time zone displayed.

xsp.user.timezone.roundtrip

This property is a simple Boolean option, as shown in Listing 1.41.

Listing 1.41 xsp.properties Snippet for the xsp.user.timezone.roundtrip Property

```
#
#xsp.user.timezone.roundtrip=true
```

When the option `xsp.user.timezone` is `true`, so that the user's browser time zone is being used, you can sometimes change the round-trip option to `false`, as an optimization.

The round-trip it refers to is part of how XPages detects the browser time zone. The time zone information is not automatically passed by the browser to the server. So XPages computes a time zone using a piece of JavaScript in a temporary page and then submits the result to the server and redirects to the actual XPage. The piece of JavaScript produces a rough guess of the time zone, so it finds **GMT+5** instead of **America/New_York**. If the first XPage in your application does not use any time zone information, you can change the setting to avoid the temporary page and prevent that round-trip to the server. In that case, the piece of JavaScript is output in the first XPage you open and the browser time zone is available only after the user submits for the first time. Often it is not possible to use that option because users might bookmark pages within your website, so you cannot predict which will be the first page they open in any session.

The AJAX Properties

The option in this section controls some of the behavior during partial update (also known as partial refresh). In a partial update, a subsection of the web page is updated by submitting and retrieving an updated snippet of the page from the web server. It uses an AJAX request, which is a popular Web 2.0 technique. The acronym stands for asynchronous JavaScript and XML, although the XPages partial update request actually transfers a snippet of HTML rather than XML.

xsp.ajax.renderwholetree

This option allows a performance improvement while processing a partial update (also known as partial refresh). Listing 1.42 shows the summary information provided in the **xsp.properties** file.

Listing 1.42 xsp.properties Snippet for the xsp.ajax.renderwholetree Property

```
# This property defines if the JSF tree should be
# completely processed during the render phase,
# including the components that are not rendered. When set to false
# it gives better performance but with potential side effects
# if some components are changing data during the render phase (which
# should be avoided anyway)
#xsp.ajax.renderwholetree=true
```

The option was added in version 8.5.1 and defaults to `true`, meaning to use the old slower way of handling partial update, the same as the behavior in 8.5.0. Version 8.5.2 implemented a change so that applications that are newly created in Designer 8.5.2 or later have this setting set to `false`, with faster handling of partial updates. If you have existing applications that were created in version 8.5.0 or 8.5.1, it is usually possible to set this value to `false` so that partial updating is faster. However, some applications

that depended on the old behavior might fail when this value is changed, so you must retest applications after you change this setting.

To change this value in an application, you can edit the **xsp.properties** file directly or change the check box in the Application Properties. In the **Application Properties**, **XPages** tab, in the section **Performance**, the check box is named **Evaluate the entire page on partial refresh,** as detailed in Figure 1.19.

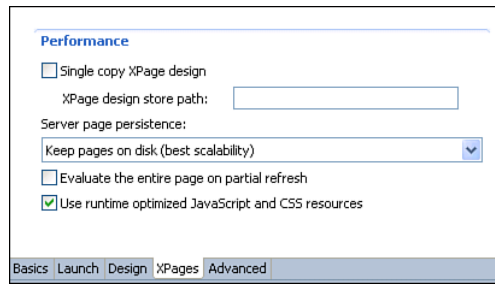


Figure 1.19 The Application Properties, XPages tab, Performance section

The behavior this option controls occurs during the server-side processing of partial update events and affects only the Render phase of the JSF lifecycle.

The default, slower behavior means that the render phase navigates through all controls in the server-side tree of controls. It doesn't generate HTML for the controls outside the update area, and it doesn't execute their "rendered" property, but it does publish the data sources for each control. In fact, all variables that are published by controls are made available, including the Repeat control's `indexVar` variable and the Custom Control's `compositeData` variable.

The improved behavior means that, during the render phase, it navigates through the controls until it finds the update area. After generating the HTML for the update area, it does not navigate through the rest of the controls. Also with the improved behavior, when navigating through controls before the update area, if it finds controls that implement the `NamingContainer` interface, it does not search through any naming container that does not contain the update area. So in effect, the naming container protects its contents' controls from being navigated through and having their variables published. The most common naming container controls are the repeating controls (Repeat, Data Table, and View Panel) and the Custom Control container.

The application for this chapter contains the sample XPage **testRenderWholeTree** demonstrating this behavior. The page has print statements that are emitted to the server console when various data are published, when the render phase starts and ends, and so on. The page starts with an update button to update Panel4; Panel1 is the first panel configured with data to output a print statement to the server console, a Repeat control contains Panel2, then after the Repeat is Panel3 (which has a child Panel4 that is updated), and after Panel3 is Panel5.

To test, set the `xsp.ajax.renderwholetree` option to `true`, meaning to navigate the whole tree. Then click the button **Update Panel4** on that page. Listing 1.43 shows the console output.

Listing 1.43 Console Output for the Sample Page with xsp.ajax.renderwholetree=true

```
beforeRenderResponse
data in Panel1 published
data in Panel2 published, in Repeat1, index 0
data in Panel2 published, in Repeat1, index 1
data in Panel2 published, in Repeat1, index 2
data in Panel3 published
data in Panel4 published
data in Panel5 published
afterRenderResponse
```

The output indicates that the data is published in all the controls before the updated Panel4. Then the data is published in Panel4 itself and, finally, the data is published in the control after Panel4, in Panel5.

Next, to test the optimized behavior, set the `xsp.ajax.renderwholetree` option to `false`. Then click the button **Update Panel4** on that page. Listing 1.44 shows the updated console output.

Listing 1.44 Console Output for the Sample Page with xsp.ajax.renderwholetree=false

```
beforeRenderResponse
data in Panel1 published
data in Panel3 published
data in Panel4 published
data in Panel4 published
afterRenderResponse
```

With the updated behavior, after the render phase has found Panel4, it stops searching, so the data for Panel5 is not present in the output. The data in Panel2 within the Repeat1 is not output because the Repeat control is a `NamingContainer` and it acts to prevent data within that area from being published. The data in Panel4 is published twice—once while searching and again while generating the HTML output.

You can see that when the option is set to `false`, less of the control tree is navigated through and less of the data is published. This means that less work is done on the server and the response is available to the browser more quickly.

An application might fail to function if this option is changed from `true` to `false`. The application may have been designed to compute and set values during property evaluations that happen while data is published. For example, when the Custom Control container publishes the `compositeData` variable, all property values set to that Custom

Control tag are evaluated. If the application is relying on those property evaluations happening at specific times and you change the option to `false`, those property evaluations might not occur and your application might break. It is usually possible to redesign your application to avoid such problems, or it might be convenient to set this option back to `true` so that the application runs slightly slower but still functions.

The Script Cache Size Properties

Two available properties within this category enable you to control the optimization of compiled script expressions within the XPages runtime. The first time an expression is used at runtime, the Server Side JavaScript, and any XPath computed expressions, are further decomposed into optimized, line-for-line expressions. The optimized expressions are then cached for later reuse. This enables an application to execute faster at runtime because there is no need for the XPath runtime to keep parsing Server Side JavaScript or XPath expressions for every XPage request. Instead, the XPages runtime leverages an expression-caching mechanism to accommodate Server Side JavaScript and XPath-computed expressions. When an application is loaded, the expressions within any given XPage are cached for use during any subsequent request of that XPage.

Server Side JavaScript is the default scripting language for XPages. However, a built-in XPath expression engine supports expressions for working against XML-based document stores. This XPath engine has been available within XPages since version 8.5. Furthermore, because XPages is built upon the JavaServer Faces framework, there is indeed a third scripting language option by way of Expression Language, or EL, as it is more commonly termed. This is the scripting language technology provided by the underpinning JSF framework and is readily usable within XPages applications also. However, it is important to point out that there is no facility for caching EL expressions.

ibm.jscript.cachesize

The Server Side JavaScript expression cache is set by default to a maximum limit of 400 computed expressions, as shown in Listing 1.45.

Listing 1.45 xsp.properties Snippet for the ibm.jscript.cachesize Property

```
# This controls the number of compiled JavaScript expressions.
#ibm.jscript.cachesize=400
```

In a server with high available JVM memory, this property value can be increased accordingly based on benchmark testing results to find an optimal setting based on the demands of the particular application use case.

ibm.xpath.cachesize

The XPath expression cache is set by default to a maximum limit of 400 computed expressions, as shown in Listing 1.46.

Listing 1.46 xsp.properties Snippet for the ibm.xpath.cachesize Property

```
# This controls the number of compiled XPath expressions.
#ibm.xpath.cachesize=400
```

Similar to the `ibm.jscript.cachesize` property, any adjustment of this property value must be determined by adequate benchmark testing to establish an optimal setting.

For your interest, Listing 1.47 shows the XSP markup code for an XPage containing XPath-computed expressions.

Listing 1.47 Example of an XPage Containing XPath-computed Expressions

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="www.ibm.com/xsp/core">
      <xp:this.beforePageLoad>
       <![CDATA[#{javascript:
            var document = DOMUtil.createDocument();
            var person = document.createElement("person");
            document.appendChild(person);
            var firstName = document.createElement("firstName");
            firstName.setStringValue("Joe");
            person.appendChild(firstName);
            var lastName = document.createElement("lastName");
            person.appendChild(lastName);
            lastName.setStringValue("Bloggs");
            requestScope.put("document", document);
       }]]>
       </xp:this.beforePageLoad>
      <xp:text escape="true" id="computedField1"
            value="${xpath:document:/person/firstName}">
       </xp:text>
      <xp:text escape="true" id="computedField2"
            value="${xpath:document:/person/lastName}">
       </xp:text>
</xp:view>
```

The Active Content Filtering Properties

Active Content Filtering (ACF) was introduced to XPages in Notes/Domino V8.5.1 as a means of protection from malicious content entered via requests or responses in an XPages application. The ACF properties are the following:

```
xsp.htmlfilter.acf.config
xsp.richtext.default.htmlfilter
xsp.richtext.default.htmlfilterin
```

Good summary information on the Active Content Filtering properties is included in the **xsp.properties** file, as shown in Listing 1.48.

Listing 1.48 xsp.properties Snippet for the xsp.htmlfilter.acf.config Property

```
# ########################################
# ACTIVE CONTENT FILTERING
# ########################################
#
# Defines which filter should be used by default for some controls
# (richtext)
# The valid values are:
#     - acf: use the acf library
#     - identity: no filtering is applied
#     - empty: the entire text is stripped out
#     - striptags: all the html tags are stripped out.
#                  Only the text pieces remain
#xsp.richtext.default.htmlfilter=acf
#xsp.richtext.default.htmlfilterin=
#
# Defines the acf library config file to use
# If empty, then it uses the default config file provided by the
library
# Else, it looks for a file locating in data/properties. For example:
# acf-config.xml
#xsp.htmlfilter.acf.config=<file name>
```

Dynamic scripting languages such as JavaScript can pose a threat if used maliciously by a hacker—for example, if some JavaScript code is embedded in the HTML content on a rich text control and then submitted as part of a request. Similarly, output fields can contain malicious content, such as where dynamically computed text contains code that attempts to perform privileged or illegal operations. To counteract such threats, each output control, such as Labels and Computed Fields, has had an `htmlFilter` property since version 8.5.1, and input controls such as the Edit Box and the Rich Text editor additionally have an `htmlFilterIn` property. These nominate a filtering engine that can check the field data for malicious content and take appropriate action.

The standard values applied to these properties are as follows:

- **acf.** Parses the HTML text and filters out the unsafe constructs. The filter used is based on a default configuration shipped with the XPages runtime. The default configuration can be overridden by specifying a custom **acf-config.xml** configuration file in your **Notes/Domino data/properties** directory.
- **striptags.** Removes all the tags using a regular expression: `replaceAll("\\<.*?>","")`

- **identity.** Does nothing but return the original string. This option is useful if you have the engine set to `acf` and you want to override this setting for one particular control.
- **empty.** Removes everything and returns an empty string.

The first two ACF settings in the **xsp.properties** file, `xsp.richtext.default.htmlfilter` and `xsp.richtext.default.htmlfilterin`, merely provide a means of applying the same filters on a more global basis than the control properties themselves can facilitate. That is, you can set any of the same four aforementioned values to the **xsp.properties** file within an NSF and thus have them apply to all input/output fields on all pages of the application (saving a lot of typing). Moreover, setting these property values in the **xsp.properties** file on a given server applies the nominated filters to all applications running on that server—and the same for the Notes client.

The third ACF setting, `xsp.htmlfilter.acf.config`, is a little more interesting. This property allows a custom configuration file to be applied as the ACF filter engine, for example:

```
xsp.htmlfilter.acf.config=acf-config.xml
```

On your Notes client or Domino server, in the `data/properties` directory, a file named `acf-config.xml.sample` is provided as an example containing some custom filtering rules. You can examine this file and perhaps add your own extended ACF rules. Of course, you must rename the file to `acf-config.xml` for the rules to take effect. And remember, this must be performed on each server in a given cluster because the configuration file is not part of the replication process. Listing 1.49 shows a snippet containing some sample filter rules.

Listing 1.49 Sample ACF Rules

```
<?xml version="1.0"?>
<config>

    <filter-chain>
        <filter name='base'
            class='com.ibm.trl.acf.impl.html.basefilter.BaseFilter'
            verbose-output='false' use-annotation='false' />
    </filter-chain>

    <filter-rule id='base'>
        <target scope=''>
            <!-- C14N rules -->
            <rule c14n='true' all='true' />

            <!-- Base rules -->
            <rule attribute='on' attribute-criterion='starts-with'
```

```
                action='remove-attribute-value' />
            <rule attribute='${' attribute-criterion='starts-with'
                action='remove-attribute-value' />
            <rule attribute='href' value='javascript:'
                value-criterion='contains'
                action='remove-attribute-value' />
            <rule attribute='style' action='remove-attribute-value' />

            <rule tag='script' action='remove-tag' />
            <rule tag='style' action='remove-tag' />
        </target>
    </filter-rule>
</config>
```

Most of the keywords in the listing are self-explanatory. For example:

```
<rule attribute='on' attribute-criterion='starts-with'
    action='remove-attribute-value' />
```

This rule means to remove attributes that start with the sequence of letters `on`. If the input contains any tag attributes, such as `onmouseover` or `onclick`, these are removed while still leaving the enclosing tag. If you want to strip out the complete tag, use a rule similar to the following:

```
<rule tag='script' action='remove-tag' />
```

This rule removes all the `script` tags.

One important point to remember with ACF filtering is that it is based on a blacklist approach. This means that everything is allowed and only code matching the specified patterns is removed. As new vulnerabilities are discovered, the blacklist must be updated.

ACF filtering can also be applied programmatically. The XPages Server Side JavaScript context global object provides two methods:

```
filterHTML(html:String, processor: String) : String
filterHTML(html:String) : String
```

The methods accept a string of markup that is then filtered using the specified processor or engine. The processed string is returned as the result. If no engine is specified, `acf` is used. A typical use case might be one in which you want to verify that the result of several input fields do not form a string with malicious content when concatenated—in other word, where the sum of various parts amounts to something malicious.

Finally, the ACF technology that XPages uses is a common component shared across other products in the IBM software portfolio, such as IBM WebSphere® sMash. In-depth information on ACF is now available in the development community for

WebSphere sMash, known as Project Zero. At the time of writing, searching the Internet for ACF yielded good information at the projectzero.org site, as follows: www.projectzero.org/sMash/1.0.x/docs/zero.devguide.doc/zero.acf/ActiveContentFiltering.html

This resource is certainly worth exploring if you want to deep dive on ACF customizations.

The Resource Servlet Properties

The XPages runtime supports a range of different resource providers. These are responsible for serving common resource content, such as CSS, Images, and JavaScript files, to a requesting browser or client. Such content is marked for inclusion in a browser-caching policy, to reduce the number of duplicate requests for any given global resource against a server. One available property enables you to optimize this process.

xsp.expires.global

The `xsp.expires.global` property is set to 10 days by default (as shown in Listing 1.50). This means that, for any given resource served by any of the XPages runtime resource providers, that resource is set to be included within the requesting browser or client cache for a maximum of 10 days.

Listing 1.50 xsp.properties Snippet for the xsp.expires.global Property

```
# Defines the default expiration duration for global resources
# When not defined, it is 10 days
#xsp.expires.global=10
```

If the defined period has passed, then for any requests issued against the server for that same resource, there will be a new download (refresh) of that resource within the requesting browser or client cache. Again, the expiry duration is set to 10 days into the future from that point in time.

This mechanism works in isolation to the range of cache expiry duration settings found within the Applications Properties editor, as shown in Figure 1.20.

These cache expiry duration values are written into the application's **database.properties** file. This is due to the dual use by the Classic Domino web engine and the XPages runtime of these values. You can see this in the following code fragment in Listing 1.51, taken from the **database.properties** file.

Listing 1.51 Sample database.properties Snippet

```
<item name='$CSSExpires'><text>365</text></item>
<item name='$ImageExpires'><text>365</text></item>
<item name='$JSExpires'><text>365</text></item>
<item name='$FileExpires'><text>365</text></item>
```

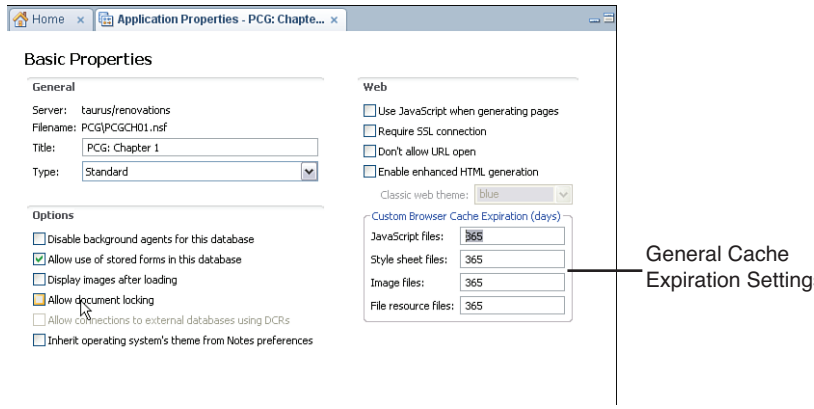


Figure 1.20 The general cache expiry duration settings in the Application Properties in Designer

Such isolated requests against the **XPages Resource Servlet** can typically be identified by a special URL alias, such as **/.ibmxspres/**, within the resource URL's of a rendered XPage. These URLs are issued against the global resource providers and not directly to the underlying Domino HTTP engine.

The following URLs demonstrate a standard resource request against the Domino server for an image resource versus a request for a global resource issued against the **XPages Resource Servlet** (resource provider) for CSS content:

http://taurus/domjava/xsp/theme/common/images/expanded.gif

versus

http://taurus/xsp/.ibmxspres/.mini/css/@Oa&@Ob&@Da&@Ib&@Ta&@Tb&@Tc.css

In this example, the first URL retrieves the image resource and caches this resource based on the standard caching expiry for image resources found within the Application Properties. The second URL is issued directly at the **XPage Resource Servlet**. This can be identified here by the inclusion of the **/.ibmxspres/** alias within the URL. This runtime servlet then provides the resource based on the **xsp.expires.global** property setting. If you have not specified a value for this property, the default expiry duration of 10 days is specified in the response headers to the requesting browser or Notes client.

The Repeating Control Properties

This category has a single option and controls an aspect of the behavior of the content-repeating controls. In the XPages runtime, the repeating controls are the View Panel, the Data Table, and the Repeat control, although any third-party implementation of a repeating control (in other words, anything that extends **UIData** or implements **FacesDataIterator**) should honor the behavior this property defines.

xsp.repeat.allowZeroRowsPerPage

This option was added in version 8.5.2. The behavior of the repeating controls is the same as in the previous releases, but now this option can be used to change the behavior for an individual application. Listing 1.52 shows the summary information provided in the **xsp.properties** file.

Listing 1.52 xsp.properties Snippet for the xsp.repeat.allowZeroRowsPerPage Property

```
# Defines the behavior in repeating controls when the rows property
# evaluates to 0. By default, in that situation 30 rows are displayed
# per page. With this option set to true no rows would be displayed.
# This may be useful when computing the number of rows to display,
# as there may be situations when no rows should be displayed
# but the control should still be rendered.
#xsp.repeat.allowZeroRowsPerPage = false
```

The repeating controls are the View Panel, the Data Table, and the default configuration of the Repeat control. When the Repeat control is configured with the property **repeatControls="true"**, it repeats its contents only once, at page load time. Hence, it does not engage in the repeating behavior controlled by this option and is not affected by changes to this option. The repeating controls can be bound to a list of items, and in the HTML page, the contents of the repeating control are redisplayed for each item, usually with subsequent contents displaying vertically under the previous items so that the contents appear in multiple rows. The list of items commonly consists of the contents of a Notes/Domino view, appearing as a list of documents.

Each of those repeating controls has a property `rows`, which controls how many rows or documents are initially visible in the repeating control. For example, you may have an application with 10,000 documents and a View Panel control displaying the All Documents view. The control does not attempt to display all 10,000 documents at once because the browser page would be too long and unwieldy, and it would take a long time for the page to load. Instead, by default, the control displays the first 30 documents. The user can use a Pager control `Next` link to move to the next 30 documents and to navigate through the rest of the documents. The `rows` property can be used to change the number of documents shown initially and in each subsequent page of documents. So if it is set to 10, the first 10 documents will be initially shown; clicking `Next` then would show the 11th to 20th documents.

It is possible to compute the `rows` property and also the `first` property, which controls the document index where the initial display starts. This allows use cases in which you can design a page to show any range of documents in a view. For example, you could show three documents starting at the fifth document. If you've designed a page to show a dynamic range of documents, sometimes the range might be empty. That is, if none of the documents is suitable for the criteria you're selecting, you might want to show zero documents.

By default, when the `rows` property is set to `0`, 30 documents are displayed. That might seem odd, but it is the behavior of the underlying JSF framework. The possibility is that existing XPages have been designed to depend on that behavior, so changing the behavior might cause problems. However, it also means that you need special configuration when attempting to display a range of documents. This `xsp.repeat.allowZeroRowsPerPage` option can be set to `true` in an application to change the behavior for that application so that when the `rows` property is `0`, no rows are displayed. It can also be set to `true` at a server-wide level, but be aware that you might need to retest the existing applications on that server, to verify that they do not depend on the old behavior of `0` displaying 30 documents.

The Partial Update Properties

This category contains one property relevant to AJAX requests issued against the XPages runtime.

xsp.partial.update.timeout

The XPages partial update feature enables an area of the web page to be submitted and updated with the response from the web server. Sometimes the browser does not receive the web server response; perhaps the web server is down, the network connection is slow, or the user has accidentally unplugged the network cable. Eventually, the browser determines that the request has timed out—that it is taking more time than is allowed—and displays an error dialog to the user. Listing 1.53 shows the relevant section of the **xsp.properties** file.

Listing 1.53 xsp.properties Snippet for the xsp.partial.update.timeout Property

```
# This allows a user to configure the partial update timeout in
➥Designer
# The default is 20 seconds
#xsp.partial.update.timeout=20
```

Since version 8.5.3, the dialog has this text:

```
An error occurred while updating some of the page. timeout exceeded
```

The user can click an OK button. In earlier releases, the dialog had this text:

```
Problem submitting an area of the page. timeout exceeded
Submit the entire page?
```

The user could click OK to submit the page or click Cancel to prevent the page submission. The dialog has been changed because people found the text confusing and weren't sure which button to click.

This option can be used to change the amount of time the browser will wait before displaying that dialog. The value is specified in seconds; it currently defaults to waiting 20 seconds when not specified.

This option has been available only since version 8.5.2. Also before 8.5.2, the time-out duration was 6 seconds, by default. In this case, people with slow network connections saw time-out problems when the server response was just taking time and was not actually unavailable, so the default was increased to 20 seconds.

The time-out can also be changed programmatically. It has been possible to change the time-out duration like this since release 8.5.0, so you can use this as a workaround for too-short time-out problems in versions 8.5.0 and 8.5.1. To programmatically change the timeout, in Client Side JavaScript, set the value of `XSP.submitLatency` to some number of milliseconds (not seconds), as in the example XPage control in Listing 1.54.

Listing 1.54 XSP Markup Example Showing Programmatically Setting the Partial Update Timeout

```
<xp:scriptBlock id="scriptBlock1">
    <xp:this.value><![CDATA[
XSP.addOnLoad(function(){
    // change submitLatency to 2 seconds (defaults to 6 or 20):
    XSP.submitLatency = 2 * 1000; // milliseconds
});
]]></xp:this.value>
</xp:scriptBlock>
```

Note that the option `xsp.ajax.renderwholetree` also affects partial update behavior, so you should study and understand it in relation to the partial refresh feature.

The Link Management Properties

Although it might not be immediately obvious, the fine art of managing the humble anchor link is sufficiently complex that it requires not one, but two dedicated **xsp.properties** settings. The sections that follow explain these settings.

xsp.default.link.target

This property can be assigned one of two values, namely `_self` or `_blank`. Anyone familiar with the standard HTML anchor link tag, `<a>`, will recognize these attribute values. The former instructs the browser to open a link in the current window, replacing current content; the latter means that the linked page is to be opened in a new tab. Applying either value in the **xsp.properties** file establishes the default link behavior for all applications on the platform. In this case, the platform is the Notes client exclusively. Listing 1.55 shows the relevant section of the **xsp.properties** file.

Listing 1.55 xsp.properties Snippet for the xsp.default.link.target Property

```
# ######################################
# DEFAULT LINK TARGET
# ######################################
# The Default Link Target when not specified directly on link
#xsp.default.link.target=
```

Why just the Notes client and not the web? The answer lies in the management of application data. The logic and behavior of any application often depend on data stored in application scope—that is, data shared by all instances of a given application. In a nutshell, if a single XPages application instance has several pages open at any one time in different tabs within the Notes client, each open page can be reliably associated with the application instance to which it belongs. This is because the XPages runtime has very granular control over the window-management features of the Notes platform and has built custom runtime logic to keep tabs on multiple windows. On the web, however, if a single XPages application instance has multiple pages open in different browser tabs, it is not possible to deterministically identify the owning application instance. As a result, applications that make dynamic use of application-scope data could become unstable in a multitabbed web application. Thus, the feature can be honored only in the Notes client. Moreover, the traditional user experience on the Notes client is for multitabbed or multiwindowed applications, and it is important that XPages support this native mode of execution when running on that platform.

So because the value specified in `xsp.default.link.target` is used as a default value for link behavior on the Notes client, what links does it apply to, when is it enforced, and how is it overridden?

In essence, the **xsp.properties** setting is applied to any standard link control located on an XPage if the link itself does not explicitly specify a target value. To cater to the use case in which a particular XPage contains many link controls, a **defaultLinkTarget** property can be set on the XPage itself to dictate link behavior for all links on the XPage, saving the application developer a lot of typing. If the required link behavior does not vary from one XPage to another, but is consistent across the application as a whole (as one would expect to be the norm), it is easier to simply set the overall link behavior in the **xsp.properties** file. When no value is set at any of these three levels (control, page, or application), a default value of `_self` is automatically applied and linked pages open in the original window or tab.

Another link also needs to be accounted for: the column link that can be optionally rendered inside a `<xp:viewPanel>` control. Here also, any underlying document link displayed for a row of the view can be opened either in a new tab or in the same pane when running in the Notes client. This is achieved using the same `target` property name and attribute values as the link control, but applied on the view control.

This extended link behavior was added in Notes/Domino V8.5.2. Table 1.4 displays a summary of all options.

Table 1.4 Link Management Property Summary

Container	Property	Values
<xp:link>	target	`_self`, `_blank` values determine whether the link opens in the same page or a new tab
<xp:viewPanel>	target	Uses same values to define link behavior for all columns in the view
<xp:view>	defaultLink-Target	Uses same values to define default behavior for all links on the page
xsp.properties	xsp.default.link.target	Uses same values to define default behavior for all links in the application

Bear in mind that the XPages link control renders as a standard HTML anchor link. In addition, standard HTML link attribute values are supported within XPages and simply passed through at runtime if specified in the XSP markup. Thus, a value of `_blank` can be applied to a link on the web and will result in a new tab being opened in the browser, but the application developer must be cognizant of the potential for application scope errors mentioned earlier and must also understand that neither the XPage nor **xsp.properties** default target setting will be applied to applications running on the web.

xsp.save.links

This property defines what format is used to save native Notes/Domino links in XPages. Listing 1.56 shows the default setting.

Listing 1.56 xsp.properties Snippet for the xsp.save.links Property

```
# ######################################
# SAVING LINKS IN DOMINO DOCUMENTS
# ######################################

# Defines how doc/db/view/app links should be saved in a Domino
Document
# Valid Values are:
#           - UseNotes: Saves your links with the notes protocol
#           - UseWeb:   Saves your links as web links
#xsp.save.links=UseNotes
```

To understand the semantics of this property, it is necessary to briefly discuss the formats used to store rich text in Notes/Domino applications, namely MIME and CD. Rich text content created within an XPages application on the Notes/Domino platform is always stored in MIME format. This is also true of documents created using applications running on the classic Domino web server. On the other hand, any rich text document fields created using the native Notes client are typically stored in a proprietary

Composite Data (CD) format. If your application needs to be supported on a mixed Notes/Domino platform—for example, if it runs using XPages on the web and also as a native Notes client application—**xsp.save.links** can help you avoid incompatibilities that can arise between these two formats.

A cursory examination of most any Notes/Domino application inevitably demonstrates heavy usage of links within the documents stored in an NSF. For example, a typical document in a Discussion or TeamRoom application instance, by virtue of being a collaborative application, will be littered with document, view, and application links. When working with such a document in XPages, if it was created or last updated in the native Notes client, it must go through a CD-to-MIME format conversion. Any such links are converted in the process from Notes links to Domino URLs. For example, a generic view link on Domino looks like this: `http://[server]/appname.nsf/viewName`

A view link on the Notes client typically is of this form:

```
notes://[server]/appname.nsf/viewName
```

You will immediately observe the differing protocols (`http` versus `notes`) between both types. The square bracket notation also indicates that the server name is optional. Before the advent of XPages, documents that went through the CD-to-MIME conversion process were expected to run *only* on the web—that is, on the Domino web server—and so were converted as relative instead of absolute URLs. This meant that, for applications residing on a server, the server name was omitted from the Domino URL. This was okay because the Domino web engine running on that server could resolve these locally. When XPages in the Notes client (XPiNC) was released in version 8.5.1, it was suddenly easy to take XPages applications offline by creating a local client replica of an XPages web application. Any Notes/Domino links opened in XPiNC could not be resolved, however, because the server name was not present in the converted URL and the client could not resolve the relative URL.

To be a good citizen on both the Notes client and the web, the XPages runtime attempts to massage native links opened in XPages applications so that the converted links resolve correctly, regardless of either their provenance or the runtime platform on which they are opened. However, a lot of the massaging can be avoided in the first place if links are saved in a common format that both the Notes client and the Domino web server can resolve. The `xsp.save.links` property thus gives the application developer the opportunity to choose a Notes/Domino link format that is most compatible with the mix of runtime platforms for which the application is to be deployed. Table 1.5 suggests appropriate values for different application configurations.

Table 1.5 Application Type by Notes/Domino Platform

Notes Client Application	Web Application	Appropriate Value(s)
Native	XPages	`UseNotes` (default)
XPages	Classic	`UseWeb`
XPages	XPages	`UseNotes` (default)

The implementation of this feature went through a number of iterations. In Notes/Domino 8.5.3 (or Notes/Domino 8.5.2 FixPack 2), the default link format used by XPages when saving links is as follows:

```
notes:///__replicaID/resourceUNID?Redirect&Name=ServerName
```

When the XPages runtime encounters such links on the web, the `notes` protocol is automatically replaced with the `http` protocol. This format is the best common denominator for links in a mixed-runtime environment.

The Control Library Properties

Generally, *extensibility* refers to the capability for XPages controls provided by third parties to be used in XPages applications so that applications are no longer limited to the default set of controls that comes with the XPages runtime (the Edit Box, Button, and View Panel controls, and so on).

Libraries of controls can be developed using the instructions on the web page "XPages Extensibility API Developers Guide." Existing libraries can be purchased from third parties or downloaded for free from open source sites on the Internet. The most prominent control library is the XPages Extension Library project, available on OpenNTF.org. This library has mostly been developed by people working for IBM, although it is provided under an Apache license and is not an IBM product offering.

xsp.library.depends

This option was new in version 8.5.2 as part of XPages Extensibility. Good summary information is included in the xsp.properties file, as shown in Listing 1.57.

Listing 1.57 xsp.properties Snippet for the xsp.library.depends Property

```
# A comma-separated and trimmed list of library IDs of
# XPages control libraries that the current application depends on.
# This option should only be set in the application xsp.properties file,
# not in the server Data/properties/xsp.properties file.
#xsp.library.depends=
```

You can install any libraries of controls that you want to use in your application in Domino Designer, as well as probably onto the Domino server you use for testing, using the instructions in the Extensibility guide. If your application is using XPages in the Notes client, you will find instructions on installing the library to Notes client installations where the application will be used.

After you have installed the library into Designer, when editing an XPage, the Controls palette contains all the extra controls provided by all the libraries installed in Designer. If you drag a library control onto an XPage in your application, you get a prompt dialog asking to enable the control library in this application, as shown in Figure 1.21.

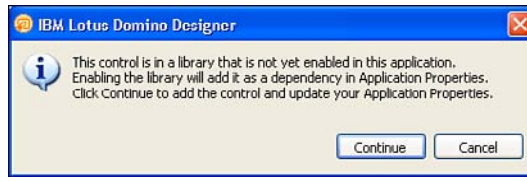


Figure 1.21 The Designer prompt to add a library dependency to an application

When you click Continue at that dialog, this `xsp.library.depends` option in the application's **xsp.properties** file is updated to include the ID of the library containing the control you have used. That configures the application to allow use of all controls in that library.

When you have used controls from multiple libraries in your application, the option value is a comma-separated list of the IDs of libraries that your application depends on. Listing 1.58 provides an example.

Listing 1.58 xsp.properties Snippet Showing Sample Dependency Declaration

```
xsp.library.depends=com.ibm.xsp.extlib.library,com.example.library
```

You can also set the option in the Application Properties in Designer. It is usually set there, or by adding controls from the palette, instead of directly editing the **xsp.properties** file. In the Advanced tab, in the XPages Libraries section, you can see a list of all the third-party libraries that are installed into this Designer instance, as shown in Figure 1.22. Check boxes indicate the libraries that this application depends on. You can add or remove dependencies on a particular application by checking or unchecking those check boxes.

If you uncheck a check box for a library but the application is using controls from that library, the application will fail to compile. The Problems view then will contain "Unknown tag" errors, complaining that it does not recognize the controls.

You should check boxes for only the libraries that your application depends on. When the application runs on a server, it validates the list of library dependencies against the list of libraries installed on the server. If any of the required libraries are not present on the server, the application will fail to display, with an error message similar to this:

```
Cannot find the library com.example.library, required by the applica-
tion TestApp.nsf.
```

The XPages runtime is actually implemented as a set of XPage libraries, but there is special handling for those libraries so that they do not need to be explicitly listed in the `xsp.library.depends` option. Those libraries are always depended upon so that attempts to use the Edit Box, Button, and other core runtime controls do not give an "Unknown tag" error.

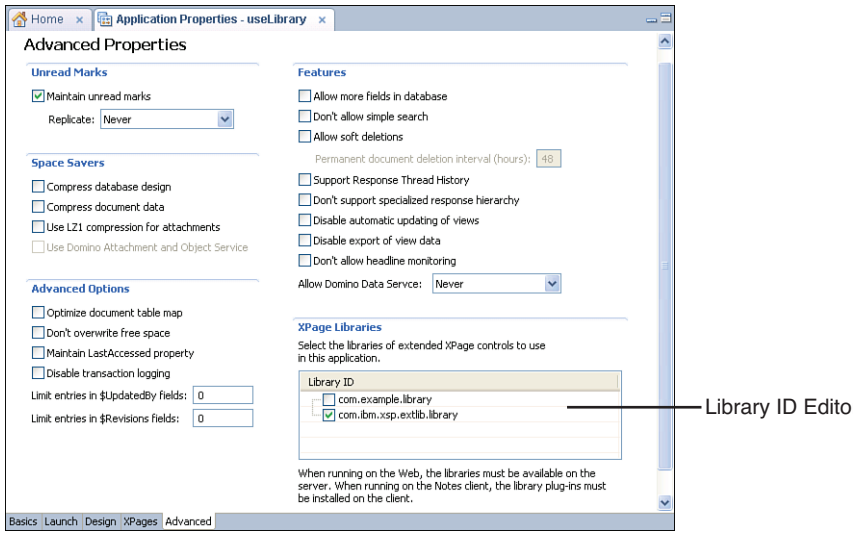


Figure 1.22 The Application Properties, Advanced tab, showing multiple libraries

The Composite Data Properties

XPages Theme files can be created in an application to change the appearance of every instance of a control in the application. Generally, there is a default behavior for each property of a control—for example, an Edit Box may have a default size of 20 characters wide. Theme files support overriding the default property value, for example—for example, setting Edit Boxes to have size of 25 and to have a style indicating a thick border instead of the default border width. The theme values for the control properties are applied to all Edit Boxes in the application, except for those for which the XPage has specifically set a value. It is also possible to apply a theme property value to a smaller group of controls using the control `themeId` property. For example, on all Edit Boxes that are designed to accept a user's age, you might set a `themeId` such as **ageEditBox**. Then it would be possible to use a theme file to configure that all **ageEditBox** controls must have a size of three characters wide.

You can create Custom Controls in an application that contain some snippet of XPage content. The Custom Control then becomes available in the controls palette and can be used in different XPages or used multiple times in the same XPage. Every place where a Custom Control is used is an individual Custom Control instance and appears in the XPage source tab as an XML tag. When editing a Custom Control's content, in the Properties view is a Property Definition tab where you can define properties of the Custom Control. You then can set that property to different values on each Custom Control instance tag. Within the Custom Control content, it is possible to refer to the property values passed in from the outer Custom Control tag instance. A Custom Control property named `header` can be referred to in a computed expression as `compositeData.header`, which contains the property value set on the current Custom Control tag (or which is empty if the property has not been set).

In version 8.5.0, it was not possible to set Custom Control property values using theme files. In the Custom Control contents, if you selected the root tag, set the `themeId` to `customHeaderArea`, and then configured a theme to set the `header` value of all `customHeaderArea` controls, the value set in the theme file would not be available as `compositeData.header` in computed expressions.

Since version 8.5.1, in applications where this option is not set or is set to `true`, it is now possible to set Custom Control property values using a theme file. So in the previous example, if your theme is configured to set all `customHeaderArea` controls so the `header` property value is `Welcome`, then computed expressions referring to `compositeData.header` will resolve to the value `Welcome` unless the header property has been explicitly set to some other value in this particular Custom Control tag instance.

xsp.theme.preventCompositeDataStyles

This option relates to a change in behavior for Custom Controls in version 8.5.1. The option can be set to `true` to revert an application to the old version 8.5.0 behavior, although it usually is better to update the application to work with the new behavior. Refer to the summary information provided in **xsp.properties**, as shown in Listing 1.59.

Listing 1.59 xsp.properties Snippet for the xsp.theme.preventCompositeDataStyles Property

```
# In 8.5.0 style and styleClass would be set as a base property
# but in 8.5.1 they are set as compositeData properties
# and referred to in the inner custom control
# as compositeData.style/styleClass.
# Set to true to revert to 8.5 behavior, default is false.
#xsp.theme.preventCompositeDataStyles=false
```

The behavior in question relates to how theme property values are applied to Custom Control properties named `style` and `styleClass`. The default behavior changed in version 8.5.1.

The only disadvantage to this, and the reason you might want to revert the behavior, is that, previously, a quirk in the implementation meant that even where you had not declared Custom Control properties named `style` and `styleClass`, it was possible to set values for those nonexistent properties in the theme file. The property values then actually appeared in the output HTML on a `DIV` tag wrapped around the Custom Control content.

If you have already implemented an application that is using the old behavior to set a `style` or a `styleClass` onto a Custom Control in a theme file, it may be useful as a temporary workaround to set this option to `true` to revert to the old behavior. It will revert the behavior for only the `style` and `styleClass` properties. Other Custom Control property values will continue to be settable in theme files in the new manner; however, any `style` or `styleClass` property set in a theme file will appear in the

HTML DIV tag but will not be available when referenced by `compositeData.style` or `compositeData.styleClass` expressions.

Instead of using this option to revert the behavior, it might be best to update your Custom Controls to support the desired style properties through normal Custom Control property definitions. To achieve this, in the Property Definition tab, define new properties named `style` and `styleClass`. Then in the Custom Control content, select the root tag acting as a container for the Custom Control contents and compute `style` and `styleClass` properties using the expressions `compositeData.style` and `compositeData.styleClass`. You then can set the `style` and `styleClass` properties using the theme file and have them appear in a DIV tag around the Custom Control content. The Source tab of the Custom Control content will look like that shown in Listing 1.60.

Listing 1.60 XSP Markup Snippet for a Custom Control with Custom Style Properties

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="www.ibm.com/xsp/core"
  style="#{javascript:compositeData.style}"
  styleClass="#{javascript:compositeData.styleClass}">
    custom control content
</xp:view>
```

Other Ways of Applying xsp.properties Settings

At this point, the chapter has explained all the properties and you have seen that they can be applied at various scopes, such as at an application-wide level and a server-wide level. In truth, it gets more sophisticated than that: Many properties can be applied on an individual XPage basis, for the lifetime of a particular request, or even by a completely different mechanism, as with an XPages theme. These operations might not be common, but they are worth exploring.

Viewroot Properties

Making an **xsp.properties** setting apply to just a single XPage is easy. Open Domino Designer and create a new XPage. In the Outline navigator, select the XPage element itself (that is, the root node of the tree of components that comprise an XPage) and then select the All Properties panel for the page. Under the data category, you will see a `properties` entry. You can select this element and add new entries. By clicking the `properties` entry, you can see from both the panel editor and the source panel that Designer wants you to add a new name/value pair. In this case, you can add any of the **xsp.properties** shown in Table 1.1 and apply an appropriate value. To take a simple and recent example, add `xsp.default.link.target` as a property name and assign `_blank` as the property value. Then drop a link control on to the page from the Core Controls palette and assign `www.masteringxpages.com` as the `value` attribute. Your XPage markup should correspond to that shown in Listing 1.61.

Listing 1.61 Applying xsp.properties to a Single XPage

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="www.ibm.com/xsp/core"
       xmlns:xe="www.ibm.com/xsp/coreex">
      <xp:this.properties>
            <xp:parameter name="xsp.default.link.target"
                        value="_blank">
                  </xp:parameter>
      </xp:this.properties>
      <xp:link escape="true" text="Link" id="link1"
            value="www.masteringxpages.com">
      </xp:link>
</xp:view>
```

Now preview this XPage in the Notes client (remember that this is a client-only feature) and click the link when the XPage is rendered. The target web page is loaded in its own tab window in the Notes client, as you might expect when `_blank` is the property value. Now close the new tab and the previewed XPage, and go back to Domino Designer. Change the `xsp.default.link.target` property name value to `_self` and preview the page in the Notes client again. Observe that, this time, the target web page is loaded inside the previewed page. That is, it replaces the current window content and does *not* open a new tab window for the target web page.

Thus, the `xsp.default.link.target` property has been applied to just a single XPage. The term *viewroot,* by the way, can be easily explained. In JSF parlance, pages are known as *views*, not to be confused with the view design element in the Notes/Domino world. The XPage itself is always the root node of the tree of controls and other components that comprise an XPage. So the term *viewroot* is synonymous with the base XPage element itself.

Request Properties

Taking this notion a step further, you can simply set an **xsp.properties** parameter for a particular request. You do this by using the `com.ibm.xsp.context.RequestParameters` object obtained from the `facesContext` global object. Simply use the `setProperty` method on this object to set an **xsp.property**. (Note that not all **xsp.properties** can be applied using this mechanism—see the last section of this chapter.) To retrieve the value of any given property, you can call the `context.getProperty(String propertyName)` method. This method returns a property value based on an order of precedence. First, it checks the `RequestParameters` object for the given property, if not defined in the request (through prior use of the `RequestParameters` object). It then checks the `viewRoot` of the actual XPage. If the given property is not defined at this level, it checks the session, or application properties. Note that this is equal to checking using the `context.getSessionProperty(String propertyName)` method. If the given property is

not defined within the session or application properties, a check is done within the currently running theme. Thereafter, a final check of the system properties is done.

Remember that any property set on the `RequestParameters` object takes absolute precedence over any other defined setting for that property. This means that any Application Properties property value is ignored for a property if set in the `RequestParameters` object for a request.

A simple but illustrative example of this arises in changing the new version 8.5.3 Resource Aggregation feature setting on a per-request basis. (Note that this example will not work in earlier releases of Notes/Domino.) Study the **requestPropertySetter** XPage within the PCGCH01.nsf application to understand how this is done. For your convenience, Listing 1.62 details the XSP markup for this XPage.

Listing 1.62 XSP Markup Used to Set the xsp.resources.aggregate Property on a Per-Request Basis

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="www.ibm.com/xsp/core" style="margin:30px">
  <xp:text escape="false">
    <xp:this.value>
      <![CDATA[#{javascript:"Application Property: " +
        facesContext.getApplication().
          getApplicationProperty("xsp.resources.aggregate", null);
      }]]>
    </xp:this.value>
  </xp:text>
  <xp:br></xp:br>
  <xp:br></xp:br>
  <xp:text escape="false">
    <xp:this.value>
      <![CDATA[#{javascript:"Request Property: " +
        context.getProperty("xsp.resources.aggregate", null);
      }]]>
    </xp:this.value>
  </xp:text>
  <xp:br></xp:br>
  <xp:br></xp:br>
  <xp:button value="Aggregate" id="button2">
    <xp:eventHandler event="onclick" submit="true"
      refreshMode="complete">
      <xp:this.action>
        <![CDATA[#{javascript:
          facesContext.getRequestParameters().
            setProperty("xsp.resources.aggregate", "true");
        }]]>
```

```
      </xp:this.action>
    </xp:eventHandler>
  </xp:button>
  <xp:button value="Do Not Aggregate" id="button3">
    <xp:eventHandler event="onclick" submit="true"
      refreshMode="complete">
      <xp:this.action>
        <![CDATA[#{javascript:
          facesContext.getRequestParameters().
            setProperty("xsp.resources.aggregate", "false");
        }]]>
      </xp:this.action>
    </xp:eventHandler>
  </xp:button>
</xp:view>
```

The PCGCH01.nsf application has its `xsp.resources.aggregate` property already set to `true` using the Application Properties editor in Designer. But in this example, a user can click either the Aggregate or Do Not Aggregate buttons, and the XPages Runtime will process the request and toggle resource aggregation handling. This is best seen by viewing the source of the HTML markup where the CSS and JavaScript resources are either aggregated or not, based on each request.

Applying Properties Using a Theme

If you like to work with XPages themes, you'll be glad to know that you can also apply the **xsp.properties** parameters using a theme. This can be useful in providing customized settings of **xsp.properties** based on the currently selected theme. Each theme within an application provides different settings for the same **xsp.property**. For example, consider the earlier example of setting the **xsp.default.link.target** property. By alternatively using a theme file, you can achieve this same behavior for every XPage within a given application—but you also gain the benefit of decoupling the logic to set the **xsp.property** from within the XPage itself to being discretely done inside the theme file. Listing 1.63 shows the fragment of theme code required to achieve this.

Listing 1.63 Fragment of Theme Code to Set the xsp.default.link.target Property

```
<?xml version="1.0" encoding="UTF-8"?>
<theme>
    <property>
        <name>xsp.default.link.target</name>
        <value>_blank</value>
    </property>
</theme>
```

What Works Where?

Even though you can use these mechanisms to attempt to apply any of the **xsp.properties**, not all properties are applicable in all circumstances. For example, you cannot apply a new random server persistence folder through the `xsp.persistence.dir.xspupload` property via a single request or XPage property. This is a static server property that can be applied only via the **xsp.properties** file that resides on the server itself. Your own intuition will be a good guide as to whether a property is applicable via a particular mechanism, and it is easy to quickly test it out, as shown here with the viewroot and request properties examples.

Conclusion

In this chapter, you explored *every* **xsp.properties** setting and saw how they can be used to adapt XPages runtime behavior to suit particular use cases. Hopefully, you have learned about some properties that you can apply to meet requirements you have encountered in the field so that you can produce highly tuned, well-adapted applications—and, of course, happier XPages customers.

This page intentionally left blank

CHAPTER 2

Working with Notes/Domino Configuration Files

In Chapter 1, "Working with XSP Properties," you saw how the **xsp.properties** file served as a configuration file for the XPages runtime. In an analogous manner, both the Notes client and the Domino server have a configuration file that sets parameters for various elements of core behavior: **notes.ini**.

Whereas the **xsp.properties** file is specific to XPages, **notes.ini** has many masters to serve; it is a common resource that all components in the Notes/Domino core share. As a consequence, the **notes.ini** files in your particular Notes/Domino installations might contain hundreds of diverse settings. For instance, a current check of the **notes.ini** file for my own Notes client shows that it has 198 entries, while my Domino server **notes.ini** contains 112 entries. The good news is that only a small fraction of these entries are directly relevant to XPages.

INI Variables You Should Know About

The **notes.ini** file is located in the root installation folder of the client and server. Listing 2.1 shows the first 20 lines of a sample Domino server **notes.ini** file, which includes all the entries that relate to XPages (directly or indirectly) as of Notes/Domino 8.5.3. These entries are emphasized using a shaded background.

Listing 2.1 Sample Snippet from a Domino Server notes.ini Configuration File

```
[Notes]
NotesProgram=C:\Domino
Directory=C:\Domino\data
JavaEnableDebug=1
JavaDebugOptions=transport=dt_socket,server=y,suspend=n,address=8000
KitType=2
InstallType=4
HTTPJVMMaxHeapSize=256M
HTTPJVMMaxHeapSizeSet=1
JavaMaxHeapSize=64M
JavaMinHeapSize=16M
PartitionNumber=1
XPagesPreload=1
XPagesPreloadDB=teamdisc.nsf/ByAuthor.xsp
FaultRecovery_Build=Build V853_06152011NP
Timezone=0
OSGI_HTTP_DYNAMIC_BUNDLES=update/extlibupdate.nsf,sbtupdate.nsf
```

```
NSF_QUOTA_METHOD=2
JavaUserClasses=D:\jdbcutils\jdbcdrivers.jar;D:\java\customlibs
ServerName=lehenaghmore/XYZ
```

Table 2.1 describes these entries.

Table 2.1 Sample Domino Server notes.ini Settings

INI Variable	Description
HTTPJVMMaxHeapSize	Sets the maximum heap size to be used by the HTTP task's JVM. Defaults to 256MB on 32-bit systems and 1GB on 64-bit systems. All heap size variables must be accompanied with a unit of size suffix (K, M, G, and so on).
HTTPJVMMaxHeap SizeSet	A flag the HTTP task uses to determine whether the **HTTPJVMMaxHeapSize** variable has been previously set programmatically.
JavaMaxHeapSize	Specifies the maximum heap size for JVMs running *outside* the HTTP task. Defaults to 256MB on all systems.
JavaMinHeapSize	Specifies the minimum heap size for *all* JVMs. Defaults to 48MB on all systems.
JavaEnableDebug	Notifies the Domino server that it should start the JVM in debug mode. This variable, along with all the remaining variables in this table, are not enabled in **notes.ini** by default.
JavaDebugOptions	Provides a comma-separated list of arguments to the JVM.
JavaUserClasses	Specifies a semicolon-separated list of one or more path locations, JAR files, or ZIP files containing Java classes that are then included in an extended Java class path.
JavaOptionsFile	Provides a path location to a file that contains arguments passed to the JVM on startup.
JavaVerbose	Turns on verbose logging within the JVM.
JavaTraceFile	Provides a path location to the JVM where the verbose logging messages will be stored. This variable, along with `JavaOptionsFile` and `JavaVerbose`, are discussed in more detail in Chapter 6, "Server-Side Debugging Techniques."
OSGI_HTTP_DYNAMIC_ BUNDLES	Specifies the location(s) of update site databases that contain OSGi bundles that are to be installed dynamically by the HTTP process.
XPagesPreload	Applying a value of `1` instructs the Notes/Domino core to preload the XPages Java runtime classes in memory at client and/or server startup.
XPagesPreloadDB	Takes a comma-separated list of XPages applications to preload in memory at client and/or server startup.

Table 2.1 describes the various **notes.ini** variables that can be added to debug XPages applications, extend their capabilities, and optimize their performance. Unless otherwise stated, it can be assumed that any given INI variable under discussion applies to the Domino server rather than the Notes client. In addition, any INI variable can be added or edited in one of three ways:

- By directly editing the **notes.ini** file using a text editor.
- On the server, using a Domino console command of this form:
  `set conf varname=value`
- On a server using the Name & Address Book (NAB) configurations parameters document. This approach is particularly useful in server clusters because the configuration replicates across all servers. As a result, the administrator doesn't need to manually configure each server.

On the Domino server, after a **notes.ini** variable is set, it is typically necessary to restart the HTTP task for the server to read and use the variable. In some cases, it is necessary to restart the server itself. Tasks on the Domino server that have Java programs embedded within them typically initialize and run a separate instance of the server's JVM per task. For instance, the Agent Manager, the HTTP task, and others all run unique instances of the server's JVM. It is also worth pointing out that, in many cases, **notes.ini** settings apply across all server tasks and are not necessarily tied to the HTTP task. Variables that apply only to the HTTP task are prefixed with *HTTP* in the variable name. If a variable applies to more than the HTTP task, users must use caution when modifying those variables; the consequences of the changes made will reach beyond the XPages runtime and, indeed, the HTTP task.

Depending on the task, the requirements on the JVM instance differ. As a result, various INI variables have been added to **notes.ini** to make the different JVM instances as configurable as possible. This enables the Domino server administrator to fine-tune the memory consumption of the Domino server while simultaneously optimizing the performance delivered to the end user.

Because the XPages runtime is written almost entirely as a Java program, the HTTP process must have a larger maximum Java heap size than was historically necessary for most Domino servers. That is not to say that XPages applications consume more memory than "traditional" Notes applications; however, resources now must be allocated to the Java process where traditionally they were not required.

Administrators have always been able to apply heap size minimum and maximum limits across all Java processes running on the Domino server via two INI variables, `JavaMaxHeapSize` and `JavaMinHeapSize`. Both are discussed later. With the release of Domino 8.5.2, two new variables have been added that increase the flexibility of these limits: `HTTPJVMMaxHeapSize` and `HTTPJVMMaxHeapSizeSet`. Before delving into the various heap *size* variables that are available to Domino's JVM, it is worth explaining what the Java heap is and how it affects performance.

The Java Heap

The JVM's heap is responsible for storing all objects that a Java program creates. The JVM creates a Java object when the constructor for an object is called within a running program. The JVM stores the object (on its heap) until all references to the object have been released. Only when all references have been released does the JVM allow garbage collection to occur on the object. The term *garbage collection* describes the process by which Java deletes previously created objects and makes the memory previously allocated to those objects available for other objects to use.

The size of the Java heap determines the frequency at which garbage collection needs to be run. In larger applications, it is not uncommon to set the size of `-Xms` (minimum heap size) and `-Xmx` (maximum heap size) to be the same, to give a fixed heap size. This helps to improve overall performance, particularly boot performance (performance when the application is bootstrapping/starting on the server).

One of the main memory goals of a JVM is to keep its memory footprint within the constraints provided via its `vm` arguments (that is, within the minimum and maximum heap limits). As a result, when a JVM notices that the amount of memory it is using is nearing the amount of memory allocated to it, the JVM runs garbage collection in an attempt to keep its memory footprint below the amount currently allocated. If the minimum heap size is a small number, garbage collection must run frequently to keep the memory footprint as small as possible. Each time the amount of memory the JVM uses nears the amount of memory currently allocated to the JVM, garbage collection is run. Garbage collection algorithms have been optimized significantly with each release of Java, but garbage collection doesn't come for free. A performance impact occurs each time the garbage collector must be executed. Furthermore, when garbage collection is running, it takes priority over all other processes running within the JVM. Figure 2.1 illustrates sample memory consumption of the HTTP task's JVM during startup when the minimum heap size is set to 16MB. The same battery of tests was run against the XPages runtime in all the following graphs. You can see that, as minimum heap size increases, the amount of time spent performing garbage collection decreases.

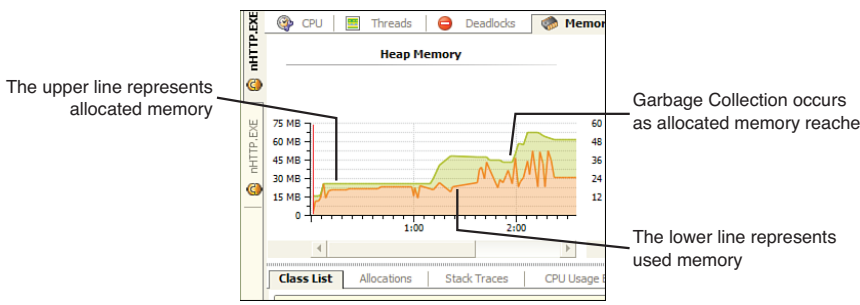


Figure 2.1 Memory consumption by HTTP JVM when JavaMinHeapSize is set to 16MB

As you can see in Figure 2.1, each time the amount of memory the JVM uses nears the allocated memory threshold, garbage collection is run. The graph shows that there will be an impact on performance during startup due to the low minimum heap size. When the JVM determines that, after garbage collection, the amount of allocated memory is not enough to create a new object, the JVM allocates more memory for itself, if possible.

Figure 2.2 illustrates how much time the JVM spends performing garbage collection during the same time period in Figure 2.1.

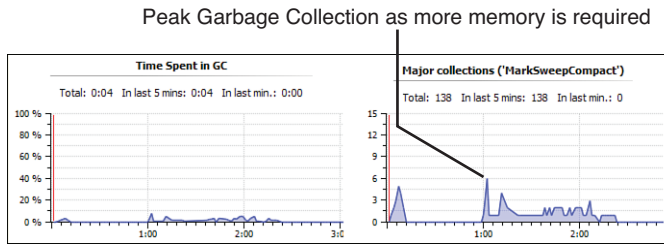


Figure 2.2 Time spent performing garbage collection when JavaMinHeapSize is set to 16MB

From looking at Figure 2.1 and 2.2, you can easily see that the JVM performs garbage collection approximately two minutes into the scenario. Next, you can see from Figure 2.3 and 2.4 that, as the minimum heap size is increased, the amount of time spent in garbage collection decreases.

Figure 2.3 illustrates sample memory consumption for the same scenario, but this time with the minimum stack size set to 48MB.

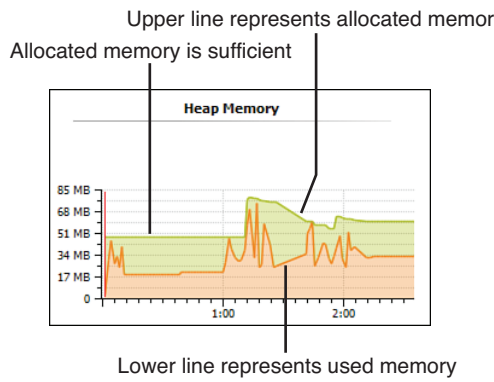


Figure 2.3 Memory consumption by HTTP JVM when JavaMinHeapSize is set to 48MB

Finally, Figure 2.4 illustrates the amount of time spent in garbage collection during the same scenario.

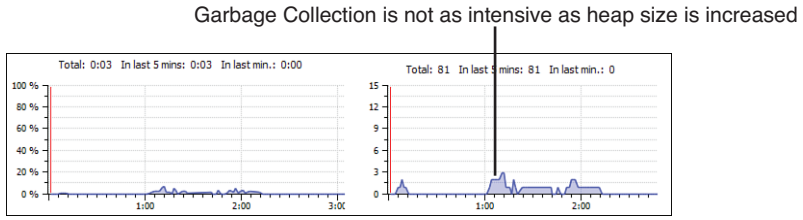


Figure 2.4 Time spent performing garbage collection when JavaMinHeapSize is set to 48MB

Although it is difficult to give an exact figure for optimal JVM performance, if the server administrator/developer knows that the amount of memory the JVM requires is likely to regularly exceed the minimum allocated to the JVM, it is advisable to increase the minimum heap size, if possible. This results in less time spent performing garbage collection and more time being allocated to processes that perform tasks that the end user can see.

We have discussed the reasoning behind setting `JavaMinHeapSize` to a reasonable size, but we have not mentioned the impact of setting the opposing settings `HTTPJVMMaxHeapSize` and `JavaMaxHeapSize`. Both of these settings are responsible for setting the maximum heap size of the JVMs of the HTTP task and all other tasks, respectively. The maximum heap size, as the name suggests, is the maximum size that the JVM's heap can expand to. If after garbage collection the JVM calculates that it does not have enough memory to create an object *without exceeding* the maximum heap size, it throws an `OutOfMemoryException`. If XPages developer or server administrators notice that the Domino server is throwing `OutOfMemoryExceptions` regularly, the `HTTPJVMMaxHeapSize` variable should be the first area for investigation.

HTTPJVMMaxHeapSize Variable

This variable sets the maximum Java heap size to be used by the Domino server's HTTP task. This INI variable passes its value through to the HTTP task's JVM. This variable correlates directly with Java's `-Xmx vm` argument. The value of this INI variable gets passed to the JVM as the value of the `-Xmx vm` argument. This setting was introduced in Domino 8.5.2 to allow administrators to set (in most cases) a higher heap limit for the JVM running within the HTTP task, and thus a higher heap limit for the XPages runtime.

In the past, it was typical to recommend setting a 64MB heap limit for all JVMs running on the Domino server (running on a 32-bit platform). With the introduction of XPages, it was quickly acknowledged that, for servers running several XPage applications simultaneously with many concurrent users, allocating more memory to the JVM serving the XPage applications was necessary. The initial recommendation prior to version 8.5.2 was to increase the `JavaMaxHeapSize` setting to accommodate the need for increased memory. However, increasing this setting for servers on 32-bit systems could cause them to run out of memory, as all JVMs (not just the HTTP task's JVM) were running

with this higher limit. A new solution was required, and two new INI variables resulted (HTTPJVMMaxHeapSize and HTTPJVMMaxHeapSizeSet).

For servers that have intensive XPage needs, it is now normal to recommend setting a maximum HTTP JVM heap size of 256MB (for 32-bit systems). A 1GB JVM maximum heap size is recommended for 64-bit systems. The task of identifying how much memory is *enough* memory to allocate as the maximum heap size is a delicate balancing act. If this number is too small, the performance of your XPages runtime will be negatively affected. Conversely, assigning a number that is too high adversely affects the performance of the entire system. Developers and system administrators should be wary of setting this value to an unreasonably high value for the system in question. If this variable is set too high, other applications running on the system will likely slow down and performance as a whole for the system will degrade. Users should be vigilant when changing this variable on the Notes client because it has a direct impact on the application's memory footprint.

Sample usage:

```
HTTPJVMMaxHeapSize=256M
```

HTTPJVMMaxHeapSizeSet Variable

This variable notifies the HTTP task that the HTTPJVMMaxHeapSize setting has been specifically set and should not be programmatically overwritten. The HTTP task checks for the HTTPJVMMaxHeapSize variable each time the HTTP task starts. If the variable is not present, the HTTP task queries notes.ini for the JavaMaxHeapSize variable. If this variable is present, it assigns the value of JavaMaxHeapSize to HTTPJVMMaxHeapSize and sets HTTPJVMMaxHeapSizeSet to 1. If JavaMaxHeapSize is not present the HTTP task sets the value of HTTPJVMMaxHeapSize to 64MB for 32-bit systems and to 1GB for 64-bit systems, and sets HTTPJVMMaxHeapSizeSet to 1.

If HTTPJVMMaxHeapSize *is set to less* than 256MB on a 64-bit system, it is automatically reset to 256MB on HTTP startup and HTTPJVMMaxHeapSizeSet is set to 1.

Users should be vigilant when changing this variable on the Notes client because it has a direct impact on the memory footprint of the Notes client.

JavaMaxHeapSize Variable

Domino tasks that run Java programs require a JVM instance to execute the Java program. Until release 8.5.2, the JavaMaxHeapSize variable was responsible for controlling the maximum heap size for all JVMs running on the server. As of release 8.5.2, the HTTPJVMMaxHeapSize variable has responsibility for controlling the size of the heap of the HTTP task's JVM. The JavaMaxHeapSize variable controls the maximum heap size of all other JVMs running on the server. The JavaMaxHeapSize variable is responsible for passing the -Xmx vm argument to all server JVMs (apart from HTTP's JVM).

Because the JavaMaxHeapSize setting affects potentially multiple JVM instances and multiple tasks, the server must be restarted after changing the variable's value. Users

should be vigilant when changing this variable on the Notes client because it has a direct impact on the memory footprint of the Notes client.

Sample usage:

```
JavaMaxHeapSize=256M
```

JavaMinHeapSize Variable

This variable sets the minimum heap size for all JVMs running within the Domino server (and Notes client). The minimum heap size is the amount of memory initially set aside for the JVM. This variable is responsible for passing its value through to the JVM as the `-Xms vm` argument.

Users should think carefully before increasing this variable on the Notes client because it impacts the memory footprint of the client.

Sample usage:

```
JavaMinHeapSize=16M
```

JavaEnableDebug Variable

Chapter 6 discusses this variable in much greater detail. This variable signals to the Domino server that its JVM should be started in debug mode. Enabling this variable is the first step required when configuring the Domino server for remote debugging. In the following example the variable is set to a value of `1`. This variable's value acts like a Boolean flag, with `1` being true (or on) and `0` being false (or off).

Sample usage:

```
JavaEnableDebug=1
```

JavaDebugOptions Variable

Chapter 6 discusses this variable in much greater detail. This variable is used to pass debugger information, such as debug port numbers and debug transport protocol names, to the server's JVMs.

Sample usage:

```
JavaDebugOptions=transport=dt_socket, server=y,suspend=n,address=8000
```

JavaUserClasses Variable

Setting the **notes.ini** parameter `JavaUserClasses` allows for a class to be loaded via the JVM system loader so that classes (and jars) can be shared across JVM instances. This variable is discussed in the section, "Enabling Extended Java Code with the java.policy File," later in this chapter.

This section touched on a few of the more commonly used JVM `vm` arguments. However, literally dozens of arguments to the JVM are available. You can find a full list here: www.tinyurl.com/JVMLaunchOptions.

You likely will never need to use many of them, but having a reference to these arguments is worthwhile because some will prove useful in the long term.

OSGI_HTTP_DYNAMIC_BUNDLES Variable

The XPages APIs have been available since Domino 8.5.2. Through the use of the XPages extension APIs, customers can create their own extensions of the XPages runtime. Customers can create their own XPages artifacts, such as (but not limited to) XPage controls, converters, validators, and data sources.

Typically, these artifacts are packaged within OSGi bundles (plug-ins). The artifact creator distributes the OSGi bundles. Under normal circumstances, a server administrator is expected to install the OSGi bundles (via copy and paste) into a folder location within the Domino server.

Feedback from administrators and developers alike indicated that this method of bundle installation is not optimal, mainly because organizations are hesitant to place unsigned JAR files into their Domino server.

Domino 8.5.3 introduced the `OSGI_HTTP_DYNAMIC_BUNDLES` variable. This variable enables the bundle provider to place OSGi bundles within an update site database that can be signed and provided to the server administrator. The plug-in signer must be among the server's list of individuals who can run unrestricted methods and operations. Those familiar with Domino server administration will understand that granting this level of authority to an individual or group should not be done without first ensuring that the signer is completely trusted. Allowing individuals or groups to run unrestricted in this manner implies that those listed have access to run agents or XPages that can access the underlying OS APIs.

The HTTP task automatically provisions (installs) the OSGi bundles from the database into the server, without the need for physically copying the bundles into the server (note that the update site database in question must be based on the version 8.5.3, or newer, update site template). The bundles at all times remain within the database, and the HTTP task is responsible for dynamically adding the bundles to the list of runtime OSGi bundles as the task is starting. As with any databases/code received from third parties, administrators should ensure that the update site databases are not malicious in nature before installing them on a production server. This model is recommended for use where an update site needs to be deployed across multiple servers. If OSGi bundles need to be installed on multiple servers in a cluster, administrators are advised to create an update site database (based on the 8.5.3 update site template). Administrators should next upload their OSGi bundles to the update site database. After the bundles are uploaded, a replica of the update site database should be created on each server where the bundles are to be installed. Finally, in the server's Name and Address Book, a Configuration/Parameters document must be created with the `OSGI_HTTP_DYNAMIC_BUNDLES` INI variable set to the location of the update site database. Using this model ensures that the bundles are installed on each server where the update site database is replicated to. After the bundles are deployed, the HTTP task on each server must be restarted.

Figure 2.5 shows the update site database with an update site imported into the database. As you can see from the figure, multiple toolbar buttons within the database support the easy import of existing update sites.

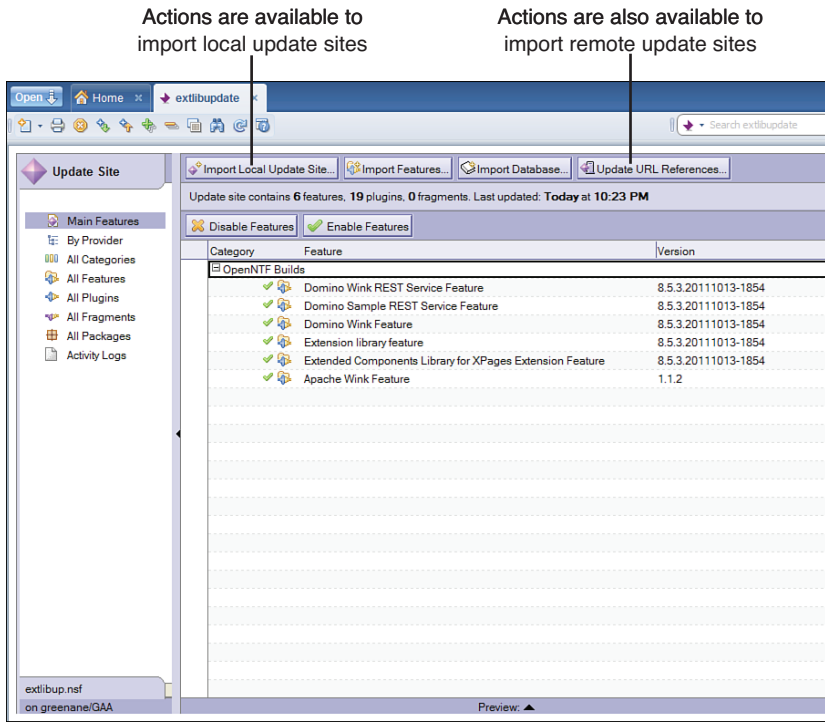


Figure 2.5 Notes 8.5.3 Eclipse update site database

Sample usage:

```
OSGI_HTTP_DYNAMIC_BUNDLES=update/extlibbundles.nsf,sbtbundles.nsf
```

XPagesPreload Variable

This feature, along with its `XPagesPreloadDB` sidekick, was introduced in Notes/Domino 8.5.3 to boost the *initial* startup time of XPages applications. When set to `1`, the `XPagesPreload` variable causes the Notes/Domino Java class loader to load the XPages Java runtime classes when the client or server is starting up. Two distinct groups of classes are loaded:

- Java classes from the XPages runtime plug-ins (`com.ibm.xsp.core` and so on)
- The Java classes referenced in the `*-faces-config.xml` files in the core plug-ins (`core-faces-config.xml, extsn-faces-config.xml`, and so on)

The first group is loaded from a fixed list of runtime classes (465 objects total in version 8.5.3), not only from the core `com.ibm.xsp.*` plug-ins, but also from common utility classes, JavaScript wrapper classes, and underlying JSF runtime classes.

In the second instance, all the `faces-config.xml` files in the following core plug-ins are read and the classes declared within are loaded in a batch. These mostly consist of XPages control renderers, data sources, and complex types.

- `com.ibm.xsp.core`
- `com.ibm.xsp.extsn`
- `com.ibm.xsp.designer`
- `com.ibm.xsp.domino`
- `com.ibm.xsp.rcp`

Note that `com.ibm.xsp.rcp` is a Notes client–only plug-in and that both groups of classes are loaded simultaneously on separate threads. Also, unless these preload variables are set in **notes.ini**, no XPages class loading of any description is performed before the first XPages application is opened in a given client or server session.

XPagesPreloadDB Variable

Whereas the `XPagesPreload` variable is concerned with loading critical pieces of the core runtime platform, `XPagesPreloadDB` is concerned with the application layer. The `XPagesPreloadDB` variable points to one or more NSF applications, which can be local to the client or server or on a remote server. Any nominated NSF can also include an optional XPages **.xsp** filename. The following sample **notes.ini** variable declaration includes an example of a local NSF that specifies an XSP page (**ByAuthor.xsp**) and an NSF on a remote server (**bigIron**) that does not:

```
XPagesPreloadDB=teamdisc.nsf/ByAuthor.xsp,bigIron!!expenseDb.nsf
```

In the normal course of events, when an end user from a browser loads an XPages application, the XPages runtime loads the NSF as a virtual web application module. The concept behind preloading is to load the web application module into memory at startup time in the same way it would occur normally if a real user had submitted an actual application load request. Thus, the XPages runtime fakes a real request for each argument in the `XPagesPreloadDB` comma-separated list. This means that an XPages URL is constructed for each argument and sent to the web application server. The web application server loads the application module, caches it, and renders back markup if an XPages file is specified. The XPages runtime simply throws away the response from the web application server, but the module has been loaded into memory and will be available instantly for the next real user request.

When and Why Is Preloading Important?

Preloading the runtime technology and/or XPages applications is important when initial application load time is a project requirement—in other words, preloading is important

almost all the time! While the XPages preloading features work on both the Notes client and Domino server, it can be especially significant when an XPages application is run in the Notes client but the NSF itself resides on a remote Domino server. In this scenario, application startup performance can be slower than normal, depending on the load and latency of the network infrastructure on which the applications are deployed. To understand this fully, however, you need to quickly look at how XPages runs in the Notes client.

In a typical, albeit oversimplified, XPages web application scenario, a user makes a request for an XPage to a Domino server that responds by generating the XPages content and emitting the markup back to the user's browser client. The same applies when running an XPages application in the Notes client, except that the Expeditor (XPD) web application server and (XULRunner) browser client are embedded in the client platform and thus are running on the local desktop computer. When running with a local client NSF application—say, a replica of an XPages web application that has been taken offline—this all runs the same as the typical web scenario and with a similar performance profile. Figure 2.6 summarizes the XPages client runtime model.

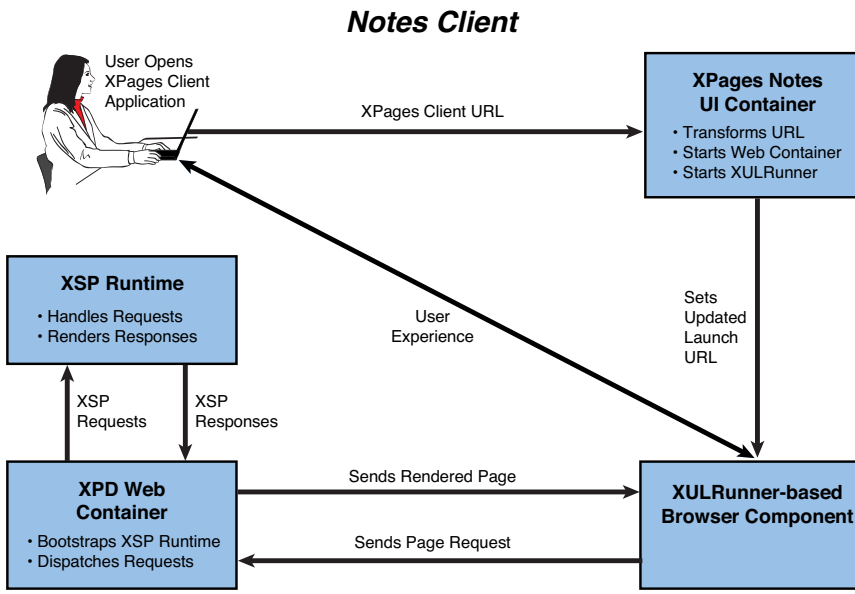


Figure 2.6 XPages in the Notes client

However, when the NSF is not a local replica, but a file on a remote server, all the data displayed as content in the XPage must be fetched across the network from the remote server. This can result in numerous network transactions being carried out just to populate the XPage. On top of this, however, all the XPages and Custom Control design elements themselves must be fetched from the remote NSF to be executed in the local Notes web application server. In terms of performance optimization, network

transactions should always be minimized because applications inevitably slow down due to the extra time required to complete network transactions (that is, over and above the time required to fulfill equivalent local disk access operations). On networks with high latency, the performance impact can be significantly exacerbated. Preloading an application ensures that any remote network activity required for application design artifacts is absolutely minimized by the time the end user first touches the application. It also ensures that the web container has been started in advance of the first user request, to eliminate overhead. Thus, the XSP preload INI variables can really help mitigate the performance issues outlined here.

TIP Administrators can apply INI settings like this to many client computers in one fell swoop using a desktop policy settings document. The process is described in detail in this IBM support document: www.tinyurl.com/PushNotesIniSettings

Avoid Unnecessary Network Transactions in Your Application Code

Now that you understand the potential negative impact of network transactions on page loading when running a remote NSF application in the Notes client, it is important to apply this lesson to your own application code: In other words, do not fall into the same trap by loading application data inefficiently. A use case that we have encountered on more than one occasion with sample customer applications is highlighted here—avoid it at all costs. This example has to do with customizing views.

XPages makes it easy to access and display view data using container controls such as the View panel, Repeat control, or Data Table. These controls attach to a Domino view via a data source and display its content by iterating though the view entries, accessing the row data, and formatting it for display according to various design-time parameters. The controls offer the powerful capability of adding customized view columns to the output, which often transform back-end view data or compute new values based on Domino view data for display purposes. However, there are efficient and inefficient ways of customizing view column data. The egregious offender is the column value computation that uses Server Side JavaScript code to do something like this:

```
return rowData.getDocument().getItemValueString("Customer");
```

Here, `rowData` is the sample variable name parameter on the container control (that is, the name assigned to the `var` property) that gives access to the current row in the view as it is being read and rendered at runtime. When the page content is being built, the underlying document for the current view entry is opened and an item value is read from the document instance. This operation is repeated for every row displayed in the view control. In one particular real-world instance, a View control was configured with two custom columns, both of which opened the underlying document to extract item data, and the View control was configured to show 25 rows at a time. Thus, in one fell swoop, 50 extra and expensive network transactions had to be carried out for this page when this NSF was run as a remote application on the Notes client. Ouch!

When custom columns are required, it is always best to work with columns that are already defined in the back-end Domino view and then retrieve the column content for

custom computation using the `getColumnValue()` API call on the `NotesXspViewEntry` JavaScript class. If the required column does not exist in the back-end Domino view, add it if you have sufficient design privilege, but do *not* open the underlying document to read the item and then have this bad practice repeated iteratively for every view entry to be displayed.

A simple example of a customized column is the concatenation of two back-end view column values into a single XPages View control column, formatted in a particular way. You can achieve such a custom value using Server Side JavaScript (in particular, by leveraging the aforementioned `getColumnValue()` API call) to extract the column data from the Domino view:

```
return "" + rowData.getColumnValue("Area") + ":" +
➥rowData.getColumnValue("Customer");
```

Optimizing Client Memory Usage

Most of the usage scenarios outlined in this chapter deal with **notes.ini** variables and are of primary importance when used within the Domino server. However, a number of settings used on the Notes client can help configure the memory-management settings of the Notes client.

The Notes client is based on Eclipse since release 8.0. As a result, Notes requires a JVM to launch and run. The JVM that Notes uses is installed as a part of the Notes client and launches each time the Notes client is launched. The JVM is installed to the **jvm** folder, which is a child of the Notes program directory (for example **C:\Notes\jvm**).

The Notes JVM can be configured by modifying **jvm.properties,** which resides in **<notes_program_directory>\framework\rcp\deploy**. Users should be careful when modifying the **jvm.properties** setting: Changing some settings can negatively impact the Notes client's performance. It is advisable to keep a backup copy of **jvm.properties** before modifying it.

Similar to the **notes.ini** heap variables discussed previously, two Notes client **jvm.properties** attributes provide the same `vm` arguments to the Notes client's JVM. Table 2.2 outlines both of those properties.

Table 2.2 Notes Client JVM Memory Management Properties

jvm.properties Property	Description
vmarg.xms	Sets the minimum heap size to be used by the Notes client's JVM
vmargs.xmx	Sets the maximum heap size to be used by the Notes client's JVM

Users should take note when modifying the value of these properties. If the user is confident that the system has enough overall system memory to support increasing the

memory allocated to the Notes client, increasing the minimum and maximum JVM heap size for the Notes client can yield performance improvements for the Notes client.

vmarg.Xms

This property corresponds to the `-Xms vm` argument. The value of this property is passed to the JVM as the minimum heap size allowed for the JVM. Increasing this property leads to an increased initial heap size for the Notes client JVM.

vmarg.Xmx

This property corresponds to the `-Xmx vm` argument. The value of this property is passed to the Notes client's JVM as the maximum heap size allowed for the JVM. Increasing this property allocates more memory to the Notes client JVM and, ultimately, the Notes client. Listing 2.2 shows how to set the minimum and maximum memory arguments via **jvm.properties**.

Listing 2.2 Sample Notes client jvm.properties Values

```
vmarg.Xmx=512m
vmarg.Xms=128m
```

Any variable can be passed into the Notes client's JVM using **jvm.properties**. `vm` arguments are passed to the Notes client's JVM using the `vmarg.arg` notation. The argument name succeeding the dot is passed directly to the Notes client's JVM as in `vmarg.Xnolinenumbers=-Xnolinenumbers`.

Enabling Extended Java Code with the java.policy File

Another configuration file to know about is the **java.policy** file. The XPages Java Security Manager uses this file to determine what classes are trusted in the XPages runtime environment of the Notes client and Domino server. It is located in the **jvm\lib\security** folder under the Notes/Domino root installation directory. The Notes client has an additional **java.policy** file in its root directory. This is done to support the Mac platform. The content of both files is effectively concatenated as one by the security manager on the Notes client. Listing 2.3 shows a snippet from the **java.policy** file on an 8.5.3 Domino server, with some lines highlighted for special attention.

Listing 2.3 Sample Snippet from a Domino Server java.policy Configuration File

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/*" {
     permission java.security.AllPermission;
};
```

```
// default permissions granted to all domains

grant {
     // Allows any thread to stop itself using the java.lang.Thread.
➥stop()
     // method that takes no argument.
     // Note that this permission is granted by default only to remain
     // backwards compatible.
     // It is strongly recommended that you either remove this
➥permission
     // from this policy file or further restrict it to code sources
     // that you specify, because Thread.stop() is potentially unsafe.
     // See "http://java.sun.com/notes" for more information.
     permission java.lang.RuntimePermission "stopThread";

     // allows anyone to listen on un-privileged ports
     permission java.net.SocketPermission "localhost:1024-", "listen";

     // "standard" properies that can be read by anyone

     permission java.util.PropertyPermission "java.version", "read";
     permission java.util.PropertyPermission "java.vendor", "read";
     permission java.util.PropertyPermission "java.vendor.url", "read";
     "java.vm.specification.name", "read";
     permission java.util.PropertyPermission "java.vm.version", "read";
     // etc ...
     permission java.util.PropertyPermission "java.vm.vendor", "read";
     permission java.util.PropertyPermission "java.vm.name", "read";

     permission java.util.PropertyPermission "javax.realtime.version",
➥"read";

};

// Notes java code gets all permissions

grant codeBase "file:${notes.binary}/*" {
     permission java.security.AllPermission;
};

grant codeBase "file:${notes.binary}/rjext/*" {
     permission java.security.AllPermission;
};
```

```
grant codeBase "file:${notes.binary}/ndext/*" {
     permission java.security.AllPermission;
};

grant codeBase "file:${notes.binary}/xsp/-" {
     permission java.security.AllPermission;
};

grant codeBase "file:${notes.binary}/osgi/-" {
     permission java.security.AllPermission;
};
```

The first `grant` statement in Listing 2.3 declares that the security manager trusts any Java JAR files located in the **jvm\lib\ext** folder under the root Notes/Domino directory. This means that you could drop your own custom Java libraries into this location, and they would be included in the class path and trusted by the security manager at runtime. This is not a recommended practice, however, because the location is intended for global system libraries; including private custom libraries there would pollute the model.

At the bottom of the listing are some other locations that are also granted all permissions by the security manager. Shaded in gray is the **xsp** subfolder; you can include your own custom Java libraries at this location (say, by including a JAR file in the **xsp\shared** folder). This is no longer a recommended practice, but it is still supported for historical reasons. You can also encapsulate your custom Java classes in a plug-in and place them in the OSGi location (the last entry in the listing), to ensure that your classes are successfully loaded. However, this is also not recommended because the upgrade installer removes everything under the **osgi** folder whenever the server is next upgraded. The recommended way is to deploy custom plug-ins in the workspace subfolder path under the server data directory (**domino\workspace\applications\eclipse\plugins**). This location automatically inherits all Java 2 security settings of the OSGi directory, and contents are preserved in the event of an upgrade.

If you want to include custom Java code in an NSF and reuse it in other XPages applications, you must add a grant declaration, such as that shown in Listing 2.4.

Listing 2.4 A Grant Declaration for XPages Java Code Contained in an NSF Application

```
grant codeBase "xspnsf://server:0/xsp85code.nsf/-" {
 permission java.security.AllPermission;
};
```

The `xspnsf` protocol identifies the code source as coming from XPages in an NSF file. The server, port, and actual NSF file are then specified in the remaining part of the URL. You can refine the scope of the approved code source by further modifying the URL. For example, to allow only Java code called via Server Side JavaScript to execute, add a `script` path identifier. Listing 2.5 shows the modified grant declaration.

Listing 2.5 A Grant Declaration for XPages Java Called Via SSJS in an NSF Application

```
grant codeBase "xspnsf://server:0/xsp85code.nsf/script/-" {
 permission java.security.AllPermission;
};
```

Note also that some URLs are suffixed with an asterisk character (`*`), whereas others end with a hyphen (`-`). The former includes all files in the designated location; the latter is recursive, meaning it also includes any libraries found in subfolders of the location.

The other entries in the **java.policy** file may be vaguely interesting to you. For example, you can see where the other standard Java components, such as Notes/Domino extensions (`ndext`), are declared. There is also a collection of individual permission declarations:

```
permission java.util.PropertyPermission "java.version", "read";
```

This simply means that anyone is allowed to read the version of Java running on Notes/Domino. The other individual statements are equally straightforward.

Finally, custom libraries can also be included in the Java class path using the `JavaUserClasses` INI variable.

JavaUserClasses

This setting existed long before the advent of XPages in Notes /Domino 8.5, but it is still effective today as a directive to include custom or third-party libraries in the class path. The example included in Listing 2.1 shows how to add a JAR file or a file path location to the class path:

```
JavaUserClasses=D:\jdbcutils\jdbcdrivers.jar;D:\java\customlibs
```

The list of JARs, ZIPs, or path locations is ultimately limited to 255 characters. This can be an issue when third-party libraries are installed into long-winded file system paths, but there are workarounds for this limitation. One solution is provided by way of another INI variable, named `JavaUserClassesExt`, that was introduced subsequently to perform the same function but is designed to accept a list of tags instead of a list of literal resources. Listing 2.6 shows an example.

Listing 2.6 A notes.ini Snippet Showing JavaUserClassesExt and Related Tags

```
JavaUserClassesExt=ibm,json
ibm=C:\IBM\financelib.jar
json=D:\utils\jsonlib.jar
};
```

Note that the XPages Java Security Manager does not trust any Java class files added to the class path in this way. That is, the class files may be loaded, but any attempt to

perform protected operations will be denied. You can use the `AccessController.doPrivileged` action if you need to add privileged operations to your custom Java code. Listing 2.7 shows a simple example of this.

Listing 2.7 A Simple Code Snippet That Calls Third-Party Java Code Via Server Side JavaScript in XPages

```
String getUserNameProtected() {
   String user = (String) AccessController.doPrivileged(
      new PrivilegedAction() {
         public Object run() {
            return System.getProperty("user.name");
            }
         }
      );
    return user;
}
```

If you want to experiment with extending the Domino class path, setting up a minimal test case is easy. For example, take a third-party standalone Java library, such as the Apache Commons BeanUtils, and install the JAR in one of the aforementioned jar locations, or put it in a private folder and add it to the class path using the `JavaUserClasses` setting. On an XPage, you can attach some Server Side JavaScript to a button such as the one shown in Listing 2.8.

Listing 2.8 A Simple Code Snippet That Calls Third-Party Java Code Via Server Side JavaScript in XPages

```
<xp:button value="Label" id="button1">
      <xp:eventHandler event="onclick" submit="true"
         refreshMode="complete">
            <xp:this.action><![CDATA[#{javascript:
// Instantiate the array stack, add an entry and print it out
var as = new org.apache.commons.collections.ArrayStack();
as.push("foo");
println(as.size());
println(as.get());}]]>
            </xp:this.action>
      </xp:eventHandler>
</xp:button>
println(as.get());
```

Clicking the button in the XPage on a web browser at runtime causes the array size (`1`) and array content (`foo`) to be printed on the server console if the Apache Commons BeanUtils library is properly loaded.

Conclusion

In this chapter, you explored some core Notes/Domino configuration files (namely **notes.ini**, **jvm.properties**, and **java.policy**) and saw how the variable settings can control many aspects of behavior of the XPages runtime on the Notes client and Domino server. These settings mostly serve as XPages debugging, performance, and extensibility aids. Hopefully, you found some nuggets that you can apply to solve or simplify problems you have encountered in your everyday XPages application development adventures.

CHAPTER 3

Working with the Console

When working with an application running on an application server, it is often necessary to interact with the server's console to analyze, troubleshoot, and debug any problems that might arise. This is also true for XPages applications and controls. The Domino server console provides the developer with a wide variety of commands, ranging from starting a server task to reporting the status of an OSGi bundle running on the server. At some point, the XPage developer inevitably will need to call on the console to analyze why an application is not working or functioning as desired.

The Domino server has a long history. Over time, the Domino server's console has served Domino administrators and developers alike as the first line of attack when troubleshooting problems. In an effort to maintain this level of service over the evolution of the server, and to enable administrators and developers to quickly get to the root of issues relating to XPages applications, the server's console has been instrumented with a large array of commands specifically built with the XPages runtime in mind.

About the XSP Command Manager

The XPages runtime is embedded within the Domino server's HTTP task. The XSP Command Manager serves as the common link for the Domino HTTP task, the server's JVM, and the XPages runtime. The XSP command manager is responsible for dispatching XPages' requests received from the HTTP task and the Domino console, and is also ultimately responsible for the XPages runtime's lifecycle. The XSP Command Manager has many useful commands built in that enable the administrator or developer to quickly analyze whether a particular XSP setting is causing an issue. It also can generate Java dumps that the development team can analyze.

How to Execute the XSP Command Manager Commands

Commands are executed via the XSP Command Manager similar to any other command on the Domino server. The XSP Command Manager is running within the HTTP task, so the commands it executes must be fed through the HTTP task, as in this example:

```
tell http xsp <<xsp command manager command>>
```

Table 3.1 lists all the XSP Command Manager Commands.

Table 3.1 XSP Command Manager Commands

Command Name	Description
show data directory	Shows the location of the Domino server's data directory.
show program directory	Shows the location of the Domino server's program directory.
show version	Displays the exact version of the XPages runtime that is installed and running on the Domino server.
show settings	Shows all the variables/properties that have been set on the server's **bootstrap.properties** file. If a **bootstrap.properties** file does not exist, the XPages runtime provides reasonable recommended defaults.
show modules	Displays the *modules* loaded in the system. The XPages runtime dynamically loads each Domino database as a web application *module*.
refresh	Causes the services in the XPages runtime to be refreshed. This is mainly reserved for future use.
heapdump	Performs a live dump of all objects on the Domino server's Java heap. Creates a dump file that must be read by other tools (such as the Eclipse Memory Analyzer); the file is not human readable.
javadump	Performs a Java dump, sometimes referred to as a thread dump or JavaCore dump, of the Domino server's JVM. The information collected during the dump operation is stored in human-readable format.
systemdump	Performs a full system dump, sometimes referred to as a core dump, of the Domino server's JVM. The dump information is platform specific and contains all the memory, process, and thread information for the JVM at the time the dump occurred.

These commands can greatly aid administrators and developers when trying to analyze particular issues. The dump commands are of particular importance because they perform diagnostic dumps on the server's JVM but do not cause the JVM or the server to stop operation.

show data directory

As the name suggests, this command simply tells the user where the Domino server's **data** directory resides on the operating system's file system. In a Domino server environment, the **data** directory stores all the databases that are available through the Domino server. The location of this directory is significant because all applications running on the server will reside in this directory or within a subdirectory of this directory.

Sample usage:

```
tell http xsp show data directory
```

Figure 3.1 shows the results of running the `show data directory` command on a Domino server.

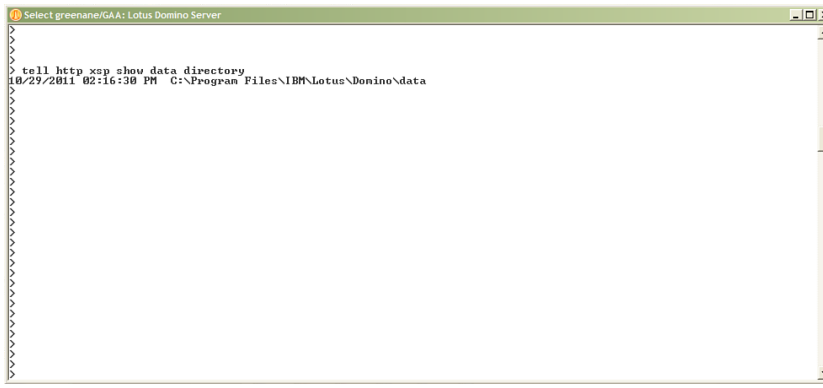


Figure 3.1 Result of running the show data directory command

show program directory

This command tells the user where the Domino server's program directory resides on the operating system's file system. This command can be convenient for developers who are not familiar with a particular setup of an individual server machine. The command enables developers or administrators to quickly identify the file system location of the Domino server's program directory.

Sample usage:

```
tell http xsp show program directory
```

Listing 3.1 shows the result of running the `show program directory` command in the Domino server console.

Listing 3.1 Result of Running the show program directory Command in the Console

```
> tell http xsp show program directory
09/20/2011 10:52:33 PM  C:\Program Files\IBM\Lotus\Domino
```

show version

This command shows the exact version of the XPages runtime that is installed and running on the Domino server. The version number is updated only when upgrading from one release to another of XPages core runtime. Adding or upgrading extensions such as

the XPages Extension library does not update the version number. This command typically is used when a developer or administrator needs to confirm which version of the XPages runtime is running on a particular server. New features are added to the XPages runtime with each release. These features can range from new properties on existing controls to entirely new controls. Over time, a developer or administrator must confirm that the version of XPages runtime is at the appropriate release level for the applications running on the server. This command enables the developer or administrator to quickly confirm the XPages runtime version.

Sample usage:

```
tell http xsp show version
```

Listing 3.2 shows how to determine the version of the XPages runtime.

Listing 3.2 Result of Running the show version Command in the Console

```
> tell http xsp show version
09/20/2011 04:34:21 PM  XSP Runtime Version: [DSI8.5.3] 20110629.1645
```

In the previous example, the version number can be broken down as follows:

- The `DSI` prefix is a constant, which does not vary from release to release.
- `8.5.3` represents the *Major.Minor.Maintenance* version number. The first digit is updated with each major feature release, the second digit is updated with each minor feature release, and the last digit is updated with each maintenance release.
- The final number (`20110629.1645`) represents the time stamp (yyyyMMdd.hhmm) at which the build in question occurred.

show settings

This command makes a request to the XPages runtime to print all the settings in use by the runtime. By default, the XPages runtime is configured with a host of default settings. These settings can be overwritten by adding a **bootstrap.properties** file to the **xsp** directory, which resides in the Domino server's program directory (for example **C:\domino\xsp**). As a result of being able to override the default settings in the XPages runtime (via **bootstrap.properties**), it is not guaranteed that the XPages runtime defaults will apply from server to server. This command enables developers and administrators to quickly list all the current settings without needing to manually access various file system locations to determine which properties are being applied.

Sample usage:

```
tell http xsp show settings
```

Listing 3.3 shows the XPages runtime default settings being output to the Domino server console.

Listing 3.3 Result of Running the show settings Command in the Console (Default Case)

```
> tell http xsp show settings
09/16/2011 11:24:26 AM  xsp.commas.not.delimiters.in.cookie=false
09/16/2011 11:24:26 AM  com.ibm.faces.USE_UNENCODED_CONTEXT_PATH=/xsp
09/16/2011 11:24:26 AM  xsp.gc.on.shutdown=false
09/16/2011 11:24:26 AM  xsp.sessionid.name=SessionID
09/16/2011 11:24:26 AM  xsp.default.charset=UTF-8
09/16/2011 11:24:26 AM  xsp.log.severe.stack.trace=false
09/16/2011 11:24:26 AM  xsp.default.post.buffer.size=1024
09/16/2011 11:24:26 AM  xsp.allow.cookie.sessionid=true
09/16/2011 11:24:26 AM  xsp.global.context.path=/xsp
09/16/2011 11:24:26 AM  xsp.send.set.cookie2.header=true
09/16/2011 11:24:26 AM  xsp.max.cookies.per.session=50
09/16/2011 11:24:26 AM  xsp.allow.packagenames=false
09/16/2011 11:24:26 AM  xsp.allow.url.sessionid=true
09/16/2011 11:24:26 AM  xsp.default.chunk.post.buffer.size=8
```

In some cases, it is necessary to set extra system settings or even overwrite existing settings. Being able to quickly analyze which settings have changed can be invaluable. Listing 3.4 shows a case in which some settings (`xsp.sessionid.name`) have been overwritten by **bootstrap.properties** and some new logging settings (`log_configuration` and `logdir`) have been added. Chapter 6, "Server-Side Debugging Techniques," explains these settings

Listing 3.4 Result of Running the show settings Command in the Console

```
> tell http xsp show settings
09/16/2011 11:01:47 PM  xsp.commas.not.delimiters.in.cookie=false
09/16/2011 11:01:47 PM  com.ibm.faces.USE_UNENCODED_CONTEXT_PATH=/xsp
09/16/2011 11:01:47 PM  xsp.gc.on.shutdown=false
09/16/2011 11:01:47 PM  log_configuration=xsp/log.properties
09/16/2011 11:01:47 PM  xsp.sessionid.name=FOOID
09/16/2011 11:01:47 PM  xsp.default.charset=UTF-8
09/16/2011 11:01:47 PM  xsp.log.severe.stack.trace=false
09/16/2011 11:01:47 PM  xsp.default.post.buffer.size=1024
09/16/2011 11:01:47 PM  xsp.allow.cookie.sessionid=true
09/16/2011 11:01:47 PM  xsp.global.context.path=/xsp
09/16/2011 11:01:47 PM  xsp.send.set.cookie2.header=true
09/16/2011 11:01:47 PM  xsp.max.cookies.per.session=50
09/16/2011 11:01:47 PM  xsp.allow.packagenames=false
09/16/2011 11:01:47 PM  xsp.allow.url.sessionid=true
09/16/2011 11:01:47 PM  logdir=c:/Domino/log
09/16/2011 11:01:47 PM  xsp.default.chunk.post.buffer.size=8
```

show modules

Each Domino database (.NSF) that is running within the XPages runtime is loaded by the XPages runtime as an application *module*. The `show modules` command shows all the databases (NSF modules) that are currently running within the XPages runtime. This command also shows registered system service modules that the XPages runtime automatically loads. This command is convenient for server administrators who need to know which XPages applications are being served by the XPages runtime at any point in time.

Sample usage:

```
tell http xsp show modules
```

Listing 3.5 shows all the active modules running within a Domino server that has sessions open for three XPages applications.

Listing 3.5 Result of Running the show modules Command in the Console

```
> tell http xsp show modules
09/16/2011 11:47:36 AM  XSP Resources
09/16/2011 11:47:36 AM  Default Http Registry Module
09/16/2011 11:47:36 AM  OSGI WebContainer Bridge Service
09/16/2011 11:47:36 AM  oauthtokenstore.nsf
09/16/2011 11:47:36 AM  lsdemo2011.nsf
09/16/2011 11:47:36 AM  xpagessbt.nsf
```

In Listing 3.5, six modules are listed. Three of these modules are XPages runtime system modules; the other three modules represent XPages applications that are currently running on the server.

- `xpagessbt.nsf`, `lsdemo2011.nsf`, and `oauthtokenstore.nsf` are all XPages applications that were running on the server when the command was executed.
- XSP Resources is a module loaded by the XPages runtime; it is not configurable.
- Default Http Registry Module is a module loaded by the Domino web container; it is not configurable.
- OSGI WebContainer Bridge Service is a module loaded by the Domino to OSGi bridge; it is not configurable.

The core runtime modules are not configurable and can be removed or added to in future releases.

refresh

This command was implemented with future extensions of the XSP Command Manager's HTTP service in mind. As of release 8.5.3 of the Domino server, this command

does nothing. It is intended to be used with HTTP services and will enable services to be refreshed as necessary without restarting the HTTP task or the XPages runtime.

Sample usage:

```
tell http xsp refresh
```

heapdump

The `heapdump` command performs a live dump of all objects on the Domino server's Java heap. The operation creates a dump file that must be read by third-party tools; the file is not human readable. The dump file can be read using tools such as the Eclipse Memory Analyser Tool (www.eclipse.org/mat). Because the dump file is written in the IBM JVM heap dump format, it is necessary to install further add-ons to the Eclipse Memory Analyser Tool to read the heap dump information. You can download the add-on for the Eclipse Memory Analyzer tool from www.ibm.com/developerworks/java/jdk/tools/dtfj.html. The `heapdump` command causes a dump file to be generated in the server's program directory, as demonstrated in Figure 3.2.

Sample usage:

```
tell http xsp heapdump
```

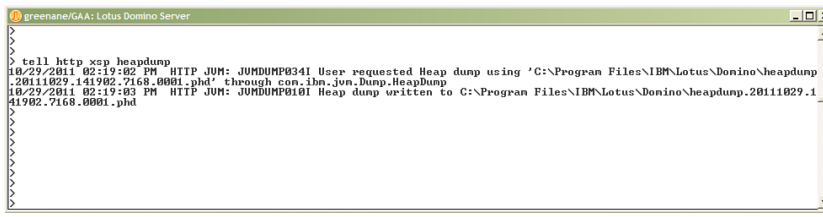


Figure 3.2 Result of running the `heapdump` command in the console

When configured, the Eclipse Memory Analyzer tool enables the user to read the content of the dump file and provide information on where memory is potentially being leaked and which objects are in use when the dump occurs. Figure 3.3 shows sample output from the Eclipse Memory Analyzer Tool.

javadump

Running the `javadump` command causes the server's JVM to create a Java Dump file. Sometimes referred to as a thread dump or a Javacore dump, the dump information is written to disk in a human-readable format—the contents of the dump file can be opened with applications such as Microsoft Notepad. The information stored as a result of a `javadump` is generally diagnostic information relating to the threads, stacks, locks, and memory that were in use by the JVM when the dump occurred. Javadump files are of particular use where the developer or administrator needs to quickly obtain system information (such as operating system version, JVM version, and loaded threads).

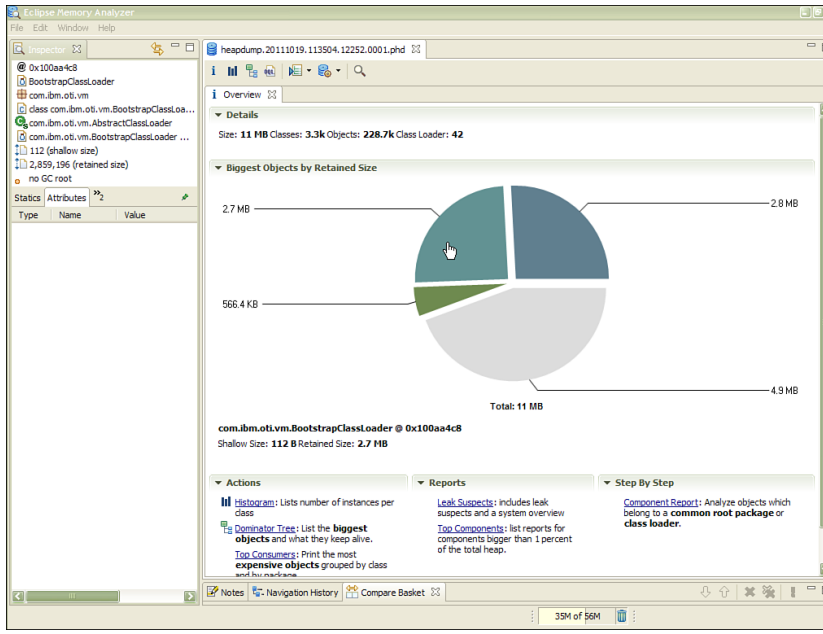


Figure 3.3 Eclipse Memory Analyzer Tool

The Javadump file is lightweight by nature and can help to quickly identify which threads are hung in the system.

Sample usage:

```
tell http xsp javadump
```

Listing 3.6 shows the console output when the `javadump` command is executed.

Listing 3.6 Result of Running the javadump Command in the Console

```
> tell http xsp javadump
10/18/2011 11:40:00 PM  HTTP JVM: JVMDUMP034I User requested
Java dump using 'C:\Program Files\IBM\Lotus\Domino\
javacore.20111018.233959.8220.0001.txt' through com.ibm.jvm.Dump.
➥JavaDump
10/18/2011 11:40:01 PM  HTTP JVM: JVMDUMP010I Java dump written to C:\
Program Files\IBM\Lotus\Domino\javacore.20111018.233959.8220.0001.txt
```

In Listing 3.6, you can see the result of executing the `javadump` command. A Javadump file is written to the location specified in the console output. It is beyond the scope of this book to go into the details of reading the contents of dump files. However, in the case of Javadump files, a few tips can easily be bestowed upon the reader to make

reading the contents of the Javadump file easier. The dump file can essentially be broken down into different sections:

- Date and time of the javadump.
- Operating system signal information (who initiated the javadump and how it was initiated). The signal information tells the reader whether the user initiated the dump or whether the operating system did so due to a program fault. The signal information is operating system specific.
- Java (JVM) version.
- Information about threads running when the javadump occurred.
- Operating system and processor details.
- Native libraries loaded by the JVM.
- Full command line, including arguments, that the Domino server used to launch the JVM.
- JVM monitor information.
- Current stack for each thread running within the JVM.

For further in-depth information on how to read the contents of the Javadump file, see the following article from IBM support:

`www-01.ibm.com/support/docview.wss?uid=swg21181068`

Alternatively, you can search for information on how to read a javacore dump file in your favorite Internet search engine.

systemdump

The `systemdump` command is the most intensive of the three dump commands available through the XSP Command Manager. As a result, the footprint of the resulting systemdump file can be quite large. The systemdump file contains detailed information on the JVM's threads, memory, and active processes. When a Java application crashes as a result of general protection fault failure or as a result of a major JVM error, a systemdump file is generated by default.

Sample usage:

```
tell http xsp systemdump
```

Listing 3.7 shows the console output when the `systemdump` command is executed.

Listing 3.7 Result of Running the systemdump Command in the Console

```
09/20/2011 12:36:30 AM  HTTP JVM: JVMDUMP034I User requested
System dump using 'C:\Program Files\IBM\Lotus\Domino\
core.20110920.003630.8220.0002.dmp' through com.ibm.jvm.Dump.SystemDump
```

```
09/20/2011 12:38:26 AM  HTTP JVM: JVMDUMP010I System dump written to
C:\Program Files\IBM\Lotus\Domino\core.20110920.003630.8220.0002.dmp
```

The dump file is stored in a platform-specific format and, as a result, must be read by tools specific to the platform on which the dump was created. The IBM Dump Analyzer enables you to read and analyze the contents of a system dump that is performed on the Domino server. For more information on the IBM Dump Analyzer tool, refer to the following websites:

- "Java Diagnostics, IBM Style, Part 1: Introducing the IBM Diagnostic and Monitoring Tools for Java—Dump Analyzer," at IBM.com: www.ibm.com/developerworks/java/library/j-ibmtools1/
- "Installing the IBM Monitoring and Diagnostic Tools for Java—Dump Analyzer," at IBM.com: www.tinyurl.com/IBMJavaDumpAnalyzer

The information generated by a system dump is extremely granular in nature. An XPage developer rarely will need to create a system dump because the information the dump generates details information about every process executing on the system, not just the information pertinent to the JVM. A system dump generally is needed only when the failure is as a result of complex interactions with programs running outside the Domino server.

Working with the OSGi Console

Before delving into the inner workings of the OSGi console, it is best to briefly explain OSGi. OSGi stands for Open Services Gateway initiative framework. This framework allows software to be written and executed as independent components. In OSGi-speak, these components are referred to as *bundles*. OSGi is used in a wide range of applications, from client programs such as Eclipse and IBM Lotus Notes, to mobile phones, to server applications such as IBM Lotus Domino. As a result of their modular nature, OSGi bundles can be started, stopped, and debugged on an individual basis, without the need for stopping or restarting the entire platform. Both the Domino server and the Notes client use Eclipse's implementation of OSGi (Equinox) as their OSGi runtime platform.

OSGi was added to the Domino platform in release 8.5.2. As a result, in Domino 8.5.2, the XPages runtime was repackaged to run as OSGi bundles (instead of just a regular collection of Java JARS), also referred to as *Eclipse plug-ins*.

The OSGi console allows for the input of commands that the OSGi platform then performs. The platform posts the results of such commands back to the console. The OSGi platform itself has a whole host of commands that can simplify the troubleshooting of problems. The OSGi console can assist developers in developing XPages controls and applications, as well as assisting support personnel in diagnosing runtime issues. Developers who extend the XPages runtime by creating libraries will find the OSGi console commands to be a particularly powerful tool in analyzing problems. The OSGi console

is of particular use when the developer/administrator needs to know whether individual plug-ins (or sets of plug-ins) are loading correctly or which version of a plug-in is in use.

As mentioned earlier, OSGi is embedded within both the Notes client and the Domino server. Depending on where your XPages application is running (whether on the client or the server), your method of accessing the OSGi console will vary. We start by explaining how to access the OSGi console on the Domino server.

OSGi is embedded within the HTTP task on the Domino server, as a result, the OSGi console is started automatically whenever the HTTP task is started. OSGi console commands are routed to the OSGi console via the HTTP task. That is, when entering an OSGi console command on the Domino server, the user must tell the HTTP task to route the specified command to the OSGi console—for example:

```
tell http osgi <<command>>
```

Here, `<<command>>` is the name of the OSGi console command. Any OSGi command can be executed using the preceding syntax.

When it comes to OSGi commands, every developer and administrator should know several rudimentary commands. These commands can be your "go to" commands when problems arise—say, when you suspect bundle loading might be a factor. Even when you do not *think* that bundle loading is the problem, it is often best to first confirm that the bundle is actually loaded before proceeding with other debugging techniques.

Table 3.2 lists some of the more commonly used OSGi commands that are available to use for diagnosing plug-in issues on the Domino server (and also the Notes client). In Table 3.2, *bundle-symbolic-name* is referenced extensively. This is the name by which the OSGi platform references bundles. *bundle-symbolic-name* correlates directly to the *Bundle-SymbolicName* manifest header, often referred to as the *plug-in name*.

Table 3.2 OSGi Console Commands

Command Syntax	Description
tell http osgi diag <bundle-symbolic-name>	Diagnoses the status of the bundle whose name is provided. Determines whether the bundle is resolved and, if not, states why the bundle is not resolved.
tell http osgi ss <bundle-symbolic-name>	Lists the status of all bundles in the system. Optionally, a symbolic name or a symbolic name prefix can be provided to obtain the status of a particular bundle or a subset of bundles.
tell http osgi start <bundle-symbolic-name>	Starts the bundle with the specified symbolic name.
tell http osgi stop <bundle-symbolic-name>	Stops the bundle with the specified symbolic name.
tell http osgi b <bundle-symbolic-name>	Prints metadata relating to the specified bundle.

Command Syntax	Description
tell http osgi headers <bundle-symbolic-name>	Lists the OSGi headers for the specified bundle.
tell http osgi help	Lists all the OSGi command available on the server, along with some text describing each command.

All the commands listed in Table 3.2 can be entered via the Domino server console, with the results of such commands being printed back to the console, as illustrated in Figure 3.4.

Figure 3.4 Running an OSGi command on the Domino Server Console

All the commands referenced in Table 3.2 can alternatively take the *bundle id* as a parameter (instead of the *bundle-symbolic-name)*. The bundle id is a numeric ID that the OSGi runtime assigns to the bundle during platform initialization. The ID might vary from instance to instance of the platform, but users might find it easier to input than having to enter the entire bundle symbolic name. Examples of how to determine and use the bundle id are given later in this chapter.

Rarely does a single OSGi console command answer all the questions on why a plug-in is not loading or operating as expected. The following sections explain in greater detail how you can use each of these commands and the results you can expect to see from executing such commands.

diag <bundle-symbolic-name>

This is one of the most valuable commands in your arsenal and will likely be the one you'll use most frequently when diagnosing issues. You can use this command to determine whether a bundle is resolved within the OSGi platform. The status returned by this command will be one of the following:

- `No unresolved constraints`
- `Unresolved constraint`

If `No unresolved constraints` is the returned status, it suggests that the system has recognized the bundle and that all dependencies of the bundle are satisfied. When an `Unresolved constraint` status is returned, it suggests that one or more bundles

or packages that the bundle requires are missing or cannot be loaded. It is worth noting here that a bundle might still fail to *start* even though the OSGi console reports that the bundle has been *resolved*. If a bundle fails to start and is resolved, some code in the bundle's activator likely is failing (throwing an exception).

Sample usage:

```
tell http osgi diag com.ibm.xsp.core
```

Listing 3.8 shows the typical output of running the `diag` command against the com.ibm.xsp.core plug-in.

Listing 3.8 Result of Running the diag Command Against a Specific Bundle—Successful Case

```
> tell http osgi diag com.ibm.xsp.core
10/17/2011 09:43:14 PM
  initial@reference:file:../../shared/eclipse/plugins/com.ibm.xsp.
core_8.5.3.20110629-1645/[119]
10/17/2011 09:43:14 PM    No unresolved constraints.
```

In this case, the `diag` command reports that there were `No unresolved constraints` against the entered bundle symbolic name—in other words, the system recognizes the given bundle. Upon closer examination, the user can obtain further information about the bundle in question. It can determine where the bundle being used by the platform is installed, and the platform-assigned *bundle id* can also be obtained.

From reading the console output, the user can see that the bundle is installed to **../../shared/eclipse/plugins/com.ibm.xsp.core_8.5.3.20110629-1645.** The location specified is relative to the **osgi/rcp/eclipse** directory, which is a child of the Domino program directory. In this case, the console output indicates that the plug-in is installed at: **<domino program directory>/osgi/shared/eclipse/plugins**.

Finally, the output states the platform-assigned bundle id for the specified bundle. `119` is the id assigned to this bundle in this example. As stated previously, the OSGi commands listed here can use the bundle id interchangeably. In this example, executing the following command has identical output to that in Listing 3.8.

Sample usage:

```
tell http osgi diag 119
```

Listing 3.9 shows sample output of running the `diag` command in an unsuccessful scenario.

Listing 3.9 Result of Running the diag Command Against a Specific Bundle—Error Case

```
> tell http osgi diag com.ibm.xsp.extlib.sbt 09/09/2011 04:05:51 PM
update@../../../data/domino/workspace/applications/eclipse/plugins/com.
ibm.xsp.extlib.sbt_8.5.3.201108111413.jar [116]      09/09/2011 04:05:51
```

```
PM   Direct constraints which are unresolved: 09/09/2011 04:05:51 PM
Missing host com.ibm.xsp.extlib_0.0.0.
```

In Listing 3.9, you can see that the OSGi platform reports that the bundle in question is not resolved as a result of a missing dependency. We can see from the console output that the OSGi platform has actually found the bundle that we are looking for (`com.ibm.xsp.extlib.sbt`), but as one of the bundles that `com.ibm.xsp.extlib.sbt` depends on is not resolved, the `com.ibm.xsp.extlib.sbt` bundle does not get resolved itself. Looking a little more closely at the console output, we can determine the following:

The bundle `com.ibm.xsp.extlib.sbt` is installed at **../../../data/domino/workspace/applications/eclipse/plugins/com.ibm.xsp.extlib.sbt_8.5.3.201108111413.jar**. We now know that this path is relative to the **<domino program directory>/osgi/rcp/eclipse** directory. Hence, we can deduce that `com.ibm.extlib.sbt` is installed at the **<domino program directory>/data/domino/workspace/applications/eclipse/plugins/** directory.

The OSGi platform–assigned *bundle id* for this bundle is `116`.

One other tidbit of information can be extracted from the console output, in this case. The final line of the output tells us that the *host* is missing:

```
Missing host com.ibm.xsp.extlib_0.0.0
```

This tells us that the bundle we are looking for (`com.ibm.xsp.extlib.sbt`) is, in fact, a plug-in fragment, and the unresolved constraint (`com.ibm.xsp.extlib`) is the host plug-in.

ss, ss <bundle-symbolic-name>, or ss <bundle-name-prefix>

Similar to the `diag` command, this command quickly determines the status of a particular bundle—or all the bundles installed in the platform. Users can optionally specify a bundle name or a bundle name prefix to get the status of specific bundles. The returned status shows the bundle id, state, and bundle name of all bundles. In many situations, this command is just as useful as the `diag` command because it also reports the status of a bundle. This command does not tell the user why a particular bundle is not loading, but it does tell the user the state of a bundle.

Sample usage:

```
tell http osgi ss
```

Listing 3.10 shows the result of running the `ss` command without any parameters.

Listing 3.10 Result of Running the ss Command Without Any Bundle Name Parameter

```
> tell http osgi ss
09/09/2011 01:46:07 PM  Framework is launched.
09/09/2011 01:46:07 PM  id      State      Bundle
09/09/2011 01:46:07 PM  0       ACTIVE    org.eclipse.
osgi_3.4.3.R34x_v20081215-1030-RCP20110624-1648
09/09/2011 01:46:07 PM                    Fragments=57, 76, 88, 89, 235
09/09/2011 01:46:07 PM  1       RESOLVED  org.eclipse.equinox.
event_1.1.0.v20080225
09/09/2011 01:46:07 PM                    Fragments=32
09/09/2011 01:46:07 PM  2       RESOLVED  com.ibm.pvc.jndi.provider.
java.nl_6.2.3.20110625-0109
09/09/2011 01:46:07 PM                    Master=71
09/09/2011 01:46:07 PM  3       RESOLVED  com.ibm.eclipse.equinox.
preferences.nl_6.2.3.20110624-1648
09/09/2011 01:46:07 PM                    Master=85
09/09/2011 01:46:07 PM  4       <<LAZY>>  com.ibm.icu.
base_3.8.1.v20080530
09/09/2011 01:46:07 PM  5       RESOLVED  com.ibm.pvc.servlet.
jsp_2.1.0.20110625-0109
09/09/2011 01:46:07 PM  6       RESOLVED  org.apache.commons.
logging_1.0.4.20110625-0109
```

Listing 3.10 lists a subset of the information that displays when this command is run in a normal server environment. However, the listing does show all the information needed to understand the output of the command.

The command outputs several important pieces of information about each bundle:

- `Bundle-id`—for example, `2`, which is the OSGi platform–assigned ID of the bundle.
- `Bundle state`—for example, `RESOLVED`, which is the state of the bundle within the OSGi platform. A bundle can have one of seven states. Table 3.3 explains all of these.
- `Bundle name`—for example `com.ibm.eclipse.equinox.preferences.nl_6.2.3.20110624-1648`, which is the bundle symbolic name with its version information appended to the name.
- `Master` or `Fragments`—for example, `Master=71`. This data tells whether the bundle in question is a plug-in or a fragment. If the bundle specifies neither `Master` nor `Fragments`, it is automatically implied that the bundle is a plug-in bundle. The digits corresponding to the fragments or plug-ins are the OSGi platform–assigned bundle ids of the fragments or the master plug-in of the bundle in question.

Sample usage:

```
tell http osgi ss com.ibm.xsp.extlib
```

Listing 3.11 shows the result of running the `ss` command with a bundle prefix specified.

Listing 3.11 Result of Running the ss Command, Specifying a Bundle Prefix

```
> tell http osgi ss com.ibm.xsp.extlib
09/09/2011 02:25:36 PM  Framework is launched.
09/09/2011 02:25:36 PM  id      State        Bundle
09/09/2011 02:25:36 PM  108     RESOLVED com.ibm.xsp.extlib.
conns_8.5.2.20110724
09/09/2011 02:25:36 PM                       Master=117
09/09/2011 02:25:36 PM  109     RESOLVED     com.ibm.xsp.extlib.
domino_8.5.2.201107241628
09/09/2011 02:25:36 PM                       Master=117
09/09/2011 02:25:36 PM  112     RESOLVED     com.ibm.xsp.extlib.
oneui_8.5.2.201107241628
09/09/2011 02:25:36 PM                       Master=117
09/09/2011 02:25:36 PM  115     RESOLVED     com.ibm.xsp.extlib.
stime_8.5.2.201107241628
09/09/2011 02:25:36 PM                       Master=117
09/09/2011 02:25:36 PM  117     ACTIVE       com.ibm.xsp.
extlib_8.5.2.201107241628
```

Similar to Listing 3.10, Listing 3.11 shows the results of executing the `ss` command, only this time the command is passed a bundle symbolic name as a parameter. The `ss` command finds all bundles on the system that either start with the parameter or have a bundle symbolic name that is the same as the parameter. Listing 3.11 lists all the bundles, along with their bundle id and state.

Table 3.3 lists all the possible states of an OSGi bundle.

Table 3.3 OSGi Bundle States

State	Description
UNINSTALLED	The bundle is uninstalled and is unusable.
INSTALLED	The bundle has been installed, but the platform has not yet resolved it.
RESOLVED	The bundle has been resolved and is in a position to be started. Note that it is still possible for the bundle to fail to start, even though it has been resolved by the environment.
<<LAZY>>	Similar to `RESOLVED`, the platform has resolved the bundle and is in a position to be started. The bundle is not yet `ACTIVE` because it has been configured (via its bundle manifest) to be initialized lazily—that is, only when another `ACTIVE` bundle references the bundle will it be activated.

State	Description
STARTING	The bundle is in the process of starting. Either another bundle has specifically caused the bundle to start (by referring to a class within the bundle) or the user has manually started the bundle via the console. Rarely is a bundle in this state because it is transient.
STOPPING	The bundle is in the process of shutting down. Similar to `STARTING`, a bundle rarely is in this state.
ACTIVE	The bundle is running within the OSGi platform.

Developers and administrators should be aware that, on the Domino server, the state of a bundle is not persisted from one session to the next—that is, after the HTTP task is restarted, any bundles that were started manually in the previous session must be started again. Luckily, the `ss` command has an argument for filtering all bundles in a given state. The `ss` command can filter the bundles based on their state, by appending `-s [state]` to the command syntax.

Sample usage:

```
tell http osgi ss -s active
```

Figure 3.5 shows the output of running the `ss` command with the `-s` active argument.

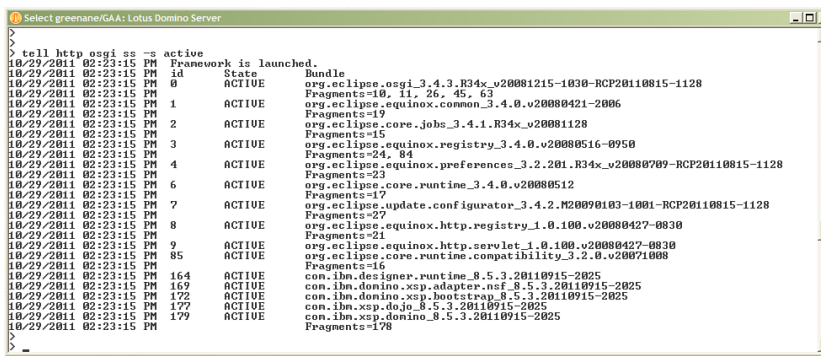


Figure 3.5 Result of running the ss command in the Domino server console

start <bundle-symbolic-name>

This command requests that the platform manually start the specified bundle. Calling this command does not guarantee that the specified bundle will be started. An exception can still occur during bundle initialization that would cause the bundle initialization to fail. Performing an `ss` command after the `start` command reports the status of the bundle. This command is helpful when a new bundle has been installed on the server,

but the administrator or developer is not in a position to restart the HTTP task to start the new bundle.

Sample usage:

```
tell http osgi start com.ibm.xsp.extlib.sbt
```

Figure 3.6 shows that, by running a combination of the `ss` and `start` commands, a bundle can be started and its state can be verified.

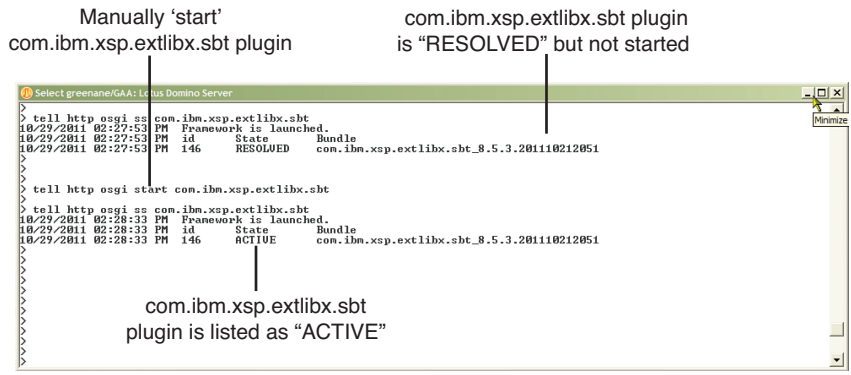


Figure 3.6 Result of running the start and ss commands in the console

stop <bundle-symbolic-name>

This command tells the platform to stop the specified bundle. Users should be careful when calling this on a production environment. In some cases, it might not be possible for the platform to stop the bundle. If this is the case, the reason will be printed to the console.

Sample usage:

```
tell http osgi stop com.ibm.xsp.extlib.sbt
```

Figure 3.7 shows how running a combination of the `ss` and `stop` command stops a bundle and verifies its state.

b <bundle-symbolic-name>

This command prints all metadata relating to the specified bundle. The metadata includes imported packages, required bundles, exported packages, bundle location, and so on. This command is useful when the developer needs to quickly verify that the bundle loaded by the platform has the meta information that the developer believes it has.

Sample usage:

```
tell http osgi b com.ibm.xsp.extlib
```

Listing 3.12 shows a subset of the output from running the `b` command against a specified bundle.

Listing 3.12 Sample Result of Running the b Command Against a Specified Bundle

```
tell http osgi b com.ibm.xsp.extlib
09/09/2011 02:15:21 PM
update@../../../data/domino/workspace/applications/eclipse/plugins/com.
ibm.xsp.extlib_8.5.2.201107241628NTF.jar [117]
  09/09/2011 02:15:21 PM     Id=117, Status=<<LAZY>>     Data Root=C:\
Program Files\IBM\Lotus\Domino\data\domino\workspace\.config\
  org.eclipse.osgi\bundles\117\data
  09/09/2011 02:15:21 PM     No registered services.
  09/09/2011 02:15:21 PM     No services in use.
  09/09/2011 02:15:21 PM     Exported packages
  09/09/2011 02:15:21 PM       com.ibm.xsp.extlib.actions.client;
version="0.0.0"[exported]
09/09/2011 02:15:21 PM       com.ibm.xsp.extlib.actions.client.data;
version="0.0.0"[exported]
09/09/2011 02:15:21 PM       com.ibm.xsp.extlib.actions.client.dojo;
version="0.0.0"[exported]
09/09/2011 02:15:21 PM       com.ibm.xsp.extlib.actions.client.dojo.fx;
version="0.0.0"[exported]
09/09/2011 02:15:21 PM       com.ibm.xsp.extlib.actions.server;
version="0.0.0"[exported]
```

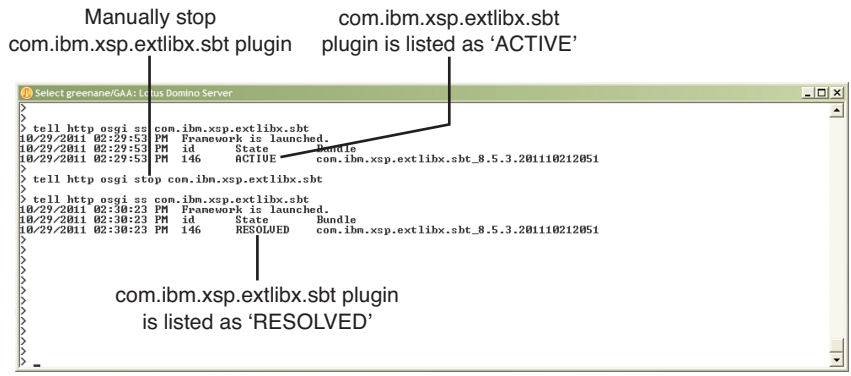


Figure 3.7 Result of running the start and ss commands in the console

headers <bundle-symbolic-name>

This command causes the OSGi header information for the specified bundle to be printed to the console. This command is convenient for checking information such as the packages that a specific bundle exports or the bundles that the specified bundle depends

upon. All the information stored in the bundle's `manifest.mf` file is printed to the console.

Sample usage:

```
tell http osgi headers com.ibm.xsp.extlib.sbt
```

Listing 3.13 shows the result of running the `headers` command on the Domino server console.

Listing 3.13 Sample Result of Running the headers Command with a Specified Bundle Name

```
tell http osgi headers com.ibm.xsp.extlib.sbt
  09/09/2011 04:34:52 PM  Bundle headers:
  09/09/2011 04:34:52 PM   Bundle-ClassPath = .,lib/httpclient-
4.0.1.jar,lib/httpcore-4.0.1.jar,lib/commons-codec-1.3.jar,lib/
oauth-20100527.jar,lib/
oauth-consumer-
20090617.jar,lib/oauth-consumer-20100527.jar,lib/oauth-httpclient4-
20090913.jar,lib/oauth-provider-20100527.jar
  09/09/2011 04:34:52 PM   Bundle-ManifestVersion = 2
  09/09/2011 04:34:52 PM   Bundle-Name = IBM Social Business Toolkit
09/09/2011 04:34:52 PM   Bundle-SymbolicName = com.ibm.xsp.extlib.
sbt;singleton:=true
  09/09/2011 04:34:52 PM   Bundle-Vendor = IBM
  09/09/2011 04:34:52 PM   Bundle-Version = 8.5.3.201108111413
  09/09/2011 04:34:52 PM   Export-Package =
com.ibm.xsp.extlib.fragment,com.ibm.xsp.extlib.model,com.ibm.xsp.extlib.
resources,com.ibm.xsp.extlib.sbt.activitystreams,com.ibm.xsp.extlib.sbt.
activitystreams.entry,com.ibm.xsp.
extlib.sbt.activitystreams.queue,com.ibm.xsp.extlib.sbt.connections,com.
ibm.xsp.extlib.sbt.connections.meta,com.ibm.xsp.extlib.security.
authorization,com.ibm.xsp.extlib.security
.authorization.beans,com.ibm.xsp.extlib.security.oauth_10a,com.ibm.xsp.
extlib.security.oauth_10a.servlet
  09/09/2011 04:34:52 PM   Fragment-Host = com.ibm.xsp.extlib
  09/09/2011 04:34:52 PM   Manifest-Version = 1.0
```

Listing 3.13 lists many different OSGi headers. You can find a full list of OSGi headers and their descriptions in the official OSGi specification: www.osgi.org/download/r4v43/r4.core.pdf.

help

This command tells the OSGi platform to print all commands that it supports, along with a short description of each command.

Sample usage:

```
tell http osgi help
```

Figure 3.8 shows the sample output from running the `help` OSGi command on the Domino server console.

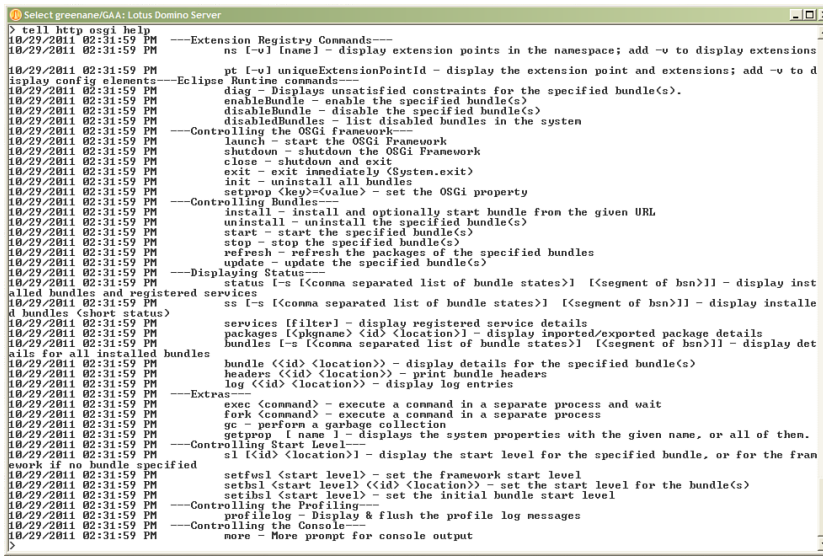

Select greenane/GAA: Lotus Domino Server
> tell http osgi help
10/29/2011 02:31:59 PM ---Extension Registry Commands---
10/29/2011 02:31:59 PM ns [-v] [name] - display extension points in the namespace; add -v to display extensions
10/29/2011 02:31:59 PM pt [-v] uniqueExtensionPointId - display the extension point and extensions; add -v to d
isplay config elements---Eclipse Runtime commands---
10/29/2011 02:31:59 PM diag - Displays unsatisfied constraints for the specified bundle(s).
10/29/2011 02:31:59 PM enableBundle - enable the specified bundle(s)
10/29/2011 02:31:59 PM disableBundle - disable the specified bundle(s)
10/29/2011 02:31:59 PM disabledBundles - list disabled bundles in the systen
10/29/2011 02:31:59 PM ---Controlling the OSGi Framework---
10/29/2011 02:31:59 PM launch - start the OSGi Framework
10/29/2011 02:31:59 PM shutdown - shutdown the OSGi Framework
10/29/2011 02:31:59 PM close - shutdown and exit
10/29/2011 02:31:59 PM exit - exit immediately (System.exit)
10/29/2011 02:31:59 PM init - uninstall all bundles
10/29/2011 02:31:59 PM setprop <key>=<value> - set the OSGi property
10/29/2011 02:31:59 PM ---Controlling Bundles---
10/29/2011 02:31:59 PM install - install and optionally start bundle from the given URL
10/29/2011 02:31:59 PM uninstall - uninstall the specified bundle(s)
10/29/2011 02:31:59 PM start - start the specified bundle(s)
10/29/2011 02:31:59 PM stop - stop the specified bundle(s)
10/29/2011 02:31:59 PM refresh - refresh the packages of the specified bundles
10/29/2011 02:31:59 PM update - update the specified bundle(s)
10/29/2011 02:31:59 PM ---Displaying Status---
10/29/2011 02:31:59 PM status [-s [<comma separated list of bundle states>] [<segment of bsn>]] - display inst
alled bundles and registered services
10/29/2011 02:31:59 PM ss [-s [<comma separated list of bundle states>] [<segment of bsn>]] - display installe
d bundles (short status)
10/29/2011 02:31:59 PM services [filter] - display registered service details
10/29/2011 02:31:59 PM packages [<pkgname> <id> <location>] - display imported/exported package details
10/29/2011 02:31:59 PM bundles [-s [<comma separated list of bundle states>] [<segment of bsn>]] - display det
ails for all installed bundles
10/29/2011 02:31:59 PM bundle <<id> <location>> - display details for the specified bundle(s)
10/29/2011 02:31:59 PM headers <<id> <location>> - print bundle headers
10/29/2011 02:31:59 PM log <<id> <location>> - display log entries
10/29/2011 02:31:59 PM ---Extras---
10/29/2011 02:31:59 PM exec <command> - execute a command in a separate process and wait
10/29/2011 02:31:59 PM fork <command> - execute a command in a separate process
10/29/2011 02:31:59 PM gc - perform a garbage collection
10/29/2011 02:31:59 PM getprop [name] - displays the system properties with the given name, or all of them.
10/29/2011 02:31:59 PM ---Controlling Start Level---
10/29/2011 02:31:59 PM sl [<id> <location>] - display the start level for the specified bundle, or for the fram
ework if no bundle specified
10/29/2011 02:31:59 PM setfwsl <start level> - set the framework start level
10/29/2011 02:31:59 PM setbsl <start level> <<id> <location>> - set the start level for the bundle(s)
10/29/2011 02:31:59 PM setibsl <start level> - set the initial bundle start level
10/29/2011 02:31:59 PM ---Controlling the Profiling---
10/29/2011 02:31:59 PM profilelog - Display & flush the profile log messages
10/29/2011 02:31:59 PM ---Controlling the Console---
10/29/2011 02:31:59 PM more - More prompt for console output
>

Figure 3.8 Result of running the help command in the console

How to Launch Notes/Designer Along with the OSGi Console

As of release 8.0 of Lotus Notes and release 8.5.0 of Domino Designer, both applications have been built upon the Eclipse platform. Eclipse itself is built upon the OSGi platform. As of Notes 8.5.1, it is now possible to run XPages applications within the Notes client.

With the emergence of the official XPages extension APIs in Notes/Domino 8.5.2 and the powerful functionality delivered as extensions to XPages (such as the XPages Extension Library), it is becoming more likely that, over time, end users will have Extension Library plug-ins installed into the Notes client platform. Either this can occur directly as a result of the user manually installing them or the plug-ins may be autoprovisioned to the platform via policy directives. Undoubtedly, at some point, XPages developers will need to debug the Notes client to figure out why certain XPages applications or functionality is not working as expected. The first step in such debugging should almost always

be analysis to determine whether the extended plug-ins in question are actually installed and running on the Notes client. The most accurate way to determine whether a plug-in is installed and running within the Notes client (or Domino Designer) is through the use of the OSGi console.

All the commands previously discussed and documented are available both on the Domino server and on the Notes client (and Domino Designer). However, the OSGi console that runs with the Notes client is a pure OSGi console, so it is not necessary to enter the HTTP task prefix required on the Domino server console. In the case of the Notes client OSGi console, it is necessary only to type the actual OSGi command—for example:

```
diag com.ibm.xsp.core
```

as opposed to

```
tell http osgi diag com.ibm.xsp.core.
```

To display the OSGi console for the Notes client or Domino Designer, the user must launch Notes with some additional arguments that tell the core Notes code to launch the console in a separate window when the Notes client is launching.

To do this, the user must navigate to the Notes program directory in a DOS prompt and enter the following DOS command:

```
notes.exe –RPARAMS –console
```

The `RPARAMS` argument for Notes and Domino Designer signals to both programs that the user is entering arguments that are to be redirected to the Eclipse and OSGi runtime. It may be useful to create a new shortcut on your desktop that enables you to easily launch the OSGi Console with Notes or Domino Designer. To do this, simply copy your existing Notes or Domino Designer launch shortcut and modify the `Target` information as follows:

```
C:\Notes85\notes.exe -RPARAMS -console "=C:\Notes85\notes.ini"
```

Here, **C:\Notes85** is the location of your Notes program directory. All the remaining shortcut information should be the same as your existing Notes or Domino Designer shortcut, as shown in Figure 3.9.

Arguments after the –`RPARAMS` parameter are sent to the Eclipse and OSGi runtimes for processing. Users should be aware that closing the Notes OSGi console window directly is not supported and can cause undesired behavior, such as causing the Notes program to hang. All instances of Notes, Domino Designer, and Domino Administrator should be shut down before running this command. Figure 3.10 shows the OSGi console running with Notes.

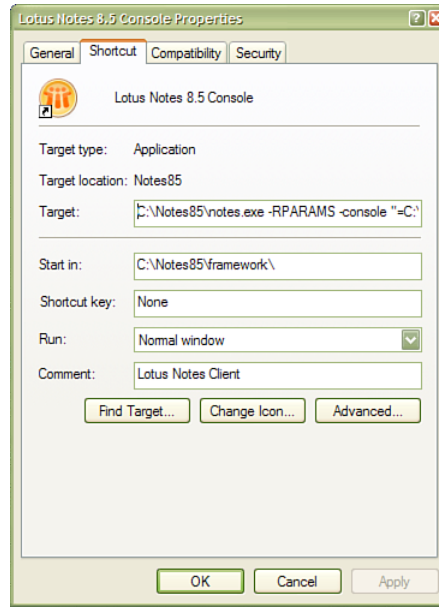


Figure 3.9 Shortcut to launch Notes with the OSGi console

Figure 3.10 Notes client running with the OSGi console

You can find more information on specific OSGi commands at these sites:

http://eclipse.org/equinox/

http://fusesource.com/docs/esb/4.1/command_ref/ESBosgi.html

Common Console Commands You Should Know

Beyond the realm of OSGi and the XSP command manager, the Domino server has a rich set of commands. Knowing at least a subset of them will greatly benefit any budding XPages developer or administrator. Table 3.4 lists some of the more commonly used commands.

Table 3.4 Common Domino Server Commands

Command	Description
help	Displays a list of server commands, with a brief description
load [task name]	Loads the named Domino server task
load [task name] -?	Gets help for the specified command
quit	Tells the Domino server to shut down
restart server	Tells the Domino server to shut down completely and restart
tell [task name] quit	Tells the named Domino server task to shut down
restart task [task name]	Tells the name Domino server task to restart
show server	Prints all basic statistics relating to the server to the console
show conf [notes.ini variable]	Prints the value of the server's **notes.ini** variable to the console
set conf [notes.ini variable=value]	Sets the value of the server's **notes.ini** variable to the specified value
tell adminp [options]	Performs various administrative tasks on the Domino server
load chronos [options]	Updates full-text indexes that are marked to be updated hourly or daily
load updall [path] [options]	Updates the view indexes and the full-text index for the specified database (or for all databases, if one is not provided)
load design [source] [target] [options]	Updates all databases with design updates from their master templates
load fixup [path] [options]	Locates and fixes corrupted databases on the server
show allports	Shows all enabled and disabled ports on the server
show diskspace	Displays the amount of free disk space on the server
show heartbeat	Displays a value if the server is responding

Command	Description
show memory	Displays the amount of RAM available on the server
show tasks	Displays the names of all the Domino server tasks running

You can obtain a much more extensive list of server commands by reading the Domino Administrator help, which is installed on the Domino server under the **help** directory.

help

This command displays a list of server console commands, with a brief description of each command, the command's arguments, and a sample of the syntax of each command.

Sample usage:

```
help
```

Figure 3.11 shows a subset of the sample output from running the `help` command on the Domino server console.

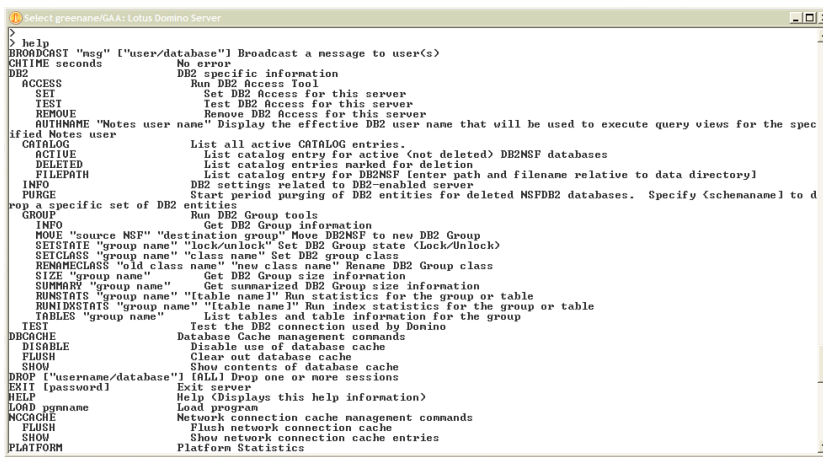


Figure 3.11 Result of running the help command on the Domino server console

load [task-name]

This command loads and starts the specified server task. It loads tasks that run continually until the server is stopped or loads a task that runs until complete. Further task arguments can be passed to the task as needed. This command is convenient because it

enables developers and administrators to dynamically start server tasks without needing to restart the entire server. For example, the HTTP task can be started without affecting other tasks running on the Domino server.

Sample usage:

```
load http
```

In this example, the HTTP task is loaded, allowing the Domino server to act as a HTTP server.

Listing 3.14 shows the console output of running the previous command.

Listing 3.14 Result of Running the load http Command on the Domino Server Console

```
> load http
09/19/2011 08:05:03 PM  HTTP Server: Using Web Configuration View
09/19/2011 08:05:07 PM  JVM: Java Virtual Machine initialized.
09/19/2011 08:05:07 PM  HTTP Server: Java Virtual Machine loaded
09/19/2011 08:05:07 PM  HTTP Server: DSAPI Domino Off-Line Services
HTTP extension Loaded successfully
09/19/2011 08:05:12 PM  XSP Command Manager initialized
09/19/2011 08:05:12 PM  HTTP Server: Started
```

load [task-name] -?

This command displays help information that relates to the task specified. In general, the help information lists any options or flags that can or should be passed to the task.

Sample usage:

```
load chronos -?
```

Listing 3.15 shows the sample output from running the help command against a specific task name.

Listing 3.15 Sample Output from Running the help Command Against the Chronos Task

```
> load chronos -?
>
Purpose:   Performs automatic hourly and daily full text indexing.
Usage:     Load CHRONOS [options]...
[options]:
hourly     Update all hourly full text indexes.
daily      Update all daily  full text indexes.
```

quit

This command stops the server. The server shuts down completely after running this command.

Sample usage:

```
quit
```

Figure 3.12 shows output from running the `quit` command on the Domino server console.

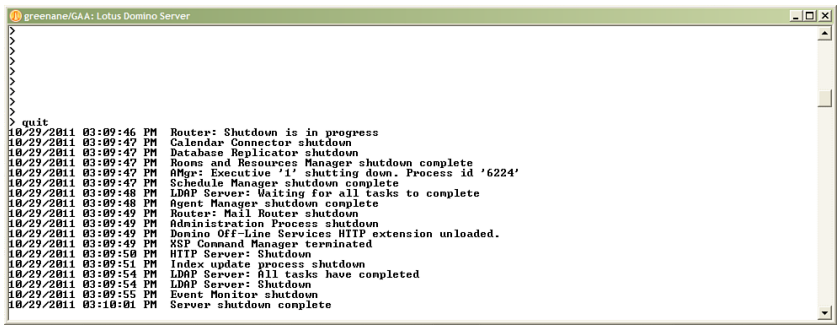


Figure 3.12 Result of running the quit command on the Domino server console

restart server

This command stops the server completely and then restarts the server after a brief delay.

Sample usage:

```
restart server
```

Figure 3.13 shows output from running the `restart server` command on the Domino server console.

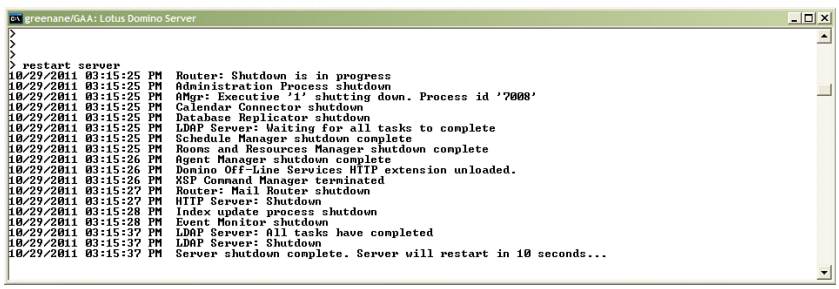


Figure 3.13 Result of running the restart server command on the Domino server console

tell [task-name] quit

This command stops the named task. All other server tasks remain in their current state.

Sample usage:

```
tell http quit
```

Listing 3.16 shows the sample console output after executing the quit command on a specific task.

Listing 3.16 Domino Server Console Output from Running the quit Command on the HTTP Task

```
> tell http quit
10/19/2011 08:50:21 PM  Domino Off-Line Services HTTP extension
unloaded.
10/19/2011 08:50:21 PM  XSP Command Manager terminated
10/19/2011 08:50:22 PM  HTTP Server: Shutdown
```

This sample terminates the HTTP task so that the Domino web server and all other HTTP functions are shut down. XPages developers might find this useful if the web server needs to be quickly and independently recycled—say, to reread and apply new XSP runtime settings.

restart task [task-name]

This command stops and restarts the named task. All other server tasks remain in their current state. XPages developers will find this to be a particularly powerful command because it enables them to completely and quickly restart the XPages runtime. This is of particular importance when debugging OSGi bundles running on the server. Chapter 6 discusses this in greater detail.

Sample usage:

```
restart task http
```

Listing 3.17 shows the Domino server output that results from restarting a specific task.

Listing 3.17 Sample Output from Running the restart task http Command

```
> restart task http
10/19/2011 09:03:10 PM  Domino Off-Line Services HTTP extension
unloaded.
10/19/2011 09:03:10 PM  XSP Command Manager terminated
10/19/2011 09:03:11 PM  HTTP Server: Shutdown
10/19/2011 09:03:13 PM  HTTP Server: Using Web Configuration View
10/19/2011 09:03:16 PM  JVM: Java Virtual Machine initialized.
10/19/2011 09:03:16 PM  HTTP Server: Java Virtual Machine loaded
10/19/2011 09:03:16 PM  HTTP Server: DSAPI Domino Off-Line Services
```

```
HTTP extension Loaded successfully
10/19/2011 09:03:19 PM  XSP Command Manager initialized
10/19/2011 09:03:19 PM  HTTP Server: Started
```

show server

This command prints all the basic information to the server's console, including (but not limited to) the server's name, data directory location, amount of time since the server was started, and total number of transactions completed by the server since it was started.

Sample usage:

```
show server
```

Listing 3.18 shows sample output from executing the `show server` command on the Domino server console.

Listing 3.18 Sample Output from the show server Command

```
> show server
  Lotus Domino (r) Server (Build V853_06302011 for Windows/32)
09/14/2011 07:28:42 PM
Server name:              greenane/GAA - Greenane
Domain name:              ibm
Server directory:         C:\Program Files\IBM\Lotus\Domino\data
Partition:                C.Program Files.IBM.Lotus.Domino.data
Elapsed time:             1 day 01:38:37
Transactions/minute:      Last minute: 10; Last hour: 200; Peak: 997
Peak # of sessions:       60 at 09/14/2011 06:50:06 PM
Transactions: 4524        Max. concurrent: 40
ThreadPool Threads:       40  (TCPIP Port)
Availability Index:       100 (state: AVAILABLE)
Mail Tracking:            Not Enabled
Mail Journalling:         Not Enabled
Number of Mailboxes:      10
Pending mail: 0           Dead mail: 0
Waiting Tasks:            0
DAOS:                     Not Enabled
Transactional Logging:    Not Enabled
Fault Recovery:           Not Enabled
Activity Logging:         Not Enabled
Server Controller:        Not Enabled
Diagnostic Directory:     C:\Program Files\IBM\Lotus\Domino\data\
➥IBM_TECHNICAL_SUPPORT
Console Logging:          Enabled (10240K)
Console Log File:         C:\Program Files\IBM\Lotus\Domino\data\
➥IBM_TECHNICAL_SUPPORT\console.log
DB2 Server:              Not Enabled
```

show conf [notes.ini variable]

This command enables the developer or administrator to examine the value of any given **notes.ini** variable without needing to physically open the **notes.ini** file residing in the Domino server's program directory. This is a powerful command because it allows developers and administrators alike to view the values of **notes.ini** variables that the runtime is using without needing to wade through the array of variables present in the Domino server's **notes.ini** file.

Sample usage:

```
show conf HTTPJVMMaxHeapSize
```

Listing 3.19 shows sample output as a result of executing the `show conf` command on the Domino server console.

Listing 3.19 Result of Executing the show conf Command Using the HTTPJVMMaxHeapSize Variable

```
> show conf HTTPJVMMAxHeapSize
HTTPJVMMAXHEAPSIZE=256M
```

set conf [notes.ini variable=value]

This command enables developers and administrators to quickly and easily set a **notes.ini** variable in the Domino server's **notes.ini** without actually physically opening the file and editing the value. This command is particularly useful because it enables users to set the **notes.ini** variable while the server is running. A typical use case for this command is one in which the administrator wants to increase the minimum Java heap size of the HTTP task's JVM without worrying about accidentally overwriting any other server settings that may have been written to **notes.ini** in the time the file was open for editing.

Sample usage:

```
set conf JavaMinHeapSize=64M
```

Figure 3.14 shows how the `JavaMinHeapSize notes.ini` variable can be reset using the `set conf` command and displays how the setting is applied by restarting the HTTP task.

tell adminp [options]

This command performs various automated administration tasks on the server. A wide range of options can be passed to this task; you can obtain the complete listing of `adminp` options from the Lotus Domino Administrator help, installed in the **help** directory of the Domino server.

Sample usage:

```
tell adminp show databases
```

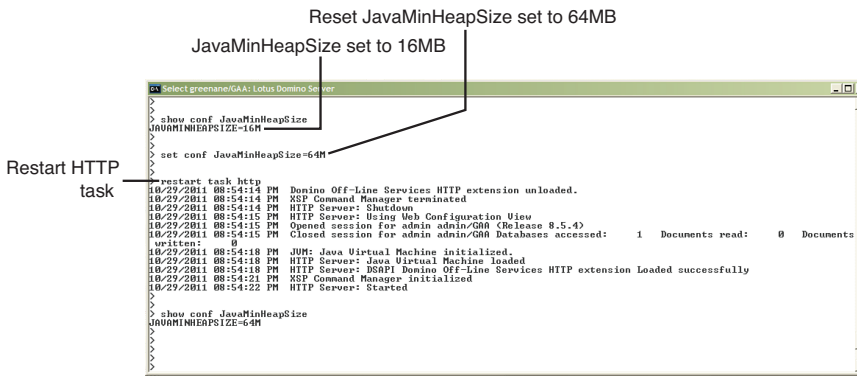


Figure 3.14 Result of running the set conf command on the JavaMinHeapSize notes.ini variable

Listing 3.20 shows the output from executing `adminp` with the `show databases` option specified.

Listing 3.20 Result of Executing the adminp Task on the Domino Server Console

```
> tell adminp show databases
10/20/2011 04:11:32 PM  Admin Process: These databases have greenane/
➥GAA designated as their Administration Server.
10/20/2011 04:11:32 PM  Title: Administration Requests Path: admin4.nsf
10/20/2011 04:11:32 PM  Title: CPP FreeBusy WebService Path:
➥cppfbws.nsf
10/20/2011 04:11:32 PM  Title: Domino Directory Cache (6) Path:
➥dbdirman.nsf
10/20/2011 04:11:32 PM  Title: Offline Services Path: doladmin.nsf
10/20/2011 04:11:32 PM  Title: greenane's Log Path: log.nsf
10/20/2011 04:11:32 PM  Title: admin admin Path: mail\aadmin.nsf
10/20/2011 04:11:32 PM  Title: Eileen Leonard Path: mail\eleonard.nsf
10/20/2011 04:11:32 PM  Title: Frank Adams Path: mail\fadams.nsf
```

load chronos [options]

This command loads the `chronos` task on the Domino server. The task is responsible for updating the full-text indexes of databases that are marked to be updated daily or hourly. This is useful to XPage developers when the full-text index of a database is needed to test particular functionality. This task enables developers to force the creation or update of the index without needing to modify the indexing schedule.

Sample usage:

```
load chronos hourly
```

Listing 3.21 shows the sample output from running the chronos task.

Listing 3.21 Sample Console Output from Running the chronos Task

```
>load chronos hourly
09/14/2011 08:35:06 PM  Chronos: Performing hourly full text indexing
09/14/2011 08:35:09 PM  Chronos: Full text indexer terminating
```

load updall [path] [options]

This command updates all changed views and/or all full-text indexes within the given database or all databases on the server. Obviously, this is quite useful if you are working with `FTSearch` features in your XPages application because testing and debugging requires an up-to-date full-text index.

You can pass a wide range of options to this task. The Lotus Domino Administrator help, installed in the **help** directory of the Domino Server, includes a complete listing of `adminp` options.

Sample usage:

```
load updall XPagesSBT.nsf –f
```

Listing 3.22 shows the output received on the Domino server console from running the `updall` task on the Domino server.

Listing 3.22 Sample Console Output from Running the updall Task to Update Full-Text Indexes on a Specified Application

```
> load updall XPagesSBT.nsf -f
09/14/2011 08:44:39 PM  Index update process started:  XPagesSBT.nsf -f
09/14/2011 08:44:39 PM  Updating views in C:\Program
➥Files\IBM\Lotus\Domino\data\XPagesSBT.nsf
09/14/2011 08:44:39 PM  Index update process shutdown
```

load design [source] [target] [options]

This command updates all databases on the server with design updates from their master template. This command can be quite useful when an administrator has accidentally modified the design of a particular database and needs to update the design of that database from the master template outside the regular design update schedule.

Sample usage:

```
load design rossacussane.swg.myco.com greenane.swg.myco.com –f
➥XPagesSBT.nsf
```

Listing 3.23 shows the Domino server console output received from executing the design task on the Domino server.

Listing 3.23 Sample Console Output from Running the Design Task

```
> load design rossacussane.swg.myco.com greenane.swg.myco.com
➥-f XPagesSBT.nsf
09/14/2011 08:54:52 PM  Database Designer started
09/14/2011 08:54:52 PM  Opened session for rossacussane/GAA (Release
➥8.5.3)
09/14/2011 08:54:55 PM  Closed session for rossacussane/GAA Databases
accessed: 3   Documents read: 0   Documents written:  0
09/14/2011 08:54:55 PM  Opened session for greenane/GAA (Release 8.5.3)
➥09/14/2011
08:54:55 PM  Closed session for greenane/GAA Databases accessed:
1   Documents read: 0
Documents written:  0
09/14/2011 08:54:55 PM  Opened session for greenane/GAA (Release 8.5.3)
09/14/2011 08:54:55 PM  Database Designer shutdown
09/14/2011 08:54:55 PM  Closed session for greenane/GAA Databases
➥accessed: 1
Documents read: 0   Documents written:  0
```

load fixup [path] [options]

This command runs the `fixup` task on the specified database or on all databases on the server. The `fixup` task scans for databases that contain inconsistencies from partially written operations that may have occurred during a previous failure, such as a hardware failure or a crash. You can pass a wide range of options to this task. The complete listing of `adminp` options is available from the Lotus Domino Administrator help, installed in the **help** directory of the Domino server.

Sample usage:

```
load fixup XPagesSBT.nsf -l
```

Listing 3.24 shows the result of running the `fixup` command against a particular database on the Domino server.

Listing 3.24 Sample Console Output from Running the fixup Command

```
> load fixup XPagesSBT.nsf -l
 09/14/2011 09:08:55 PM  Database Fixup: Started:  XPagesSBT.nsf -l
 09/14/2011 09:08:55 PM  Checking database C:\Program Files\IBM\
➥Lotus\Domino\data\XPagesSBT.nsf
 09/14/2011 09:08:55 PM  Performing consistency check on
➥XPagesSBT.nsf...
 09/14/2011 09:08:56 PM  Completed consistency check on XPagesSBT.nsf
 09/14/2011 09:08:56 PM  Performing consistency check on views in
➥database XPagesSBT.nsf
```

```
 09/14/2011 09:08:56 PM  Completed consistency check on views in
➥database XPagesSBT.nsf
 09/14/2011 09:08:56 PM  Database Fixup: Shutdown
```

show tasks

This command shows the names of all the Domino Server tasks that are running on the server. Administrators will find this useful for determining which tasks are running on any given server.

Sample usage:

```
show tasks
```

Figure 3.15 shows the sample output received when running the `show tasks` command on a Domino server.

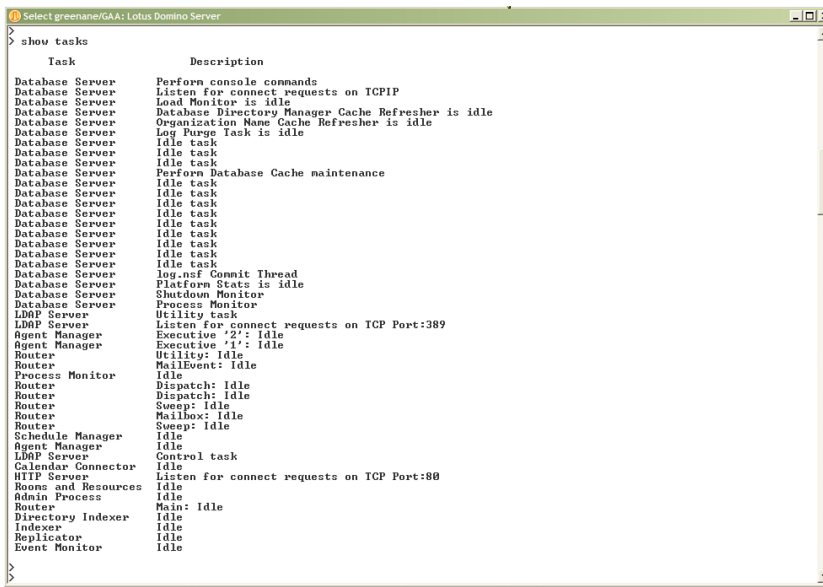


Figure 3.15 Result of running the show tasks command on the Domino server console

show allports

This command prints the configuration of all enabled and disabled ports on the server.

Sample usage:

```
show allports
```

Listing 3.25 shows the result of executing the `show allports` command on the Domino server console.

Listing 3.25 Sample Console Output from Running the show allports Command

```
> show allports
Enabled Ports:
TCPIP=TCP, 0, 15, 0
Disabled Ports:
LAN0=NETBIOS, 0, 15, 0
LAN1=NETBIOS, 1, 15, 0
LAN2=NETBIOS, 2, 15, 0
LAN3=NETBIOS, 3, 15, 0
LAN4=NETBIOS, 4, 15, 0
LAN5=NETBIOS, 5, 15, 0
LAN6=NETBIOS, 6, 15, 0
LAN7=NETBIOS, 7, 15, 0
LAN8=NETBIOS, 8, 15, 0
```

show diskspace

This command prints the amount of disk space available on the server.

Sample usage:

```
show diskspace
```

Listing 3.26 displays the results from executing the `show diskspace` command.

Listing 3.26 Sample Console Output from the show diskspace Command

```
> show diskspace
Available disk space 83,342,319,616 bytes
```

show heartbeat

This command prints a value to the console if the server is still responding.

Sample usage:

```
show heartbeat
```

Listing 3.27 shows the result of running the show `heartbeat` command on the Domino server console.

Listing 3.27 Sample Console Output for the show heartbeat Command

```
> show heartbeat
greenane/GAA's elapsed time: 100827 seconds
```

Conclusion

This chapter outlined the most relevant commands available to you as an XPages developer via the Domino server console and the Notes OSGi console. Over time, these commands will undoubtedly prove to be powerful tools in the resolution of issues. Although executing the commands is a relatively simple exercise, the result they yield will often lead you directly to the source of a problem. These commands will also improve productivity by reducing the amount of time needed to test an application. For example, scheduled tasks, such as indexing operations, can be run on demand using these commands, without having to wait for tasks to execute on schedule. Make the most of them!

CHAPTER 4

Working with the XSP Client Side JavaScript Object

All XPages applications execute within a J2EE-compliant web container and are rendered to the end user via a web browser. This is an example of a classic client/server application architecture, where everything that executes within the web application container is server side, while everything that displays and executes within the web browser container is client side. This chapter is exclusively concerned with the latter—specifically, with describing the tools that are at your disposal for manipulating the client side of the model.

Over the past decade, particularly with the advent of Web 2.0, the user experience within web applications has become progressively richer. To a large extent, this is made possible through advances in Client Side JavaScript and CSS. In terms of Client Side JavaScript, the XPages runtime enables you to add your own custom client-side script code to your applications. This could be anything from a simple inline expression to an arbitrarily complex library of JavaScript code. In addition to this, XPages also provides powerful Client Side JavaScript objects that you can directly leverage to build a richer front-end for your applications. The first of these is a home-grown creation called the XSP Client Side JavaScript object. The other is the Dojo framework, an open-source modular JavaScript toolkit that facilitates rapid development of rich browser-based applications. This chapter discusses both in depth, starting with the XSP Client Side JavaScript object.

Before you dive in, be sure to download `PCGCH04.nsf` and open it in Domino Designer so that you have all the examples covered here. As usual, the sample application is available from this website: www.ibmpressbooks.com/title/0132943050

What Is the XSP Client Side JavaScript Object?

In Notes/Domino releases prior to version 8.5.3, you can locate the XSP Client Side JavaScript object by simply navigating to the following subfolder from the Notes client or Domino server data folder:

```
domino\js\dojo-1.x.x\ibm\xsp\widget\layout
```

In this file path, `1.x.x` denotes the version number of the Dojo Toolkit that ships with a particular Notes/Domino release. In Notes/Domino 8.5.3, things are slightly different: The aforementioned directory path still exists and is based specifically on Dojo 1.5.1, but the XPages runtime does *not* use it by default (other components, such as Domino iNotes, use it). Instead, XPages provides an OSGi plug-in that contains the Dojo 1.6.1 library. The XPages runtime uses this as the default Dojo library in Notes/Domino 8.5.3 unless otherwise specified by the `xsp.client.script.dojo.version` property, as described previously in Chapter 1, “Working with XSP Properties.” This plug-in is

packaged as a JAR file and located in a subfolder under the Notes/Domino root installation folder. In Domino, the location is like this:

```
osgi\shared\eclipse\plugins\com.ibm.xsp.dojo_8.5.3.yyyymmdd-hhmm.jar
```

Here, `yyyymmdd-hhmm` represents a specific time stamp version of the plug-in.

If you open the JAR archive file and navigate to the following inner path, you will find all the XPages runtime Client Side JavaScript resources for version 8.5.3:

```
resources\dojo-version\ibm\xsp\widget\layout
```

Within this plug-in directory are numerous JavaScript files, including the set of `xspClient*.*` files shown in Listing 4.1. However, it might be simpler just to look at these resources on the file system in the original location; for the purposes of this chapter, it makes no difference to the material discussed.

Listing 4.1 XSP Client Side JavaScript Modules Provided by XPages

```
xspClientCA.js
xspClientCA.js.gz
xspClientCA.js.uncompressed.js
xspClientDebug.js
xspClientDebug.js.gz
xspClientDebug.js.uncompressed.js
xspClientDojo.js
xspClientDojo.js.gz
xspClientDojo.js.uncompressed.js
xspClientDojoUI.js
xspClientDojoUI.js.gz
xspClientDojoUI.js.uncompressed.js
xspClientLite.js
xspClientLite.js.gz
xspClientLite.js.uncompressed.js
xspClientMashup.js
xspClientMashup.js.gz
xspClientMashup.js.uncompressed.js
xspClientRCP.js
xspClientRCP.js.gz
xspClientRCP.js.uncompressed.js
```

No doubt you observe that every XSP client-side object comes in three flavors. The plain `.js` version contains obfuscated JavaScript code, with the file size reduced by placing the entire code contents on one line, minimizing variable names and so forth. The `.js.gz` version is a **gzipped** version of the same file (compressed using the GNU zip algorithm) that provides a minimized payload for browsers that can accept zipped

content. The third version, `.js.uncompressed.js`, contains the fully formatted text resource, which is a more human-readable form of the JavaScript code.

> **TIP** If you are using a version of Notes earlier than 8.5.3, or if in 8.5.3 you have explicitly chosen to use a Dojo version earlier than 1.6.1, you can debug an XSP client-side object in the following way. You should back up and remove the `.js` and `.js.gz` versions of the particular JavaScript object from the folder and then rename the `.js.uncompressed.js` version to the `.js` version. This enables you to step through the fully formatted JavaScript code in your chosen JavaScript client-side debugger.

Ignoring the three differing flavors for now, Table 4.1 summarizes the single logical entities.

Table 4.1 XSP Client Side JavaScript Object

Entity Name	Description
xspClientCA	An extension to the base XSP client-side object containing JavaScript functions that are useful when an XPage is part of a Notes composite application (CA).
xspClientDebug	An XSP client-side object containing JavaScript debugging functions for logging errors and dumping diagnostic information.
xspClientDojo	The root XSP Client Side JavaScript object containing a wide range of utility functions, such as validators and event handlers.
xspClientDojoUI	An XSP Client Side JavaScript object that simply includes the core Dojo UI modules as one simple resource reference.
xspClientLite	An XSP client-side object intended for use as a lightweight client-side framework in the absence of Dojo or other frameworks. It is not automatically included in the XPages markup in any use case by the runtime code and does not appear to be commonly used in the field. This object is still included in XPages, for backward compatibility as much as for anything else.
xspClientMashup	An extension to the base XSP client-side object containing JavaScript functions that are useful when an XPage is part of a web mashup application. For example, the object contains functions that facilitate intercomponent communication.
xspClientRCP	An extension to the base XSP client-side object containing JavaScript functions that are useful when an XPage is running in the Notes client, also known as the Rich Client Platform (RCP).

As you can observe from Table 1.1, no single monolithic XSP client-side object exists. Instead, a number of them are dynamically assembled by the XPages runtime for your particular runtime context. For example, if you are running an XPages composite application in the Notes client, the runtime provides instances of both the `xspClientCA`

object and the `xspClientRPC` object on your XPage, whereas a plain old XPiNC application (XPages in the Notes Client) is rendered with only the `xspClientRCP` object. A simple example illustrates the point clearly.

1. In Domino Designer, create a new XPage.
2. Do not add any content; leave the page empty.
3. Turn off JavaScript aggregation by unchecking the **Application Properties > XPages > Use runtime optimized JavaScript and CSS resources** option.
4. Preview the page both on the Notes client *and* on the web.
5. View and compare the HTML source for both pages.

Even though the XPage is just a skeleton, XPages must provide the bare-bones page structure appropriate for each runtime environment (Notes and web). This includes setting up the necessary XSP Client Side JavaScript objects. Listing 4.2 shows the `<script>` tags that are included in the XPage on the Notes client. If you compare this to the markup emitted for the web, you will observe that the second tag containing the `xspClientRCP` JavaScript object is not present.

Listing 4.2 HTML Script Resources Included for a Blank XPage on the Notes Client

```
<script type="text/javascript" src="/xsp/.ibmxspres/dojoroot-1.6.1/ibm/
xsp/widget/layout/layers/xspClientDojo.js">
</script>

<script type="text/javascript">
    dojo.require('ibm.xsp.widget.layout.xspClientRCP')
</script>
```

As a next step, open the **xspClientRCP.js.uncompressed.js** file and examine its content. Some interesting points will help you understand how the collection of different **xsp*.js** files conflate together, depending on runtime platform, to form a concrete XSP Client Side JavaScript object. Listing 4.3 contains two snippets.

Listing 4.3 JavaScript Snippets from xspClientRCP.js.uncompressed.js

```
////////////////////////////////////
// Display an alert
////////////////////////////////////

     XSP.alert=function x_al(s)
     {
          var o = new Object();
          if( s == null )s = "null";
          o.text = s;
          return XSP.callJavaAction( "XSP.alert", o, true );
```

```
      };

// ... lots of other intervening code ...

if(typeof XSP.RCPConstructor == "undefined"){
      XSPRCPConstructor.call(XSP);
}
```

As you can see, an `XSP.alert` function is defined in the **RCP** JavaScript file, which takes a string input argument and ultimately passes it to the XPages runtime via a JavaScript-to-Java messaging bridge. This enables the XPages runtime plug-in for Notes to display a native RCP dialog, thus providing a more natural user experience for that platform.

You will also note that, at the end of this file, the `xspClientRCP` object specifies a constructor function. This constructor ensures a single instantiation of the object and extends the XSP Client Side JavaScript object by way of the built-in JavaScript `.call()` method. This enables the `xspClientRCP` object to extend the base XSP Client Side JavaScript object. As shown in Listing 4.2, the base XSP object is created immediately prior to `xspClientRCP` because of the preceding inclusion of `xspClientDojo` in the HTML markup—sequential loading ensures this preinclusion. However, this base object already defines an `alert` function, as shown in Listing 4.4.

Listing 4.4 JavaScript Snippet from xspClientDojo.js

```
////////////////////////////////////
      this.alert = function x_al(s) {
            // Use the browser alert mechanism
            alert(s);
      }
```

Thus, when running on the web, an `XSP.alert("Hello World!")` call is displayed in the browser's dialog, whereas the same code in Notes results in a native Notes RCP dialog (and not the standard embedded XULRunner browser dialog, which would just look wrong in this context). Although this is a simple example, it illustrates how the XSP object is, in fact, an aggregated class definition that is overridden and extended as necessary, depending on the running platform. Figure 4.1 illustrates the hierarchy of XSP client-side objects. Note that `xspClientDojoUI.js` and `xspClientDebug.js` are standalone objects that do not extend the root XSP object. The former is simply a Dojo layer file (groups a given collection of Dojo files for inclusion), whereas the latter contains some discrete debugging and logging functions.

Note that if you are running a composite application on the client, *both* `xspClientRCP.js` and `xspClientCA.js` will be included in your XPiNC pages. Also, `xspClientCA.js` and `xspClientMashup.js` have functions in common `(XSP.publishEvent())` that are designed to do the same task in different enviroments—composite applications versus web mashups.

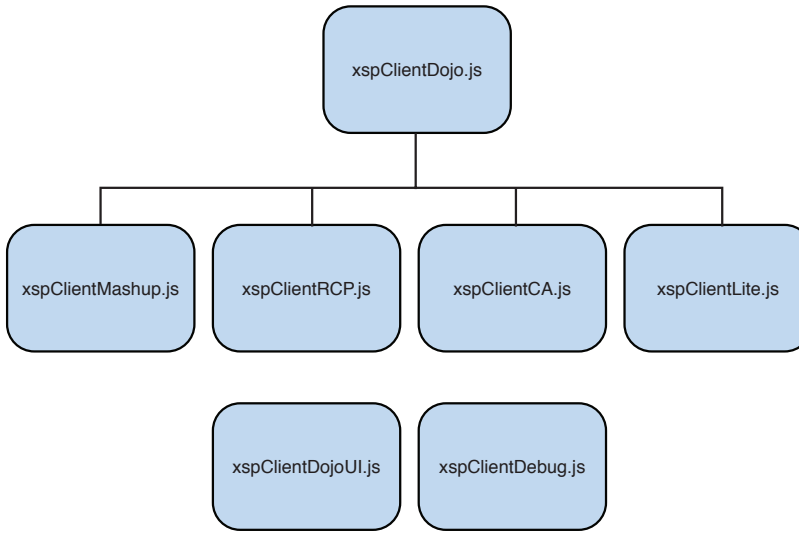


Figure 4.1 XSP client-side object hierarchy

Finally, because the XPages runtime automatically includes the required XSP Client Side JavaScript object on the rendered page for you, you can use a Client Side JavaScript debugger to quickly determine what functions are at your disposal. Figure 4.2 shows an example of this using the empty XPage discussed earlier, performed on the web using the Firebug debugger in Firefox.

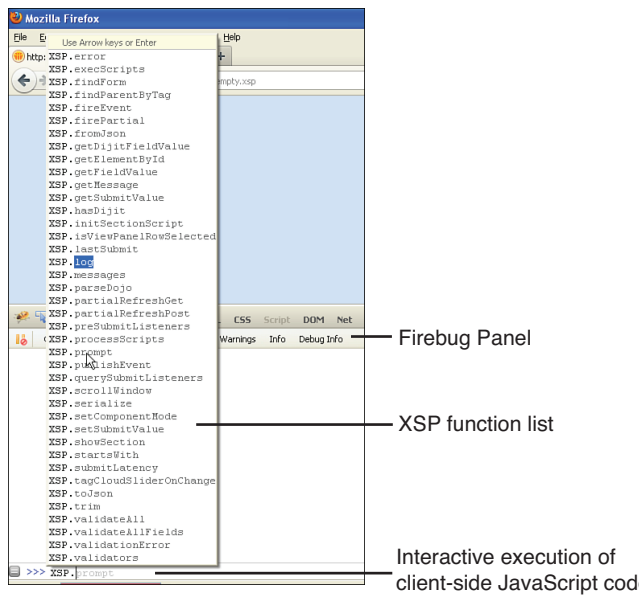


Figure 4.2 Using a Client Side JavaScript debugger to expose the XSP object functions

In the bottom-left corner of the Firebug debugger, you can interactively call any Client Side JavaScript on the page using the command editor, with the added benefit of function type-ahead (simply type the characters `XSP.` and pause to be prompted with a list of suggestions). All the available XSP Client Side JavaScript functions are thus exposed (other Client Side JavaScript debuggers should offer similar capabilities).

Summary of the XSP Client Side JavaScript Object Functions

This section aims to give you a summarized view of the XSP Client Side JavaScript object functions. It is important to realize that a set of private functions is declared on the XSP Client Side JavaScript object. These are typically denoted by a leading underscore character in the function name (for example, `XSP._doFireEvent`), but some previous releases did not apply this naming convention. Such private functions could not subsequently be renamed because doing so would potentially break backward compatibility—that is, they could not be prefixed in a later release with an underscore character to denote their private scope.

This set of private functions is not intended for reuse within any custom Client Side JavaScript you write. This is due to the very nature of their function, which typically is to provide some direct XPages runtime execution service. Therefore, trying to reuse these functions within any custom Client Side JavaScript code you write only complicates your task; you must provide specific parameters and ensure correct invocation within a sequence of other function calls. Furthermore, the server-side component tree representation of the associated XPage being viewed in the client-side browser or device must also be in a certain state to handle such private function requests correctly. You are therefore discouraged from trying to use these private functions within your own custom Client Side JavaScript code. Nonetheless, it will still be helpful for you to understand the purpose of such functions.

It is also important to point out the fact that the list of public XSP Client Side JavaScript object functions detailed in this chapter can change through subsequent Notes/Domino releases. This is further influenced based on any modifications or extensions of the XSP Client Side JavaScript object that the XPages Extension Library provides. The XPages Extension Library has its own set of extended XSP Client Side JavaScript object functions and overrides. However, we try to accommodate changes to the XSP Client Side JavaScript object API during its evolution by providing backward compatibility and deprecation mechanisms accordingly.

Thus, this section gives you summary information about *all* the XSP Client Side JavaScript object functions, both those that will serve you well within your own custom code and those that are within the private set. Table 4.2 summarizes the public XSP Client Side JavaScript object functions, and Table 4.3 details the private XSP Client Side JavaScript object functions. Both these tables list each function by name, along with a brief description and indication of the scope of each function. You should refer to these tables when considering using a particular XSP Client Side JavaScript object function within your own custom Client Side JavaScript code.

Recall from reading the previous section "What Is the XSP Client Side JavaScript Object?" that there is a hierarchy and extension tree for the XSP Client Side JavaScript object, depending on the running platform. Therefore, Tables 4.2 and 4.3 list all public and private XSP Client Side JavaScript object functions from the XSP web, mobile, Notes, Composite Application, IBM Mashup Center, and Debug extensions (IBM Mashup Center is not part of the Notes/Domino product family, but is an IBM WebSphere offering). The first column of each table gives specifics about availability within each platform. Where a function is available within all platforms, it is specified within square brackets as **All**. Otherwise, the available platforms are listed individually. Supported platforms are **Web**, **Mobile**, **Notes**, **CA** (denoting composite application), and **MU** (denoting IBM Mashup Center). The description column also includes the name of the source JavaScript file, for your convenience.

Table 4.2 Summary of the Public XSP Client Side JavaScript Object Functions

Function Name	Description
XSP.alert(message) [All]	Displays a generic alert dialog. Standard dialog in a web browser/mobile device, and RCP dialog in the Notes client. This is an overridable extension point. [xspClientDojo.js]
XSP.confirm(message) [All]	Displays a generic confirm dialog, with OK and Cancel buttons. Standard dialog in a web browser/mobile device, and RCP dialog in the Notes client. This is an overridable extension point. [xspClientDojo.js]
XSP.error(message) [All]	Displays a generic error dialog. This is an overridable extension point. [xspClientDojo.js]
XSP.prompt(message, defaultCaption) [All]	Displays a generic prompt dialog containing the given message, along with the `defaultCaption`, if specified. The `defaultCaption` value appears within the prompt dialog's edit box, enabling the user to input a value that returns when the dialog is closed. This is an overridable extension point. [xspClientDojo.js]
XSP.djRequire(name) [All]	Loads a Dojo module into the context of the current page. Use this only when you do not want to include the required module in aggregated resources. Otherwise, use `dojo.require()`. [xspClientDojo.js]

Function Name	Description
XSP.addPreSubmit Listener(formId, listener, clientId, scriptId) [All]	This function adds a custom Client Side JavaScript function, termed the listener, to a queue of zero or more other listeners. This queue of listeners gets executed just before the page is submitted to the server side. The `formId` parameter must specify the ID of the current form the listener will be triggered against on submission—this is a mandatory parameter. The `listener` parameter is a function reference to a custom Client Side JavaScript function—this is a mandatory parameter. The `clientId` parameter can specify the client-side fully namespaced ID of any single `eventHandler` or button control within the current page. This causes the presubmit listener to trigger only when the specified `clientId eventHandler` or button is invoked. This parameter is optional but should at least be specified as `null`. When set to `null`, the presubmit listener is triggered when any `eventHandler` or button tries to submit the current page. Also note that the custom client-side listener function does not need to return any result because this is ignored in the processing of the presubmit listener queue. [xspClientDojo.js]
XSP.addQuerySubmit Listener(formId, listener, clientId, scriptId) [All]	Adds a custom Client Side JavaScript function, termed the listener, to a queue of zero or more other listeners. This queue of listeners is executed when the page tries to submit to the server side. Based on the return results from any `querysubmit` listener functions in the queue, submission can proceed or be stopped. The `formId` parameter must specify the ID of the current form the listener will be triggered against on submission—this is a mandatory parameter. The `listener` parameter is a function reference to a custom Client Side JavaScript function—this is a mandatory parameter. The `clientId` parameter can specify the client-side fully namespaced ID of any single `eventHandler` or button control within the current page. This causes the `querysubmit` listener to trigger only when the specified `clientId eventHandler` or button is invoked. This parameter is optional but should at least be specified as `null`. When set to `null`, the `querysubmit` listener is triggered when any `eventHandler` or button tries to submit the current page. Also note that the custom client-side listener function should return a Boolean result because this is used in the processing of the `querysubmit` listener queue to either allow or stop page submission. [xspClientDojo.js]

Table 4.2 Summary of the Public XSP Client Side JavaScript Object Functions (cont'd)

Function Name	Description
XSP.canSubmit() [All]	Should be called before any page submission, to prevent page resubmission, either accidentally when a user double-clicks on a link or if the user is impatient compared to the expected time for a page refresh. If the page has recently been submitted, it returns `false` and the submission should be abandoned. Otherwise, it assumes that the submission will occur and updates the `lastSubmit` value, the date stamp of the last submission. If the page is not submitted after this is called, `XSP.allowSubmit()` should be invoked so that further user actions can submit the page. The time allowed between submissions is configured through the `submitLatency` variable. Avoid using this function—rare use cases exist when you need to use this function, such as if Client Side JavaScript performs page submission, as with `document.forms[0].submit()`. Therefore, controlling page submission using `XSP.canSubmit()` can provide more robust page behavior. [xspClientDojo.js]
XSP.allowSubmit() [All]	If the page is not submitted after a call to `canSubmit()`, this should be invoked to re-enable page submission. Avoid using this function—rare use cases exist when you need to use this function, such as if Client Side JavaScript performs page submission, as with `document.forms[0].submit()`. Therefore, controlling page submission using `XSP.allowSubmit()` can provide more robust page behavior. [xspClientDojo.js]
XSP.setSubmitValue (submitValue) [All]	The `SubmitValue` property is the value sent to the XPages server and is available from the context object. This function usually can be called while processing a client-side event, right before the event is submitted to the server. It can be used, for example, for passing component-related data. [xspClientDojo.js]
XSP.getSubmitValue() [All]	The `SubmitValue` property is the value sent to the XPages server and is available from the context object. This function can be called while processing a client-side event, usually right before the event is submitted to the server. [xspClientDojo.js]

Function Name	Description
XSP.validateAll(formId, valmode, execId) [All]	Runs client-side converters and validators before page submission to the server. The `formId` parameter must specify the form to validate. This is useful in a multiform page. The `valmode` integer parameter must specify a value of `0`, `1`, or `2`. `0` = No validation. `1` = Run converters only. `2` = Run Converters and Validators. The `execId` string parameter is optional and can be used to specify a single control within the current page to validate. If `null`, the whole page is validated. [xspClientDojo.js]
XSP.getFieldValue(node) [All]	Returns the value of the given HTML DOM node parameter. A string value is returned for a single value node, or a comma-separated string is returned for a multiple-value node (such as an option control). [xspClientDojo.js]
XSP.getDijitFieldValue(dj) [All]	Returns the value of the given Dijit instance, based on the existence of the `dijit.getValue()` function. [xspClientDojo.js]
XSP.validationError(clientId, message) [All]	An overridable extension point similar to the `XSP.alert()` and other dialog-based functions. This gives developers an opportunity to provide a custom error display to the end user. The default behavior is to display an error message dialog. The `clientId` string parameter must specify the ID of the failing control. The `message` string parameter is used to relay the failing message or warning. [xspClientDojo.js]
XSP.scrollWindow(x, y) [All]	Scrolls the current window contents to the specified x and y integer coordinates. [xspClientDojo.js]

Table 4.2 Summary of the Public XSP Client Side JavaScript Object Functions (cont'd)

Function Name	Description
XSP.partialRefreshGet(refreshId, options) [All]	Programmatically executes `GET`-based partial refresh requests to the XPages server-side runtime. The `refreshId` string parameter value must specify a fully namespaced HTML DOM element ID. This is the target receiver of the partial refresh response content. The `options` parameter is optional and used to send custom parameters to the server side as `GET` request parameters. It can also be used to specify `onStart`, `onError`, and `onComplete` event function callbacks. These are triggered accordingly during the request lifecycle. [xspClientDojo.js]
XSP.partialRefreshPost(refreshId, options) [All]	Programmatically executes `POST`-based partial refresh requests to the XPages server-side runtime. The `refreshId` string parameter value must specify a fully namespaced HTML DOM element ID. This is the target receiver of the partial refresh response content. The `options` parameter is optional and used to send custom parameters to the server side as `POST` request parameters. It can also be used to specify `onStart`, `onError`, and `onComplete` event function callbacks. These are triggered accordingly during the request lifecycle. An `immediate` request parameter can also be included in the options content to control validation execution. [xspClientDojo.js]
XSP.attachClientFunction (targetClientId, _event, clientSideScriptName) [All]	Connects a client-side function to an event on an XPages control. [xspClientDojo.js]
XSP.attachClientScript(target ClientId, _event, clientScript) [All]	This function is used to connect a client-side script call to an event on an XPages control. [xspClientDojo.js]
XSP.addOnLoad(listener) [All]	Attaches a Client Side JavaScript function to the current page's `onLoad` event. The `listener` function-reference parameter is used to specify the Client Side JavaScript function. [xspClientDojo.js]
XSP.showSection(sectionId, show) [All]	Toggles the expanded state of the specified section control using the client-side `sectionId` string parameter value and `show` Boolean parameter value. [xspClientDojo.js]

Function Name	Description
XSP.findForm(nodeOrId) [All]	Returns the parent form element for the given node or client-side element ID. [xspClientDojo.js]
XSP.findParentByTag(nodeOrId, tag) [All]	Returns the nearest parent element matching the specified `tag` string parameter value. [xspClientDojo.js]
XSP.getElementById(elementId) [All]	Retrieves an element from the current HTML DOM based on the specified `elementId` string parameter value. Note that the `elementId` represents the fully namespaced client-side element ID. [xspClientDojo.js]
XSP.hasDijit() [All]	Determines the presence of any Dijit objects and the `dijit.byId()` function within the current HTML page. [xspClientDojo.js]
XSP.trim(s) [All]	Returns the `s` string parameter value, trimmed of leading and trailing whitespace. [xspClientDojo.js]
XSP.startsWith(s, prefix) [All]	Returns a Boolean value indicating whether the given `s` string begins with the specified prefix string value. [xspClientDojo.js]
XSP.endsWith(s, suffix) [All]	Returns a Boolean value indicating whether the given `s` string ends with the specified suffix string value. [xspClientDojo.js]
XSP.toJson(o) [All]	Converts an object to a String serialization of that object. [xspClientDojo.js]
XSP.fromJson(s) [All]	Parses a JSON string to return a JavaScript object. [xspClientDojo.js]
XSP.log(message) [Web/MU]	Opens a new browser window, with the given message parameter value written into the window contents. [xspClientDojo.js]

Table 4.2 Summary of the Public XSP Client Side JavaScript Object Functions (cont'd)

Function Name	Description
XSP.dumpObject(obj, options) [Web/MU]	Returns a list of property/value pairs available on the given `obj` parameter. [xspClientDebug.js]

Table 4.3 Summary of the Private XSP Client Side JavaScript Object Functions

Function Name	Description
XSP.getMessage(key) [All]	Retrieves a translated string from the localized XSP client-side locale bundles. [xspClientDojo.js]
XSP._pushListener(listeners, formId, clientId, scriptId, listener) [All]	Used internally by the `XSP.addPreSubmitListener` and `addQuerySubmitListener` functions and others. [xspClientDojo.js]
XSP._SubmitListener(formId, listener, clientId, scriptId) [All]	Used internally by the `XSP.addPreSubmitListener` and `addQuerySubmitListener` functions and others. [xspClientDojo.js]
XSP._processListeners(listeners, formId, clientId) [All]	Processes an array of listeners, either the `querySubmit` or `preSubmit` listeners. When processing the `querySubmit` listeners, it stops at the first listener that returns `false`. [xspClientDojo.js]
XSP.attachValidator(clientId, required, converter, validator1, ..., multipleValueSeparatorString)	Connects a control with any converter and validation objects specified for that control. Calls to this function are automatically generated by the XPages runtime. This is a private function. [xspClientDojo.js]
XSP._Validator(clientId, required, converter, validatorList, multiSep) [All]	Used internally by the `XSP.attachValidator()` function. This is a private function. [xspClientDojo.js]

Function Name	Description
XSP.DateConverter(dateFormat, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.TimeConverter(timeFormat, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP. DateTimeConverter(dateFormat, timeFormat, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.IntConverter(message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, then there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]

Table 4.3 Summary of the Private XSP Client Side JavaScript Object Functions (cont'd)

Function Name	Description
XSP.NumberConverter(dot, tho, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.RequiredValidator (message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.DateTimeRangeValidator (minTime, maxTime, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.LengthValidator(min, max, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, then there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]

Function Name	Description
XSP.NumberRangeValidator(min, max, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.RegExpValidator(expr, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.ExpressionValidator(expr, message) [All]	Instances of this object are generated by the server-side converter. You should never need to explicitly invoke this method in Client Side JavaScript yourself. If you found a use case, then there would be cross-site scripting vulnerabilities because all client-side validation should have matching server-side validation, to prevent invalid data from being accepted when the browser is compromised. This is a private function. [xspClientDojo.js]
XSP.attachEvent(clientId, targetClientId, _event, clientSideScriptName, submit, valmode, execId) [All]	Connects an `eventHandler` to a control in the client side. Therefore, when the specified event is triggered in the client side against the `targetClientId` control, a request is issued to the server-side component to trigger any corresponding server-side `eventHandler` code or simple actions. This is a private function. [xspClientDojo.js]
XSP._getEventData(targetNode, targetId, eventName) [All]	Used internally by the `XSP.attachEvent()` function and others. This is a private function. [xspClientDojo.js]

Table 4.3 Summary of the Private XSP Client Side JavaScript Object Functions (cont'd)

Function Name	Description
XSP.fireEvent(evt, clientId, targetId, clientSideScriptName, submit, valmode, execId) [All]	Used internally by the `XSP.attachEvent()` function and others. This is a private function. [xspClientDojo.js]
XSP._doFireEvent(evt, form, clientId, clientSideScriptName, submit, valmode, execId) [All]	Used internally by the `XSP.attachEvent()` function and others. This is a private function. [xspClientDojo.js]
XSP._scrollPosition() [All]	Used internally by the `XSP.scrollWindow()` function and others. This is a private function. [xspClientDojo.js]
XSP._setAllowDirtySubmit(flag) [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModifiedFlag`. This is a private function. [xspClientDojo.js]
XSP._isAllowDirtySubmit() [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModifiedFlag`. This is a private function. [xspClientDojo.js]
XSP._setDirty(flag, formId) [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModifiedFlag`. This is a private function. [xspClientDojo.js]
XSP._isDirty() [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModifiedFlag`. This is a private function. [xspClientDojo.js]
XSP._getDirtyFormId() [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModifiedFlag`. This is a private function. [xspClientDojo.js]

Function Name	Description
XSP.attachDirtyListener(clientId) [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModifiedFlag`. This is a private function. [xspClientDojo.js]
XSP.attachDirtyUnloadListener (saveMessage) [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModified-Flag` and `modifiedMessage`. This is a private function. [xspClientDojo.js]
XSP._validateDirtyForm(formId, clientId) [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModi-fiedFlag`. This is a private function. [xspClientDojo.js]
XSP._saveDirtyForm(evt, cli-entId, targetId, clientSideScript-Name, submit, valmode, execId) [All]	Used internally by the Dirty Save feature—see the `<xp:view>` properties for `enableModified-Flag`. This is a private function. [xspClientDojo.js]
XSP._doFireSaveEvent(evt, form, clientId, clientSideScript-Name, submit, valmode, execId) [All]	Used internally by the Dirty Save feature to save any data sources on the current page—see the`<xp:view>` properties for `enableModified-Flag`. This is a private function. [xspClientDojo.js]
XSP.attachPartial(clientId, targetId, execId, eventName, scriptName, valmode, refreshId, onStart, onComplete, onError) [All]	Used to connect a partial refresh–enabled `eventHandler` to a control in the client side. Therefore, when the specified event is triggered in the client side against the `targetId` control, a partial refresh request is issued to the server-side component to trigger any corresponding server-side `eventHandler` code or simple actions. This is a private function. [xspClientDojo.js]
XSP.firePartial(evt, clientId, targetId, execId, scriptName, val-mode, refreshId, onStart, onCom-plete, onError) [All]	Used internally by the `XSP.attachPartial()` /`partialRefreshGet`/`partialRefreshPost` functions. This is a private function. [xspClientDojo.js]

Table 4.3 Summary of the Private XSP Client Side JavaScript Object Functions (cont'd)

Function Name	Description
XSP._partialRefresh(method, form, refreshId, options) [All]	Used internally by the `XSP.attachPartial()` /`partialRefreshGet`/`partialRefreshPost` functions. This is a private function. [xspClientDojo.js]
XSP._replaceNode(refreshId, content) [All]	Used internally by the `XSP.firePartial()` function and others. This is a private function. [xspClientDojo.js]
XSP.processScripts(s, ex) [All]	Used internally by the XSP object function and others. This is a private function. [xspClientDojo.js]
XSP.execScripts(a) [All]	Used internally by the XSP object function and others. This is a private function. [xspClientDojo.js]
XSP.parseDojo(node) [All]	Used internally by the XSP object function and others. This is a private function. [xspClientDojo.js]
XSP.attachSimpleConfirm Submit(clientId, targetClientId, _event, message) [All]	Used to connect a confirm handler to the current page. Generated by the Confirm Simple Action. This is a private function. [xspClientDojo.js]
XSP.tagCloudSliderOnChange (sliderValue, sliderId) [All]	Used by the Tag Cloud Custom Control in the Discussion Template to control the visualization of the expanding/contracting values. This is a private function. [xspClientDojo.js]
XSP._loaded() [All]	Used internally by the XSP Object when a page loads. It is used in several page load and submission functions. This is a private function. [xspClientDojo.js]
XSP.attachViewColumn CheckboxToggler(viewId, colId) [All]	Used internally by the XSP Object to attach a column header check box toggler for the `View` control. This is a private function. [xspClientDojo.js]
XSP._toggleViewColumn CheckBoxes(viewId, colId) [All]	Used internally by the XSP Object to process column header check box toggling for the `View` control. This is a private function. [xspClientDojo.js]

Function Name	Description
XSP.isViewPanelRowSelected (viewId, ckId) [All]	Used internally by the XSP Object to identify any checked rows within a `View` control. This is a private function. [xspClientDojo.js]
XSP.initSectionScript (targetSectionId, sectionId, expand) [All]	Used internally by the XSP Object to initialize expand/collapse behavior for the `Section` control. This is a private function. [xspClientDojo.js]
XSP._moveAttr (fromNode, toNode, attrName) [All]	Used internally by the XSP Object to move an element attribute from one element to another by the `Section` control. This is a private function. [xspClientDojo.js]
XSP.serialize(o) [All]	Deprecated in favor of the `XSP.toJson()` function. It is being preserved to avoid breakage of existing scripts. [xspClientDojo.js]
XSP.logw(message) [Web/MU]	Used by the `XSP.log()` function. This is a private function. [xspClientDebug.js]
XSP._dumpObject(obj, name, indent, depth, options) [Web/MU]	Used by the `XSP.dumpObject()` function. This is a private function. [xspClientDebug.js]
XSP.publishEvent (eventName, payload, payloadType) [CA/MU]	Empty implementation in `xspClientDojo.js`. This is overridden by `xspClientCA` and `xspClientMashup.js` implementations. This function is automatically output by the XPages runtime when `ComponentPublish*Action` simple actions exist on the XPage. This is a private function. [xspClientDojo.js/xspClientCA.js/xspClientMU.js]
XSP.dispatchEvent(source, name, value, event) [All]	Automatically generated by the XPages runtime for cross-communication with the XPages View Part in the Notes client. This is a private function. [xspClientDojo.js]

Table 4.3 Summary of the Private XSP Client Side JavaScript Object Functions (cont'd)

Function Name	Description
XSP.setComponentMode(mode, params) [MU]	Empty implementation in `xspClientDojo.js`. This is overridden by the `xspClientMashup.js` implementation. This function is automatically output for Set Component Mode simple actions that exist on the XPage. This is a private function. [xspClientDojo.js -> xspClientMashup.js]
XSP.dispatchJSONEvent(source, name, value, event) [CA]	Automatically generated by the XPages runtime for cross-communication with the Composite Application container. This is a private function. [xspClientCA.js]
XSP.onComponentLoaded() [MU]	Used by the XSP Object within a mashup to initialize the widget when the page loads. This is a private function. [xspClientMashup.js]
XSP.callJavaAction(actionId, params, needReturn) [Notes]	Used internally by the XSP Object in XPages in the Notes client application. This is a private function. [xspClientRCP.js]
XSP._embedControl(id, handle) [Notes]	Used internally by the XSP Object in XPages in the Notes client application. This is a private function. [xspClientRCP.js]
XSP._resize() [Notes]	Used internally by the XSP Object in XPages in the Notes client application. This is a private function. [xspClientRCP.js]

The Public XSP Client Side JavaScript Object Functions

This section provides you with details on each of the publicly scoped XSP Client Side JavaScript object functions listed in Table 4.2. Note that it does not provide any detail on the private functions listed in Table 4.3. Notes/Domino 8.5.3 has 33 public and 58 private XSP Client Side JavaScript object functions. You can also refer to the **PCGCH04.nsf** sample application, where you will find the **publicXSPFunctions** XPage. This XPage contains working examples of the 33 public XSP Client Side JavaScript object functions. Figure 4.3 shows this XPage in a browser.

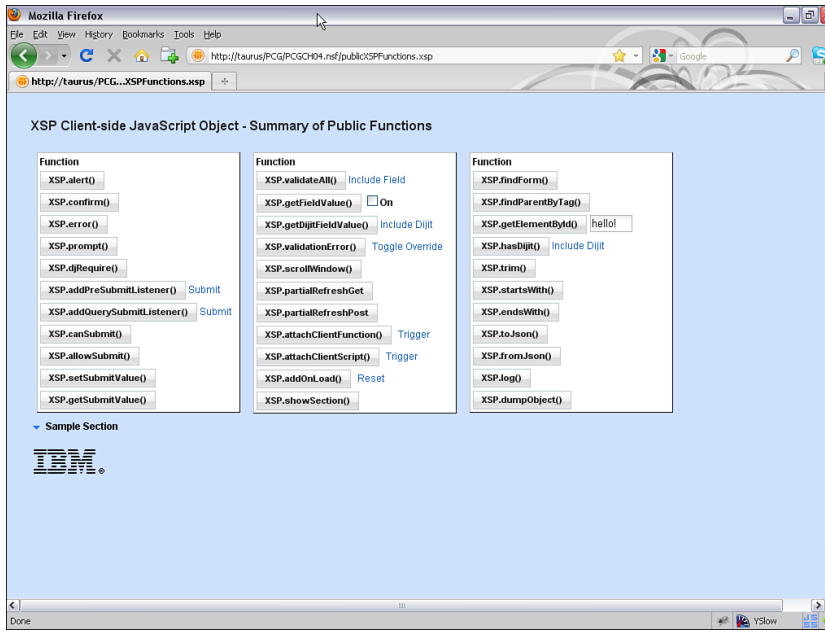


Figure 4.3 The publicXSPFunctions XPage in a browser

XSP.alert(message) : void

This function displays a generic alert dialog. It returns `void`.

A standard browser dialog is displayed in a web browser or mobile device, whereas a native Rich Client Platform (RCP) dialog is displayed in the Notes client. This behavior is handled transparently by a reimplementation of the built-in Client Side JavaScript `XSP.alert()` function within the Notes container. This allows the `XSP.alert()` function to bridge into the underlying Notes client runtime to display the native dialog.

This is an overridable Client Side JavaScript function. Everywhere that the XPages client code uses an alert dialog, this method is used to invoke the dialog. So code in your application can override the `XSP.alert()` method to do a different behavior instead of displaying a default alert dialog. You might instead make the text appear in a special error area on your page, to integrate with the UI styling of your website better than the default browser pop-up dialog. You can also implement your own application scripts to use this method so that there is consistent behavior between your own alert dialogs and the dialogs the XPages runtime provides.

Listing 4.5 details an example call of the `XSP.alert()` function within an event handler.

Listing 4.5 Example Call of the XSP.alert() Function

```
<xp:button value="XSP.alert()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[XSP.alert("Hello World!")]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.confirm(message) : boolean

This function displays a generic confirm dialog. It returns a Boolean value—`true` if the **OK** button is clicked, `false` if the **Cancel** button is clicked.

A standard browser dialog is displayed in a web browser or mobile device, whereas a native rich client (RCP) dialog is displayed in the Notes client. This behavior is handled transparently by a reimplementation of the built-in Client Side JavaScript `XSP.confirm()` function within the Notes container. This allows the `XSP.confirm()` function to bridge into the underlying Notes client runtime to display the native dialog.

This is an overridable extension point. Everywhere the XPages client code uses a confirm dialog, this method invokes the dialog. So code in your application can override the `XSP.confirm()` method to do a different behavior instead of displaying a default confirm dialog. You might instead make the text appear in a special error area on your page, to integrate with the UI styling of your website better than the default browser pop-up dialog.

Listing 4.6 details an example call of the `XSP.confirm()` function within an event handler.

Listing 4.6 Example Call of the XSP.confirm() Function

```
<xp:button value="XSP.confirm()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var result = XSP.confirm("Are you sure?");
                XSP.alert("You chose " + result);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.error(message) : void

This function displays a generic error dialog. It returns `void`.

A standard browser dialog is displayed in a web browser or mobile device, whereas a native rich client (RCP) dialog is displayed in the Notes client. This behavior is handled transparently by a reimplementation of the built-in Client Side JavaScript `XSP.error()` function within the Notes container. This allows the `XSP.error()` function to bridge into the underlying Notes client runtime to display the native dialog.

This is an overridable Client Side JavaScript function. Everywhere that the XPages client code uses an error dialog, this method invokes the standard error type dialog. Code in your own application could override the `XSP.error()` method to provide a different behavior instead of displaying a default error dialog. You might instead make the text appear in a special error area on your page, to integrate with the UI styling of your website better than the default browser pop-up dialog.

Listing 4.7 details an example call of the `XSP.error()` function within an event handler.

Listing 4.7 Example Call of the XSP.error() Function

```
<xp:button value="XSP.error()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[XSP.error("An error has occurred!")]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.prompt(message, defaultValue) : string

This function displays a generic prompt dialog with an input field. If specified, the optional `defaultValue` parameter populates the input field within the prompt dialog. This function returns the input field value from the prompt dialog when the **OK** button is clicked. It returns `null` if the **Cancel** button is clicked.

A standard browser dialog is displayed in a web browser or mobile device, whereas a native rich client (RCP) dialog is displayed in the Notes client. This behavior is handled transparently by a reimplementation of the built-in Client Side JavaScript `XSP.prompt()` function within the Notes container. This allows the `XSP.prompt()` function to bridge into the underlying Notes client runtime to display the native dialog.

This is an overridable Client Side JavaScript function. If future implementations of XPages applications require a prompt dialog, this method should be used. So code in your application can override the `XSP.prompt()` method to do a different behavior instead of displaying a default prompt dialog. You might instead make the text and edit box appear in a special area on your page, to integrate with the UI styling of your website better than the default browser pop-up dialog.

Listing 4.8 details an example call of the `XSP.prompt()` function within an event handler.

Listing 4.8 Example Call of the XSP.prompt() Function

```
<xp:button value="XSP.prompt()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var defaultAge = parseInt(Math.random()*100);
                var result = XSP.prompt(
                    "What age are you?", defaultAge
                );
                XSP.alert("You are age: " + result);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.djRequire(moduleName) : object

This function programmatically includes a Dojo module. It returns the module as an object. Use this function only if you do not want to include a particular Dojo module in the aggregated resources of an XPage (where resource aggregation has been enabled in the application properties of a version 8.5.3 or higher application). Otherwise, use the `dojo.require()` function instead. This function and `dojo.require()` perform the exact same task; the only difference is that the resource aggregator ignores any `XSP.djRequire()` calls. Both functions cache the required module on the initial call within the page to avoid repeat calls to the same module over a network. Listing 4.9 details an example call of the `XSP.djRequire()` function within an event handler.

Listing 4.9 Example Call of the XSP.djRequire() Function

```
<xp:button value="XSP.djRequire()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                XSP.djRequire("dijit.Dialog");
                myDialog = new dijit.Dialog({
                    title: "My Dialog",
                    content: "test content",
                    style: "width: 300px"
                });
                myDialog.show();
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.addPreSubmitListener(formId, listener, clientId, scriptId) : void

This function adds a custom Client Side JavaScript function, termed the listener, to a queue of zero or more other presubmit listeners. This queue of listeners gets executed just before the page is submitted to the server side. The `formId` parameter must specify the ID of the current form the listener will be triggered against on submission—this is a mandatory parameter. The `listener` parameter is a function reference to a custom Client Side JavaScript function—this is a mandatory parameter. The `clientId` parameter can specify the client-side fully namespaced ID of any single `eventHandler` or button control within the current page. This causes the presubmit listener to trigger only when the specified `clientId eventHandler` or button is invoked. This parameter is optional but should at least be specified as `null`. When set to `null`, the presubmit listener is triggered when any `eventHandler` or button tries to submit the current page. The `scriptId` parameter is also optional but declares a unique string identifier for the listener within the listener queue. If you do not specify a `scriptId`, one is automatically assigned.

The presubmit listener queue is invoked after the query submit listener queue. Also note that the custom client-side listener function does not need to return any result because this is ignored in the processing of the presubmit listener queue. Listing 4.10 details an example call of the `XSP.addPreSubmitListener()` function within an event handler.

Listing 4.10 Example Call of the XSP.addPreSubmitListener() Function

```
<xp:scriptBlock id="scriptBlock7">
    <xp:this.value>
        <![CDATA[
            function preSubmitListener(){
                XSP.alert("preSubmitListener: called presubmit!");
            }
        ]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.addPreSubmitListener()" id="button7">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var button = document.getElementById("#{id:button7}");
                var eventHandlerId = "#{id:eventHandler1}";
                var form = XSP.findForm(button);
                XSP.addPreSubmitListener(
                    form.id, preSubmitListener,
                    eventHandlerId, "preSubmitListenerId"
                );
            ]]>
        </xp:this.script>
```

```
    </xp:eventHandler>
</xp:button>
<xp:link id="link8" text="Submit">
    <xp:eventHandler id="eventHandler1" event="onclick" submit="true"
        refreshMode="complete" immediate="true">
        <xp:this.action>
            <![CDATA[#{javascript:
                print("Submission Occurred");
                context.reloadPage();
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:link>
```

XSP.addQuerySubmitListener(formId, listener, clientId, scriptId) : void

This function adds a custom Client Side JavaScript function, termed the listener, to a queue of zero or more other listeners. This queue of listeners gets executed when the page tries to submit to the server side. Based on the return results from any `querysubmit` listener functions in the queue, submission can proceed or be stopped. The `formId` parameter must specify the ID of the current form the listener will be triggered against on submission—this is a mandatory parameter. The `listener` parameter is a function reference to a custom Client Side JavaScript function—this is a mandatory parameter. The `clientId` parameter can specify the client-side fully namespaced ID of any single `eventHandler` or button control within the current page. This causes the `querysubmit` listener to trigger only when the specified `clientId eventHandler` or button is invoked. This parameter is optional but should at least be specified as `null`. When set to `null`, the `querysubmit` listener is triggered when any `eventHandler` or button tries to submit the current page. The `scriptId` parameter is also optional but declares a unique string identifier for the listener within the listener queue. If you do not specify a `scriptId`, one is automatically assigned.

The query submit listener queue is invoked before the pre-submit listener queue. Also note that the custom client-side listener function should return a Boolean result because this is used when processing the `querysubmit` listener queue to either allow or stop page submission. Listing 4.11 details an example call of the `XSP.addQuerySubmitListener()` function within an event handler.

Listing 4.11 Example Call of the XSP.addQuerySubmitListener() Function

```
<xp:scriptBlock id="scriptBlock8">
    <xp:this.value>
        <![CDATA[
            function querySubmitListener(){
                var result = XSP.confirm(
                    "querySubmitListener: Proceed to submit?"
```

```
                );
                return result;
            }
        ]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.addQuerySubmitListener()" id="button8">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var button = document.getElementById("#{id:button8}");
                var eventHandlerId = "#{id:eventHandler2}";
                var form = XSP.findForm(button);
                XSP.addQuerySubmitListener(
                    form.id, querySubmitListener,
                    eventHandlerId, "querySubmitListenerId"
                );
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link id="link9" text="Submit">
    <xp:eventHandler id="eventHandler2" event="onclick" submit="true"
        refreshMode="complete" immediate="true">
        <xp:this.action>
            <![CDATA[#{javascript:
                print("Submission Occurred");
                context.reloadPage();
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:link>
```

XSP.canSubmit() : boolean

Try to avoid programmatic page submissions in XPages using Client Side JavaScript code such as `document.forms[0].submit()` or `dojo.doc.forms[0].submit()`. This is because XPages provides a robust and well-managed event-handling and page-submission mechanism. Of course, in rare use cases, legacy Client Side JavaScript code or libraries your application uses might be performing page submission using `document.forms[0].submit()` or the like. Therefore, consider revising this code to avoid page submission, instead favoring XPages features or, alternatively, consider modifying such code to use the `XSP.canSubmit()`, `XSP.allowSubmit()`, and `XSP.validateAll()` functions to provide a more robust page submission behavior.

The `XSP.canSubmit()` function can be called just before page submission to prevent page resubmission, either accidentally when a user double-clicks on a link or if the user retries to submit the same page before a response has been received. If the page has recently been submitted, it returns `false` and the submission should be abandoned. Otherwise, it assumes that the submission will occur and updates the `XSP.lastSubmit` value with the time stamp of the last submission. If the page is not submitted after this function is called, `XSP.allowSubmit()` should be called to reenable page submission. The time allowed between submissions is configured through the `XSP.submitLatency` variable, which can also be controlled by the **xsp.properties** option `xsp.partial.update.timeout` discussed in Chapter 1. Listing 4.12 details an example call of the `XSP.canSubmit()` function within an event handler.

Listing 4.12 Example Call of the XSP.canSubmit() Function

```
<xp:button value="XSP.canSubmit()" id="button9">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="norefresh">
        <xp:this.script>
            <![CDATA[
                XSP.alert("1. lastSubmit: " + XSP.lastSubmit);
                var canSubmit = XSP.canSubmit();
                XSP.alert("2. canSubmit: " + canSubmit);
                var reset = XSP.prompt("Reset canSubmit?", "Yes");
                if(reset == "Yes"){
                    XSP.allowSubmit();
                    XSP.alert("3. lastSubmit: " + XSP.lastSubmit);
                    canSubmit = XSP.canSubmit();
                    XSP.alert("4. canSubmit: " + canSubmit);
                }
                return canSubmit;
            ]]>
        </xp:this.script>
        <xp:this.action>
            <![CDATA[#{javascript:print("Submission Occurred")}]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

XSP.allowSubmit() : void

Try to avoid programmatic page submissions in XPages using Client Side JavaScript code such as `document.forms[0].submit()` or `dojo.doc.forms[0].submit()`. This is because XPages provides a robust and well-managed event-handling and page-submission mechanism. Of course, in rare use cases, legacy Client Side JavaScript code or libraries your application uses might be performing page submission using

`document.forms[0].submit()` or the like. Therefore, consider revising this code to avoid page submission, instead favoring XPages' own features or alternatively modifying such code to use the `XSP.canSubmit()`, `XSP.allowSubmit()`, and `XSP.validateAll()` functions to provide a more robust page submission behavior.

If the page is not submitted after a call to `XSP.canSubmit()`, this function should be called to reenable the page submission mechanism. Listing 4.13 details an example call of the `XSP.allowSubmit()` function within an event handler.

Listing 4.13 Example Call of the XSP.allowSubmit() Function

```
<xp:button value="XSP.allowSubmit()" id="button1">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="norefresh">
        <xp:this.script>
            <![CDATA[
                var canSubmit = XSP.canSubmit();
                XSP.alert("1. canSubmit: " + canSubmit);
                XSP.alert("2. lastSubmit: " + XSP.lastSubmit);
                if(XSP.lastSubmit > 0){
                    XSP.alert("3. reset using allowSubmit");
                    XSP.allowSubmit();
                }
                canSubmit = XSP.canSubmit();
                XSP.alert("4. canSubmit: " + canSubmit);
                XSP.alert("5. lastSubmit: " + XSP.lastSubmit);
                return canSubmit;
            ]]>
        </xp:this.script>
        <xp:this.action>
            <![CDATA[#{javascript:print("Submission Occurred")}]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

XSP.setSubmitValue(submitValue) : void

This function passes the `submitValue` parameter value to the server side where it becomes available on the `context` object through the `context.getSubmittedValue()` method. This function should be called while processing a client-side event, just before the event is submitted to the server. The most recent call of this function takes precedence when multiple calls of this function are made before submission. Listing 4.14 details an example call of the `XSP.setSubmitValue()` function within an event handler.

Listing 4.14 Example Call of the XSP.setSubmitValue() Function

```
<xp:button value="XSP.setSubmitValue()" id="button1">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="norefresh">
        <xp:this.script>
            <![CDATA[
                XSP.setSubmitValue(navigator.userAgent)
            ]]>
        </xp:this.script>
        <xp:this.action>
            <![CDATA[#{javascript:
➥print(context.getSubmittedValue())}]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

XSP.getSubmitValue() : object

This function retrieves the submit value set using the `XSP.setSubmitValue()` function prior to event submission. After event submission occurs, this function returns `null`. Listing 4.15 details an example call of the `XSP.getSubmitValue()` function within an event handler.

Listing 4.15 Example Call of the XSP.getSubmitValue() Function

```
<xp:button value="XSP.getSubmitValue()" id="button1">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="norefresh">
        <xp:this.script>
            <![CDATA[
                var value = XSP.prompt("1 + 1 equals?", 2);
                XSP.setSubmitValue(value);
                if(XSP.getSubmitValue() != 2){
                    XSP.setSubmitValue("Incorrect");
                }
            ]]>
        </xp:this.script>
        <xp:this.action>
            <![CDATA[#{javascript:
➥print(context.getSubmittedValue())}]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

XSP.validateAll(formId, valmode, execId) : boolean

This function can be used to run client-side converters and validators before page submission to the server. The `formId` parameter must specify the form to validate. This is useful in a multiform page. The `valmode` integer parameter must specify a value of `0`, `1`, or `2`. `0` = No validation. `1` = Run converters only. `2` = Run Converters and Validators. The `execId` string parameter is optional and can be used to specify a single control within the current page to validate. If `null`, the whole page is validated. Listing 4.16 details an example call of the `XSP.validateAll()` function within an event handler.

Listing 4.16 Example Call of the XSP.validateAll() Function

```
<xp:button value="XSP.validateAll()" id="button21">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var button = document.getElementById("
➥#{id:button21}");
                var form = XSP.findForm(button);
                var valmode = 2;
                var execId = "#{id:inputText3}";
                XSP.log(
                    "form.id: " + form.id +
                    " valmode=" + valmode +
                    " execId=" + execId
                );
                var result = XSP.validateAll(form.id, valmode,
➥execId);
                XSP.log("validation passed: " + result);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link id="link6">
    <xp:this.text>
        <![CDATA[#{javascript:
            viewScope.containsKey("r") ?
                "Exclude Field" :
                "Include Field"
        }]]>
    </xp:this.text>
    <xp:eventHandler event="onclick"
        submit="true" refreshMode="complete" immediate="true">
        <xp:this.action>
            <![CDATA[#{javascript:
```

```
                viewScope.containsKey("r") ?
                viewScope.remove("r") :
                viewScope.put("r", null)
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:link>
<xp:br></xp:br>
<xp:inputText id="inputText3"
    rendered="#{javascript:viewScope.containsKey('r')}"
    required="true">
    <xp:this.converter>
        <xp:convertDateTime type="both"></xp:convertDateTime>
    </xp:this.converter>
    <xp:this.validators>
        <xp:validateDateTimeRange minimum="2011-01-01T00:00:00"
            maximum="2011-12-31T00:00:00"
➥message="Out of date range!">
        </xp:validateDateTimeRange>
        <xp:validateRequired message="Please supply a date!">
        </xp:validateRequired>
    </xp:this.validators>
    <xp:dateTimeHelper></xp:dateTimeHelper>
</xp:inputText>
```

XSP.getFieldValue(node) : string

This function returns `null` if the node parameter is `null` or no value is set within the target node. Otherwise, it returns the value as a string. This function returns a comma-separated string value for the options of a multiple type node. This function returns the checked value of a radio or check box type node. In the case of Date or Currency type fields, the return value typically is in nonconverted and nonlocalized format. Listing 4.17 details an example call of the `XSP.getFieldValue()` function within an event handler.

Listing 4.17 Example Call of the XSP.getFieldValue() Function

```
<xp:button value="XSP.getFieldValue()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var cbx = XSP.getElementById("#{id:checkBox1}");
                if(null != cbx){
                    XSP.log("field value: " + XSP.getFieldValue(cbx));
```

```
                }
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.getDijitFieldValue(dj) : object

This function returns `null` if the `dj` Dijit type parameter is `null` and is not a Dijit that supports the `getValue()` function. Otherwise, it returns the value of the `dj.getValue()` function as an object type—typically, this is a string value. Listing 4.18 details an example call of the `XSP.getDijitFieldValue()` function within an event handler.

Listing 4.18 Example Call of the XSP.getDijitFieldValue() Function

```
<xp:button value="XSP.getDijitFieldValue()" id="button15">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                if(XSP.hasDijit()){
                    var dj = dijit.byId("#{id:inputText4}");
                    if(null != dj){
                        var s = XSP.getDijitFieldValue(dj);
                        XSP.log("digit field value: " + s);
                    }
                }else{
                    XSP.alert("Please click \"Include Dijit\"");
                }
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link id="link5">
    <xp:this.text>
        <![CDATA[#{javascript:
            viewScope.containsKey("dj1") ?
                "Exclude Dijit" : "Include Dijit"
        }]]>
    </xp:this.text>
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                viewScope.containsKey("dj1") ?
```

```
                viewScope.remove("dj1") :
                viewScope.put("dj1", null)
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:link>
<xp:br></xp:br>
<xp:inputText id="inputText4"
    rendered="#{javascript:viewScope.containsKey('dj1')}">
    <xp:typeAhead mode="partial" minChars="1"
        ignoreCase="true" valueListSeparator=",">
        <xp:this.valueList>
            <![CDATA[1,2,3,4,5,6,7,8,9,10,20,30,40,50]]>
        </xp:this.valueList>
    </xp:typeAhead>
</xp:inputText>
```

XSP.validationError(clientId, message) : void

This function displays the `XSP.error()` dialog with the given `message` parameter. It then shifts focus to the control specified by the `clientId` parameter. This function returns `void`. All the XPages runtime client-side converters and validators use this method to report any problems with entered data prior to page submission. Developers who are building their own client-side validation code can use this function to report problems using the default behavior or by overriding this function.

When XPages Client Side JavaScript code reports a validation or converter error, this function invokes the `XSP.error()` dialog. Code in your application can override the `XSP.validationError()` function to perform different behavior than that of displaying the error dialog. For example, you might make the text appear beside the control that has the problem, as well as in the pop-up dialog, so that the information about the problem is still available when the user has clicked OK in the dialog. Or you may want to display a ToolTip message beside the offending input field.

Listing 4.19 details an example override and call of the `XSP.validationError()` function within an event handler.

Listing 4.19 Example Call of the XSP.validationError() Function

```
<xp:scriptBlock id="scriptBlock6" loaded="${param.v == 'y'}">
    <xp:this.value>
        <![CDATA[
            XSP.validationError = function(clientId, message){
                XSP.djRequire("dijit.Tooltip");
                new dijit.Tooltip({
                    connectId: [clientId],
```

```
                    label: message
                });
                var e = this.getElementById(clientId);
                if(e) {
                    if(e.select) e.select();
                    if(e.focus) e.focus();
                }
            }
        ]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.validationError()" id="button16">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                XSP.validationError(
                    "#{id:inputText5}",
                    "Please provide a value!"
                );
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link style="margin-left:5px" text="Toggle Override" id="link7">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var href = document.location.href;
                if(XSP.endsWith(href, "?v=y")){
                    href = href.substring(0, href.length -4);
                    document.location.href = href;
                }else{
                    document.location.href = href + "?v=y";
                }
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:link>
<xp:br></xp:br>
<xp:inputText id="inputText5" loaded="${param.v == 'y'}">
➥</xp:inputText>
```

XSP.scrollWindow(x, y) : void

This function scrolls the current window to the specified `x` and `y` pixel coordinates. Listing 4.20 details an example call of the `XSP.scrollWindow()` function within an event handler.

Listing 4.20 Example Call of the XSP.scrollWindow() Function

```
<xp:button value="XSP.scrollWindow()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var x = screen.availWidth;
                var y = screen.availHeight;
                XSP.scrollWindow(x, y);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.partialRefreshGet(refreshId, options) : void

This function programmatically executes `GET`-based partial refresh requests to the XPages server-side runtime. The `refreshId` string parameter value must specify a fully namespaced HTML DOM element ID (generated using `#{id:<controlId>}` syntax, for example). This is the target receiver of the partial refresh response content. If the target element cannot be found within the HTML DOM based on the given `refreshId`, an error dialog appears with details of the missing element when this function is triggered. The `options` parameter is optional and used to send custom parameters to the server-side as `GET` request parameters. It can also specify `onStart`, `onError`, and `onComplete` event function callbacks. These are triggered accordingly during the request lifecycle. Listing 4.21 details an example call of the `XSP.partialRefreshGet()` function within an event handler.

Listing 4.21 Example Call of the XSP.partialRefreshGet() Function

```
<xp:scriptBlock id="scriptBlock5">
    <xp:this.value>
        <![CDATA[
            function partialRefreshOnStart(){
                console.log("Partial Refresh Started");
            }
            function partialRefreshOnError(){
                console.log("Partial Refresh Error");
            }
            function partialRefreshOnComplete(){
```

```
                console.log("Partial Refresh Completed");
            }
        ]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.partialRefreshGet" id="button6">
    <xp:eventHandler id="eventHandler2" event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var secs = new Date().getUTCSeconds();
                var milliSecs = new Date().getUTCMilliseconds();

                var partialRefreshOptions = {
                    "secs" : secs,
                    "milliSecs" : milliSecs
                };

                XSP.partialRefreshGet("#{id:partialRefreshGetField}",
                    {params : partialRefreshOptions,
                     onStart : partialRefreshOnStart,
                     onError : partialRefreshOnError,
                     onComplete : partialRefreshOnComplete}
                );
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:text escape="true" id="partialRefreshGetField">
    <xp:this.value>
        <![CDATA[#{javascript:
            if(!param.isEmpty()){
                return param.secs + " : " + param.milliSecs;
            }
        }]]>
    </xp:this.value>
</xp:text>
```

XSP.partialRefreshPost(refreshId, options) : void

This function programmatically executes `POST`-based partial refresh requests to the XPages server-side runtime. The `refreshId` string parameter value must specify a fully namespaced HTML DOM element ID. This is the target receiver of the partial refresh response content. If the target element cannot be found within the HTML DOM based on the given `refreshId`, an error dialog appears with details of the missing element

when this function is triggered. The `options` parameter is optional and used to send custom parameters to the server side as `POST` request parameters. It can also specify `onStart`, `onError`, and `onComplete` event function callbacks. These are triggered accordingly during the request lifecycle. An `immediate` request parameter can also be included in the `options` content to control validation execution. Listing 4.22 details an example call of the `XSP.partialRefreshPost()` function within an event handler.

Listing 4.22 Example Call of the XSP.partialRefreshPost() Function

```
<xp:scriptBlock id="scriptBlock5">
    <xp:this.value>
        <![CDATA[
            function partialRefreshOnStart(){
                console.log("Partial Refresh Started");
            }
            function partialRefreshOnError(){
                console.log("Partial Refresh Error");
            }
            function partialRefreshOnComplete(){
                console.log("Partial Refresh Completed");
            }
        ]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.partialRefreshPost" id="button13">
    <xp:eventHandler id="eventHandler3" event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var mins = new Date().getUTCMinutes();
                var secs = new Date().getUTCSeconds();
                var milliSecs = new Date().getUTCMilliseconds();

                var partialRefreshOptions = {
                    "mins" : mins,
                    "secs" : secs,
                    "milliSecs" : milliSecs
                };

                XSP.partialRefreshPost("#{id:partialRefreshPostField}",
                    {params : partialRefreshOptions,
                     onStart : partialRefreshOnStart,
                     onError : partialRefreshOnError,
                     onComplete : partialRefreshOnComplete,
                     immediate: true}
```

```
                );
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:text escape="true" id="partialRefreshPostField">
    <xp:this.value>
        <![CDATA[#{javascript:
            if(!param.isEmpty()){
                return param.mins + " : " + param.secs +
                            " : " + param.milliSecs;
            }
        }]]>
    </xp:this.value>
</xp:text>
```

XSP.attachClientFunction(targetClientId, eventType, clientScriptName) : void

This function attaches the `clientScriptName` function reference parameter to the specified `eventType` string parameter value of the `targetClientId` string parameter value element. The `targetClientId` must be a fully namespaced client-side element ID. This function returns `void`. The `clientScriptName` parameter expects a Client Side JavaScript function reference. Therefore, only function references without parameters can be specified. Use the `XSP.attachClientScript()` if you need to specify complex function calls that include parameters. Listing 4.23 details an example call of the `XSP.attachClientFunction()` function within an event handler.

Listing 4.23 Example Call of the XSP.attachClientFunction() Function

```
<xp:scriptBlock id="scriptBlock3">
    <xp:this.value>
        <![CDATA[
            var User = function(){
            var _un = "#{javascript:@UserName()}";
            function _UN(){ XSP.alert(_un) }
                return { name : _UN }
            }
            var user = new User();
        ]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.attachClientFunction()" id="button49">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
```

```
            <![CDATA[
                XSP.attachClientFunction(
                    "#{id:link4}", "onclick", user.name
                )
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link style="margin-left:5px" text="Trigger" id="link4">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                // client script will be triggered!
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:link>
```

XSP.attachClientScript(targetClientId, eventType, clientScript) : void

This function attaches the `clientScript` string parameter to the specified `eventType` string parameter value of the `targetClientId` string parameter value element. The `targetClientId` must be a fully namespaced client-side element ID. This function returns `void`. The `clientScript` parameter is of string type. Therefore, you can construct a complex `clientScript` function call that includes parameter values within this string value if required. Listing 4.24 details an example call of the `XSP.attachClientScript()` function within an event handler.

Listing 4.24 Example Call of the XSP.attachClientScript() Function

```
<xp:scriptBlock id="scriptBlock3">
    <xp:this.value>
        <![CDATA[
            var User = function(){
            var _un = "#{javascript:@UserName()}";
            function _UN(){ XSP.alert(_un) }
                return { name : _UN }
            }
            var user = new User();
        ]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.attachClientScript()" id="button49">
    <xp:eventHandler event="onclick" submit="false">
```

```
        <xp:this.script>
            <![CDATA[
                XSP.attachClientScript(
                    "#{id:link4}", "onclick", "user.name()"
                )
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link style="margin-left:5px" text="Trigger" id="link4">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                // client script will be triggered!
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:link>
```

XSP.addOnLoad(listener) : void

This function attaches a Client Side JavaScript function to the current page's `onLoad` event. The `listener` function reference parameter must specify the Client Side JavaScript function to be triggered on page load. Listing 4.25 details an example call of the `XSP.addOnLoad()` function within an event handler.

Listing 4.25 Example Call of the XSP.addOnLoad() Function

```
<xp:scriptBlock id="scriptBlock1" loaded="${param.x == 'y'}">
    <xp:this.value>
        <![CDATA[XSP.addOnLoad(function(){alert("hello world!")})]]>
    </xp:this.value>
</xp:scriptBlock>
<xp:button value="XSP.addOnLoad()" id="button39">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var href = document.location.href;
                if(XSP.endsWith(href, "?x=y")){
                    href = href.substring(0, href.length -4);
                }
                document.location.href = href + "?x=y";
            ]]>
```

```
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link style="margin-left:5px" text="Reset" id="link1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var href = document.location.href;
                if(XSP.endsWith(href, "?x=y")){
                    href = href.substring(0, href.length -4);
                }
                document.location.href = href;
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:link>
```

XSP.showSection(sectionId, show) : void

This function toggles the expanded state of the specified section control using the client-side `sectionId` string parameter value and `show` Boolean parameter value. Listing 4.26 details an example call of the `XSP.showSection()` function within an event handler.

Listing 4.26 Example Call of the XSP.showSection() Function

```
<xp:button value="XSP.showSection()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var id = "#{id:section1}";
                var section = document.getElementById(id);
                if(null != section){
                    var closed = document.getElementById(
                        id + "_closed").value;
                    closed = (closed == "true") ? true : false;
                    XSP.showSection("#{id:section1}", closed);
                }
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:section id="section1" header="Sample Section" initClosed="true">
    <xp:image url="/xpIBMLogo.gif" id="image1"></xp:image>
</xp:section>
```

XSP.findForm(nodeOrId) : object

This function returns the parent `FORM` element from the HTML DOM of the current page relative to the specified `nodeOrId` object parameter. Listing 4.27 details an example call of the `XSP.findForm()` function within an event handler.

Listing 4.27 Example Call of the XSP.findForm() Function

```
<xp:button value="XSP.findForm()" id="button41">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var buttonId = "#{id:button41}";
                var button = XSP.getElementById(buttonId);
                var idForm = XSP.findForm(buttonId);
                var nodeForm = XSP.findForm(button);
                XSP.log("buttonId: " + buttonId);
                XSP.log("button: " + button);
                XSP.log("idForm: " + idForm);
                XSP.log("nodeForm: " + nodeForm);
                XSP.log("forms match: " + (idForm === nodeForm));
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.findParentByTag(nodeOrId, tag) : object

This function returns the nearest parent element relative to the specified `nodeOrId` target element that matches the specified `tag` string parameter value. Listing 4.28 details an example call of the `XSP.findParentByTag()` function within an event handler.

Listing 4.28 Example Call of the XSP.findParentByTag() Function

```
<xp:button value="XSP.findParentByTag()" id="button41">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var buttonId = "#{id:button41}";
                var button = XSP.getElementById(buttonId);
                var idParent = XSP.findParentByTag(buttonId, "table");
                var nodeParent = XSP.findParentByTag(button, "table");
                XSP.log("buttonId: " + buttonId);
                XSP.log("button: " + button);
                XSP.log("idParent: " + idParent);
                XSP.log("nodeParent: " + nodeParent);
```

```
                XSP.log("parents match: " + (idParent ===
➥nodeParent));
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.getElementById(elementId) : object

This function retrieves an element from the HTML Document Object Mode (DOM), based on the given `elementId` string parameter. This is similar to the common `document.getElementById()` object function call but instead ensures cross-browser operability. If the given fully namespaced client-side `elementId` is not found within the DOM, this function returns `void`. Listing 4.29 details an example call of the `XSP.getElementById()` function within an event handler.

Listing 4.29 Example Call of the XSP.getElementById() Function

```
<xp:button value="XSP.()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
            var element = XSP.getElementById("#{id:inputText1}");
            if(null != element){
                XSP.djRequire("ibm.xsp.widget.layout.xspClientDebug");
                var elementDump = XSP.dumpObject(element);
                var elementValue = element.value;
                XSP.log("Element Dump: " + elementDump);
                XSP.log("Element Value: " + elementValue);
            }
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:inputText id="inputText1"
    value="hello!" style="width:50px"></xp:inputText>
```

XSP.hasDijit() : boolean

This function determines the existence of the `dijit.byId()` function within the current HTML page. If `dijit.byId()` is present, you can then access Dojo widgets directly using this function. This means you can retrieve Dojo widget objects directly from the DOM using `dijit.byId()` for elements you understand to be Dojo widget types. This contrasts with `XSP.getElementById()`, which always returns a plain HTML DOM node. Listing 4.30 details an example call of the `XSP.hasDijit()` function within an event handler.

Listing 4.30 Example Call of the XSP.hasDijit() Function

```
<xp:button value="XSP.hasDijit()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var hd = XSP.hasDijit();
                XSP.alert("Dijit exists: " + hd);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
<xp:link id="link2">
    <xp:this.text>
        <![CDATA[#{javascript:
            viewScope.containsKey("dj") ?
                "Exclude Dijit" : "Include Dijit"
        }]]>
    </xp:this.text>
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                viewScope.containsKey("dj") ?
                viewScope.remove("dj") :
                viewScope.put("dj", null)
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:link>
<xp:br></xp:br>
<xp:inputText id="inputText2"
    rendered="#{javascript:viewScope.containsKey('dj')}">
    <xp:dateTimeHelper id="dateTimeHelper1"></xp:dateTimeHelper>
    <xp:this.converter>
        <xp:convertDateTime type="date"></xp:convertDateTime>
    </xp:this.converter>
</xp:inputText>
```

XSP.trim(s) : string

This function returns the given `s` string parameter value trimmed of leading and trailing whitespace. Listing 4.31 details an example call of the `XSP.trim()` function within an event handler.

Listing 4.31 Example Call of the XSP.trim() Function

```
<xp:button value="XSP.trim()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var s = "   hello world!   ";
                var b = s.length;
                s = XSP.trim(s);
                var a = s.length;
                XSP.alert("s length before trim: " + b);
                XSP.alert("s length after trim: " + a);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.startsWith(s, prefix) : boolean

This function determines whether a given string value contains the specified prefix string at the beginning of its value. The specified prefix can be one or more characters in length. This function returns a Boolean value. Listing 4.32 details an example call of the `XSP.startsWith()` function within an event handler.

Listing 4.32 Example Call of the XSP.startsWith() Function

```
<xp:button value="XSP.startsWith()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var str = "#{javascript:@UserName()}";
                var result = XSP.startsWith(str, "Anon");
                if(result){
                    XSP.alert(str + " starts with 'Anon'");
                }else{
                    XSP.alert(str + " does not start with 'Anon'");
                }
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.endsWith(s, suffix) : boolean

This function determines whether a given string value contains the specified suffix string at the end of its value. The specified suffix can be one or more characters in length. This

function returns a boolean value. Listing 4.33 details an example call of the `XSP.endsWith()` function within an event handler.

Listing 4.33 Example Call of the XSP.endsWith() Function

```
<xp:button value="XSP.endsWith()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var str = "#{javascript:@UserName()}";
                var result = XSP.endsWith(str, "ymous");
                if(result){
                    XSP.alert(str + " ends with 'ymous'");
                }else{
                    XSP.alert(str + " does not end with 'ymous'");
                }
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.toJson(o) : string

This function converts a JavaScript object parameter value to a String serialization of that object. It then returns the JSON serialization of that object. Listing 4.34 details an example call of the `XSP.toJson()` function within an event handler.

Listing 4.34 Example Call of the XSP.toJson() Function

```
<xp:button value="XSP.toJson()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var object = { x : "e=mc2", y : 3.14, z : false };
                var json = XSP.toJson(object);
                XSP.log("Type: " + typeof json);
                XSP.log("JSON: " + json);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.fromJson(s) : object

This function parses the specified `s` string parameter to return a JavaScript object. The `s` parameter should contain JSON syntax that will be used in the transformation. If the

given `s` string parameter contains malformed JSON syntax, this function throws an exception. The exception includes details of the syntax error. Listing 4.35 details an example call of the `XSP.fromJson()` function within an event handler.

Listing 4.35 Example Call of the XSP.fromJson() Function

```
<xp:button value="XSP.fromJson()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                var json = "{ x : 'e=mc2', y : 3.14, z : false }";
                try{
                    var object = XSP.fromJson(json);
                }catch(e){
                    XSP.alert(e);
                }
                XSP.log("Type: " + typeof object);
                var dump = XSP.dumpObject(object);
                XSP.log("Dump: " + dump);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.log(message) : void

This function is useful as a client-side debug utility for opening a separate web browser window where the given `message` string parameter values will be logged into. The Notes client does not support this function. The log window is based on the following characteristics:

```
width=300, height=600, scrollbars=yes, resizable=yes
status=no, location=no, menubar=no, toolbar=no
```

The given `message` parameter value is written into the newly created window contents in descending order. That is, the most recent invocation of `XSP.log()` is written at the top of the window, pushing previous log messages downward. The logging window remains open during a browser session, allowing multiple `XSP.log()` messages to be sequentially written into the log window. This function cannot be used in the Notes client. Listing 4.36 details an example call of the `XSP.log()` function within an event handler.

Listing 4.36 Example Call of the XSP.log() Function

```
<xp:button value="XSP.log()" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
```

```
            <![CDATA[XSP.log("Hello World!")]]></xp:this.script>
    </xp:eventHandler>
</xp:button>
```

XSP.dumpObject(object) : string

This function is useful as a client-side debug utility for retrieving the members, properties, and functions of a given DOM element. This function returns such detail in string format. It cannot be used in the Notes client. Listing 4.37 details an example call of the `XSP.dumpObject()` function within an event handler.

Listing 4.37 Example Call of the XSP.dumpObject() Function

```
<xp:button value="XSP.dumpObject()" id="btnDumpObject">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
            XSP.djRequire("ibm.xsp.widget.layout.xspClientDebug");
            var options = {name:"Dump of button object", depth:2};
            var button = document.getElementById
➥("#{id:btnDumpObject}");
            var dumpResult = XSP.dumpObject(button, options);
            XSP.log(dumpResult);
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

How XPages Uses the Dojo Framework

The Dojo Toolkit is an open-source JavaScript framework, available from dojotoolkit.org. Among its many functions, Dojo has a set of utilities for use in your scripts, such as `dojo.isIE()` to test whether the browser is Internet Explorer. Dojo includes a set of controls, such as a Slider control allowing a number to be chosen by selecting a point on a rule, and the BorderContainer control, which can split the page into multiple areas—perhaps with menus on the left, a header at the top, and the main content in the center. Dojo also has a handy mechanism for arranging your JavaScript files as modules that depend on each other. For example, you could put the main scripts for your page in a module named **app.Main**, which might depend on your **app.Styling** script and on the Dojo **dojo.fx** animation effects module. Your HTML page would need to declare only a dependency on the **app.Main** module, and all the other **.js** files would be loaded automatically. Leveraging the Dojo Toolkit means that the toolkit usually takes care of differences in the context that are available to JavaScript in different browsers. It provides a useful set of well-documented functionalities.

The XPages runtime uses the Dojo Toolkit as the underlying technology for most of its browser JavaScript functions. The XSP Client JavaScript Object is implemented using Dojo. Some of the XPages controls are implemented as controls using the Dojo widget framework. The XPages Date/Time Picker control is Dojo based, as is the Type Ahead control, which is configured through the Edit Box editor's Type Ahead tab. The XPages Rich Text control used the Dojo Editor control in versions 8.5.0 and 8.5.1. Since version 8.5.2, the Rich Text Control has been using the CKEditor from the company CKSource, with a thin Dojo wrapper used to integrate it into the XPages.

It is often advisable to use the Dojo technology directly in your applications. As discussed in the previous section, for simpler applications, you don't actually need to use Dojo: The XSP Client JavaScript Object has all the functionality you need. However, if you are doing more advanced scripting, it will probably be useful to use the Dojo APIs instead of developing your own utilities or attempting to use some other browser framework. Examples of such advanced scripting might be using JavaScript manipulation of the browser-side element tree (the Document Object Model, or DOM), programmatically applying styles, and attempting to build reusable controls from HTML with JavaScript for the user interaction behaviors. Also, although you don't really need Dojo, the set of controls it provides can give a richer experience to your users with little extra effort on your own part.

Chapter 1 discusses where to find Dojo on the Domino server file system, the different versions of Dojo that are included in different Domino server versions, and how to change the current Dojo version. See the discussion of the option **xsp.client.script.dojo.version** and the related **xsp.client.script.dojo.djConfig** option.

Before diving deeper, if you want to get more information on Dojo in general, the documentation provided on dojotoolkit.org is probably a good place to start: http://dojotoolkit.org/documentation/.

Dojo Types and Attributes

Properties on various XPage controls facilitate easy configuration of Dojo options and Dojo controls. If you have existing Dojo-based code that you want to reuse, it is mostly possible to just treat the Source tab of the XPages editor like an HTML editor and insert snippets of HTML that use the Dojo controls. However, it is usually more convenient to use the XPages Dojo integration—indeed, this may be necessary if you are using input controls that need to save field values to Domino documents. The main properties that support Dojo are the properties `dojoType` and `dojoAttributes`, available on most controls. You also need to configure the XPage root control to enable the `dojoParseOnLoad` property and to configure some Dojo Module resources. Also, if you are attempting to use a Dojo UI control instead of a utility, you must configure the themes and styling for the page.

To illustrate, we next show how to enable the Dojo Number Spinner control. This control contains an edit box with small plus and minus buttons beside it to increase or decrease the number value present in the edit box. Because control inputs a value, we

need to select a server-side XPages control that knows how to process the entered value and save it to the document field. Here we start with a simple XPage containing an Edit Box control (with tag `xp:inputText`) that has been configured to accept number values, as in Listing 4.38.

Listing 4.38 Sample Page with Edit Box Before Adding Dojo Number Spinner

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
      <xp:inputText id="inputText1" value="#{viewScope.age}">
            <xp:this.converter>
                  <xp:convertNumber type="number"></xp:convertNumber>
            </xp:this.converter>
      </xp:inputText>
</xp:view>
```

When using a Dojo control that is simply presentational, doing some layout instead of inputting a value, you might want to start with an XPages Panel control or the XPages Paragraph control.

The **dojoType** property is where you indicate which Dojo control will replace this XPage control in the browser. When you look up the Dojo documentation for the Number Spinner control, you will see that its full `dojoType` name is `dijit.form.NumberSpinner`. *Dijit* is pronounced like *digit* and is short for Dojo widget. The full names of the Dojo visual controls begin with `dijit`; the utilities begin with `dojo`. You can set that full `dojoType` name as the value of the Edit Box `dojoType` property.

The **dojoAttributes** property is used to further configure the Dojo control. In the Dojo documentation, you will find the list of Dojo properties for the Number Spinner control. For example, you might set the `smallDelta` property, indicating by how much the value in the edit box will be increased or decreased when clicking the plus and minus buttons. To set the `smallDelta` to 5, you would add a Dojo Attribute, set its name to `smallDelta`, and set its value to `5`. You can add more Dojo Attribute objects if you need to further configure the Dojo control.

At this point, we need the **dojoParseOnLoad** property of the XPage root control. If you open the XPage in the browser with only the `dojoType` and `dojoAttributes` present, the HTML page will contain those values, but the edit box will appear unchanged as a normal Edit Box control. You need to enable an option to ensure that the Number Spinner `dojoType` value in the HTML is noticed as corresponding to the Dojo Number Spinner control so that it attempts to replace the edit box with the Number Spinner control. This process, which finds all `dojoType` values and creates the corresponding Dojo control, is known as the Dojo Parser. Set the `dojoParseOnLoad` property of the XPages root control to `true` to enable the Dojo Parser.

The next step is to add the **Dojo Module resource** for the number spinner to the XPage root control. Without the Dojo Module resource, you will be see a browser JavaScript

error complaining that it doesn't understand the full name for the number spinner. The error message looks like this:

```
Could not load class 'dijit.form.NumberSpinner'
```

That problem occurs when you attempt to use a `dojoType` without declaring a dependency on that full `dojoType` name in the header of the web page. The dependency is established by adding a Dojo Module resource to the XPage root control. You need to add module resource only once for a specific dojoType; after that, you can use that Dojo control as often as you like in the XPage. In the XPages editor, select the **XPage root control**; in the **Resources** tab, choose **Add, Dojo Module**; and enter the full `dojoType` name as `dijit.form.NumberSpinner`.

Usually the last step is to set the **dojoTheme** property on the XPage root control to `true`. At this point, the Dojo control appears on the web page but may appear badly styled, with some elements of the control appearing badly shaped, in ugly colors or not correctly positioned relative to other elements. The solution here is to ensure that the CSS styles for the Dojo control are available in the page header and that the page body is using those styles. The CSS for XPages generally is configured through XPages theme files, which are XML files that reference CSS files and can define styles for different types of controls. The XPages theme for an application is configured using the **xsp.theme** option, explained in Chapter 1. Dojo has its own Dojo themes, which are named sets of CSS files providing styling for all the Dojo controls. The integration between both kinds of themes is achieved through the `dojoTheme` property on the XPages root control. That property should be set to `true` in an XPage to indicate that this page is using Dojo controls and, hence, that some Dojo theme CSS files should be included for this page. The default XPages themes, known as **webstandard** and **oneui**, honor that `dojoTheme` property by adding the CSS files for the Dojo default theme, known as **tundra**. (The tundra color scheme generally uses a lot of blue and gray colors and looks attractive.) If you are providing your own XPages runtime themes, keep in mind that you need to use one of the Dojo themes when the XPages are using Dojo controls, as some of the controls are nonfunctional without some vital CSS behavior.

The last few Dojo-related properties in XPages are the **dojoForm** property and the **Dojo Module Path** resource, both on the XPages root control. The `dojoForm` property indicates that the automatically generated HTML `form` tag that appears in every XPage should have the attribute `dojoType="dijit.form.Form"`. It is not usually necessary—see the Dojo documentation for that control for details. It is also possible to explicitly specify your own form control by setting the XPages root control `createForm` property to `false` and adding the tag `xp:form` to the Source tab of the XPages editor. The Dojo Module Path resource indicates a location from which Dojo modules, JavaScript files, may be loaded. The resource is used when providing Dojo modules within your application or somewhere on the server file system instead of just reusing the existing modules provided by the Dojo Toolkit. For an example, see the discussion of the **xsp.client.script.dojo.djConfig** option in Chapter 1.

When the page has been fully configured to use the Dojo Number Spinner, the XPage Source tab is as shown in Listing 4.39.

Listing 4.39 Sample Page Using a Dojo Number Spinner

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core" dojoParseOnLoad="true"
    dojoTheme="true">
  <xp:this.resources>
    <xp:dojoModule name="dijit.form.NumberSpinner"></xp:dojoModule>
  </xp:this.resources>
  <xp:inputText id="inputText1" value="#{viewScope.age}"
    dojoType="dijit.form.NumberSpinner">
    <xp:this.converter>
      <xp:convertNumber type="number"></xp:convertNumber>
    </xp:this.converter>
    <xp:this.dojoAttributes>
      <xp:dojoAttribute name="smallDelta" value="5"></xp:dojoAttribute>
    </xp:this.dojoAttributes>
  </xp:inputText>
</xp:view>
```

Working with Dojo Dijits

Aside from the basic instructions on using Dojo Dijit controls given in the previous section, you need to be aware of various other issues when using Dojo controls with XPages.

Common issues related to the interaction of Dojo and XPages are listed here:

- The format of IDs in the HTML source means the `#{id:` syntax is required.
- Scripts accessing Dojo controls need to use `dijit.byId`.
- Dojo controls are not available while the HTML page is loading.
- Bad AJAX requests to an XPage can cause loss of data.
- XPages input validation can interact with Dojo layout controls.
- Dojo control interaction with XPages partial update.

Those issues will be discussed in more detail in the following sections.

IDs in the HTML Source and the Requirement to Use the "#{id:" Syntax

In XPages, when you specify the ID of an Edit Box as `inputText1`, the ID that appears in the browser looks like `view:_id1:inputText1`—that is, a prefix, a colon (:), and then the ID that was present in the XPage source. The actual ID that appears in the browser is known as the `clientId` and is likely to change as you develop your application. This has an impact when you develop applications using Dojo because it is

common in Dojo to find the element corresponding to an ID to check the current value, to change the element's styling or otherwise interact with the element.

The purpose of the `clientIds` in XPages is to ensure that IDs in the HTML page will always be unique to that HTML element. For example, `clientIds` are used to ensure HTML ID uniqueness within an XPages repeat control. So for an Edit Box with ID `inputText1`, in a Repeat control named `repeat1`, the generated HTML IDs would be like so:

```
view:_id1:repeat1:0:inputText1
view:_id1:repeat1:1:inputText1
view:_id1:repeat1:2:inputText1
```

Similarly, `clientIds` ensure uniqueness when different Custom Controls are used. The XPages XML editor ensures ID uniqueness within a Custom Control, but different Custom Controls are likely to be using the same IDs for similar controls. If there are Custom Controls with IDs `custom1` and `custom2`, and each contains an Edit Box with ID `inputText1`, then in the HTML source, the Edit Box controls will have HTML IDs like this:

```
view:_id1:custom1:inputText1
view:_id1:custom2:inputText1
```

Usually, Custom Control tag instances do not have explicit IDs. In this case, control IDs are automatically assigned, as with `_id5` and `_id6` in the following example. So the previous example would more likely appear like this:

```
view:_id1:_id5:inputText1
view:_id1:_id6:inputText1
```

Those automatically generated IDs are unstable while the application is in active development. As you work on pages, adding or removing controls before the edit boxes, the `id5` number increases or decreases accordingly.

The likelihood of `clientId` values changing during application development means that the normal Dojo means of accessing elements will not consistently work in XPages. Listing 4.40 shows a number of such examples.

Listing 4.40 JavaScript Snippet Showing IDs That Do Not Work in XPages

```
// Bad scripts. These will not work
// consistently in an XPage.
var editBox;
editBox = XSP.getElementById("inputText1");
editBox = dojo.byId("inputText1");
editBox = dojo.byId("view:_id1:inputText1");
editBox = dijit.byId("inputText1");
editBox = dijit.byId("view:_id1:inputText1");
```

Instead, you should compute the IDs using a server-side computation snippet inserted into the browser JavaScript scripts. So the first function call in Listing 4.40 could be changed to either of the examples shown in Listing 4.41.

Listing 4.41 JavaScript Snippet Showing ID Formats That Work in XPages

```
editBox = XSP.getElementById(
             "#{javascript:getClientId('inputText1')}");
editBox = XSP.getElementById("#{id:inputText1}");
```

The first example there uses the normal syntax for doing a Server Side JavaScript computation inline in some browser JavaScript. It invokes the SSJS global function `getClientId(String): String`, which finds the first Edit Box control instance with ID `inputText1` in the server-side control tree, relative to the context of the `getClientId()` call (within a Custom Control versus within a whole page). The second example is the specialized `id:` script engine; because the use case to look up the `clientId` occurs so often, there is a special shortened syntax for the lookup. The second example is functionally the same as the first example; it finds the `clientId` of the closest control with the control ID `inputText1`.

A limitation states that the `id:` script engine and the Server Side JavaScript syntax will be evaluated only for client JavaScript that is inline within an XPage. They are not evaluated in Client Side JavaScript libraries (within application *.js files). If you do use the `id:` syntax within a *.js file, the Client Side JavaScript engine of the target browser simply treats it as a string literal. There are some common workarounds for that scenario. You might be able to compute the clientIds in the XPage and pass the computed IDs as parameters to the functions you're invoking in your script library. If you know outright that the control you're working with is a direct parent or first child of the DOM node you need to interact with, you can use the DOM `parentNode` reference or whatever DOM element API is particular to your situation. Finally, you can set a specific style class to act as a marker on the control and use the Dojo utilities to do a lookup by class. So for an age field in an XPage, you might set `styleClass="ageField"`, and then `dojo.query(".ageField")` will find the list of nodes with that style class. The dot in the query means it is matching a class name instead of the node name `(div, input` etc)—see the dojo.query documentation for more detail.

Scripts Accessing Dojo Controls Need to Use dijit.byId

Client JavaScript scripts used on an XPage often look up and access the HTML element corresponding to a control. The most common use case is to check the value of an input control in the browser, like this:

```
var value = XSP.getElementById("#{id:inputText}").value;
```

In this example, the Edit Box control is outputting a single HTML input tag that corresponds to an element in the browser DOM tree; the value of the element is available by referencing the `.value` property.

Dojo controls are different than plain HTML controls, in that they often consist of multiple elements in the browser tree. For example, the XPages Date Time Picker control consists of a container area, an edit box, a button to launch a calendar pop-up, and a calendar pop-up. In this case, when you attempt to find the element that corresponds to the date picker ID, the element found is the container area, which does not have a value. There is no easily predictable way to find the element within a Dojo control that contains the actual control value.

The solution is, instead of finding the element corresponding to the `clientId`, to find the Dijit object that corresponds to the `clientId`. The Dijit object is created when the page is loaded and represents the entire Dojo control. The Dojo Toolkit website has API documentation that explains all the functions and properties available on the controls provided with Dojo. The Dojo controls that are intended to behave as input controls usually have a `.value` property, used to set and retrieve the control value. The method to find the Dijit object is `dijit.byId`. Note that the `dojo.byId` is the same as `XSP.getElementById` and returns just the HTML element with the given ID—you need to ensure that, when dealing with Dojo controls, you use the `dijit.byId` method instead of accessing the element. The previous example then becomes this:

```
var value = dijit.byId("#{id:inputText}").value;
```

The last point to note about the `dijit.byId` function is that it is not always available to call on an XPage; it is available only when there is some Dojo control on the page. The main `dijit` object is usually available, though, so before calling the function, you might need to explicitly check whether the function itself is available. If you're not sure whether there is a Dojo control on the XPage, the code to look up a Dijit object looks like this:

```
var editBoxDijit = XSP.hasDijit() ? dijit.byId
➥("#{id:inputText1}"):null;
```

Dojo Controls Are Not Available While the HTML Page Is Loading

Dojo developers will be familiar with the issue surrounding Dojo control availability while the HTML page is loading, but classic XPages developers might not have encountered it. An issue arises when you write browser JavaScript code in an HTML `script` element or in an XPages Output Script control (`xp:scriptBlock`) that interacts with the HTML elements or the Dojo Dijit objects corresponding to Dojo controls. Code within an HTML `script` element is executed while the HTML page is loading in the web browser—code at the top of the page is executed before the server sends the end of the HTML page. The Dojo Parser process, which creates Dojo Dijit objects from the HTML output, executes only after the HTML page has finished loading in the browser because it needs to search the entire HTML page to process every element in the page. The Dojo controls usually edit the DOM browser tree of elements, replacing the HTML content sent by the server with a template of HTML particular to this control. For instance, the XPages Date Time Picker control replaces a single HTML input element with a container area, an input control, and a button that can launch the calendar pop-up. So because the HTML element tree changes during the Dojo Parser process, scripts

that interact with the HTML tree upon page load should usually wait until after the Dojo Parser has completed. Before the parser has completed, the Dojo Dijit objects will not be available, and the HTML elements that are present initially will be deleted and replaced by different HTML elements after the Dojo controls apply their control-specific HTML template.

Any scripts that need to run while the page is initially loading should use either the `onClientLoad` event or the `XSP.addOnLoad` method.

In version 8.5.2, the `onClientLoad` event was added to the Panel control, the XPage root control, and the Custom Control root control. The event is triggered after the Dojo parser has finished processing the HTML page. You can specify some client JavaScript for the event in the Domino Designer XPages Editor, in the **Events** view, on the **Client** tab. Listing 4.42 shows example XSP markup for the `onClientLoad` event.

Listing 4.42 XPage Snippet Accessing Dijit Objects When a Page Loads, After the Dojo Parser

```
<xp:panel>
  <xp:eventHandler event="onClientLoad" submit="false">
    <xp:this.script><![CDATA[
      var inputText1 = dijit.byId("#{id:inputText1}");
      // modify the Dojo input control here ..
    ]]></xp:this.script>
  </xp:eventHandler>
</xp:panel>
<xp:scriptBlock id="scriptBlock1">
  <xp:this.value><![CDATA[
    XSP.addOnLoad( function sb1_func(){
      var inputText1 = dijit.byId("#{id:inputText1}");
      // modify the Dojo input control here ..
    });
  ]]></xp:this.value>
</xp:scriptBlock>
```

Before version 8.5.3, `onClientLoad` was not available, so the solution was to use `XSP.addOnLoad` in the Output Script control, or in the HTML `script` element. See the previous listing for an example. For more details, see the section on the `XSP.addOnLoad` function.

Bad AJAX Requests to an XPage Can Cause Loss of Data

Certain Dojo controls are commonly used with AJAX requests. The acronym, which is short for Asynchronous JavaScript and XML, refers to the technique of using browser JavaScript in a web page to make a background request to a server for some data. In some Dojo controls, the data being displayed is separate from the UI widget displaying the data. In that case, the browser data store object is implemented using the `dojo.data`

APIs. Those data store objects commonly use AJAX requests to fetch the values to be displayed from the server. An example of a control that uses data stores in this manner is the `dojox.grid.DataGrid` control, which displays a spreadsheet-like representation of a table of values.

When making such AJAX requests in an XPages context, the requested page usually is either a request to an XPage or a request to another server via the Domino HTTP-Proxy Servlet. You can search the Domino Administrator help documents to find details of the HTTP-Proxy. When making repeated requests to an XPage, a danger of data loss arises for values entered in the current page. When a browser initially opens an XPage, a control tree instance is created server side for that XPage, and further interaction with the same page uses the same server-side control tree until you navigate to a different page. Those control trees are saved on the server in a cache, with only the 5 or 16 most recently accessed control trees preserved, depending on the cache configuration. If you configure your page to request the HTML from some XPage and you do not specify the ID of the control tree to be used, a new control tree is created on the server. If you repeatedly request new XPage control trees, the cache eventually drops the main page that the browser displays. In that case, the next time you try to interact with the main page control tree, it will not be available and all the values entered in the browser will be lost as the page displays in its initial state. You can find further discussion of the control tree cache in Chapter 1, in the section discussing the **xsp.persistence.mode** option, especially the subsection "Cache Size Limits and XPages Behavior When Limits Are Encountered."

To prevent loss of data when doing AJAX requests, including using the commonly embedded XHR APIs within all modern browsers, you need to include the ID of the control tree to be restored in your request. The simplest way is to use `XSP.partialRefreshGet` and `XSP.partialRefreshPost`, which can request only the current XPage. Those functions ensure that the request includes the control tree ID—they are discussed earlier in this chapter. The control tree ID is available in the web page in a hidden input field with the name **$$viewid**. If you need to request data from a different XPage instead of the current XPage, you should read that page's control tree ID from the first page response and use the same ID in all subsequent requests to that XPage.

Note that there are a few techniques for using an XPage to serve data to be used with the `dojo.data` API. In each case, you must ensure that the control tree ID is sent in the request. Your XPage can serve some JSON or XML content by invoking a partial update of a Computed Field control with `style="display:none"` and computed content in the JSON format. Another technique for using an XPage to serve non-HTML content was developed by Domino users on the Internet. The technique involves using the JSF **FacesContext** and **ExternalContext.getResponse()** objects—search for **XPages responseComplete** for details. Finally, the project XPages Extension Library on openNTF.org has a REST Service control that provides more intuitive ways of serving JSON data from an XPage.

XPages Input Validation Can Interact with Dojo Layout Controls

In XPages, it is possible to mark an Edit Box as required. When users attempt to submit a page where the required value is absent, they will get a pop-up dialog explaining

that the value is required, or the error message may appear in a Display Errors control. Similarly, if you have an Edit Box that is configured to accept a date value and the user enters some nondate text, an error displays explaining that the value must be a valid date. Those error messages can occur whenever an edit box has some configured validators or converter.

The input validation has implications for Dojo layout controls that do not display all their content at the same time. For example, the Dojo Tab Container control contains multiple tab areas that can contain normal controls such as Edit Boxes, but only the contents of the currently selected tab are shown at any time. The user can click on a different tab to see that tab's content area, hiding the previous tab's content area. Similarly, the Dojo Stack Container control has multiple content areas, with only one area visible at any time.

In the non-Dojo XPages Tabbed Panel control, when the HTML is output for the page, only the content of the current tab is sent to the browser. When you switch to a different tab, any validation problems in the current tab are displayed and prevent the switch to the different tab. Only when all the validation problems are resolved is it possible to switch tabs.

When using the Dojo Tab Container control, usually all the contents of all the tabs are sent to the browser; when you switch tabs, no interaction with the server takes place and validation does not occur. When you attempt to do a submit action on a page with a Tab Container control, the validation occurs for all the content in all the tabs. So you might get error messages that a required field value is absent, complaining about a field that is not visible because it is in a deselected tab's content area.

When such errors occur, usually the edit box values are not saved and the action associated with the submit event does not occur. If your submit event is not intended to act as a save action, you can avoid the validation errors by configuring the submit event. To configure the submit event to neither update the values nor display validation messages, you can use the Event Handler `immediate` property. Since version 8.5.2, it is possible to configure the submit event to both convert and update the values, but not to use the validators, so only conversion error messages can occur, not validation error messages. That is configured by using the Event Handler `disableValidators` property. If your submit event is intended to do a save action, the user should enter all the required field values before doing the save. In that case, you should make it easier for the user to understand which fields have associated error messages, by supplying custom error messages for each field. The default error message for the required validation is `Validation Error: Value is required`. You might change that to something like `The age value is required in the User Details tab`.

Dojo Control Interaction with XPages Partial Update

The XPages partial update functionality allows an area of a web page to be updated. The current browser state is sent to the server, and only the HTML output of the target area is returned and updated in the web page. From a Dojo point of view, it is necessary to

understand the low-level behavior. When the updated area is returned from the server, the old contents of the area are removed and disposed; then the new content of the area is inserted and initialized. So the Dojo controls within the target area are discarded and then re-created.

The most obvious repercussion is that the state of the Dojo control is lost. So if you have a Dojo Tab Container control where the second tab is selected and you do a partial update of an area that includes that Tab Container, the Tab Container reverts to its initial state with the first tab selected. The only state that is likely to remain after a partial update is the value of input controls. Input control values are submitted, and the server-side Edit Box control redisplays the submitted value in the response.

The other implications should apply only when you are building your own Dojo control.

When building a Dojo control, you should verify its behavior when placed inside a partial update area. People tend to implement their initialization behavior correctly because they need it to verify that the control works, but they might not pay as much attention to their destroy function. The key is to ensure that your control doesn't leave any artifacts in the page after it is destroyed. For instance, if your control creates an HTML div element under the main body tag (or anywhere outside of the area that will be deleted), it should remove that element during the destroy function, to prevent duplication of the element after the new target area content is inserted. Also, if your control programmatically creates and registers any Dijit objects in the Dijit registry besides the main Dijit object corresponding to the control ID, you must ensure that your dispose method removes the Dijit object. Otherwise, when the updated content is inserted, the Dijit registry will not allow the new copy of the Dijit object to be registered—it will fail, complaining about duplicate IDs.

Finally, when building a Dojo control, you should verify its behavior when the control is the target of a partial update—that is, when your Dojo control is being explicitly targeted, not just when your control is within an updated area. Issues might arise if your control is designed to have multiple Dijit objects. The XPages partial update code expects the Dijit object corresponding to the control ID to act as the container object and will destroy that Dijit and replace its `domNode` HTML element with the response from the server. If you have multiple Dijit objects that correspond to an XPage control, and the Dijit using the control ID is not the outermost container Dijit, the outer Dijits can remain after the main Dijit object is destroyed. This can lead to a build-up of outer elements because each newly created outer element will remain while the inner Dijit with the control ID is repeatedly replaced by both an outer and an inner Dijit. If possible, the solution should be to change your Dojo control so that the outermost Dijit object has the control ID. Otherwise, you can work around the issue by assigning the outermost Dijit object the ID with suffix `Container`. That is, when updating an area with the HTML ID `view:_id1:inputText1`, the partial update function will check for the existence of a Dijit object with the ID `view:_id1:inputText1_Container`. If found, that Dijit's `domNode` will be used as the target area.

Client-Side Debugging Techniques

Now that you have learned about the XSP Client Side JavaScript object and XPages use of the Dojo framework, no doubt you want to dive in and start writing Client Side JavaScript code. Before you do so, a quick review of the debugging options available to you is useful, in case your Client Side JavaScript code does not work according to plan every time.

When Client Side JavaScript code breaks down, it can be difficult to detect the source of the error without the aid of debugging APIs and tools. The most primitive initial step you often see used in tracking down errors is the insertion of `alert()` statements directly into the JavaScript code, as the developer desperately seeks to establish whether a particular piece of code is being called—or, if so, to show the value of critical variables at that point in the execution stack. This is a perfectly valid technique that works as well in XPages as any other browser-based development environment. Moving beyond that, some more advanced debugging features are relevant in both the XSP object and the Dojo framework, described as follows.

XSP Object Debug Functions

In the earlier section, "The Public XSP Client Side JavaScript Functions," you learned details on two useful debug utility functions, `XSP.log()` and `XSP.dumpObject()`. Both are listed with worked examples and are used throughout several of other examples, particularly the `XSP.log()` function. However, both of these functions are restricted to the web and mashup platforms. Therefore, incorporating these functions into your XPiNC application will not break the application because they simply will not do anything.

The important point to take away is that typical XPages development happens initially within the web platform before other platforms are considered. Therefore, both of these functions can be incorporated into your application and used for web debugging from the start.

One rather useful technique to consider using within your Client Side JavaScript is to use a debug flag with test blocks. The intent of this technique is to provide a single global Boolean debug variable or function acting as the debug flag. Then throughout your Client Side JavaScript code, `if` test blocks are either entered or bypassed, based on the debug flag value during execution. If entered, you can do such things as use the `XSP.log()` and `XSP.dumpObject()` functions to output vital debug and state information. Listing 4.43 details an example of the aforementioned technique.

Listing 4.43 Example of the Debug Flag and Test Block Debug Technique Using the XSP.log() and XSP.dumpObject() Functions

```
<xp:scriptBlock id="scriptBlock7">
    <xp:this.value>
        <![CDATA[
            // flag in Client Side JavaScript scriptBlock function
```

```
            // value is preprocessed using a bean in the server-side
            function isClientSideDebugMode(){
                var _DEBUG_MODE =
                    #{javascript:adminBean.isClientSideDebugMode()};
                if(null == _DEBUG_MODE){
                    _DEBUG_MODE = true;
                }
                return _DEBUG_FLAG;
            }
        ]]>
    </xp:this.value>
</xp:scriptBlock>
...
<xp:button value="Execute" id="button1">
    <xp:eventHandler event="onclick" submit="false">
        <xp:this.script>
            <![CDATA[
                // output client-side debug info for debug mode...
                if(isClientSideDebugMode()){
                    XSP.djRequire(
                        "ibm.xsp.widget.layout.xspClientDebug"
                    );
                    var options = {depth:4};
                    var iFld = document.getElementById("#{id:iFld}");
                    var dumpResult = XSP.dumpObject(iFld, options);
                    XSP.log(dumpResult);
                }

                // process client-side app workflow / logic...
                addToShoppingCart();
                presentCustomerConfirmation();
                // ...
            ]]>
        </xp:this.script>
    </xp:eventHandler>
</xp:button>
```

Client-Side Debugging with Dojo

As mentioned in the introduction to this section, the use of the `alert()` statement as a basic debugging print output utility has a similar companion in debugging tools, namely the `console.log()` function. Basically, this function accepts a string value as an input parameter and prints it to the debug console, where it can be viewed at runtime using your favourite JavaScript debugging tool. Unlike the `alert()` statement, it does not

cause a modal dialog to interrupt the execution of your code. Apart from that, however, they are typically used for the same purposes from a debugging perspective—printing variable values, showing code path hits in the execution stack, and so forth.

The Dojo framework has many more powerful debugging features that can be turned on or off via the `djConfig` object. You were already introduced to the `djConfig` object in the section "xsp.client.script.dojo.djConfig" in Chapter 1. There you learned how to use the `xsp.properties` file to assign debug parameter values to the `djConfig` option, as with `isDebug:true`. This property instructs Dojo to load its extended debugging machinery, which can then be surfaced in debugging tools such as Firebug. For example, Dojo's built-in debug console is exposed in this way, and you can see the output of any `console.log()` statements included in your JavaScript code. Some other djConfig parameter values that are useful for debugging are `debugAtAllCosts` and `debugContainerId`. The former guarantees an accurate stack trace for any `try/catch` errors that may occur in the client-side modules, while the latter can be used to associate a debug inspector window, such as the FirebugLite console window, with a particular DOM element. More information is available on these options on dojotoolkit.org—for instance, here: http://dojotoolkit.org/reference-guide/quickstart/debugging.html.

Debugging Dojo 1.6.1 or Later Versions

In an earlier tip, a technique was described for debugging versions of Dojo prior to 1.6.1. Since Dojo 1.6.1 (and any subsequent Dojo version that ships with the XPages runtime) is kitted as a plug-in, a modified technique is needed to be able to debug the JavaScript code in an intuitive way. The steps are outlined as follows:

1. Switch to the traditional Dojo location unjder the Notes/Domino data folder **domino\js\dojo-1.x.x\ibm\xsp\widget\layout**.
2. Create a new directory comprised of the Dojo version name followed by a **source** suffix—for example, **Dojo-1.6.1.source**.
3. Locate the Dojo plug-in the **OSGi** folder under the **Notes/Domino** root directory (an example follows) and unpack the JAR into the new folder: **osgi\shared\eclipse\plugins\com.ibm.xsp.dojo_8.5.3.yyyymmdd-hhmm.jar**.
4. Find the JavaScript file(s) you want to debug. Back up and remove both the **.js** and **.js.gz** versions of the file(s) and rename the **.js**.**uncompressed.js** copy as the **.js** file. For example:

   ```
   rename DateTextBox.js.uncompressed.js DateTextBox.js
   ```

5. Open the application you need to debug in Domino Designer and turn off JavaScript resource aggregation by deselecting the option **Application Properties > XPages > Use runtime optimized JavaScript and CSS resources**.
6. Open the **xsp.properties** file for this application and set the **xsp.client.script.dojo.version** property to the version of new Dojo folder, **xsp.client.script.dojo.version = 1.6.1.source**.
7. Restart the server, or simply restart the HTTP task or Notes client, and load the XPage with a Client Side JavaScript debugger enabled.

The XPages runtime will then use the **Dojo-1.6.1.source** folder as the Dojo version *for this application only*. When it is up and running, you should be able to debug the Client Side JavaScript code using the fully formatted uncompressed JavaScript code.

Other Miscellaneous Client-Side Debugging Information

This section gives options for choosing a Client Side JavaScript debugger and discuss some quirks when debugging applications using XPages in the Notes Client (XPiNC).

XPiNC Quirks

It is important to call attention to some unique aspects of client-side debugging XPiNC applications. Unlike traditional XPages web applications, XPiNC applications run in an embedded XULRunner browser component, as opposed to a full-fledged desktop browser client. This implies some restrictions from a debugging standpoint. For instance, you cannot install full browser extensions such as Firebug (which uses Firefox menus), although you can install simple browser plug-ins such as Firebug Lite. Also, you do not have an independent web browser menu with XULRunner, so performing simple debug actions such as viewing the source markup must be done in an alternative way. You learn how in this section.

To compensate for the absence of a traditional browser menu, each XPiNC application instance has its own toolbar (see Figure 4.4).

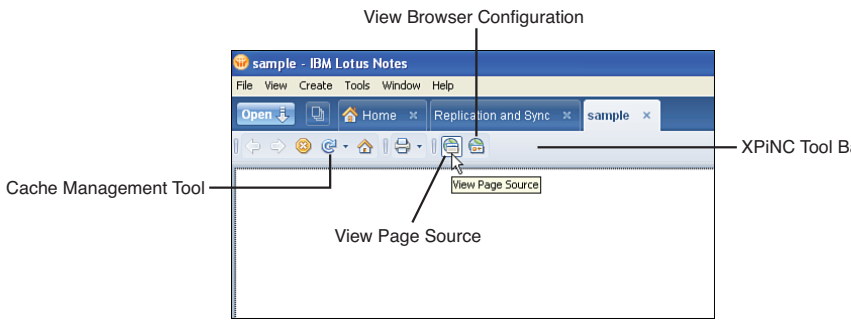


Figure 4.4 XPages Client toolbar

The toolbar has some handy utilities to aid with debugging. In particular, the **Clear Private Data** button is handy in overcoming stubbornly cached resources (such as CSS or JavaScript) that have been updated in the application design and need to be replaced in the client browser. In addition, **View Browser Configuration** might be useful to help you tweak some application settings that affect caching, character sets handling, and so forth (although it is strongly suggested that you know exactly what you're doing if you venture into this domain). You can view the HTML page source using the XPages client toolbar shown using the icon highlighted in Figure 4.4. Its content in version 8.5.3 is searchable, whereas previously it was not. Both the **View Browser Configuration** and **View Page Source** buttons are visible in the toolbar only if Domino Designer is

installed—in other words the toolbar infers from the presence of Domino Designer that the Notes client user is a developer because these options would not typically be exposed to the end user.

Any Client Side JavaScript errors that occur in your code should be reported in the Notes status bar. For example, a simple typo in an alert instruction is caught and displayed at runtime in Figure 4.5.

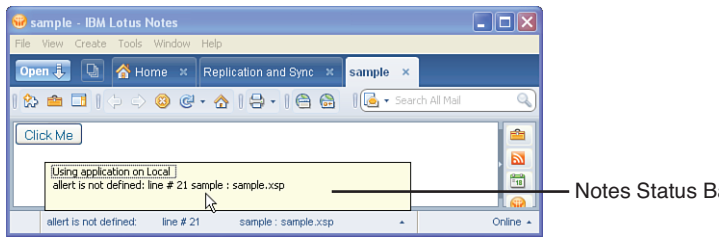


Figure 4.5 Client Side JavaScript error in the Notes status bar

This status bar shows the faulty **alert** instruction, the XPage on which it is located, and the line number in the rendered page. Note that this is the line number not in the source XPage, but in the rendered HTML page. Incidentally, the `dojo.global.onerror` function in the XSP client-side object does this for you (`xspClientRCP.js`, to be specific).

To use a client-side debugger, you need to enable Firebug Lite in your XPiNC application pages. All you need to do is include one JavaScript resource in your XPage. You can do this by entering the following tag directly into the XPages source or by adding the `src` portion of the tag as the link value for a JavaScript library resource in the Designer **Resources** property sheet. Listing 4.44 shows the required snippet of XSP markup.

Listing 4.44 Firebug Lite Tag for XPiNC Applications

```
<xp:script
            src="http://getfirebug.com/firebug-lite.js"
            clientSide="true">
</xp:script>
```

You can often reproduce client-side problems that occur in XPiNC apps by running the application on the web with the equivalent version of Firefox. Remember, XULRunner is the platform on which the Firefox browser is based, so if you can match up the XULRunner and Firefox versions, you stand a good chance of reproducing the issue in a standalone browser environment. This gives you the option to use more advanced tooling, such as Firebug instead of Firebug Lite. The astute might think it simpler to navigate to the XPiNC application within a standalone browser if the URL for the XPiNC application is known. In an ideal world, this is the case, but due to security sandboxing

of the XPiNC web container, such external requests cannot be issued against an XPiNC application.

To determine the version of XULRunner installed with your Notes client, locate the XULRunner plug-in in the Notes installation. For example, on Windows in Notes/Domino 8.5.x, you can find it under the Notes framework folder, like this:

```
framework\rcp\eclipse\plugins\com.ibm.rcp.xulrunner.runtime.win32.
➥x86_6.2.2.yyyymmdd-hhmm
```

If you then move to the `xulrunner` subfolder and execute the command `xulrunner.exe /v`, a dialog box displays the version of XULRunner. Figure 4.6 shows an example.

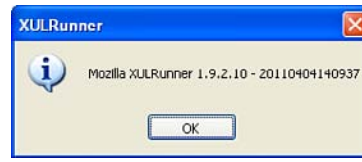


Figure 4.6 XULRunner version dialog for Notes 8.5.3

The dialog tells you that Notes 8.5.3 ships with XULRunner 1.9.2.10. A quick Internet search using those terms leads you to the **XULRunner 1.9.2 Release Notes** document on developer.mozilla.org, which states that this XULRunner 1.9.2 matches Firefox 3.6.23—bingo!

Picking a Client Side JavaScript Debugger

At this point, it is clear that different debugging tools are available for each web platform. Although Firebug Lite should run in any browser, extended tools are also available for the various browser clients available today. We do not recommend one particular tool over another; if you need to debug XPages applications on a particular browser and are not sure where to begin, Table 4.4 can help you get started.

Table 4.4 Client Side Debugging Tools

Browser	Debugger
XULRunner—XPiNC	Firebug Lite (runs on any browser)
Firefox	Firebug (available as a browser add-on)
Google Chrome	Browser has built-in Developer Tools
Safari	Browser has built-in Web Inspector tooling
Internet Explorer	IE Developer Toolbar (available as a browser add-on)

Finally, other client-side tools are available to help debug and troubleshoot other issues, such as client-side application performance. Tools such as Yahoo! YSlow and Google

Page Speed are certainly worth evaluating from that perspective. For advanced Dojo testing, you can also leverage the Dojo Objective Harness unit testing framework, commonly referred to as DOH. This provides a way for you to unit-test Dojo objects in isolation of any particular browser. DOH is documented at the following URL: http://dojotoolkit.org/reference-guide/util/doh.html.

Conclusion

In this chapter, you learned about the XSP Client Side JavaScript object. You learned how this object is divided into a hierarchy based on the running platform—web, Notes client, composite application, Mashup Center, and so forth. An extensive examination of the public and private functions available on the XSP Client Side JavaScript object taught you about the purpose of each function. You also have a useful resource at hand in the **publicXSPFunctions** XPage within this chapter's supporting application. This XPage gives you a worked example of all 33 public XSP Client Side JavaScript functions as of Notes/Domino 8.5.3. You then learned about the ways XPages uses and extends the Dojo framework. Finally, you saw various aspects of client-side debugging and trouble-shooting techniques. Hopefully, the sum of these parts provides you with both the big picture and the small details of working with XPages applications at the front end.

In the next chapter, you learn about the other side of the coin, with a detailed examination of server-side scripting. This complements the details described in this chapter, to give you an overall understanding of JavaScript development in XPages.

This page intentionally left blank

CHAPTER 5
Server-Side Scripting

If you inspected the design of any given Notes/Domino application, you could well find quite a sprinkling of different programming languages woven through the various design elements. The platform itself is more than 20 years old, so it should not be too surprising to find that some cool and modern XPages application were originally written in the early 1990s and contain copious amounts of LotusScript, Notes Formula Language, XML, JavaScript, and Java code. This is both good and perhaps not so good. On one hand, it is impressive that a slick Notes/Domino application could embrace the Web 2.0 model without going through an extensive rip-and-replace procedure, and instead seamlessly evolve as new technologies emerged over time. On the other hand, you might worry that you need to understand a hodge-podge of programming languages to build a cool Notes/Domino application. For the latter, worry not!

Despite the multitude of possible coding languages in Notes/Domino, the XPages programming model is simple. JavaScript is the default programming language for the XPages runtime, on both the front-end and back-end sides of the client/server equation. Thus, if you are creating a new XPages application from scratch, JavaScript is *the* language you need in almost all coding situations on both the client side and the server side. The same holds true if you are extending an existing Notes/Domino application to use XPages—with the added benefit that you can also leverage preexisting development assets such as LotusScript libraries and agents by calling them from Server Side Javascript (SSJS) code. If you are familiar with the LotusScript back-end classes, you can use a parallel set of Java classes for the same purposes via SSJS. If you have spent years learning the Notes Formula Language, you'll appreciate that the `@Functions` you know and love are already provided for you as built-in SSJS XPages functions. If you already know the JavaScript language from client-side programming in the web browser, you do not need to learn a different language to do server-side XPages development.

In short, JavaScript is the core XPages programming language for the application developer, and Server Side JavaScript is a cornerstone of the overall XPages programming model. Much of the core feature set of XPages SSJS is already described in various publications, not the least of which is the Domino Designer help documentation itself. This chapter explores many of the lesser-known but valuable server-side Java Script nuggets.

As usual, you should download the NSF application containing the examples used in this chapter—namely PCGCH05.NSF. Open it in Domino Designer and, ideally, have it available as you read along. From Domino Designer, you can preview the sample XPages in Notes and/or on a web browser (although the **Design > Preview in Web Browser > Default Browser** option appears to be problematical, the other preview options all work well). All the sample NSFs are downloadable from this website: www.ibmpressbooks.com/title/0132943050

What Can I Do with Server Side JavaScript?

In the XPages programming model, Server Side JavaScript is a many-splendored thing. That is, it has many different bits and pieces that combine to make it a powerful tool. A good place to start is to look at two key SSJS elements: its object model and its collection of scripting objects and system libraries.

XPages Object Model

XPages provides its own object model for Server Side JavaScript. This object model is a combination of the JavaServer Faces object model, the Domino object model, and some new objects XPages provides to make the application developer's life easier. Using Server Side JavaScript, you can:

- Manipulate the elements of the XPage—that is, programmatically modify the structure, properties, and behavior of your application on the server side
- Read information about the current request, such as parameters, current user, and user's locale
- Interact with the runtime state—for example, determine whether the response has been rendered
- Get information about the current application state
- Use the Domino back-end classes to access the application data, typically via Domino documents and views
- Access any arbitrary Java library that happens to be made available on the client or server (as already discussed in the Chapter 2 section, "Enabling Extended Java Code with the java.policy File")

When you create an XPage and add controls, you are actually defining a hierarchical component tree. Each tag in an XPage corresponds to one or more components. You can access these components programmatically and manipulate them using Server Side JavaScript.

Server-Side Scripting Objects and System Libraries

XPages provides a rich set of objects and libraries to support Server Side JavaScript coding. The reference tab in the JavaScript editor shown in Figure 5.1 provides access to the list of available global objects and methods plus the system libraries. By default, most classes and methods are displayed, but you can select the **Show advanced JavaScript** option to display the complete list. You can also double-click on any entry to add that element to your script editor.

Table 5.1 summarizes the objects listed under the Global Objects drop down item featured in Figure 5.1.

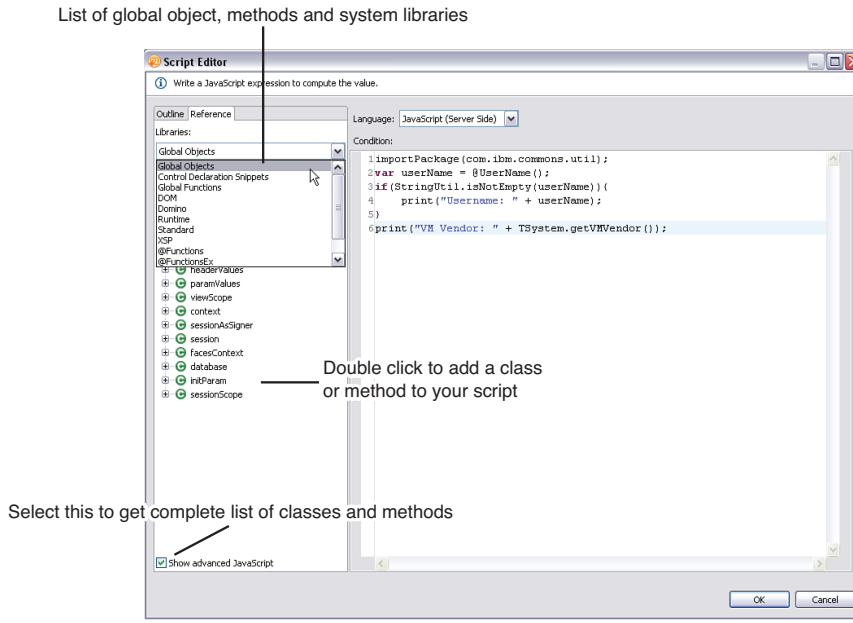


Figure 5.1 JavaScript editor reference tab

Table 5.1 Server Side JavaScript Global Objects

Object	Description
applicationScope	Map containing the application scope variables.
context	Server Side JavaScript object representing the XSPContext (or XPages runtime). The `context` object provides access to the XPages `XSPContext` Java object and provides XPages-specific contextual information about the current request—for example, access to the associated user, time zone, locale, and so forth. It also provides a number of utility methods that can be used within your application logic, such as page navigation and HTML filtering.
cookie	Map containing the cookies for the current request.
currentDocument	NotesXspDocument object representing the current in-scope document data source.
database	Server side JavaScript object representing the current Notes/Domino database. The `database` server side JavaScript object provides access to the `Database` Java object that is an instance of the `lotus.domino.Database` class. The `database` object provides access to the current Domino database and supports a wide range of database-centric operations.

Object	Description
facesContext	Server Side JavaScript object containing state information relating to the current request.
header	Map containing the HTTP header values for the current request.
headerValues	Map consisting of String arrays containing all the header values for the HTTP headers in the current request.
initParams	Map containing the initialization parameters for the current application.
param	Map containing the request parameters for the current request.
paramValues	Map consisting of String arrays containing all the parameter values for the parameters in the current request.
requestScope	Map that lasts for the duration of a request and contains the request attributes for the current request.
session	Server Side JavaScript object representing the current user's Notes/Domino session. The `session` Server Side JavaScript object provides access to the `Session` Java object, which is an instance of the `lotus.domino.Session` class. The `session` is assigned credentials based on those of the current user. The `session` is restricted by the application's ACL and the security tab of the server's Domino Directory entry.
sessionAsSigner	Server Side JavaScript object representing a Notes/Domino session based on the signer's ID. The `sessionAsSigner` server side JavaScript object provides access to a `Session` Java object, which is an instance of `lotus.domino.Session`. The session is assigned credentials based on those of the signer of the XPages design element. The session is restricted by the application's ACL and the security tab of the server's Domino Directory entry.
sessionAsSigner-WithFullAccess	Server Side JavaScript object representing a Notes/Domino session, based on the signer's ID with full access privileges. The `sessionAsSignerWithFullAccess` object provides access to a `Session` Java object, which is an instance of `lotus.domino.Session`. The session is assigned credentials based on those of the signer of the XPages design element and also allows full administrative access to the application's data. The signer must have permission for full administrative access, or this session is not created and will not be available.
sessionScope	Map that lasts for the duration of a user session within an application and contains any session variable values.
view	UIViewRoot object of the current component tree (page).
viewScope	Object that lasts for the duration of a view (XPage instance) and stores variable values.

All the objects listed in Table 5.1 are implicit global variables. This means that these objects are already there—you do not need to create them because the XPages runtime has already done so. Some of these come directly from the underlying JSF layer; others, such as `currentDocument`, have been added as XPages extensions. Most of the objects are self-explanatory and require little or no further expansion. However, the four scope objects merit some special attention.

Scope Objects

The `requestScope`, `viewScope`, `sessionScope`, and `applicationScope` global objects represent a set of objects that are essentially Maps—that is, Java collection objects that store data items that can be looked up using a key. You can use these to buffer variables within a defined lifecycle for each object.

Three of the four scope objects come directly from the underlying JavaServer Faces implementation, on which XPages is built. The fourth, the `viewScope` object, is provided specifically by XPages. All four are named and referenced in Server Side JavaScript code exactly as listed in Table 5.1: `requestScope`, `viewScope`, `sessionScope`, and `applicationScope`. Each object has a well-defined lifetime.

requestScope

The `requestScope` object lasts for the duration of a single request. This object is a Map, so you can add your own variables keyed by name. This Map is not empty by default, so you need to ensure that you are not overwriting some of the request-specific variables. A quick check using the Server Side JavaScript `_dump(requestScope)` reveals the contents for you. The variables you add do not need to be serializable. A serializable object is one whose in-memory state can be persisted to a file, typically so that it can be re-created at a later time by reversing the process (deserialization). The serialization process is sometimes also referred to as marshalling or deflating.

viewScope

The `viewScope` object enables you to scope your own variables to the lifetime of the associated view (XPage). As mentioned earlier, the state of a view can be cached between requests so that multiple requests act on the same state of the XPage. The view is restored at the beginning and saved at the end of each request, and any view scope variables are saved and restored as part of this process. The `viewScope` object is a Map, so you can add your own variables keyed by name. This Map is empty by default, so you can select whatever names you want without concern for name clashes. The variables you add must be serializable for their state to be saved. In Java, a serializable object must implement the `java.io.Serializable` interface. In general, you do not have to concern yourself with these details, but note that certain native Domino objects, such as `Date` and `DateRange`, are not serializable. If you are working with these objects, you must come up with your own scheme for writing and restoring object state.

sessionScope

The `sessionScope` object lasts for the duration of the user's session—that is, until the user session timeout expires or the user logs out. This object is a Map, so you can add your own variables keyed by name. This Map is empty by default, so you can select whatever names you want without concern for name clashes. The variables you add do not need to be serializable.

applicationScope

The `applicationScope` object lasts for the duration of the application being loaded in XPages runtime memory. This object is a Map, so you can add your own variables keyed by name. This Map is empty by default, so you can select whatever names you want without concern for name clashes. The variables you add do not need to be serializable. Developers should be vigilant about the number and size of the objects stored in `applicationScope`, because these objects will be stored even after the user logs out.

Listing 5.1 shows XSP markup from the **scopedObjects** XPage in the supporting **PCGCH05.nsf** application. You can open this in Domino Designer to read and preview the code. This XPage populates a variable in each of the scoped objects, based on the current system time. It also includes computed fields to display each of the variable values. This demonstrates the lifespan of each of the scoped objects.

Listing 5.1 Example XPage Demonstrating Lifespan of the Scoped Objects

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
    <xp:this.afterPageLoad>
        <![CDATA[#{javascript:
            var now = new Date();
            if(!requestScope.containsKey("requestVar")){
                requestScope.put("requestVar",
                    "Request scope variable added: " + now
                );
            }
            if(!viewScope.containsKey("viewVar")){
                viewScope.put("viewVar",
                    "View scope variable added: " + now
                );
            }
            if(!sessionScope.containsKey("sessionVar")){
                sessionScope.put("sessionVar",
                    "Session scope variable added: " + now
                );
            }
            if(!applicationScope.containsKey("applicationVar")){
                applicationScope.put("applicationVar",
```

```
                    "Application scope variable added: " + now
                );
            }
        }]]>
    </xp:this.afterPageLoad>
    <xp:table>
        <xp:tr>
            <xp:td>Request scope variable:</xp:td>
            <xp:td>
                <xp:text escape="true" id="computedField1"
                    value="#{requestScope.requestVar}">
                </xp:text>
            </xp:td>
        </xp:tr>
        <xp:tr>
            <xp:td>View scope variable:</xp:td>
            <xp:td>
                <xp:text escape="true" id="computedField2"
                    value="#{viewScope.viewVar}">
                </xp:text>
            </xp:td>
        </xp:tr>
        <xp:tr>
            <xp:td>Session scope variable:</xp:td>
            <xp:td>
                <xp:text escape="true" id="computedField3"
                    value="#{sessionScope.sessionVar}">
                </xp:text>
            </xp:td>
        </xp:tr>
        <xp:tr>
            <xp:td>Application scope variable:</xp:td>
            <xp:td>
                <xp:text escape="true" id="computedField4"
                    value="#{applicationScope.applicationVar}">
                </xp:text>
            </xp:td>
        </xp:tr>
    </xp:table>
    <xp:button value="Refresh" id="button1">
        <xp:eventHandler event="onclick" submit="true"
            refreshMode="complete" immediate="false" save="true">
        </xp:eventHandler>
    </xp:button>
```

```
    <xp:button value="Reload" id="button2">
        <xp:eventHandler event="onclick" submit="true"
            refreshMode="complete">
            <xp:this.action>
                <![CDATA[#{javascript:context.reloadPage()}]]>
            </xp:this.action>
        </xp:eventHandler>
    </xp:button>
</xp:view>
```

The `requestScope` variable becomes unavailable when you click the refresh button; the most typical use of request scope variables is to pass parameters from one page to another. The `viewScope` variable becomes unavailable when you click the reload button or reload the web page in your browser. These variables are useful when you want to compute a value once and then make it available to use in multiple places within a page. The `sessionScope` variable becomes unavailable when the user session expires—for example, if you are previewing, you can restart the browser. The typical use case is to store some information about a user that will be needed for the entire user session; for example, a user name. The `applicationScope` will still be there and unmodified; to dispose of it, you need to restart Domino Designer when in preview mode (for a deployed application, you need to restart the `http` task on the Domino server or restart the Notes client, for an XPiNC application). Application-scope variables are used when you need to compute something once and share it across the entire application so that all users will see the value. Be careful about the security and multithreading implications of using these variables.

Other Global Objects

The standard Help documentation available on the various aspects of SSJS has increased voluminously since the initial release in version 8.5. Figure 5.2 shows the help pages available in Domino Designer 8.5.3 for all the SSJS elements displayed in Figure 5.1.

In Domino Designer, you can access these pages via the **Help > Help Contents > Lotus Domino Designer XPages Reference**. This documentation is also available online at this website: www.tinyurl.com/XPagesSSJS

The reference information provided in the Help documentation is comprehensive, so instead of just repeating it all here, this section focuses on areas where you can add real value. The prime candidates for deeper exploration are the server-side global functions.

Summary of Server-Side Global Functions

The Server Side JavaScript global functions are some common utilities for interacting with the server-side controls. No new global functions have been added since the initial release of XPages, version 8.5. New global functions might conflict with application

developers' functions in Server Side JavaScript libraries. Instead, new utility functions have been added to the `context` global object, such as the `context.bundle` methods added in version 8.5.2.

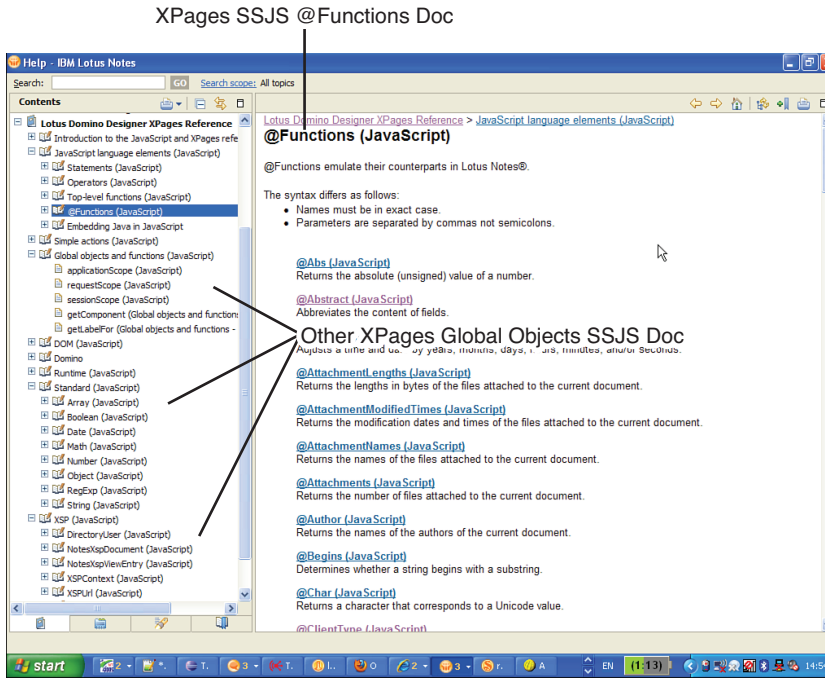


Figure 5.2 Server Side JavaScript documentation in Domino Designer help pages

All these functions are evaluated relative to the current XPage control. The current XPage control is determined by where in the XPage this snippet of JavaScript computation is defined. JavaScript computations are generally present as computed values of a control property or as computed values on other objects, such as a Data Source, which are themselves defined on a control. To be relative to the current XPage control, the global functions need to start with the current control so that the function can evaluate. For example, the `getView` function starts at the current XPage control and searches its ancestors to find the XPage root control, known as the *view*. The other functions do different kinds of searches, starting from the current XPage control.

Server Side JavaScript library functions that are invoked from a property computation in an XPage are considered relative to the control that invokes the function. Consider the script library in Listing 5.2.

Listing 5.2 Script Library Using Global Functions While the Library Loads

```
function testGetComponent(){
       return getComponent("inputText1");
}
//loadTimeFoundComponent = testGetComponent();
```

That script library loads without error. You can invoke the `testGetComponent()` function from an XPage, and it will evaluate correctly, relative to the control invoking the function.

However, there is a limitation. Server Side JavaScript script library code should not attempt to use the Global Functions while the script library is loading. While the library is loading, there is no current XPage control, and the global functions will not be available.

If you uncomment the last line of that listing, the value `loadTimeFoundComponent` will be evaluated while the script library is initially loading. At that time, the `getComponent` method will not be available and an error will occur with the message `'getComponent' not found`.

The global functions can be used in a script library (just not while it loads), which is why they are available in the Global Functions area of the Server Side JavaScript Script Library editor in Domino Designer.

The current control is available in an XPage through the variable name `this`. So the following Computed Field value evaluates to the Computed Field control instance:

```
<xp:text escape="true" id="computedField1" value="#{javascript:this}">
➥</xp:text>
```

The web page output for that control looks like this:

```
com.ibm.xsp.component.xp.XspOutputText@2ce22ce2
```

However, in a script library, the value `this` does not evaluate to the current control; it evaluates instead to the JavaScript context, either the global JavaScript context or the current JavaScript object. The global functions, such as `getComponent`, still work when the `this` reference is not a control because the functions don't rely on the `this` variable name to find the current control. To use the current control in a script library function, the `this` value should be evaluated in the XPage and passed as a parameter to the function.

When using the `this` variable name in the Domino Designer Events view, the JavaScript script is saved to an Event Handler control. So, for instance, in a button **onclick** event, `this` resolves to an event handler control and `this.getParent()` resolves to the button, the event handler's parent.

The current control is assigned to each JavaScript computation object while the XPage control tree is loaded. Advanced users who are programmatically creating controls might

need to assign the current control programmatically by calling `ValueBindingEx.setComponent` or `MethodBindingEx.setComponent` on the computation object.

getComponent(id:String): UIComponent

The `getComponent` global function is used to find a server-side control object that corresponds to a control ID. The ID used is the control ID present in the XPage Source tab, not the `clientId` that is output in the HTML source. The ID of an Edit Box looks like `inputText1` instead of `view:_id1:inputText1`. The return type full classname is `javax.faces.component.UIComponent`, which is the base type of all server-side control objects. If no control is found with that ID, the function returns `null` instead of reporting an error. If multiple controls in the server-side control tree have the same ID, the search finds the matching control that is closest to the control where the `getComponent` function is invoked. The next few paragraphs discuss the method of searching that determines which control is considered closest.

The usual reason to search for control values is to find the current value of that control. It can also be useful to find other properties of the control object—whether it is disabled or enabled, the current style, and so on. Since version 8.5.2, the Domino Designer Server Side JavaScript editor contains an option in the **Reference Libraries** drop-down called **Control Declaration Snippets**. When you select the **Control Declaration Snippets**, you see the ID of each control in the XPage. When you double-click such an ID, some JavaScript code is generated that uses the `getComponent` method to find the control with that ID and declares a variable corresponding to the ID. The variable is defined with the fully qualified type corresponding to the server-side control object's Java class. The generated code for an Edit Box with ID `inputText1` looks like this:

```
var inputText1:com.ibm.xsp.component.xp.XspInputText =
➥getComponent("inputText1");
```

Because the variable has the fully qualified type, you can type in `inputText1` and press Ctrl+space in the Server Side JavaScript editor to see the full list of methods available for that class. The **Edit Box** has a method named `getValue`, as do the other controls used to input a value from the browser (**List Box**, **Combo Box,** and so on). Before version 8.5.2, the type was not automatically available, so it prompted only with the methods from the base `UIComponent` class; the method `getValue` was not in the list. To find the control class in version 8.5.1 and earlier, you print it to the server console using: `print(inputText1.getClass().getName())`, and then update your code to provide the type inline as in the generated code so that you can see the list of methods available. Those control classes are documented as part of the XPages Extensibility APIs.

If you are writing Java code and you need to find a control by searching for the control ID, a utility method is documented as part of the XPages Extensibility APIs. The method is:

```
com.ibm.xsp.util.FacesUtil.getComponentFor(UIComponent, String):
➥UIComponent
```

Do not use the `UIComponent.findComponent` method: It searches only the current XPage, Custom Control, or Repeat control (or other control implementing the `NamingContainer` interface). It does not search the entire control tree when passed a normal control ID. The JSF API documentation describes that method more fully.

Before getting into the details of the search algorithm, you should be aware of certain details of the `getComponent` search. First, if there is no control with the given ID in the control tree, the search will take a relatively large amount of time. Larger control trees take proportionally more time, so for performance reasons, it's best to search for only controls that will be present. For example, if you've computed the `loaded` property of some control so that it sometimes does not load and you then want to check whether it has loaded in some script, it is better not to search for that specific control ID. Instead, you might wrap a container `div` tag around that control and then search for the container and check whether its child is present, like so:

```
div.getChildCount() > 0
```

Second, in versions 8.5.1 and earlier, the `getComponent` method does not find pass-through HTML tags. Pass-through or markup tags are HTML tags typed into the XPages Source tab in the Domino Designer editor, as opposed to XPages controls that are created using the Controls palette. Since version 8.5.2, the IDs of pass-through tags are used as control IDs, so the `getComponent` method finds the control objects that correspond to the pass-through tags.

The last issue to be aware of is that prior to version 8.5.3 `getComponent` does not work with runtime facet areas. Generally, when a control is placed in another control in an XPage, the contained control is present at runtime in the `children` list. However if the container control declares a named facet area, Domino Designer allows contained controls in a `this.facets` tag in the XPage. Each contained control then has an attribute `xp:key="facetName"` that corresponds to a facet area name. So facets are just contained controls that are not part of the `children` list.

At runtime, when the server-side control tree is loaded, the facet area may then be stored in a `facets` Map instead of in the normal `children` list. In the earlier releases, when searching for a control with a given ID, the `getComponent` method does not search within the runtime `facets` Map or in any content within the facet areas. The most common occurrence of facets in an XPage is when using a Custom Control that contains an Editable Area control. However, the Editable Area facets are changed at load time from being facets to being present in the children list, so the `getComponent` function does work with Editable Areas. The other facets in the XPages runtime controls are the `header` and `footer` facet areas on the Repeat, View Panel, and Data Table controls. Those are runtime facets, so the `getComponent` function does not find controls within those areas. This issue is also likely to be encountered when using third-party controls that use facets. For example, the XPages Extension Library project on OpenNTF.org contains some controls that use facets heavily. That project provides the utility method to use instead of the `getComponent` method:

```
ExtLibUtil.getComponentFor(UIComponent, String).
```

Here we discuss how the `getComponent` method searches through the server-side control tree to find the closest control with the given ID. Multiple controls might have the same ID because IDs are enforced to be unique only in the Source tab of a single XPage or Custom Control file. The runtime control tree, on the other hand, contains controls from all Custom Controls that are referenced by the XPage, so the controls from different files with the same ID are all present in the control tree. It is useful to understand the search algorithm when an application is finding the wrong control for a given ID. An inefficient search for a control also can show up as a performance problem, causing an application to run more slowly. Knowing the search algorithm can help you choose a better control as a starting point for the search, yielding faster response times. (The XPages Toolbox project on openNTF.org can help with such performance problems, as is discussed in Chapter 6, "Server-Side Debugging Techniques.")

The XPages `getComponent` method actually uses two kinds of search algorithms. First, it searches using a technique particular to the XPages runtime. If that does not find a match, it tries a broader search technique particular to JSF (JavaServer Faces, the server-side framework XPages uses). To illustrate, Listing 5.3 gives a representation of a server-side control tree, using three Custom Controls, three Edit Boxes, and a Button. The XPage and Custom Controls are provided in the sample application for this chapter. The sample has `styleClass` properties that we use to refer to the controls, to avoid confusion because we're searching for an ID. Assume that the `getComponent` evaluation is starting from the `xp:eventHandler` control under the Button and is searching for a control with ID `inputText1`. `xp:eventHandler` is shaded in gray in Listing 5.3.

Listing 5.3 Server-Side Control Tree Representation

```
xp:view styleClass="rootControl"
  xp:div styleClass="XPageContainerDiv"
   xc:firstCustom styleClass="firstCustomControl"
     xp:div styleClass="firstCustomDiv"
     xc:secondCustom styleClass="secondCustomControl"
       xp:div styleClass="secondCustomOuterContainer"
         xp:div styleClass="secondCustomHeaderContainer"
           xp:inputText styleClass="firstEditBox"
         xp:repeat styleClass="repeatContainer"
           xp:div styleClass="secondCustomMainContainer"
             xp:div styleClass="secondCustomInnerContainer"
               xp:button styleClass="mainButton"
                 xp:eventHandler
               xp:div styleClass="secondCustomDiv"
                 xc:thirdCustom styleClass="thirdCustomControl"
                   xp:div styleClass="thirdCustomContainer"
                     xp:inputText styleClass="secondEditBox"
```

```
    xp:div styleClass="firstCustomDiv2"
  xp:div styleClass="XPageDiv2"
    xp:inputText styleClass="thirdEditBox"
```

The initial search using the XPages-specific technique starts at the current control (the Event Handler). It then selects each ancestor control and searches through all its children, excluding Custom Control children. It stops when it reaches a container Custom Control or the XPage root control. The search compares each control's ID to the searched-for ID, except that it does not recheck the previously searched children. So for the example in Listing 5.3, the sequence of controls whose ID is checked involves first the Event Handler, then the button, then the `secondCustomInnerContainer` and its children (excluding the button), then the `secondCustomDiv`, and then the `third CustomControl` but not its contents (because Custom Control contents are excluded). Next, the `secondCustomMainContainer` is checked, followed by the `repeat Container`, the `secondCustomOuterContainer`, `secondCustomHeader Container`, and `firstEditBox`. Then the `secondCustomControl` is checked. The search does not move further up the control tree because the XPages-specific search searches only the current Custom Control or XPage (unlike the JSF search, which is relative to `NamingContainer` controls).

If the XPages-specific technique does not find a control, the JSF technique is used and searches the entire control tree. The JSF technique depends on which controls implement the `NamingContainer` interface. The controls included in the XPages runtime that implement `NamingContainer` are the Custom Control tag; the Repeat, View Panel, Data Table, XPage root control; and the Form control. In the JSF technique, instead of searching up through each ancestor, the search starts at the nearest `NamingContainer` ancestor and searches down from there, before moving up to the next `Naming Container` ancestor and repeating until it searches from the XPages root control. Again, areas of the control tree that have already been searched are not rechecked. In this case, starting at the Event Handler control, the search goes up through the ancestors to find a `NamingContainer` control. In this example, the Repeat control is found. It searches `repeatContainer`, `secondCustomMainContainer`, `secondCustom InnerContainer`, `mainButton`, `eventHandler`, `secondCustomDiv`, `third CustomControl`, `thirdCustomContainer`, and `secondEditBox`. Note that it does descend into Custom Controls used in the current page. After the `secondEditBox`, if it still hasn't found the control with the desired ID, it finds the Repeat control's next `NamingContainer` ancestor, `secondCustomControl`, and searches through that, so checking the `secondCustom` Control, `secondCustomOuterContainer`, `secondCustomHeaderContainer`, and `firstEditBox`. It skips `repeatContainer` because it was already searched. The next `NamingContainer` ancestor is searched, so it searches `firstCustomControl` and `firstCustomDiv`, skips `secondCustomControl`, and searches `secondCustomDiv`. Then the next `NamingContainer` is the XPage root control is searched; it searches its contents, `XPageContainerDiv`, `XPageDiv2`, and `thirdEditBox`. At this stage, all controls have been checked, so if the control has not been found, the `getComponent` method returns `null`.

getClientId(id:String): String

The `getClientId` global function converts from the control ID, as it would appear in the XPage Source tab in the Domino Designer editor, to the HTML client ID that is present in the HTML source of the page in a web browser. So it would convert the ID of an edit box from `inputText1` to `view:_id1:inputText1` or to `view:_id1:_id5:repeat1:5:inputText1` or to whatever the HTML value should be, based on where the edit box is found in the server-side control tree.

Chapter 4, "Working with the XSP Client Side JavaScript Object," discusses why client IDs need to be different from control IDs in the section "Working with Dojo Dijits." It also discusses the `#{id:inputText1}` syntax, which can be used to compute the client ID in a shorter syntax. The function finds the control that corresponds to the given control ID using the technique described for the `getComponent` global function. The `UIComponent.getClientId(facesContext:FacesContext)` method is then used to retrieve the client ID. So if your code has already used `getComponent` to find the control object, you can invoke the control `getClientId(FacesContext)` method directly.

The `getClientId` function has a few nonintuitive behaviors. First is the interaction with the Repeat control. As previously discussed, the Repeat control alters the client ID of its children depending on which row of the repeat data is currently being processed. So the same Edit Box control has many different client IDs, like this:

```
view:_id1:repeat1:0:inputText1
view:_id1:repeat1:1:inputText1
view:_id1:repeat1:2:inputText1
```

This means that scripts that request the ID of an Edit Box in a Repeat will get different values, depending on where the script using the global function is itself situated. If the script is being executed from the first row, it will evaluate to the client ID containing `:0:`. If the global function is being executed in the third row, it will evaluate to the client ID containing `:2:` However, the really odd part is that if the global function is being evaluated in a script that's outside the Repeat control, the result of the expression will be like `view:_id1:repeat1:inputText1`. That is, from outside the Repeat control, the Repeat contents appear to have client IDs that do not contain any repeat row number. That is incorrect—the HTML IDs in the web page all contain repeat row numbers for the different Edit Box instances. The client ID from outside the Repeat control will not resolve to any element in the web browser DOM tree. Basically, you shouldn't invoke the `getClientId` method for elements in a Repeat control in code that's executing outside the Repeat control. Any time you need to perform some action on the content of a Repeat control, the simplest solution is to put the action button into the Repeat control so that it can act on the Edit Box in the same row of the Repeat. The other option is to use View Panel controls in Domino Designer: You can enable a check box to appear in each row. Then in server-side actions, you can use `UIViewPanel.getSelectedIds():String[]` to get the document IDs of all the rows that were selected in the browser.

Another issue relating to `getClientId` is that it shouldn't be used at page load time—while the server-side control tree is being constructed. Scripts that are executed during page load include the `beforePageLoad` and `afterPageLoad` events on the XPage and Custom Control root controls, and property computations that use the Script Editor dialog's **Compute on Page Load** option instead of the default **Compute Dynamically**. Also for this discussion of `getClientId`, the `afterRestoreView` event on the XPage root control exhibits similar behavior. A few problems arise with using `getClientId` during page load.

First, and similar to the issue with the Repeat control, the client IDs computed at that point will be inaccurate and will not correspond to controls in the HTML page output. That will lead to errors if you are attempting to programmatically partial update the control or otherwise use the clientID.

Second, the invocation of `getClientId` on controls causes the automatically assigned control IDs to be assigned earlier than usual. So a control that would normally have been assigned a generated ID of `_id26` might have an ID of `_id2` instead. This can occasionally cause problems at runtime as the automatically generated ID is different than those which may be expected either by scripts or test automation.

Finally, and most seriously, the control hierarchy at page load time might not reflect the control hierarchy at runtime. This occurs when controls load their content initially and then move them to a different place in the server-side control tree during the page load phase. Examples include the View Panel control, the Editable Area control, and the Repeat with `repeatContents="true"`. There are others as well. When this problem occurs, not only is the client ID incorrect at page load time, but it is also incorrect while the HTML is being generated during the initial page display (because the client ID value is cached). The client ID reverts to the correct ID in subsequent page redisplays. This leads to hard-to-diagnose problems, in which the page displays and works initially but breaks when you click some button or when you experience a partial update in an area of the page. This issue generally occurs during the performance-enhancing phase of application development; developers follow the guidelines which encourage reducing computations by ensuring that values that can be computed dynamically are changed to compute only at page load time, to prevent wasteful recomputation. In summary, the computation of client IDs should not be done at page load time and must be computed dynamically.

getLabelFor(component:UIComponent):UIComponent

This global function finds an XPages Label control whose target `for` property points to the given control. Usually, a label control is paired with an Edit Box or some other input control, as the `for` target. When a `for` target is provided, the Label control outputs an HTML `label` element; otherwise, it just outputs as text. The HTML label element is an accessibility-friendly way of providing text for the purpose of a control where it is used in an XPage. Screen reader software for visually impaired people reads out the label associated with an input control when the input control gains focus. This `getLabelFor` function can be used in error-handling or validation-handling code, to retrieve the label control and label text corresponding to some edit box. It is useful when attempting to

reuse validation messages such as **The field {0} is required**, where the edit box label might be dynamically inserted at the **{0}** location instead of translating multiple validation messages for each edit box.

Note that the search for the label control starts from the control passed as an argument, using a search technique similar to the JSF search technique explained in the previous section on the `getComponent` function. Because the search is relative to the Edit Box control, the result does not depend on the location of this script in the XPage.

An alternate form of this function is used like `getLabelFor("inputText1")`. In this case, the argument is the String control ID of the Edit Box or the label's target control. The ID is resolved to a control using the `getComponent` search, and the `getLabelFor` function searches for a label control targeting the resolved control.

getView(): UIViewRoot

The `getView` method is usually the same as the `view` global object. It finds the XPage root control at the top of the current XPage's server-side control tree. It is resolved starting at the current control and iterates up through the ancestors using the `UIComponent.getParent()` method. This means that it does not resolve during load-time computed values; such values are computed after a control is created but before it is added to the server-side control tree, so no ancestors can be found. It is otherwise available during the page load phase, though—it is resolved in the `beforePageLoad` and `afterPageLoad` events. Accessing the view through the global object is slightly faster because it avoids searching through the ancestor controls.

The XPage root object is useful to access the various values it holds. For example, the current XPage name is available through `view.getPageName()`. The result is similar to `/page1.xsp`, with a preceding `/` and `.xsp` at the end. It also maintains the ID of the server-side control tree instance, available through `view.getViewId()`. For more, see the XPages Extensibility API JavaDocs on the `UIViewRootEx` control.

getForm(): UIForm

This global function searches through the ancestors of the current control until it finds a control based on the class `javax.faces.component.UIForm`. Usually, each XPage has one form control that the XPage root control generates automatically. To disable the automatic form, you can set `createForm="false"` on the XPage root control and use the `xp:form` tag to configure a form in the XPage Source tab of the Domino Designer editor. The main reason to find the form control is to get the form's client ID for use in browser scripts, like so: `getForm().getClientId(facesContext)`. The form is involved in the page submission to the server from the browser, although there are some issues to be aware of before you attempt to programmatically trigger form submission. The "Working with Dojo Dijits" section of Chapter 4 discusses some issues to ensure that the server-side control tree instance is correctly restored in response to requests. You might also want to use the `XSP.validateAll` methods and related `XSP.canSubmit` to ensure that your form submission behaves well with regard to XPages form validation. Chapter 4 also discusses those methods.

save():void

The `save` global function saves all data sources in the server-side control tree. The implementation resolves the root control as described for the `getView` function and uses the `UIViewRootEx.save()` method to save the data sources. It behaves the same as the **Save** simple action, available in the Events view in Domino Designer. The save behavior starts at the XPage root control and searches through the entire server-side control tree, saving each data source it encounters. If you know for certain that your page is editing only a single Domino Document data source, you can avoid searching the control tree by using the **Save Document** simple action instead. The **Save Document** simple action saves one specific named data source instead of doing a general save of all data sources. For noneditable data sources, such as the **Domino View** data source, this `save` function has no effect.

Working with Java Made Simpler

As you now know, Server Side JavaScript can readily access and manipulate Java objects. With the release of Domino Designer 8.5.3, access to Java artifacts from within the Domino Designer client has become more straightforward with the introduction of the Java design element. Knowing how to access the resulting Java code from within your Server Side JavaScript code will add greatly to your skill set.

Importing Java Packages into Server Side JavaScript

As a means of making it easier to reference Java classes, you can use the `importPackage()` global function in SSJS code. This eliminates the need to fully qualify Java classes with their respective package name. This is particularly useful when writing script code that references several Java classes multiple times.

Listing 5.4 details Server Side JavaScript code using the `importPackage()` global function. This example is taken from the **importingJava** XPage in the `PCGCH05.nsf` supporting application. It demonstrates how Java utility classes in this package, such as `StringUtil` and `TSystem`, can be used without having to use the fully qualified class names in each case. Note how the SSJS snippet also includes an `@Function` call (`@UserName()`). Remember that, in XPages, `@Functions` are implemented in JavaScript and can be called in SSJS code just like any other SSJS function.

Listing 5.4 Server Side JavaScript Code Using the importPackage() Global Function

```
<xp:button value="Import Package" id="button2">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                importPackage(com.ibm.commons.util);
                var userName = @UserName();
                if(StringUtil.isNotEmpty(userName)){
```

```
                print("Username: " + userName);
            }
            print("VM Vendor: " + TSystem.getVMVendor());
        }]]>
    </xp:this.action>
</xp:eventHandler>
</xp:button>
```

Creating Custom Java Classes

Before version 8.5.3, it was possible to integrate Java code into your XPages application, but it took a little more effort. Developers typically created their Java class within their application simply by switching to the Java perspective and creating a new Java class file from there. One of the major pitfalls of this technique was that a consistent location for the Java code was never defined. As a result, developers were placing their Java code in arbitrary folder locations within the database. In most cases, this worked okay, but in some cases it caused maintenance issues because it was not obvious to developers where the Java code resided for any given application. The Java design element was introduced to Domino, Notes, and Domino Designer as of release 8.5.3.

Another, more significant issue was also quickly realized. In many cases, developers were placing custom Java code within the **Local** folder within their database. The **Local** folder is a reserved location in the application structure in Domino Designer. The contents of the **Local** folder are saved directly to the file system instead of being saved into the NSF application. Thus, Java files saved to the **Local** folder were never saved to the database, meaning that if the developer had the misfortune to delete the local copy of the database, or performed a clean build on the database, the custom Java code was lost. Version 8.5.3 introduced the Java design element mainly to provide a consistent method of integrating Java code into XPages applications. The Java design element serves as a fully fledged design element that allows XPage developers to write Java code that can be integrated into their XPage applications. The Java code within the Java design element can be called directly from within your Server Side JavaScript code. As described in Chapter 6, it is possible to debug the Java code directly from within Domino Designer.

Creating Managed Beans

In Notes/Domino 8.5.2, XPages introduced support for Managed Beans, which are just plain old JavaBeans whose creation and lifespan are “managed” by the underlying runtime framework. (They come to XPages from JSF.) It should be pointed out though, that XPages makes it easy to develop an application using Managed Beans. Indeed, the Notes/Domino 8.5.2 Discussion Template actually includes a Managed Bean to make the **allDocumentsView** Custom Control much more interactive and efficient.

As the name implies, some degree of automated management is involved. A Managed Bean has both an execution lifecycle and a scope under which it lives. The managed part is related to the management of that lifecycle and scope. This makes it pretty easy to develop Managed Beans: All the infrastructural code is already in place within the JSF

layer. Where XPages lends a further helping hand is in its support for Managed Beans. This support provides a registration mechanism through the `faces-config.xml` file, but it also enables you to work directly with Managed Beans in your Server Side Java Script code. You also do not have to worry about initializing or constructing any Managed Bean instances; the XPages runtime takes care of this the first time you call a method on a Managed Bean in Server Side JavaScript code.

Reviewing a Working Example

The PCGCH05.nsf application was actually created from the Notes/Domino 8.5.3 Discussion Template. This means that it contains the same Managed Bean code provided by the Discussion Template. The first place to look in Domino Designer is the WebContent/WEB-INF/faces-config.xml file using the **Package Explorer** view. When you have found it, simply double-click on it to open it in Domino Designer. Listing 5.5 shows the content of this file.

Listing 5.5 The faces-config.xml from the PCGCH05.nsf Application

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config>
    <managed-bean>
        <managed-bean-name>previewBean</managed-bean-name>
        <managed-bean-class>
            com.ibm.xpages.beans.PreviewBean
        </managed-bean-class>
        <managed-bean-scope>view</managed-bean-scope>
    </managed-bean>
    <!--AUTOGEN-START-BUILDER: Automatically generated by
        IBM Lotus Domino Designer. Do not modify.-->
    <!--AUTOGEN-END-BUILDER: End of automatically generated section-->
</faces-config>
```

This is an XML-based file used to declare JSF-related artifacts for an application. In this case, one Managed Bean is being declared:

1. The `<managed-bean-name>` element declares the name that will be used to reference the Managed Bean in Server Side JavaScript or EL Language code. Names should be unique within each application.
2. The `<managed-bean-class>` element declares the implementation Java class.
3. The `<managed-bean-scope>` element declares the scope under which the Managed Bean will live. Valid scopes are `application`, `session`, `request`, and `view`. These scopes are comparable to the Server Side JavaScript-scoped objects detailed in the "Summary of the Scoped Objects" section earlier in this chapter.

You can declare as many Managed Beans as you need within each of the scopes using the `faces-config.xml` file. For instance, you might require several Managed Beans in your application that do different things within the view scope, and perhaps another couple that work with the session scope.

The next part to study is the implementation Java class for this Managed Bean. As shown in Listing 5.5, the `<managed-bean-class>` element declares `com.ibm.xpages.beans.PreviewBean` to be the implementation Java class. In Domino Designer, you should use **Package Explorer** view to examine the **Build Path** for the **PCGCH05.nsf** application. This shows you that a directory named **source** has been configured to be included in the compilation build path for the application. This means that any ***.java** source files within that directory are automatically compiled. The compiled ***.class** files are then part of the executable application. Figure 5.3 shows the **Java Build Path** editor for the **PCGCH05.nsf** application.

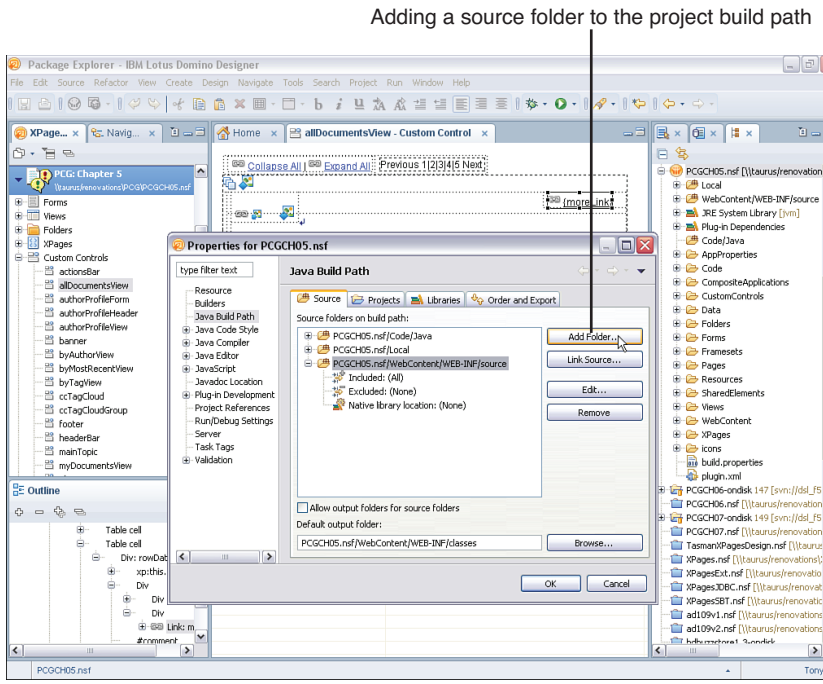


Figure 5.3 The Java Build Path editor

Using the Java Build Path editor, you can see that the **WebContent/WEB-INF/source** directory is on the build path. You are free to create directories under the **WebContent** folder as required—in this example, the **source** directory resides under the **WebContent/WEB-INF** folder so that its content is not accessible using a web URL. Any content under the **WebContent/WEB-INF** folder is protected from web URL access.

Close the **Java Build Path** editor and return to the **Package Explorer** view, where you should fully expand the **WebContent/WEB-INF/source** Java folder. Inside you will find the declared Managed Bean implementation Java package and class file, as shown in Figure 5.4.

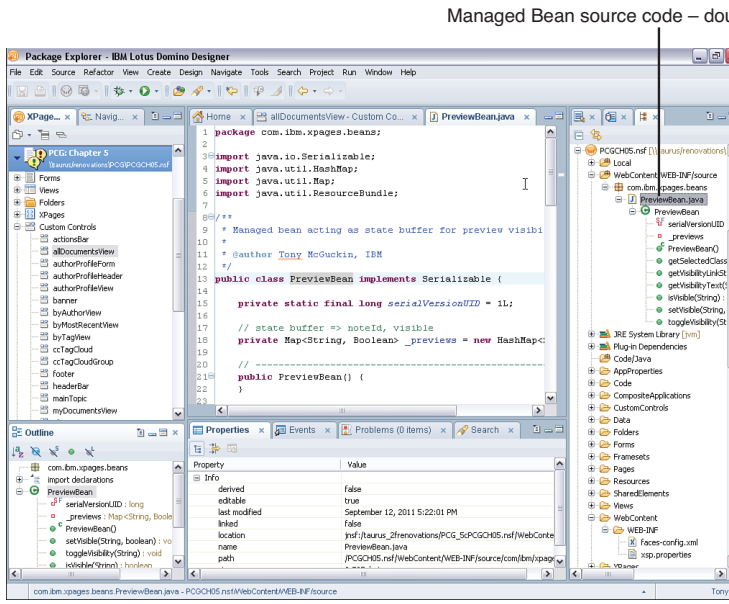


Figure 5.4 The declared Managed Bean implementation Java package and class

If you double-click the **PreviewBean.java** file, it opens in a Java editor within Domino Designer. Listing 5.6 also details the main parts of the code within this class file.

Listing 5.6 Source Code for the com.ibm.xpages.PreviewBean Class

```
package com.ibm.xpages.beans;
...
public class PreviewBean implements Serializable {
    ...
    private Map<String,Boolean> _previews=new
HashMap<String,Boolean>();

    public PreviewBean(){}

    public void setVisible(final String noteId, final boolean visible)
{
        if(_previews.containsKey(noteId)) {
            if (false == visible) {
```

```
                _previews.remove(noteId);
                return;
            }
        }
        _previews.put(noteId, true);
    }

    public void toggleVisibility(final String noteId) {
        if(_previews.containsKey(noteId)) {
            _previews.remove(noteId);
        }else{
            _previews.put(noteId, true);
        }
    }

    public boolean isVisible(final String noteId) {
        if(_previews.containsKey(noteId)) {
            return (_previews.get(noteId).booleanValue());
        }
        return (false);
    }

    public String getVisibilityText(
        final String noteId, final ResourceBundle resourceBundle) {
        String moreLinkText = "More";
        String hideLinkText = "Hide";

        if(null != resourceBundle){
            moreLinkText = resourceBundle.getString(
                "alldocuments.more.link"
            );
            hideLinkText = resourceBundle.getString(
                "alldocuments.hide.link"
            );
        }
        if(_previews.containsKey(noteId)) {
            return (hideLinkText);
        }
        return (moreLinkText);
    }

    public String getSelectedClassName(final String noteId) {
        if(_previews.containsKey(noteId)) {
            return ("xspHtmlTrViewSelected");
```

```
        }
        return ("xspHtmlTrView");
    }

    public String getVisibilityLinkStyle(final String noteId) {
        if(_previews.containsKey(noteId)) {
            return ("visibility:visible");
        }
        return ("visibility:hidden");
    }
}
```

The implementation class for this Managed Bean is not complex. It simply declares a number of public methods that are used by Server Side JavaScript code in the **allDocumentsView** Custom Control, as you will see shortly. The main point to remember here is that a Managed Bean should declare a public no-parameter constructor and should also implement the `java.io.Serializable` interface. This enables the Managed Bean to be serialized and deserialized between requests to an XPage that uses the Managed Bean. This supports the scope mechanism; without it, the Managed Bean would not persist between requests, therefore invalidating the notion of any declared scope.

The final item to examine here is the **allDocumentsView** Custom Control, to see how Server Side JavaScript code leverages this Managed Bean. Open this Custom Control in Domino Designer and, within the **Design** editor, click on the link with the ID **moreLink**. Then switch over to the **Source** editor, where you see the full range of Server Side JavaScript calls being used by this link control against the Managed Bean. Listing 5.7 shows the key lines of code in the XSP markup for the **moreLink** control.

Listing 5.7 XSP Markup for the moreLink Link in the allDocumentsView Custom Control

```
<xp:link id="moreLink"
text=
"#{javascript:previewBean.getVisibilityText(rowData.getNoteID(), res)}"
style=
"#{javascript:previewBean.getVisibilityLinkStyle(rowData.getNoteID())}">
    <xp:eventHandler event="onclick" submit="true" ...>
        <xp:this.action>
        <![CDATA[
        #{javascript:previewBean.toggleVisibility(rowData.getNoteID())}
        ]]>
        </xp:this.action>
    ...
    </xp:eventHandler>
</xp:link>
```

As you can see in Listing 5.7 (and also within the **allDocumentsView** Custom Control, if you have Domino Designer opened), the **moreLink** is making extensive use of the Managed Bean. The interesting aspect of this is the direct reference to the Managed Bean name, `previewBean`, within the Server Side JavaScript.

Now take the opportunity to preview the **allDocuments** XPage. With this new knowledge about how the **allDocumentView** Custom Control is working, you should toggle the **moreLink** on different rows of the view and also page back and forth through the view. Take note of how the `previewBean` is maintaining the state of expanded and collapsed rows for the **allDocumentsView** Custom Control, changing the style of the rows, and also changing the text of the **moreLink** for each row (see Figure 5.5).

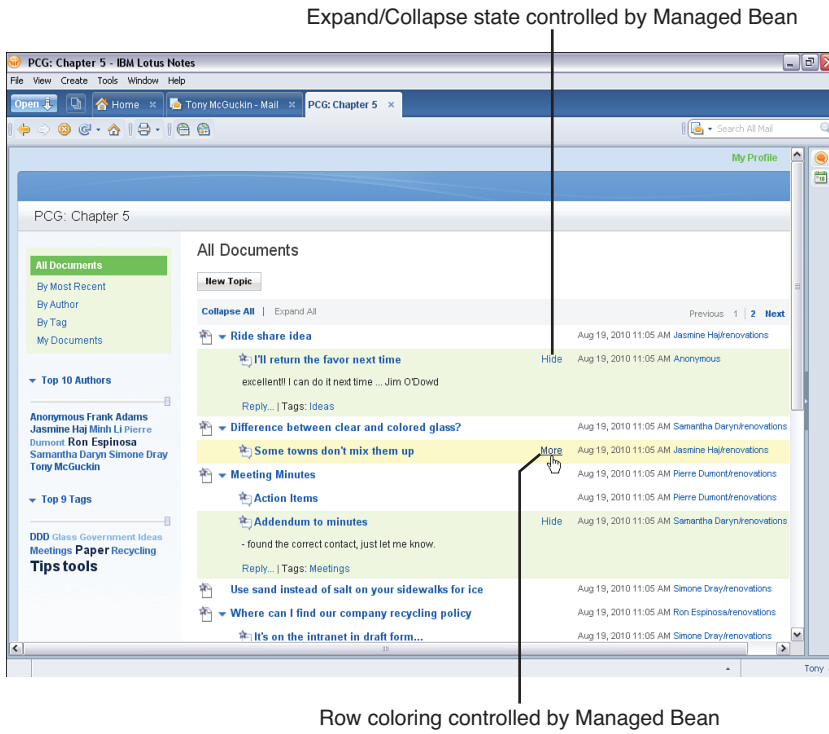


Figure 5.5 The allDocumentView Custom Control and previewBean in action

Managed Bean Properties and Server Side JavaScript

We have already described the powerful functionality that Managed Beans provide. Another feature of Managed Beans that merits mention is Managed Bean Properties. Similar to Managed Beans themselves, Managed Bean Properties enable the developer to associate properties with the bean that can be resolved by the XPages runtime. These

managed properties may also be directly referenced from within your Server Side Java Script code. This functionality provides a huge amount of flexibility to the application developer because it enables applications—or, more specifically, Managed Beans—to be configured with default properties on an application-by-application basis. Managed Bean Properties allow applications to be highly configurable. The Managed Bean Property enables the application developer to create properties that vary from installation to installation and can be configured by a system administrator upon installation.

As is often the case, an example best illustrates the power of Managed Bean Properties.

Listing 5.8 shows the markup necessary to define basic Managed Bean Properties via `faces-config.xml`.

Listing 5.8 The faces-config.xml Definition of Managed Bean Properties

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config>
        <managed-bean>
          <managed-bean-name>connections</managed-bean-name>
          <managed-bean-class>com.ibm.ser.FileService
➥</managed-bean-class>
          <managed-bean-scope>session</managed-bean-scope>
          <managed-property>
            <property-name>proxyEnabled</property-name>
            <value>true</value>
          </managed-property>
          <managed-property>
            <property-name>authentication</property-name>
            <value>basic</value>
          </managed-property>
          <managed-property>
            <property-name>authenticationPage</property-name>
            <value>_BasicLogin</value>
          </managed-property>
        </managed-bean>
</faces-config>
```

In Listing 5.8, the developer has created a Managed Bean within an application and associated a number of properties with it. Note that the syntax for declaring the Managed Bean is identical to that described earlier. The following three steps describe the extra XML markup necessary to define a Managed Bean Property.

1. The `<managed-property>` tag denotes the beginning of a managed property definition.
2. The `<property-name>` markup denotes the name of the Managed Bean Property to be set.
3. The `<value>` tag sets a default value for the Managed Bean Property.

It is worth noting at this point that the Managed Bean Properties are actual properties that can be set or gotten from the Managed Bean class. To avoid runtime errors, the Managed Bean class *must* have `getter` and `setter` methods for each Managed Bean Property associated with the Managed Bean. For instance, in Listing 5.8, the `File Service` class *must* have a `getter` and `setter` method for `proxyEnabled (getProxyEnabled(), setProxyEnabled(String))`, `authentication (getAuthentication(), setAuthentication(String))`, `authentication-Page (getAuthenticationPage()`, and `setAuthenticationPage(String))`. Such `getter` and `setter` methods should be compliant with the standard Java Bean specification. This also means that Boolean types should have an `isXYZ` type `getter`. If the `getters` and `setters` are not provided for any `faces-config.xml` declared Managed Bean property, a runtime error occurs.

> **TIP** If you are not already familiar with JavaBeans, then a quick-start tutorial is recommended. Many such resources are freely available on the web. You can start by looking at the JavaBeans 101 tutorial here: http://java.sun.com/developer/onlineTraining/Beans/bean01/

In the example, the developer has created a FileService Managed Bean class that connects to various online file services. However, each service that the Managed Bean can connect to can itself be configured differently. Instead of having to write a different application for each configuration, the developer can provide properties that the application administrator/owner can set, depending on the configuration of the desired service they are connecting to. Another convenience of Managed Bean Properties is that they can be directly referenced from within Server Side JavaScript code or EL property binding using simple notation: `beanName.propertyName`.

Listing 5.9 shows a snippet from `PCGCH05.nsf`'s `faces-config.xml`. Listing 5.10 illustrates how the Managed Bean Properties defined in `faces-config.xml` are referenced directly from within Server Side JavaScript code (in the **managedBeans** XPage in the same application).

Listing 5.9 Snippet from faces-config.xml in PCGCH05.nsf Application

```
<managed-bean>
  <managed-bean-name>myApplicationBean</managed-bean-name>
  <managed-bean-class>
  com.ibm.xpages.beans.MyApplicationBean</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>title</property-name>
    <value>#{javascript:database.getTitle()}</value>
  </managed-property>
</managed-bean>
```

Those with a keen eye will quickly notice one important subtlety in the definition of the `title` Managed Bean Property in Listing 5.9: The value of the property is computed using Server Side JavaScript.

As you can imagine, the potential uses for such functionality are far reaching; not only do Managed Bean Properties allow the developer to provide default properties for Managed Beans on an application-by-application basis, but these properties are also computable. The XPages runtime supports computing these property values using Server Side JavaScript or Expression Language. You can also use either approach to directly reference some other declared direct reference to Managed Beans, as with this:

```
<value>#{myOtherManagedBean.beanProperty}</value>
```

Using this approach, you should ensure that the other referenced Managed Bean instance will be within an available scope to the current bean. For example, a bean declared within the `applicationScope` that is configured to get the value of a `requestScope` bean property will not have access to this property. This is because the `applicationScope` bean is instantiated before the `requestScope` bean instance. Therefore, when the `faces-config.xml` property declaration is processed, an exception occurs because the `requestScope` bean property cannot be found. If this scenario were configured the other way, with the `requestScope` bean property configured to obtain the value of an `applicationScope` bean property, no exception would occur. This is because the `applicationScope` bean would be within scope at instantiation time of the `requestScope` bean.

Listing 5.10 demonstrates how both methods can be combined to access Managed Bean properties.

Listing 5.10 Snippet from the managedBeans XPage in PCGCH05.nsf Application Using Managed Bean Properties

```
<xp:tr>
  <xp:td>
  <xp:span style="font-weight:bold">
  myApplicationBean.title</xp:span>data binding</xp:td>
  <xp:td>
    <xp:text escape="true" id="computedField4"
    value="#{myApplicationBean.title}"></xp:text>
  </xp:td>
</xp:tr>
<xp:tr>
  <xp:td>
  <xp:span style="font-weight:bold">
  myApplicationBean.title</xp:span>Server Side JavaScript</xp:td>
  <xp:td>
    <xp:text escape="true" id="computedField5">
        <xp:this.value>
```

```
        <![CDATA[#{javascript:
            return "Using SSJS " + myApplicationBean.title}]]>
        </xp:this.value>
    </xp:text>
  </xp:td>
</xp:tr>
```

You can also set Managed Bean property values to be passed by reference of other bean instances. Listing 5.11 shows an example fragment from a `faces-config.xml` file where a bean property value is set to accept another bean instance. As described earlier, the rules of scoping also apply when using this approach.

Listing 5.11 Snippet from faces-config.xml Showing a Bean Instance Property Value

```
<managed-bean>
  <managed-bean-name>workFlowManagerBean</managed-bean-name>
  <managed-bean-class>
  com.ibm.xpages.beans.WorkFlowManagerBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>workFlowHelper</property-name>
    <value>#{workFlowHelperBean}</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>workFlowHelperBean</managed-bean-name>
  <managed-bean-class>
  com.ibm.xpages.beans.WorkFlowHelperBean</managed-bean-class>
  <managed-bean-scope>view</managed-bean-scope>
</managed-bean>
```

Finally, Managed Bean properties can be declared with an explicit property type. This is done by specifying the property type class in the `faces-config.xml` property declaration. Listing 5.12 shows an example in which the `serverName` property has its `<property-class>` set to `java.lang.String`. This is a standard Java language datatype and differs from the `appBean` property type, in that it is a user-defined type. Again, the `<property-class>` has been set to the fully qualified class name accordingly.

Listing 5.12 Snippet from faces-config.xml Showing a Bean Property Type Class Declaration

```
<managed-bean>
    <managed-bean-name>myRequestBean</managed-bean-name>
    <managed-bean-class>
        com.ibm.xpages.beans.MyRequestBean
```

```
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
      <property-name>serverName</property-name>
      <property-class>java.lang.String</property-class>
      <value>#{javascript:database.getServer()}</value>
    </managed-property>
    <managed-property>
      <property-name>appBean</property-name>
      <property-class>
          com.ibm.xpages.beans.MyApplicationBean
      </property-class>
      <value>#{myApplicationBean}</value>
    </managed-property>
</managed-bean>
```

PCGCH05.nsf contains several examples of how to use Managed Bean Properties. Over time, the use of such properties will prove to be an invaluable asset in your repertoire of XPages development skills.

Conclusion

This chapter took you on a grand tour of server-side programmability. As you can see, server-side scripting has many different facets and requires a certain amount of trial and error to build up the level of expertise required to exploit all the tools at your disposal. In recognition of the fact that your learning curve will inevitably lead to occasional unexpected results, the next chapter is dedicated to debugging XPages applications on the server side.

CHAPTER 6

Server-Side Debugging Techniques

As of Notes/Domino 8.5.3, Domino Designer has no Server Side JavaScript (SSJS) debugger. However, it has a powerful Java debugger. You already know from Chapter 5, "Server-Side Scripting," that you can write SSJS code to directly incorporate Java object code into your applications. This built-in Java debugger can be an essential tool when it comes to debugging Java code called from SSJS. But what about debugging the actual script code itself? In the absence of a full-blown Server Side JavaScript debugger, you can still help yourself by using some simple XPages features. This chapter starts by describing two useful server-side global functions and a simple programming construct that you can readily use as a debugging aid in your SSJS code—hence the title of the next section, "The 'Poor Man's' Debugger." It then moves on to look at the various tools and tricks you can employ to debug everything from your own custom Java classes, to Managed Beans, to plug-ins that extend the XPages runtime.

Before starting this chapter, you should download the **PCGCH06.nsf** supporting application and then open and sign it with your own Notes ID in Domino Designer. You will then have all the examples covered here. The sample application is available from this website: www.ibmpressbooks.com/title/0132943050.

The "Poor Man's" Debugger

In Domino Designer, open the **poorMansDebugger.xsp** page found in **PCGCH06.nsf** and run it in a browser to work through the examples discussed here. Figure 6.1 shows the **poorMansDebugger** XPage in a browser.

print(message) : void & println(message) : void

This is the XPages Server Side JavaScript equivalent of the `XSP.log()` Client Side JavaScript XSP Object function. The given `message` parameter is typecast to a `String` value and output to the console writer. For an XPages application running in a Domino server, this output is sent to the Domino server console window and also to the Domino console log file, which can be found under the Domino data folder in **IBM_TECHNICAL_SUPPORT/console.log**. For an XPages application running in the Notes client, this output can be viewed using the **Help > Support > View Trace** menu option. Ultimately, all Notes trace files reside under the Notes data folder in **workspace/log/tracelog-*n*.xml** (where *n* is replaced with an integer value between 0 and 9, 0 being the log for the most recent Notes client session). If you have launched your Notes or Domino Designer client using the `RPARAMS -console` switch (as described in Chapter 3, "Working with the Console"), the `print()` or `println()` output will also be readable in the OSGi Console window. Both the `print()` and `println()` functions produce the same output.

Developers should use debug constructs in their code to avoid having print statements execute throughout production code, as this can have an impact on performance (where large recursive operations are writing to the log file).

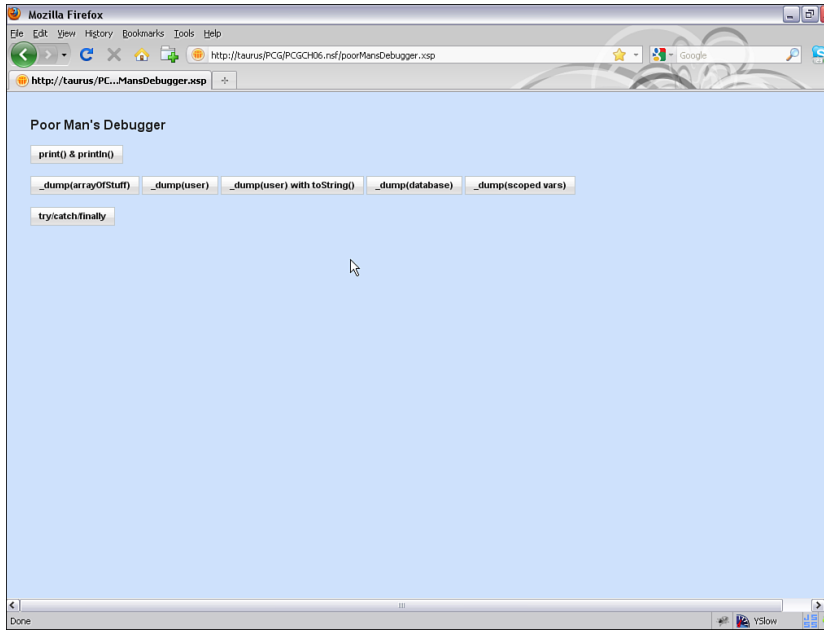


Figure 6.1 The poorMansDebugger XPage in a browser

Listing 6.1 details example calls to the `print()` and `println()` server-side functions.

Listing 6.1 Example Calls of the print () and println() Functions

```
<xp:button id="button1">
     <xp:this.value>
          <![CDATA[print() & println()]]>
     </xp:this.value>
     <xp:eventHandler event="onclick" submit="true"
          refreshMode="complete">
          <xp:this.action>
               <![CDATA[[#{javascript:
                    print("UserName: " + @UserName());
                    println("Database Name: " + database.getFileName());
                    print("10.5 x 10.5 = " + (10.5 * 10.5));
               }]]>
          </xp:this.action>
     </xp:eventHandler>
</xp:button>
```

Listing 6.2 shows the console output after running the `print()` and `println()` Server Side JavaScript functions, as detailed in Listing 6.1.

Listing 6.2 Console Output of the print() and println() Functions

```
HTTP JVM: UserName: Anonymous
HTTP JVM: Database Name: PCGCH06.nsf
HTTP JVM: 10.5 x 10.5 = 110.25
```

_dump(object) : void

This is the XPages Server Side JavaScript equivalent of the `XSP.dumpObject()` Client Side JavaScript XSP Object function. The given `object` parameter is interrogated for presence of a `toString()` method on the object. If present, this is invoked. Also, an internal reflection call occurs on the `object` parameter to summarize the member/values of the `object`. The output behavior is the same as for the `print()` function.

Listing 6.3 details an example call and console output of the `_dump()` server-side function. A simple one-dimensional array object is passed into the function.

Listing 6.3 Example Call of the _dump() Function for a Simple 1D Array Object

```
<xp:button value="_dump(arrayOfStuff)" id="button5">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                var arrayOfStuff:Array = new Array();
                arrayOfStuff[0] = "hello";
                arrayOfStuff[1] = "world";
                _dump(arrayOfStuff);
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

Listing 6.4 shows the sample output is recorded to log after executing the script in Listing 6.3.

Listing 6.4 Sample Output from Call of _dump() Function for a Simple 1D Array Object

```
#1  HTTP JVM: Array
#2  HTTP JVM: =
#3  HTTP JVM: hello,world
#4  HTTP JVM:
#5  HTTP JVM: +-
```

```
#6  HTTP JVM: [0]
#7  HTTP JVM: :
#8  HTTP JVM: string
#9  HTTP JVM: =
#10 HTTP JVM: hello
#11 HTTP JVM:
#12 HTTP JVM: +-
#13 HTTP JVM: [1]
#14 HTTP JVM: :
#15 HTTP JVM: string
#16 HTTP JVM: =
#17 HTTP JVM: world
```

In the case of Listing 6.4, line numbers (such as `#1`) have been manually inserted for clarity. The same is true for the remaining listings in this section that list Domino console output. In this instance, the `toString()` method is found on the `Array` object. This results in the output `hello,world` in Listing 6.4, line 3. Thereafter, the internal reflection member/values are seen in the console output. This results in output for each of the object members. Line 8 shows a string type member with the value `hello` in line 10, and so forth.

Listing 6.5 details an example call to the `_dump()` server-side function where a custom Server Side JavaScript object is passed into the function. This custom object does not declare a `toString()` method in its class. This results in reflection output only on the console output, as shown in Listing 6.6. Interestingly, you also see reflection output on the console for the `commonName` function definition—this shows that more than just simple values are visible using the `_dump()` function.

Listing 6.5 Example Call of the _dump() Function for a Custom Server Side JavaScript Object Without a toString() Method Available

```
<xp:button value="_dump(user)" id="button2">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                var User = function(){
                    var _cn = session.getCommonUserName();
                    function _CN(){return _cn;}
                    return {commonName : _CN}
                }
                var user = new User();
                _dump(user);
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

Listing 6.6 shows the console output of running the code snippet in Listing 6.5.

Listing 6.6 Sample Console Output of the _dump() Function for a Custom Server Side JavaScript Object Without a toString() Method Available

```
#1  HTTP JVM: Object
#2  HTTP JVM: =
#3  HTTP JVM: [object Object]
#4  HTTP JVM:
#5  HTTP JVM: +-
#6  HTTP JVM: commonName
#7  HTTP JVM: :
#8  HTTP JVM: Function
#9  HTTP JVM: =
#10 HTTP JVM: [function Function]
#11 HTTP JVM:
#12 HTTP JVM:
#13 HTTP JVM: +-
#14 HTTP JVM: prototype
#15 HTTP JVM: :
#16 HTTP JVM: Object
#17 HTTP JVM: =
#18 HTTP JVM: [object Object]
```

Listing 6.7 details an example call of the `_dump()` server-side function in which a custom Server Side JavaScript object is passed into the function. The difference between this example and the previous one in Listing 6.5 is the fact that this time the custom object declares a `toString()` method in its class. This results in the reflection output showing both the output from the `toString()` invocation and the standard reflection information on the console output, as shown in Listing 6.8.

Listing 6.7 Example Call of the _dump() Function for a Custom Server Side JavaScript Object with a toString() Method Available

```
<xp:button value="_dump(user) with toString()" id="button3">
     <xp:eventHandler event="onclick" submit="true"
          refreshMode="complete">
          <xp:this.action>
               <![CDATA[#{javascript:
                    var User = function(){
                         var _cn = session.getCommonUserName();
                         function _CN(){return _cn;}
                         function _tos(){return "commonName(_cn)=" + _cn;}
                         return {
                              commonName : _CN,
```

```
                        toString : _tos
                    }
                }
                var user = new User();
                _dump(user);
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

Listing 6.8 shows the console output that results from executing the code snippet in Listing 6.7.

Listing 6.8 Example Console Output from Executing the _dump() Function for a Custom Server Side JavaScript Object with a toString() Method Available

```
#1  HTTP JVM: Object
#2  HTTP JVM: =
#3  HTTP JVM: commonName(_cn)=taurus
#4  HTTP JVM:
#5  HTTP JVM: +-
#6  HTTP JVM: commonName
#7  HTTP JVM: :
#8  HTTP JVM: Function
#9  HTTP JVM: =
#10 HTTP JVM: [function Function]
#11 HTTP JVM:
#12 HTTP JVM:
#13 HTTP JVM: +-
#14 HTTP JVM: prototype
#15 HTTP JVM: :
#16 HTTP JVM: Object
#17 HTTP JVM: =
#18 HTTP JVM: [object Object]
#19 HTTP JVM:
#20 HTTP JVM: +-
#21 HTTP JVM: toString
#22 HTTP JVM: :
#23 HTTP JVM: Function
#24 HTTP JVM: =
#25 HTTP JVM: [function Function]
#26 HTTP JVM:
#27 HTTP JVM:
#28 HTTP JVM: +-
#29 HTTP JVM: prototype
```

```
#30 HTTP JVM: :
#31 HTTP JVM: Object
#32 HTTP JVM: =
#33 HTTP JVM: [object Object]
```

The `toString()` output is listed as `HTTP JVM: commonName(_cn)=taurus` at line 3 in Listing 6.8.

Listing 6.9 details an example call to the `_dump()` server-side function in which the Server Side JavaScript `database` object is passed into the function. Note the `print()` call just before the `_dump()` call. This highlights the fact that the `toString()` method output is equivalent within each of these calls—one invoking it explicitly, the other invoking it implicitly.

Listing 6.9 Example Call of the _dump() Function for the Server Side JavaScript Database Object

```
<xp:button value="_dump(database)" id="button4">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                print("database.toString() = " + database.toString());
                _dump(database);
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

Listing 6.10 shows the console output from running the code snippet in Listing 6.9. Note the presence this time of the fully qualified datatype `com.ibm.domino.xsp.module.nsf.NSFComponentModule$XPagesDatabase` for the `database` object. This is inline with the expected output of the `_dump()` function.

Listing 6.10 Console Output from Calling the _dump() Function for the Server Side JavaScript Database Object

```
HTTP JVM: database.toString() = PCG\PCGCH06.nsf
HTTP JVM: com.ibm.domino.xsp.module.nsf.NSFComponentModule$XPages
➥Database
HTTP JVM: =
HTTP JVM: PCG\PCGCH06.nsf
```

Listing 6.11 details an example call to the `dump()` server-side function in which the Server Side JavaScript `requestScope` and `viewScope` objects are passed into the function. The `requestScope` object contains several items based on the current request,

as shown in Listing 6.12. The `viewScope` in this example has been assigned a custom variable that also appears in the console output.

Listing 6.11 Example Call of the _dump() Function for the Server Side JavaScript requestScope and viewScope Objects

```
<xp:button value="_dump(scoped vars)" id="button5">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                print("requestScope _dump:");
                _dump(requestScope);
                print("viewScope _dump:");
                viewScope.pi = 3.14159265;
                _dump(viewScope);
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

Listing 6.12 shows the sample console output after running the code snippet in Listing 6.11.

Listing 6.12 Console Output of the _dump() Function for the Server Side JavaScript requestScope and viewScope Objects

```
#1  HTTP JVM: requestScope _dump:
#2  HTTP JVM: com.sun.faces.context.RequestMap
#3  HTTP JVM: =
#4  HTTP JVM: {cookie={SessionID=javax.servlet.http.Cookie@17a417a4},
#5  context=com.ibm.xsp.designer.context.ServletXSPContext@9d009d,
#6 _xspconvid=null,
#7  componentParameters=com.ibm.xsp.application.ComponentParameters@288,
#8  database=PCG\PCGCH06.nsf, com.ibm.xsp.SESSIO_ID=CYYUY5WTEG,
#9  session=CN=taurus/O=renovations}
#10 HTTP JVM: viewScope _dump:
#11 HTTP JVM: javax.faces.component.UIViewRoot$ViewMap
#12 HTTP JVM: =
#13 HTTP JVM: {pi=3.14159265}
```

Using try/catch Blocks

A simple technique you can apply to your Server Side JavaScript code is the use of `try/catch` blocks. An optional `finally` clause also is available for a `try/catch` block. The principles behind this programming construct are the same as in C++, Java,

and any other programming language that supports it. Basically, code within the `try` clause is executed within a reversible instruction stack. If the code causes an error, the instruction stack is unwound and the `catch` clause is entered. Within the `catch` clause, you can then handle the error accordingly by outputting debug statements and so forth. You also have access to the raised error object through the `catch` error parameter—note that this can be any arbitrarily named reference. From this reference, you can obtain the error message implicitly by simply referencing the error parameter or explicitly by calling `e.getMessage()` (where `e` is the arbitrarily named error reference used in this example).

Listing 6.13 details an example `try/catch/finally` block along with console output.

Listing 6.13 Example try/catch/finally Block

```
<xp:button value="try/catch/finally" id="button2">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
            <![CDATA[#{javascript:
                var x = @Now();
                try{
                    x = someNonExistentObject;
                }catch(e){
                    print("error occurred: " + e);
                }finally{
                    // finally clause is optional
                    print("now: " + x);
                }
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

Listing 6.14 shows the console output from executing the code snippet in Listing 6.13.

Listing 6.14 Console Output from Executing a try/catch/finally Code Block

```
HTTP JVM: error occurred: 'someNonExistentObject' not found
HTTP JVM: now=30/09/11 11:38
```

How to Set Up a Server for Remote Debugging

The vast majority of XPages applications are run on a Domino server. Sooner rather than later, the XPages developer must debug an application that does not run on a local computer. When that time arrives, the XPages developer needs to be equipped with the know-how and confidence to connect to a remote server and debug the issue.

It is worth clarifying at this juncture that, in Java terms, to debug remotely simply means to debug code that is running in a separate JVM instance. That is, the Domino server might be on the same machine as Domino Designer, but it requires a remote Debug Configuration to debug the code on the Domino server.

To debug an application running remotely, the developer or administrator must first set up the server in such a way that it can be debugged. This is achieved by adding a number of variables to the server's **notes.ini** file, some of which Chapter 2, "Working with Notes/Domino Configuration Files," briefly touched upon.

Table 6.1 describes the various **notes.ini** variables that can be added to allow the server to be debugged. None of these are specified in **notes.ini** by default. After these variables are added to the **notes.ini** file, the HTTP task must be restarted for the JVM to start in debug mode.

Table 6.1 NOTES.INI Java Debug Variables

Variable Name	Description
JavaEnableDebug	Notifies the Domino server that it should start the JVM in debug mode. Specifying a value of `1` enables this setting.
JavaDebugOptions	Provides a comma-separated list of arguments to the JVM.
JavaOptionsFile	Provides a full path to a text file that contains a list of parameters to be passed to the JVM as VM args.

The `JavaEnableDebug` and `JavaDebugOptions` variables are utility variables that enable the user to pass debug arguments to the JVM without having to specify a `JavaOptionsFile`. In some cases when it is necessary to pass a number of arguments to the JVM, a more convenient method is to use the `JavaOptionsFile` parameter in place of the `JavaEnableDebug` and `JavaDebugOptions` parameters. Listing 6.15 shows a sample application of the `JavaEnableDebug` and `JavaDebugOptions` variables.

Listing 6.15 notes.ini Snippet to Enable Java Debugging on the Server

```
# Enable server's JVM to start in debug
JavaEnableDebug=1
JavaDebugOptions=transport=dt_socket,server=y,suspend=n,
➥address=8000
```

Listing 6.15 shows how to start the server's JVM in debug mode. Setting `JavaEnableDebug` to `1` tells the server to start the JVM instances in debug mode by passing the `-Xdebug` VM argument to the JVM instance. The parameters set with the `JavaDebugOptions` variable are passed directly to the JVM as options for the `-Xrunjdwp` VM argument. Passing the `-Xdebug` and `-Xrunjdwp` arguments to the JVM causes the JVM to start in debug mode in a state to accept Java debug connections.

TIP You can start the local preview *server* (the server that is used when you preview an application on your local computer instead of a server) in debug mode by adding the same variables to the **notes.ini** located in your Notes client program directory (`JavaEnableDebug` and `JavaDebugOptions`). If you are debugging applications on a Domino server and on the local preview server at the same time, you should set the debug address for the local preview server to be a different number than the Domino server—for instance, 8005 instead of 8000. This requires modifying the port number in the Debug Configuration used in Domino Designer also.

All the parameters passed to the JVM via the `JavaDebugOptions` variable must conform with the official Oracle VM invocation options. Table 6.2 describes the various options that were passed to the JVM in Listing 6.15.

Table 6.2 JavaDebugOptions Parameters

Option Name	Required	Description
transport	Yes	Name of transport method to use when connecting to the debugger application. Typically set to `dt_socket`, which is the default transport socket for debugging Java applications.
server	Yes	In the case of the Domino server, this option should *always* be set to `true`. This option notifies the JVM that it is acting as a debug *server*—that is, it will be listening for debugger connections.
suspend	No	Defaults to `y` (yes). If set to `y`, all threads within the JVM are suspended on startup *until* a debugger connects to the JVM. In general, setting this option to `y` is advisable only when an issue is occurring during initialization; otherwise, the server will not complete initialization until a debugger connects to the JVM.
address	Yes	Tells the server's JVM which port to listen to for debugger connection requests.

Table 6.1 also lists a third parameter that can be used to enable a server to be remotely debugged. `JavaOptionsFile` enables the user to set a file path to a text file that contains all the debug options that the user wants to pass into the server's JVM to enable debugging. Listing 6.16 shows a sample usage of the `JavaOptionsFile` **notes.ini** variable.

Listing 6.16 notes.ini Snippet to Enable Java Options File on the Domino Server

```
# Enable server to read from options file
JavaOptionsFile=C:\Domino\MyJavaOptions.txt
```

Listing 6.17 shows sample content for the `JavaOptionsFile` variable (in this case, the `MyJavaOption.txt` file).

Listing 6.17 Possible Value of Java Options File to Enable Server-Side Debugging

```
# Enable server to read from options file
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000
```

The net result of Listings 6.16 and 6.17 is the same as that of Listing 6.15. In both cases, the server's JVM is started in debug mode. The case for using the `JavaOptionsFile` variable is more often made when it is necessary to pass many VM arguments to the JVM upon startup, such as when you need to profile your XPages application using a profiling tool such as the XPages Toolbox. This is described in the "XPages Toolbox" section at the end of this chapter.

After your server has been configured to start in debug mode, a message is printed to the server console. You should verify that such a message has been received before you attempt to connect a debug client to the server. Figure 6.2 illustrates how the console looks when the server is ready to accept debug connections.

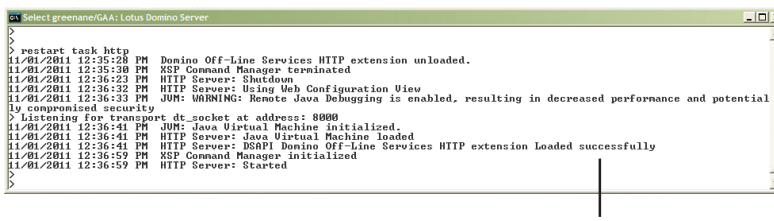


Figure 6.2 Domino server's HTTP task started in debug mode

Debugging Java Code and Managed Beans

After the developer/administrator has started the Domino server in debug mode, the server is in a position to accept debug connections from your Java debug client of choice. For XPages developers, this is likely Domino Designer. As of Notes/Domino 8.5, Domino Designer is built upon the Eclipse IDE platform. Eclipse is equipped with a rich set of Java debugging features, all of which are available to Domino Designer users.

When developers are creating and debugging their own XPages Extension Library plug-ins, they might want to use the Eclipse IDE to develop and debug XPage library plug-ins instead of using Domino Designer. After all, in this situation, the developer is dealing with pure Eclipse plug-ins. Both scenarios are outlined in the following sections.

In the first scenario, a user has created a Java class that contains some application logic. The Java class is referenced from within some Server Side JavaScript inside an XPage. After executing the business logic, the developer realizes that the Java code is not producing the correct results. To make this use case easier to understand, the supporting **PCGCH06.nsf** application has an XPage and Java class that you can use to follow along. This example is designed to work in Domino and Domino Designer 8.5.3

or later. Because this chapter is dedicated to debugging applications on the server, you need to copy the sample application onto your Domino server, start it in debug mode (as described in the previous sections), and open the application within Domino Designer. The XPage in question is named **remoteJavaDebugger**, and the Java class is called **DateCalculator**. Figure 6.3 shows both of these design elements within Domino Designer.

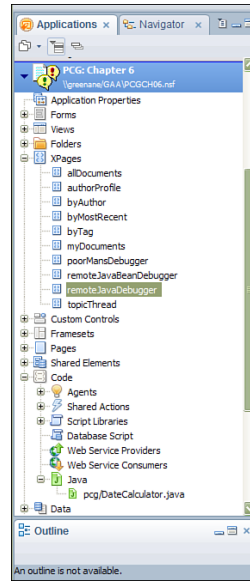


Figure 6.3 remoteJavaDebugger and DateCalculator in the Domino Designer Applications Navigator

To debug the business logic, the developer must first open the suspect Java class, **DateCalculator**. For the purposes of this example, you should set a breakpoint in the first line of the offending method; in this example, this is line 12, highlighted with a shaded background in Listing 6.18.

Listing 6.18 Contents of DataCalculator.java Design Element

```
1 package pcg;
2
3 import java.util.Calendar;
4 import java.util.Date;
5 import java.util.GregorianCalendar;
6
7 public class DateCalculator {
8     /*
9      * Add 5 years and 5 months to the start date
```

```
10       */
11      public static Date computeSabbatical(Date startDate){
12           if(startDate == null){ // <-- SET BREAKPOINT AT THIS LINE
13                return null;
14           }
15           Calendar calendar = new GregorianCalendar();
16           calendar.setTime(startDate);
17           calendar.add(Calendar.YEAR, 5);
18           calendar.add(Calendar.MONTH, 4); // <-- error typo
19           return calendar.getTime();
20      }
21 }
```

To set a breakpoint, the developer simply right-clicks in the leftmost column of the Java Editor and selects the **Toggle Breakpoint** menu item, as illustrated in Figure 6.4.

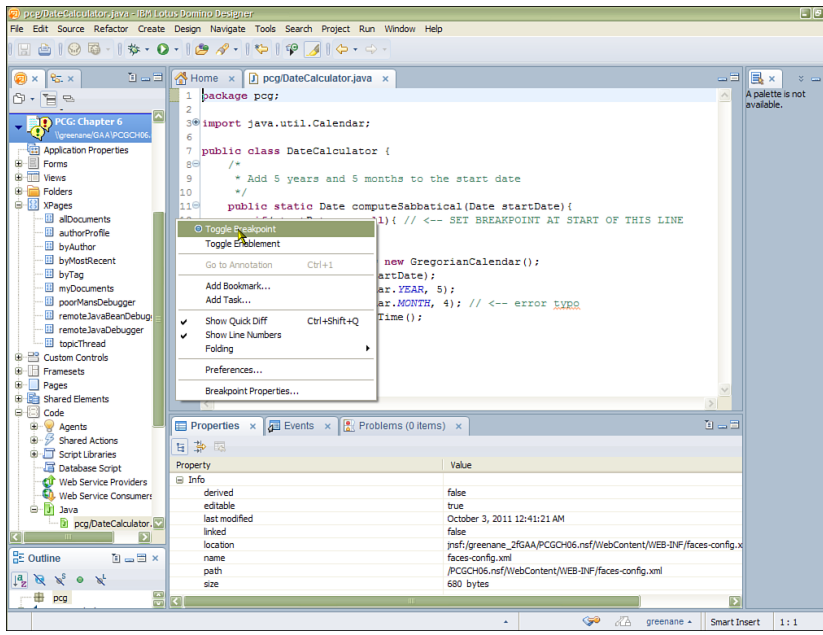


Figure 6.4 Enabling a breakpoint in the Domino Designer Java Editor

When the breakpoint is set, the next step is to create a Debug Configuration. This is where all your debug settings are stored and applied whenever your application is launched in debug mode. To do this, you should select the **Run > Debug Configurations** menu option, as shown in Figure 6.5. To make this menu option available, simply open the Java editor (by opening a Java class), or switch to the Debug perspective by using the

Window > Show Perspective > Other menu item and choosing the **Debug** perspective from the resulting dialog. In Eclipse, a perspective is a particular configuration of views and editors designed for a particular purpose. Domino Designer has its own "Domino Designer" perspective, which you probably work in all the time. There is another perspective for Java developers, a specialized perspective for debugging, and so forth.

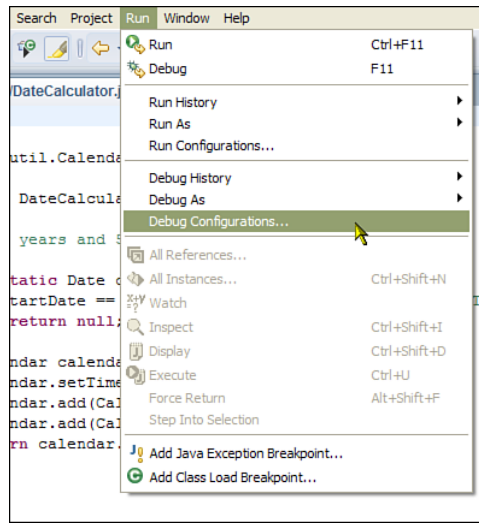


Figure 6.5 Debug Configurations menu item in Domino Designer

This launches the **Debug Configurations** dialog. As explained earlier in the chapter, you need to create a Remote Java Application Debug Configuration because the XPages runtime is not running within the same JVM instance as Domino Designer. To create a new Remote Java Application Debug Configuration, simply right-click the Remote Java Application item in the list of possible Debug Configurations and click the **New** menu item, as shown in Figure 6.6.

Depending on the context from which you created the Debug Configuration, the fields of the Debug Configuration dialog might be prepopulated. If they are not, you must provide a **Configuration Name**. The **Project Name** should be set as the name of the database (project) that contains the design element being debugged—in this example, **PCGCH06.nsf**. You need to select a `Connection Type`, which should always be set to `Standard (Socket Attach)`. Finally, you must enter a `Host Name` and `Port Number` to connect to. If the server you are debugging is running on the local machine, **localhost** will suffice as the hostname; otherwise, a qualified IP address is required. The port number supplied must be identical to the host number configured as the listen port in **notes.ini** on the server, as shown in Listing 6.17 (in this case, `8000`). Figure 6.7 shows how the debug configuration for the PCGCH06.nsf application should be set up when the application resides on a server running on the local machine.

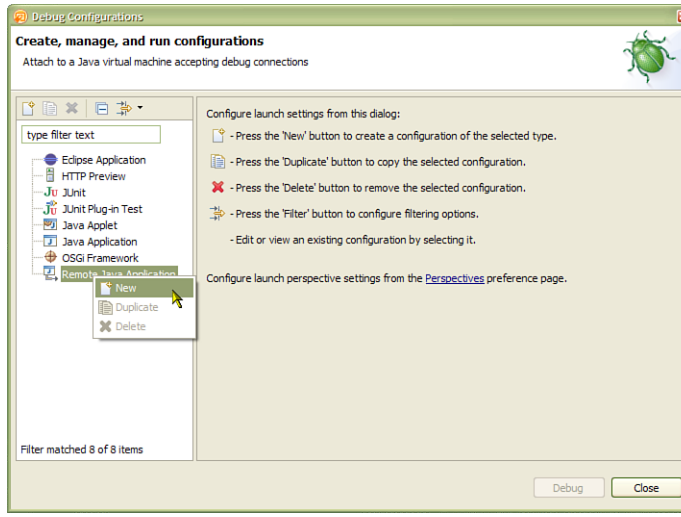


Figure 6.6 Creating a Remote Java Application Debug Configuration in Domino Designer

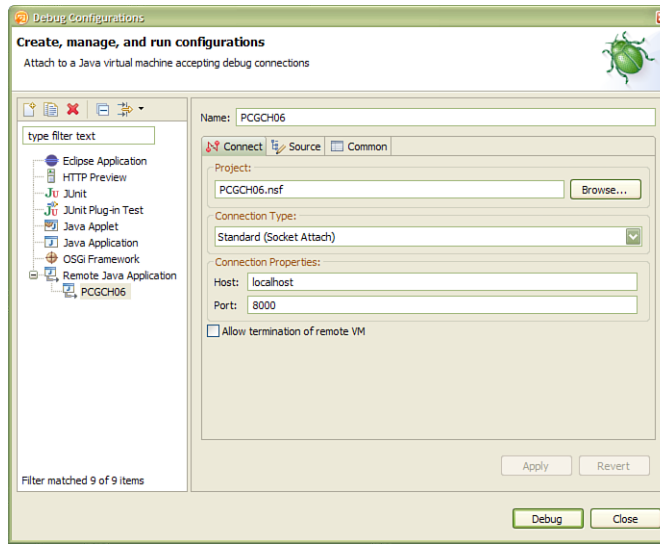


Figure 6.7 A Remote Java Application Debug Configuration in Domino Designer

Congratulations, you are now ready to start debugging! When you have confirmed that the Domino server has been restarted with its Java debug INI variables in place, the next step is to launch the Debug Configuration you have just created. To do this, simply bring up the **Debug Configurations** dialog, select the newly created configuration, and press the **Debug** button. If you have not already switched to the Debug perspective, you

should do so now. Figure 6.8 shows the Debug perspective in Domino Designer when the Java Debugger is connected to a remote JVM.

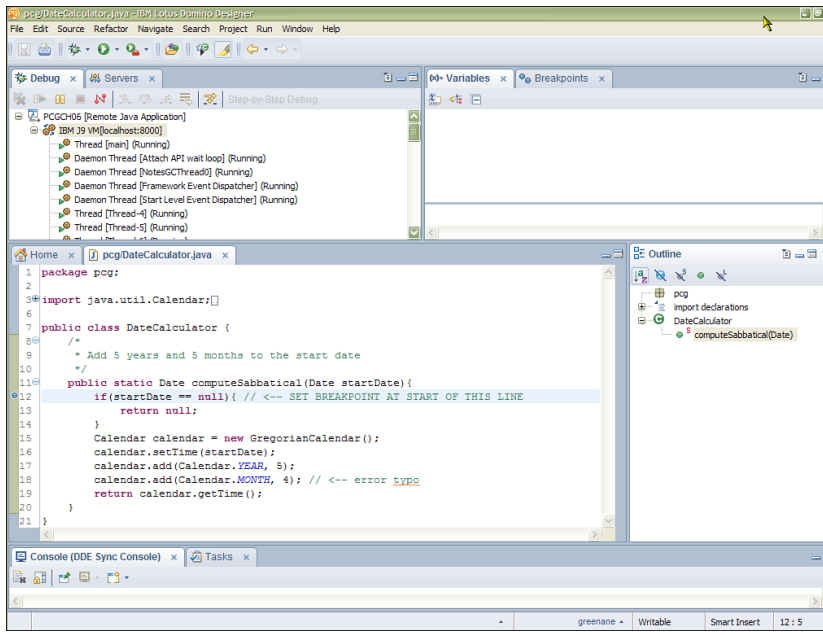


Figure 6.8 Domino Designer connected to a remote JVM

To debug the Java code, simply attempt to preview the **remoteJavaDebugger** XPage in a web browser. This causes the Java class to be invoked, and the debugger pauses execution of the Java code at the exact place where the breakpoint is set, as shown in Figure 6.9. Eclipse provides Domino Designer with a slew of Eclipse Views that make it easy to debug the Java code. From this point, you can debug through the Java code just as you would with any regular Java application.

It is beyond the scope of this book to provide exhaustive Java debugging techniques when debugging XPages Java code. However, we recognize that many XPages developers might be new to Java development, so some basic tips on how to get up and running in the Eclipse Debug perspective might be helpful. Figure 6.10 provides a number of callouts highlighting various parts of the Eclipse Debug view and the Eclipse Variables view.

Table 6.3 lists all the areas highlighted in Figure 6.10 and gives a brief description of the functionality provided by the same.

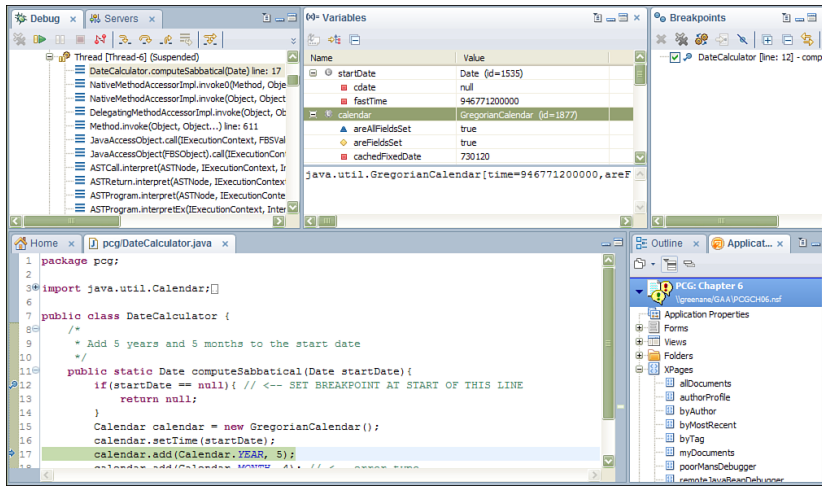


Figure 6.9 Domino Designer's Debug, Variables, and Breakpoint views

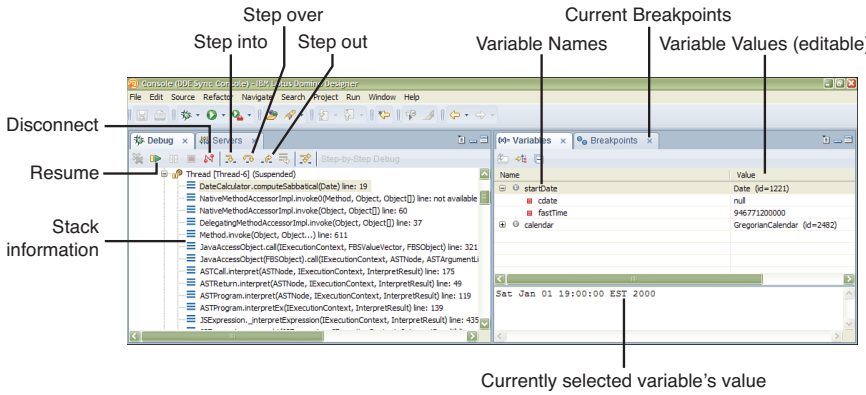


Figure 6.10 Eclipse Debug view and Eclipse Variables view

Table 6.3 Descriptions of Debug and Variables Views UI

UI Item	Description
Stack information	Provides a full view of the stack of method and class calls made to arrive at the current breakpoint's position.
Resume	Results in the JVM continuing to execute the program as normal until the next breakpoint is encountered.
Disconnect	Responsible for terminating the connection between the debugger client and debug server—in this case, between Domino Designer and the Domino server.

UI Item	Description
Step into	Tells the debugger that the user wants to step into the current method or constructor. If the Java source code for the method/constructor is not available, the user is presented with the Eclipse Java Class editor. This editor basically lists some rudimentary information about the class file but does not enable the user to read the Java source code for the class.
Step over	Tells the debugger to step over the current Java statement. The current Java statement is evaluated and the program progresses to the next statement to be executed.
Step out	Tells the debugger to step out of the current method or constructor. All remaining statements in the current method/constructor are executed and debugging resumes at the next statement in the class/method that called the current method/constructor.
Variable names	Column in the Variables Eclipse view that lists the names of all the variables currently in scope. These variables include variables defined within the current method, along with class variables that are in scope.
Variable values	Column in the Variables Eclipse view that lists the values of all the variables that are currently in scope. It is worth noting one powerful feature of this column makes it possible to change the value of primitive variables (and Strings) via this column. The developer can change the current value of a variable simply by double-clicking in the value column of the variable to edit. To set the value of an Object, the developer can right-click the variable and select the **Change Value** menu option. This feature provides a huge amount of flexibility to developers by enabling them to change the values of variables within the code as the code is executing.
Currently selected variable's value	Shows the value of the currently selected variable as computed using the variable's `toString()` method. If the selected Object does not override the `toString()` method and the developer wants to obtain some useful debug information from the selected item, the developer can optionally override the `toString()` behavior by providing a Detail Formatter for the selected variable. This is done by right-clicking a variable and selecting the **New Detail Formatter** context menu item. This is another powerful function that developers will make much use of over time.

Table 6.3 Descriptions of Debug and Variables Views UI (continued)

UI Item	Description
Current breakpoints	Lists all the breakpoints currently configured within the application. A significant amount of hidden functionality also is available within this view. Each breakpoint can be configured to be enabled only when certain scenarios are met. These scenarios can be as simple as a certain hit count being met (that is, the breakpoint has been reached a specific number of times) to as complex as the breakpoint being dependent on the values of certain variables within the system. These conditions can be set on a breakpoint by right-clicking the breakpoint and choosing the **Breakpoint Properties** context menu item.

When debugging issues within XPages applications, developers often find that the application is failing to function because an exception is being generated either within the code or within one of the phases in the JSF lifecycle. The Eclipse debugging functionality that is embedded within Domino Designer provides one powerful feature in this regard: the capability to set breakpoints that are triggered when a specified type of exception occurs. As you can imagine, this feature is particularly useful when debugging stack traces where unfamiliar code is involved. In such cases, the developer can simply set a breakpoint for the type of exception being generated and perform the action that causes the exception to occur for the breakpoint to be invoked. To make this concept a little easier to understand, the accompanying application (**PCGCH06.nsf***)* has a simple example built in.

To see this example in action, simply preview the **remoteJavaDebugger** XPage (on a server running in debug mode). After the page renders in the browser, click the **Generate FacesExceptionEx** button. The XPage then generates an exception. We provide information and tips on how to read a stack trace such as the one generated in this situation (see "Interpreting a Stack Trace: Where to Go from Here?") later in this chapter; for now, scroll to the bottom of the stack trace to find the name of the type of exception that caused the page to fail to render. You will find that a `FacesExceptionEx` caused the failure. This is identified by the following line in the stack trace:

```
com.ibm.xsp.FacesExceptionEx: Demo exception; failed to complete
➥request
```

You are now armed with a vital piece of information. You know that a `com.ibm.xsp.FacesExceptionEx` exception is causing the page to fail to render. To backtrack through this issue, the first step is to go to the Debug perspective in Domino Designer. From the Debug perspective, locate the Breakpoints Eclipse view. The toolbar button farthest to the right enables you to add a breakpoint based on any type of exception. Figure 6.11 highlights this button and the dialog that results from pressing the button.

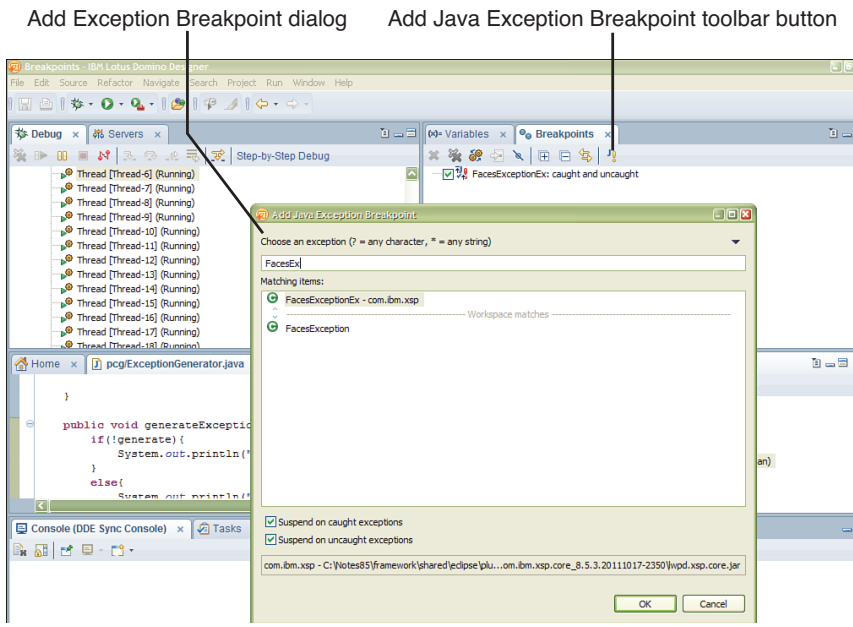


Figure 6.11 Adding a Java Exception breakpoint in Domino Designer

The **Add Java Exception Breakpoint** dialog enables the developer to enter the name of the type of exception being thrown—in this case, `FacesExceptionEx`. When set, this breakpoint will be hit any time the specified exception is thrown. Developers can add several such breakpoints to catch any range of exceptions being generated. In this example, we are concerned with only one type of exception (the `com.ibm.xsp.FacesExceptionEx` exception). When this breakpoint is set, the user must reopen the **remoteJavaDebbuger** XPage in a browser and again press the **GenerateFacesExceptionEx** button. After the button is pressed, the Domino Designer Java debugger kicks in and pauses program execution at the point in the code where the exception is about to be thrown (see Figure 6.12).

When the breakpoint is hit, the developer is free to debug the code in the same way other Java classes are debugged.

The breakpoints detailed earlier in this chapter are static in nature. However, breakpoints that are invoked as a result of encountering a certain type of exception are dynamic in nature and provide a great deal of flexibility to developers when debugging issues for which the full source for the application is not available

All XPages developers should be aware of one particular pitfall when debugging Java code. Setting up the server to start in debug mode implies that all JVMs running on the server will be started with the same debug settings. If the server has databases that have

Java agents that happen to be running when the developer attempts to launch the Debug Configuration, the debugger within Domino Designer might connect to the wrong JVM. If Domino Designer is indicating that it is connected to the remote JVM, but the debugger is *not* stopping at its breakpoints, the debugger likely has connected to the wrong JVM. To resolve this, disconnect the debugger and, if possible, change the schedule of the Java agents on the server so they do not conflict with your XPage debug session. Alternatively, you may temporarily switch off the Agent Manager by running the following command in the Domino server console:

```
tell AMgr quit
```

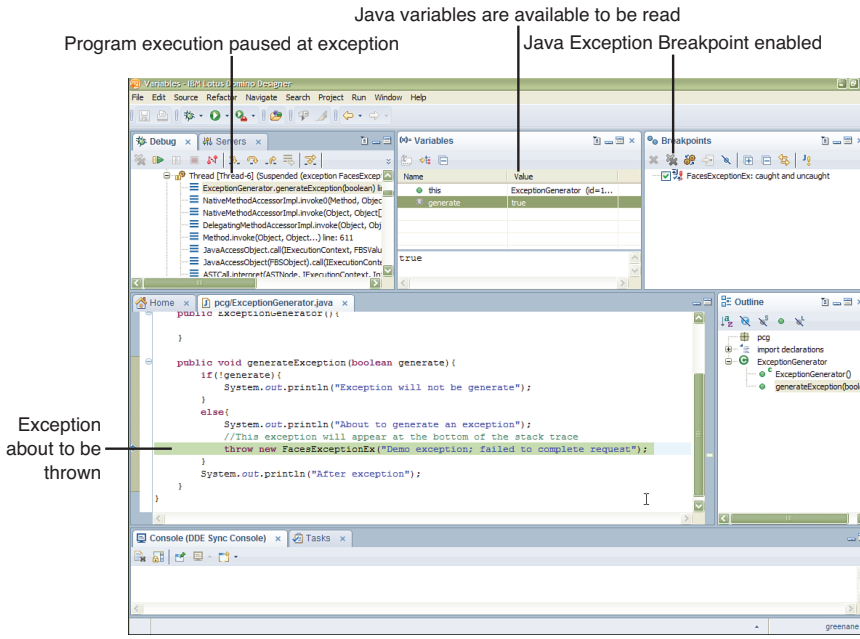


Figure 6.12 Java Exception breakpoint in action

This command tells the server to stop the Agent Manager task. Note that all agents (not just Java agents) will be disabled until the Agent Manager is re-enabled. To reload the Agent Manager, simply type the following command into the Domino server's console:

```
load AMgr
```

As described in Chapter 5, it is possible to add Java application logic to your XPage applications through the use of Managed Beans. The good news for the XPage developer is that the method of debugging Managed Beans is identical to the process used for debugging Java classes, as previously described in this chapter.

PCGCH06.nsf contains an XPage and Managed Bean Java class that can be debugged in a manner similar to that just described here. In this case, set a breakpoint at the start

of the `getFahrenheit()` method in the `Temperature` Managed Bean—or, to be completely explicit:

```
com.ibm.xpages.beans.TemperatureBean.getFahrenheit()
```

After the breakpoint is set and the Debug Configuration has been launched, simply run the **remoteJavaBeanDebugger** XPage in a web browser to debug the Managed Bean. Over time, this means of debugging server-side Java code residing within the application will become second nature and prove to be an invaluable skill.

Debugging XPages Extension Plug-ins

As the creation of XPage controls becomes more mainstream, developers will require more powerful tools to quickly deploy and debug their extension library plug-ins. Enter the IBM Lotus Domino Debug plug-in. The IBM Lotus Domino Debug plug-in is an Eclipse update site that adds powerful capabilities to the Eclipse IDE. As discussed in Chapter 2, the OSGi runtime is layered on top of the Domino HTTP task. The XPages runtime is packaged as OSGi bundles (plug-ins) that run within the OSGi platform. Similarly, any extensions to the XPages runtime are also packaged as OSGi bundles.

The IBM Lotus Domino Debug plug-in enables developers to create a Debug Configuration (similar to the Debug Configuration previously discussed in this chapter) from within the Eclipse IDE. This makes the OSGi platform running on the Domino server aware of the plug-ins within the developer's workspace. This offers many benefits, the most significant being that developers can debug their plug-ins directly from their workspace without needing to export and deploy the plug-ins every time a change is made. The IBM Lotus Domino Debug plug-in is an OpenNTF project, freely available for download. It can be found at OpenNTF.org: http://tinyurl.com/3jxsvux.

The project is accompanied by complete documentation detailing how to install the tool into Eclipse. It is worth going over some of the higher-level concepts of the tool so that you can understand the benefits of the tool.

This tool was developed solely to make the development of extension plug-ins easier. The tool enables developers to create an OSGi Debug Configuration that sets a host of settings within the Domino server's OSGi platform. The settings instruct the platform to read plug-ins from a second location, the developer's own workspace. The workspace in question must be on the same computer as the server being debugged. Thus, when the OSGi platform is starting, it reads the plug-ins from the Domino server and also from the developer's workspace, where the latter takes precedence. The developer can then remotely debug the plug-ins as they are executing on the Domino server. This eliminates the need for the developer to deploy the plug-ins to the Domino server before debugging them, which is a tedious process, at best.

We have all encountered situations in which it is difficult to tell from the symptoms of the problem the end user reports where exactly the code is failing. In such situations, it can be frustrating to try to set breakpoints at various points in the code and basically backtrack to a point in the code where the problem is thought to be originating. In such

cases, it is normally much more practical (and efficient) to enable logging in the system and judge from the output of the logs where the issue is occurring.

How to Configure notes.ini and rcpinstall.properties for Logging

The XPages runtime has a significant amount of logging built into it. All the loggers within the XPages runtime conform to the JSR47 specification for logging in Java. You will see through the following examples that the loggers are themselves highly configurable, with options to output the log content to various locations, along with the capability to filter the amount of content produced by the loggers on a logger-by-logger basis. Fortunately, enabling and disabling logging is easy within XPages and within the Domino server in general.

Logging for the XPages runtime is enabled in three ways:

- **notes.ini**, which enables logging for the JVM
- **rcpinstall.properties**, which enables logging for different parts of the XPages runtime
- **bootstrap.properties**, which enables the developer to specify a custom file that is responsible for configuring the XPages runtime's loggers

Table 6.4 lists **notes.ini** variables that you can use to debug the Domino server's JVM.

Table 6.4 NOTES.INI Java Logging Variables

Variable Name	Description
JavaVerbose	Enables verbose logging within all JVM instances running on the Domino server
JavaVerboseGC	Enables verbose logging for all Garbage Collection events in the Domino server JVM instances

Listing 6.19 shows how these variables are set in **notes.ini**. Users should be aware that setting these variables will cause a lot of information to be printed to the console. The logs these variables generate are of most use when class loading issues are suspected as the cause or at least a contributor to a runtime issue.

Listing 6.19 NOTES.INI Variables to Enable JVM Logging

```
JavaVerbose=1
JavaVerboseGC=1
```

Listing 6.20 shows a snippet of the Domino server console output showing some of the messages logged with verbose JVM logging enabled.

Listing 6.20 Sample Domino Server Console Output with JavaVerbose and JavaVerboseGC Enabled

```
10/02/2011 01:58:04 PM  HTTP JVM: class load: java/io/
➥InvalidClassException
10/02/2011 01:58:04 PM  HTTP JVM: class load: com.ibm.xsp.application.
➥FileStateManager$1 from: file:/C:/Program
Files/IBM/Lotus/Domino/osgi/shared/eclipse/plugins/com.ibm.xsp.
➥core_8.5.3.20110629-1645/lwpd.xsp.core.jar
10/02/2011 01:58:04 PM  HTTP JVM: class load:
➥java/util/concurrent/ThreadPoolExecutor$Worker
10/02/2011 01:58:04 PM  HTTP JVM: class load: java/util/concurrent/
➥locks/LockSupport
10/02/2011 01:58:04 PM  HTTP JVM: class load: com.ibm.xsp.http.
➥IUploadRequestWrapper from: file:/C:/Program
Files/IBM/Lotus/Domino/osgi/shared/eclipse/plugins/com.ibm.xsp.
➥extsn_8.5.3.20110629-1645/lwpd.xsp.extsn.jar
10/02/2011 01:58:04 PM  HTTP JVM: class load: com.ibm.xsp.http.
➥FileUploadRequestWrapper from: file:/C:/Program
Files/IBM/Lotus/Domino/osgi/shared/eclipse/plugins/com.ibm.xsp.
➥extsn_8.5.3.20110629-1645/lwpd.xsp.extsn.jar
10/02/2011 01:58:22 PM  HTTP JVM: class load:
➥org.eclipse.osgi.internal.resolver.StateWriter from: file:/C:/Program
Files/IBM/Lotus/Domino/osgi/rcp/eclipse/plugins/org.eclipse.
➥osgi_3.4.3.R34x_v20081215-1030-RCP20110624-1648.jar
```

When analyzing issues in XPages for which the cause is not obvious, it is often necessary to enable logging to get a better picture of where exceptions are occurring. Logging is enabled in a couple ways. Both are equally effective, but depending on the frequency at which the logging needs to be enabled and disabled, one technique might be more convenient.

XPages logging is enabled by directly editing **rcpinstall.properties** located in the **domino/workspace/.config** subfolder under the Domino data directory. **rcpinstall.properties** governs many of the settings used by the OSGi platform, so it is advisable to make a backup copy of this file before you edit it. It is normal to enable logging via this method if the logging is to be enabled for a relatively long period of time.

Loggers in the XPages runtime are divided into logical groups. Instead of providing one logger that will output messages for every functional part of the XPages runtime, several loggers have been provided so that the developer or administrator can get log information for targeted parts of the runtime. You can see a full list of the logical groups by scrolling to the bottom of **rcpinstall.properties**. All these loggers are disabled by default; the collective output would probably drown out the details needed to identify a problem. Instead, the developer or system administrator must enable loggers as needed and also set the level of logging required from the enabled loggers.

All the loggers defined within the XPages runtime conform to the official JSR47 standards for logging. Each logger can be configured to output varying amounts of detail, depending on the level at which the logger is configured. Nine predefined logging levels exist; Table 6.5 describes them.

Table 6.5 Logging Levels Available to XPages Runtime Loggers

Level Name	Definition
ALL	This is the most verbose level available. Tells the logger framework that all logged events should be reported to the log file.
OFF	This level tells the logger framework that no logged events should be reported to the log file.
SEVERE	This is the least verbose of the logging levels that allows for log messages to be output. This level is reserved for all severe system events, typically events that prevent the system from continuing to function normally. This level normally has an exception associated with it, as in `Failed to open database test.nsf, could not find file on server serverA/ibm`.
WARNING	A little more verbose than SEVERE, this level is reserved for system events that are of interest to end users and system administrators. The events reported at this level report potential problems, but problems that do not prevent the system from functioning. An example is `Failed to connect to server serverA/ibm, failed over to serverB/IBM`.
INFO	Reserved for events that may be of interest to end users but that do not inhibit system function. Typically, messages at this level report successful events/operations, as in `created database test.nsf at 11:45AM`.
CONFIG	This level is normally used to provide static configuration information generated by the system. Typically used to provide OS, system version, memory info, and so on.
FINE	The three levels FINE, FINER, and FINEST are meant to be relative levels. FINEST is the most verbose level, and FINE the least verbose. The FINE level is normally used to log information that is interesting to developers debugging the system.
FINER	This level is normally reserved for granular tracing events. Events logged at this level should provide a clear indication of the code path within the system—for example, `entering method xyz`.
FINEST	This tells the logger to output even the most detailed messages and is rarely needed. It is usually thought of as developer or debug tracing. This level is sometimes used during development to track the behavior of the system at a fine level or when trying to diagnose difficult issues.

To enable a logger, it is necessary to add a definition for the logger to **rcpinstall.properties**. The definition is *always* of this form:

```
loggerName.level=level
```

The JSR47 provides a great deal of flexibility when defining loggers and how their messages are to be output and stored. The JSR47 specification allows for `handlers` and `formatters` to also be defined.

Handlers are responsible for exporting the contents of the log message. The contents can be exported to wherever the handler defines. Typically, a handler exports the contents of the log message to a console (the Domino server console) or to a file.

As the name suggests, formatters are responsible for formatting the contents of a log message. Typically, a formatter is applied to a handler and is responsible for formatting the contents of the information exported by the handler. Listing 6.21 shows the contents of `rcpinstall.properties` that deals with defining loggers, handlers, and formatters.

Listing 6.21 Sample rcpinstall.properties

```
# JSR47 Logging Configuration
handlers=com.ibm.domino.osgi.core.adaptor.DominoConsoleHandler
com.ibm.rcp.core.internal.logger.boot.RCPLogHandler
com.ibm.rcp.core.internal.logger.boot.RCPTraceHandler.level=WARNING
com.ibm.rcp.core.internal.logger.boot.RCPLogHandler.level=WARNING
com.ibm.rcp.core.internal.logger.boot.RCPTraceHandler.level=FINEST

com.ibm.domino.xsp.bridge.http.manager.level = ALL
com.ibm.domino.xsp.bridge.http.config.level = ALL
```

We explain what each line of the previous listing means and how they affect logging on the server.

```
handlers=com.ibm.domino.osgi.core.adaptor.DominoConsoleHandler
com.ibm.rcp.core.internal.logger.boot.RCPLogHandler
com.ibm.rcp.core.internal.logger.boot.RCPTraceHandler
```

This sets the handlers to be used by the logging framework to save the contents of the log. As you can see, it is possible to save the contents to several different locations. In this case, the contents of the log are output to the Domino server console, to a log file, and to a trace file. `RCPLogHandler` and `RCPTraceHandler` are both handlers that the Domino server provides; they save the log files in an XML format (identical to the format used by the Notes client). For convenience, the Notes client and Domino server provide an XSL viewer to enable users to view trace and log files formatted in a

user-friendly format. The viewer is invoked when trace or log files are viewed within a browser. By default, the XSL viewer resides within **<Notes data directory>/workspace/logs** within the Notes client and within the **<Domino data directory>/domino/workspace/logs** directory on the Domino server.

```
.level=WARNING
```

This sets the level of all loggers that do not specifically have their own level set. In this case, all loggers default to a level of WARNING unless they are specifically set to another logging level via either **rcpinstall.properties** or **bootstrap.properties**.

```
com.ibm.rcp.core.internal.logger.boot.RCPLogHandler.level=WARNING
com.ibm.rcp.core.internal.logger.boot.RCPTraceHandler.level=FINEST
```

Each handler can be configured to export only log messages of a certain level and above. In this case, the `RCPLogHandler` is configured to save only messages that are declared to be WARNING messages and above. The `RCPTraceHandler` is configured to save all log messages.

```
com.ibm.domino.xsp.bridge.http.manager.level = ALL
com.ibm.domino.xsp.bridge.http.config.level = ALL
```

Finally, we define which loggers are actually to be enabled. In this case, we have defined that both loggers are to output all messages.

TIP You will find the XPages log files within the following client or server installation directory:

```
<Notes/Domino root folder>\data\IBM_TECHNICAL_SUPPORT
```

Each time the Notes client or Domino server is booted, a new XPages log file is created for that session when an XPages runtime exception occurs. If no runtime exception occurs during that session, there will be no associated XPages log file. The naming format for the XPages log files is as follows:

```
xpages_exc_server_yyyy_mm_dd@hh_mm_ss.log
```

For example:

```
xpages_exc_taurus_2011_09_21@08_15_23.log
```

You also find a `console.log` within this directory. Typically, if an exception occurs that the XPages runtime cannot handle, the Domino HTTP stack handles the exception instead. Such exceptions are severe and get logged in the `console.log` file.

Basically, any stack trace information that you see in the XPages runtime error page or take from the `requestScope.error` object is also written into the XPages log and console log files in greater detail. Therefore, it is always a good idea to analyze the XPages log and console log files if a problem is not obvious. This log information is also written to a separate log, in a tabular format within an XML log file. The XML log file can be found at **workspace\logs** under the Notes data folder, and at the **domino\workspace\logs** directory under the Domino server data folder.

As you can see from reading **rcpinstall.properties**, the file contains many settings. Editing this file often is not desirable because it increases the chance of unintentionally modifying a setting. As an alternative to editing **rcpinstall.properties** directly,

developers (or administrators) can also add a **bootstrap.properties** and **log.properties** file to the XSP directory residing in the Domino program files directory (for example **C:\Domino\xsp**). The developer (or administrator) need only modify the contents of **log.properties** to configure the log settings on the Domino server, without needing to worry about unintentionally changing some other setting stored within **rcpinstall.properties**. On non-Windows platforms, users must have root access to create files located under the Domino **bin** directory. Users must also set the correct permissions on **bootstrap.properties** and **log.properties** so the Domino server can read them.

Listing 6.22 shows the contents of a sample `bootstrap.properties`. Listing 6.23 shows the sample contents of **log.properties**.

Listing 6.22 Sample Contents of bootstrap.properties

```
#log groups enablement
log_configuration=xsp/log.properties
# Log system out
logdir=c:/Domino/log
```

Listing 6.23 shows sample content for **log.properties** that can be used to enable various levels of logging.

Listing 6.23 Sample Contents of log.properties

```
# Specify the handlers to create in the root logger
# The following creates two handlers
handlers = java.util.logging.ConsoleHandler java.util.logging.
➥FileHandler
# Save the log info to the following file
java.util.logging.FileHandler.pattern = c:\\domino\\log\\jdklog.txt
#Set the level of the root logger
.level=ALL
# Set the default logging level for new ConsoleHandler instances,
# only show ALL in the console
java.util.logging.ConsoleHandler.level = ALL
# Set the default logging level for new FileHandler instances
java.util.logging.FileHandler.level = ALL
# Set the default formatter for new ConsoleHandler instances
java.util.logging.ConsoleHandler.formatter = java.util.logging.
➥SimpleFormatter
java.util.logging.FileHandler.formatter = java.util.logging.
➥SimpleFormatter
# Set the default logging level for the logger named com.mycompany
com.ibm.domino.xsp.bridge.http.manager.level = ALL
com.ibm.domino.xsp.bridge.http.config.level = ALL
com.ibm.domino.xsp.bridge.http.native.level = ALL
com.ibm.domino.xsp.bridge.http.servlet.level = ALL
```

We have already explained most of the settings defined in Listing 6.23. However, Listing 6.23 introduces two new types of handler: `java.util.logging.ConsoleHandler` and `java.util.logging.FileHandler`. Both of these are standard Java handlers. Their names are self-descriptive. `java.util.logging.ConsoleHandler` prints all log messages to the `System.err output stream`, which, in the case of the XPages runtime, is conveniently the Domino server console. `java.util.logging.FileHandler` enables the developer to output the log messages to a designated `.log` file that the developer can access.

As a test, you should try some of these settings. You will be surprised by the amount of detailed information that you can gather from the resulting log files.

Interpreting a Stack Trace: Where to Go from Here?

If you read the "Error Management Properties" section in Chapter 1, "Working with XSP Properties," you are already familiar with XPages error handling options such as **Display XPage runtime error page > Standard server error page**. Indeed, you may have experimented with providing your own custom error page. For the really adventurous, there is also the possibility of embedding captured error information within a custom error page using Server Side JavaScript. This error information, otherwise known as the stack trace information, is made available via the `requestScope.error` object.

Regardless of whether the stack trace information is displayed as an error page in the client browser or within Notes/Domino log files, you are faced with the task of having to reverse-engineer this information to draw a conclusion and, ultimately, find the cause of an exception. This can be an even bigger challenge if, for example, a customer sends you a log file—that is, you have pretty much nothing else to work with to decipher the cause of some critical error within the application. As the developer, tester, or administrator, you must be able to interpret and translate these typically terse stack traces. The following sections detail various concepts and techniques that make reading and understanding these mystical stack traces a lot easier. Developing this skill will undoubtedly set you ahead of the pack—and also make your own job a much more enjoyable once in such situations.

Understanding the XPages Request Handling Mechanism

As a starting point for interpreting a stack trace, it is important to understand the fundamentals of how the XPages runtime handles a request. This enables you to understand where the entry point into the XPages runtime is and where a request ultimately ends up within the runtime before failing or causing an exception. You can then use this knowledge to either set breakpoints within your own custom Java source code (if you are using this) or within the runtime `.class` files themselves (you won't have the source, but you can still set breakpoints on the binaries). You can then view and modify the running variables stack in Domino Designer's Debug perspective. You learned about setting up a remote debugging configuration in Domino Designer earlier in this chapter.

First, when a request enters the XPages runtime on a Domino server, it does so through the XSP Command Manager's `service()` method. This is a dedicated request handler that is embedded within the HTTP stack process, **ntthp.exe**. In a stack trace, you can identify this by the following class/method name:

```
com.ibm.domino.xsp.bridge.http.engine.XspCmdManager.service()
```

This is slightly different in XPiNC, in which a different web container handles XPages requests. The entry point in this case is through the following class/method:

```
com.ibm.domino.xsp.module.nsf.NSFService.doService()
```

Following these two platform-specific entry points, the XPages runtime takes over. The incoming request is processed through a sequence of request-handling classes and methods before reaching the next interesting class/method:

```
com.ibm.xsp.webapp.DesignerFacesServlet.service()
```

This is the common denominator, regardless of platform used, whether Domino server or XPiNC. From this point, the incoming request follows the exact same execution path into the XPage component tree, the underlying server-side object representation of the XPage itself. Any exceptions that occur within the component tree are specific to that XPage, so you must analyze the stack trace accordingly for details of any failing component tree object method.

The important point to note here is that the previously mentioned entry points should be your first port of call when it comes to debugging and tracing an incoming request against the XPages runtime.

Understanding the XPages Request Processing Lifecycle

As a starting point, it is important to understand the fundamentals of the XPages request processing lifecycle. This lifecycle bounds a request submitted to the XPages runtime, which can potentially pass through six distinct processing phases.

The JSF request processing lifecycle consists of a number of well-defined phases that describe how each request is handled. Of course, these phases also apply to XPages. The phases on the standard request processing lifecycle are as follows:

1. Restore View
2. Apply Request Values
3. Process Validations
4. Update Model Values
5. Invoke Application
6. Render Response

Figure 6.13 illustrates how the processing lifecycle operates.

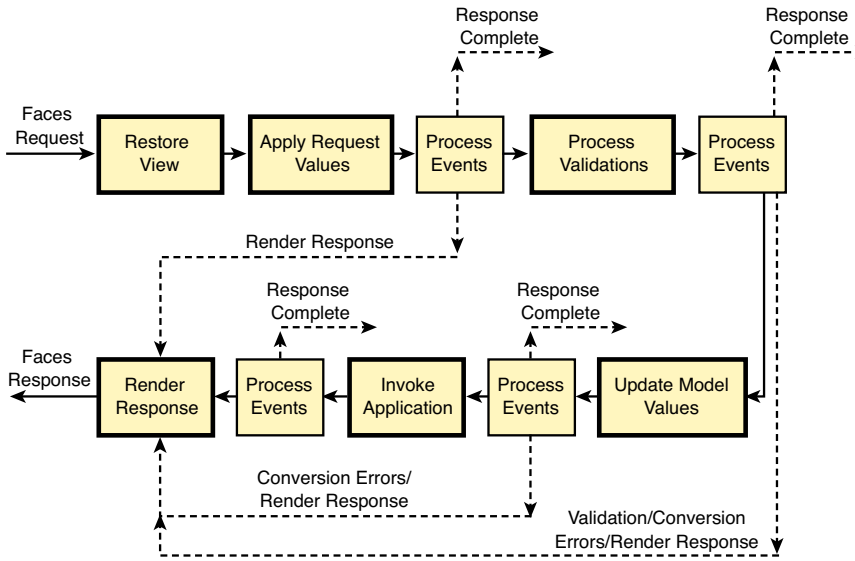


Figure 6.13 JSF request processing lifecycle

The Restore View phase retrieves the JSF view for the request. If no JSF view exists, a new one is created and cached for later use. Maintaining a consistent representation of the JSF view between requests simplifies the programming task for the application developer. It simplifies the application logic to focus on the business problem without having to maintain information about the state of the view.

The Apply Request Values phase enables the JSF components to update their state based on the values from the current request—if the component represents an editable value, the component stores the current value. Action and editable components have a special behavior during this phase. If the component `immediate` property is set to `true`, the JSF lifecyle is short-circuited. For an action component, the action processing happens at the end of this phase instead of later during the lifecycle. For an editable component, the validation processing happens immediately.

The Process Validations phase enables any validators associated with components in the view and any built-in validation associated with a specific component to be executed. All components that can be used to edit a value and that support validation will have a property named `valid` (to indicate whether the current value is valid). When validation errors occur, messages are queued and the `valid` property of the associated component is set to `false`. Validation error messages can be displayed to the end user using the `xp:message` or `xp:messages` tags. Validation errors typically terminate the lifecycle processing and result in a response sent immediately to the end user.

If the Update Model Values phase is reached, the values provided in the request are assumed to be valid (as defined by any validators specified in the view). The current values are stored in the `localValue` property of the associated component. During this phase, the application data is updated with the new values. In the case of an XPages application, the values are written to the Domino document during this phase.

If the Invoke Application phase is reached, the application data is assumed to have been updated. The relevant application logic specified in the view is executed during this phase. In an XPages application, if application logic is associated with a button and that button caused the page to be submitted, the logic is executed now.

The Render Response phase generates the response and saves the state of the view. In the XPages case, the response is an HTML page and the rendering is performed using a platform-specific render kit. The application developer has control over the state saving—that is, the developer can decide not to save any state, to optimize server performance.

In terms of being able to interpret a stack trace, it is important to understand that the processing of a request can fail within any one of the six request processing lifecycle phases, therefore causing the lifecycle to failover. In this condition, the XPages runtime either skips all lifecycle phases apart from the final sixth phase and renders a page with validation errors, or alternatively displays the chosen error page or default server error page. In the former case, the user is presented with an XPage rendering that typically includes validation error information, enabling the user to correct the problem. This is a highly controlled and robust feature of the XPages runtime that actually is triggered during the third request processing lifecycle phase, namely, the Process Validations phase. In the latter case, the request processing lifecycle has failed with a critical exception that cannot be handled within the context of the XPages runtime in a stable and recoverable manner. Hence, the end user is presented with the configured error page option, and the exception stack trace information is written into the XPages runtime log files for analysis purposes.

This is where having a grasp on the request processing lifecycle becomes important. When reading through stack traces in the XPages log files or in a browser, your first port of call is in identifying which phase of the six phase XPages request processing lifecycle the runtime exception occurred. This information might not always be present within the stack trace—for example, a critical exception might occur that does not allow the request to enter the request processing lifecycle. Knowing which phase has failed, along with understanding the intricacies of the lifecycle itself, will help you quickly resolve runtime issues that would otherwise be extremely difficult to identify even with powerful debugging tools.

For example, Listing 6.24 shows a stack trace fragment in which the triggering of a server-side `onclick` event caused the Invoke Application phase to fail. In this case, it is obvious from reading the stack trace summary information that a problem occurred when the Server Side JavaScript code was executed. The summary information details

the name of the application, the source XPage, and the actual control and type of event where the exception occurred.

Listing 6.24 Stack Trace Detailing Failure Within the Invoke Application Phase

```
Exception Thrown
Context Path: /PCG/PCGCH06.nsf
Page Name: /stackTrace.xsp
Control id: button1
Property: onclick
Script interpreter error, line=3, col=9: [TypeError] Method
➥XSPContext.redirectToPage(XSPUrl) not found, or illegal parameters
      1: var url:XSPUrl = new XSPUrl("viewResults");
->    2: context.redirectToPage(url);
javax.faces.FacesException: Error while executing JavaScript action
➥expression
       at com.sun.faces.lifecycle.InvokeApplicationPhase.
➥execute(InvokeApplicationPhase.java:102)
       ...
com.ibm.domino.xsp.bridge.http.engine.XspCmdManager.
➥service(XspCmdManager.java:272)
Caused by: com.ibm.xsp.exception.EvaluationExceptionEx: Error while
executing JavaScript action expression
      ... 19 more
Caused by: com.ibm.jscript.InterpretException: Script
interpreter error, line=2, col=9:[TypeError] Method XSPContext.
redirectToPage(XSPUrl) not found, or illegal parameters
      ... 26 more
```

Unfortunately, not all runtime exceptions occur during the Invoke Application phase. Listing 6.25 is a fairly typical example, demonstrating failure of an `eventHandler` SSJS code during this phase. The stack trace is self-explanatory.

But consider the case in which an exception occurs before the Invoke Application phase is reached. Assume, for instance, that you have `print()` or `__dump()` calls within an `eventHandler` and, thus, you are expecting to see debug information output. In this case, the `eventHandler` code is not processed and you will not see any console output from the `print()` and `_dump()` calls. You must therefore be able to establish the reason for the nonexecution of the `eventHandler` code. This is where the stack trace is useful. Listing 6.26 shows a stack trace fragment for this case, in which the expected triggering of an event handler as detailed in Listing 6.25 never occurred because the Update Model Values phase failed. This, in turn, caused the request processing lifecycle to failover by not continuing with subsequent phases and raising the exception. By failing in this manner, it prevented the event handler from being executed in the Invoke Application phase.

Listing 6.25 XSP Markup and Stack Trace Detailing Failure Within the Update Model Values Phase

```
<xp:button id="button4" value="Save">
    <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete" immediate="false" save="true">
        <xp:this.action>
            <![CDATA[#{javascript:
                if(adminBean.isDebugMode()){
                    print(">> memberBean debug info:")
                    _dump(memberBean);
                    print(">> end");
                }
                context.redirectToPage("members");
            }]]>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
```

Listing 6.26 shows a sample stack trace that results while executing the code listed in Listing 6.25.

Listing 6.26 Sample Stack Trace in Which the Update Model Values Phase Has Failed

```
Exception Thrown
The last packet sent successfully to the server was 0 milliseconds
ago. The driver has not received any packets from the server.
        at sun.reflect.NativeConstructorAccessorImpl.
➥newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(Native
➥ConstructorAccessorImpl.java:56)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance
➥(DelegatingConstructorAccessorImpl.java:39)
        at java.lang.reflect.Constructor.newInstance(Constructor.
➥java:527)
        ...
com.ibm.xsp.beans.MemberBean.setDisplayName(MemberBean.java:51)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        ...
com.ibm.xsp.component.UIInputEx.updateModel(UIInputEx.java:573)
        at javax.faces.component.UIInput.processUpdates(UIInput.
➥java:484)
        ... com.ibm.xsp.component.UIViewRootEx.
➥processUpdates(UIViewRootEx.java:1492)
        at com.sun.faces.lifecycle.UpdateModelValuesPhase.
➥execute(UpdateModelValuesPhase.java:98)
```

```
        at com.sun.faces.lifecycle.LifecycleImpl.phase(LifecycleImpl.
➥java:210)
        at com.sun.faces.lifecycle.LifecycleImpl.execute(LifecycleImpl.
➥java:96)
        at com.ibm.xsp.controller.FacesControllerImpl.
➥execute(FacesControllerImpl.java:250)
        at com.ibm.xsp.webapp.FacesServlet.serviceView(FacesServlet.
➥java:223)
        at com.ibm.xsp.webapp.FacesServletEx.
➥serviceView(FacesServletEx.java:200)
        at com.ibm.xsp.webapp.FacesServlet.service(FacesServlet.
➥java:160)
        at com.ibm.xsp.webapp.FacesServletEx.service(FacesServletEx.
➥java:137)
        at com.ibm.xsp.webapp.DesignerFacesServlet.
➥service(DesignerFacesServlet.java:103)
        ...
com.ibm.domino.xsp.bridge.http.engine.XspCmdManager.
➥service(XspCmdManager.java:272)
```

The important point to take from this example is that if you did not read the log file or browser stack trace, it would be extremely difficult to establish the fact that the `eventHandler` code never triggered. Understanding that the Update Model phase runs before the Invoke Application phase where `eventHandler` code is executed enables you to establish the fact that the Invoke Application phase is never actually reached in this example.

When reading a stack trace, look for details on the lifecycle phase, if available, as a starting point. Table 6.6 gives details on the XPages request processing lifecycle phases. You can use this to identify the phases and order of execution when reading a stack trace. Remember, Server Side JavaScript code is typically executed within `eventHandler` objects that are executed during the Invoke Application phase. Hence, any failure in the lifecycle phases before this phase can potentially prevent your Server Side JavaScript code from running.

Table 6.6 Lifecycle Phases and Running Order

Phase Name	Class Name	Order
Restore View	com.sun.faces.lifecycle.RestoreViewPhase	1st
Apply Request Values	com.sun.faces.lifecycle. ApplyRequestValuesPhase	2nd
Process Validations	com.sun.faces.lifecycle. ProcessValidationsPhase	3rd
Update Model Values	com.sun.faces.lifecycle. UpdateModelValuesPhase	4th
Invoke Application	com.sun.faces.lifecycle. InvokeApplicationPhase	5th
Render Response	com.sun.faces.lifecycle.RenderResponsePhase	6th

Therefore, having a good understanding of the XPages request processing lifecycle can prove beneficial when it comes to deciphering stack trace information where exceptions have occurred during lifecycle execution.

XPages Toolbox

As a final section in this chapter on debugging, it is fitting to mention the XPages Toolbox. The XPages Toolbox is an OpenNTF project that provides a powerful set of analysis features that enable the developer to closely analyze the performance of an XPages application. The XPages Toolbox is a set of web-based tools that provide the following features to the XPages developer:

- CPU profiler
- Memory profiler
- Back-End Class profiler
- Runtime monitoring
- Java logging group management

The XPages Toolbox is freely available for download from OpenNTF: www.openntf.org/projects/pmt.nsf/ProjectLookup/XPages%20Toolbox.

The CPU profiler is a high-level profiler that records how much CPU time is spent in various parts of the XPages code. Using the XPages Toolbox, the XPages developer can enable and disable profiling on demand, allowing certain events or actions to be profiled as needed.

The Memory profiler shows how much memory the XPages runtime is using and shows the applications using that runtime. Just as you would expect from a memory-profiling tool, it is possible to create memory *snapshots* with the XPages toolbox. These snapshots enable the XPages developer to quickly figure out which parts of the XPages application are holding on to memory.

The Back-End Class profiler gathers information on how the Domino back-end classes are being called and how much time is spent in each call. This enables XPages developers to determine which back-end calls are the most expensive and to limit the number of calls to such methods or classes as much as possible.

The Runtime monitoring functionality of the XPages Toolbox enables the XPages developer to gather a high-level view of what memory the HTTP JVM is using, as well as the number of active applications on the server.

As discussed in the previous section, various loggers are available within the XPages runtime; each of these loggers is configurable. The XPages Toolbox allows the developer (or administrator) to enable or disable loggers on the fly (without needing to restart the HTTP task). The XPages Toolbox also enables the developer to set the level of each of the loggers within the XPages runtime.

Extensive documentation is available on OpenNTF on how to install and use the XPages toolbox. Read that documentation to get further information on how to use the XPages Toolbox and also on what patterns you can apply to your XPages applications to make full use of the XPages Toolbox when debugging and profiling your applications.

Conclusion

This chapter gave you as much information as possible on built-in debug features and techniques for server-side debugging. Yes, it's true that Domino Designer does not have a Server Side JavaScript debugger—yet. But you can do many things to help yourself while debugging server-side code. Beyond this, you do have a powerful Java debugger within Domino Designer. If you are doing Server Side JavaScript and Java coding together, you have a useful and efficient debugging platform. You also learned how to set up a Domino server for logging and how to make sense of the logging information within. As XPages applications become more sophisticated, you will more likely need these skills sooner than later.

APPENDIX A
Definitive Resources

As you no doubt are aware, XPages is based on JSF, which is a standardized Java framework through the Java Standard Request (JSR) process. As is the case with many web technologies, XPages is based on and uses many different web standards. This appendix serves as a reference to those standards. Some technologies (such as HTML 5) are not fully supported by XPages (as of Release 8.5.3), whereas others, such as JSF, are fully supported.

Table A.1 Definitive XPages Resources

Name	URL
JSF Specification	www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
Java 1.5 Specification	http://java.sun.com/j2se/1.5.0/docs/api
J2EE 1.5 Specification	http://download.oracle.com/javaee/5/api
W3C DOM Specification	www.w3.org/DOM
HTML 5 (not final at time of this writing)	http://dev.w3.org/html5/spec/Overview.html
CSS Specification	www.w3.org/Style/CSS
Dojo Toolkit	www.dojotoolkit.org
Eclipse Equinox	www.eclipse.org/equinox
XPages Domino Object Map	www-10.lotus.com/ldd/ddwiki.nsf/dx/XPages_Domino_Object_Map_8.5.2
Domino Java APIs	www.tinyurl.com/DominoJavaAPIs
Domino Designer and XPages Reference Guides	www.tinyurl.com/DominoDesignerAndXPagesRef
Domino Designer & XPages 8.5.3 APIs	www-10.lotus.com/ldd/ddwiki.nsf/dx/Domino_Designer_Extensibility_APIs_Javadoc_8.5.3
XPages 8.5.2 APIs	www-10.lotus.com/ldd/ddwiki.nsf/dx/XPages_Extensibility_API_Documentation
Understanding XPages	www.tinyurl.com/UnderstandingXPages
XPages Extension Library	http://extlib.openntf.org
XULRunner	https://developer.mozilla.org/en/XULRunner

This page intentionally left blank

APPENDIX B
Useful Online Resources

You'll find lots of useful XPages resources out there on the web. Table B.1 provides a snapshot of some of the authors' favorites—sorry if your site is missing!

Table B.1 Useful XPages Resources

Name	URL
Debugging with Dojo	www.dojotoolkit.org/reference-guide/quickstart/debugging.html
Domino Debug Plugin	www.tinyurl.com/DominoDebugPlugin
DominoGuru.com	www.dominoguru.com
Eclipse Memory Analyser	www.eclipse.org/mat
Explore Eclipse's OSGi Console	www.ibm.com/developerworks/library/os-ecl-osgiconsole
IBM Active Content Filtering Technology	www.tinyurl.com/IBMProjectZero
IBM Dump Analyser	www.ibm.com/developerworks/java/library/j-ibmtools1
IBM Memory Analyser	www.ibm.com/developerworks/java/jdk/tools/dtfj.html
Installing IBM Dump Analyser	www.tinyurl.com/IBMJavaDumpAnalyzer
IQJam	www.iqjam.net/iqjam/iqjam.nsf/home.xsp?iqspace=Domino+Development%7EXPages
JavaBeans 101	java.sun.com/developer/onlineTraining/Beans/bean01
John Mackey's Blog	www.jmackey.net
JVM Launch Options	www.tinyurl.com/JVMLaunchOptions
Mastering XPages Development	www.ibmpressbooks.com/bookstore/product.asp?isbn=9780132486316
Matt White's Blog	www.mattwhite.me
Notes/Domino 8.5 Forum	www-10.lotus.com/ldd/nd85forum.nsf/Dateallthreadedweb?OpenView
Notes/Domino Application Development Wiki	www.lotus.com/ldd/ddwiki.nsf/xpViewTags.xsp?categoryFilter=xpages
NotesIn9 Screencast	www.notesin9.com
OpenNTF	www.openntf.org
OpenNTF Blog	www.openntf.org/blogs/openntf.nsf/FullArchive?openview
OSGi Console Commands	www.tinyurl.com/OSGiConsoleCommands

Name	URL
Planet Lotus	www.planetlotus.org/search.php?search=xpages&sort=1
Pushing Policy Settings to the Notes Client	www.tinyurl.com/PushNotesIniSettings
Reading the Contents of a JavaDump File	www.ibm.com/support/docview.wss?uid=swg21181068
Taking Notes Podcast	www.takingnotespodcast.com
The Learning Continuum Company	www.tlcc.com
The XCast	www.thexcast.net
XPages Blog	www.xpagesblog.com
XPages Info Site	www.xpages.info
XPages Toolbox	www.tinyurl.com/OpenNTF-XPagesToolbox
XPages Wiki	www-10.lotus.com/ldd/ddwiki.nsf
Pages.TV	www.xpages.tv
XPages101 Video Training	www.xpages101.net
YouAtNotes XPages Wiki	www.xpageswiki.com

APPENDIX C

Make Your Own Journal

Index

D

E

F

G

H

I

J-K

L

M

N

O

P

Q–R

S

T

U

V-W

X-Y-Z

This page intentionally left blank

THIS PRODUCT

informit.com/register

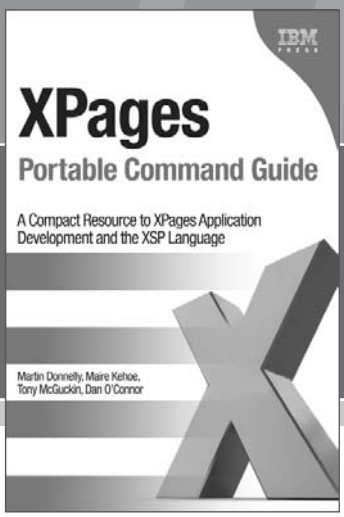
Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.



Safari Books Online

FREE Online Edition

Your purchase of ***XPages Portable Command Guide*** includes access to a free online edition for 45 days through the **Safari Books Online** subscription service. Nearly every IBM Press book is available online through **Safari Books Online**, along with thousands of books and videos from publishers such as Addison-Wesley Professional, Cisco Press, Exam Cram, O'Reilly Media, Prentice Hall, Que, Sams, and VMware Press.

Safari Books Online is a digital library providing searchable, on-demand access to thousands of technology, digital media, and professional development books and videos from leading publishers. With one monthly or yearly subscription price, you get unlimited access to learning tools and information on topics including mobile app and software development, tips and tricks on using your favorite gadgets, networking, project management, graphic design, and much more.

Activate your FREE Online Edition at informit.com/safarifree

STEP 1: Enter the coupon code: GIGREAA.

STEP 2: New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

Addison Wesley · Adobe Press · Alpha · Cisco Press · FT Press · IBM Press · Microsoft Press · New Riders · O'Reilly · Peachpit Press · Prentice Hall · Que · Redbooks · Sams · SAS Publishing · VMware Press · Wiley · Wrox