



Lotus software

Produced in
collaboration with:  **Redbooks®**

Content in this document was produced in collaboration with Lotus® and IBM® Redbooks®.

Redbooks Wiki: Building Domino Web Applications using Domino 8.5.1



Pascal David
Muhammad Ali Sabir
Jérôme Deniau
Bruce Lill
Abhishek Jain
John Bergland

Contents

Important Note about referencing the latest information.....	5
Meet the Authors.....	5
Special thanks to the following people for contributing to this effort	7
Sample Materials and Code Resources available for download.....	8
Developer tools and resources	8
Useful sites for Domino Web information.....	8
Useful Web development Web sites	8
JavaScript libraries and reference sites	9
Useful sites for Web Development tools	9
Section I. Introduction	12
Objective of this Redbooks Wiki	12
Section II. High level introduction to XPages	13
Introduction to this section	13
New Design elements for XPages	14
Reviewing the XPages Design Element	14
Reviewing Custom controls	17
Introduction to the Outline Eclipse View and the Component Eclipse View.....	18
Considerations for Composite Application Development and XPages	20
XPages definition	20
XPages based on JSF.....	21
Understanding the XPages definition within a Notes and Domino Context.....	21
Why should I care about XPages technology in Domino	24
What are the benefits over traditional Domino Development	25
Performance Improvements related to XPages.....	31
Closing arguments in favor of XPages.....	32
In favor of ONE.....	32
Section III. Value of Domino as a platform and Designer as the tool for building Web applications.....	34
Objective of this section	34
Client apps vs Web apps	34

Domino as a Web server	34
What kind of applications can be developed.....	36
What is new in Designer 8.5.1 and improvements over earlier versions	36
Section IV.	
Reviewing underlying technologies and skills required to build Domino XPages web applications	39
Introduction to Domino Object Model	39
Introduction to HTML CSS XML and JS and their role in XPages.....	40
Introduction to JSF	42
A peek inside Dojo	42
Do I need to learn JAVA	43
Section V. Domino design elements specific to building an XPages application	45
Introduction to this section	45
XPage View control	45
Searching Techniques	51
XPage Repeat control.....	53
Themes.....	58
Reuse of current Domino code	62
Active Content Filtering ACF	65
Adding Java to template and build path	66
Section VI - Sample 1: Introduction to enabling existing Notes Client Applications using XPages	69
Overview of the pages which make up this tutorial	69
The document library template Notes client application.....	70
Scope of the case study	72
Conceptual design.....	74
Getting started - XPage enabling an existing Notes client application	75
Create the website layout - XPage enabling an existing Notes client application.....	88
Form Design - XPage enabling an existing Notes client application	109
Workflow - XPage enabling an existing Notes client application	141
Create and display responses - XPage enabling an existing Notes client application	161
Sample 1 - Tutorial Summary.....	173
Introduction to Sample 2: Building a new XPage web application from scratch.....	175
Introduction	175

Step 1 – Sample 2: Selecting a CSS Framework	178
Step 2– Sample 2: Creating the database and importing images.....	179
Step 3 – Sample 2: Incorporating the CSS Framework using Themes	181
Step 4 – Sample 2: Understanding the web application layout.....	186
Step 5 – Sample 2: Developing the web application layout using Custom Controls	188
Step 6 – Sample 2: Developing custom controls for global content.....	216
Step 7 – Sample 2: Creating forms and views.....	249
Step 8 – Sample 2: Developing custom controls for specific content	258
Step 9 – Sample 2: Creating the XPage for documents section.....	358

Important Note about referencing the latest information

This PDF Document represents the overall **Redbooks Wiki for Building Web Applications using Domino 8.5.1**.

In addition to this information contained in this document, please also refer to the complete Lotus Redbooks Wiki at the following URL:

[http://www-10.lotus.com/ldd/ddwiki.nsf/dx/Master Table of Contents for Building Domino Web Applications using Domino 8.5.1](http://www-10.lotus.com/ldd/ddwiki.nsf/dx/Master_Table_of_Contents_for_Building_Domino_Web_Applications_using_Domino_8.5.1)

Meet the Authors



Pascal David is an IBM Certified Senior Consultant at GFI Belgium. He's been working for 15 years as a Lotus Notes and Domino development specialist. Being a web development consultant since the first release of Lotus Domino, Pascal has worked on countless web projects at customers, from small business website developments to corporate intranet platform implementations. He is also product manager of GFI's own Lotus Notes based Web Content Management solution, [WebDotNSF](#).



Muhammad Ali Sabir is a Sr. Software Engineer at AAC Inc -- an IT services company based in Vienna, VA. He has over 10 years of extensive hands-on experience with Notes/Domino and Java EE and is certified in both of them. In addition, he holds an MS in Information Systems from George Washington University and is certified as "Chief Information Officer" by CIO University. He has led the development to dozens of corporate Domino and Java based web applications. He is also an architect and lead developer for award-winning suite of applications for Unified Communications sold under brand name [PhoneTop](#). His applications have been highlighted in industry journals such as CRN, BusinessWeek and Computerworld.



Jérôme Deniau is an IBM Business Partner. He started Lotus-Notes in 1993 with Lotus-Notes R3 on Windows, HP-UX and SCO Xenix. He has started teaching Lotus-Notes since then and became a Certified Lotus-Notes Instructor for administration and development (from R3 to ND 8.x). Located in France, he is currently maintaining the DFUG (Domino French speaking Users Group) a Lotus-Notes/Domino clients only users group using Domino and Quickr that will be on-line just before LotusSphere 2010. He also developed a mail tracking application running on Windows 32/64 bits, Linux and AIX. He can be reached at jerome.deniau@inform-france.com, and met - as always - at the CLP lounge at LotusSphere.



Bruce Lill is a partner at [Kalechi Designs](#). He started working with Lotus Notes V2.1 for a large telecommunication company. He was given a project to allow users from across the company to collaborate on new products and found Notes was the perfect fit. He has been dedicated to Notes since then. He was certified as a developer, administrator and Instructor for R4, R5 and R6. He has implemented Notes infrastructures that span 3 continents and 6 countries, built apps to help state police track violent gangs and web sites to monitor for bio-terrorism attacks.. He was a member of the Redbook Domino 8 Best Practices Web Development wiki, the first wiki by Redbook. As a Lotus Community Advocate he helps drive new content and public awareness. He can be reached at bruce@kalechi.com, on twitter as Kalechi and Linked-in under Bruce Lill.



Abhishek Jain has been working with IBM India Software Labs since 2004 and has been in the industry for 8 years. He is currently working as an IT Specialist with Lotus Lab Services and has skills on various Lotus technologies. He has considerable experience on Lotus and Java/J2EE technologies and is certified on both of them. He has also co-authored Customizing Lotus Quickr 8.1 Redbooks Wiki and developerWorks article on Integrating IBM Lotus Forms with IBM Lotus Domino.



John Bergland is a project leader at the ITSO, Cambridge Center. He manages projects that produce IBM Redbooks®, focusing on IBM Lotus and WebSphere technology. Before joining the ITSO in 2003, John worked as an IT Specialist with IBM Software Services for Lotus® (ISSL), specializing in Notes and Domino® messaging and collaborative solutions. He holds an MBA and MS in Information Technology from Boston University.

Special thanks to the following people for contributing to this effort

We wish to acknowledge a special thank you to the following sponsors and key stakeholders from the Lotus Forms Development, Product Management and Lotus IDC Teams:

- **Amanada Bauman** - Everyone Writes and IDC Wikis Program Manager
- **Peter Janzen** - Senior Product Manager, Domino Designer
- **Steve Castledine** - Advisory Software Engineer - XPages, Lotus Notes Domino, Wiki, blogs
- **Paul Hannan** - Software Engineer - Lotus Domino XPages
- **Chris Toohey** - Blogger, podcaster, writer, and self proclaimed “geek”, Chris Toohey covers topics from application development to the latest must-have-gadgets. A special thanks to Chris for his efforts to publicize this effort and deliverable.

Sample Materials and Code Resources available for download

For the samples provided in this Redbooks Wiki effort , you can download the sample code from the following page:







<http://www->

[10.lotus.com/ldd/ddwiki.nsf/dx/Sample_Materials_and_Code_Resources_available_for_download](http://www-10.lotus.com/ldd/ddwiki.nsf/dx/Sample_Materials_and_Code_Resources_available_for_download)

Developer tools and resources







This is a list of developer tools and resources that have helped us to learn and build better XPage applications.

Useful sites for Domino Web information






Site	Description
Lotus Developer Domain 	The main site for Domino information
PlanetLotusg 	A feed from almost Lotus blogs that is updated hourly. Here you will find the best blogs on XPage development.
OpenNTF organization 	The home of open source Notes applications. There are a number of XPage applications that you can download and view to learn more about Xpages. This wiki template is a project there.
Lotus Greenhouse 	Lotus Greenhouse is a premier showcase Web site to experience Lotus products.
Notes.ini settings 	developerWorks Notes ini list - viewable by starting letter.
Notes.ini settings 	User maintained - viewable by name, categories, product, or version. The database is fully searchable and available via a Notes 8 Sidebar (on OpenNTF.org).

Useful Web development Web sites

Site	Description
------	-------------
















Blueprint CCS Framework 	An open source CSS framework that provides the look of a structured site without a lot of hassles. If you are migrating from a table or frame based site this will let you retain the overall look using web standards and cross-browser support..
Blueprint Generator 	Use this site to build the Blueprint CSS you need. Enter the parameters of you site and it will create the Blueprint CSS to fit it.
aListApart 	CSS and Web best practices web site
w3c organization 	The home of the Internet standards
w3cSchools 	The basic how-to for HTML and JavaScript
kuler 	kuler allows you to create, rate and download color themes. This site provides inspiration when trying to create a color scheme for a Web site or Notes application.







JavaScript libraries and reference sites

Site	Description
ECMAScript 	The home of the JavaScript standards home.
Dojo 	Home of the Dojo JavaScript Library
Ajaxian 	A site dedicated to improving Web development
JSON home 	The home page for JSON information
JSON 2 HTML 	View your JSON graphically so see what the structure really is.

Useful sites for Web Development tools

Site	Description
------	-------------

Eclipse 	Eclipse project has lots of add-on tools. It is the basis of the Notes 8 client, which makes it a good platform to start learning it now.
CSS Tab Designer 	A tool to build nice tags without being a graphic artist.
Notepad++ 	Editor for text, HTML, JavaScript, C, and so on with extensions.
Balsamiq Mockups 	An easy to use tool to create mockups of your web pages. It will help while you design your web pages and it's looks like hand drawn so you won't spend the meeting trying to get color and positioning exact instead of overall look.
aDesigner 	Test accessibility of your web pages.
Mozilla Firefox  tools	
RestClient 	RESTClient is an application use to visit and test RESTful services
Web Developer Toolbar 	This is one of the 2 must have extensions. It provides such useful Web development tools as live CSS edit, .form information and the ability to set you browser to match screen resolutions for testing.
Firebug 	The other must have extension with which you can edit, debug, and monitor CSS, HTML, and JavaScript live in any Web page.
View Source Chart 	Creates a graphic view of your web page's html structure.
HTML Validator 	Helps you validate the your web pages page. The extension is based on Tidy and OpenSP (SGML Parser).
FireShot 	A browser based screen capture tool. Great for getting screen captures of your web pages for hte documentation.
LiveHTTPHeaders 	View HTTP headers of a page while browsing.
Javascript Debugger 	The way to debug JavaScript.
JSONView 	With the JSONView extension, JSON documents are shown in the browser similar to how XML documents are shown.

Microsoft Internet Explorer  Tools	
Microsoft Script Debugger and Debug toolbar 	Contains all Microsoft add-ons for Internet Explorer.
Fiddler 	An HTTP debugging proxy that logs all HTTP traffic between your computer and the Internet.
DebugBar 	Debugger for IE lets you see the HTML, JavaScript and CSS.
Google Chrome  Tools	
Google Chrome Extensions 	Extension for the Google Chrome browser.

Section I. Introduction

Welcome to the Redbooks Wiki on using Best Practices for Domino 8.5 Web Application Development.

The focus of this Redbooks Wiki is to help you understand improvements in IBM Lotus Domino 8.5 as a web development platform. The primary focal point is the introduction of XPages in IBM Lotus Domino 8.5 and discussing how XPages dramatically shortens the learning curve for Notes and Domino developers. Not only do XPages now allow you to incorporate Web 2.0 features and functionality into your web applications, but XPages continue to leverage your existing Domino development skills, while intuitively incorporating new design element features.

Objective of this Redbooks Wiki

The objective of this Redbooks Wiki is twofold:

1. Provide guidance on Best Practices for Domino Web Development, with emphasis on the new XPages design element. We introduce the key new features of XPages and discuss how and why these are meaningful within the context of Domino Web Development.
2. Within this Redbooks Wiki, we provide two hands on tutorials and sample code which provides you with an excellent chance to begin working with XPages and understanding the improvements in Domino 8.5 Web application development. As a foundation application for each of the tutorials, we begin with a well known Lotus Notes and Domino application - the Document Library Template.
 - [Sample Tutorial 1](#) shows you how to extend the life of your existing Lotus Notes Client applications by enabling them into an XPages application. You can review the scope and objectives of this [scenario example here](#).
 - [Sample Tutorial 2](#) shows you how to build a new XPage web application from scratch, creating the functionality of the Document Library while incorporating best practices such as using the OneUI theme at the foundation of your application.

Section II. High level introduction to XPages

Introduction to this section

XPages are new design elements in Lotus Notes and Domino 8.5 and 8.5.1. To begin using and designing XPages, you need to have a Domino 8.5 - or even better - a Domino 8.5.1 server and a Lotus Notes Designer client.

So what are immediate benefits of using XPages which make them so important?

First, these new elements allow an Application Developer to quickly and easily create rich web applications in Domino with a Web 2.0 application look and feel.

- XPages can be used to incrementally enhance an existing application or write a brand new one
- XPages are fully integrated within Domino Designer
- XPages decrease the time for writing Web Applications
- Starting from 8.5.1 they also run in the client promoting *same source – multiple rendering application development*

Additionally, XPages provides the developer with many “out of the box” features that previously would require lots of custom development to achieve such as:

- Type-ahead
- Field Validation
- Partial Refresh
- Date Time Picker
- One UI theme - so great default UI without any work
- Others – File Upload, File Download controls, Dojo Rich Text Editor

Note - The 8.5.1 version can now execute XPages code in the Lotus Notes client. They are available in any database you use in Lotus Notes Designer, *without* requiring you to change the ODS of your database.

The necessary steps to start is to have a Domino 8.5 or 8.5.1 server running, and of course a Lotus Notes Designer 8.5 or 8.5.1 client. You should ask your Domino administrator to launch the HTTP task to be able to start a whole new great way to develop Lotus Notes application for the web. Since XPages are fully integrated within Domino Designer, you have nothing additional to setup, except to launch the Domino HTTP task on your Domino server. If you are an advanced administrator you would like to know how the Domino server can be tuned to your requirements in terms of security and performance, please consult the Server Configuration section of this Redbooks wiki.

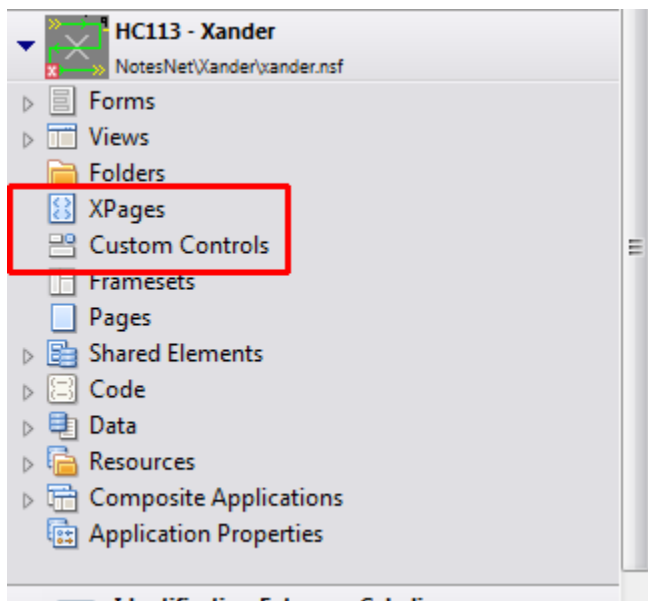
New Design elements for XPages

Once you have opened the Notes designer client, you can access two new design elements:

- one is simply **XPages**,
- the second is **Custom Controls**.

So what is the relationship between **XPages** and **Custom Controls**?

- XPage development is structured by using **XPages** and **Custom Controls**
- XPages are the main “scenes” for each part of your application
- Custom controls are the componentized “parts” of your applications functionality
- Custom controls are essentially “Sub Forms on Steroids”

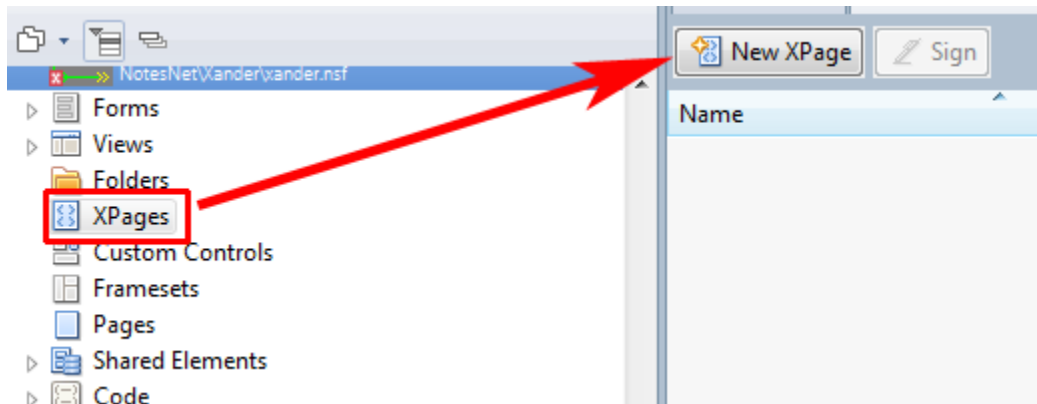


Reviewing the XPages Design Element

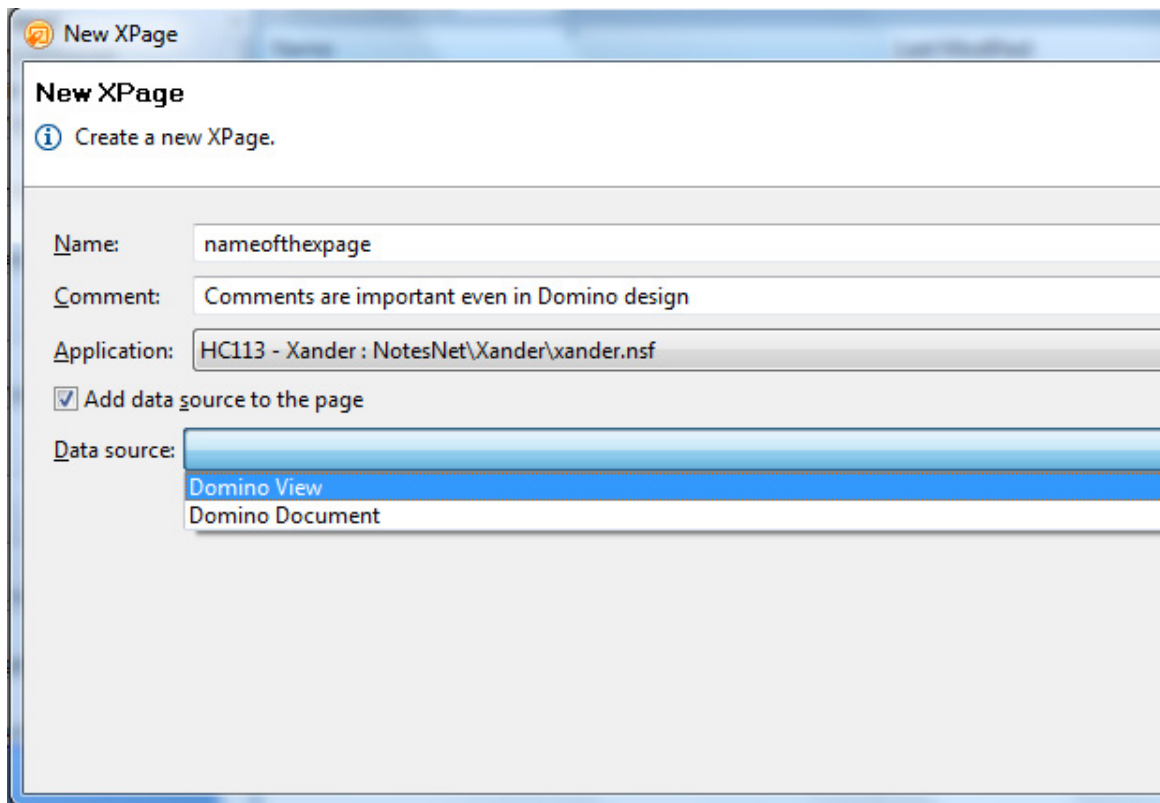
For developers already familiar with Notes and Domino development, the XPages design element is as simple as a new form or view design element. For developers new to Notes and Domino application development, XPages are now an integral design element, similar to the other Lotus Notes database design elements.

Creating and exploring a new XPage

To start a new XPage, you just have to select the XPage component, and you will then be able to design an XPage component.



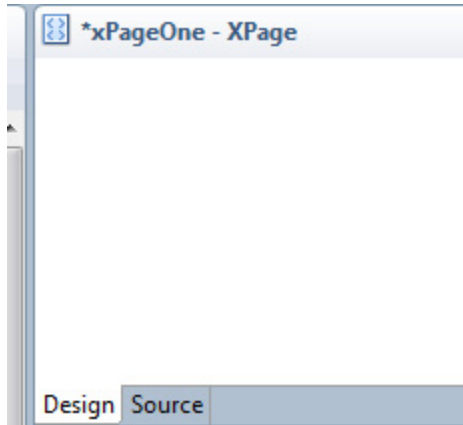
Clicking on XPages will allow you to start creating a new design element in your Notes database.



As you can see from the screenshot above, you need to name your new component. Enter in some comments about the component as a best practice to document your application, then select in which application you are about to create the new design element. By default, it is supposed to be created in your own application.

Note - An important detail is the checkbox **Add data source to the page**. This selection is new for common Lotus Notes designers. We advise you not to select this option in the very beginning of an XPage design process. Instead, when you become more accustomed to XPages development you will

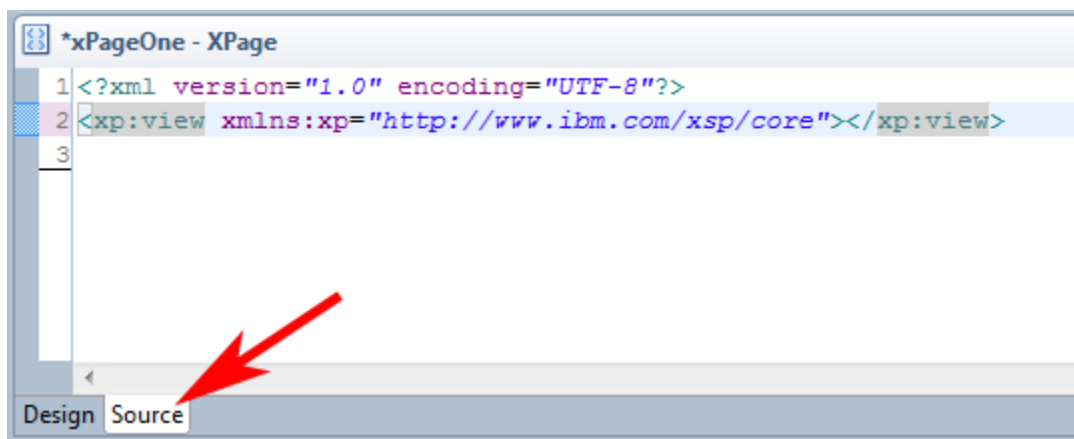
probably wish to check this option. What does that mean? Basically an XPage can live on its own (a main page for instance), but in most cases you will use XPages to allow people to have inputs, to show portions of Domino views, and much more. Accordingly, checking this option will allow you to define the source of the data you want to manipulate.



Once you create the new XPage component, you are presented with the same, familiar Lotus Notes blank design element as shown in the screenshot above. (The appearance of a new XPage component is similar to any form or subform). To begin, we have decided to just show the basic component. As you can see, you do have just two tabs, the **Source Tab** and the **Design tab**.

Clicking the **Design** tab allows you to simply drag and drop *components* or type in some text. *Components* we said? Yes the great improvement for XPages is that you can now start applications with internal and external components to help you speed up your web design development within Lotus Notes. For the seasoned Lotus Notes designer, we can really compare this option to the use of **subforms** and/or **shared fields**. As we will demonstrate throughout this wiki, the use of components in XPages enables much faster and more efficient development.

Clicking on the **Source** tab you will discover what is behind the simple blank page!



As you can see from the image above, yes! Domino goes XML at last! So the fabulous blank page we

have just defined is actually a powerful XML component. That means that now Domino will allow us to use powerful XML based design concepts. XML is generated automatically as you drag and drop controls or while you are working on the XPages.

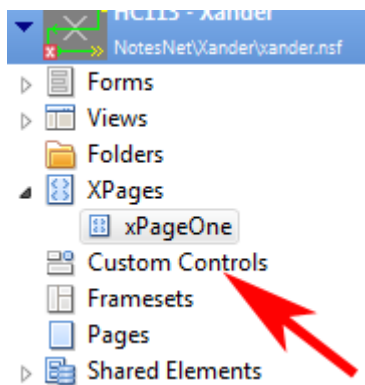
Note - Before proceeding into development design, you will need to familiarize yourself with the new Domino Designer interface. While there may be a new and updated look and feel, IBM development team has just made the product better and easier. In many ways, it's close to what a Notes designer had in earlier versions, but now includes many eclipse based features. For eclipse based coders, the Domino designer contains these features they are familiar with, with some additional features from the Notes heritage. As we proceed with this article, we will explain and highlight how to use many of these new features.

Reviewing Custom controls

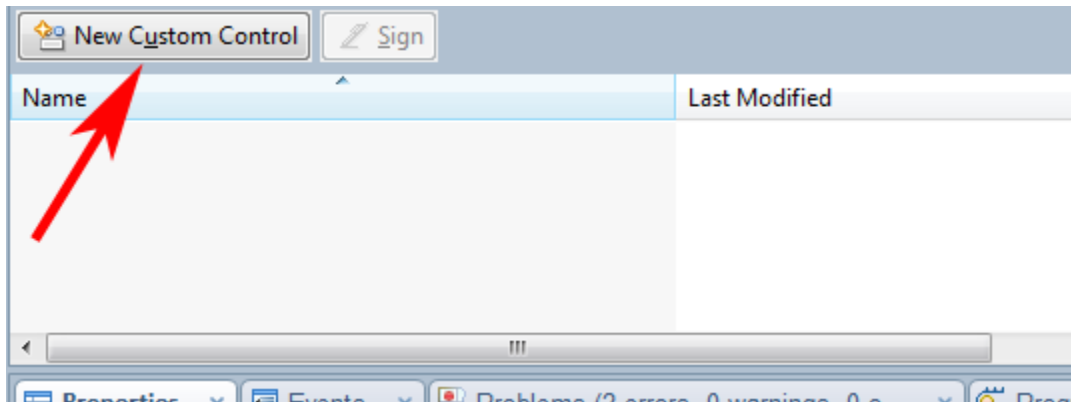
Another new design element is the **Custom Controls**. To better understand Custom Controls, keep in mind that XPage development is structured by using **XPages** and **Custom Controls**.

- **XPages** are the main “scenes” for each part of your application
- **Custom controls** are the componentized “parts” of your application's functionality
- **Custom controls** are essentially “Sub Forms on Steroids”

To create a new Custom Control, you will find them located just beneath the XPages option and above **Framesets**.



If you select this Lotus Notes design element you will be able to create a new custom control element or edit existing one, similar to working with any of the Lotus Notes design elements.



Note - For an experienced Lotus Notes Designer, **Custom Controls** bear similarity to the standard Lotus Notes elements such as **Subforms**, **Shared fields** and so on. You will be able to use controls developed by you, as well as those from other developers. These controls can then be integrated into your Lotus Notes XPages.

Custom controls are the componentized “parts” of your application's functionality - promoting re-use and more efficient development.

For instance, remember what you most likely did for your initial Lotus Notes development work? You most likely created two subforms: one called "Header" and another one called "Footer"? Initially, you may start with approach when working with XPages controls, but you will soon discover that this new element is far more powerful than the older Notes and Domino design elements.

Similar to subforms, any modification you can do on a custom control will be propagated to the current application.

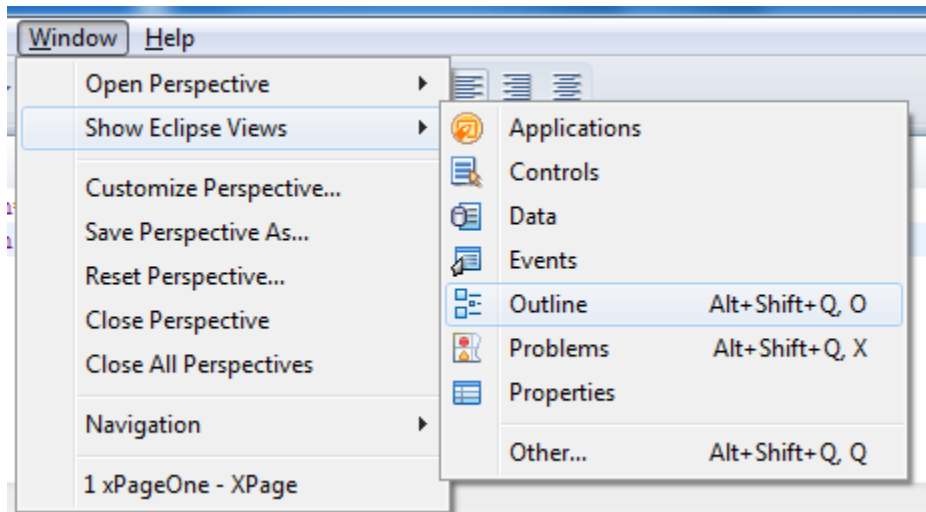
If designed and developed with reuse in mind, custom controls can be **written once and reused many times** in your application and in other applications saving many development hours.

Introduction to the Outline Eclipse View and the Component Eclipse View

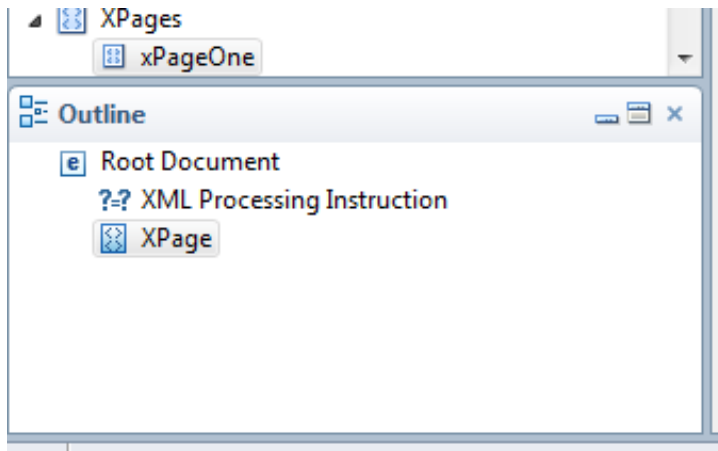
The following section introduces the **Outline View** and the **Component View**, two essential views which allow you to work efficiently with your components.

The Outline Eclipse view

The **Outline view** allows you easily select any component on your XPages. To access the outline view, first click on **Show Outline Views**, then click on **Outline**, as shown here.



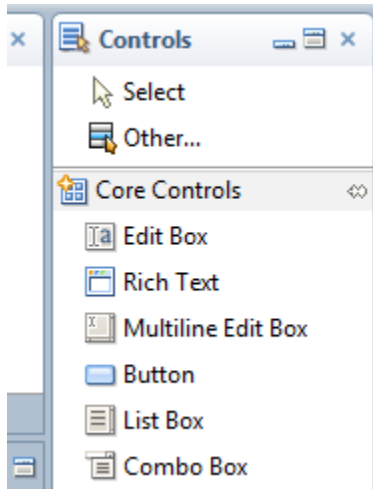
The screenshot below illustrates what you will see within the **outline view**.



It allows you to move quickly from one component to another.

The Component Eclipse view

The Component Eclipse view is an essential tool which you will also find very useful. It is a view which contains all standard components as your very own components. You will only have to drag and drop your components onto your XPages to just use them. (Keep this ease of use in mind as you think back to the earlier approach in Domino design of using subforms and shared fields concepts in standard Lotus Notes design).



Considerations for Composite Application Development and XPages

Composite application development was first introduced in Lotus Notes and Domino 8. The new release allows designers to add XPages to composite applications. So, basically you can develop custom controls and add them into your composite application editor as components.

Design precisions

As an experienced Lotus Notes Designers you probably have noticed that we did not talk about alias (or synonym) for the two new design elements, right? Well you will need to forget that option for those two new components, as it is no longer relevant.

XPages definition

Note - Goals and Objectives of this section - The following section discusses the underlying technology which make up XPages. While the following section starts with a direct dive into JSF technology, the goal is to then bring this back into a context which is more familiar for the traditional Domino Developer. Here are some key points to keep in mind as you read this next section:

- Under the covers, XPages is based on JSF (Java Server Faces) technology and also uses the Dojo Toolkit (for out of the box controls and for developer reuse)
- At the beginning, understanding the details of these technologies are not important as you can continue developing Domino applications without this knowledge. it is fun to learn though!
- You build your applications using the new Eclipse based Domino Designer
- Other associated new design elements are **Themes** and **Server Side Script Libraries** (JavaScript) or SSJS for short

XPages based on JSF

XPages technology is based on Java Server Faces (JSF) from Sun. The JSF Framework is dedicated to build Web 2.0 applications and is based on components that Java developers can compare to Swing or AWT (Abstract Windows Toolkit). The component state is evaluated during execution and for the request time of life.

JSF simply uses the MVC architecture model for the modeling part (Model, view and controller) and JSP by default. But concretely, you can use it with other technologies, for instance, the IBM Lotus Notes client 8.5.1 is using XUL to render XPages.

JSF is composed of a set of APIs to render and manage components, their states and events and interoperability between them.

The only limitation is that JSF standard components are too limited for Enterprise development for companies. So, JSF turns to be the start of a better interface for developers who want to publish a better Framework.

Accordingly - IBM has used the XPages to accommodate JSF limitations and integrate the possibilities of the Framework into Lotus Notes for Web developers and simplify the Lotus Notes development process and work on the Web side.

This technology was introduced to simplify web development Inside Lotus Notes for future web applications. Even if you can upgrade existing Domino Web application to XPages, starting from scratch is a good solution to learn and manage the XPages concepts. (We discuss both approaches more in the sections on Building Sample XPages applications - See the sections for [Introduction to enabling existing Notes Client Applications using XPages](#), and [Building an XPage web application from scratch](#))

So how can we define XPages technology in Domino? Domino replaces the servlet aspect used by JSF using a pure XML engine. So the http task includes the XSP engine. Temptation is easy to compare XPages to eXtensible Server Pages (XSP). Of course, they both focus on separation of content, logic and presentation, but XPages are a component of a Lotus Notes database, which is not limited to either server-side, nor to Java. Moreover, XSP is loosely defined and contains a limited set of instructions.

Understanding the XPages definition within a Notes and Domino Context

For java developers, Swing can be considered as participating in the [Model-View-Controller \(MVC\) model](#). But for Swing , it plays the Controller part which can use internal views to interact with the user interface. Swing components are lights if you compare them to AWT. AWT is using native components.

XPages introduces that MVC notion. But let's first go back to the traditional Lotus Notes designer to put this into a more specific context for Notes and Domino Developers.

With Lotus Notes (the client), developers do have the possibility to use views, forms, subforms (the *View* in MVC), documents and profiles (the *M* in *Model*), and Lotus Notes and Domino (the *Controller*). But in this architecture, the form also plays the role of the model, so the form and the documents are tightly bound together and that can result in some tricky situations when you do need to go on the web. As you know, code is mixed with components elements and porting the application to the web can make the maintenance harder. For the Domino developer (aka web designer), Domino gives them the possibility to divide some portion of code into the MVC model. Remember the « **\$\$ViewTemplateDefault** » and « **\$\$ViewTemplate For** » objects you can use and abuse? But, you will need to use the « **WebQueryOpen** » and « **WebQuerySave** » events to define the logic of the application. So, you will not be able to divide the *model* from the *view component* of the MVC model.

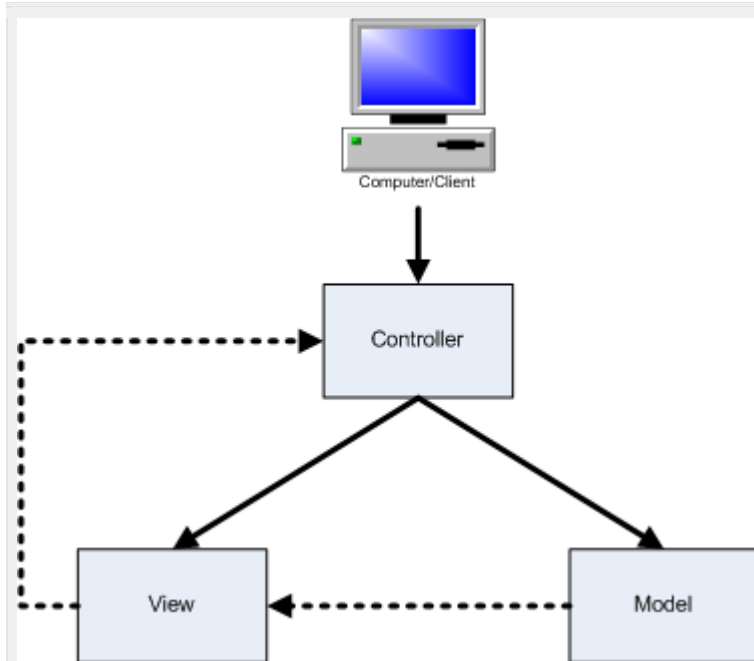
The XPage technology will help you make a more logical separation. First the repository will be the form where you just define the data types and Storage. The controller is Domino or the Lotus Notes client in release 8.5.1.

In designer you will use the XPage to define the user interface and business logic, while the form will be used for data Storage. Based on this statement, you may be thinking, "well, where is the MVC then?" This is really simple: the XPage is the View in the MVC model. In the XPage, you will be able to define the business logic running on the server (*then here is our C for controller*) which is managed by the http task of the Domino server.

The two diagrams below help to explain and support this approach:

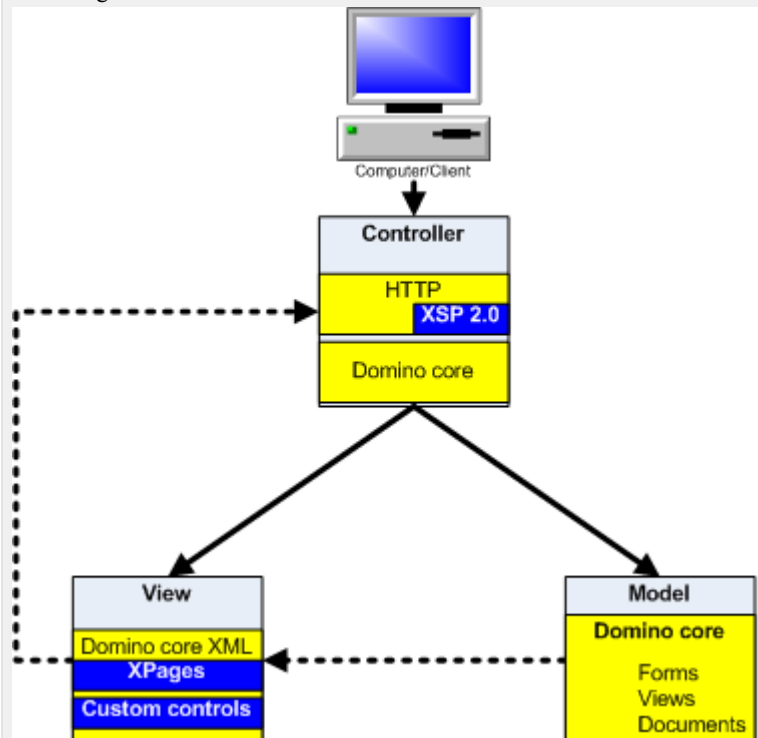
The classic MVC model

This image illustrates the classic MVC Model architecture



The MVC model for Domino

This image illustrates the The MVC model for Domino



Note - Although we do not specify the place for the Dojo toolkit in the graphic, you can use Dojo in XPages and components but also in forms and subforms. The dojo integration in Lotus Notes is there to alleviate the Java part and make development easier for all Lotus Notes designers. We should direct you to the IBM's project zero (IBM Websphere Smash you can access at this url : <http://www.projectzero.org/>).

Why should I care about XPages technology in Domino

If your are not a Domino only shop, but also use additional Lotus products such as Quickr, Sametime, and, or WebSphere Portal, you will most likely soon be confronted by your end user community to develop a coherent User Interface among all the software servers you are using. XPages technology will simplify this effort, thanks to the OneUI themes. Alternatively, maintaining such an interface with all these different products will be significantly harder to do if you instead continue using the traditional Domino development design elements. Even CSS files are not sufficient to share information between all platforms.

Note - In Section VII, we provide a very specific example and tutorial which illustrates how to build applications which incorporate the OneUI Theme.

Additional benefits which result from shifting toward XPage development include:

Separating Data from Presentation

The ability to more easily achieve a common User Interface between products and applications, while important, is certainly not the only goal to strive for. The more important objective is to separate the data layer from the presentation layer in all developments. With traditional Domino design, most presentation layers are tightly integrated into the data layer (with the forms, subforms, etc.), and in some complex development instances, this can result in an application which is extremely difficult to maintain. Keep in mind, it is not necessarily the result of the product by itself, but that many Lotus Notes designers have not documented their work. Accordingly, in terms of maintenance, it may be necessary to get third party products and spend significant time and effort to reverse engineer the original approach.

More possibilities for incorporating Java and executing LotusScript

Many designers have already developed within Notes using object oriented LotusScript libraries (or of course Java). But, we all faced the same limitations in that LotusScript can be executed only under certain conditions with Web applications, primarily using Agents. Java is also limited in terms of its integration with with traditional Lotus Notes development. So if you rely heavily on Java, XPages will allow you to go far beyond your earlier expectations with Lotus Notes.

Built on standard, proven technologies

The advantage is that Domino relies on standard technologies, so there is no proprietary code to manage. The technologies in use are Java, Javascript and CSS, so you will be able to find a vast array of resources on the web to help you with any development, sharing information, code samples and tips with designers from different horizons. Additionally, you will be able to delegate any design or development parts of your application to other team members with skills in these common technologies.

Custom Controls simplify development

And last but not least, the use of Custom Controls with XPages, will simplify your development effort *tremendously*! You will be able to exchange and reuse many components developed either by you directly, or by OpenNTF or other sources (including IBM). You are able to take advantage of these components, while still using the standard Lotus Notes mechanism to upgrade your applications (templates, etc.) so you will gain in productivity and efficiency.

In conclusion, using XPages will allow you to be more efficient in your development of great looking Web 2.0 applications *without* losing any existing Lotus Notes knowledge you already have in your organization.

What are the benefits over traditional Domino Development

Finally, we address the question of benefits from XPages development over traditional Domino development techniques. We will approach this topic using a framework of traditional Domino Design Elements for reference. As an experienced Domino Web designer you are accustomed to use many features that IBM Lotus development team had developed and ultimately "fine tuned" for web delivery. Basically Domino can run fine on the Web, but due to some inherent design limits, you needed to use some specific features bound to Lotus Notes architecture. For example, you have most likely developed some special forms to display views (the well-known \$\$ViewTemplate..... named objects) and some other components. In this section, we will list some of these traditional limitations and discuss how you can benefit by embracing the new approach in XPages development.

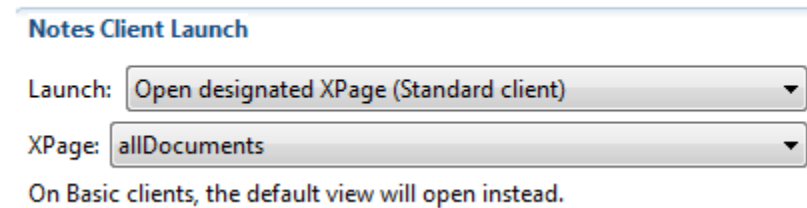
One approach for developing in a mixed environment

For web designers developing in a mixed environment (e.g. developing for Notes client users **and** web users), this often requires developing the same view twice, with each one slightly modified for the Notes or browser client. For instance, a Lotus Notes client user may need to see text headers, while the browser client should display more elegant column headers in a view. XPages provides direct benefit in this case. Using XPages, you can develop the code once and have it render properly as an XPages application - with an identical rendering in either a browser, or using an embedded browser in the Notes 8.5.1 Client.

So as a Designer you will use the same technology for the Domino server (web clients) and for the standard Lotus Notes client.

What is behind all of this? This is possible by selecting a simple option in the database property.

Launch Properties



Notes Client Launch

Launch: Open designated XPage (Standard client)

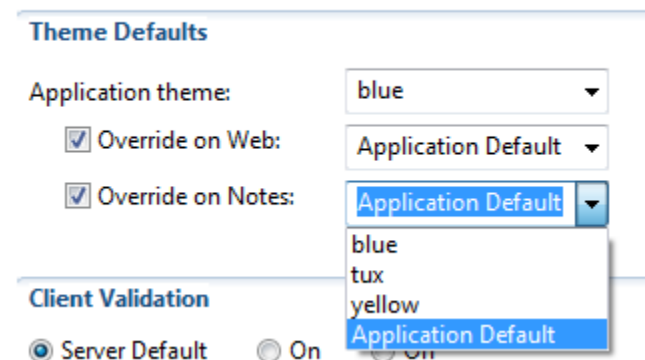
XPage: allDocuments

On Basic clients, the default view will open instead.

Alternatively, if you are running a Basic Lotus Notes client, (a version earlier than Notes Client 8.5.1) you will have to manage an option to tell the user that your application will not be able to take advantage of the new XPage features. One possible strategy for this is to dedicate the Lotus Notes basic client to a limited environment or to older machines with less powerful CPUs.

The Lotus Notes client can also handle its own theme if you want to distinguish the interface between the Notes and Web client. The theme can be defined globally to all applications on the server. Alternatively, some applications can be refined to use the global theme or use them on their own. We urge you to take a look to the sample applications and [tutorials](#) described within this Redbooks wiki, and to review the [Section for Server Configuration](#) for the location of the themes.

XPage Properties



Theme Defaults

Application theme: blue

☒ Override on Web: Application Default

☒ Override on Notes: Application Default

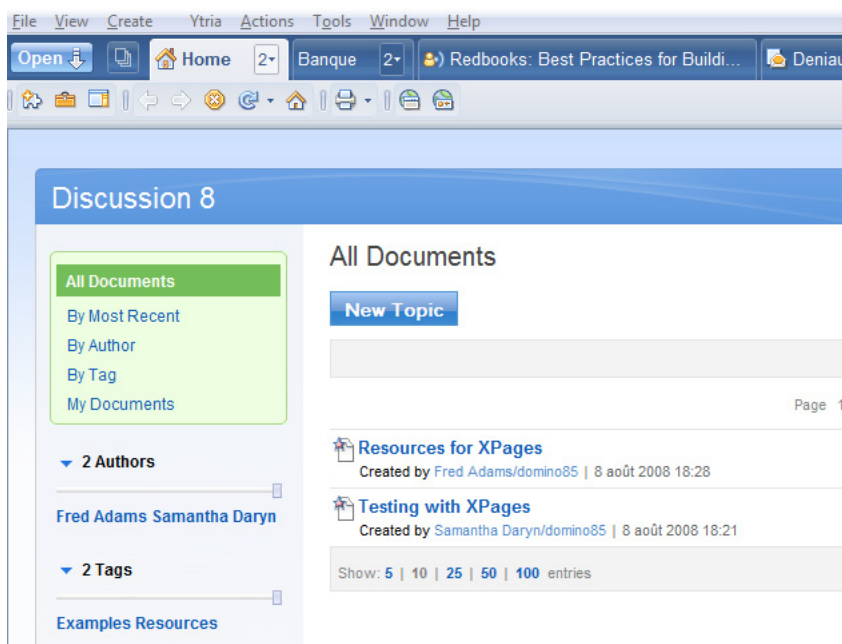
Client Validation

☒ Server Default ☐ On ☐ Off

Yes, this is a very powerful concept - "so if I do develop an XPages oriented application, I will no longer be faced with many of the earlier limits which existed for running applications on the Lotus Notes client?" For example:

- One view with multiple columns and multiple XPages presenting the data according to my needs all on the fabulous Lotus Notes client?
- Must I go to the web also for the thin web client?
- What about off-line applications? For web applications, we know that these can be used off-line if developed for a local replica, but the other important point is that the Lotus Notes client can also go off-line with XPages as a normal Lotus Notes application. The only requirement for the end user is to know how to replicate the database to the Domino server from their local client machine.

The next screenshot shows a standard XPage application running into the standard Lotus Notes client. The application is inheriting from the Lotus Notes discussion database.



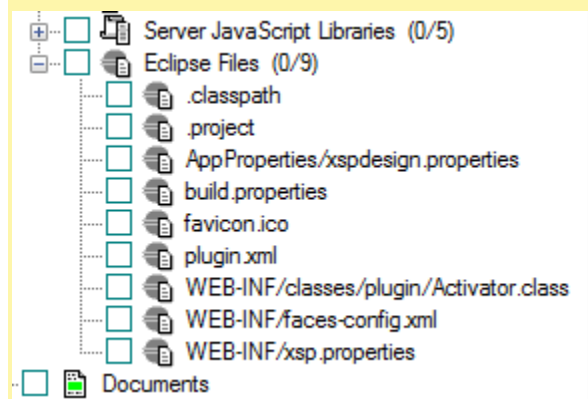
As we know, the Lotus Notes client is limited compared to the Domino server in terms of extended capabilities. The IBM team in charge of this new feature decided to integrate an independent web technology into the Notes client. The core is of course XML. since an XPage is basically XML. This allows a new Lotus Notes developer to be able to get into Lotus Notes design on their own for free (yes, the designer is free) without the use of a heavy Domino server on a machine such as the typical laptop.

What is happening behind the scenes - Basically the whole system is using the XULRunner on the client side. XULRunner is an application which executes XUL applications. XUL is a language based on XML that describes the graphical interface. You are already familiar with at least two applications: FireFox and Mozilla. Now we have a third one for consideration: Lotus Notes with its XPages.

Currently XUL is an engine built to render HTML/CSS and Javascript, so the Lotus Notes client limits to XPages reside in these three objects. So when you develop an XPages application for the Notes client,

your specifications need to be accurate and to keep in mind that the Notes client will be capable of rendering only HTML/CSS and Javascript. In simple terms the Lotus Notes client intercepts the XPages code into a XUL browser. The internal application launcher is stored in the Lotus Notes database (Activator.class) which is in charge to activate the environment for the whole XPages interface. All XPages are using internally two classes to perform fast actions. So when you develop an XPage, Designer will create two classes one is using the same name and the other one is using this convention naming `<XPage name>$<XPage name>Page.class`.

To mimic a file system directory all these files are internally stored in WEB-INF/classes/xsp directory except the Activator class which is located in the WEB-INF/classes /plugin. The directory is inside the Lotus Notes database and you probably will never have to bother about it. The plugin.xml file is created inside the Lotus Notes database to start the application on the client combined with the AppProperties/xspdesign.properties file. These files are located inside the Lotus Notes database. All these files are used with the local Lotus Notes client or on the Domino server, you do not have to modify anything except the database launch property to run your XPage on your Lotus Notes client. Here is an example of a Lotus Notes database content:



On the client side, the Lotus Notes client is using jar files located in the xsp directory of the Lotus Notes client. The jar files are dedicated to a special task (activation.jar, etc....) to run any XPage with any feature. The next folder contains Domino extension to run the XPages locally for extended features.

The xsp/shared/lib folder contains the core files for XUL to run locally and create the virtual machine that will interconnect the XPages application inside the Lotus Notes client.

The properties folder located in the data folder of your client and/or your domino server is the same.

Improvements in Views

View design elements are arguably the main point of entry into any Lotus Notes application. We mainly use views as an entry point to allow end-users to access, create and / or modify documents. Accordingly, as a Lotus Notes designer, you need to spend a lot of time anticipating and determining the specific

information that end-users want to see and manipulate.

Another problem can arise when talking about methods used to sort columns. To solve this challenge and arrive at the properly sorted / categorized view, some Domino designers have decided to design many views. Keep in mind however, that all those views are negatively impacting the overall Domino server performance. Additionally, a solution requiring multiple views that contributes to more analysis of your code from administrators who, in many cases, will not accept this design and development approach on a production architecture. Wouldn't it be much simpler to just have one view with all sorting options set and just use a design element that will allow you to select which column to use without impacting the server performance? XPages provides this solution. An XPages designer can select which column will be displayed just by selecting the columns from the view depository. This is a significant step forward for Lotus Notes design.

Another common challenge with views is that showing multiple views on a single page is not so easy. Of course, Lotus Notes can show multiple views, but our goal is to show *only a portion* of a view in one case and *another, different* portion of the view in another case. The XPage allows you do that and also to refine sort columns according to the context of your application. The designer simply needs to activate the sort option on the view column, and then choose whether or not to use it on the XPage. This allows for sorting options according to some specific conditions.

Finally, traditional Domino Development has made it difficult to link one view to another. In the past, we used framesets or better, composite applications, to solve this issue. Going forward, you will now only have to use XPages.

Improvements over traditional View templates

Designing forms named "\$\$VewTemplateDefault" and "\$\$ViewTemplate For <view name>" allows designers to display Domino views in a great looking way on the Web. This is certainly one approach, but is also limited. To work beyond these limitations, you will need to use Style Sheets and thus to define a correct style sheet within your application. Furthermore, you need to configure the whole server to have the same style across all your applications on that server to arrive at a common look and feel.

Alternatively, using a theme, a feature within XPages , will now help you in achieving the same goal. Themes contain information on which style sheets to load (computable) and any theme information for the core controls (also computable) – you set the theme to use in the database properties. Most importantly, using themes allows you to implement this in such a simple way that you will be more productive and efficient in your application development.

Improvements based on the traditional Form element

Working with XPages requires a slight adjustment to your thinking and approach to using traditional Notes and Domino Form Design Elements. No, you cannot use a form directly within XPages. Instead of

using a Form design element to display a document, you will instead need to use an XPage to display the document. For example, to display documents developed with a form called "Memo", you will have to develop an XPage called "Memo". By default Domino will try to find an XPage of the same name. In the form property, you will be able to change the behavior and to select a specific XPage to display the data. Keep in mind that when we say the "display", this is a very restrictive term. An XPage can be used to *create/display and modify* data.

So how will you work now? If a form cannot be used directly in XPages, what should we do? Should developers simply ignore Forms? **No!** XPages will just simplify your design.

What was wrong with traditional forms and why don't they apply going forward? Well, think back to your early stages of development in Lotus Notes. You simply added a form into a Lotus Notes Database to gather inputs and display information to the end-user. Basically it was easy at first sight. Eventually however, as the application grew larger, you often end up with a form being an object containing code, interface components, etc. So why don't we use the form simply to store information into a Lotus Notes database?

Here is where the XPages element comes into the picture. The XPage uses the form to store data into the Notes database, and only data, so the form ends up to be a repository for data and will not contain code anymore. Forms are used only to *store* data, but not to *display* the data, thereby eliminating the presentation aspect. Accordingly the end user will never "see" the form again. You can now consider the form as "hidden fields" and data storage.

All the application logic controls are now done at the XPages level. As a designer, you will be able to select where and when this control will be done. For example, you determine if the control will be executed at the server level or not, and during which event the control is executed. In the past you needed to use agents for control execution with web development.

Additionally, if you still want to do some data processing within the form, you can simply activate the **computewithform event** in the XPages properties for Documents.

Finally, another common challenge requiring maintenance and workarounds was how to show multiple documents linked together on the same page. Few Domino Developers had used the Editor embedded element that was present in the Domino 6 system to solve that issue. Now the XPage technology will help you doing that a simple way. Data can come from documents from within another Lotus Notes database or even from a non-nsf database.

Improvements to the functionality of WebQueryOpen and WebQuerySave

These two agents - namely the WebQueryOpen and WebQuerySave agents have constantly been a real headache for Domino administrators. The administrators needed to verify security and performance

before deploying these to any web application.

As a reminder, **WebQueryOpen** agents were designed to allow some modification to the document just before Domino renders it to the HTTP client. The **WebQuerySave** agent is a tool that allows designers to modify some information before the http submitted document is saved to the Lotus Notes database.

Essentially, this approach interferes with the document on load and then during the submit events. So to accommodate this, you often need to use Javascript on your own using javascript libraries. XPages now will help you solve such limitations.

You will no longer need to develop specific agents, but instead, the core XPages functionality will allow you to define code on specific events of your application.

Improvements to Rich Text Field Management

Richtext field management has been a complex area for Domino Developers, often requiring a special tool to properly manipulate Rich text data. Within XPages, the editor is now rendered with Dojo and there is an active rendering filtering that helps designer to forget about the complexity of rich text rendering while editing or rendering display state.

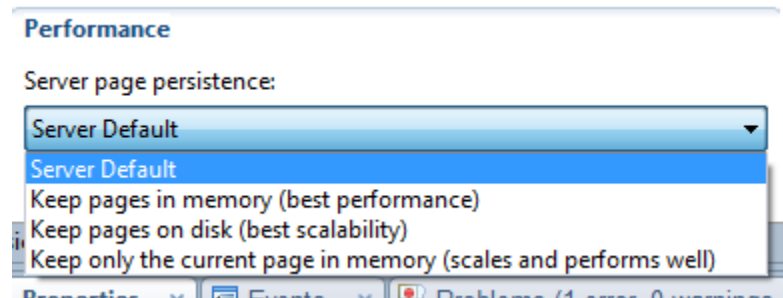
Built in AJAX support within XPages

To update lists fields, Domino submits the whole page back and forth to the sever. That is less than ideal when you need a responsive application on the Web. To solve this, one approach is to setup AJAX in your traditional Lotus Notes development environment. The downside is that setting up AJAX requires time and energy. Additionally, when you are working on views, changing the sort order will enforce you to use XML to do fast treatment. Wouldn't it be great to get a better way to do this with just one click and just right out of the box? XPages addresses this through built-in AJAX support so designers will be able to use partial page refresh and type-ahead features.

Performance Improvements related to XPages

XPages provide significant performance benefits for your Domino hosted environment. If you compare XPages to the traditional Lotus Notes development techniques for the Web, XPages brings new performance improvements. Much of this is due to the fact that in the traditional Domino development approach, both the Data layer and the Presentation layers are linked together with the form architecture. As a result, the Domino server has to maintain in cache the whole document and form until the end of the user action. Within XPages, IBM has setup the **NotesXspDocument**, which in fact is only alive when an exchange between the client and the Domino server is needed. When rendering an XPage, the Domino server loads the Notes document in memory, creates the NotesXSPDocument, and then removes it from memory. This performance efficiency will be especially noticeable on an overloaded Domino HTTP server.

A simple way to optimize your application is to define how the Domino server will handle XPages performance.



The server default option can be configured in the server properties. (**Note** - please refer to the [Section on Domino Server Configuration for XPages](#) for more specific information about this setting.) Alternatively, the other option that can help designers to define exactly how the server can manage the application without impacting the Domino server. According to the application type, the option selection will be defined easily by the designer. Moreover the administrator can also easily change the application behavior without launching the Lotus Notes Designer.

Closing arguments in favor of XPages

The primary problem for Domino and Web designers is one of productivity - how to develop the best applications in the most efficient manner. Lotus Notes development excellence is our goal, but in today's competitive environment, Lotus Notes designers need to be efficient, deploying simple or complex reusable objects you would use as in any object oriented language. Ideally, the only technical competency which should be required *should* simply reside in a mouse (namely the ability to drag and drop) or by using the keyboard to copy and paste the controls onto a blank page and start using your application. Using XPages, the pre-built Web 2.0 components and AJAX features make your life easier, while coupled within the robust secured workflow enabled framework provided by Lotus Domino. You also can take immediate advantage of custom controls that will help you clearly and simply define your interface and actions.

In favor of ONE

In summary, what definitely can help you decide to go XPages is based on the concept of "One". For example, we always - as veteran Lotus Notes designers - tried to have one of everything:

- One client
- One template
- One interface
- One server
- One language
- etc...

The simplicity of One is certainly appealing!

Eventually however, this evolved to require using:

- Multi-languages (@formulas and @command, LotusScript, Java, C/C++ api)
- Multi objects (views, forms)
- Multi everything

We can now say that much of the fuss about XPages is based on the appeal and simplicity of "One" again! Let's face it, as a Domino developer I now can develop for *one* client. Yes one client, since IBM has launched Lotus Notes 8.5.1 with the ability to render an XPage application directly within an embedded browser. So imagine, one form, one view for all!

Section III.

Value of Domino as a platform and Designer as the tool for building Web applications

Objective of this section

Domino is a platform for you to build powerful applications that can run on all major operating systems. You can build web based applications or client based applications that can run on Windows, Macintosh or Linux PCs. The Domino Designer client provides the tools to build all the applications you need whether they are web based or client based.

Client apps vs Web apps

There isn't a magic answer as to which type of application you should build. It will depend upon the requirements for the applications. If you are needing to support users that are not under your control, then a browser based solution would be the answer. With Notes supporting XPages in the client you will be able to start building applications that can run in both the browser and the client.

With a client you have direct control over the application and a lot of features that can not be supported in the browser. With Domino Policies you can make sure all the client version and setup are similar. With a browser you have little control over it's setup. Offline applications are easier to build with a client since replications handles the data.

Web based application may require less training since most users are familiar with a browser and links. Most users have had experience with the web and if you stick to standard web practices, they should be able to pick up your application.

While each method has it's own distinct advantages, the biggest advantage for web development is that you are no longer confined to the structure of the Notes Client. You have the flexibility to define your own structure to the UI. You can make a normal Notes view be anything you want, from a navigation menu to a catalog with images. You will be able to seamlessly blend multiple Notes applications into one web application. With XPages you can now add web 2.0 features without having to build your own. The dojo JavaScript library is included with the server and is part of the controls you can add to any web page.

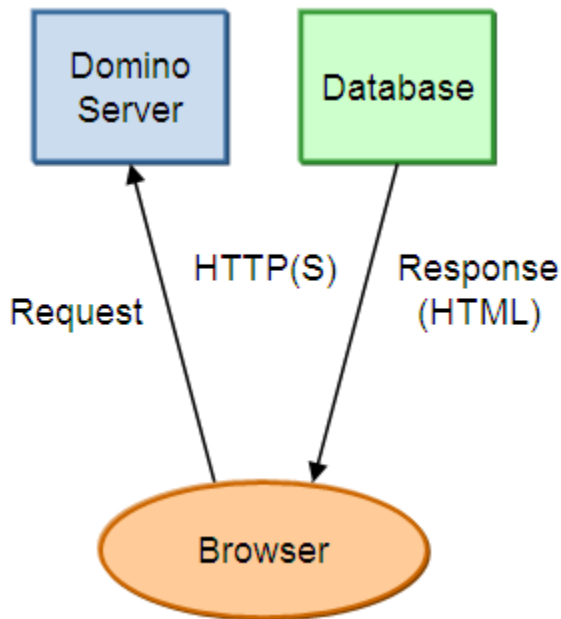
Domino as a Web server

IBM Lotus Domino provides an integrated Web application hosting environment which can be used to host web based applications containing appropriate code for the web environment. Domino translates all the requests from the browser into HTML and uses HTTP(S) protocol for displaying

the UI.

Developers have the option to store the web pages either as static HTML pages in the Domino file system or in a nsf file. The latter approach has its distinct advantages as it can display real time updates and be easily replicated to other servers.

The following diagram shows how the Domino Web server works.



Any web site based on Domino can consist of either single or multiple databases.

Domino supports following Web server features:

- Pass-thru HTML on Notes forms and pages.
- Security for applications using standard Domino security.
- Java™ applets.
- JavaScript™.
- CGI programs.
- Static HTML pages.
- Caching.
- URL extensions.
- Redirecting and remapping URLs and directories to another location.
- Multiple Web sites with separate DNS names on a single server.
- Failover using clustering.

Blogs and RSS feeds can also be hosted on Domino Web servers. IBM provides database templates (ntf)

for both of these as part of standard server deployment.

You can also enable Domino Web XML services on the server to connect with other non-Domino servers.

What kind of applications can be developed

A Domino web server can be used to host any kind of application that can be exposed on web. However, the power of Domino is best utilized when developing workflow based applications which can be up to any level. You can also develop dynamic workflow application which can be updated from the browser client itself.

A Domino based application can also be configured to work with other applications hosted on either Domino or non-Domino servers.

Other applications can access Domino data using any of the following options:

- DIIOP (Corba classes)
- Web Services
- XML
- DSAPI
- JSON

Web Services is the most recommended method for connectivity to other applications as Domino now provides inbuilt classes for both consuming and providing web service and they can be developed either in native Lotus Formula Language, LotusScript or Java™.

With the introduction of XPages in Release 8.5 the development of a web based application has been greatly simplified for Domino developers. A lot of tasks which earlier took many lines of code to complete are now just a matter of simple drag and drop.

We will see further advantages of XPages in later sections.

What is new in Designer 8.5.1 and improvements over earlier versions

The Domino Designer 8.5.1 client has been greatly improved with many features requested by customers. With the Designer now based on Eclipse, there is a major change in the look and feel. Do plan to take some time to become familiar with the new layout.

Here is a brief overview of some of them:

- **Working Sets**
Now you can group the databases that you work on in to sets. You can then select one or more sets to view at a time. The left navigation is no longer a long list of every application you worked on.
- **Design Element filtering**
You can filter the list of design elements using the filter field. Instead of searching through 60 views, you can enter a value to filter the view list to a manageable list.
- **LotusScript editor for Eclipse**
The new improved LotusScript editor based on Eclipse will make coding so much easier. You get class browser, templates, sort functions and more.
- **Template and Comment Templates**
There are new Notes Preferences under a section of the Domino Designer for LotusScript Editor. You can now define templates containing code used frequently that can be inserted into your code. Comment Template allow you to define documentation for each type of code module that is then automatically inserted each time you create a new module (class, sub, property get etc.).
- **Class Editor**
The new class editor allows the developer to see the outline of a library broken down into classes, properties, and methods. The code of each class is displayed as a single paragraph. The properties and methods are shown in alphabetical order.
- **Eclipse based Java editor**
Use one of the best Java editors around for editing agents, libraries, and Web services. All the features of Eclipse are available to you.
- **Notes Client Java UI API**
Now you can control the UI through Java. You don't have to resort to LotusScript.
- **Domino Designer Extension API**
You can build additional features and plugins for the Designer. You can utilize the large amount of information available from the Eclipse community.
- **Error Reporting**
The LotusScript error handling has been improved. The developer no longer needs to save the code to see if they have errors. As you complete statements, the editor places a red line next to each line containing an error. The editor allows you to save the LotusScript code even though it contains errors. The complete list of errors is displayed in a special error panel for more convenience.
- **Class Definitions**
The editor now recognizes custom classes. The developer just needs to hover over any declaration so that a definition of the datatype or class appears as a tool tip. If the class is a custom class, it just displays information about that class.
- **New LotusScript/COM/OLE classes**
The following 6 methods are new: NotesRichTextDocLink.RemoveLinkage, NotesDocument.GetRead, NotesDocumentCollection.StampAllMulti, NotesView.ResortView, NotesViewEntry.GetRead, and NotesViewEntryCollection.StampAllMulti
- **New Java/CORBA classes**
20 new methods and 1 new property were added.
- **XPage Support**
XPage will give you a powerful tool to build your applications. You will see what Xpages can do by reading this entire article.

- **XPage for Notes Client**
Build an application and have it available in the Notes Client and the web browser!
- **XPage offline support**
Run the XPage applications offline with the ability to replicate the data and designs, so you can work without a network.
- **Components based on XPages**
Now you can use XPages in the composite applications that you build.
- **Composite Application Editor**
The editor has been improved and new features added. For a full list see the Domino Designer Help.
- **Themes**

Section IV.

Reviewing underlying technologies and skills required to build Domino XPages web applications

The skills needed to build XPage applications are not difficult to acquire. You just need to understand the Domino data model, JavaScript, HTML, CSS and the Eclipse toolset. Your first stop would be to download the free Domino Designer and get familiar with it. If you are going to build web apps then you will need to have the main browsers and developer tools available.

Introduction to Domino Object Model

The Domino Object Model is a combination of a conceptual model along with a set of programs.

The conceptual model provides the framework for database access from programming languages such as LotusScript and Java and is the same for each programming language, with unique syntax for each environment.

Lotus provides a set of interfaces or APIs, unique for each language, that allows the language to access its core functions. For example, although LotusScript and Java are two different programming languages, the Domino functionality is provided for both.

The DOM can be broken into two areas:

- Front-end or UI classes
- Back-end or Data classes

Basically front-end classes are used to work with and manipulate objects directly visible to the user at that instant and the back-end classes are used to work with objects like stored documents or views.

Domino allows you to write an application that will run on a Web browser and access Domino databases. Since web browsers typically do not know anything about Notes or Domino, being able to access Domino databases from the Web browser in a client machine requires quite a bit of development the scale of which incidentally has been reduced a lot with the advent of XPages in 8.5.

Introduction to HTML CSS XML and JS and their role in XPages

Introduction to *Hyper Text Markup Language*

<http://en.wikipedia.org/wiki/Html>

Introduction to *Cascading Style Sheets*

<http://en.wikipedia.org/wiki/CSS>

Introduction to *eXtensible Markup Language*

<http://en.wikipedia.org/wiki/XML>

Introduction to *JavaScript*

<http://en.wikipedia.org/wiki/javascript>

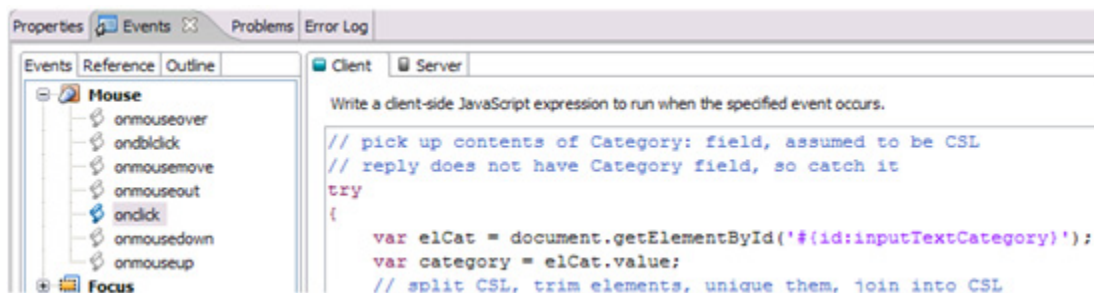
Role of these technologies in XPages

Although XPages utilizes JSF and the Dojo toolkit on the server end, the development is done with the help of new Eclipse based Domino Designer which contains an built-in XML based source code editor.

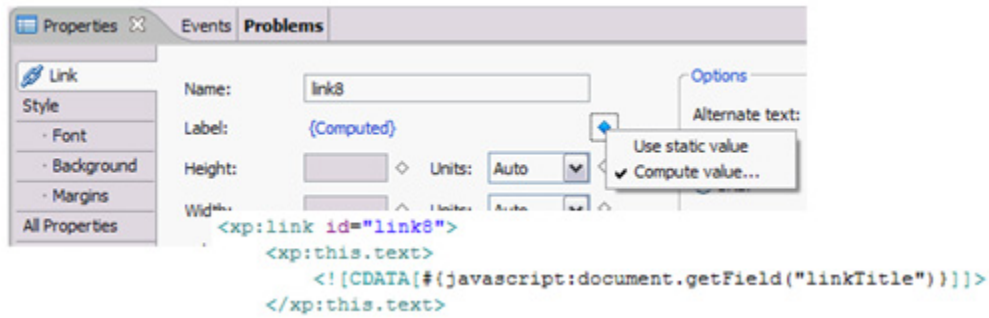
XML

XML forms the 'X' in XPages. Almost entire base XPages code when generated is in XML with its specific tags and developers can easily relate the code to the XML structure.

JavaScript is the “official” scripting language for XPages and can be used to act on both server side and client side events. Of course a server side script does add some load on the server.



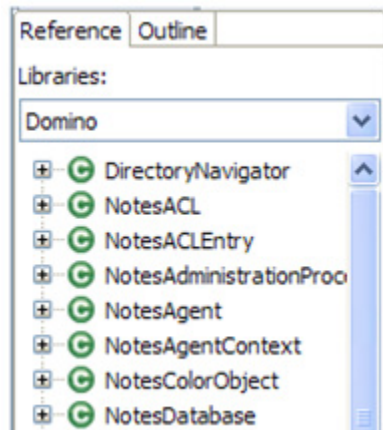
Simple actions are available for common operations and almost every property is computable using JS.



JavaScript

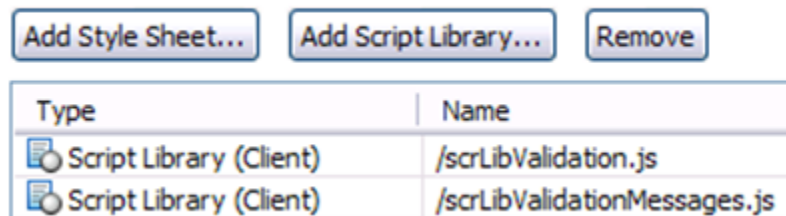
There is also built-in support for AJAX functions.

JavaScript in XPages works on top of the back-end JAVA API and has also been extended to support @Functions.



JavaScript Libraries are available as a resource to XPages and can also reference Dojo modules.

Resources



CSS

The entire look and feel of an XPages based application can be governed by CSS. There is full support for all the in-line classes of CSS available. You can also use CSS file resources just like in earlier versions. All the controls in XPages let you select a style from the CSS associated with the page. While you can set the font type, size, margins , etc., it is easier to create similar pages if you use a style sheet instead. With a style sheet, you only have one place to change to modify the style of a heading through out the application.

HTML

All the HTML tags and attributes are supported by XPages and can be easily embedded in the code as-is. Most HTML will be entered through the source view of the page where you see the actual XML source for the XPage.

Introduction to JSF

For an introduction to JSF and to better understand how this relates to Domino 8.5.1, please refer to this section ([XPages definition](#))

A peek inside Dojo

Dojo is a open source JavaScript library that is now bundled with the Domino server. The library provides controls and functionality that can be applied to web pages. While you could build the same basic functionality, it would be difficult to also provide cross-browser and localization support that dojo does..You can use the dojo library in classic web development or as part of a XPage provided control such as the rich text editor. Dojo Toolkit is used for some of the drag and drop core controls such as Rich Text, Date Time, Typeahead etc

Dojo consists of 3 parts:

- Core - Ajax, events, packaging, CSS-based querying, animations, JSON, language utilities, and a lot more. All at 26K (gzipped). Full support for CSS classes or inline styles
- Dijit - Skinnable, template-driven widgets with accessibility and localization built right in—the way you want it. From accordions to tabs, we have you covered. Uses CSS file resources

- DojoX - Inventive & innovative code and widgets. Visualize your data with grids and charts. Take your apps offline. Cross-browser vector drawing. And a lot more.

Lotus has modified some of the control to work better with Domino.

Rich Text Control - Now if your users need to enter more than just plain text, you can use the rich text control. It provides for a lot of control and it's improving with each release. Content entered through the control is saved in MIME format and will be converted to Notes CD if edited in a Notes client.

Warning - not everything converts correctly. The areas that may cause you issues are:

- Fonts - Web fonts are x-small, small, medium, large, etc while Notes fonts are point sizes.
- Table borders
- Embedded images – visible but lost when saved
- Tab tables – only the visible row is saved
- “Hide when” - if hidden from web are lost during the save

You do have the security concern if the user enters html with malicious JavaScript. The user can enter code that will be rendered when the page is viewed

For more information on Dojo and its use with the Domino server and client see this [article](#) in the designer wiki. (http://www-10.lotus.com/ldd/ddwiki.nsf/dx/XPages_Doj_1.3.2_toolkit_in_the_Domino_8.5.1_Server_and_Notes_8.5.1_Client) For information on the dojo library go to the home of the dojo toolkit:<http://dojotoolkit.org/>

Do I need to learn JAVA

The answer to this question is “NO”. Although XPages run on top of JSF there is absolutely no need for developers to learn Java.

What they do need to learn though are the entire core XPages technologies as explained earlier. It is not necessary for basic users but for advanced usage it is imperative that the developers should be familiar with all the technologies used. Java is available to the developer for agents, client UI, widgets and extending XPage functionality. Pick the language that you feel most comfortable with and go with that. Learning JavaScript though is a must, you will need it to do the field validation and creating server side JavaScript for workflow control.

Also, it is highly recommended that you get familiar with the Eclipse UI and terminology before using the new Domino Designer, otherwise, it can be a little difficult to get used to.

Section V.

Domino design elements specific to building an XPages application

Introduction to this section

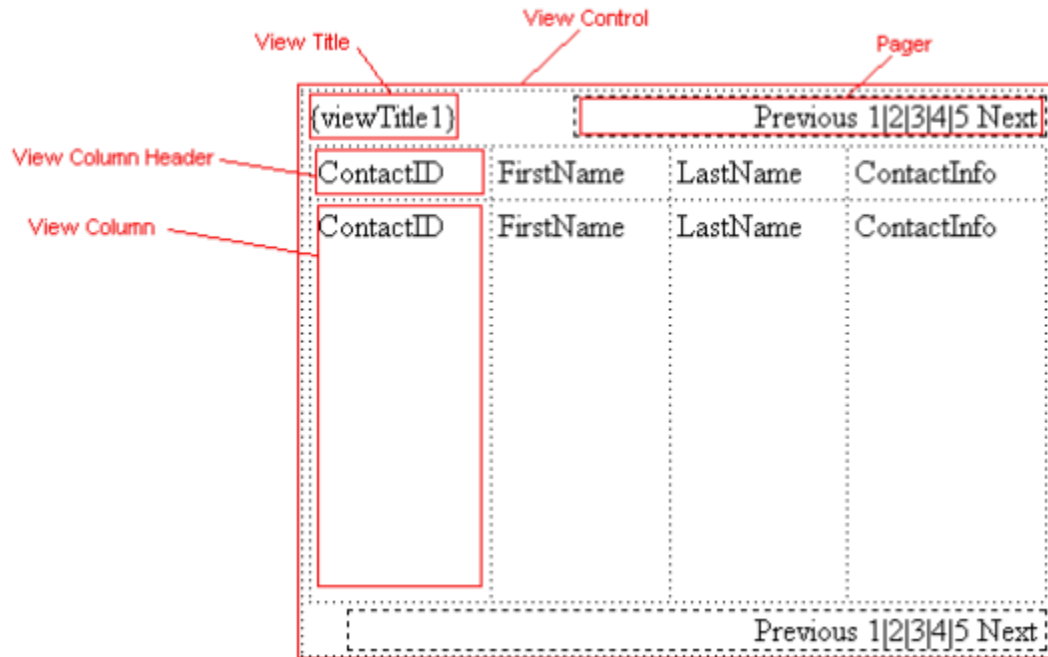
This section is intended to help you better understand the Domino design elements which are especially relevant within an XPage application. It also provides some recommendations for dealing with existing Domino code.

The design elements highlighted here include:

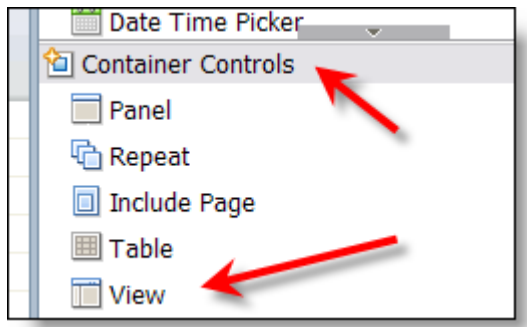
- The XPage View Control
- The XPage Repeat Control
- Themes
- Recommendations and Considerations for Re-use of existing Domino Code
- Methods for Active Content Filtering
- Methods for adding Java to the template and build path

XPage View control

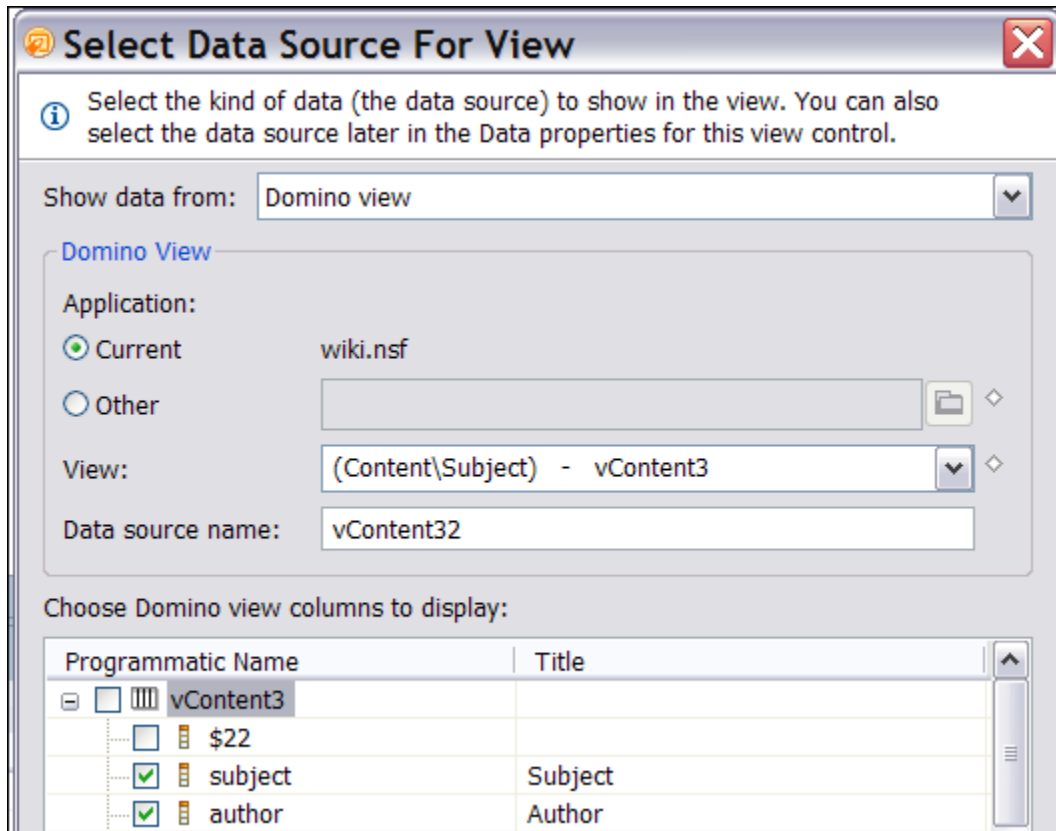
The **XPage View Control** is the main XPage building block for displaying tabular data that is quick and easy to use. The rows, columns, titles and pager are all added for you, allowing you to quickly customize it after it's created. You can embed multiple views in the same XPage, even from different nsf's. The View control automatically shows category “twisties” for categorized columns as before. Finally, note that it is not a property – it is read from view design.



The XPages View Control is a Container Control, and is Wizard driven, allowing you to get the view created quickly.

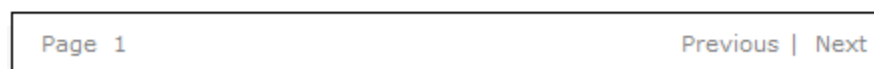
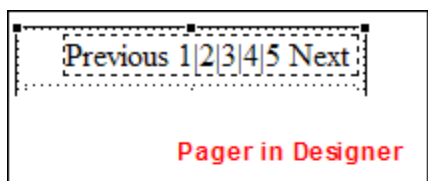


The image below illustrates the Wizard used for creating and defining the new View Control.



Pager

The pager is the navigation control used to move between pages of tabular data. It is automatically added as part of the view control



The pager has further configuration options that let you select which individual controls to show. These are set within the properties.

The screenshot shows the 'Pager style' dropdown set to 'Custom'. Below it, the 'Custom: Pager Controls' section is expanded, showing two lists: 'Available controls' and 'Display in pager:'. The 'Available controls' list contains: First, Previous, Next, Last, Page Selector, Current Page, and Separator. The 'Display in pager' list contains: Separator and Page Selector. A right arrow button is between the two lists. To the right of the 'Display in pager' list are three buttons: an up arrow, a down arrow, and a close (X) button. At the bottom, there is a text field for 'Number of pages to show in Page Selector:' with a small diamond icon to its right.

Pager style: Custom ▼

Custom: Pager Controls

Available controls:

- First
- Previous
- Next
- Last
- Page Selector
- Current Page
- Separator

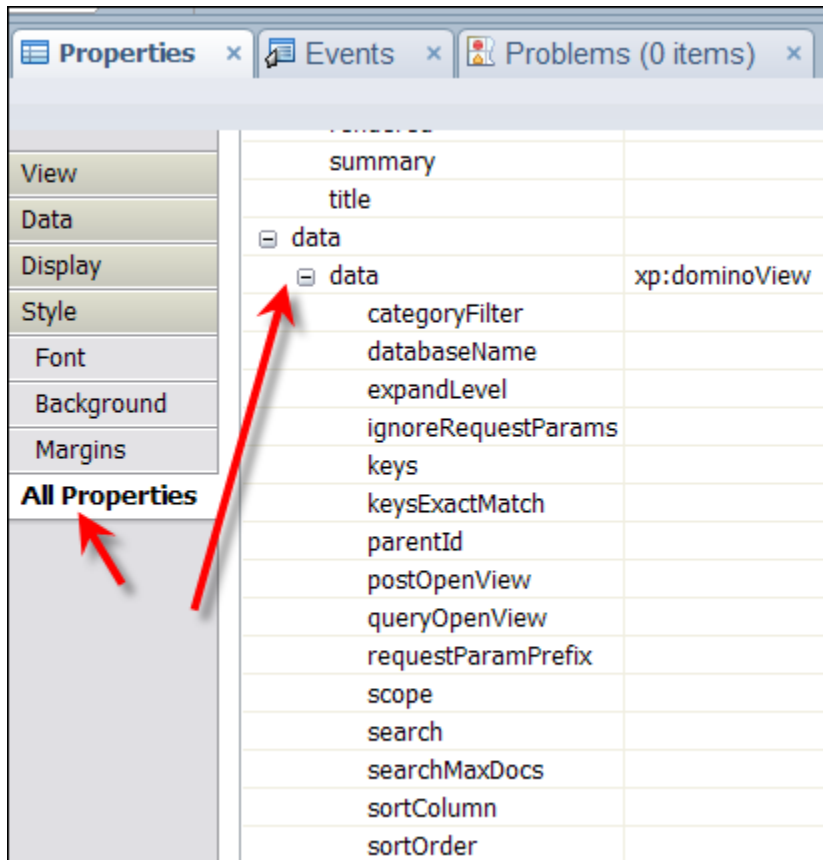
Display in pager:

- Separator
- Page Selector

Number of pages to show in Page Selector: ◇

Filtering within the view

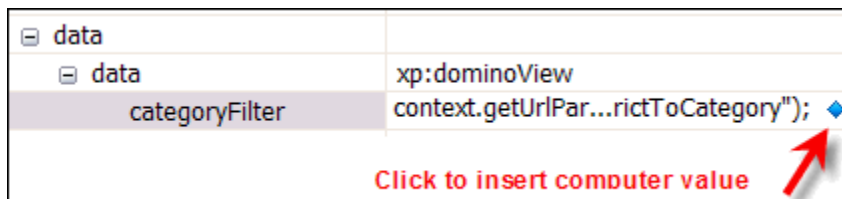
Views can easily be filtered via the **All Properties** of the view control panel.



Filtering the basic categorized view was done with the the setting "&restricttcategory=" in a standard Domino View or embedded view via "show single category".

Category	Subject	Created
▼ Customer self-assist		
	Domino Configuration Tuner (DCT)	03-Jul-2008
	Domino Policies - FAQ	23-Oct-2008
	How the new Dynamic Group Policies can reduce your administration workload	18-Dec-2008
► Deployment scenarios		

Now this is done through the property **categoryFilter**.



This can be a hard coded text value, computed value (e.g.from a custom control setting), or taken from a URL. To take a value from a URL use `context.getUrlParameter("RestrictToCategory")`. This would pull from the URL `"/xpage.xsp?&RestrictToCategory=Customer self-assist"` and restrict the view to the category "Customer Self-assist"

Filtering is done by using keys similar to the LotusScript `view.getAllDocumentsByKey`. This enables you to filter by more than one column by using the property "keys". This can be a hard coded text value, computed value, or taken from a URL

The example below illustrates how the filter takes a `java.util.Vector` as a parameter:

```
var v = new java.util.Vector();
v.addElement("Customer self-assist");
return v;
```

Style	title	
Font	data	
Background	data	xp:dominoView
Margins	categoryFilter	
All Properties	databaseName	
	expandLevel	
	ignoreRequestParams	
	keys	# var v = new java.util.Vector();...
	keysExactMatch	

Note - You can use the property "keysExactMatch" to specify an exact match value. The default is false.

Searching within the view

Filtering a view by using a search parameter is similar to standard filtering, but uses the parameter "search". It works then like a normal view control (navigation via pager etc).

Margins	keysExactMatch	
All Properties	parentId	
	postOpenView	
	queryOpenView	
	requestParamPrefix	
	scope	
	search	# context.getUrlParameter("searchVal...
	searchMaxDocs	

Note - You can also use the optional parameter using “**searchMaxDocs**” to restrict the number of results returned for search.

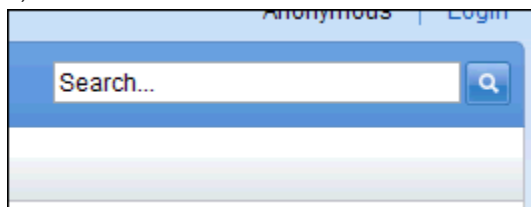
Searching Techniques

The following section describes one technique to request and process search.

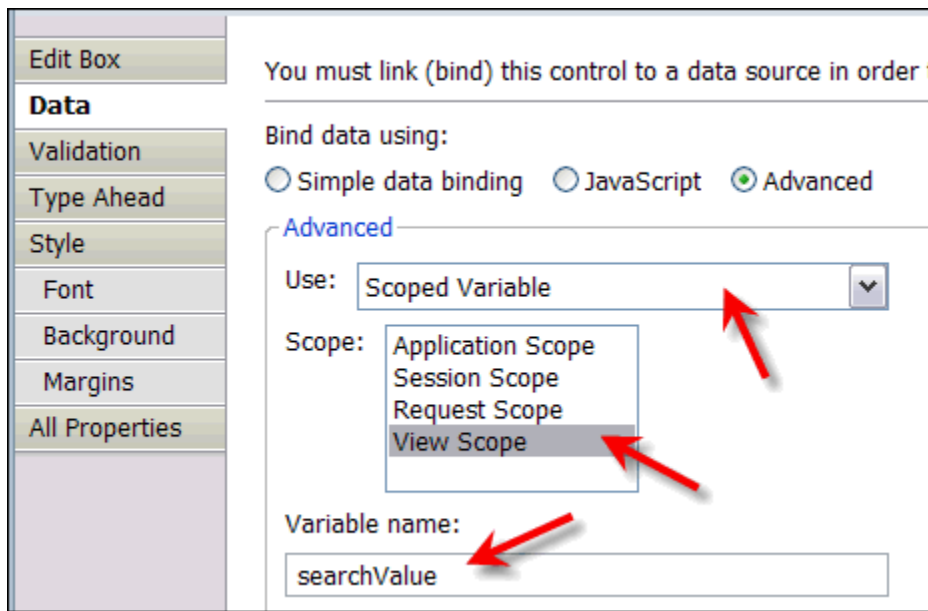
There are two parts to this:

- First Step - supply Edit Box to take search parameter. To do this,

1) Add the control Edit Box



2) In Data properties, link to “Scoped Variable” - “View Scope” and give data a name (this then places the data in memory to be picked up by search action Link)



Note - You can also use the optional parameter using “searchMaxDocs” to restrict the number of results returned for search.

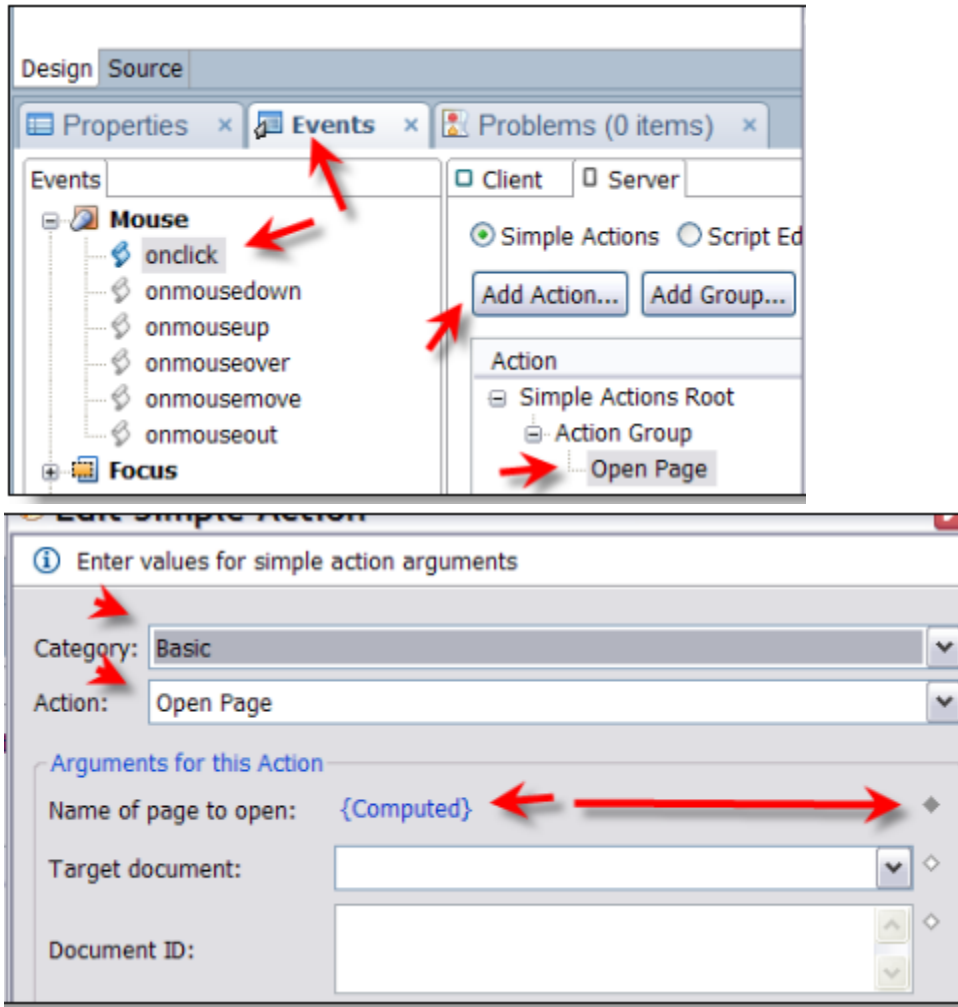
- Second step - supply the submit button to send the search parameter. To do this:

1) Add Link control

2) Using either a computed Link or onClick Event to compute the next page or url and retrieve the search value using “viewScope.searchValue”.

```
xpViewRecent.xsp?searchValue=" + viewScope.searchValue
```

3) From the wiki template – custom control “prtTitleBar” - is an onClick event to open an Xpage. The value is computed as shown below



Please refer to the additional View Control Resources on the Lotus Wiki

- [Working with View Controls](#)
- [How To Create a View Control](#)
- [Working with View Controls - Display HTML](#)
- [Working with View Controls - Computed Column](#)

XPage Repeat control

XPage control for “repeating” data is an essential developer tool. It enables you to create “views” of data like a view control, however with more degrees of flexibility (e.g. multiple rows of data per document). This can also be used for those small lists – computed sidebar links, lists of “anything” (e.g -


in the wiki – categories, tags, popular pages).

So use this approach to go from the very simple to the very complex.


Here is a simple example case shown below:

Documentation	4
Markup Examples	7
Release Notes	1
Sandbox	5
Simple	

And, here is a more complex example case. Building upon the complex example, you can allow multiple lists on the same XPage – from multiple sources.

 **Application Devt** Expand

Created by [Ravi Venkat](#) | 20-May-2009 15:48

 **Information not getting saved when submitting documents** Collapse

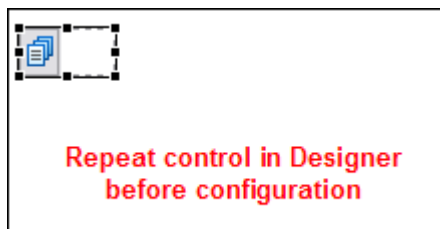
Created by [David Jayachandran](#) | 19-May-2009 21:16 | Responses: 2

Hi OpenNTF team, I am not sure why this is happening but when I save a document, sometimes the information is not getting saved. The document gets saved but there is no text (the document is blank). Sometimes I have had to re-type the text a couple of times and submit before it actually gets ...

Tags: Complex

Adding a Repeat Control to your XPage

The XPage Repeat Control is a “Container Control” that the developer manually configures using properties in Domino Designer



To begin working with the Repeat Control, first you assign the data source via properties and give a “Collection Name” as shown in the example below.

Options

☐ Create controls at page creation

Starting index:

Repeat limit:

Collection name:

Index name:

Collection Name

The image below shows how to create a link to the data source.

Repeat

Name:

☒ Visible

Iteration

☒ Simple data binding ☐ JavaScript ☐ Advanced

Data Binding

Data source:

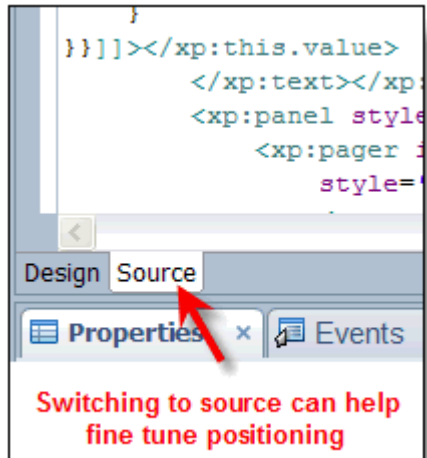
Bind to:

Need to create/link data source

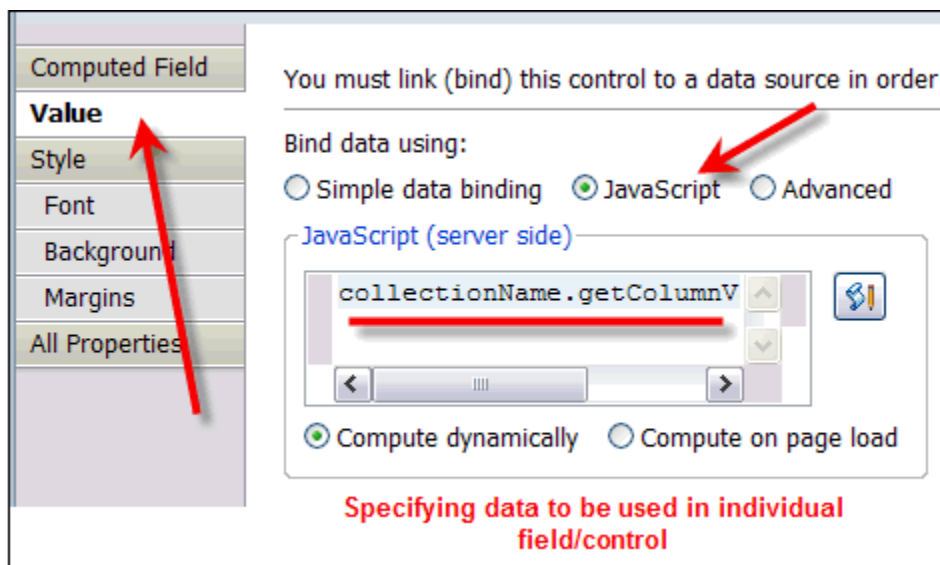
Adding Row Data

You can add anything to your repeat (text, field data, images etc). Just drag them into the repeat box (tip - use "source" for fine tuning)

You have direct access to the data source and can create your own html.



You access the data source using `collectionName.getColumnValue("field")` for each individual field/value.



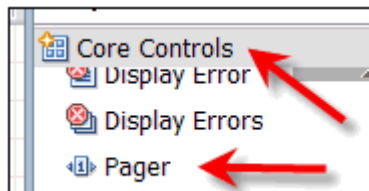
You can even nest repeat controls to give you even more power. For example, you could create a fancy list of data within each row, possibly from a multi value field of tags. The example below shows nested repeat controls.



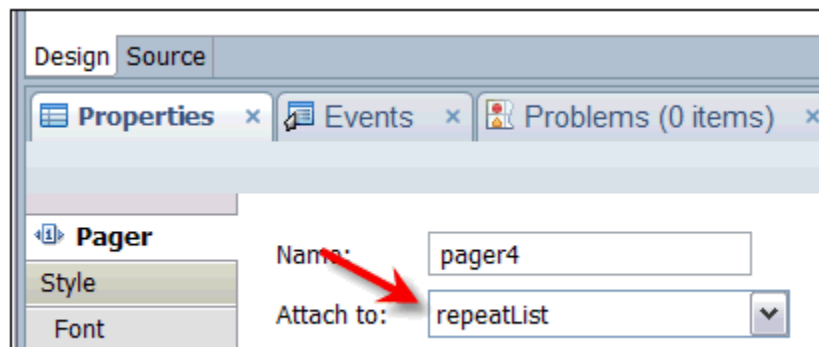
Note - See wiki template custom control “prtView” for the main view component. This uses a repeat control to display the view.

Pager

The pager is the navigation control used to move between pages of tabular data. Unlike the view control you need to add it/them manually using the “core control” “pager”.



When you add your pager you need to first attach it to the repeat control it should work with. This allows you to have multiple lists of working data on one XPage. The operation of the pager is then automatic with no further coding required.



The pager has further configuration options that let you select which individual controls to show. These are set within the properties.



Pager style: Custom ▼

Custom: Pager Controls

Available controls:	Display in pager:
<div>First</div> <div>Previous</div> <div>Next</div> <div>Last</div> <div>Page Selector</div> <div>Current Page</div> <div>Separator</div>	<div>Separator</div> <div>Page Selector</div>

Number of pages to show in Page Selector: ◇

Note - See the custom control “prtView” to see pagers top and bottom of the repeat list and the configuration used.

Filtering or Searching using a Repeat Control

Filtering or searching using a Repeat control is done identically to View Controls. (See the section above, [Filtering within the view](#))

To detect the Number of Rows, use SSJS `getComponent("repeatID")`. The count is then referenced `.getRowCount()`.

Themes

Themes are for determining which style sheets to use for your application and/or the styling for individual components.

- They are found under Resources > Themes (an XML file).

- You can set themes globally for the server (dominodata/properties/xsp.properties) or by nsf via application **properties > XPages**.
- They can inherit from other themes such as the in built WebStandard or OneUI ().
- The settings higher up the tree (themes you have extended from, styling set within the control) can be overridden by using
- Stylesheet and styling information is fully computable:
rendered="#{[javascript:context.isDirectionRTL\(\)](#)}"

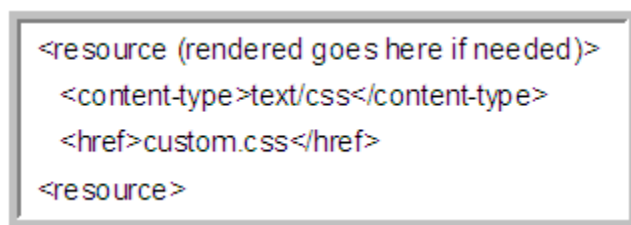
There are many advantages to using themes, although you can still just use a stylesheet on its own. Some of these advantages to using themes include:

- This allows you to build applications which are easily customized by a third party
- You can compute the styling by requested device/language type/browser type, so you can tailor your site to the user's browsing device.
- Core and developer component styling can be globally controlled (no need to set style information or class) eg buttons across all nsf's can be the same style – and one place edits this

Examples of ways to use themes

Here are some relevant examples for themes:

- Importing Resources (custom.css from nsf)



```
<resource (rendered goes here if needed)>
  <content-type>text/css</content-type>
  <href>custom.css</href>
</resource>
```

- Styling Core Controls that have no Theme id's

```
<!-- Label -->
<control>
  <name>Text.Label</name>
  <property>
    <name>styleClass</name>
    <value>xspTextLabel</value>
  </property>
</control>

<!-- Basic Input Text -->
<control>
  <name>InputField</name>
  <property>
    <name>styleClass</name>
    <value>xspInputField</value>
  </property>
</control>
```

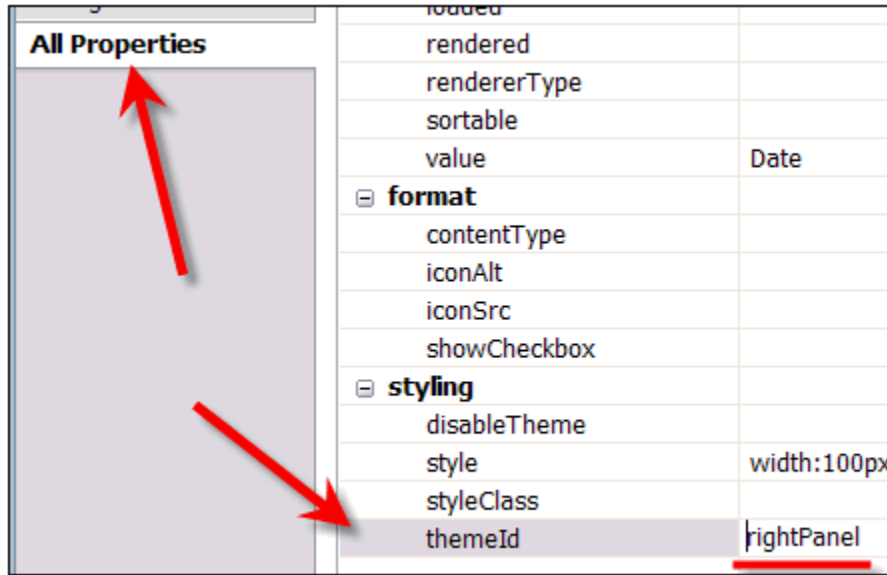
- Styling controls with Theme ids

You provide an element/control with a theme id, then refer to that id in your theme document which then assigns the styling. The benefit of doing this is having more global control and making it easy for others to provide customization's.

Here is an example of the code:

```
<control>
<name>rightPanel</name>
<property>
<name>styleClass</name>
<value>myRightPanelClass</value>
</property>
</control>
```

The illustration below shows the naming of the ThemeID within the **AllProperties** panel.



Extended use of themes

Themes can control more than styling, there are other properties can be set/overridden. For example a page title can be set within a theme file

```
<control>
<name>ViewRoot</name>
  <property>
    <name>pageTitle</name>
    <value>The page title you want to set</value>
  </property>
</control>
```

Style Classes can be set depending on scoped variables

```

      <control>
        <name>Link.collapse</name>
        <property>
          <name>styleClass</name>
<value>#{javascript:(sessionScope.ec == null || sessionScope.ec == 0) ? "xspDisabledCollapse" :
          "xspEnabledCollapse"}</value>
        </property>
      </control>
```

Reuse of current Domino code

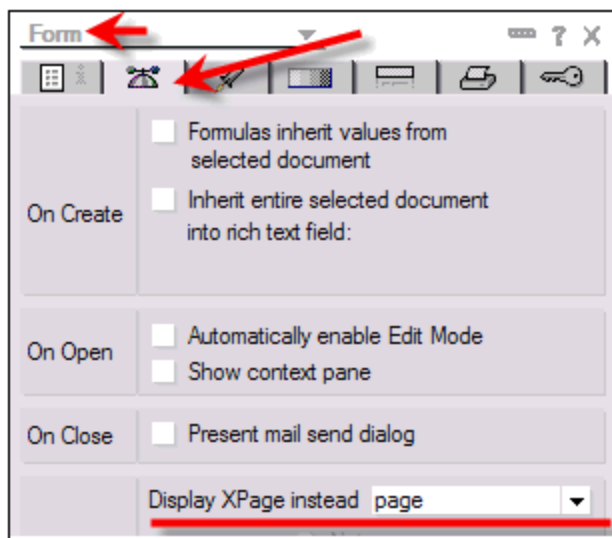
Building new applications is more fun, but we can also quickly provide old applications with a new lease on life with XPages. Things to consider:

- Bookmarks to existing web pages. The XPage url is different for pages and views
- Lotuscript is not currently supported in XPages directly - you may want to use existing Lotuscript logic to process data
- You can mix/match standard Domino and XPage design elements, however you cannot do things like re-use subforms in XPages – so you will need to recreate as a custom control

Note - Please refer to the section of the wiki for [Sample Application 1 - Modernizing existing Notes Applications - Converting an existing Notes Application into an XPage application](#).

URLs

Forms can be marked to launch the document instead using a selected XPage. Accordingly, if a user has a bookmark to an existing view/document url internally, the XPages code will redirect to the XPage url. The setting is in Form Properties, second tab “Display Xpage instead”, as shown in the image below. This is used in the Domino Wiki Template Form “Content” so the wiki template is backwards compatible with a standard Domino version.

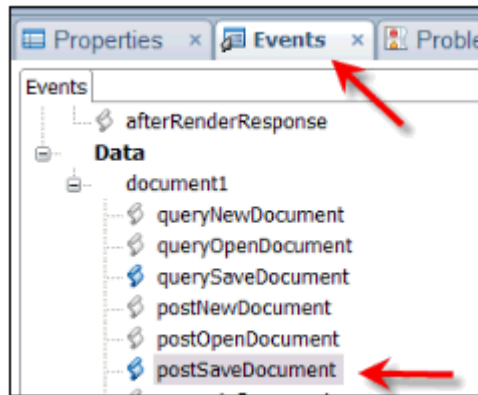


Reusing Existing LotusScript Logic

You may currently have a lot of logic sitting in LotusScript agents or libraries, and you want to use these when using XPages. While in the long term, using the XPages way of doing things is probably better, and there will hopefully continue to be improvements in this area soon, there is a way of leveraging say LotusScript based webQuerySave agents or script libraries.

You can do this by using the agent **RunOnServer(docid)** command (as when the agent executes, we do not have the usual documentContext handle) to run your agent after an XPage document save. The difference to note, is when you perform any actions on the document (via your code), it has already been saved, so you are not able to do anything before save. (These are the actions you will need to write in XPages javascript).

You can see this working in the wiki template custom control “prtContent”. The postSaveDocument code calls the agent “XPSavePage”.



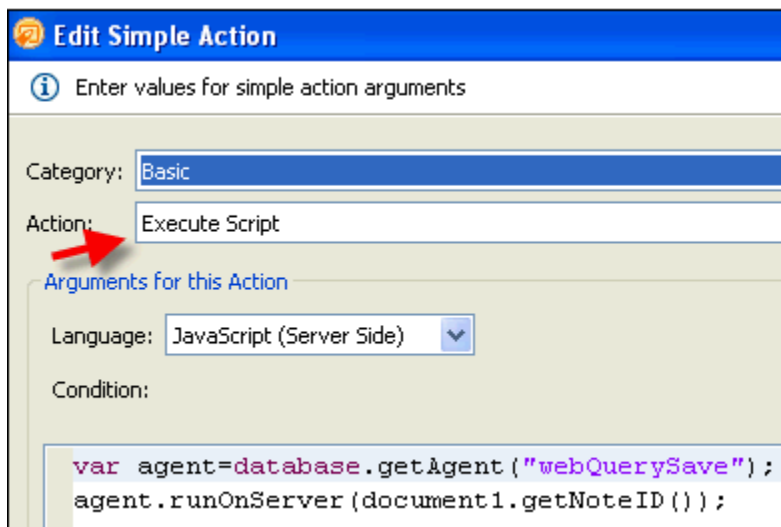
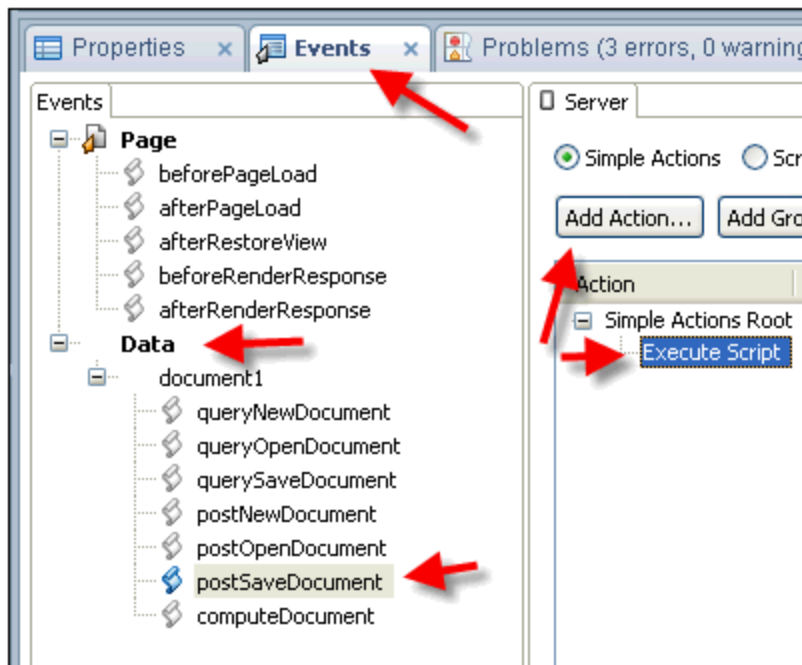
To do this you would create an agent as normal, and instead of collecting the document reference via session.documentContext, instead use agent.ParameterDocID via Set doc = db.GetDocumentByID(agent.ParameterDocID):

```
Dim session As New NotesSession
Dim agent As NotesAgent
Set agent = session.CurrentAgent
Dim db As NotesDatabase
Dim doc As NotesDocument
Set db = session.CurrentDatabase

'#### Get document (that we would normally get via documentContext)####
Set doc = db.GetDocumentByID(agent.ParameterDocID)

'#### Process Document from here ####
Msgbox doc.Subject(0)
```

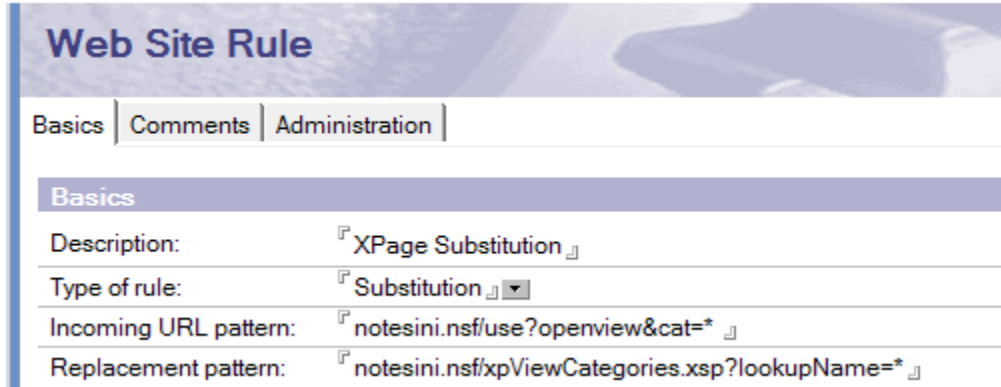
Then in your XPage, navigate to "**Events > Data > Your Document**" as specified already as a data source > postSaveDocument" add an action "Basic > Execute Script". You can then use the code var agent=database.getAgent("youragent"); to get a handle to the agent and then agent.runOnServer(document1.getNoteID()); to pass the document reference to your agent (which we collected using agent.ParameterDocID).



Use Web Site Rule Substitution

If you have a particular agent or view that takes url parameters and you want to preserve the url, but rewrite using an XPage, a Site Substitution rule could help you. Just add a new Web Site Rule to the server address book under Web / Internet Sites.

For example: to change from: `notesini.nsf/use?openview&cat=*` to `notesini.nsf/xpViewCategories.xsp?lookupName=*`, but keep the ability to pass a parameter over.

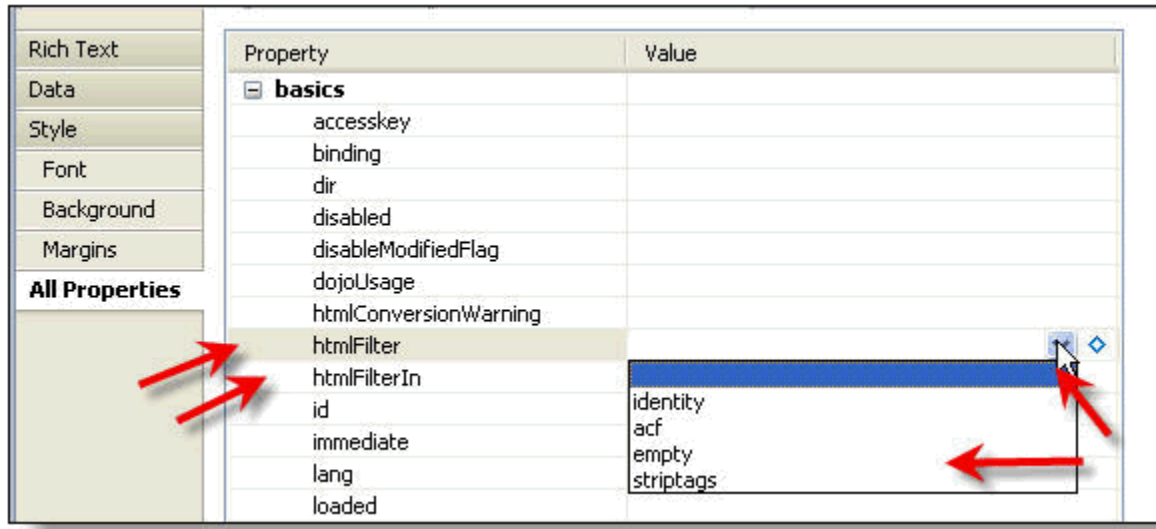


Web Site Rule		
Basics	Comments	Administration
Basics		
Description:	XPage Substitution	
Type of rule:	Substitution	
Incoming URL pattern:	notesini.nsf/use?openview&cat=*	
Replacement pattern:	notesini.nsf/xpViewCategories.xsp?lookupName=*	

Note - See the article [How can I open documents in a traditional Form rather than an XPage from a viewpanel?](#) for more information.

Active Content Filtering ACF

Uses a pure Java open source engine that is bundled with XPages/Domino. It can automatically Filters nasty html or JavaScript embedded within rich text or fields. You can switch it on through the All Properties and "htmlFilter" or "htmlFilterIn". The setting "acf" will do a full filter while "striptags" will strip html tags only.



Client Side JavaScript

You can on the Client Side reference server side code

tmp="#{serverScope.variable}" or tmp="#{[javascript:@DbName\(\)](#)}"

Client Side object location:

```
var o=document.getElementById("#{id:Name}");
```

Getting id of element:

```
"#{id:elementName}"
```

Server Side JavaScript SSJS

Full XSP and Domino object model

Can embed @Formula language within your code

Server Side object location:

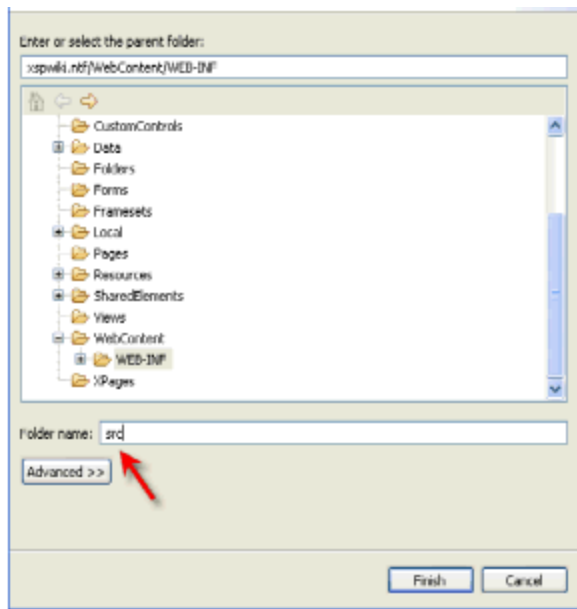
```
o=getComponent("Name");
```

Adding Java to template and build path

Goto java perspective and navigate to WebContent/WEB-INF.



Now Right Mouse click on WEB-INF - New Folder (name folder say src). Then go to Project > Properties > Java Build Path > Source > Add Folder. Select the WebContent>WEB-INF>src folder - and Click ok. Create new package Inside Local > your new folder



Using Pure Java within SSJS

You can build java classes within the java perspective of Eclipse/Designer and import them into your SSJS for use

ImportPackage (package) will do the import

```
wikiFormatting = function() {  
  
  //Import helper package  
  importPackage(com.ibm.domino.xsp.wiki);  
  
  // private class members  
  db:                NotesDatabase    = null;  
  subjectView:       NotesView        = null;  
  
  strHTML:           String           = null;  
  noWiki:            Array            = [];  
  sourceCode:        Array            = [];  
  sourceCodeCommands: Array          = [];  
  var url             = "";  
  var LINE_BREAK     = "\r\n";  
  
  //Instantiate java wiki helper class  
  var wikiP:          wikiParser      = new wikiParser();
```

Note - Special thanks to Steve Castledine and Peter Janzen for their help with the content in this section.

Section VI - Sample 1:

Introduction to enabling existing Notes Client Applications using XPages

This section describes the approach for web enabling an existing Notes client application using XPages. The approach is explained by means of a real-life case study, as it is based on the standard Lotus Document Library template. This example will help you getting started with bringing existing applications to the web. It will give you a realistic view on key issues which arise while doing this, and how to achieve the best results.

Note - This is the beginning a hands-on tutorial illustrating how to enable an existing Notes client application using XPages

This example will help you getting started with bringing existing applications to the web. It will give you a realistic view of the key issues which arise and how to deal with them during the development. Apart from providing a step-by-step tutorial to building the web version of the Document Library template, this section also explains the background of each step. It clarifies why things are done in a certain way.

In addition to the steps highlighted in this section of the Redbooks Wiki, [refer to this document to download the sample code](#) used in this example.

Overview of the pages which make up this tutorial

This section is divided into the following wiki pages:

Page 1 - current page

Describes the existing Document Library template and describes the features of the web application that is the subject of our case study. It also discusses some concepts in the approach to web enable the Notes client application.

Page 2 - [Getting started](#)

In this page, we will start building the web application and create a basic web form and view, demonstrating the use of XPage controls and the concepts of data binding and navigation.

Page 3 - [Website layout](#)

In this page, we will give the web application a nice look and feel, explaining how to use themes and how to set up a layout framework.

Page 4 - [Form design](#)

This page takes a deep dive into the design of the library document input form, covering many XPage features and explaining how to use them.

Page 5 - [Workflow](#)

In page 5, the existing Notes client workflow will be added to the XPage application.

Page 6 - [Response documents](#)

In this page we will add a reply input form to the library document XPage and we will display existing replies within the same XPage. This will reveal new XPage concepts like combining different data sources in one page and using a repeat control.

At the end of this tutorial, you will have web enabled most of the document library application, and you will be armed with a box of XPage tools and concepts.

The document library template Notes client application

As a case study for web enabling an existing Notes client application, we will be using the standard Document Library template that ships with Notes 8.51. This template, entitled "Doc Library - Notes & Web (8)" file name:

doclbw7.ntf actually hasn't been updated yet with R8 and R8.5 capabilities like composite applications and XPages.

Therefore, it acts as a perfect example for our case study. Especially because this template has been developed in earlier versions of Notes and has evolved "organically", it's a good example case for leveraging legacy applications with the new web features XPages offer.

Although the document library already does have a web interface (which is outdated), we will totally ignore this during the course of this case study: we will consider this web interface and all the related design elements as nonexistent, and we will purely use the Notes client interface as a starting point / as a reference. This is because we're focussing on web enabling a Notes client application using XPages.

For enhancing an existing domino web application with XPages. we refer to the section for **Introduction to Sample Application 2 - Building an XPage web application from scratch**, available at http://www-10.lotus.com/ldd/ddwiki.nsf/dx/Building_an_XPage_web_application_from_scratch

Document library description

The Notes client Library Document form looks like this:


Save & Close
Submit for Review
Mark Private
Cancel

Doc Library
Main Topic
Pascal David/EMD
Vandaag 11:42

Subject: Modernizing existing Notes applications
Category: XPages
Originator: Pascal David/EMD
Reviewers: Hendrik De Permentier/BE/USR/GF I-BENELUX
Review Options:
Type of review: One reviewer at a time
Time Limit Options: No time limit for each review
Notify originator after: final reviewer

Content:
This document describes the approach for web enabling an existing Notes client application using XPages. The approach is explained by means of a real-life case study: the standard Lotus Document Library template. This example will help you getting started with bringing existing applications to the web. It will give you a realistic view on most aspects that come on your way doing this.

Apart from providing a step-by-step tutorial to building the web version of the Document Library template, this section also explains the background of each step. It clarifies why things are done in a certain way.


ModernizingNotesApps.pdf

In the application's **Help - About This Application** document describes it as follows:

What does this database do?

A Document Library application is an electronic filing cabinet that stores reference documents for access by a workgroup. The database might contain anything from environmental impact statements for a group of engineers to financial statements for a group of loan officers.

Who will use this database?

Anyone who wishes to create a record of a document or review available documents may use this database.

Important Features

- *Web or Notes client:* Database can be accessed from either a Web browser or a Notes Client.
- *Review Cycle:* Used to route a document to a series of recipients.
- *Document Archiving:* Used to move expired documents to an archive database.

Main features

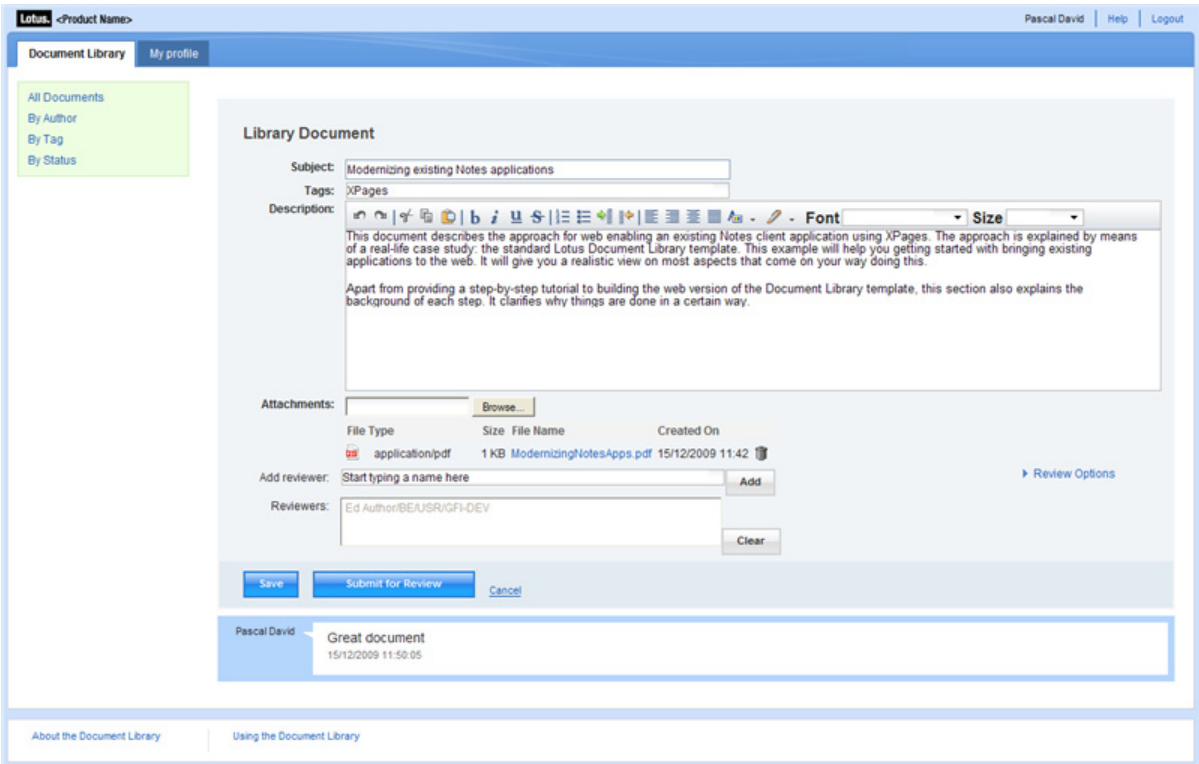
- Document library authors can create documents that can contain rich text and attachments (rich text Body).
- Documents can be organized in user-definable categories.
- A review cycle can be started. The author can select one or multiple reviewers and can select review options like Type of review (serial or parallel), Review time limit, and notification options.
- Users can mark their documents as private or public (default).
- Users can create responses on library documents and can create responses to responses
- The list of existing library documents can be viewed in a number of ways: By Author, By Category, By Review Status...
- Users can drag their favourite documents into a Favourites folder
- Archiving is possible via the standard Notes application archiving feature (File - Application Properties infobox). The document library template does not contain any particular archiving functionality

Scope of the case study

The main purpose of this case study is to demonstrate techniques for web enabling an existing Notes client application. This case study will explain how common Notes client features like workflow, response documents, views,.. can be translated into a web application with XPages. Also, we will demonstrate new capabilities of XPages which leverage your client applications with today's web applications' features.

Although this case study does not aim to deliver a completely finished, ready-to-use web application, it should bring you far enough to understand every aspect of web enabling a client application with XPages and to be able to finish the application by yourself.

A glimpse of what we are going to build: the library document input form:



Feature map

Based on the functionality of the existing Notes client version of the document library template, we will focus on web enabling the following features:

Notes client application feature	XPages web application feature mapping
Document library authors can create documents that can contain rich text and attachments (rich text Body).	Authors can add a rich text description to a document. Authors can upload attachments. Readers can download existing attachments
Documents can be organized in user-definable categories.	Authors can add their own tags when creating documents
A review cycle can be started. The author can select one or multiple reviewers and can select review options like Type of review (serial or parallel), Review time limit, and notication options.	The same review cycle options will be provided as in the Notes client

Users can mark their documents as private or public (default).	This feature will not be included
Users can create responses on library documents and can create responses to responses	Users will be able reply to existing library documents. Response to response will not be included
The list of existing library documents can be viewed in a number of ways: By Author, By Category, By Review Status...	Some views will be provided to display the list of library documents
Users can drag their favourite documents into a Favourites folder	This feature will not be included
Archiving is possible via the standard Notes application archiving feature (File - Application Properties infobox). The document library template does not contain any particular archiving functionality	This feature will not be included

Apart from just to copying existing Notes client functionality to the web, we will also demonstrate how we can make a "smarter" application than the existing one. For example, validations will be done at data entry instead of at form submit.

Conceptual design

Before starting to web enable a Notes client application, we need to think about how we are going to translate the application's functionality into XPages technology.

XPages and Custom Controls

XPages will act as the main entry points for the website: each XPage will correspond with a web page on our website and will have a unique url. The main website navigation will thus point to different XPages.

Custom Controls act as reusable subcomponents of XPages. A Custom Control is in some way comparable to subforms within a traditional Notes application. But a Custom Control offers many more possibilities than subforms: it can contain view controls, editable areas, other custom controls,... Developers usually decide to put form elements in subforms when they will be reused in several forms. With Custom Controls, other criteria for grouping elements in one control will need to be considered as well. For example, form elements that are linked to the same data source might be placed in one

Custom Control. Multiple Custom Controls with each its own data source can than be placed on a single XPage.

Layout versus content

The user interface of the Notes client is built with framesets, outlines, forms, pages, views....
Content is displayed in forms.

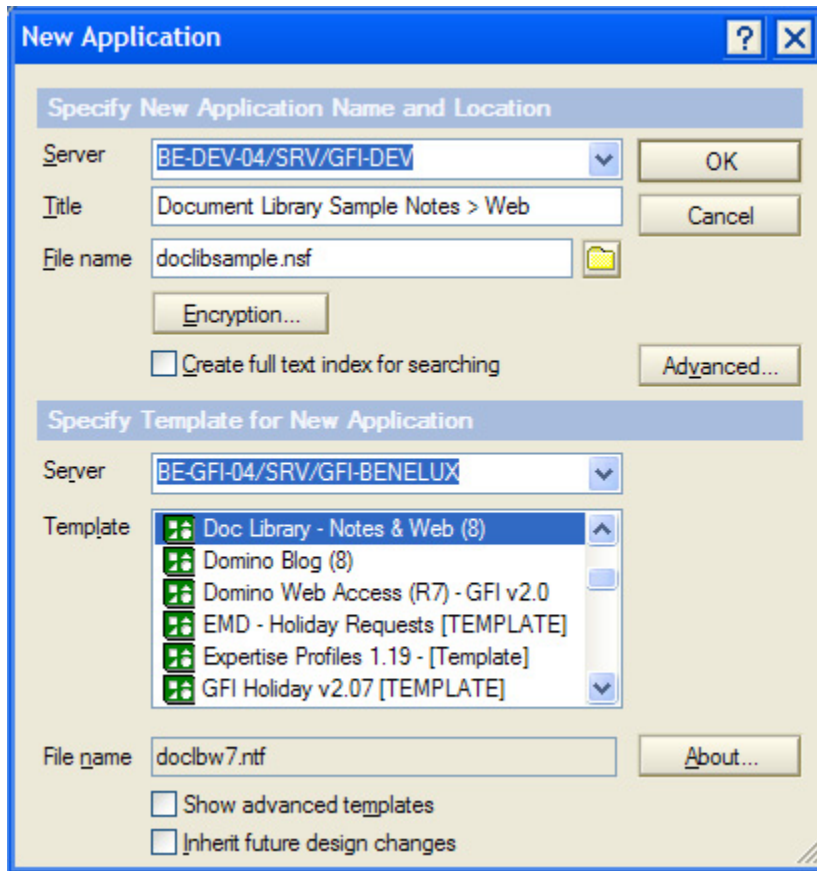
With XPages, the different layout elements are stored in custom controls, and the content is displayed in... custom controls. (or directly in the XPage)
Withing this tutorial, we will build a set of custom controls to create the website layout, and other custom controls will be used to represent the content.

Getting started - XPage enabling an existing Notes client application

In this section of the Redbooks wiki, we will build a basic input form for creating Document Library documents.

Creating the Document Library database

Create a new database based on the Document Library template:



Make sure you uncheck "Inherit future design changes" to avoid that your developments would be overwritten by a design refresh.

The default ACL set-up should be fine.

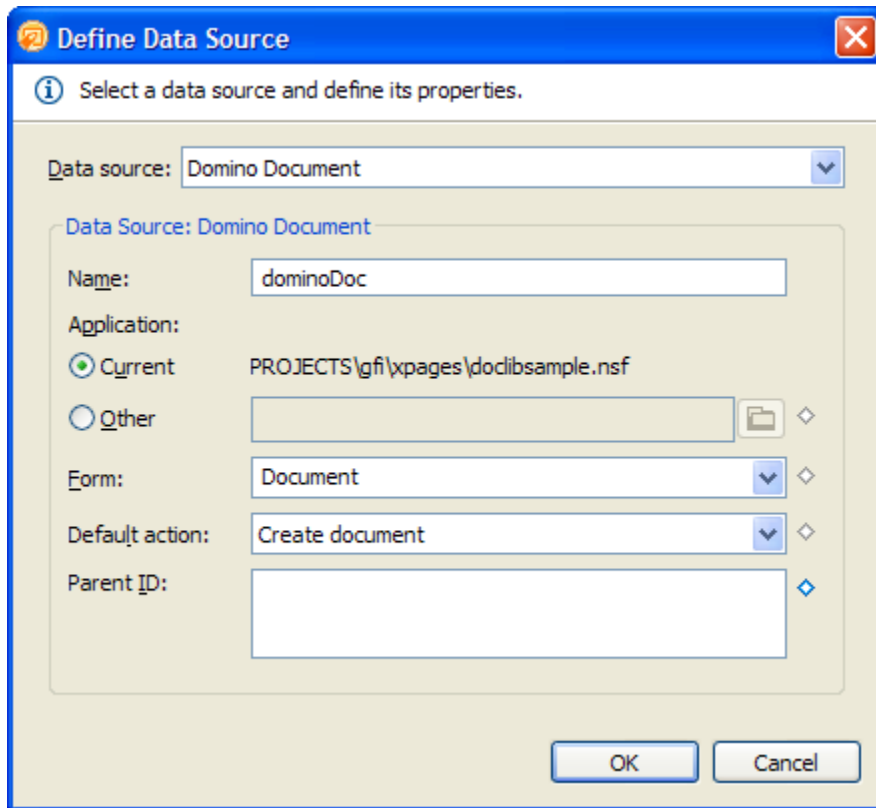
Creating a custom control

In order to build the web version of the Document form, we will

- create a custom control that will contain the form fields
- create an XPage that will include this custom control

Create a new custom control and call it **ccFormDocument**

In the data palette, define a data source of type Domino Document and call it **dominoDoc**:



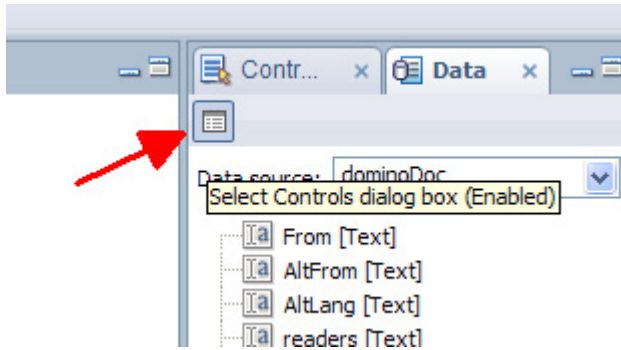
Defining a data source

There are several ways to define a data source in a custom control or XPage:

1. Via the data palette
2. In the custom control/XPage properties via the Properties view, on the Data tab
3. On each UI Control individually, via the Data tab in the UI Control properties

The first way is the easiest when creating a new custom control/XPage that needs several fields from the same data source (form or view).

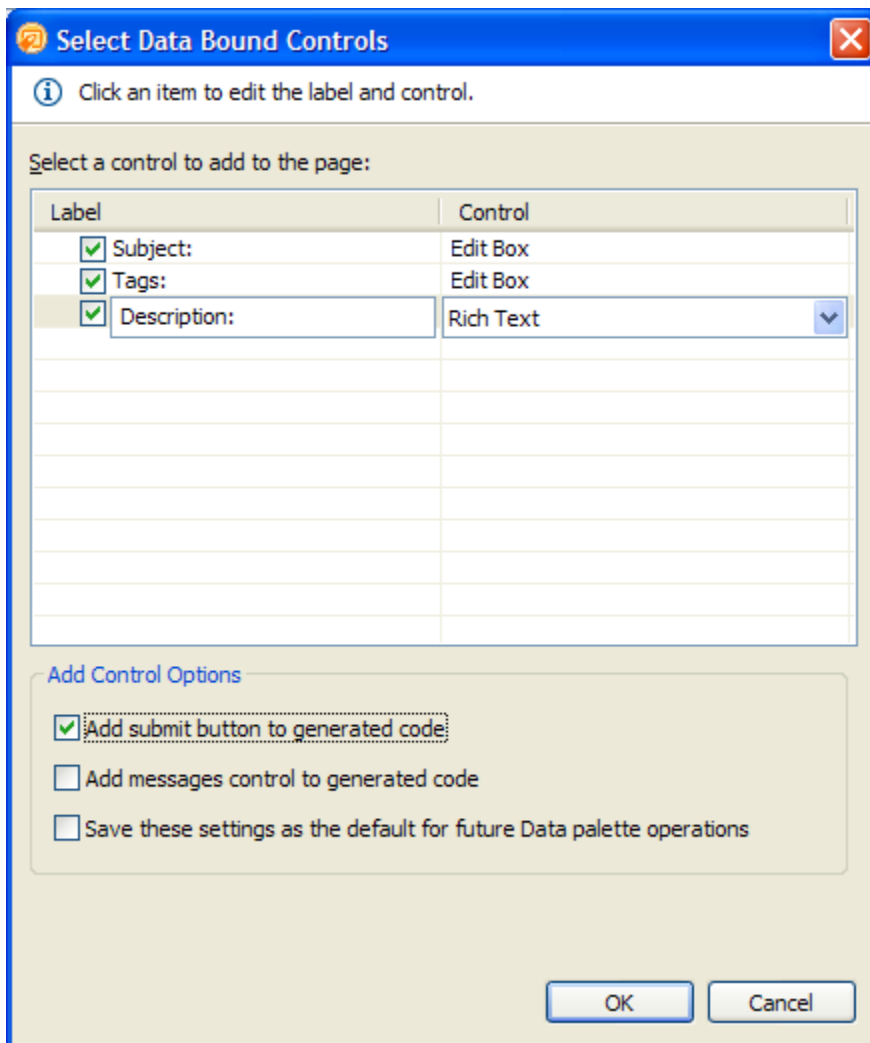
From the data palette, select the fields "Subject", "Webcategories" and "Body" and drag them into the editor. Make sure the "Select controls dialogbox" option is enabled before doing this:



This option will make a dialogbox display with options to define data types of the dragged fields. If disabled, the data fields will be directly dropped in the editor as controls of the type "Edit Box".

In the controls dialog box, adjust the field labels of "Webcategories" and "Body" and change the type of the Body field to "Rich Text".

Also, check "Add submit button to generated code".



After pressing Ok, the controls are added in the editor, and our basic form is ready

Select Data Bound Controls dialog

This dialog automates various tasks which otherwise would have to be done manually:

- Create data controls for the fields and perform data binding for each control
- Create field label controls and link them to the corresponding field ("select target control" in the label properties)
- Add a submit button, which includes a onclick event handler to submit the form. This becomes more clear when you have a look at the source code view:

```
<xp:button value="Submit" id="button1">  
  <xp:eventHandler event="onclick" submit="true"  
    refreshMode="complete" immediate="false" save="true"  
    id="eventHandler1">  
  </xp:eventHandler>  
</xp:button>
```

The first way is the easiest when creating a new custom control/XPage that needs several fields from the same data source (form or view).

Save and close the custom control.

Creating the XPage

Create a new XPage and name it **formDocument**.

From the Custom Controls palette, drag the ccFormDocument custom control into the XPage. Save the XPage and preview it in the web browser.

The result is a simple input form with a Submit button.

Enter some data in the fields and submit the form.

You will notice that, after submit, a new blank form is opened again. This is the default behaviour of an XPage after submitting: the same XPage is opened.

Verify the result in the Document Library application in the Notes client: the newly created document is stored in the database as main document. You will notice that the category field is empty. This is because we bound the Tag control to the Webcategories field, instead of the Category field. This is nothing to worry about now, as we'll handle that later on.

Renaming XPages and custom controls

You can rename existing XPages and custom controls by selecting the XPage/custom control in the list and pressing **F2** or via the menu option **File - Rename**.

Note that this rename function is **not** always available(it's grayed out), if you have set the designer preferences to open design elements via a single click (File - Preferences - General - Open mode).

The idea must have been to disable it when the design element is open. But even with no design element open, it's often grayed out. In that case, you can still rename the design elements by **in-view editing** the name in the view of XPages/custom controls: single-click twice on the name (not a double-click, 2 single clicks with about half a second time between them).

Managing attachments

With this basic form, we can create documents with rich text content, but we cannot add attachments. XPages provide controls for uploading and viewing attachments.

Open the custom control ccFormDocument

Insert 3 table rows above the bottom row of the data table (=the one with the submit button).

Leave the first row blank.

In the second row add

- a file upload control in the second column (drag it from the controls palette), and name it fileUpload
- a label (drag it from the controls palette) in the first column with as label value "Attachments: " (without quotes) and target control name fileUpload

On the data tab of the upload control properties view, select "Simple data binding" with data source "dominoDoc" and bind to "Body".

In the third row, add a file download control.

In the download control properties, check the boxes "Hide if no attachments" and "Allow delete"

On the data tab of the download control properties view, select "Simple data binding" with data source "dominoDoc" and bind to "Body".

The data table in ccFormDocument now looks like this:

Subject:	<input type="text" value="Subject"/>								
Tags:	<input type="text" value="WebCategories"/>								
Description:	<div> <div></div> </div>								
Attachments:	<input type="button" value="Browse"/>								
	<table border="1"> <thead> <tr> <th>File names</th> <th>Size (bytes)</th> <th>Type</th> <th>Created on</th> </tr> </thead> <tbody> <tr> <td colspan="4"><input type="button" value="Submit"/></td> </tr> </tbody> </table>	File names	Size (bytes)	Type	Created on	<input type="button" value="Submit"/>			
File names	Size (bytes)	Type	Created on						
<input type="button" value="Submit"/>									

Save your changes and preview the XPage formDocument. Enter a subject, a description and an attachment. Submit the form and verify in the Notes client. As you see, the document contains both the description and the attachment in its body field.

Binding mutiple controls to a single data field

In general, you cannot bind mutiple controls to a single data field. For example, if you would create to Edit Box controls and both bind them to the Subject field on dominoDoc, only the value of the last Edit Box would be saved in the document.

File upload controls make an exception to this, in combination with rich text controls. You can create a rich text control and a file upload control and bind them to the sale data source (a rich text field on the underlying form), and both of their data will be stored in the rich text field. Also, mutiple upload controls can be added on the XPage and can be bound to the same data field.

Notice that upload controls in XPages store their value differently than with traditional Domino web applications. In traditional applications, the attachment is not stored in a rich text field (as there is no data binding with a field).

Displaying runtime errors

Chances are that something went wrong while previewing the XPage in the above steps, for example because you forgot a step within the sequence, and you got a runtime error when viewing the XPage in the web browser. By default, the displayed error is very vague and almost impossible to trace.

If you don't know what I'm talking about, add a computed field custom control to ccFormDocument and give it a severside javascript value, for example `this.runtimeerror()`. This meaningless and will generate an error for sure.

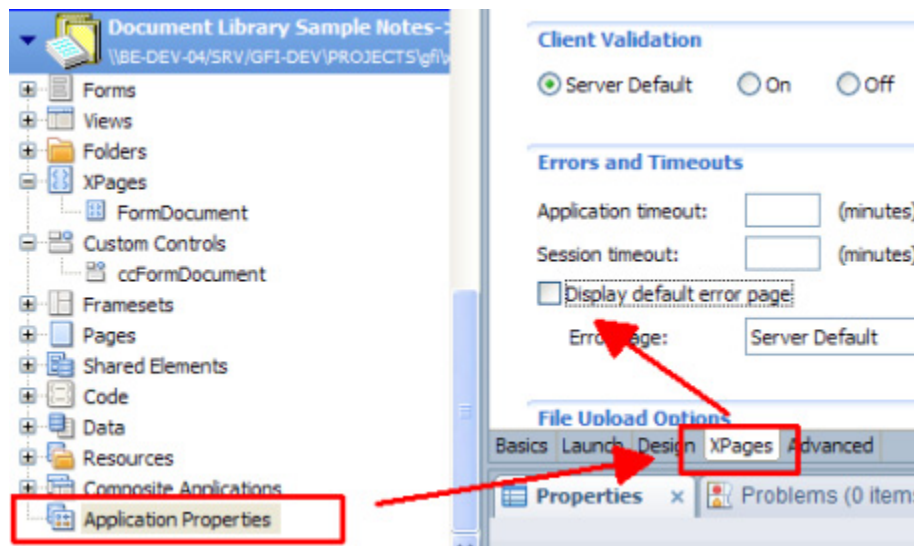
When you then preview the XPage, a blank screen will appear with the message "Http Web Server: Command Not Handled Exception".

In XPage applications, the way errors are displayed can be configured:

In the Applications Navigator, go to the "Application Properties" design element of the Sample Document Library application.

In the Application Properties, click on the XPages tab.

Under "Errors and Timeouts", the field "Error page" is set to "Server Default" and "Display default error page" is not checked:



Check the box "Display default error page".

Save and close the Application Properties and try to preview the XPage again.

The cause of the runtime error is now much easier to find:

Unexpected runtime error

The runtime has encountered an unexpected error.

Error source

Page Name: /FormDocument.xsp
Control Id: computedField1
Property: value

Exception

Error while executing JavaScript computed expression
Script interpreter error, line=1, col=6: Unknown member 'runtimeerror' in Java class 'com.ibm.xsp.component.xp.XspOutputText'

Javascript code

```
1: this.runtimeerror()
```

► Stack Trace

Displaying documents in a view

So far, we are able to create simple documents via the web. But we don't have an interface to existing documents yet.

In this topic, we will create a simple Notes-style view from which we can open existing documents.

Create a new XPage and call it "viewAllDocuments"

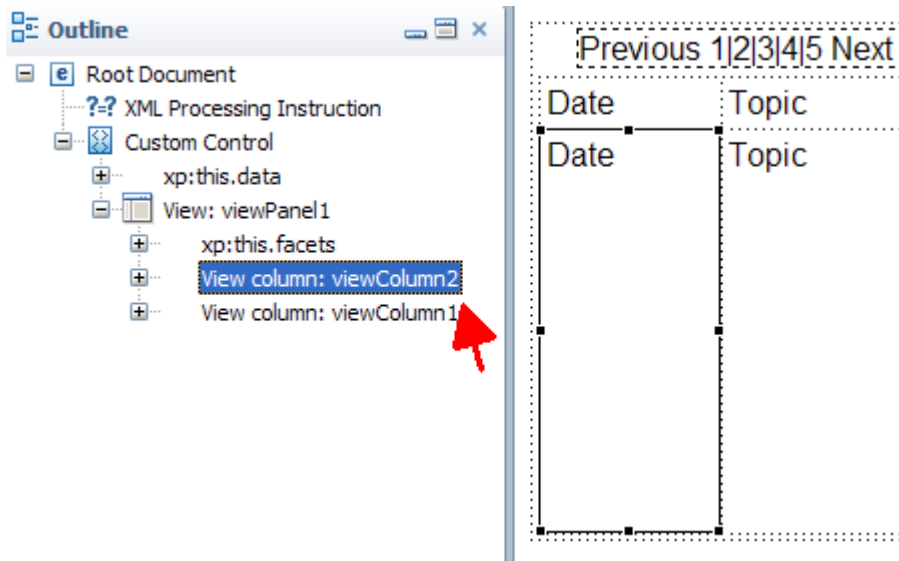
Create a new custom control, called ccViewAllDocs

Define a data source in the custom control of type DominoView. Select the view "AllDocuments - (\$All)".

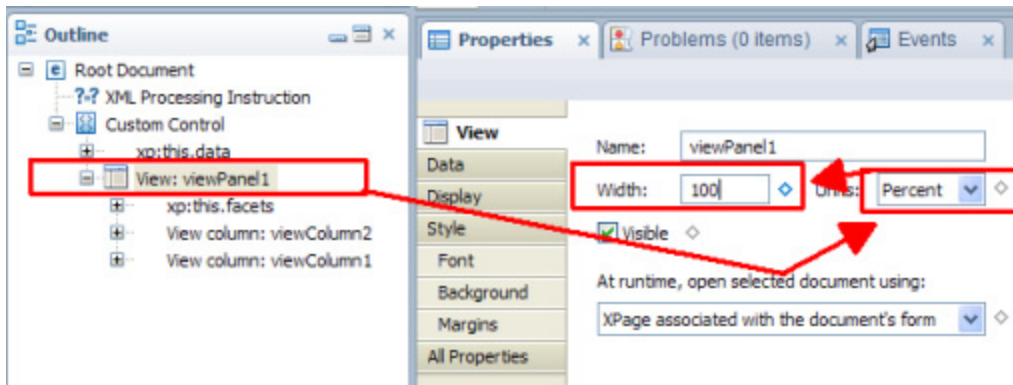
Name the data source "dominoView"

Drag and drop the columns "Topic" and "Date" from the data palette into the custom control edit area.

Make the column "Date" the first column, by dragging it above the Topic column in the outline view:



Select the entire view (click on it in the outline view) and adjust it's properties: set the view width to 100%:



Select the Topic column in the outline view and in the Properties, Column Display Properties, enable the option "Show values in this column as links" with Document open mode set to "Read-only".

Save the custom control.

Open the Xpage viewAllDocuments

Drag ccViewAllDocs in the edit area and save and preview the XPage

The result is a simple view of the existing documents. The subjects are displayed as links, but clicking in them returns an error (Item Not Found Exception). This is because no interface has been specified yet for displaying existing documents.

Linking the form with the XPage

So far, we have built an input form to create **new** documents and a view to display a list of existing documents. Now, we need to assign a web interface to **existing** documents which were created with the Document form:

Open the Notes form "Document" in Designer (yes, our good old existing Notes form) and display the form properties info box.

On the Defaults tab (=second tab), under "On Web Access" set the value of "Display XPage instead" to "formDocument."

Save and close the form.

If you now click on a link in the XPage viewAllDocuments in the browser, the corresponding document will open with the XPage formDocument.

Linking an XPage with a form

An XPage is linked with a Notes form in two ways:

1. Via data binding: if the data source is of type "Domino Form", and the default action is set to "Create new document", a new document will be created at submit and an item "form" with the form name will be added.
2. In the form properties. Since multiple XPages can be linked with the same form as data source (and each XPage can have several data sources), an additional definition is needed to specify with which XPage existing documents need to be opened. This is done in the form properties, as demonstrated above.

Completing basic navigation between form and view

Now, we are able to create new documents and view and edit existing documents. In order to have a basic, workable application, we need to finalize navigation:

- define an application entry point
- provide a button to create new document
- return to the view after a document has been submitted

Application launch options

When the Document Library database is launched in the web browser (=root database url: <http://hostname/path/database.nsf>), the application's old school web interface is displayed, rather than one of our XPages. This is normal, since we still need to tell our application to open in a different way:

Open the Application Properties design element, and go to the Launch tab.

Set the web launch options to open a designed XPage: viewAllDocuments.

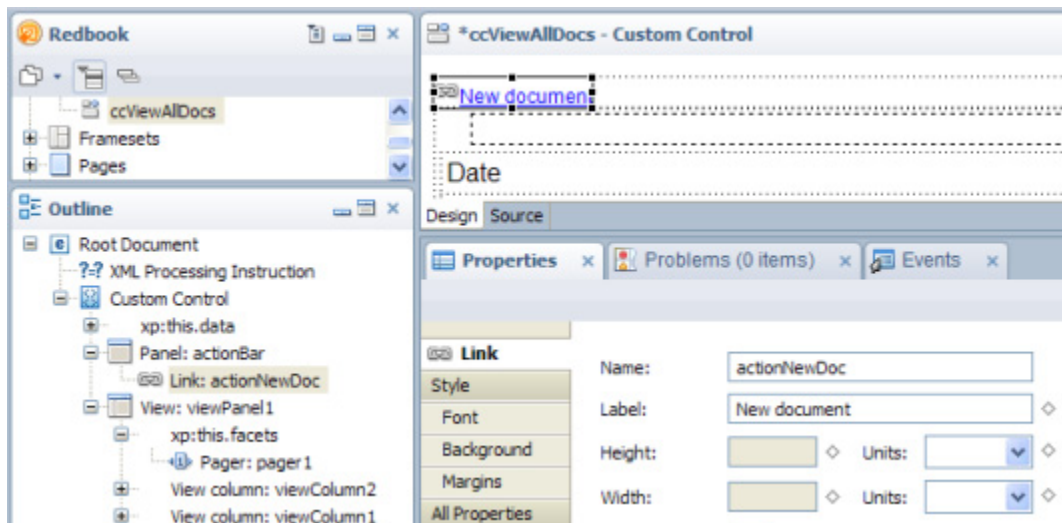
Save the Application Properties and open the database url in the browser to see the result.

New document button

From the all documents view, we want to be able to create new library documents. This will be achieved by adding a button "New" on top of the view:

Open the custom control ccViewAllDocuments and drag a Panel container control above the existing view panel. Give the panel a name, "actionBar".

From the controls palette, drag a Link control into the actionBar panel and give the link a name "actionNewDoc" and a label "New document":



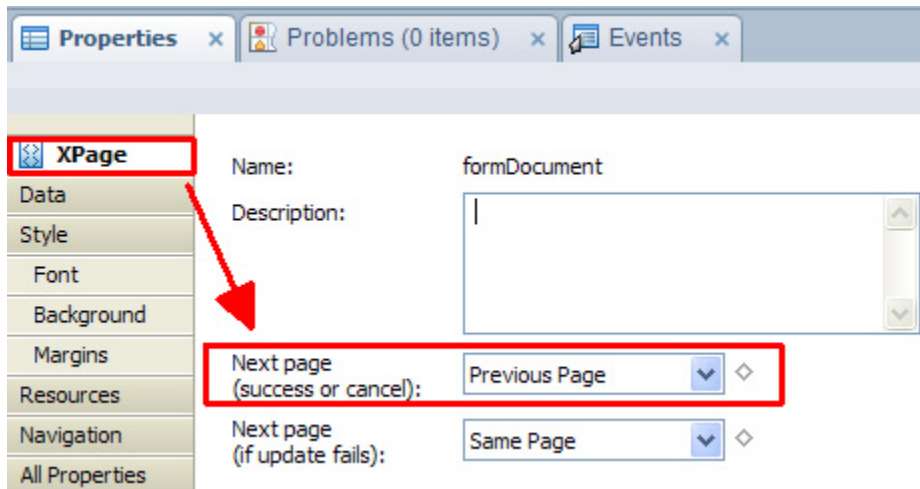
In the Events view, add a simple action: Open Page. Name of the page: formDocument. Target document: New document.

Save the custom control and preview the result in the browser. New documents can now be created from the All Documents view.

Return to the view after submit

Finally, we need to tell the XPage to return to the view of all documents after submit, instead of returning to the same page:

Open the XPage "formDocument" and go to the first tab, "XPage" in the XPage properties view. In the field "Next page (success or cancel)" change the value "Same Page" to "Previous Page":



Save the XPage.

Defining page navigation after submit

The target page to be displayed after submit can be defined either at XPage level (as in our example), or at Submit button control level. In the above example, the same result would have been achieved by adding a second simple action to the submit button, which opens the previous page. In that case, the control level navigation would override the globally defined navigation.

In the above example, selecting "viewAllDocuments" instead of "Previous Page" as option for "Next page (success or cancel)" would have given the same result. But "Previous Page" is more generic: when more views will be created, this option will return the user to the view he was in prior to opening the form. Just be aware that "Previous page" doesn't work in case the opens the page url directly in the browser (for example, when he added it as a favourite, or when he clicks on the url link in an e-mail). The next page can be made conditional to handle this (click on the diamond to compute the next page).

Summary for Section – Sample 1 - Getting Started

In this section, we have created basic form and view functionality, comparable to traditional Notes client applications. Basic XPage concepts have been introduced, including data binding and navigation. The next sections will focus on layout and form enhancements, revealing the power of XPages as a platform for rapidly web enabling existing Notes client applications.

Create the website layout - XPage enabling an existing Notes client application

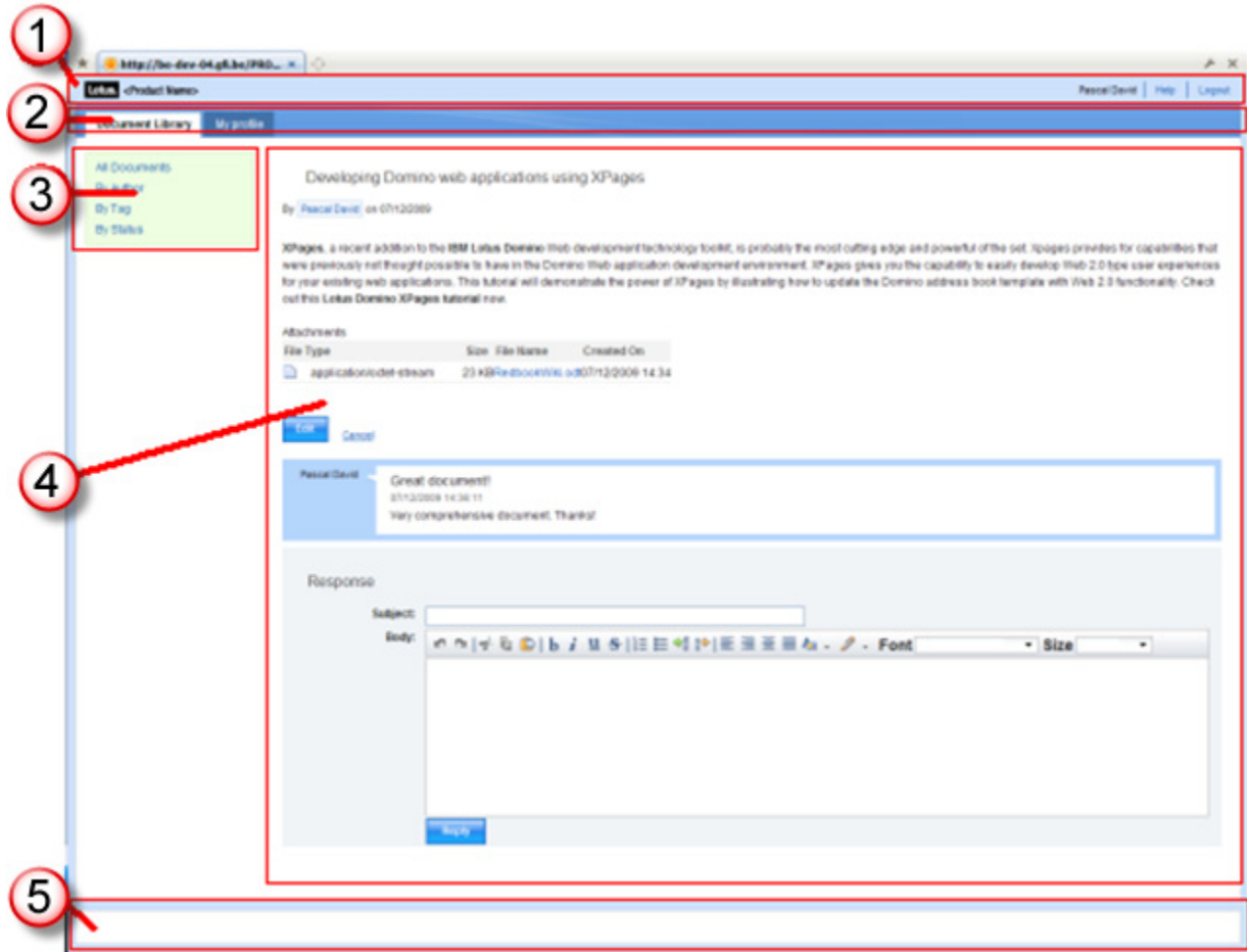
In this section, we will create a basic website structure with various website components (header, footer, navigation, etc...), and we will apply styles to our website.

We will provide our application with the look and feel of IBM's OneUI layout, as described in the [IBM Lotus User Interface Developer Documentation](#). This documentation describes version 2 of OneUI. Both version 1 and 2 are included in the standard Domino 8.5.1 installation. This will be explained further down in this section (Using themes).

Website layout setup

The idea of the website layout setup is that the layout is centrally stored in the database and is reused for each individual page. This is achieved by using custom controls and editable areas (facets).

In order to determine the different design components to be used, we need identify the different layout components on the final web application. The below pictures shows the parts of the application layout (click on the image to see a bigger image).



1. **Banner:** the banner sits at the very top of the page and contains the application logo on the left and utility links (help, login) on the top right. Optionally, it can contain links to other applications that make part of the website.
2. **Title bar:** just below the banner, the title bar contains navigational tabs. Additional items, like a search function can be included on the right.
3. **Left navigation:** this will be our main navigation within the Document Library: just like a Notes client navigator, it will contain a list of links to the different views within the document library.
4. **Content:** this is the part that will contain the application's main content, like form and view contents
5. **Footer:** the footer can contain additional links like About this application contents, etc...

In order to centrally store the layout, we will create one custom control per layout component and then reuse these custom control in each of our XPages.

Reuse custom controls versus reuse XPages

There are two possible strategies for centrally managing the layout of a website. You could either create a set of custom controls and reuse them in each individual XPage (as proposed above), or you could create a single XPage with all layout in it, and then add a kind of "computed" custom control to display the variable content. Although a computed custom control does not exist in a literal sense (unlike computed subforms), there are ways to dynamically change the custom control in an XPage. For example, by using an editable area with a computed facet name.

But even then, the first strategy remains far more preferable. Especially because using the second strategy (1 XPage, variable custom controls) would have as consequence that the same url is used for the entire application, since the url is defined by the XPage name. This would make linking individual pages impossible.

Create the layout

In this section, the different layout components/custom controls will be created. Once each component has been created, we will add proper styling and we will assemble all components together into one layout.

On each custom control, a panel container control will be placed. This will serve as a container for its individual elements, and will give overall control for styling later on.

The banner

Create a new custom control, and call it "layout_banner".

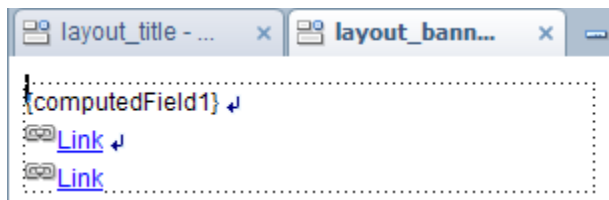
Drag a panel container control into the custom control.

In the panel, drag a computed field control. This computed field will serve to display the user name in the top left corner.

Behind the computed field add a newline (ENTER) and drag a link control on the next line.

Do the same once again: newline + drag a link.

The two links will be used as "Help" and "Login/Logout" links in our application.



Don't worry about the missing top left logo in the banner: we'll add that later via css.

Save and close this custom control for now.

The title bar

Create a new custom control "title_bar". The title bar is the tabbed bar just below the banner. Drag a panel container control into the custom control, and add two links into the panel. They will serve as tabs in our final application.



Save and close the custom control.

The left navigation

Repeat the above steps once again for a new custom control, called "layout_left". This time, you can add 4 links and give them label values:

All Documents
By Author
By Tag
By Status

As you notice, these links will be pointing to the different views within our application.



The footer

Same story here: create a custom control, "layout_footer" with two links, labelled "About the Document Library" and "Using the Document Library":



The content

In order to have a generic layout container for the page content, we will create a custom control with an editable area.

Create a custom control, "layout_content". Drag a panel container control in it, and in this panel, drag an editable area.

The editable area control can be found on the bottom of the list of Core Controls in the control palette. If you get tired of scrolling up and down, the list of controls in the control palette, right click in the palette and experiment with the options under the "Layout" and "Settings" context menu options. The "icons only" layout gives a complete overview of all controls without scrolling, and once you are getting used to working with controls, each icon will speak for itself. You will notice that all custom controls initially have the same icon, but you can change each custom control's icon individually and even group the custom controls in different categories. This is done via the custom control properties.

Save and close the layout_content custom control.

Bringing it all together

Now that we have defined each individual layout component, we will assemble our master layout into an XPage.

Open the XPage formDocument

Remove the existing custom control, ccFormDocument. The XPage is now empty.

From the controls palette, drag first the layout_banner custom control and then the layout_title custom control into the XPage.

Next, add a panel container control to the XPage: this will contain the left navigation and the content.

Since these appear next to each other, this container will help in controlling their layout.

In the panel, drag the layout_left and layout_content custom controls. layout_content will appear under layout_left, but that's fine by now.

You'll notice that the editable area of layout_content now has a "target zone" with a green circle. This is the area where you can add variable content within the editable area.

Drag custom control ccFormDocument into this zone, and the green circle will be replaced by the contents of this custom control (=our library document form).

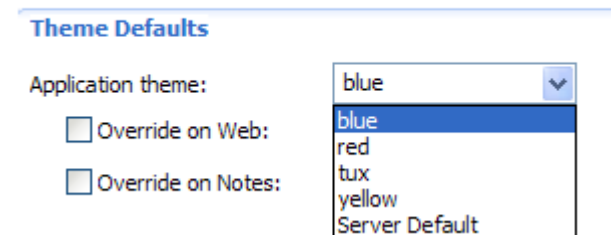
On the very bottom of the XPage, under the panel, drag layout_footer.

Save and preview the XPage. The result is far from fancy: just the same form as before with a bunch of links above and under it. This shouldn't be a big surprise: although we have build the foundations for our web site layout, no styling has been added so far. Time to get into the magic of themes!

Using themes

Introduced into Domino Designer 8.5, themes are powerful design elements that control overall layout of Domino web applications. With themes, you can give all your web applications a consisent look and feel, and include corporate branding. Themes are much more than styles. They can include various style sheets and can also override/ specify individual control's look and feel. For example, in a theme you can specify a particular style class for all submit buttons in your application. And last but not least: you can define multiple themes within an application, and then simple switch between themes via the application properties:

XPage Properties

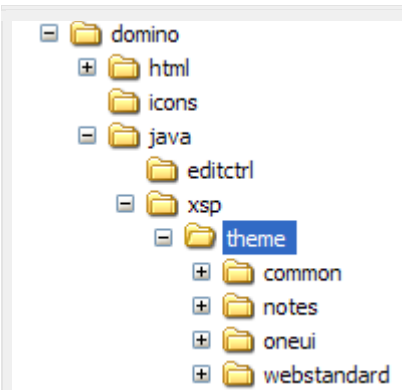


The screenshot shows the 'XPage Properties' dialog box with the 'Theme Defaults' tab selected. The 'Application theme:' dropdown menu is open, displaying a list of themes: 'blue', 'red', 'tux', 'yellow', and 'Server Default'. The 'blue' theme is currently selected. Below the dropdown, there are two checkboxes: 'Override on Web:' and 'Override on Notes:', both of which are currently unchecked.

As described in the beginning of this section, we will apply IBM's OneUI look and feel to the Document Library sample application.

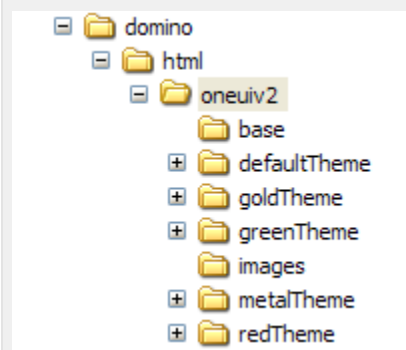
Built-in Domino themes

Domino 8.5.1 ships with three builtin themes: "webstandard", "notes" and "oneui". The source files (css and images) used these themes make part of the Notes client and Domino server installation. They can be found on the file system on the server and the client, in the notes data directory, under `domino\java\xsp\theme`:



There's a folder for each of the themes, plus a folder "common" with shared images.

The the "oneui" that is shipped with Domino is OneUI Version 1. In Domino 8.5.1, the theme OneUI V2 has been added for evaluation only and is not included as a standard theme. The source files are available in the data directory, under domino\html\oneuiv2:



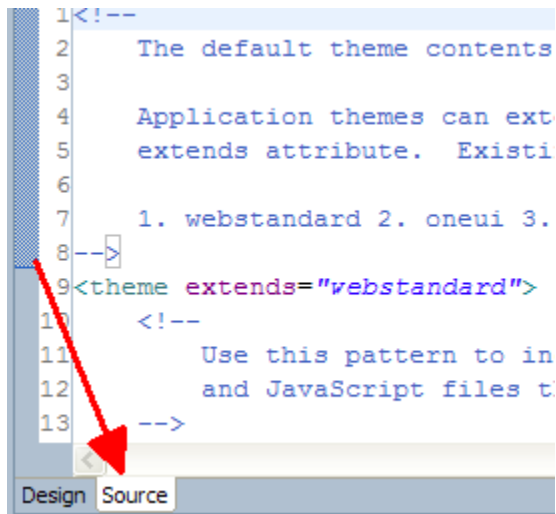
As you see in the above screenshot, the OneUI v2 theme comes in different "flavours": default, gold, green, metal and red. Each of the variations of the theme uses some common css files (folder "common") and images (folder "images").

The next tutorial step will demonstrate how to use either OneUI v1 or v2 into your application.

Creating a theme

Create a new theme resource, and call it "oneui".

You will notice that the new theme resource is opened on the Source tab of the editor, instead of the Design tab:



The source tab is the actual editing interface for themes, which are XML files.

A newly created theme will automatically contain a sample pattern to include stylesheets and one to define control properties. Also, a new theme inherits by default from "webstandard".

Replace "webstandard" by "oneui". This will automatically include the OneUI V1 theme definition. Save and close the theme.

In the Application Properties design element, go to the XPages tab and replace the application theme "Server Default" by "oneui".

Save and close the Application properties and open the application in the browser.

You will notice that the XPages now have a blue-white gradient background: this is the background defined in the OneUI V1 theme.

Also, the Submit button in the formDocument XPage now has a style applied.

Apart from that, the page layout is still dull. This is because we still need to link the different layout components to the corresponding style classes in the theme.

Defining a OneUI v2 theme

We want to apply version 2 of the OneUI theme, as this is the theme that is documented in the [IBM Lotus User Interface Developer Documentation](#).

OneUI V2 is not implemented in Domino 8.5.1 as a standard theme. In order to implement it, we need to include the source CSS files explicitly in a Theme design element.

Create a new theme, named "oneui2-core".

Leave extends="webstandard" unaltered.

Add the following code in the Source view of the theme editor:

```
<!-- One UI v2.0.1, Core -->
<resource>
  <content-type>text/css</content-type>
  <href>/.ibmxspres/domino/oneuiv2/base/core.css</href>
</resource>
<resource>
  <content-type>text/css</content-type>
  <href>/.ibmxspres/domino/oneuiv2/base/dojo.css</href>
</resource>
```

With Ctrl+Shift+F, you can nicely format the source code outline. It's also worthwhile exploring the menu options available in the context menu (right-click) under "Source". But be aware that there is one drawback/issue when using this format function in Domino 8.5.1: the function places </href> end tags on a new line rather than on the same line. As a consequence, css files are not loaded in the browser because at the end of the file name, there are unknown characters:

WRONG:

Theme:

```
<resource>
  <content-type>text/css</content-type>
  <href>/.ibmxspres/domino/oneuiv2/base/core.css
</href>
```

Browser:

```
<link rel="stylesheet" type="text/css" href="/oneuiv2/base/core.css%0A%09%09">
```

RIGHT:

Theme:

```
<resource>
  <content-type>text/css</content-type>
  <href>/.ibmxspres/domino/oneuiv2/base/core.css</href>
</resource>
```

Browser:

```
<link rel="stylesheet" type="text/css" href="/oneuiv2/base/core.css">
```

Save and close the theme.

The above code includes the core OneUI v2 theme files into the theme design element. These css files are shared amongst the different flavours of the OneUI v2 theme.

Now, we will create the actual theme to use in the application:

Create a new theme, and call it "oneuiv2-default".

Replace "webstandards" by "oneuiv2-core" as the parent theme.

Include two css files into the theme:

```
<resource>
  <content-type>text/css</content-type>

  <href>/.ibmxspres/domino/oneuiv2/defaultTheme/defaultTheme.css</href>
</resource>
<resource>
  <content-type>text/css</content-type>

  <href>/.ibmxspres/domino/oneuiv2/defaultTheme/dojoTheme.css</href>
</resource>
```

Save and close the theme.

If you now apply the theme "oneuiv2-default" in the Application Properties and preview the application in the browser, you will notice that the little bit of fanciness which had been added with OneUI v1 has disappeared again. This is because of differences in the implementation between v1 and v2. Check the html source code in the browser, and you will notice that the theme css files are included.

Referring to resource files of built-in Domino themes

You might have noticed something odd in the above explanation: the oneuiv2 theme files are in the directory <notesdata>\domino\html\oneuiv2, but the <href> resource properties point to a different location in the theme definition.

Domino actually performs a mapping of the following locations:

/.ibmxspres/global/theme/oneui/	maps to	<domino data directory>\domino\java\xsp\theme\oneui
/.ibmxspres/domino/	maps to	<domino data directory>\domino\html\

```
/.ibmxspres/dojo/
```

```
maps to
```

```
<domino data directory>\domino\js\dojo-1.3.2\
```

You can also use these mappings to refer to theme images withing your XPages:

```
<xp:image url="/.ibmxspres/domino/oneuiv2/images/loading.gif" />
```

Adding style classes to the layout components

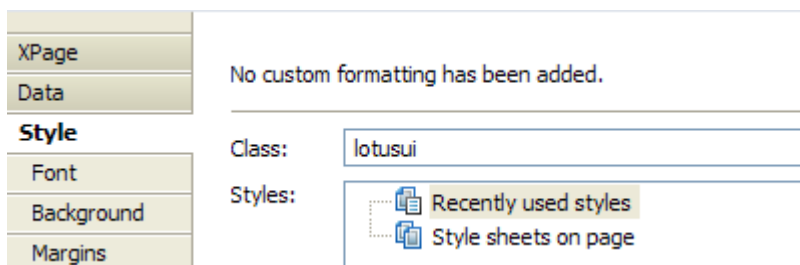
The key of applying OneUI v2 to an XPage application is assigning the right style class names to the right layout components.

OneUI v2 is all about classes. The Components section in the [IBM Lotus User Interface Developer Documentation](#) describes which classes to assign for each component. This is what were are going to do now: assign style classes to each of the layout components. We will also further finetune each component and give them the necessary containers (<div> tags) to make their look and feel correspond with the OneUI layout.

Assign a body class

Before diving into the individual layout components, we will add a style class to the body tag of our XPages:

Open the XPage formDocument and in the XPage properties view, go to the "Style" tab and assign the class "lotusui":



This class is a defined in OneUI v2 and controls overall page layout. Do the same for the other XPage, viewAllDocuments.

Preview the XPages in the browser. They now have a blue background.

The banner

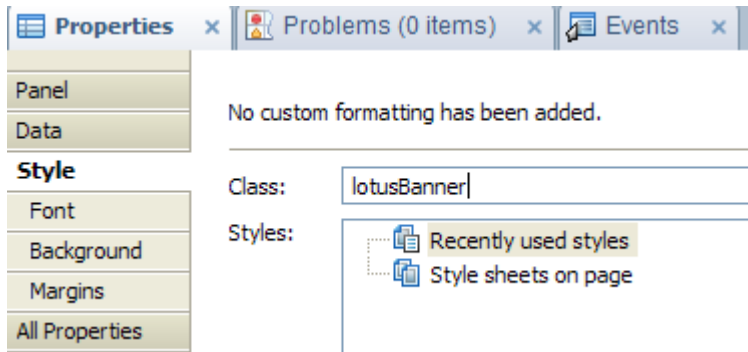
Open the layout_banner custom control and go to the Source view. The source code looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel>
    <xp:text escape="true" id="computedField1"></xp:text>
    <xp:br></xp:br>
    <xp:link escape="true" text="Link" id="link1"></xp:link>
    <xp:br></xp:br>
    <xp:link escape="true" text="Link" id="link2"></xp:link>
  </xp:panel>
</xp:view>
```

If we have a look at sample HTML in the OneUI specifications for a banner in the [IBM Lotus User Interface Developer Documentation](#), we see this code:

```
<div class="lotusBanner" role="banner">
  <div class="lotusRightCorner">
    <div class="lotusInner">
      <a href="#lotusMainContent" accesskey="S" class="lotusAccess">
        
      </a>
      <div class="lotusLogo">
        <span class="lotusAltText">Lotus &lt;Product
Name&gt;</span>
      </div>
      <ul class="lotusInlinelist lotusUtility">
        <li class="lotusFirst"><span class="lotusUser">Pat
Shani</span></li>
        <li><a href="javascript:;">Help</a></li><li><a
href="javascript:;">Logout</a></li>
      </ul>
      <ul class="lotusInlinelist lotusLinks" role="navigation">
        <li class="lotusFirst lotusSelected"><a
href="javascript:;"><strong>Home</strong></a></li>
        <li><a href="javascript:;">Application1</a></li>
        <li><a href="javascript:;">Application2</a></li>
        <li><a href="javascript:;">Application3</a></li>
      </ul>
    </div>
  </div>
</div><!--end lotusBanner-->
```

In the sample HTML, there are three div containers, each with their own class: lotusBanner, lotusRightCorner, lotusInner. We could now create two additional panel container controls in the custom control (there's already one), and assign the appropriate style classes to them:



This will work, but it will create unnecessary overhead. Instead of `<xp:panel>` tags, we could use simple `<div>` tags, as all we want to do with these containers is assign style classes to them.

Container tags in XPages

The XPages editing GUI only provides one simple container control for adding components: the panel. This translates to `<xp:panel>` in the source. A panel is useful when for example event handlers need to be added to the container control. In case the container only serves as a means to assign style classes, a simple `<div>` tag is sufficient. A third container tag, `<xp:div>`, is useful when for example a themeID needs to be assigned to the container. This will be demonstrated later on. The Designer GUI does not provide visual container controls for `<xp:div>` and `<div>`, but you can simply add them in the source code of the XPage or custom control.

Taking the above into account, we will apply the right styling to the banner by simply copying and pasting the sample HTML code into the source code of the custom control (instead of `<xp:panel>`), and by then replacing the hard coded user name in the sample by the computed field. The links "Help" and "Logout" can be copied from the sample and replaced later on. Furthermore, we remove some unneeded features, like application links (will not be used within our application).

The resulting custom control looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <div class="lotusBanner" role="banner">
    <div class="lotusRightCorner">
      <div class="lotusInner">
        <div class="lotusLogo">
          <span class="lotusAltText">Document
Library</span>
        </div>
        <ul class="lotusInlinelist lotusUtility">
          <li class="lotusFirst">
```



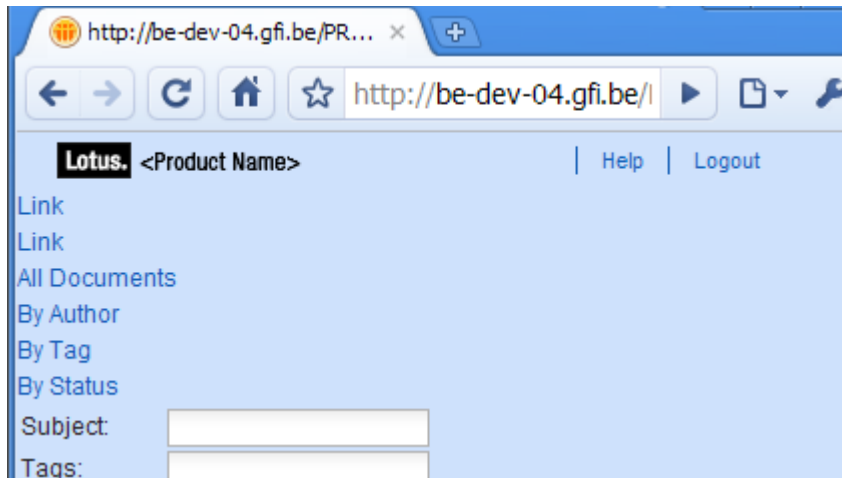
```

                                <span class="lotusUser">
                                    <xp:text escape="true"
id="computedField2"></xp:text>
                                </span>
                                </li>
                                <li>
                                    <a href="javascript:;">Help</a>
                                </li>
                                <li>
                                    <a href="javascript:;">Logout</a>
                                </li>
                            </ul>
                        </div>
                    </div>
                </div><!--end lotusBanner-->
</xp:view>

```

As you notice, the three links in the banner have been placed in an unordered list. Unordered lists can currently not be added via Designer GUI controls: you need to add them via the Source view.

Save and close the custom control and preview the XPage formDocument in the browser. The banner is displayed with a default logo and Help and Logout links:



OneUI v1 and v2 differences

A major difference in approach between OneUI v1 and v2 is that v2 uses style classes to apply css styling to components, while v1 uses the component ID (=the name of the component). If you want your application to be able to work with both themes, you can add the same style name as component name and as class name:

```
<div id="lotusRightCorner" class="lotusRightCorner">
```

Computing the username

Open the layout_banner custom control again and select the computed field.

In the computed field properties, change the name to "cfUserName".

On the Value tab, select "JavaScript" as binding option and write the following script as value:

```
@Name ( ' [CN] ', @UserName ( 0 ) )
```

Save the custom control and preview formDocument.xsp to see the result.

Be careful with punctuation and casing in the above formula: this is JavaScript, so it's different than @Formula language. Refer to the online help for an overview of available parameters with the JavaScript @Functions.

The title bar

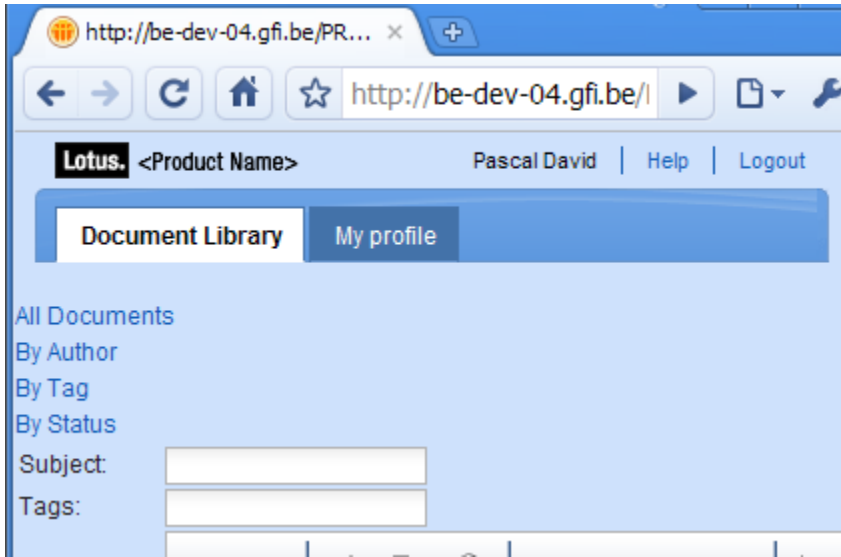
We'll perform a similar make-over to this custom control:

Review the titlebar sample code in the OneUI documentation and apply it to the custom control source code.

The result could be this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <div class="lotusTitleBar">
    <div class="lotusRightCorner">
      <div class="lotusInner">
        <ul class="lotusTabs" role="navigation">
          <!--put class="lotusSelected" on the li element
of the selected tab-->
            <li class="lotusSelected">
              <div><a
href="javascript:;"><strong>Document Library</strong></a></div>
            </li>
            <li><div><a href="javascript:;">My
profile</a></div></li>
          </ul>
        </div>
      </div>
    </div>
  </xp:view>
```

The result in the browser:



In the above code, the original link controls have been replaced by simple `<a href>` html tags. Feel free to experiment with link controls.

Also, if you prefer using panel container controls via the Designer GUI, this is perfectly possible. Just make sure you respect the layer structure and style classes.

The left navigation

Do the same for the layout_left custom control:

- add `<div>` container controls with the appropriate styles
- place the left navigation links in an unordered list

This is how the source code should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <div class="lotusColLeft">
    <div class="lotusMenu">
      <div class="lotusBottomCorner">
        <div class="lotusInner">
          <ul>
            <li>
              <xp:link escape="true" text="All
Documents" id="link1"></xp:link>
            </li>
            <li>
              <xp:link escape="true" text="By
Author" id="link2"></xp:link>
            </li>
            <li>
```


Optimize the layout framework

So far, the XPage formDocument has been given a (partial) OneUI v2 look and feel by adding the layout components in the form of custom controls.

We could now add these custom controls to the XPage viewAllDocuments and to each consecutive XPage to be built within the Document Library application. This is a common approach.

Ideally, the entire website layout should be stored into one single custom control, which would make it easier to apply the layout framework to each individual XPage, and also to manage future layout framework changes.

This can be achieved by creating a "master layout custom control" which contains all individual layout custom controls (yes, they can be nested), and which contains an editable area for the content.

Create a main layout custom control

Create a new custom control, "layout_main".

Via the control palette, drag the custom controls layout_banner and layout_title into this one.

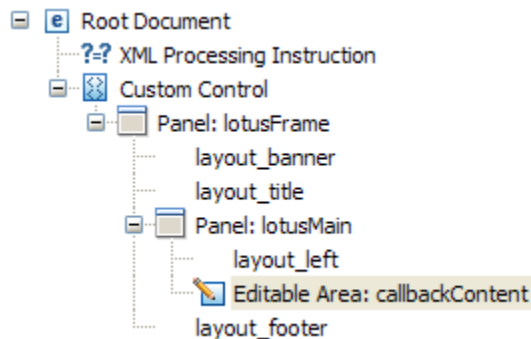
Under these controls, drag a panel container control and assign the style class "lotusMain" to it: this will be the container for the left navigation and the content. The style class will apply the appropriate OneUI styling.

In this panel, drag the custom control layout_left.

Under layout_left, but still within the panel, add an editable area, and name it "callbackContent" and give it facet name "facetContent".

Under the panel, add layout_footer.

The outline view of layout_main should now look like this:



In order to have the page content properly aligned, we need to place it within a container with the OneUI style class lotusContent:

Select the editable area facetContent, either via the Design view or via the Outline view.

Now open the Source view: the editable area is selected.

Write a div tag around this facet, with class "lotusContent":

```
1<?xml version="1.0" encoding="UTF-8"?>
2<xp:view xmlns:xp="http://www.ibm.com/xsp/core" xmlns:xc="http://www.ibm.com/xsp/custom">
3  <xc:layout_banner></xc:layout_banner>
4  <xc:layout_title></xc:layout_title>
5  <xp:panel styleClass="lotusMain">
6    <xc:layout_left></xc:layout_left>
7    <div class="lotusContent">
8      <xp:callback facetName="facetContent" id="callbackContent"></xp:callback>
9    </div>
10  </xp:panel>
11  <xc:layout_footer></xc:layout_footer>
12</xp:view>
13
```

As said before, you could also have added a panel container control via the Designer GUI instead of a <div> to achieve the same result, but as we only need a container for a style class, the div is preferable.

Save and close the custom control.

Adapt the XPages

Now we can upthe the two existing XPages with this new layout framework:

Open the XPage formDocument and remove all custom controls.

Drag the custom control layout_main into the XPage.

Then drag the custom control ccFormDocument on the "drop zone" of the editable area layout_framework (=on the green circle).

That's it! The XPage is ready. Save and preview it.

The content container panel now has a white background, thanks to the lotusMain style class.

Repeat the same procedure for the XPage viewAllDocuments:

Remove the existing contents and drag layout_main into the XPage. Then drag ccViewAllDocs into the editable area of layout_main.

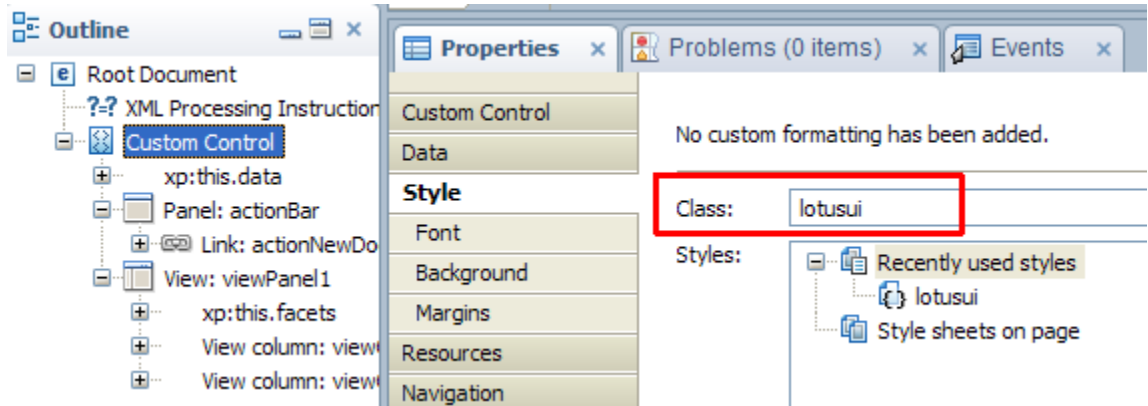
Now, you can apply the same trick to any other to be built XPage within the application: simply create a custom control for the contents and make an XPage, containing layout_main and the content custom control.

Add styling to the view All Documents

Add some additional styling to this view by adding the appropriate style classes:

Open ccViewAllDocs

Assign the style class "lotusui" to the custom control:



Add the styles "lotusActionBar lotusBtnContainer" to the actionBar panel and wrap the link in the actionBar in tags with the attributes class="lotusBtn lotusBtnAction lotusLeft"role="button". The action bar source code should look like this:

```
<xp:panel id="actionBar" styleClass="lotusActionBar lotusBtnContainer">
  <span class="lotusBtn lotusBtnAction lotusLeft" role="button">
    <xp:link escape="true" text="New document" id="actionNewDoc">
      <xp:eventHandler event="onclick" submit="true"
        refreshMode="complete">
        <xp:this.action>
          <xp:actionGroup>
            <xp:openPage name="/formDocument.xsp"
              target="newDocument">
            </xp:openPage>
          </xp:actionGroup>
        </xp:this.action>
      </xp:eventHandler>
    </xp:link>
  </span>
</xp:panel>
```

Select the view in the Outline view and assign the style "lotusTable" to it.

In the Source view, wrap the view (= <xp:viewPanel>) into a <div> tag with a style attribute set to float left:

```
<div style="float:left">
  <xp:viewPanel value="#{dominoView}" id="viewPanel1"
    viewStyle="width:100%" viewStyleClass="lotusTable">
  </xp:viewPanel>
</div>
```

```
...  
(VIEW CONTENTS)  
    </xp:viewPanel>  
</div>
```

This is a tweak to make the view appear under the action bar in other browsers than Internet Explorer. There might be better ways to get this right.

Save the custom control. When previewing the view, you'll notice that the "New document" link has been transformed into a beautiful button.

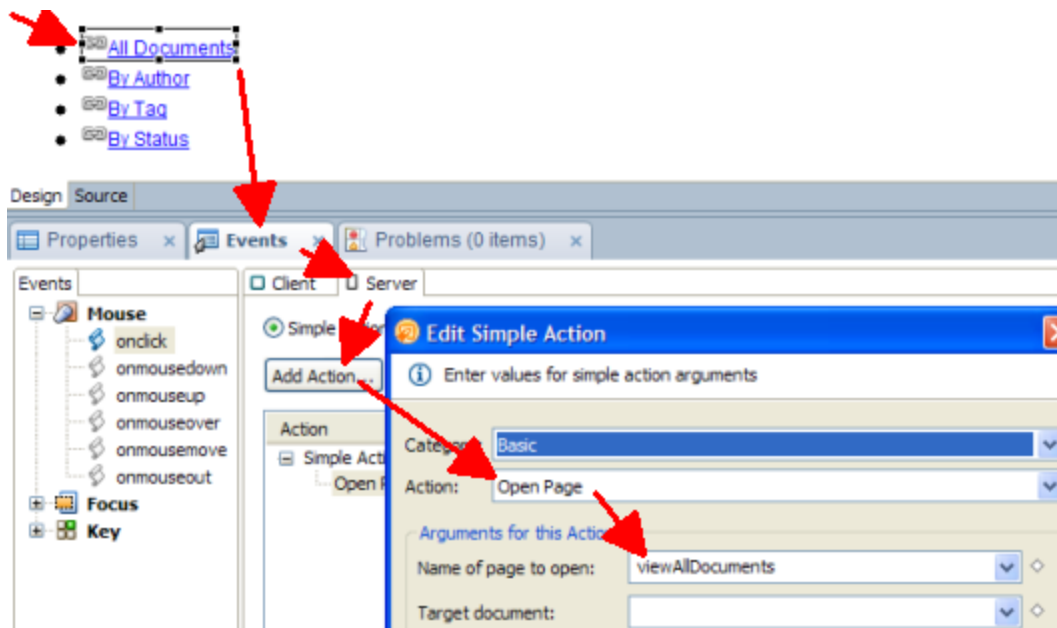
Feel free to explore the style classes in the OneUI documentation and to further enhance the view.

Define a left navigation link

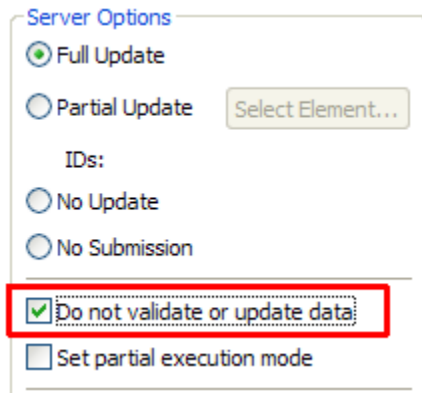
Just to make it more comfortable to work with our web application, we make the link "All Documents" work and make it open the XPage viewAllDocuments.

Open the custom control layout_left and select the first link, "All Documents".

In the Events view, add an action to the onClick event (=selected by default): server side (=default): select a basic action: open page: viewAllDocuments:



Under the server options, select "Do not validate or update data":



Server Options

☒ Full Update

☐ Partial Update

IDs:

☐ No Update

☐ No Submission

☒ Do not validate or update data

☐ Set partial execution mode

This option needs to be set, because clicking on the link "All Documents" should not perform any data validations. This will become more clear in the next section.

Save the custom control.

Since the layout is maintained centrally, the left navigation will now be updated in all XPages of the application.

Summary for Section – Sample 1 - Create the website layout

In this section, we have given our very basic web application a OneUI look and feel. We have created a theme and a reusable layout framework, and we have applied the OneUI style classes to the various layout components.

In the next section, we will focus on the Library Document form functionality.

Form Design - XPage enabling an existing Notes client application

This section describes how we will leverage the basic form created in the previous section to a fully featured Document Library input form. Form styling will be applied, action buttons will be added and field validations and type-ahead will be implemented. And finally, a review section including reviewer names and review options will be added.

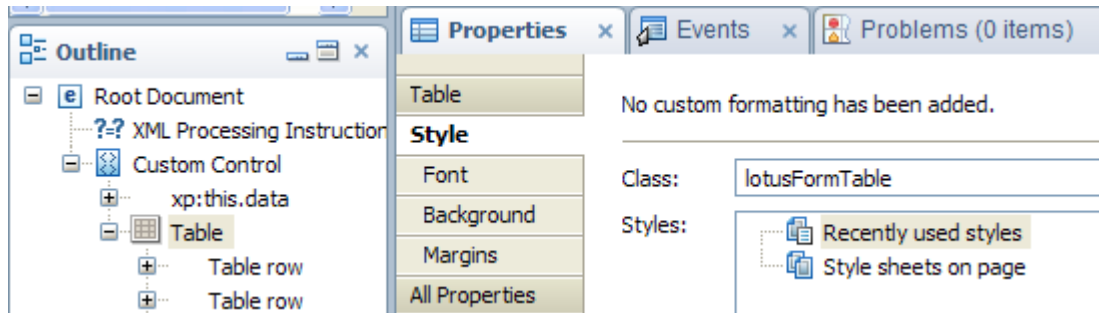
Form styling

Before starting with including form functionality, we will add some OneUI styling to the existing form components.

Field layout

Open ccFormDocument.

Select the main form table in the Outline view and give it the style class "lotusFormTable":



Select the Subject field and set its width to 400px by clicking on the field and then entering the width in the Properties view, on the Edit Box tab (select Units first and then enter a value).

Assign the style class "lotusText" to the Subject (Style tab in the properties).

Do the same for the WebCategories field: add a width 400px and the style class "lotusText".

Give the Rich Text field a height of 200px and also assign the class "lotusText".

Now click in the table cell that contains the field label "Subject". Make sure you have the table cell selected and not the Subject label itself. Give the table cell a width of 120px, and assign it the style class "lotusFormLabel".

Assign the same class to the other cells that contain field labels.

If you now preview formDocument.xsp, you will notice that the layout of the field labels hasn't changed. This is because most of the OneUI styling definition for formlabels works only in conjunction with the lotusForm style class.

Extract from the OneUI v2 forms.css:

```
.lotusForm td.lotusFormLabel {
  vertical-align:top;
  text-align:right;
  padding-right:10px;
  padding-top:1px
}
```

The lotusForm class needs to be assigned to the <form> tag in the html source code. XPages don't provide direct access to this tag: there is no "Form" element of which you can set the style. This is where the use of themes proves its usefulness again: with themes, you can assign style classes (and other properties) to html tags.

Add a control property definition to the theme

Open the theme resource called "oneui2-core" and add a "control" section to the source code:

```

<control>
  <name>Form</name>
  <property>
    <name>styleClass</name>
    <value>lotusForm</value>
  </property>
</control>

```

Save and close the theme and preview formDocument.xsp in the browser. The field labels are now right aligned and in bold. In html source code in the browser you will notice that the form tag now has the class "lotusForm" assigned.

Form title

We will add a title to the Library Document form:

In ccFormDocument, add a table row in the beginning of the form table.

Merge the two table cells (select both cells and right-click: merge)

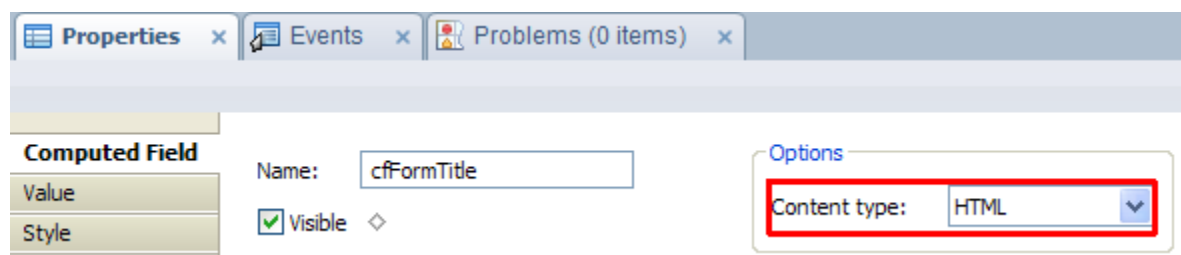
Go to the custom control Source view and in the merged table cell, add the text "Library Document" between <h2> tags:

```

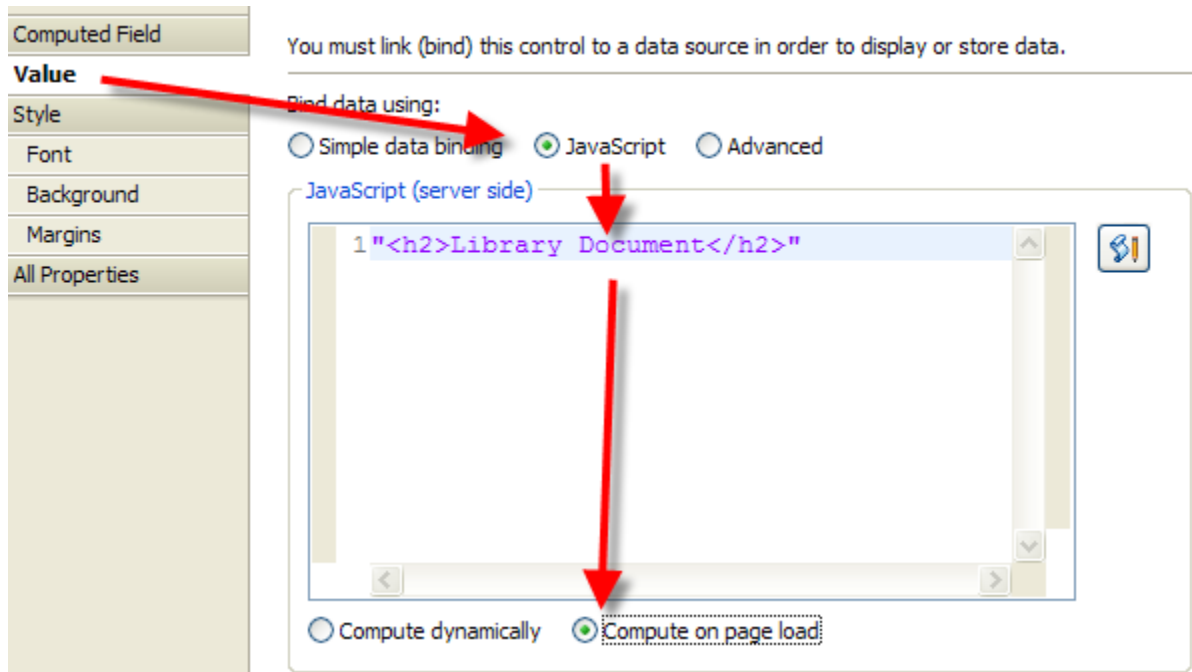
<xp:table styleClass="lotusFormTable">
  <xp:tr>
    <xp:td colspan="2"><h2>Library Document</h2></xp:td>
  </xp:tr>

```

Alternatively, if you prefer to use the Designer GUI view rather than the source code view, you could achieve the same result by dragging a Computed Field into the table cell, set its content type to "HTML":



Then, give it the value "<h2>Library Document</h2>". Make sure to select the option "Compute on page load" rather than "Compute dynamically". (it doesn't need to recompute during form edits, as it is a fixed text):



Submit button

The submit button on the bottom of the form still could use some styling.

In ccFormDocument, merge the two table cells on the bottom row of the form table (=containing the submit button).

Assign the style class "lotusFormFooter" to the merged table cell.

Select the button and give it the class "lotusFormButton".

Also, change its name to buttonSave and its label to "Save" (to avoid confusion with the "submit for review" action, which will be added later on when building the workflow).

Save and preview the result. The form looks more satisfying now.

Action buttons

Time to add some more action buttons to the form, like Edit and Cancel actions.

Edit button

In ccFormDocument, add a button next to the Submit button and give it the following properties:

Name: buttonEdit
Label: Edit
Button type: Button

Assign the style lotusFormButton

In the Events view, add a simple action: Change Document Mode: Edit Mode

Save the custom control and open a open the page viewAllDocuments.xsp in the browser. Click on a document to open it in read mode. When you click on the Edit button, you will notice the page refreshes, but nothing happens. The document remains in read mode.

Click once again on Edit. The second click opens the document in edit mode.

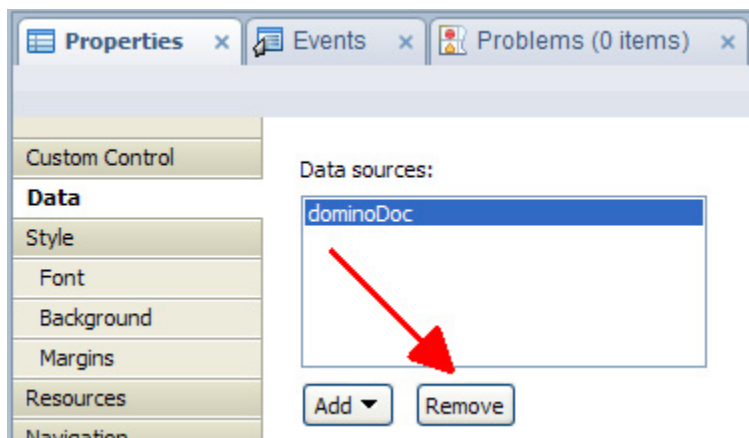
This is not the intended behaviour. This is actually a know issue in Domino 8.5.1. It is linked to the place where the data source is defined.

Edit buttons and data source definition in Domino 8.5.1

In Domino 8.5.1, the data source definition needs to be done at XPage level instead of at custom control level, in order to make an Edit button change the document to edit mode with a single click. This is a known issue which will be addressed in a future release. It is more logic to define the data source in the custom control, rather than in the XPage: and XPage can contain several custom controls, which each point to a different data source. For example, when displaying a main document and a response document on the same XPage. Because of the above constraint, you might need to be creative in order to handle this kind of situations. Later on in this tutorial, the combination of two data sources in one XPage will be discussed.

In order to solve the above issue, we will move the data binding to the XPage:

In ccFormDocument, go to the Data tab in the custom control properties view. Remove the data source "dominoDoc":



The data source list is now empty.

You might notice that the data binding at individual control level is still present: select the Subject field and click on the Data tab in its properties: the data source is still set to dominoDoc.

Data source definition and data binding

The above example demonstrates that removing the data source definition does not remove the data binding. This is because the data source does not necessarily need to be defined in the same component as the one where the data binding is used. For example, you can define a data source at XPage level, and then perform data binding to that source in various custom controls that are used in the XPage.

In any case, a data source needs to be defined **somewhere**, in order bind data to it.

Defining the data source in the same XPage/custom control as where you intend to do the binding, makes it easier to do the data binding: you can drag fields from the data palette, select the data source from a dropdown list. A possible approach to facilitate development is to first define a data source in the custom control where you want to do data binding, then use this data source for the actual binding, and then remove the data source (assuming the data source is defined elsewhere).

Save and close ccFormDocument.

Open the XPage formDocument, and in the XPage properties, go to the Data tab.

Add a data source, exactly in the same way as we have initially done in the custom control: the name must be the same: dominoDoc. And its definition needs to be the same as well: Form: Document;

Default action: Create document.

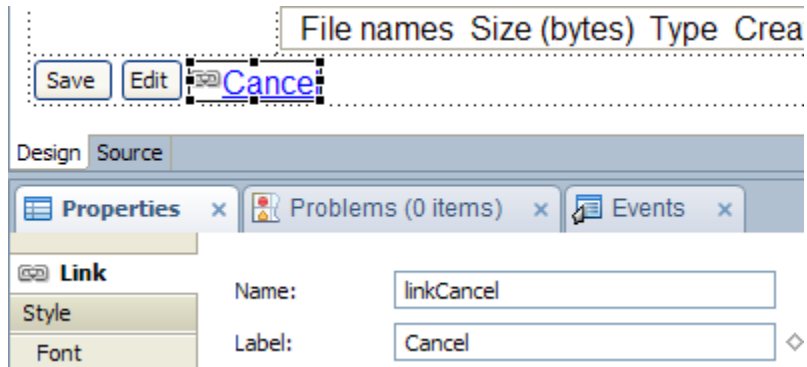
Save and close the XPage.

Open the view viewAllDocuments again in the browser and open a document in read mode. The Edit button will now work as expected: it will switch to edit mode with a single click on the button.

Cancel link

In ccFormDocument, drag a core control of type "Link" to the right hand side of the Edit button.

Give it the name "linkCancel" and the label "Cancel":



On the Style tab, add the class "lotusAction".

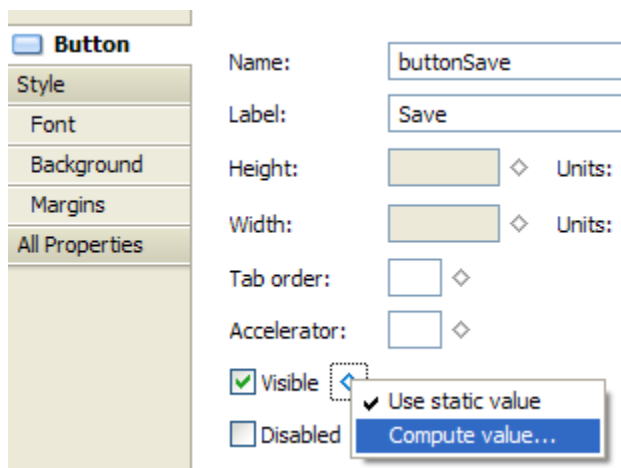
Under Events, add a simple action to the onclick event: Open Page: Previous Page

Save the custom control and open a library document in the browser (either in edit or on read mode): the cancel returns to the view from where the document has been opened.

Define action visibility

Next thing to do is make the action buttons only visible when needed (="hide-when's"): the Edit button should only be available in read mode, the Save button should only be displayed in edit mode.

In ccFormDocument, select the Save button and in the properties, on the Button tab, click on the diamond next to Visible:



Select "Compute value..."

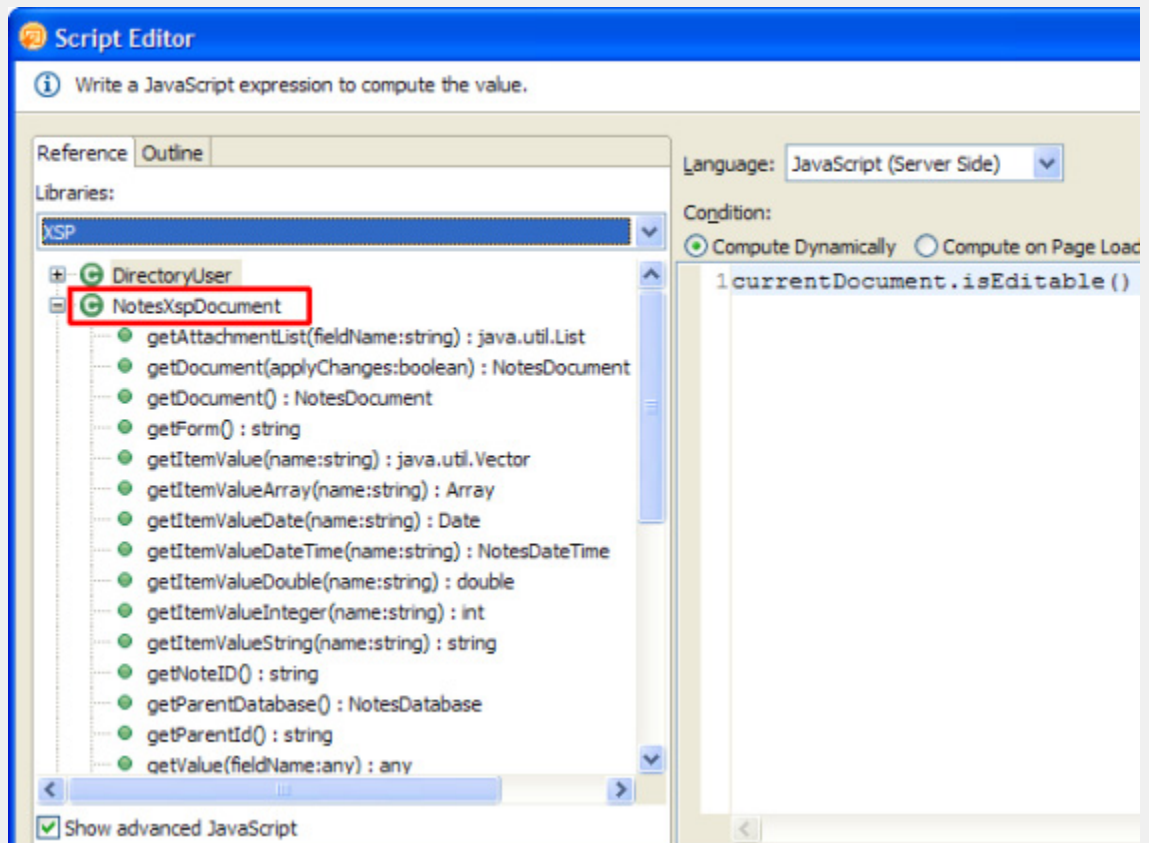
In the script editor window, write following the Server Side JavaScript:

```
currentDocument.isEditable()
```

Accessing the current document context

In the above script, **currentDocument** represents the document associated with the XPage context in which the server script is running. The variable `currentDocument` is of type `NotesXspDocument`, which is in some way comparable to the `NotesUiDocument` class in the Notes client.

In the **Script Editor** window, select the **XSP Library** on the **Reference** tab on the left and expand the **NotesXspDocument** class to view the available methods:



Alternatively, instead of using `currentDocument` in the above example, we could have used this:

```
dominoDoc.isEditable()
```

Where **dominoDoc** is the name of the data source linked to the XPage. If the XPage contains multiple data source, using the data source name lets you access a specific data source, while `currentDocument` represents the closest document in the current context. You might also want to use `currentDocument` to write reusable code, which is independent of the data source name.

The Edit action button ccFormDocument should only be visible in read mode:

Add a computed value for the visible property of the Edit button:

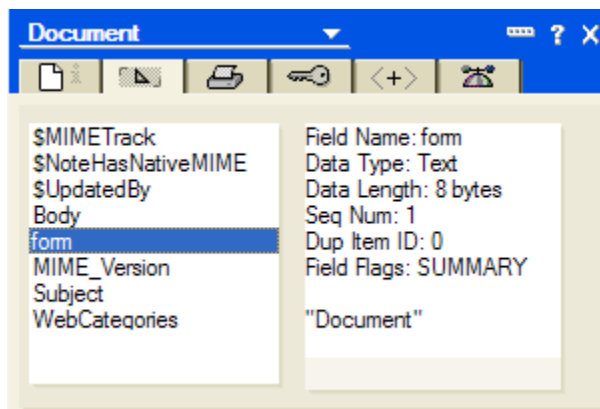
```
!currentDocument.isEditable()
```

Above, we have used computed values for visibility instead of a static value. In XPages, almost every property or value of XPage components can be computed: there is a diamond next to most of the property fields. This makes XPages very powerful and flexible.

Save ccFormDocument and check the result in the browser. The Edit and Save buttons now display in the right mode.

ComputeWithForm

In the Notes client, have a look at the document properties of a library document that was created via the web browser:



The document only contains some system fields, a form field and the 3 data fields that have been defined in ccFormDocument.

All the remaining fields defined in the notes Document form are absent.

There's no need to define them in the XPage though: there is a possibility to compute the data document with the Notes form:

Open the XPage formDocument and go to the All Properties tab in the properties view.

Under data\data\dominoDocument[0], set the property computeWithForm to "onsave":

XPage	Property	Value
Data	+	basics
Style	-	data
Font		ad
Background	-	data
Margins		dominoDocument [0]
Resources		action
Navigation		allowDeletedDocs
All Properties		computeDocument
		computeWithForm
		concurrencyMode
		databaseName

Save the XPage and create a new document on the web.

The document will now have all notesitems as defined in the Notes form, as if it was created with this form

Form calculations have been done as if the document was created with the Notes form. In other words, all data manipulation, but no front-end interactions. For example, input translations will execute, but input validations won't .

The field "Categories" will now have the same value as WebCategories.

Validations and type ahead

In this section, the Subject field will be made mandatory and field input will be enhanced with type-ahead.

Client side validation

In ccFormDocument, select the field Subject and in the Properties view, go to the Validation tab. Mark the field as required and enter a required field message.

Enter a minimum number of characters and add a validation error message:

The screenshot shows the Domino Properties view for the Subject field. The left sidebar has tabs for Edit Box, Data, Validation, Type Ahead, Style, Font, Background, Margins, and All Properties. The main area is divided into two sections: 'Required field' and 'Validate String Length'.

Required field: A checkbox is checked. Below it, the 'Required field error message:' is set to 'Please enter a subject'.

Validate String Length: This section has a title bar 'Validate String Length'. It contains:

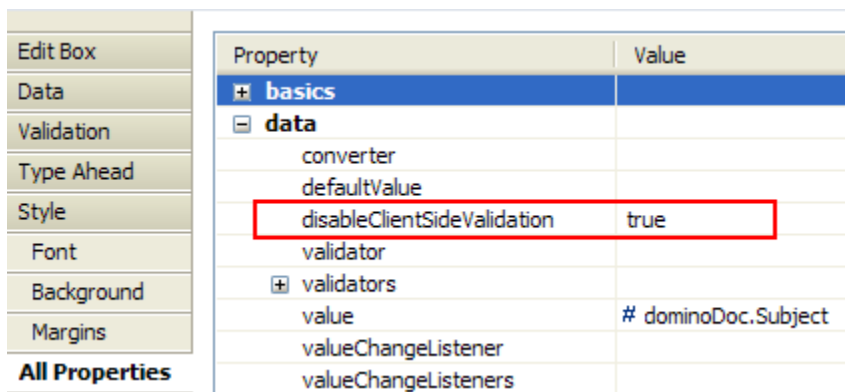
- 'Enter number of characters:' with a sub-label 'Minimum:' set to 3.
- A sub-label 'Maximum:' with an empty input field.
- 'Validation error message:' set to 'Please make sure the subject is at least 3 characters long'.

Test the result in the web browser: the validation messages appear in the form of client side JavaScript popup windows.

Server side validation

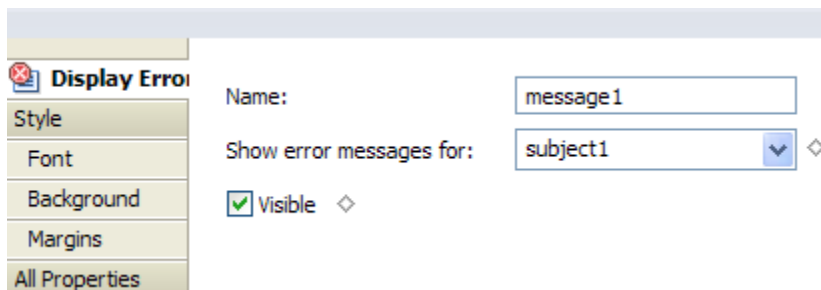
By default, valitions defined on the Validations tab of the fields are executed client side. They can also be set to execute as server side:

In ccFormDocument, go to the Properties of the Subject field and open the All Properties tab. Under "data", set the value of disableClientSideValidation to true:



If you would now save the custom control and preview the XPage, you will notice the form cannot be submitted without subject of at least 3 characters, but no error message appears. In order to see server side error messages, we need to add a control to display the messages:

Drag a core control of type "Error message" next to the Subject field in ccFormDocument. In the error message control properties, set "Show error messages for: subject1":

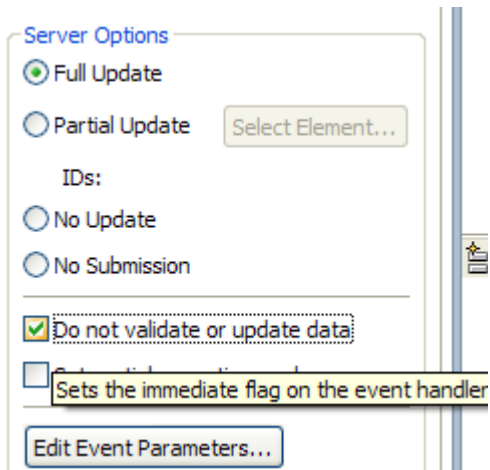


The validation error messages will now be displayed next to the Subject field.

Disable validation on links

If we now create a new document without subject and press cancel, the subject field validations will be executed as well, and the form will not be closed. Reason: by default, validation is performed for any link or button. This needs to be disabled explicitly:

Select the Cancel link in ccFormDocument and in the events view, and set the server option "Do not validate or update data":



The same needs to be done on any link in the XPage, including the ones in the layout framework (banner, title bar, left navigation, footer).

Type ahead

We will now provide the Tag field with type-ahead: when an author starts typing a tag name, a list of matching tag names that were used previously will appear:

Open the Type Ahead tab of the properties of the WebCategories field in ccFormDocument.

Enable Type Ahead and select Partial Mode.

Click on the diamond next to the suggestions field and add a computed value:

```
return @DbColumn("", "ByCategory", 1);
```

Uncheck the option "Case-sensitive".

Place a comma in the fields "Suggestion separator" and "Client separator" and enter 1 for "Minimum characters":

Edit Box	<input checked="" type="checkbox"/> Enable Type Ahead	
Data	Mode:	Partial
Validation	Suggestions:	{Computed}
Type Ahead		
Style		
Font		
Background	Suggestions separator:	,
Margins	Minimum characters:	1
All Properties	Client separators:	,
	<input type="checkbox"/> Case-sensitive	

Test the XPage formDocument in the browser: when typing a value in the Tag field, previously entered tag names appear. Also, multiple tags can be entered, and type ahead works on multiple values.

Reviewers

The Notes client Document Library application has a workflow: authors of documents can select one or more reviewers and then submit the document for review, either in a parallel or a serial approval cycle.

In this section, the interface for adding reviewers will be added to the form. In the Notes client, reviewers stored in a multi-value names field, and they are selected via an Address dialog. XPages currently don't have a built-in address dialog component. On the web, you can find some custom build address dialog solutions for XPages, or you can build your own.

In this sample application, we'll provide a simple alternative. Authors will be able to type a name in a text field with type ahead. They can then add the name to a list of reviewers:

*Reviewers:	<input type="text" value="Sara Owner/BE/USR/GFI-DEV"/>	<input type="button" value="Add"/>
	<input type="text" value="Ed Author/BE/USR/GFI-DEV"/> <input type="text" value="Sara Manager/BE/USR/GFI-DEV"/>	<input type="button" value="Clear"/>

In order to ensure data integrity, validations will be added to the input field: only names that exist in the Address Book can be added to the list, and the author cannot add himself as reviewer. This corresponds with the data validations in the Notes client version of the document library.

Preparing the form table

In ccFormDocument, add two table rows above the bottom table row (=with the action buttons).

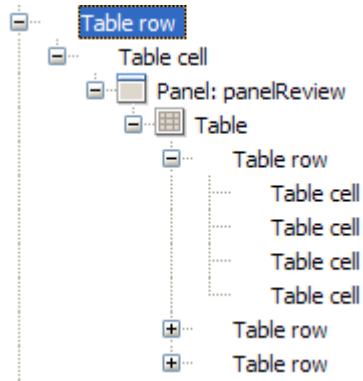
In the first of the two new rows, merge the two cells.

In this merged cell, drag a panel container control and name it "panelReview",

Drag a table container control into the panel: 3 rows x 4 columns, 100% width.

Set the width of the cell in this table to 120px.

The outline of the third last table row in the main table now looks like this:



Reviewers fields

We will create 3 fields:

- A text input field to add new names
- A listbox field to display the list of already added names
- A hidden multiline text field to store that actual name values in the back-end document

In the newly added table, add two Label core controls in the first column:

on row 2: label "Add reviewer"

on row 3: label "Reviewers:"

In column 2, add an Edit Box control in row 2 with the name "**AddReviewer**" and with a width of 400px.

Make it visible only in edit mode:

```
currentDocument.isEditable();
```

In the same column, on row 3, add a List Box core control with the following properties:

Name: "**ListReviewers**"

Height: 60px

Width: 400px

Visible

Disabled

The screenshot shows the Properties window for a **List Box** control. On the left is a sidebar with tabs: **List Box** (selected), Data, Values, Style, Font, Background, Margins, and All Properties. The main area displays the following properties:

- Name:** ListReviewers
- Height:** 60.0 Units: Pixels
- Width:** 400.0 Units: Pixels
- Tab order:** (empty box)
- Accelerator:** (empty box)
- ☒ **Visible**
- ☐ **Read-only**
- ☒ **Disabled**

On the right, there is an **Options** section with a checkbox for **Allow multiple selections**, which is currently unchecked.

Under that field, but still in the same table cell, add a Multiline Edit Box control, with the name "**DataReviewers**" with the same width and height as ListReviewers, and leave it Visible (we'll hide it later on).

This field will have the data binding with the reviewers field in the underlying Notes document:

On the Data tab of the field properties, no data source is available. This is because earlier, we have removed the data source definition from the custom control and moved it to the XPage.

Instead of using Simple data binding, click on Advanced and select to use Expression Language (EL) and enter "currentDocument.ReviewerList" as value:

The screenshot shows the Properties window for a Multiline Edit Box control, with the **Data** tab selected in the sidebar. The main area displays the following data binding configuration:

- Bind data using:**
 - ☐ Simple data binding
 - ☐ JavaScript
 - ☒ **Advanced**
- Advanced** section:
 - Use:** Expression Language (EL)
 - Value:** currentDocument.ReviewerList
- Compute** options:
 - ☒ Compute dynamically
 - ☐ Compute on page load

Adding reviewer names to the list

In order to add newly entered reviewers to the list, we will make add a button next to the AddReviewer field and make use of scoped variables to transfer the value of this field to the list box field and the multiline edit box field.

Scoped variables

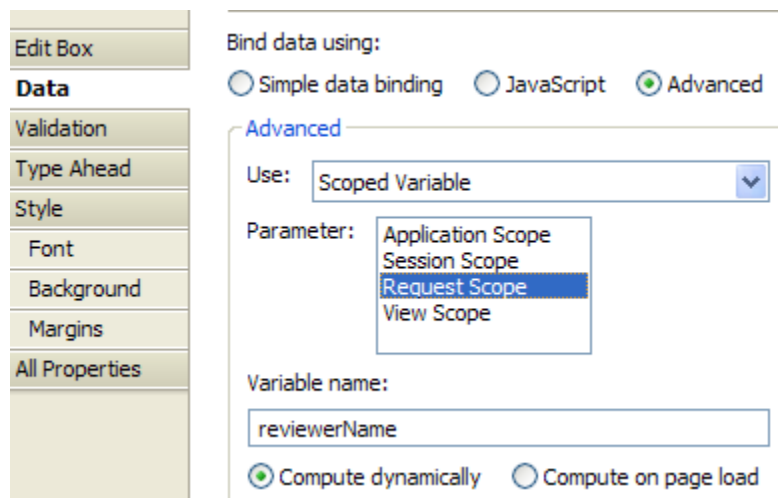
Scoped variables are JavaScript variables that share values within a certain scope:

- sessionScope: allows you to share values across pages for the duration of a session
- applicationScope: allows you to share values across pages for the duration of an application
- viewScope: allows to share values across a view. Note that this is not a Notes view but an <xp:view> (this is the root tag of an XPage)
- requestScope: allows you to share values across pages for the duration of a request. This is commonly used in search functions.

In our XPage, we will use a requestScope variable to capture the value of the AddReviewer field, and a sessionScope variable to populate the ListReviewers field:

Select the field **AddReviewer**, and in the Data tab of the Properties, select Advanced: Use Scoped Variable.

Select Request Scope and enter the Variable name "reviewerName":



Add a **button** on the right of the field AddReviewer.

Give it the label "Add" and set the Button type to "Submit" (we will change that later!)

Give it a style class "lotusBtn".

Add the following onclick event Server Side JavaScript:

```
var v = requestScope.reviewerName;
if (v!=null) {
    if (sessionScope.reviewerName!=null) {
        v += ",";
        sessionScope.reviewerName += v;
    } else {
        sessionScope.reviewerName = v;
    }

    requestScope.reviewerName = "";    //clear the AddReviewer field
}
```

The above code retrieves the value from the requestScope variable "reviewerName", which is the value of the AddReviewer field. It then adds it to a sessionScope variable with the same name (the names don't need to be the same, they are just kept the same here to make it easier to remember the variable name).

Finally, the requestScope variable is cleared. This will empty the AddReviewer field, so that another reviewer can be added.

Accessing field values in an XPage

The above example demonstrates how the value of the field AddReviewer is accessed by using a scoped variable. This works, because the field's data binding was done to that variable.

There are other ways to **access a field value via Server Side JavaScript**:

```
currentDocument.getItemValueString("AddReviewer"); //NotesXspDocument
dominoDoc.getItemValueString("AddReviewer"); //NotesXspDocument
getComponent("AddReviewer").getValue(); //getComponent gets the base object for a
UI component
dominoDoc.getDocument().getFirstItem("Subject").getValueString() //for back-end
data access: getDocument returns a NotesDocument object
```

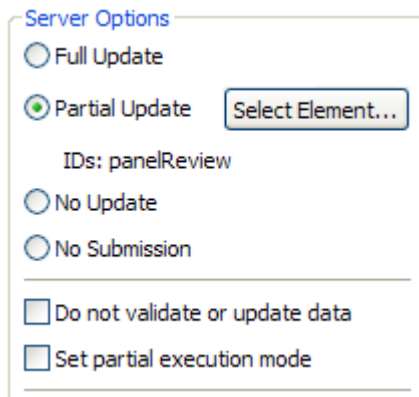
To access a field value via Client Side JavaScript, you can use this:

```
dojo.byId("#{id:AddReviewer}").value;
```

In order to write a value to a field, you could use:

```
currentDocument.replaceItemValue("AddReviewer","new value");
getComponent("AddReviewer").setValue("new value");
```

In the Server Options of the Add button's onclick event, select "Partial Update" and select the element to update: "panelReview". This is the panel containing the reviewer table and fields.



Next step is to display the values from `sessionScope.reviewerName` into the list of values of the List Box field.

In the **List Box** tab of the ListReviewers field properties, make sure the option "Allow multiple selections" is switched on. This doesn't seem to make much sense (we're not going to select values in the list box), but not having this option selected causes a run time error.

Leave the **Data** tab untouched: there is no data binding needed for this field, as it will serve for display only.

On the **Values** tab, click "Add Formula Item..." and enter the following Server Side JavaScript:

```
@Explode(sessionScope.reviewerName, ",", 0)
```

This is splitting the `sessionScope` variable into separate values.

Now open `formDocument.xsp` in the browser, and test the form:

Make sure you enter a Subject of at least 3 characters in the form before testing the Add Reviewer function.

Then, enter a reviewer name and click Add. The name is added to the list. Additional reviewers can be added to the list.

If you now empty the subject field and try to add another reviewer, you will see it won't work. This is caused by the validations on the Subject field. Since we did not disable the validations for the Add button (checkbox "Do not validate or update data" in the Event tab is disabled), the form validations are performed, but no error message is displayed.

Setting the Add button not to validate data won't solve the problem, as this will also not update the data (and thus add the reviewer name to the list).

Accessing field values before validation

Field values can be accessed before validation, by using the following methods:

```
getComponent("AddReviewer").getSubmittedValue(); //instead of getValue  
getComponent("AddReviewer").setSubmittedValue(); //instead of setValue
```

Adding reviewers without data validation

Knowing this, we can update the Add button to work without form validation:

In the onclick event of the Add button do the following:

Under Server Options, enable the option "Do not validate or update data"

In the script editor, replace the existing Server Side JavaScript by this:

```
var rev = getComponent("AddReviewer");  
var v = rev.getSubmittedValue(); //Get the value of the reviewername  
rev.setSubmittedValue("");  
if((v!="") && (v!=null)) {  
    if(sessionScope.reviewerName!=null) {  
        v += ",";  
        sessionScope.reviewerName += v;  
    }else {  
        sessionScope.reviewerName = v;  
    }  
}
```

Important

Change the button type to "**Cancel**" (instead of Submit).

I can't give a decent explanation for this, but with it appears to be the only way to get it working: with the type set to "Submit", values are not added to the list box. If it is set to "Button", a new document is created (in the background, with subject "Untitled") each time a reviewer name is added to the list.

Save ccFormDocument and test the result in the browser.

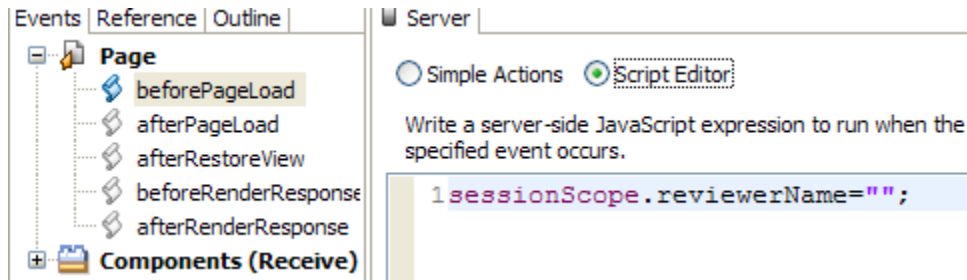
Now, you can add reviewers even when the Subject field is left blank, because the validations are not executed anymore.

Clearing the sessionScope variable

The variable sessionScope.reviewerName needs to be cleared when opening the XPage. Otherwise, it will load the ReviewersList list box in any new document form that is opened within the same session

with its existing values.

Select the Custom Control in the outline of ccFormDocument (=top level of the hierarchy) and then open the Events view. In the beforePageLoad event, clear the sessionScope variable:



When a new library document is created, the list of reviewers will now be empty.

Writing the reviewers list to the data field

We now manage to add reviewers to the list, but they are not stored yet in the document, when it is saved: we need to populate the field **DataReviewers**, which will be saved to the item ReviewerList in the Document form.

Add the following code to the Add button's onclick event, under the existing code:

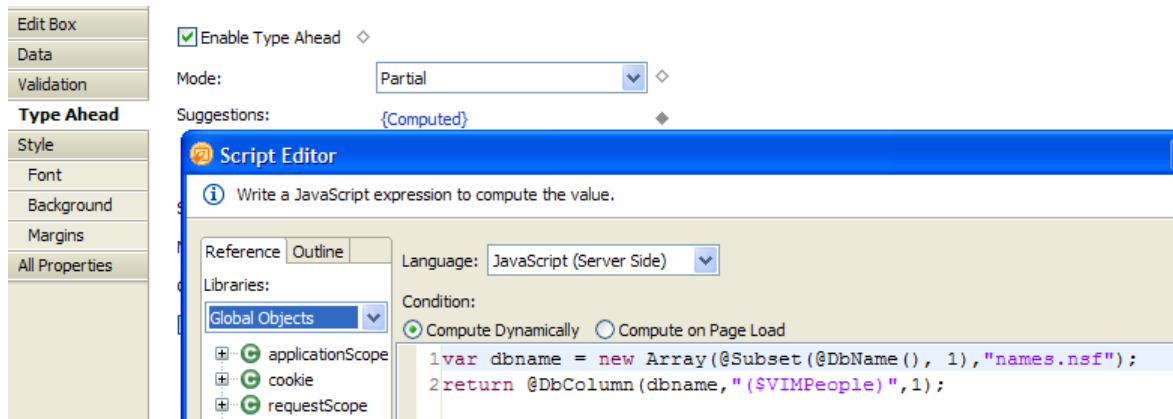
```
var lst = @Explode(sessionScope.reviewName, ",", 0);
var txt = getComponent("DataReviewers");
txt.setSubmittedValue(sessionScope.reviewName) //write the updated list of
names to the hidden text field
```

Save and preview the result: the added values are now also written to the DataReviewers field. When you open an existing library document in read mode, the List Box field is empty, but the multiline text field now is populated with the reviewer names.

Adding type ahead

In order to facilitate the input of reviewer names, we will add a type ahead feature to the AddReviewer field, and build a list of suggestions from the Public Address Book:

On the Type Ahead tab of the Properties of the field AddReviewers, enable type ahead. Select "Partial" mode and add a computed value for the suggestions:



Disable case sensitive.

Test the result: type a first letter of a name in the NAB in the AddReviewer field.

Type ahead and styles

The fields "Tags" and "Add reviewer" have a slightly different look in the web form than the other editable fields: their height is less than the subject field, and the field border is slightly different. Also, these fields are a couple of pixels indented, in comparison to other fields.

This is because the type ahead feature adds an extra tag around the field.

The form might need some additional style tweaking to get all fields aligned properly.

Field hint

Now we will add a field hint to the field AddReviewer: a text that appears in the field and explains the user what to do. Once he clicks in the field, the hint disappears.

On the Data tab of the properties of AddReviewer, add a default value:

Data

Validation

Type Ahead

Style

Font

Background

Margins

All Properties

You must link (bind) this control to a data source in order to display data.

Bind data using:

☐ Simple data binding ☐ JavaScript ☒ Advanced

Advanced

Use:

Parameter:

Variable name:

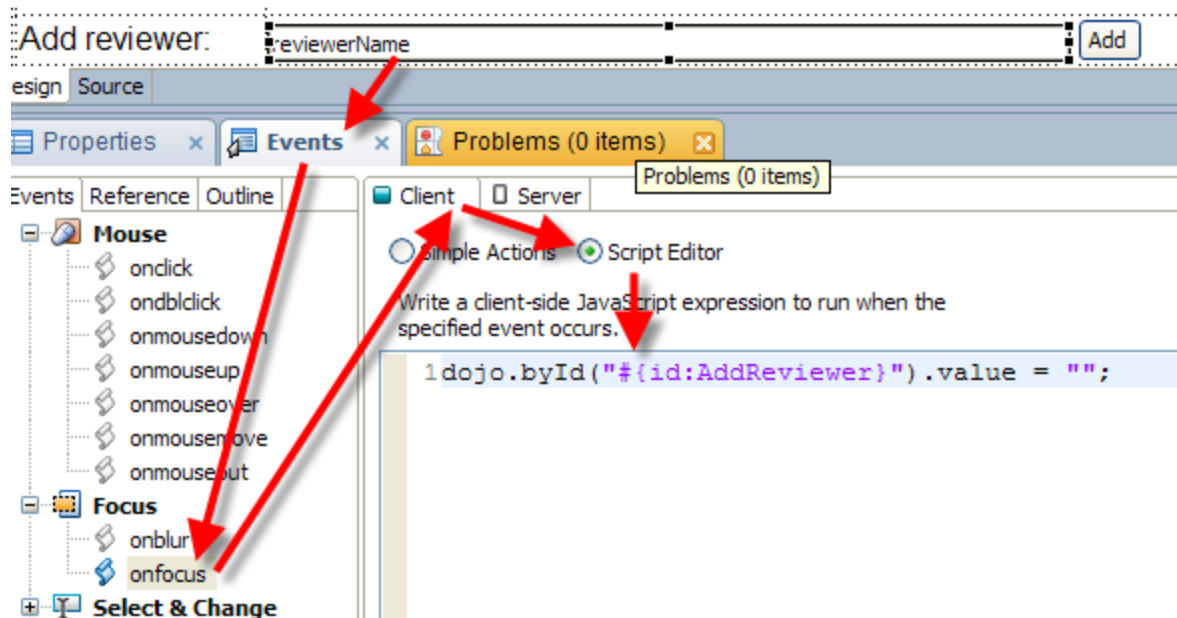
☒ Compute dynamically ☐ Compute on page load

Default value:

In the Events view, add an onfocus event for the AddReviewer field, to clear the field as soon as the user clicks in it.

Enter the following Client Side JavaScript:

```
dojo.byId("#{id:Reviewers}").value = "";
```



Save ccFormDocument and test the result.

With appropriate styling, you could make the field hint text appear as grayed out (gray font). It would take some experimenting, but there must be a way to change the font color back to black when the user enters the first reviewer name.

Understanding events

Above, we have added an onfocus event to the field AddReviewers. This creates an event handler. It is not the same as a simple onfocus property of an html input field. This last property can be set by going to the All Properties tab of the field properties:

Edit Box	Property	Value
Data	basics	
Validation	data	
Type Ahead	events	
Style	onblur	
Font	onchange	
Background	onclick	
Margins	ondblclick	
All Properties	onfocus	
	onkeydown	
	onkeypress	
	onkeyup	
	onmousedown	
	onmousemove	
	onmouseout	
	onmouseover	
	onmouseup	
	onselect	

This onfocus event has no value.

To better understand the difference, add a simple Client Side JavaScript alert to the onfocus event in the All Properties tab, and then inspect the Source code of the custom control. The simple onfocus source code looks like this:

```
<xp:this.onfocus><![CDATA[alert("onfocus")]]></xp:this.onfocus>
```

The eventhandler onfocus (=via the Events view) source code looks like this:

```
<xp:eventHandler event="onfocus" submit="false">
<xp:this.script><![CDATA[dojo.byId("#{id:AddReviewer}").value =
"";]]></xp:this.script>
</xp:eventHandler>
```

The first code will translate to a simple onfocus property in the browser, while the second will generate a JavaScript event handler function (check the html source in the browser).

Data validation

The type ahead feature in AddReviewers is helpful, but it doesn't prevent users from entering inexistent names and adding them to the reviewers list. Therefore, we will add some validations to AddReviewers:

- only existing user names should be added
- the author of the document cannot add himself as reviewer

Extend the "Add" button JavaScript with the above validations:

```
var dbname = new Array(@Subset(@DbName(), 1), "names.nsf");
var nabNames = @DbColumn(dbname, "($VIMPeople)", 1);
var user = session.getEffectiveUserName();
var userAbbr = @Name("[ABBREVIATE]", user);

var revComponent = getComponent("AddReviewer");
var reviewer = revComponent.getSubmittedValue();

var v = @Name("[ABBREVIATE]", reviewer);

if (@LowerCase(v) == @LowerCase(userAbbr)) {
    revComponent.setSubmittedValue("You cannot add yourself as reviewer");
}else if (@IsNotMember(reviewer, nabNames)) {
    revComponent.setSubmittedValue("Please enter a valid name");
}else
{
    revComponent.setSubmittedValue("");
    if ((v != "") && (v != null)) {
        v += ",";
        sessionScope.reviewerName += v;
    }
}
```



```

}

var lst = @Explode(sessionScope.reviewerName,"",0);

var txt = getComponent("DataReviewers");

txt.setSubmittedValue(sessionScope.reviewerName)

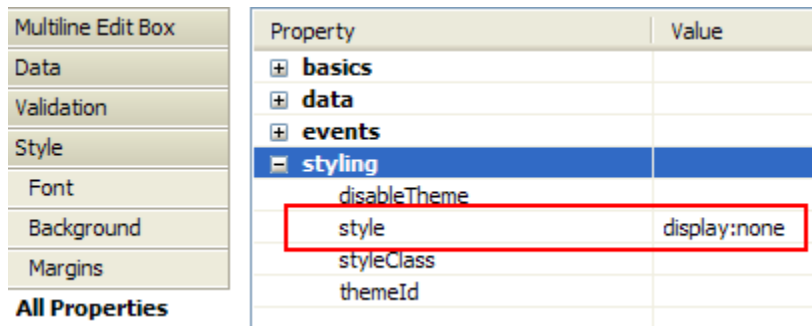
}

```

Hide the field DataReviewers

This field is intended for data storage only and not for data display. Therefore, it will be made hidden. Disabling the Visibility property is not a good idea here. The field would then simply not be rendered in the XPage and would not be accessible by the JavaScript behind the Add button. Instead, we will set the style to not display the field:

Select the field DataReviewers and in the Properties view, go to the All Properties tab. Under "styling", replace the existing value of the property "style" by "display:none":



This style applies to an input field. In read mode, the field will be rendered as text instead of an input field, and will still be visible.

In read mode, you could either display the field ListReviewers or DataReviewers. Hide the other field by adding a computed visibility:

```
currentDocument.isEditable();
```

Add the same visibility value to the "Add" button, in order to hide it in read mode.

Populate the List Box at page load

When an existing document is opened, the list box of reviewers is empty. This is because the sessionScope variable is cleared when the page is loaded (Section 5.5 Clearing the sessionScope

variable).

In order to display the list of reviewers in existing documents, we need to populate the sessionScope variable at page load:

Select the custom control in the Outline view (=top level of the three), and open the Events view.

Clear the code in the beforePageLoad event.

Instead, add the following code to the afterPageLoad event:

```
if(!dominoDoc.isNewNote()){
    sessionScope.reviewerName = getComponent("DataReviewers").getValue();
}else{
    sessionScope.reviewerName = "";
};
```

Add a Clear button

In this step, we will add a "Clear" button, allowing users to clear the list of reviewers:

Drag a new Button control next to the ReviewersList field.

Leave the type "Button" and give it a label "Clear".

Assign a style class "LotusBtn" and set its visibility to display in edit mode only.

Add an onclick event with the following Server Side JavaScript:

```
sessionScope.reviewerName = "";
var txt = getComponent("inputTextareal");
txt.setSubmittedValue("")
```

In the event's Server Options, select Partial Update: panelReview and enable the property "Do not validate or update data".

Review options

In the Notes client version of the Document Library, the author can specify some review options: type of review (serial or parallel), time limit, and notification options. In this section, we will add these options to the custom control ccFormDocument.

Review options fields

Select the 3 cells in the last column of the table contained in panelReview, and merge these cells:

The screenshot shows a Domino web form with three main sections:

- Attachments:** Includes a 'Browse' button and a table with columns: File names, Size (bytes), Type, Created on, and Delete.
- Add reviewer:** Includes a text field for 'reviewerName', an 'Add' button, and a '{ Computed Items }' label.
- Reviewers:** Includes a 'ReviewerList' field with up/down arrows and a 'Clear' button.

 A red arrow points to a vertical scrollbar on the right side of the form, indicating that the form content is scrollable.

Drag a Section container control in this cell, and give this section the name "**sectionReviewOptions**" and a label "Review Options".

Enable the option "Section is closed by default".

Assign the style class "lotusSection".

Drag a new table into this section of 4 rows and 2 columns.

In this table, add the following fields:

Label	Type	Name	Bind to
Type of review:	Combo Box	comboReviewType	dominoDoc.ReviewType
Time limit:	Combo Box	comboReviewWindow	dominoDoc.ReviewWindow
Time limit (days):	Edit Box	editReviewTime	dominoDoc.ReviewTime
Notify originator after:	Combo Box	comboNotifyAfter	dominoDoc.NotifyAfter

Give the Combo Box fields a width of 250px.

On the Data tab of the Edit Box, set Display type to "Number".

Copy field values from the Notes client

In order to populate a Combo Box field with a list of available values, you can either type the values one by one, compute the values with JavaScript, or import them as a text list.

The last option is useful when web enabling existing Notes applications: you can copy the list of choices from the Notes form and import them in the XPage Combo Box field. We are going to do this for the Review Type field.

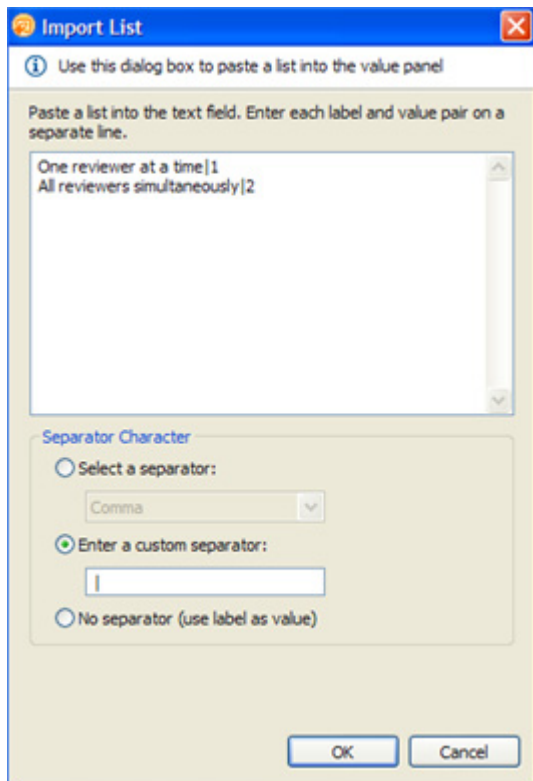
Open the form Document in Notes Designer and copy the values of the dialog list ReviewType to the clipboard.

In the Values tab of the comboReviewType field properties, click on "Import List..."

Paste the values from the form in the text field.

Now, **remove any space characters before and after the vertical bar**. This is essential to make the values work!!

On the bottom of the dialog window, specify a custom separator, and enter the vertical bar:



Press OK and the values appear in the values list in the properties view.

Use the same approach to populate the values of the field comboNotifyAfter.

In the above cases, importing values clearly doesn't speed up the work much, as there are only 2 values. But with longer lists, this is very helpful.

Copy a choicelist value formula from the Notes client

The dialog list field ReviewWindow in the Document form populates its values with an @formula, because the available options are in function of the selected review type:

```
Serial := "No time limit for each review0" :
```

```
"Move to next reviewer after time limit expires1": "Keep sending reminders
after time limit expires2";
Parallel := "No time limit for each review0" : "1" : "Keep sending reminders
after time limit expires2";
WebParallel:= "No time limit for each review0" : "Keep sending reminders
after time limit expires2";
@If(
@ClientType != "Notes" & ReviewType="2"; WebParallel;
ReviewType = "1"; Serial; ReviewType = "2" ; Parallel; "")
```

This can easily be converted to JavaScript:

Copy the formula from the Notes field.

In ccFormDocument, go to the Values tab in the properties of comboReviewWindow.

Select "Add Formula Item..." and past the formula in the Server Side JavaScript editor. As expected, a lot of errors are highlighted in the margin, because this is not JavaScript yet.

Before transforming to JavaScript, clean-up the code, by removing the parts that apply to the Notes client only:

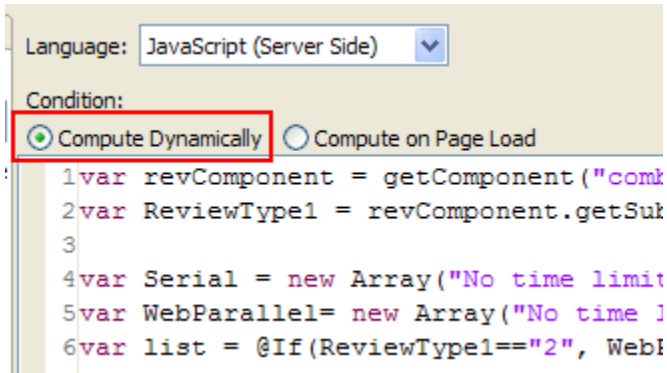
```
Serial := "No time limit for each review0" :
"Move to next reviewer after time limit expires1": "Keep sending reminders
after time limit expires2";
WebParallel:= "No time limit for each review0" : "Keep sending reminders
after time limit expires2";
@If(ReviewType="2"; WebParallel;Serial)
```

Now, replace any @Formula syntax by JavaScript syntax, and access the value of the ReviewType field by using getSubmittedValue(). The result would be something like this:

```
var revComponent = getComponent("comboReviewType");
var ReviewType = revComponent.getSubmittedValue();
var Serial = new Array("No time limit for each review0","Move to next
reviewer after time limit expires1",
"Keep sending reminders after time limit expires2");
var WebParallel= new Array("No time limit for each review0","Keep sending
reminders after time limit expires2");
var list = @If(ReviewType=="2", WebParallel,Serial);
return list
```

Again, this example is quite simple and could probably easily be rewritten from scratch, but copying the formula allows to reuse the code structure and avoid type errors in literal strings. And you can use the same @Functions in JavaScript as in @Formula.

Before exiting the JavaScript editor, make sure the compute condition is set to Compute Dynamically:



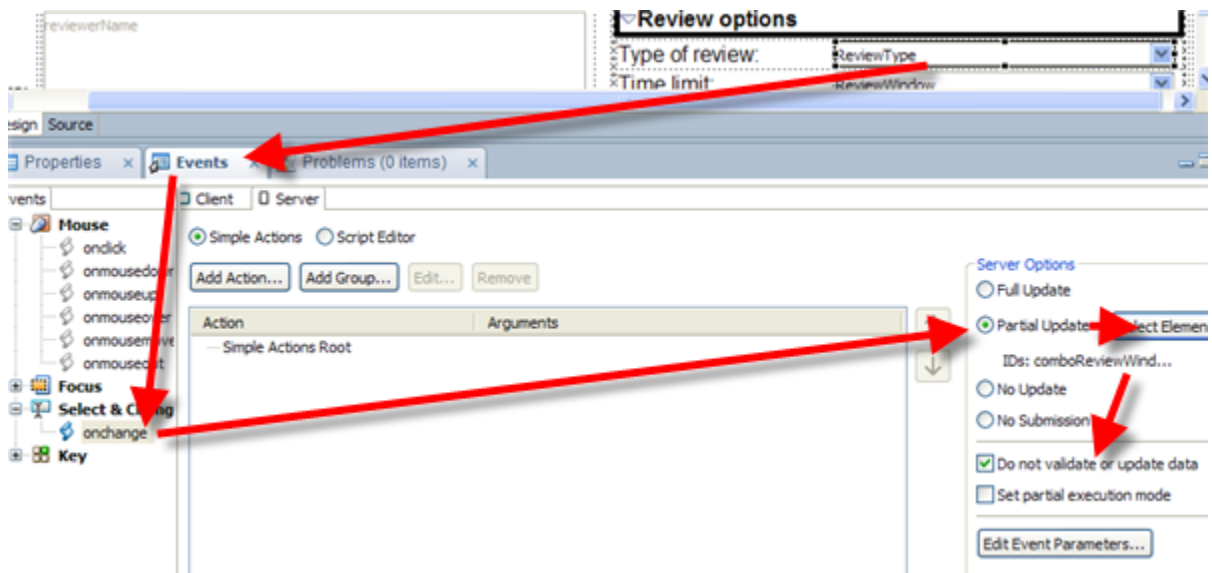
Update Time Limit in function of Review Type

In order to make the time limit options adapt to the selected review type, we need to tell the first Combo Box to update the second onchange:

Select comboReviewType and in the Events view, go to the onchange event.

Under Server Options, select Partial Update and select the element comboReviewWindow.

Enable "Do not validate or update data":



Preview the result in the browser: the time limit options are now available in function of the selected review type.

Display the review time in function of the review window

The ReviewTime field only needs to be displayed if the ReviewWindow implies it. That is, if the ReviewWindow is different from "No time limit for each review".

In order to achieve this, we need to:

- Add an onchange event in comboReviewWindow to trigger a refresh of the ReviewTime
- Compute the visibility of the field editReviewTime and its label

ReviewWindow onchange

In the Events view of the field comboReviewWindow, set the following options for the onchange event:

- Partial Update: sectionReviewOptions
- Do not validate or update data

Selecting the appropriate element for partial update

For partial updates, you can often choose between several elements to be updated. For example, in the above onchange event, we could have decided to update panelReview instead of sectionReviewOptions.

In general, you want to have the smallest possible element to update, to reduce unneeded data transfer from the server.

In the above example, sectionReviewOptions appears to be the best choice:

Partially refreshing only the ReviewTime field is not sufficient, since we also want to update the field label. We could have assigned an id to the table row containing the ReviewTime field and its label (via the All Properties tab of the table row), but that wouldn't work, because then, this row is only rendered (=visible) in function of the value of ReviewWindow. Which means that its ID wouldn't be accessible in case it is not rendered (=hidden). In that case, an error would occur:

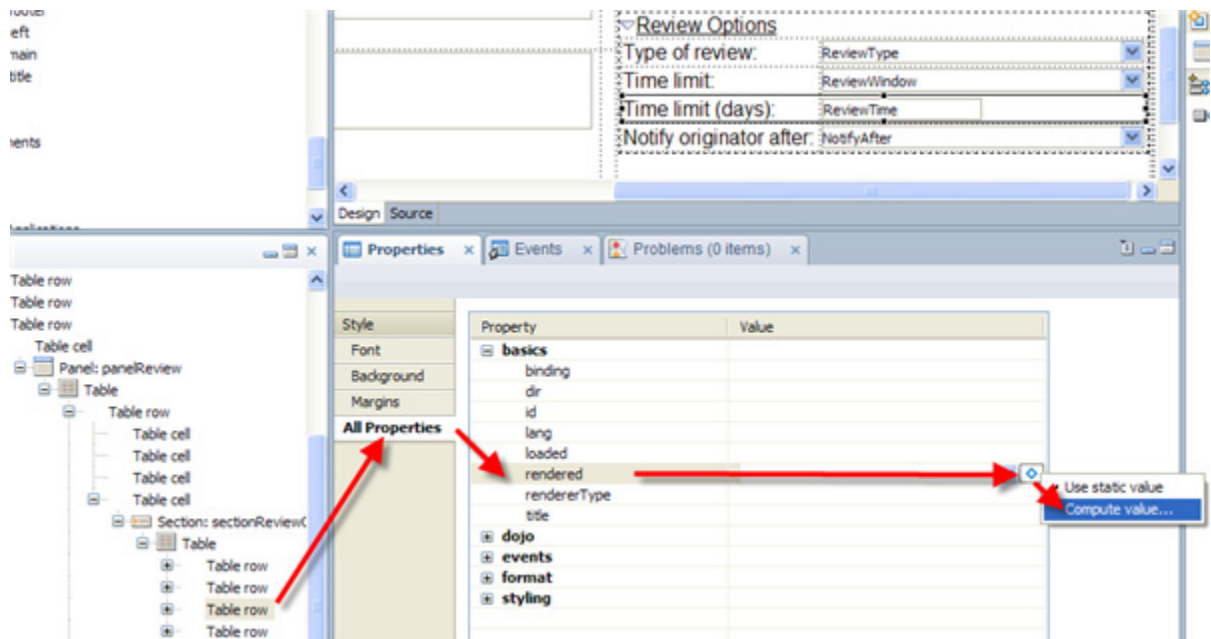


Visibility of ReviewTime

Via the Outline, select the table row containing the editReviewTime and its label.

In the All Properties tab, add a computed value for the rendered property:

```
var revComponent = getComponent("comboReviewWindow");
var revWindow = revComponent.getSubmittedValue();
revWindow != "0"
```



Result

The visibility is correctly calculated when the ReviewWindow is modified during edit of a document. But the visibility is incorrect

- when creating a new document (default for ReviewWindow is "No time limit", but the ReviewTime is still displayed)
- when opening an existing document

In both cases, getSubmittedValue doesn't return a value for the ReviewWindow, because this method only returns a value during a partial update.

The computed value for the visibility needs to be extended.

Combining getValue and getSubmittedValue

- use getSubmittedValue if it is not null
- else, use getValue
- if both are null (= when creating a new document), return false: do not display the time limit (because by default, the ReviewWindow is "No time limit")

The new computed value for the rendered property would be:

```
var revComponent = getComponent("comboReviewWindow");
var revWindowV = revComponent.getValue();
var revWindowSV = revComponent.getSubmittedValue();
if(revWindowSV!=null){
    return(revWindowSV!="0");
}else if(revWindowV!=null){
    return(revWindowV!="0");
}else{
    return(false)
}
```

The visibility of the time limit is now always correctly computed.

Summary for Section – Sample 1 - Form Design

This page was a long and interesting journey along the features and potential pitfalls when developing a web form with XPages.

After discussing form layout, we have covered common form features like action buttons, element visibility, validations, type-ahead, partial refresh,...

By doing this, we have come in touch with various XPage concepts, including source definition and data binding, scoped variables, accessing the document context and field values, accessing field values before validation and event handlers.

For now, we will consider the Library Document form as completely functional, apart from workflow, which will be discussed on the next page.

Workflow - XPage enabling an existing Notes client application

This page explains how a workflow can be implemented into a web application built with XPages. Starting from an Notes client workflow application, this page will demonstrate how most of the existing LotusScript can be used within the web workflow. Also, an alternative solution using Server Side JavaScript will be included.

Understanding the existing Notes client workflow

Before getting started with adding the workflow into the XPage application, we need to have a look at the existing Notes client workflow to fully understand how it works.

Workflow setup

In the Document form, the Notes client workflow steps are triggered by the following action buttons:

- Submit for review (author)
- My review is complete (approver)
- Clear review cycle (author)

Additionally, a **scheduled agent** called "Process Late Reviews" processes pending reviews after the specified time limit

The action buttons contain @Formulas which

- perform initial field validation
- set field values
- call a document save

The form's QuerySave event

- performs field validation (overlap with action buttons)
- sends e-mail to workflow actors (serial or parallel)
- updates fields (status, currenteditor)

QuerySave

The code in the form's QuerySave event can be divided basically into 3 parts:

The first part identifies which action has been clicked and defines, in combination with the status, whether the execution should continue or not.

The second part performs field validations.

The third part, on the very bottom, performs the actual backend processing by calling the functions SendToNext and SendToAll in the Script Library "SubmitForReview".

```

Sub Querysave(Source As Notesuidocument, Continue As Variant)
    Dim ws As New NotesUIWorkspace

    Set session = New NotesSession
    Set db = session.CurrentDatabase
    If note Is Nothing Then Set note = source.Document

    note.RemoveItem("SaveOptions")

'reset NSubmit if it was error submitting, user fixed error and did not submit
If note.NSubmitNow(0) = "1" Then
    NSubmitNow = True
Else
    NSubmitNow=False
End If

make sure that NotifyAfter is set
If note.NotifyAfter(0) = "" Then note.NotifyAfter = "0"

ParallelReviewCheck
'make sure that if review is complete and status=3 and not submitting - exit sub
If (Not(NSubmitNow) And note.Status(0) = 3) And note.complete(0)="1" Then Exit Sub

If note.ReviewType(0) = "2" And note.complete(0) = "1" Then
    Goto PContinue
End If

Serial review check: If we are not submitting for review, we don't need to do the rest of this
If (Not(NSubmitNow) Or note.Status(0) = 3) And note.Resubmit(0) = "0" Then Exit Sub

PContinue:
Validate that the ReviewerList does not include the originator
If ListIncludesOriginator Then
    Call note.replaceitemvalue("$DocLib", "RevCycle")
    MessageBox GetString(17) & GetString(18), 0, GetString(41)
    continue = False
    If FieldName = "Reviewer List" Then
        source.GoToField("ReviewerList")
    Else
        source.GoToField("NewReviewer")
    End If
    Exit Sub
End If

Validate that ReviewWindow is not blank. if it is, set to "0" (no time limit)
Set item = note.GetFirstItem("ReviewTime")
If note.ReviewWindow(0) = "" Then
    note.ReviewWindow = "0"
End If

Validate that ReviewTime has a valid entry if ReviewWindow is not 0
Set item = note.GetFirstItem("ReviewTime")
If note.ReviewWindow(0) <> "0" Then
    This verifies that it is not 0 or blank or a string
    If item.Text = "" Or item.Text = "0" Or item.Text Like "**ERROR**" Then
        InvalidReviewTime = True
    This verifies that it is a whole number > 0
    ElseIf note.ReviewTime(0) < 1 Or Int(note.ReviewTime(0)) <> note.ReviewTime(0) Then
        InvalidReviewTime = True
    End If
    If InvalidReviewTime Then
        FieldName = "Time Limit"
        MessageBox GetString(16) & GetString(18), 0, GetString(41)
        'reset flags when error occurred
        Call note.Removeitem("submitted")
        Call note.Removeitem("NSubmitNow")

        continue = False
        source.GoToField("ReviewTime")
        Exit Sub
    End If
End If

Validate that the Reviewer list is not blank

```

Identify the action

Field validations

Copy after submit

After submitting the document for review, a copy of the document, called the "original copy", is created and saved as a response document. This document cannot be edited any more. This is controlled by a QueryOpen event script, and not by author access.

Conclusion

This workflow is not very well structured. It is has rather organically/historically grown.

In the particular case of the Document Library, it would probably be more efficient to rewrite the workflow in order to web enable the application. This would make the code more understandable + maintainable.

A well-written notes client workflow would

- do only back-end processing in QuerySave scripts, and not front end calls (source.fieldgettext)
- make use of a **status** field to capture the document status and an **action** field/variable to capture the user action. The formula for a workflow action button would be for example:

FIELD Action = "submit"

@PostedCommand([FileSave])

- have a well-structured query save script with a case statement per action:

Select case action

case "submit":

....

case "approve:"

...

In order to make this exercise more relevant, we will focus on the back-end processing part of the workflow.

There is no real need to go into detail about the field validations part because

- this has already been covered on the previous page. For example, the test if the reviewer is not the current user. In the Notes client application, this test was done at submit. In our XPage application, the test is executed as soon as the reviewer name is added to the list (in the Add) button, which gives a better user experience.

- the Combo Box fields on the XPage cannot be blank, hence there's no need to test on that.

On this page, we will focus on implementing the **Submit for review** action. This action performs the most processing. And once the framework for workflow has been set up for this action, the other two actions can be added easily later on.

Create a workflow action in the form

We will start with creating a "Submit for review" action button in the form:

Between the Save and Edit buttons, add a new Button control with the following properties:

Name: buttonSubmit

Label: Submit for Review

Button Type: Submit

Style Class: lotusFormButton

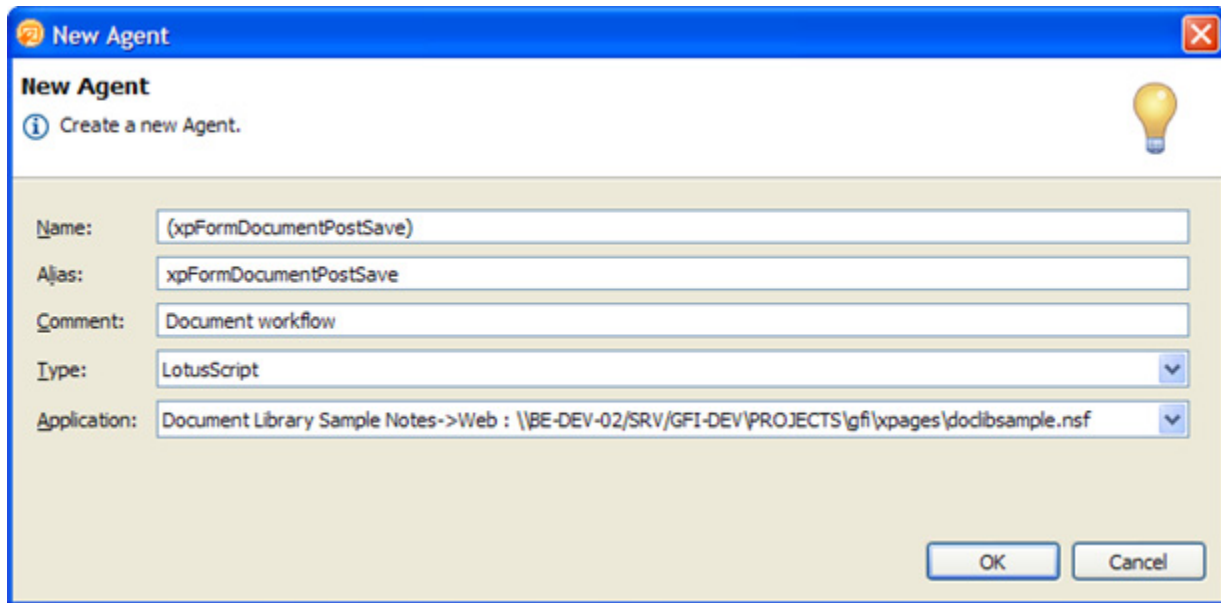
We will add coding behind the button later.

Workflow processing with a LotusScript agent

In order to reuse the existing LotusScript code, the QuerySave event code needs to be placed in an agent, which then will be placed into the postSaveDocument event of the data source of the XPage.

Create the agent

Create a new agent with the name (xpFormDocumentPostSave) and alias xpFormDocumentPostSave:



In the agent properties, choose "Agent list selection" as trigger and "None" as target.

Now we need to get add the LotusScript code of the Document form's QuerySave event into this agent, and adapt it for use as a backend agent. The main difference will be the way the current backend document is accessed.

Accessing the current document with an agent that is called from an XPage

In traditional Notes client applications, the Notesdocument object that represents the current document is accessed via **uidoc.document**.

In traditional Domino Web applications, we use **session.documentcontext**

The approach for XPages is somewhat different: the **NoteID** of the document that is calling the agent can be accessed via a property of the NotesAgent: **ParameterDocId**.

With the NoteID, the document can then be found in the database with **GetDocumentById**.

This implies that the document needs to be saved in the database before it can be accessed this way.

```
Dim doc As NotesDocument
Dim sNoteID As String
Dim agent As NotesAgent

Set agent = session.CurrentAgent
sNoteID = agent.ParameterDocId
Set db = session.CurrentDatabase
Set doc = db.GetDocumentById(sNoteID)
```

The above code gives a handle to the current document. We will use this in combination with the

backend processing part of the Document form's QuerySave agent:

'If the send (function in the SubmitForReview scriptlib) was successful, then save and close
'SaveOptions supresses the save prompt

```
If note.ReviewType(0)="1" Then
  If SendToNext Then
    note.SaveOptions = "1"
    source.Close
  Else
    Continue = False
    source.Refresh
  End If
End If

If note.ReviewType(0)="2" Then
  If SendToAll Then

    note.SaveOptions = "1"
    source.Close
  Else
    Continue = False
    source.Refresh
  End If
End If
```

The functions **SendToNext** and **SendToAll** are defined in the **Script Library "SubmitForReview"**. The QuerySave agent code suggests that these functions return a boolean with value True if the processing was successful and False if it failed.

Actually, these functions always return true: the beginning of the function SendToNext contains a statement that sets its value to true:

```
SendToNext = True
```

The function further contains no statement that returns False in case of an exception.
The same applies to SendToAll.

If, for example, in the Notes client application, you enter an inexisting reviewer name and submit the document for review, a messagebox will appear and say that no e-mail could be sent (= caused by the error handler SerMailError in the sub SerSendMail), but the script will continue execution and the document will be saved and it will have the status "submitted for review".

In other words, the If-statements for SendToNext and SendToAll can be removed in the QuerySave script.

The resulting code for the agent would then be:

```
Option Public
Option Declare
Use "SubmitForReview"
Sub Initialize()
    Dim session As New NotesSession
    Dim sNoteID As String
    Dim agent As NotesAgent

    Set agent = session.Currentagent
    sNoteID = agent.Parameterdocid
    Set db = session.currentdatabase
    Set note = db.Getdocumentbyid(sNoteID)

    If Not note Is Nothing Then

        If note.ReviewType(0)="1" Then
            Call SendToNext
        Else
            Call SendToAll
        End If
        Call note.Save(True, False, False)
    End If
End Sub
```

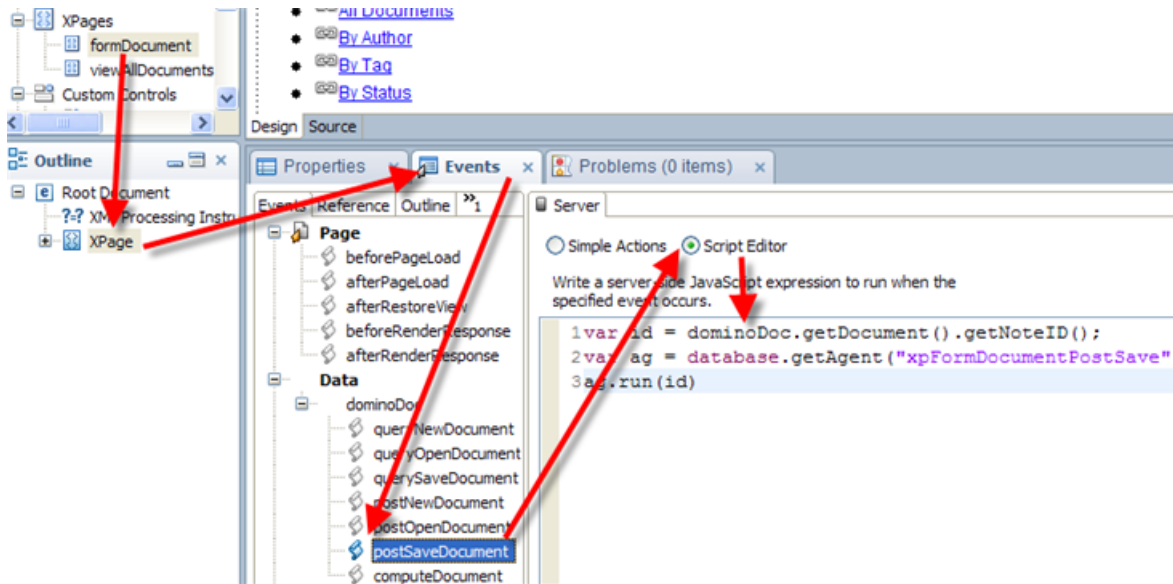
Because the functions `SendToNext` and `SendToAll` in the `SubmitForReview` script library only used backend LotusScript classes, they don't need any changes in order to have them working with the XPages.

Call the agent from the XPage

Next step is to call the agent from XPage. As indicated above, the `NoteID` of the current document needs to be passed to the agent. The agent will thus need to be called after the document was saved.

Open the XPage formDocument, and in the Events view, add the following Server Side JavaScript in the `postSaveDocument` event of the data source `dominoDoc`:

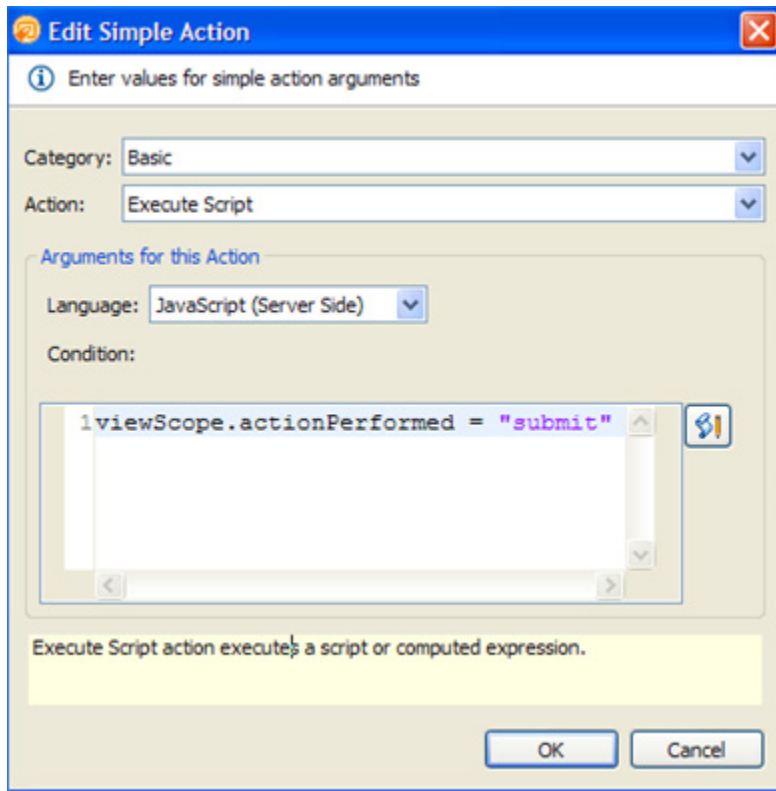
```
var id = dominoDoc.getDocument().getNoteID();
var ag = database.getAgent("xpFormDocumentPostSave")
ag.run(id)
```

Pass the action to the agent

Finally, we need to ensure that the agent only runs when the user clicks on the Submit for Review button, and not on a simple Save. This will be done by setting a scoped variable in the Submit for Review button, and reading this variable in the postSaveDocument event of dominoDoc:

Select the Submit for Review button in ccFormDocument and in the Events view, add a simple action to the onclick event:



Add a second simple action: Open Page: Previous Page.

Return to the postSaveDocument event of dominoDoc (Events of the XPage formDocument), and add a condition to only run the agent if the scoped variable has the value "submit":

```
if(viewScope.actionPerformed == "submit") {
var id = dominoDoc.getDocument().getNoteID();
var ag = database.getAgent("xpFormDocumentPostSave")
ag.run(id) }
```

When you now create a Library Document in the web browser and submit it for review, the agent code has executed: a copy of the document has been created ("Original copy"), the value of the field "status" has changed to 2, and an e-mail has been sent to the first reviewer.

Display the Submit for Review button conditionally

The Submit for Review button should only be visible if

- the document is a new document
- the status = 1 AND the current user is the author AND the document is in edit mode

Compute visibility

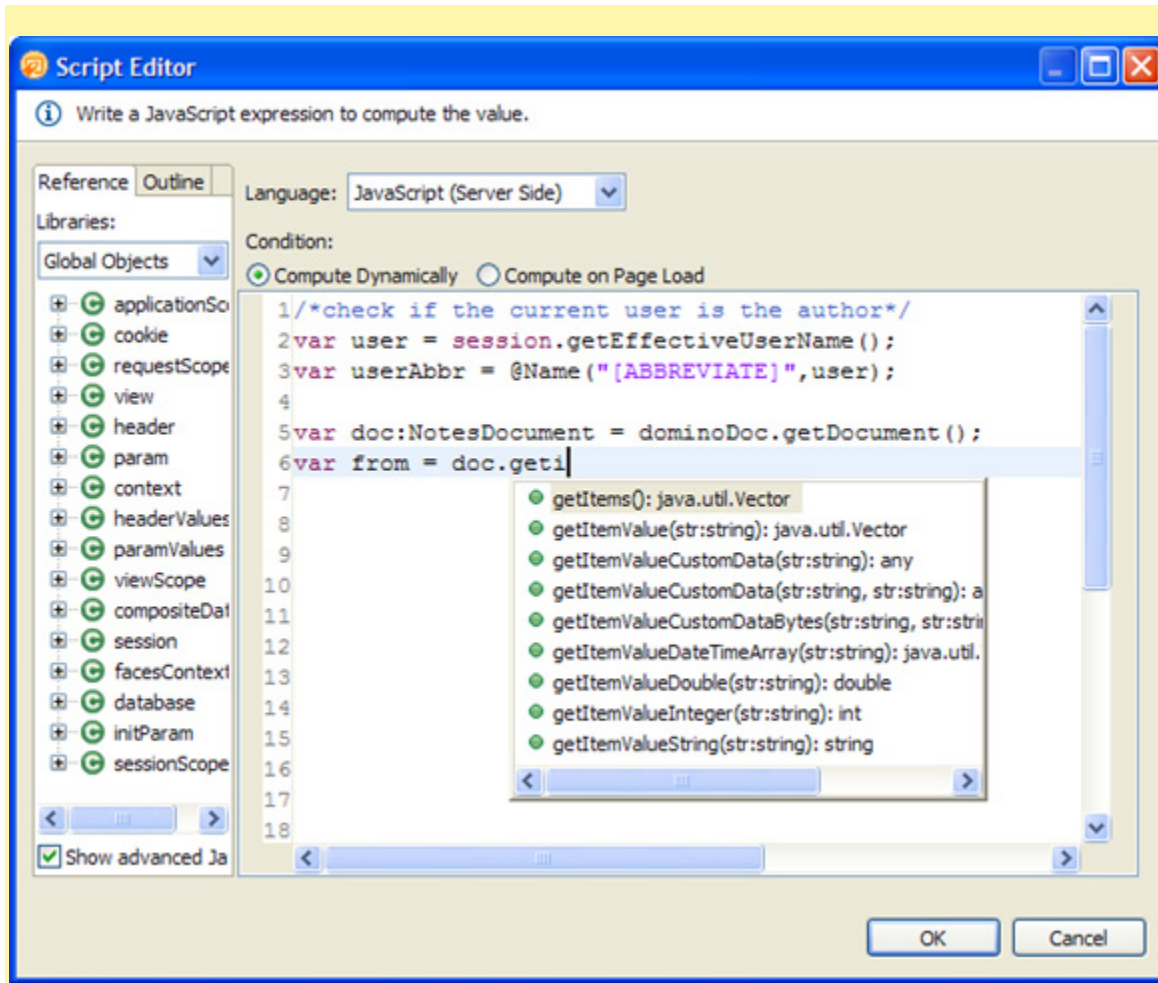
This would be the Server Side JavaScript for the computed value of the visibility of the button:

```
/*check if the current user is the author*/  
var user = session.getEffectiveUserName();  
var userAbbr = @Name("[ABBREVIATE]",user);  
  
var doc = dominoDoc.getDocument();  
var from = doc.getItemValueString("From");  
var fromAbbr = @Name("[ABBREVIATE]",from);  
  
var isAuthor = (userAbbr == fromAbbr);  
  
/*get the status*/  
var status = doc.getItemValueInteger("Status");  
  
/*final code*/  
(dominoDoc.isEditable() && (status=="1") && isAuthor) || dominoDoc.isNewNote()
```

You can explicitly specify the type of JavaScript objects:

```
var doc:NotesDocument = dominoDoc.getDocument();
```

This will enable type ahead when you use this object variable in the consecutive lines:



Use scoped variables

There might be other places in the form where we need to know whether the current user is the author of the document. Rather than computing this on each occasion, we could use a session variable and give it a value when the XPage is loaded:

The afterPageLoad event of ccFormDocument now looks like this:

```
if(!dominoDoc.isNewNote()){
    sessionScope.reviewerName = getComponent("DataReviewers").getValue();
}else{
    sessionScope.reviewerName = "";
};
/*check if the current user is the author*/
var user = session.getEffectiveUserName();
var userAbbr = @Name("[ABBREVIATE]",user);
var doc = dominoDoc.getDocument();
var from = doc.getItemValueString("From");
var fromAbbr = @Name("[ABBREVIATE]",from);
```

```
sessionScope.isAuthor = (userAbbr == fromAbbr);
```

And the adapted script to calculate the visibility of the Submit for Review button becomes:

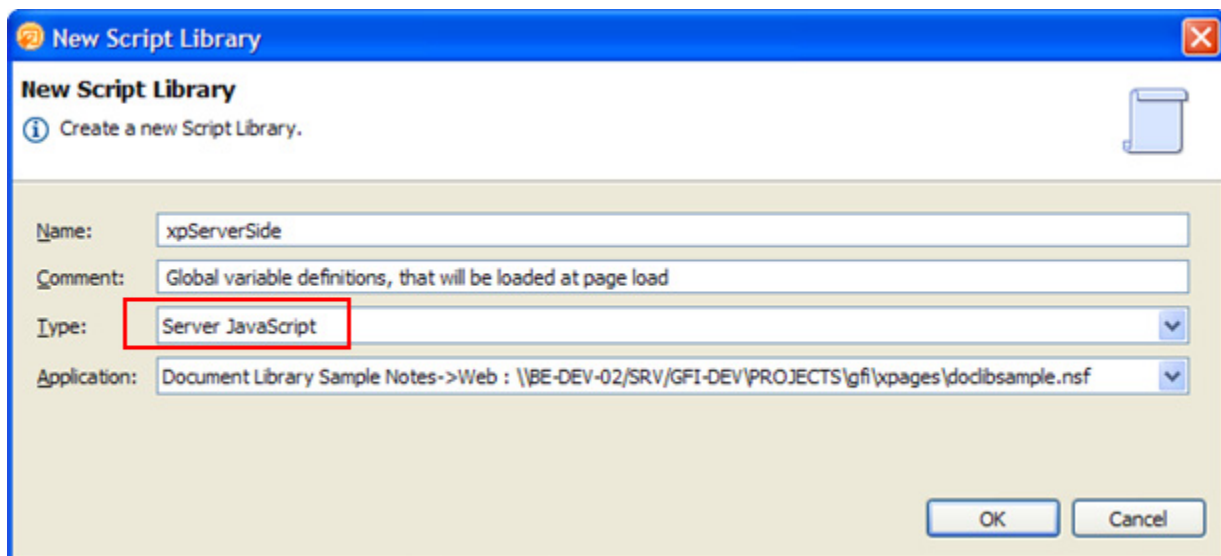
```
/*get the status*/  
  
var doc = dominoDoc.getDocument();  
  
var status = doc.getItemValueInteger("Status");  
  
  
/*final code*/  
  
(dominoDoc.isEditable() && (status=="1") && sessionScope.isAuthor) ||  
dominoDoc.isNewNote()
```

The variable sessionScope.isAuthor can now be used anywhere else in formDocument.xsp.

Create a Server Side JavaScript Library for global scoped variables

In case we want to reuse the sessionScope variable in different custom controls, it is helpful to create a Server Side JavaScript library to maintain their definitions:

Create a new Server Side JavaScript Library and name it "**xpServerSide**".



In the script library, create a function that assigns a value to sessionScope.isAuthor.

Create a second function that assigns a value to a variable sessionScope.nabNames. This variable should return a list of names from the address book, which then can be used in the type ahead and validation of the AddReviewer field.

Finally, create a function that calls the two above functions. This function will then be called from the afterPageLoad event of ccFormDocument..

The script library code could look like this:

```
function checkAuthor(){

    //check if the currentuser is the author of the document
    //write the result in the sessionScope.isAuthor variable
    var user = session.getEffectiveUserName();
    var userAbbr = @Name("[ABBREVIATE]",user);

    var doc:NotesDocument = dominoDoc.getDocument();
    var from = doc.getItemValueString("From");
    var fromAbbr = @Name("[ABBREVIATE]",from);

    sessionScope.isAuthor = (userAbbr == fromAbbr);

}

function loadNabList(){

    var dbname = new Array(@Subset(@DbName(), 1),"names.nsf");
    sessionScope.nabNames = @DbColumn(dbname,"($VIMPeople)",1);

}

/*Main function*/
function initCcForm(){

    checkAuthor();

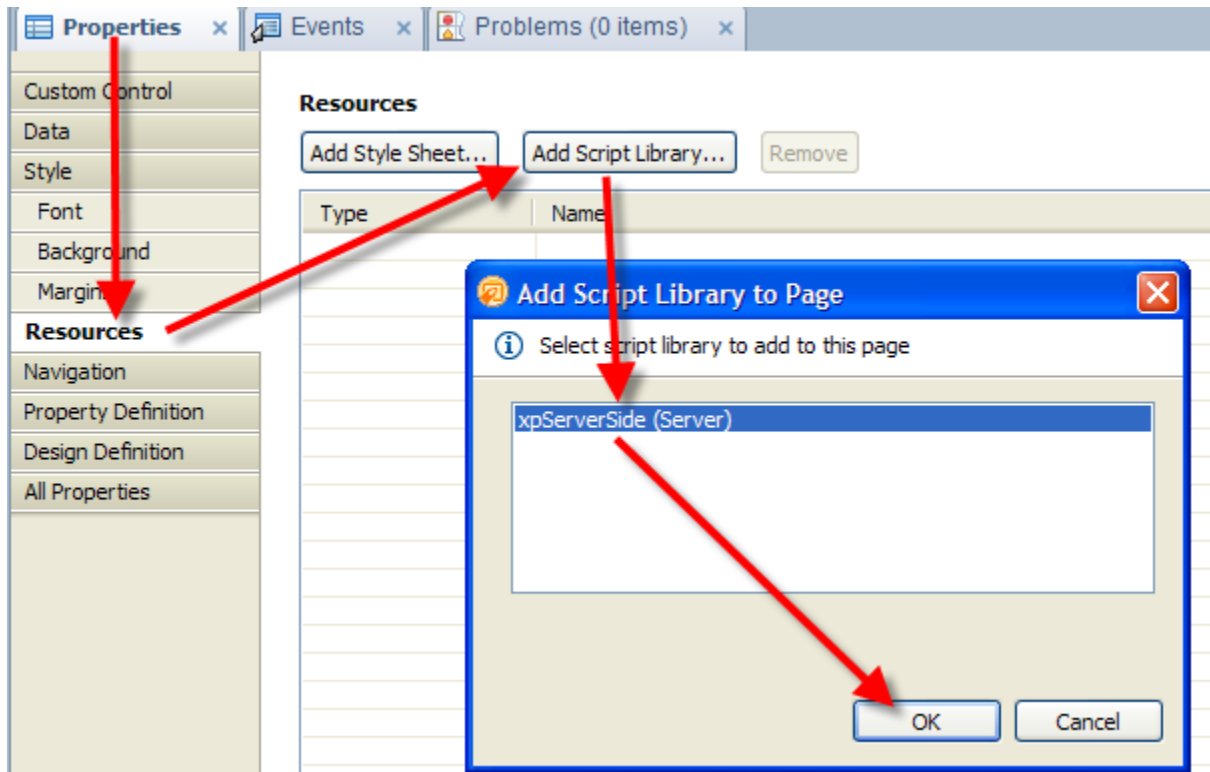
    loadNabList();

}
```

In order to use these functions in our custom control, we need to add the JavaScript Library as resource to the custom control, and we need to call the function initCcForm after page load:

In ccFormDocument, add the Script Library xpServerSide as a resource, in the Properties view of the

custom control:



In the `afterPageLoad` event of the custom control, replace the previously added `sessionScope.isAuthor` definition by a call to `initCcForm()`. The `afterPageLoad` event now looks like this:

```
if(!dominoDoc.isNewNote()){
    sessionScope.reviewerName = getComponent("DataReviewers").getValue();
}else{
    sessionScope.reviewerName = "";
};
initCcForm();
```

Now, the definition of the variable `nabNames` can also be replaced by `sessionScope.nabNames` in the Type Ahead of the field `AddReviewer` and in the onclick event of the Add button.

Use themes to display debug info

During application development it can be helpful to display certain variables in the XPage for debugging purposes.

For example, we would like to display the value of `sessionScope.isAuthor` to see if the current user is the author of the document. This can be done easily by creating a computed field and giving it the value `sessionScope.isAuthor`.

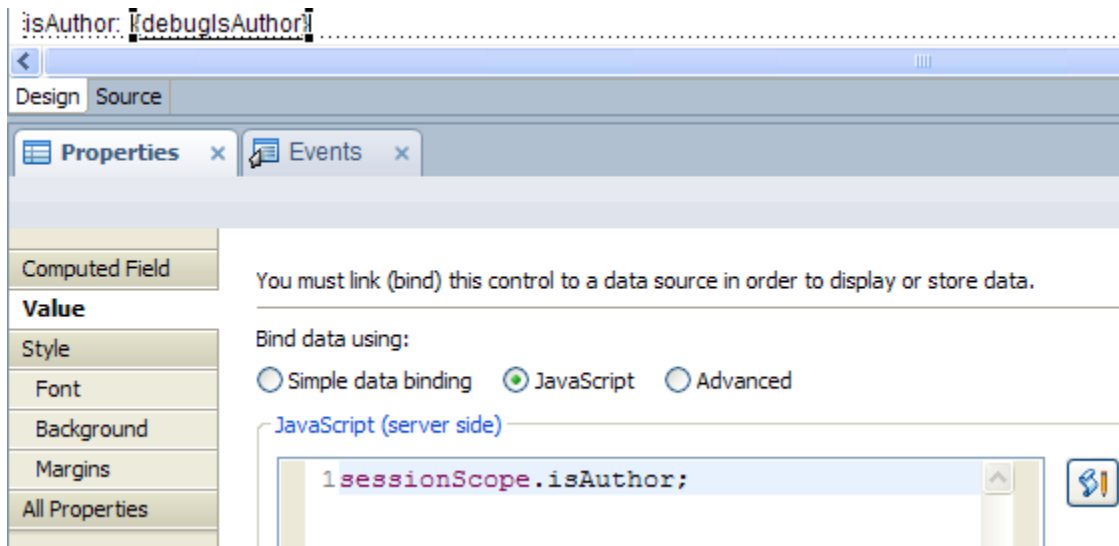
Now it would be usefull if we could easily switch between debug mode (displaying debug info) and normal mode. This can be achieved easily with themes.

We will create a panel containing all debug variables and make this panel hidden in our default theme. Then we'll create an additional theme that is a variation of the normal theme and that makes the debug info visible.

Create a container for debug info

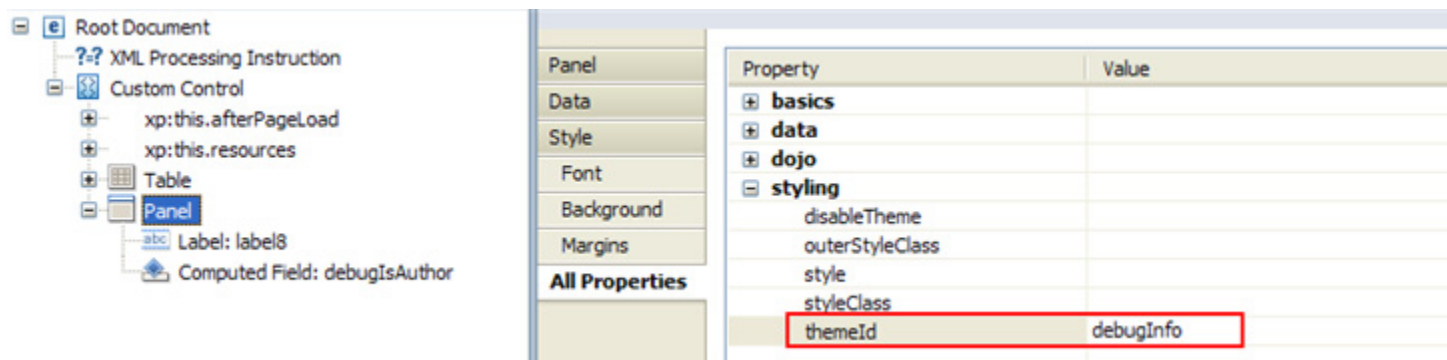
In ccFormDocument, drag a panel container control on the bottom of the custom control.

In the panel, drag a computed field and a label to display the value of isAuthor:



Add additional computed fields for other values if desired

In the All Properties tab of the panel, assign a themeId "debugInfo". This will give a handle to the panel in the themes:



Go to the Source View of ccFormDocument

Change the tag of the newly created <xp:panel>: replace the tag by <xp:div>

Why?

We don't need the functionality of an xp:panel (like event handlers), as we just need a container to access with a theme. Therefore, we can reduce overhead and use xp:div tags instead.

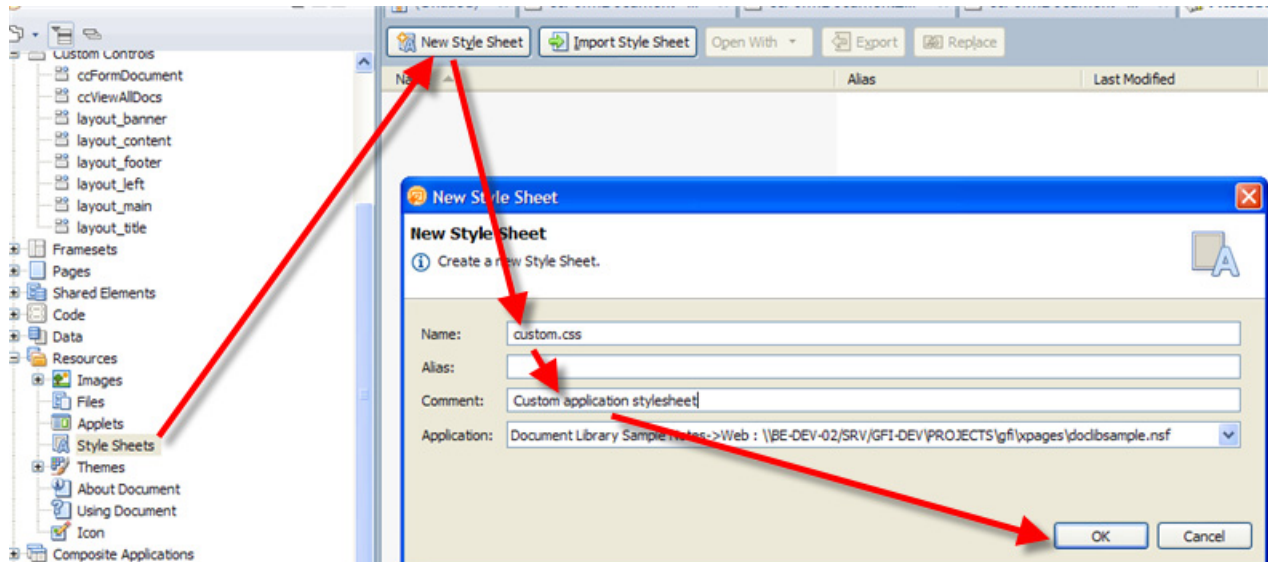
Simple <div> tags on the other hand would not work, because they are not accessible by themes.

The panel source code now looks like this:

```
<xp:div themeId="debugInfo">
  <xp:label value="isAuthor: " id="label8"></xp:label>
  <xp:text escape="true" id="debugIsAuthor"
    value="#{javascript:sessionScope.isAuthor ; }">
  </xp:text>
</xp:div>
```

Define styles for debugInfo

Create a new style sheet and call it "custom.css".



In the stylesheet create 2 new classes: one for the debuginfo in "debug mode", and one in "normal mode":

```
.debugInfoVisible {
  background-color: #EEFEE1;
  border-color: #CCEBB5;
  border: 1px solid;
  padding: 10px;
}
.debugInfoHidden{
  display:none;
```

```
}
```

Save and close the stylesheet.

Adjust the normal theme

In order to make the debug info invisible when the application is used in "normal mode", we will adjust the normal theme, and include the style info for "normal mode":

Open the theme oneuiv2-default

Add a <resource> section to include custom.css in the theme:

```
<resource>
  <content-type>text/css</content-type>
  <href>custom.css</href>
</resource>
```

Add a <control> section and assign the class "debugInfoHidden" to the themeID "debugInfo":

```
<control>
  <name>debugInfo</name>
  <property>
    <name>styleClass</name>
    <value>debugInfoHidden</value>
  </property>
</control>
```

Create a theme for debug mode

Create a new theme, "oneuiv2-default-debug"

Inherit from "oneuiv2-default"

Add a <control> section and assign the class "debugInfoVisible" to the themeID "debugInfo":

```
<theme extends="oneuiv2-default">
  <control>
    <name>debugInfo</name>
    <property>
      <name>styleClass</name>
      <value>debugInfoVisible</value>
    </property>
  </control>
</theme>
```

With the application theme set to "oneuiv2-default", the debug info will not be displayed on formDocument.xsp.

Set the application's theme in Application Properties - XPages to "oneuiv2-default-debug"

Result: the debuginfo panel is now displayed on the XPage.

Now, you can add a panel with the same themeID in other XPages and custom controls, to display debug info if the debug theme is enabled.

Workflow processing with Server Side JavaScript

Alternatively to writing the back-end workflow processing in LotusScript the workflow can also be written in ServerSide Javascript.

The approach in both cases is very identical.

In this case study, it is not very meaningful, as we already have everything written in LotusScript. But can be useful for new Web applications. The following steps are optional in this tutorial: you might want to keep the existing LotusScript workflow, or replace it by the JavaScript workflow.

The approach to implement the JavaScript workflow would be the following:

Create a serverside javascript script library.

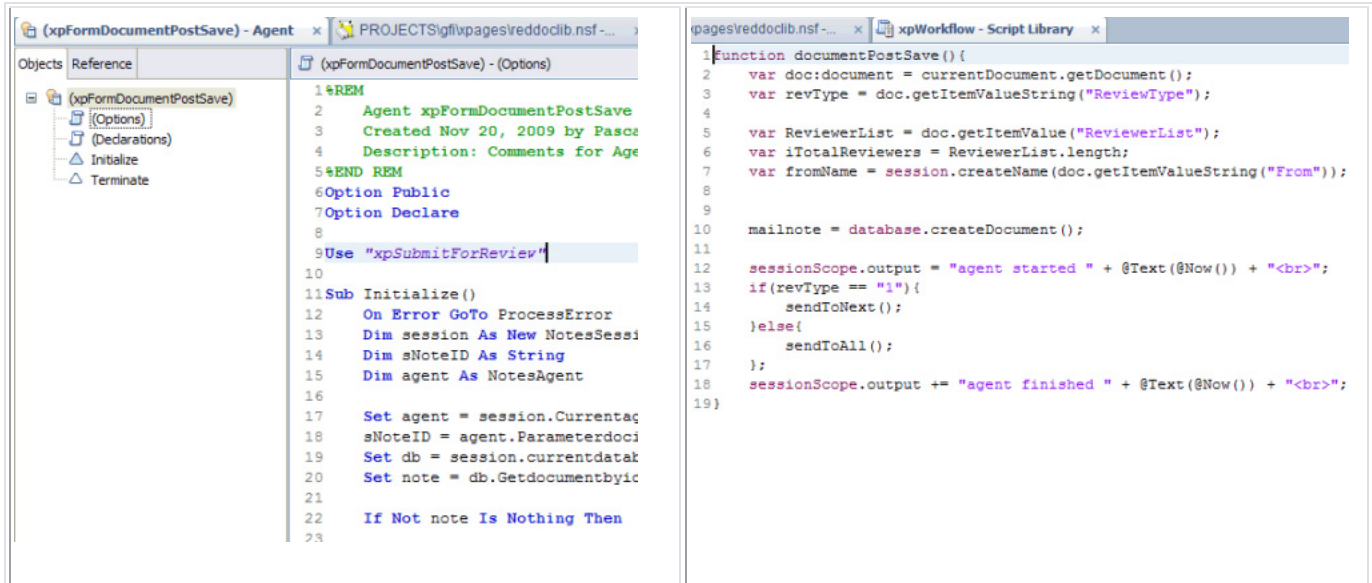
Create a main function in the script library that will handle the postSave processing.

In the XPage, include this Script library as a resource.

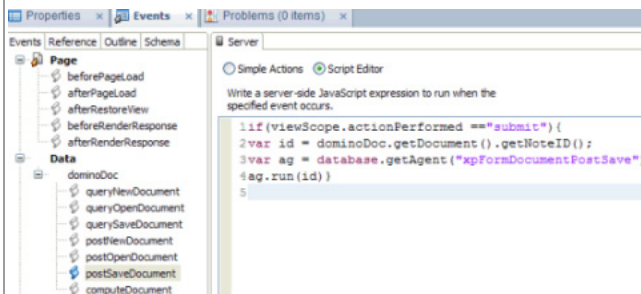
In the postSaveDocument event of the data source, you can now directly call the main function for processing the document.

The following table compares the LotusScript workflow and the Server Side JavaScript workflow side by side.

LotusScript agent	Serverside javascript script library
Starting point = a LotusScript agent (that eventually makes use of a LS script library)	Starting point = a SS JS script library, that contains the main function (cfr initialize in the LS agent)

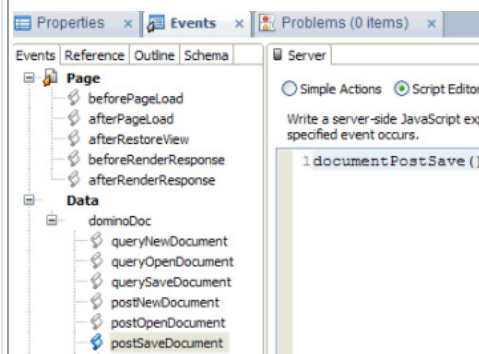
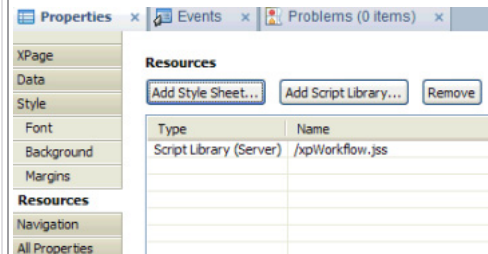


The agent is called from the XPage's Data Source postSaveDocument event:



The SS JS script library is included as resource in the XPage.

Then, its main function is called from the XPage's Data Source postSaveDocument event:



The code in LS and SS JS looks very similar as the same back-end classes are being used

<pre> 1Function SendToNext 2 If note.Status(0) = 1 Then 3 note.ReviewerNumber = 0 4 ReviewerNumber = note.ReviewerNumber(0) 5 'savenote as a copy of the original which we save as a response to the copy 6 If Not(note.HasItem("OriginalSaved")) Then 7 note.save True, True, True 8 Set savenote = New NotesDocument(db) 9 Call note.CopyAllItems(savenote, False) 10 'from the Notes client, the body field is not available to note until a ui s 11 If note.HasItem("Body") Then 12 Set ritem = note.GetFirstItem("Body") 13 savenote.RemoveItem("Body") 14 Call ritem.CopyItemToDocument(savenote, "Body") 15 Else 16 note.CopyBody = True 17 End If 18 Call savenote.MakeResponse(note) 19 savenote.ReviewerNumber = "1" 20 savenote.Subject = GetString(1) 21 savenote.save True, True, True 22 note.OriginalSaved = savenote.UniversalId 23 End If 24 Else 25 ReviewerNumber = note.ReviewerNumber(0) + 1 26 End If </pre>	<pre> 22function sendToNext() { 23 var sStatus = doc.getItemValueInteger("Status"); 24 25 if(sStatus=="1"){ 26 doc.replaceItemValue("reviewernumber",0); 27 iReviewerNumber = doc.getItemValueInteger("ReviewerNumber"); 28 //savenote as a copy of the original which we save as a response 29 if(!doc.HasItem("OriginalSaved")){ 30 doc.save(); 31 var savenote:document = database.createDocument(); 32 doc.copyAllItems(savenote,true); 33 if(doc.HasItem("Body")){ 34 var ritem = doc.getFirstItem("Body"); 35 savenote.removeItem("Body"); 36 ritem.copyItemToDocument(savenote, "Body"); 37 } else { 38 doc.CopyBody = true; 39 } 40 savenote.makeResponse(doc); 41 savenote.replaceItemValue("ReviewerNumber","1"); 42 savenote.replaceItemValue("Subject",GetString(1)); 43 savenote.save(); 44 doc.OriginalSaved = savenote.UniversalId; 45 } 46 }else{ 47 iReviewerNumber = doc.getItemValueInteger("ReviewerNumber")+1; 48 } 49 } 50 </pre>
--	--

The above table just gives an idea of how to get started with a Server Side JavaScript workflow. In this example, it doesn't add any value, but in fact, using Server Side JavaScript, would make it much easier to return values to the XPage.

Summary for Section – Sample 1 - Workflow

In this page, we have implemented the Notes client workflow in the XPage application. Only the Submit for Review action has been provided, but this should have been sufficient to demonstrate the concepts of adding a workflow to an XPage application, based upon an existing Notes client application. Also, the use of a Server Side JavaScript library to load global scoped variables has been explained, as well as using Server Side JavaScript as an alternative for LotusScript for workflow processing.

Create and display responses - XPage enabling an existing Notes client application

In the Notes client Document Library application, users can create responses to library documents. In this page, we will add the possibility to create and display responses to a main document within the XPage enabled application. Responses to a main document will be created and displayed on the same XPage as the main document. This will be achieved by using the new possibilities of XPages, like multiple data sources on one XPage, and repeat controls.

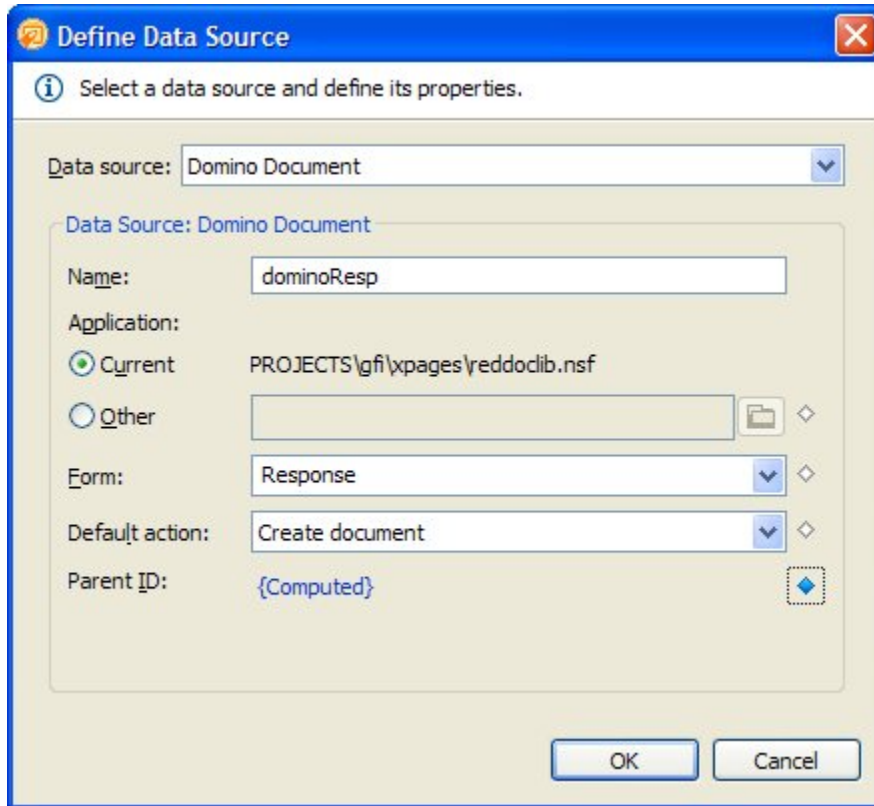
We will conclude this tutorial with a better way to display the main documents in read mode by using a separate custom control.

Create responses

Create the custom control ccFormResponse

Create a new Custom Control, **ccFormResponse**

Via the Data palette, define a data source:



Enter a computed value for the **Parent ID**: requestScope.parentID

Drag a panel container control in the custom control and give it the name "panelResponse".

Drag the Subject and Body fields from the data palette into the panel. Set the type of the Body field to Rich text, and include a Submit button. Change the button label to "Reply".

Add a title label "Reply" and apply some styling to the custom control elements (set width and apply OneUI style classes)

Save and close the custom control.

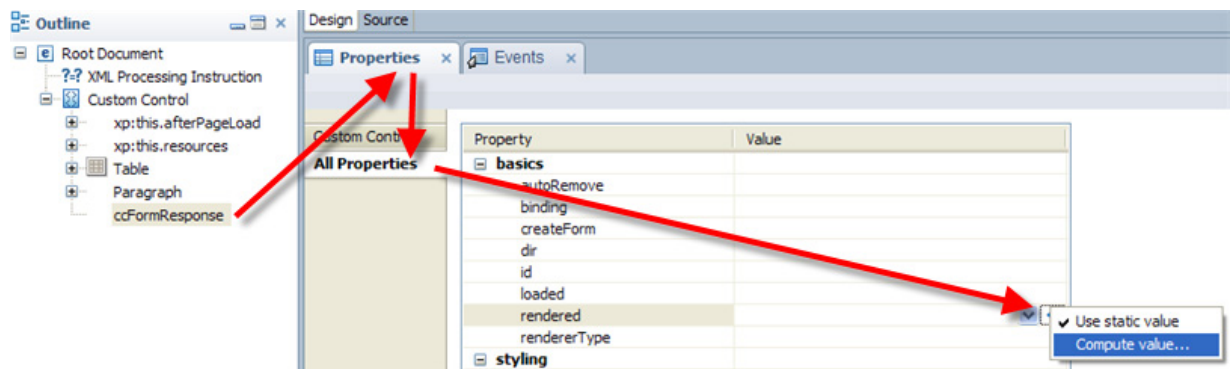
Include the custom control in ccFormDocument

Instead of displaying the response form on a different page in the web browser, we will display it straight under the main document form. This way, readers of the library document can immediately post a reply.

Open **ccFormDocument**

From the Controls palette, drag **ccFormResponse** and drop it on the very bottom of **ccFormDocument** (=under the debug info).

With **ccFormResponse** still selected within **ccFormDocument**, go to the tab **All Properties** in the Properties view, and enter a computed value for the "rendered" property:



Enter this Server Side JavaScript:

```
!dominoDoc.getDocument().isNewNote() && !dominoDoc.isEditable();
```

This will only display the response custom control if the main document is not a new document or if it is opened in edit mode.

Save and close ccFormDocument.

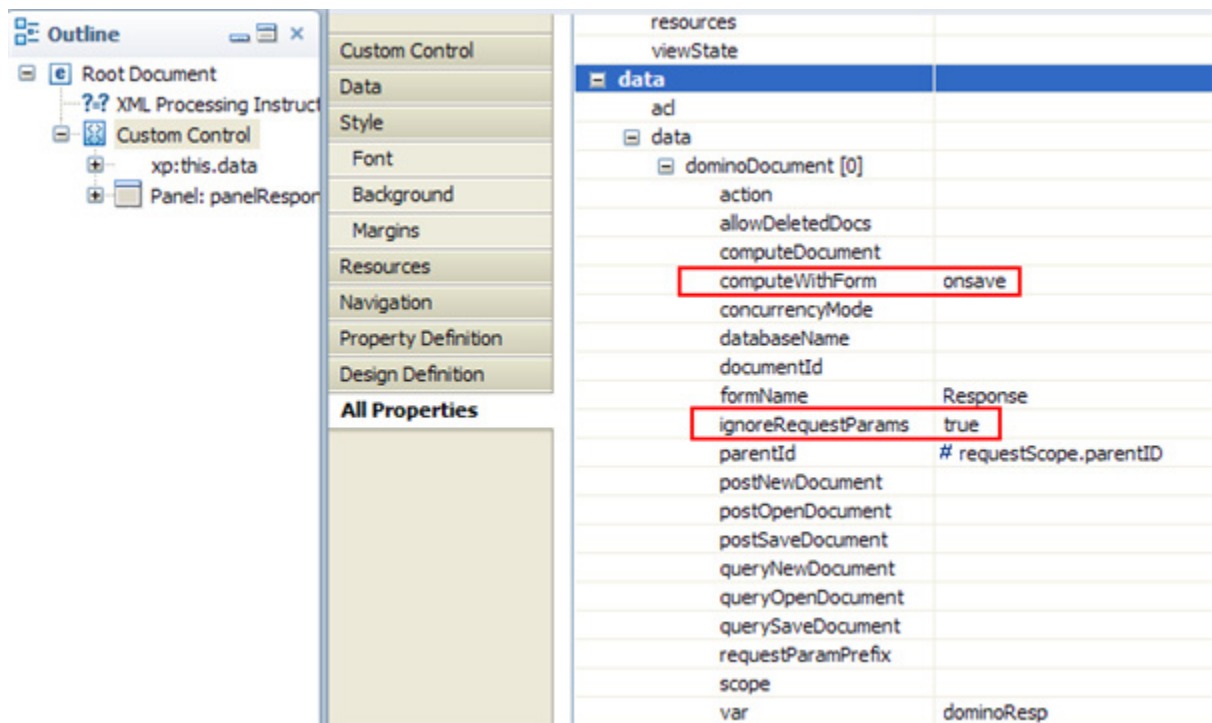
Finalize ccFormResponse

Now, we still need to do some finetuning in ccFormResponse:

Open **ccFormResponse**

Select the toplevel in the outline (= "Custom Control") and in the properties of the custom control, go to the All Properties tab.

Under data - data - dominoDocument[0], set the property **computeWithForm** to onsave, and set the property **ignoreRequestParams** to true.



It is essential to set the property **ignoreRequestParams** to true in order to work with multiple data sources and have one data source open in edit mode (=response) while the other stays in read mode (main document).

We also need to add some code to the "Reply" button:

In the onclick event, add this code:

```
dominoResp.getDocument().makeResponse(dominoDoc.getDocument())
```

In the Server Options, select "Partial Update" and select "panelResponse" as element to update.

Adapt the action buttons in ccFormDocument

Now, having 2 data sources on one XPage causes a side effect: the buttons of the type "Submit" in ccFormDocument save all data sources on the XPage. This means that, when a user clicks on the buttons "Save" or "Submit for Review" (both are of button type "Submit"), not only the main document is saved, but also the response form is submitted, creating a new response document (which will actually not be a response to the main document, because the makeResponse method was not called.) Event with ccFormResponse not being rendered, the data source is submitted.

The workaround for this, is to change the button type to "Button", and add an explicit save action to the button:

Select the "Save" button in ccFormDocument and change the button type from "Submit" to "Button". In the Events view, add two simple actions to the onclick event:

- Save Document
- Open Page: Previous Page

Also change the button "Submit for Review": change the button type to "Submit" and in the onclick event, add a simple action "Save Document" between the existing actions "Execute Script" and "Open page".

Display responses

When you test the current response form by opening a library document in read mode, and submitting a reply, you notice that a blank new response form is opened within the same Xpage. It looks as if the submitted contents is lost. Actually, it isn't: a new response document was created and it is visible in the viewAllDocuments.xsp. But since the "Reply" button in ccFormResponse performs a partial update (of panelResponse), the same form is loaded again.

This is meaningful on condition that the response that was just created is visible in the XPage. Therefore, we will now provide a means to display submitted responses in the XPage. We will make use of a repeat control for this.

Create a custom control to display responses

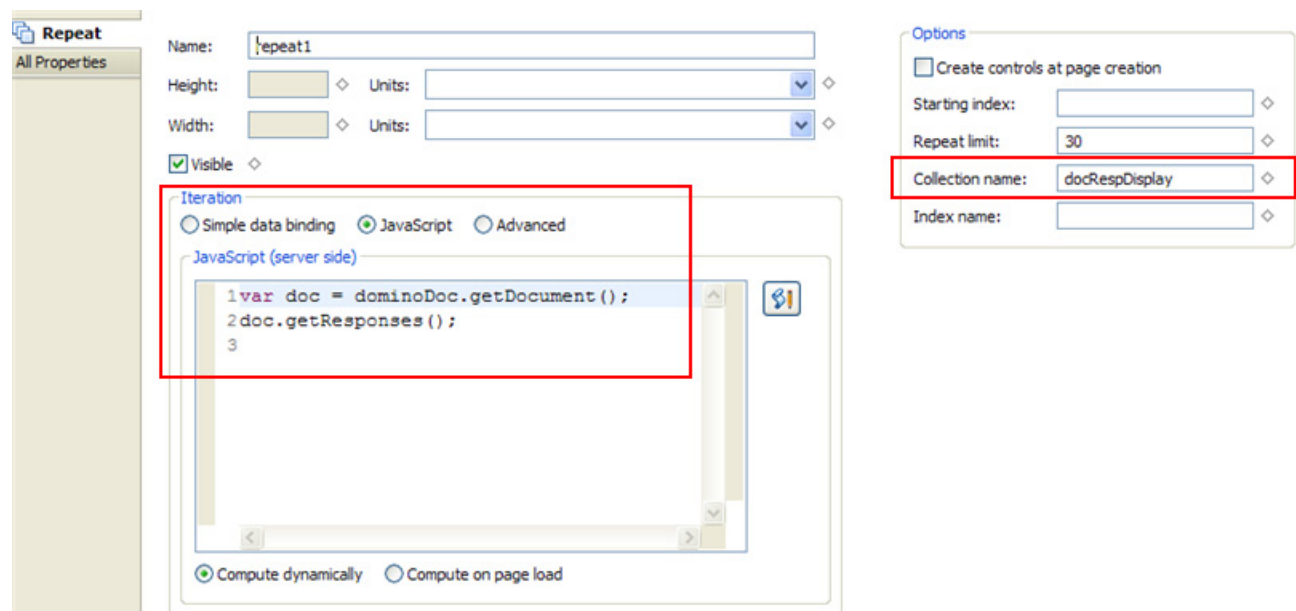
Create a new custom control and give it the name "ccDisplayResponses"

Drag a Repeat container control into this custom control.

In the repeat control properties, add a Server Side JavaScript for iteration:

```
var doc = dominoDoc.getDocument();
doc.getResponses();
```

Under "Options" set the value of Collection name to "docRespDisplay"



With repeat controls, you can easily display data of a list of documents in a layout you define yourself. In the iteration, you can use simple data binding and select the view that contains the documents to iterate in, or you can specify a JavaScript that returns either a list of NotesDocument objects or a list of Document ID's.

The Collection name is the name of the instance variable you will use to display individual values.

Drag a panel into the Repeat control and call it "panelRepeat"

Set the panel to Read-only, as it will only contain display information (Properties view, first tab, under visibility).

Drag a computed field into the panel, and call it "respSubject". It will be used to display the subjects of

the responses.

Give it the following value:

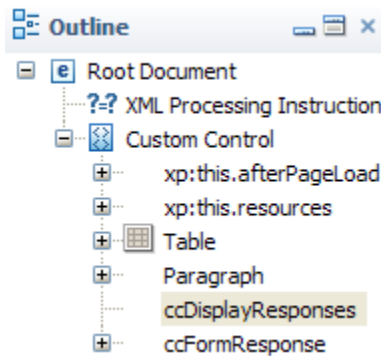
```
docRespDisplay.getItemValueString("Subject")
```

Save and close ccDisplayResponses.

Add the response list to ccFormDocument

Open ccFormDocument and drag ccDisplayResponses above ccFormResponse.

The Outline view of ccFormDocument now looks like this:

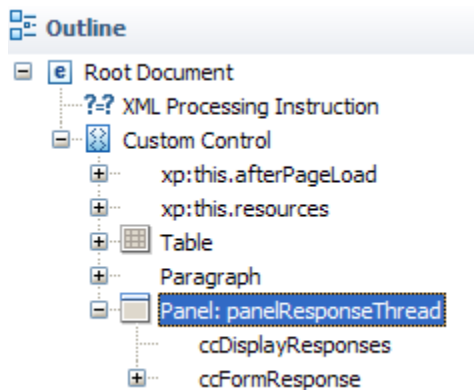


If you now would open an existing document that has multiple responses, you would see the subjects of each reply listed.

In order to immediately see newly added replies within ccFormDocument, we need to create a container that contains both the ccDisplayResponses and ssFormResponse, and when submitting a reply, perform a partial update of this container.

In ccFormDocument, add a new panel container control on the bottom and name it "panelResponseThread".

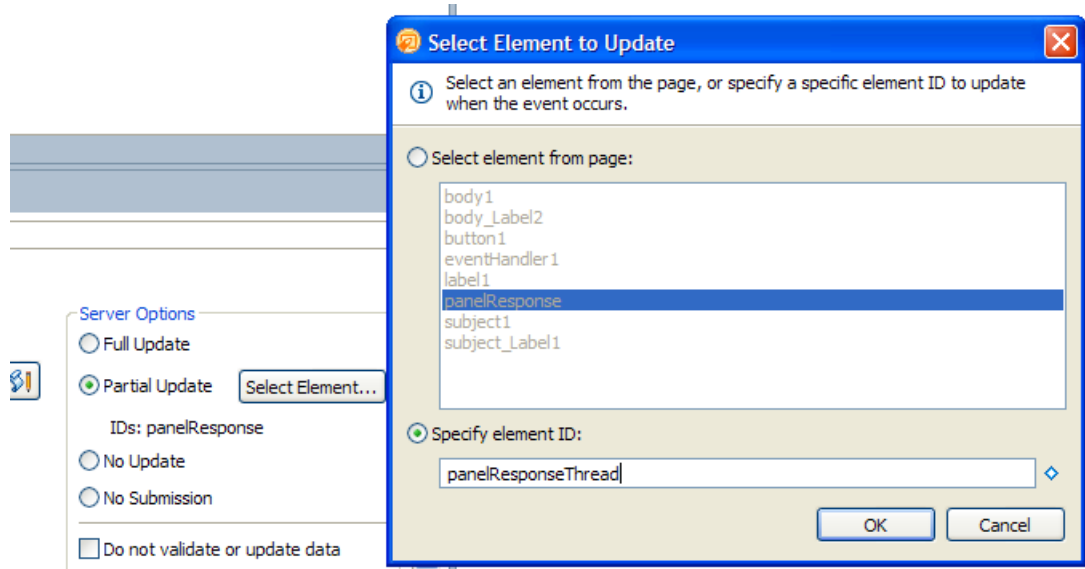
Via the Outline view, drag both ccDisplayResponses and ccFormResponse into this panel:



Save and close ccFormDocument.

Open ccFormResponse and select the "Reply" button.

In the onclick event, change the element for Partial Update: instead of selecting an element on the page, choose "Specify element by ID" and enter panelResponseThread:



Save and close ccFormResponse.

Test the result. When you now submit a reply to an existing library document, it's subject is immediately displayed in the list.

Enhance response display

Now we will add more fields to the response display list and present them in a neater way.

In ccDisplayResponses, add more computed fields:

Add a computed field above the subject, and make it return the name of the author of the reply:

```
var aName = session.createName(docRespDisplay.getItemValueString("From"));
aName.getCommon();
```

Under the subject, add another computed field with the date and time of creation as value

```
docRespDisplay.getCreated().getDateOnly() + " " +
docRespDisplay.getCreated().getTimeOnly()
```

Add a third computed field to display the body contents. Since the body of the reply is rich text, it have to be converted to text. In order to do that, add a hidden computed field on the Response form in Notes (=the old-school Notes form) with the name "BodyText" and the value @Abstract([Rule1];256;"";"Body").

The third computed field in the repeat control in ccDisplayResponses can now have as value:

```
docRespDisplay.getItemValueString("BodyText")
```

Now, add some styling by adding container controls (add <div> tags in the Source view) and assigning them OneUI style classes of your choice.

For example, the Source code of panelRepeat could look like this:

```
<xp:panel id="repeatPanel" readonly="true">
  <div class="lotusForum">
    <div class="lotusPost">
      <div class="lotusPostAuthorInfo">
        <div class="lotusPostAvatar">
          <xp:text escape="true" id="computedField2">
            <xp:this.value>
          <![CDATA[#{javascript:var aName =
session.createName(docRespDisplay.getItemValueString("From"));
aName.getCommon();}]]></xp:this.value>
          </xp:text>
        </div>
      </div><!--end author info-->
      <div class="lotusPostContent">
        <h4>
          <xp:text>
            <xp:this.value>
          <![CDATA[#{javascript:docRespDisplay.getItemValueString("Subject")}]]></xp:this.value>
          </xp:text>
        </h4>
        <div class="lotusMeta">
          <xp:text>
            <xp:this.value>
          <![CDATA[#{javascript:docRespDisplay.getCreated\(\).getDateOnly() + " " +
docRespDisplay.getCreated().getTimeOnly()}]]>
          </xp:this.value>
          </xp:text>
        </div>
        <div class="lotusPostDetails">
          <xp:text escape="true"
            id="computedField1">
            <xp:this.value><![CDATA[#{javascript:docRespDisplay.getItemValueString("BodyText")}]]>
          </xp:text>
        </div>
      </div>
    </div>
  </div>
</xp:panel>
```

```
</xp:this.value>
                                </xp:text>
                            </div>
                        </div><!--end postContent-->
                    </div>
                </div>
            </xp:panel>
```

The result looks like this:

The screenshot displays a Domino web application interface. At the top, there are two comment boxes, each with a blue header bar containing the name 'Pascal David'. The first comment box contains the text 'Great document!', the date '07/12/2009 14:36:11', and the text 'Very comprehensive document. Thanks!'. The second comment box contains the text 'XPages are powerful', the date '15/12/2009 01:28:58', and the text 'Thanks for showing me'. Below the comment boxes is a 'Response' section. It features a 'Subject:' label followed by a text input field. Below that is a 'Body:' label followed by a rich text editor toolbar. The toolbar includes icons for undo, redo, bold, italic, underline, strikethrough, bulleted list, numbered list, link, unlink, insert image, and a font color selector. Below the toolbar is a 'Size' dropdown menu.

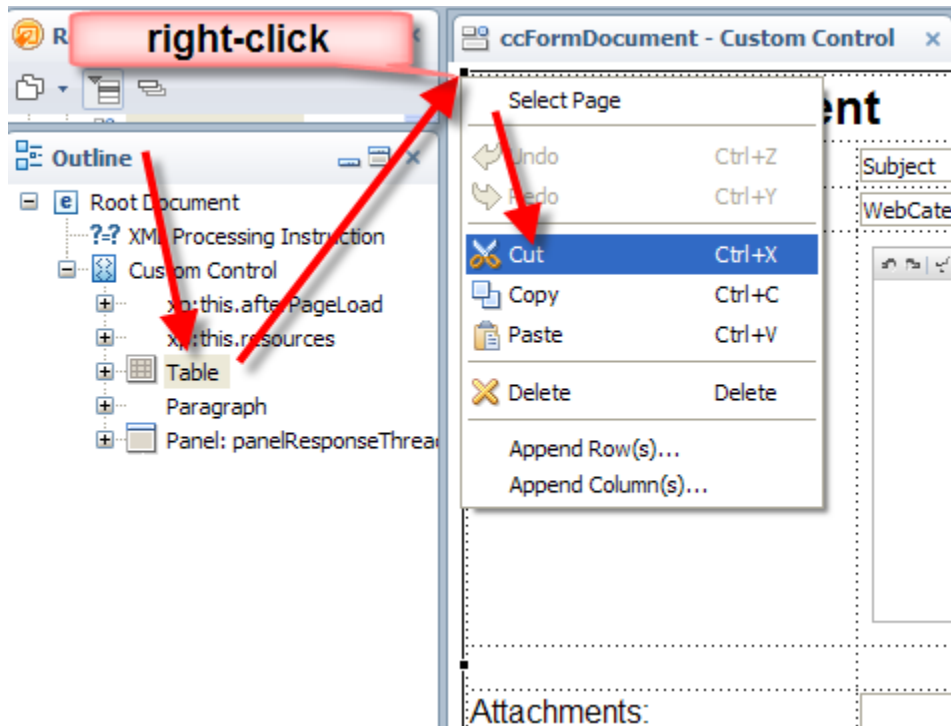
Read and edit mode custom controls

So far, the main library document input form has been optimized for new data entry / editing, and not for viewing in read mode. It looks far from elegant in read mode. In order to have more control over the look and feel of the library document in read mode, without interfering with the input form, separate custom controls will be created to display the document in edit and read mode, and the edit and read custom controls will be rendered conditionally then.

Create separate custom controls

Create a new custom control, **ccFormDocumentEdit**

Open the existing **ccFormDocument** and cut the table containing the main document fields. You can select this table in the Outline view, and once it is selected, you can right-click on the edge of the table in the editor and use the right-menu option "cut".



Paste the table in the blank ccFormDocumentEdit and save the custom control.

In ccFormDocument, drag ccFormDocumentEdit into the edit area, on top: it now replaces the form table. The result is identical as before.

Save ccFormDocument.

Create a new custom control: **ccFormDocumentRead**

Drag a panel container control in it.

Add a text label to this panel "Library document", just to have something visible. We will add more content further on.

Save ccFormDocumentRead.

Display custom controls conditionally

Open ccFormDocument and drag ccFormDocumentRead in the custom control, above ccFormDocumentEdit.

Select ccFormDocumentRead in the Outline and in the properties, add a computed visibility:

```
!dominoDoc.isEditable();
```

Do the same form ccFormDocumentEdit, but set it to display in edit mode:

```
dominoDoc.isEditable();
```

The main document is now displayed with the appropriate custom control in function of read/edit mode.

Display content in ccFormDocumentRead

In order to display data in controls, we need to bind the controls to the underlying document. Data binding can be either done globally in the Data palette, in the Custom Control properties (data tab), or individually for each data display control.

Defining the data source globally makes it easy to do the binding for the individual controls, because you can simply select the data source from the dropdown. The only drawback is that you need to remove the data source afterwards in order to make the edit button work: the data source needs to be defined at XPage level, instead of at custom control level, or the users will have to click twice on the edit button (this was discussed earlier, on the page about Form Design)

We will define the data source globally, and remove it later.

In the Custom Control properties, Data tab, add a data source of type "Domino Document" and give it the same name as the data source defined in the XPage formDocument: **dominoDocReason**: we will remove this data source later on, and just need a way to set-up the right binding for our individual controls.

Leave the default action to "Create Document". This doesn't make much sense (you would expect to make it "Open Document", but then you would need to specify a document ID.)

Now you can add content in the form of computed fields:

Add a computed field **cfFormTitle** and bind it to bind to **dominoDoc.Subject** in the Source view, place <h2> tags around this field:

```
<h2>
  <xp:text escape="false" id="cfFormTitle"
    value="#{dominoDoc.Subject}" />
</xp:text>
</h2>
```

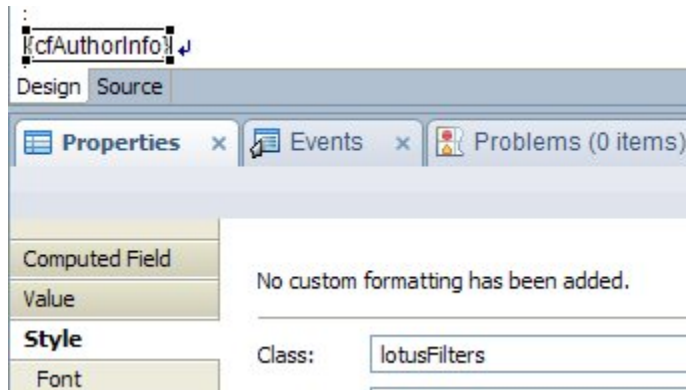
Add an additional computed field, under cfFormTitle, and call it **cfAuthorInfo**.

Change the type to "Html". This allows us to write html in the value:

```
var aName =
session.createName(dominoDoc.getDocument().getItemValueString("From"));
"By <a class='lotusFilter'>" + aName.getCommon() + "</a> on " +
dominoDoc.getDocument().getCreated().getDateOnly();
```

The idea is that the author name is clickable and when clicking on it, a list of documents from this author is displayed (the actual link has not been added yet: to be completed later on),

Add a Style class to the computed field:



Additionally, add a rich text field to the panel and bind it to **dominoDoc.Body**.

In order to make this field read-only, we could set the Readonly flag in the Rich Text properties, but it is better to set this property globally, at custom control level:

Custom Control Properties: data: data: readonly: true.

Add a file download control to display attachments.

Add action buttons to ccFormDocumentRead

Move the Edit button from ccFormDocumentEdit to ccFormDocumentRead and copy and paste the Cancel link.

In order to make the edit button work, it has to be linked to the data source defined within the XPage, rather than the one defined in the custom control. (=limitation in Domino 851, discussed before).

Therefore, remove the data source on ccFormDocumentRead.

The edit button will now work.

Summary for Section – Sample 1 - Create and Display Responses

This section explained how to work with different data sources in one XPage, and how to display information from a collection of response documents using a repeat control. And finally, we added a different interface to display the document form content in read mode.

Sample 1 - Tutorial Summary

This tutorial for Sample 1 ends here. It demonstrated how to web enable an existing Notes client application by using XPages.

The XPage Document Library sample application clearly is not completed yet: views need to be added, the workflow needs to be completed, title bar navigation added, etc... Rather than delivering a read-for-use application, this tutorial intended to provide you with all major stepstones to web enable your own Notes client applications.

Introduction to Sample 2: Building a new XPage web application from scratch

Introduction

In this section we are going to build a web application from scratch using XPages and the associated design elements in Domino 8.5.1. The goal for this exercise is to highlight the best practices of building Domino web application and to explain the rationale for using specific design decisions and approaches - as opposed to demonstrating various features of XPages. This section is written in a step-by-step tutorial format along with accompanying screenshots.

Many corporate Intranets provide a searchable list of documents for employees to download, such as policy documents, hand book, PDF forms, etc. In the sample application we are going to build the layout for a corporate Intranet and implement only the “Documents” section of the Intranet. Documents section allows users to upload, view, edit, search and download company documents. This provides a comprehensive exercise for users to learn and to expand upon. Below are a few screenshot of the final application. Note that “**Documents**” tab is highlighted from the global navigation – as this is the only section of the Intranet which is implemented in this exercise. Other tabs, such as **Announcements**, **Employee Directory** and **PO System** are included to demonstrate the context.

The screenshot shows the AAC Intranet Documents page in Mozilla Firefox. The page has a blue header with the AAC Intranet logo and navigation links: Home, Documents, Announcements, Employee Directory, and PO System. A user profile for Muhammad Sabir is visible in the top right corner with links for Help, About, and Log Out. Below the header, there are tabs for Documents and Collections, and buttons for New Document and Delete Selected. A left sidebar contains filters for By Subject, By Author, By Date, and By Tag. The main content area displays a list of documents with columns for Subject, Description, From, Date, and Size. A search bar is located in the top right corner. A tips box on the right side provides instructions on how to view and download documents.

Global Navigation

Utility Links

Secondary Navigation

Action Bar

Search

Left Navigation

Subject	Description	From	Date	Size
IT Service Management Policy Manual	This manual describes the ITSM System and delineates authorities, interrelationships and responsibilities within the system.	Anonymous	Jan 4, 2010	4818.0
Local and Long Distance Travel Policy	Guidelines for travel reimbursement.	Anonymous	Jan 8, 2010	5638.0
IT Services - Service Level Agreement	This document describes the SLA provided by IT to its customers	Anonymous	Jan 4, 2010	4818.0
IT Service Management Roles and Descriptions	This document describes ISO 20K IT Services Management (ITSM) roles and descriptions.	Anonymous	Jan 4, 2010	4818.0
Holiday Schedule - 2010	Note: Employees must be in an active full-time status for the pay period to be eligible for holiday pay.	Anonymous	Jan 4, 2010	15.0
Employee Referral Program	Company will give you \$2,000 when a candidate you have referred is hired. See attached document for details.	Anonymous	Jan 8, 2010	5532.0
Direct Deposit Authorization	Authorization Agreement for direct deposits	Anonymous	Jan 8, 2010	5643.0

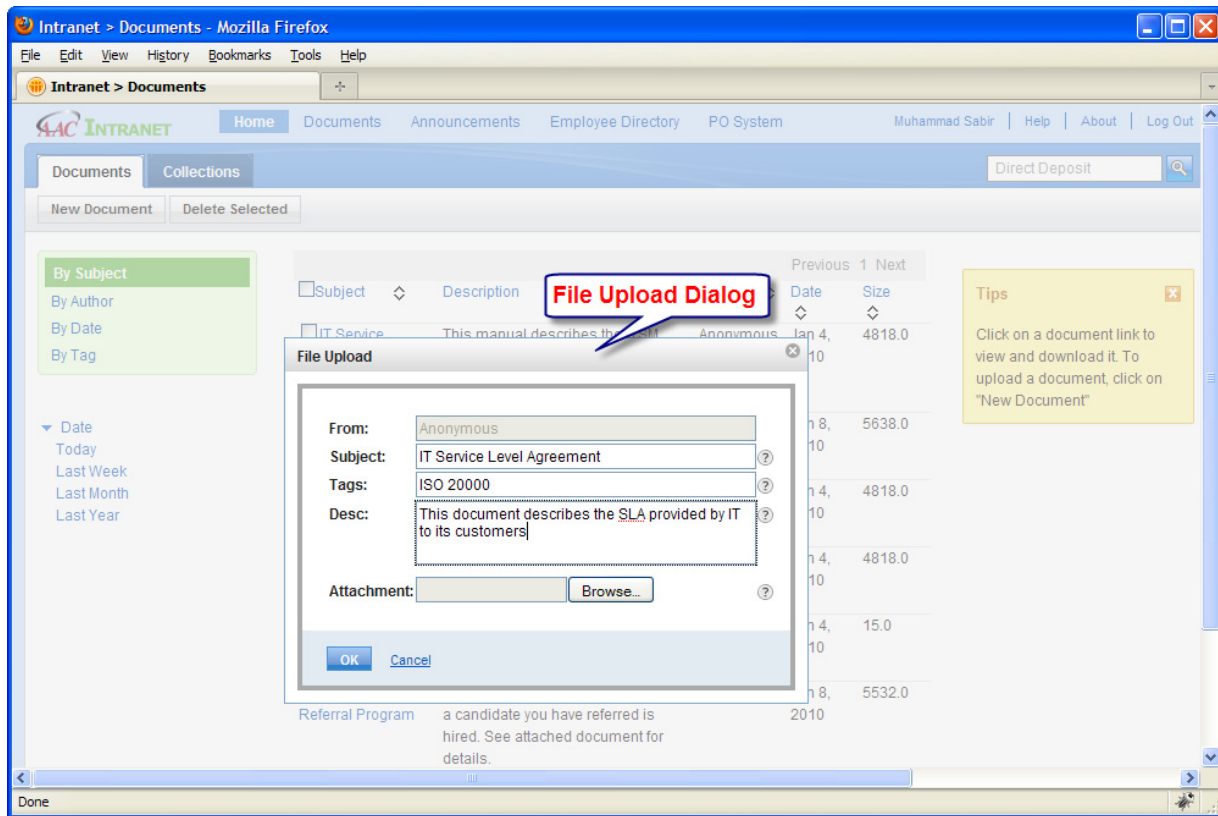
The screenshot shows the AAC Intranet Documents page with search results for 'Direct Deposit'. The search bar in the top right corner contains the text 'Direct Deposit'. Below the search bar, a box titled 'Search results for: Direct Deposit' displays a single result. The result is a document titled 'Direct Deposit Authorization' with a description 'Authorization Agreement for direct deposits', from 'Anonymous', dated 'Jan 8, 2010', and a size of '5643.0'. The left sidebar and other navigation elements remain the same as in the previous screenshot.

Search Results

Search Input

Search results for: Direct Deposit

Subject	Description	From	Date	Size
Direct Deposit Authorization	Authorization Agreement for direct deposits	Anonymous	Jan 8, 2010	5643.0



Step 1 – Sample 2: Selecting a CSS Framework

A CSS Framework is a collection of pre-built Cascading Style Sheets designed for a specific layout and styling of web applications. These frameworks are usually well documented, configurable and tested for popular browsers. A CSS Framework provides these advantages:

- Using pre-built CSS classes for layout and styling allows developers to focus on the core functionality – as opposed to the look and feel. This results in reduced development time.
- Consistent look and feel across various web applications results in improved user experience and reduced training needs.
- CSS Frameworks are designed and tested to accommodate various browser versions and screen resolutions.

Some of the drawbacks of using a CSS framework include:

- There is a learning curve involved to understand a framework and its configuration options.
- You can be potentially locked into a specific layout and navigation structure.

Some of the popular CSS Frameworks are listed below:

- Blueprint: <http://www.blueprintcss.org/>
- Elements: <http://elements.projectdesigns.org/>
- YUI 2 (from Yahoo): <http://developer.yahoo.com/yui/grids/>
- One UI (from IBM): <http://www-12.lotus.com/ldd/doc/oneuidoc/docpublic/index.htm>

Usually the benefits of using a CSS framework outweigh its drawbacks; therefore, CSS frameworks are increasingly used as a best practice for developing web applications. In order to demonstrate this best practice, the sample application will use One UI framework. The reasons for selecting One UI include:

- IBM Lotus Domino 8.5 Server and Designer ship with One UI framework. Since all the resources for the One UI framework (stylesheets and images) are already accessible to Domino Server and Designer, there is no need to import them separately for each application -- which makes it easier to use this framework within Domino applications.
- One UI is becoming a standard UI for all IBM Lotus brand products such Quickr, Connections, WebSphere Portal. By using it for Domino applications further increases the consistency. Furthermore, it comes with comprehensive documentation and sample pages.

Version 8.5.1 of Domino server and designer client includes two versions of One UI framework - version 1 and version 2. Below is the location of resources for these versions related to Domino data directory:

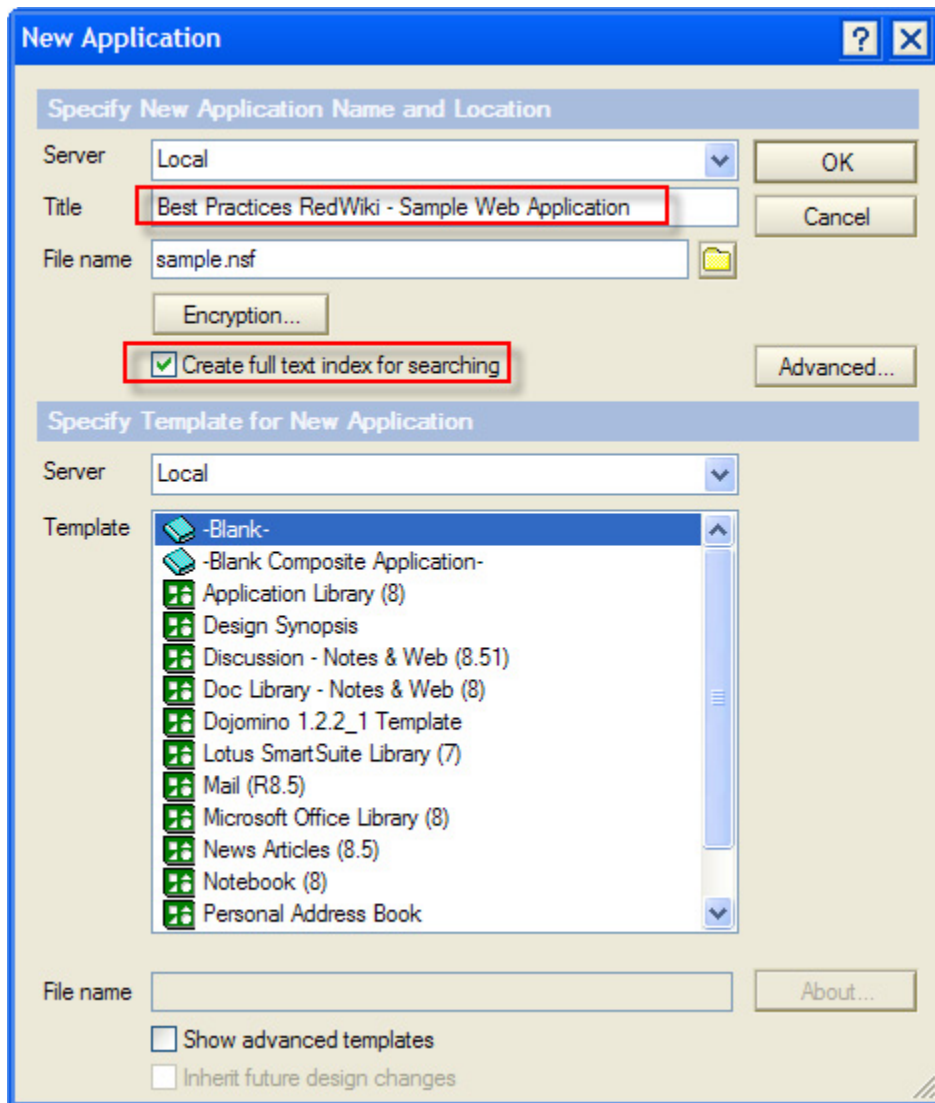
- Version 1: (data)\domino\java\xsp\theme\oneui
- Version 2: (data)\domino\html\oneuiv2

In the sample application, we will use version 2 of One UI framework.

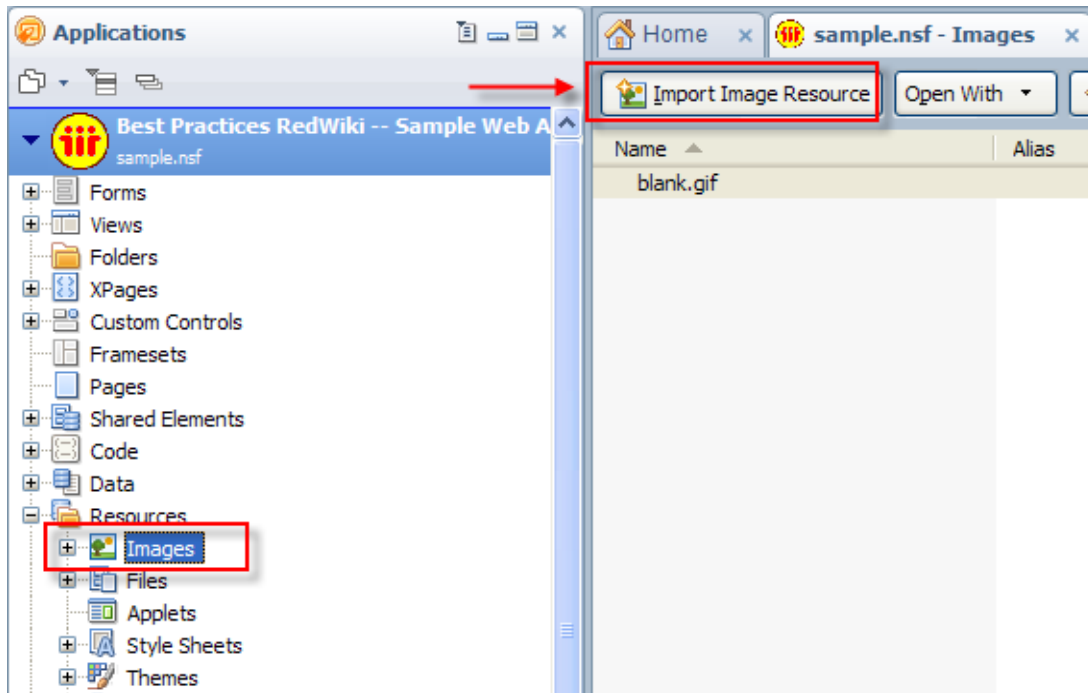
Step 2– Sample 2: Creating the database and importing images

Let's get started with the development tasks.

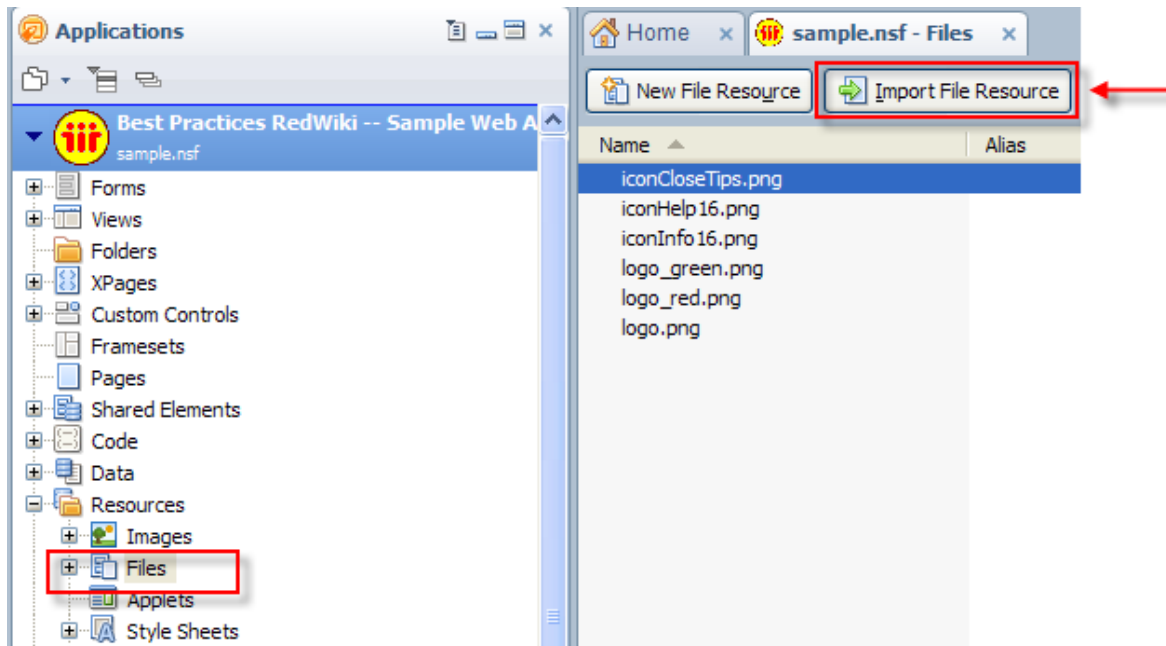
Open Domino Designer and create a new database by selecting **"File > New Application"** from the menu or by pressing **"Ctrl-N"**. Select a server from drop down and give your application a title and a file name. Check **"Create full text index for searching"** as we are going to use it for search feature within this sample application.



Next, we need to import the image resources for this sample application. From the application navigator, expand the **"Resources"** section and click on **"Images"**. Click on **"Import Image Resources"** to import **"blank.gif"** file.



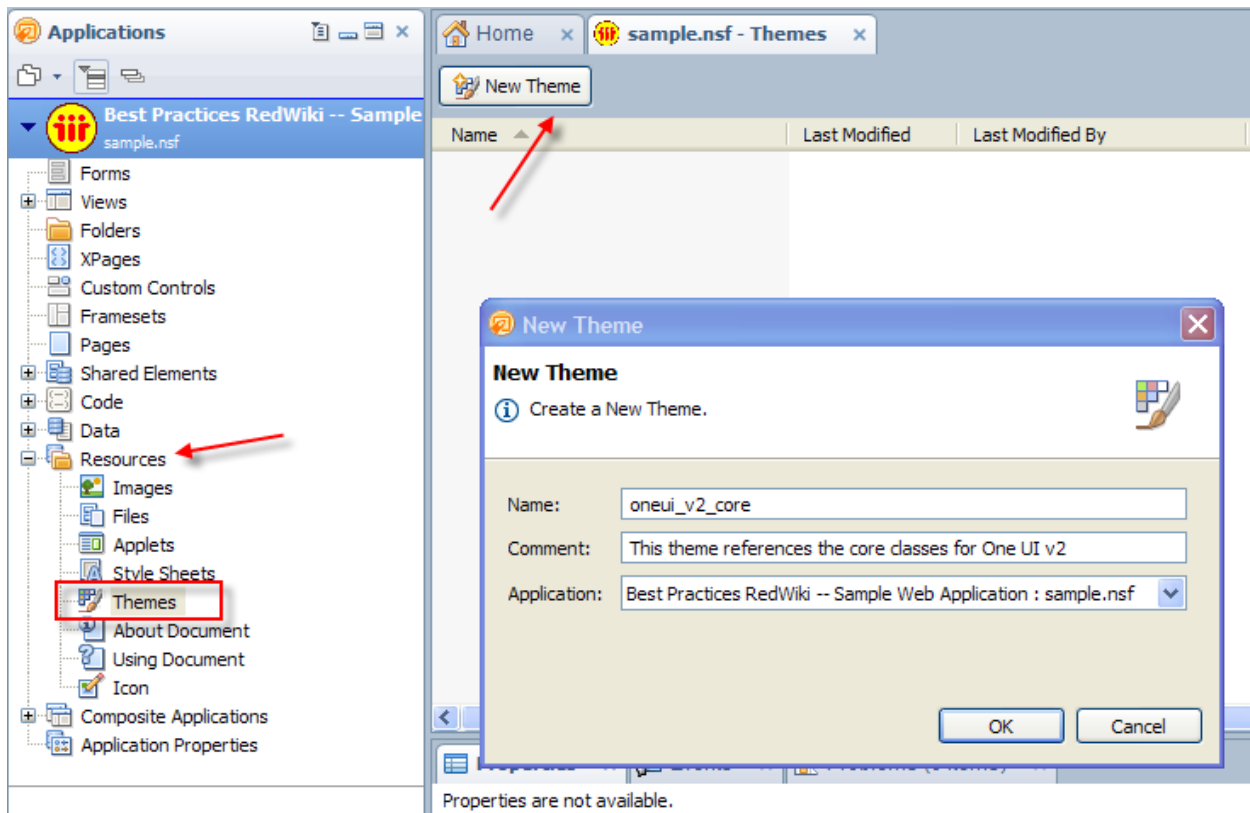
Current version of Domino Designer doesn't allow importing .png as image resources. For PNG files, we need to import them as file resources. From "Files" resources, click on "Import File Resource" to import all .png files.



Step 3 – Sample 2: Incorporating the CSS Framework using Themes

Now that we have selected a CSS framework, the next step is to develop the user interface layout using One UI framework and themes. Theme is a design element available in Domino designer and it provides a way to quickly change the look and feel of the web application without any coding. Theme is an XML file where you describe what resources (such as CSS and images) are available to your application implicitly. You can also specify implicit properties for various controls. For example, you can specify that all View Panels are going to be styled by a specific CSS class – so you don't have to assign any CSS classes to View Panel within your XPage or Custom Control. You can have multiple themes available for your web application and decide what theme to activate at deployment.

Since we are going to use One UI CSS framework, we need to make sure that appropriate resources are available to all XPages and custom controls. For this, we need to create themes. Expand the **"Resources"** section and select **"Themes"** element and click on **"New Theme"** button:



Enter **"oneui_v2_core"** as the name and optional comments as **"This theme references the core classes for One UI v2"**. Click OK. You can read through the default theme file that Domino creates to understand the structure of this XML file:

```

<!--
The default theme contents use theme inheritance.

Application themes can extend an existing global theme using the
extends attribute. Existing themes include the following options:

1. webstandard 2. oneui 3. notes
-->
<theme extends="webstandard">
  <!--
    Use this pattern to include resources (such as style sheets
    and JavaScript files that are used by this theme.
  -->
  <!--
  <resource>
    <content-type>text/css</content-type>
    <href>mystylesheet.css</href>
  </resource>
  -->

  <!--
    Use this pattern to define property name/value pairs for controls
  -->
  <!--
  <control>
    <name>[Control Name]</name>
    <property>
      <name>[property Name]</name>
      <value>[property Value]</value>
    </property>
  </control>
  -->
</theme>

```

Design Source

Note that you can extend an existing theme using “**extends**” attribute. Domino 8.5 ships with three global themes: **webstandard**, **oneui** and **notes**. Since we want to use One UI theme in our application, we can simply extend it. However, there are two versions of One UI included in Domino 8.5.1 – version 1 and version 2. If you extend “oneui”, you are extending version 1 (because version 2 is included in Domino 8.5.1 for evaluation purposes). Therefore, instead of extending a theme, we need to create a theme from scratch. This also serves as an example of how to create themes.

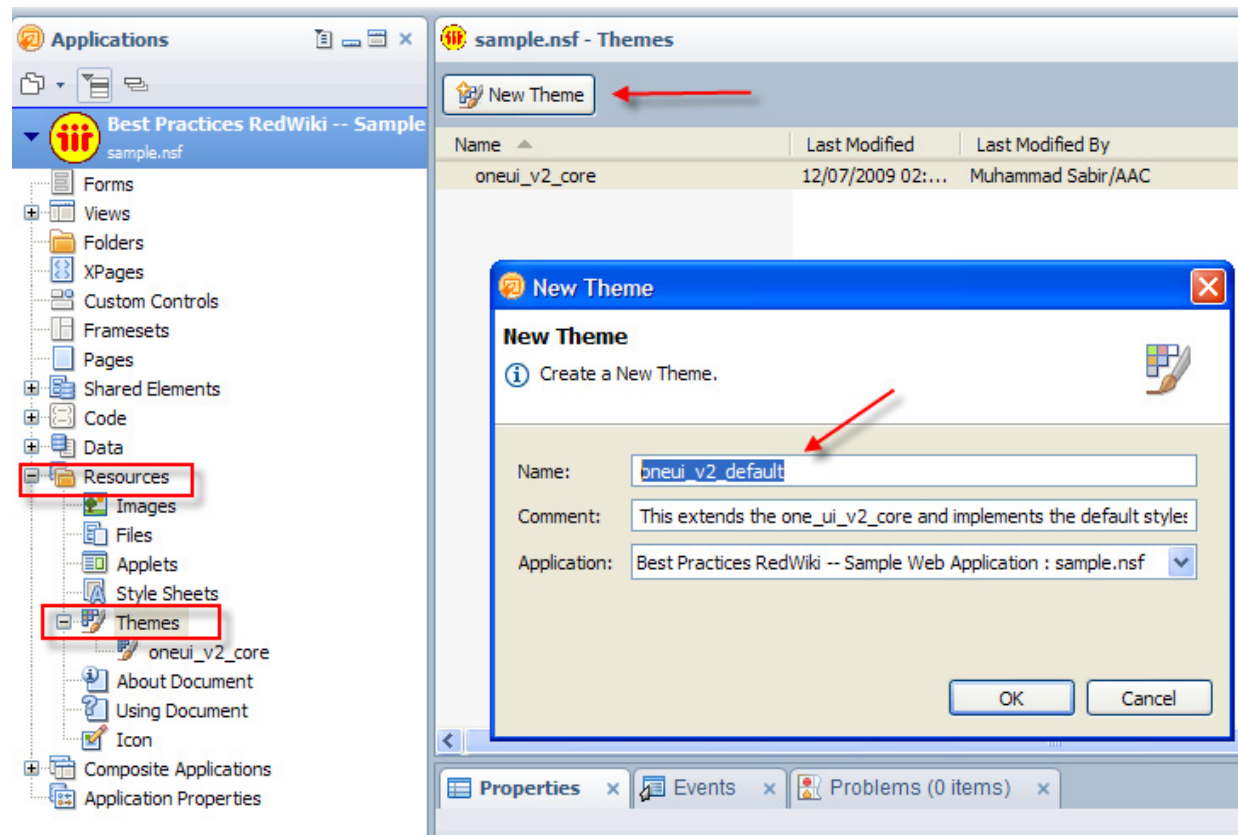
Remove the existing code and enter the following code in the theme file:

```
<theme>
  <!-- One UI v2.0.1, Core -->
  <resource>
    <content-type>text/css</content-type>
    <href>/./ibmjspxres/domino/oneuiv2/base/core.css</href>
  </resource>
</theme>
```

In the code above, we are including core.css stylesheets in our theme. Instead of hard coding the location of resources, you can use the following mapping:

Reference path	Mapped path
/ibmjspxres/global/theme/oneui/	/domino/java/xsp/theme/oneui
/ibmjspxres/domino/	<data>/domino/html/
/ibmjspxres/dojo/	<data>/domino/js/dojo-1.3.2/

One UI v2 comes with various themes, with similar positioning of elements but each with a bit different color combination and background graphics. We are going to use the **default** (blue) and **metal** themes only for this sample application. We can implement these two themes by extending the core theme. Click on “**New Theme**” button and enter “**oneui_v2_default**” as the theme name:



Remove the default text and enter the following XML code:

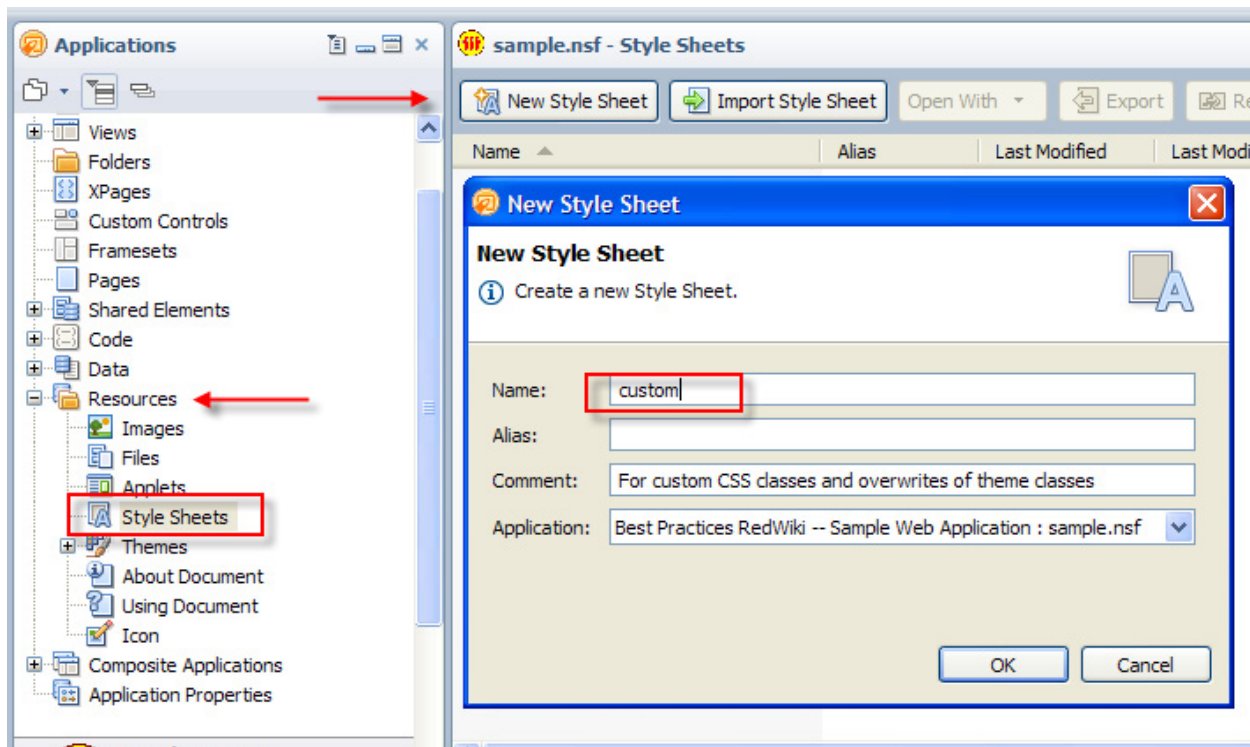
```
<theme extends="oneui_v2_core">

  <!-- One UI v2.0.1, Default Theme -->
  <resource>
    <content-type>text/css</content-type>
    <href>/./ibmjspxres/domino/oneuiv2/defaultTheme/defaultTheme.css</href>
  </resource>

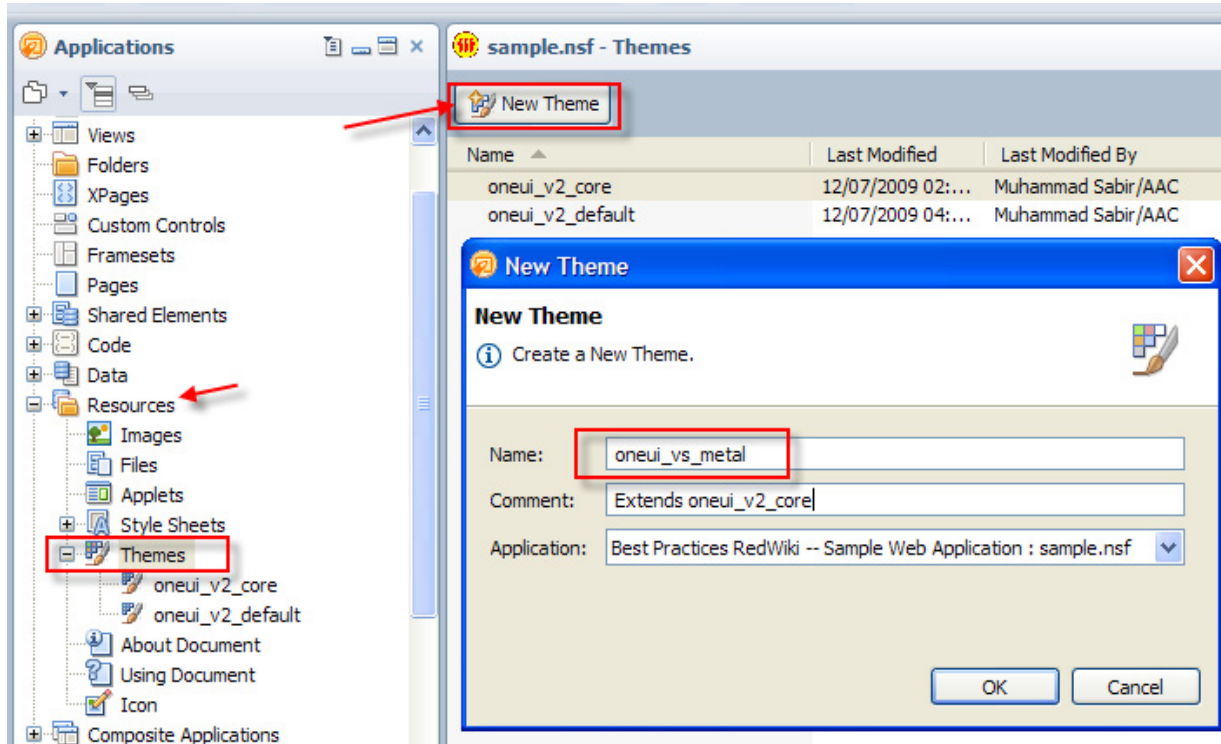
  <!-- Custom CSS for overrides or additional styles -->
  <resource>
    <content-type>text/css</content-type>
    <href>custom.css</href>
  </resource>

</theme>
```

By extending “**oneui_v2_core**”, this theme will inherit all the resources referenced in the core theme. We have added two additional references – **defaultTheme.css** and **custom.css**. DefaultTheme.css contains all the style related to the default (blue) theme included in One UI v2. Custom.css is referenced for additional styles specific to our application. Since this file does not exist yet, we need to create it. Click “**Style Sheets**” under Resources and click “**Create New Stylesheet**”. Enter “**custom**” as the file name. (You don’t need to enter .css). Close the newly created file. We are going to use it later for any custom styling used within the sample application.



We have created the default (blue colored) theme. Let's create a metal theme for alternate styling. Click on **"New Theme"** button and enter **"oneui_v2_metal"** as the theme name:



Remove the default text and enter the following XML code:

```
<theme extends="oneui_v2_core">

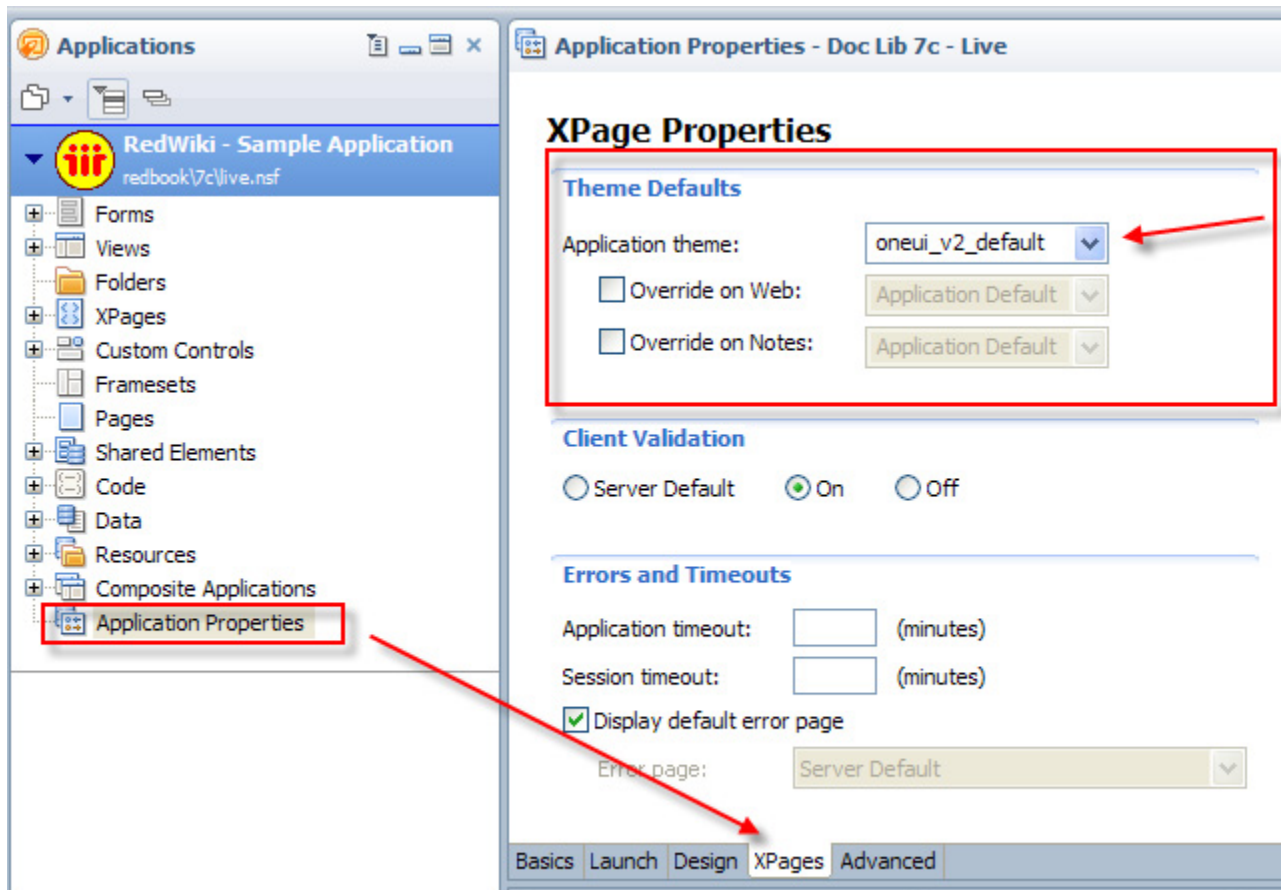
  <!-- One UI v2.0.1, Metal Theme -->
  <resource>
    <content-type>text/css</content-type>
    <href>/./ibmxspres/domino/oneuiv2/metalTheme/metalTheme.css</href>
  </resource>

  <!-- Custom CSS for overrides or additional styles -->
  <resource>
    <content-type>text/css</content-type>
    <href>custom.css</href>
  </resource>

</theme>
```

By extending **"oneui_v2_core"**, this theme will inherit all the resources referenced in the core theme. We have added two additional references. **MetalTheme.css** contains all the style related to the metal theme included in One UI v2. **Custom.css** is referenced for additional styles specific to our application.

As mentioned earlier, developers can provide multiple themes within an application but only one of them can be active at a time. To activate a theme select the desired theme from the **"Application Theme"** drop down in **"XPages"** tab of **"Application Properties"** file, as shown in the screenshot below:



From the application properties, select “**oneui_v2_default**” theme and check “**display default error page**” (This option will display the default error page in case of any runtime error with XPages or custom controls)

Step 4 – Sample 2: Understanding the web application layout

Designing the layout involves looking at the web application user interface from high-level and identifying repeatable sections (or placeholders) such as header, footer, content, navigation and so on. Since we are implementing One UI, the high-level breakdown of UI into sections is pre-determined. One UI divides the overall UI into five sections and a container section as listed below and shown in the following screenshots:

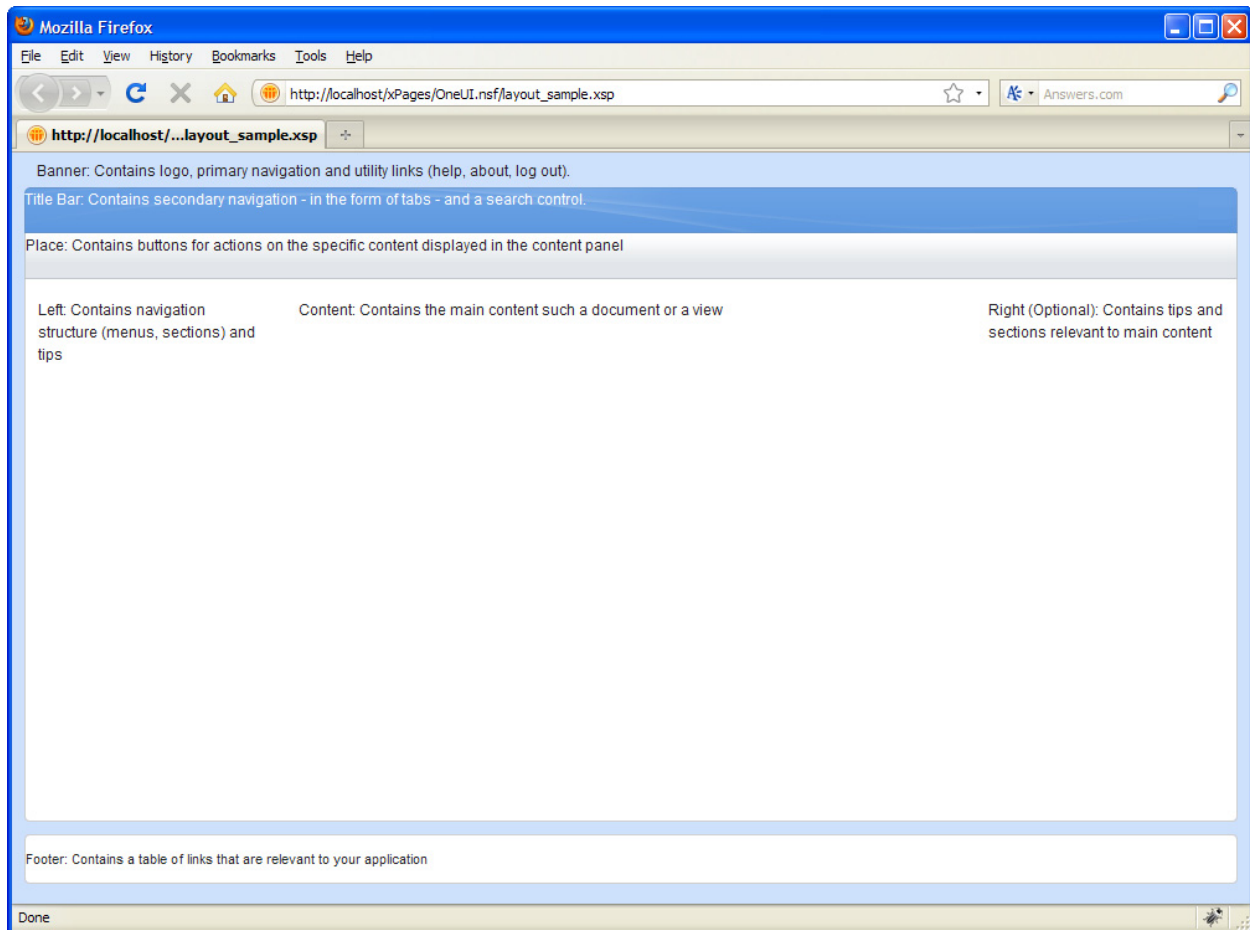
1. Frame
 - Container that contains all other sections listed below
 - <DIV> element styled with ‘**lotusFrame**’ CSS class
 - Allows CSS styles applicable across all sections
2. Banner:

- Contains logo, primary navigation and utility links (help, about, log out).
 - <DIV> element styled with '**lotusBanner**' CSS class
3. Title Bar
 - Contains secondary navigation - in the form of tabs - and a search control.
 - <DIV> element styled with '**lotusTitleBar**' CSS class
 4. Place Bar
 - Contains buttons for actions on the specific content displayed in the main section
 - <DIV> HTML element styled with '**lotusPlaceBar**' CSS class
 5. Main Section
 - Contains the main content
 - Breaks down into two or three columns: Left, Main and an optional Right column
 - <DIV> HTML element styled with '**lotusMain**' CSS class
 6. Footer
 - Contains a table of links that are relevant to your application
 - License and copyright verbiage
 - <DIV> HTML element styled with '**lotusFooter**' CSS class

Each of these sections is actually a HTML <DIV> element styled with appropriate CSS class for positioning and styling. This is final output to the browser:

```
<body class="lotusui">
  <div id="lotusFrame" class="lotusFrame">
    <div id="lotusBanner" class="lotusBanner"></div>
    <div id="lotusTitleBar" class="lotusTitleBar"></div>
    <div id="lotusPlaceBar" class="lotusPlaceBar"></div>
    <div id="lotusMain" class="lotusMain"></div>
    <div id="lotusFooter" class="lotusFooter"></div>
  </div>
</body>
```

This is how it is displayed in a browser:



To find out more details about One UI, refer to its documentation at:

<http://www-12.lotus.com/ldd/doc/oneuidoc/docpublic/index.htm>

Step 5 – Sample 2: Developing the web application layout using Custom Controls

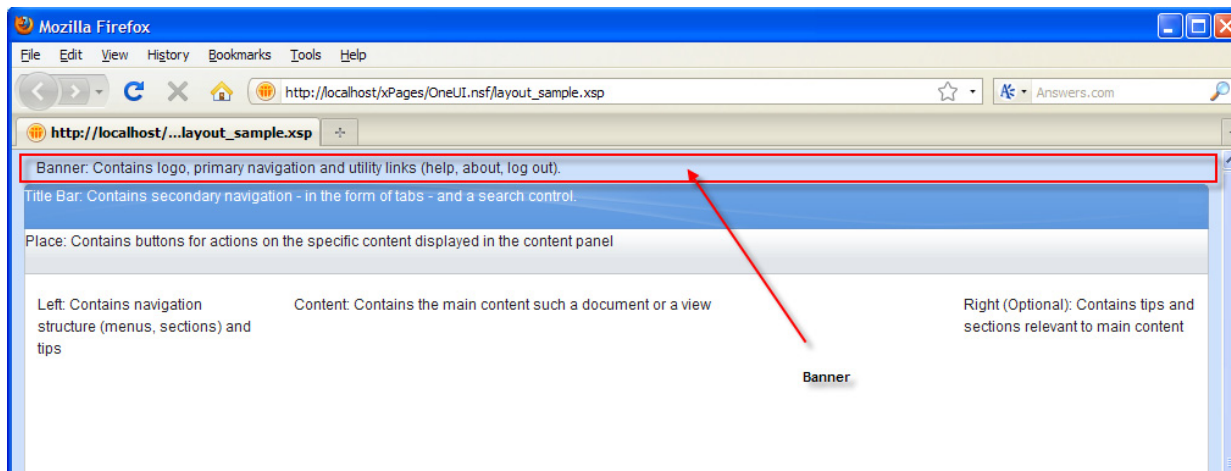
In order to build the web application layout, we will use XPages Custom Controls (CC). Below are the guidelines we are using to develop the layout:

- Identify the repeatable sections of the UI and implement those using Custom Controls: We are using Custom Controls so we can embed them within any XPage in our application and make changes in one place to update it across all XPages. This also breaks down the problem into pieces so you can focus on smaller pieces at a time.
- Separate layout Custom Controls from content Custom Controls: Using the separation of concerns principal, we are going to separate the layout Custom Controls from those containing any content. It provides these benefits:
 - Separating layout controls allows them to be re-used across various Domino applications

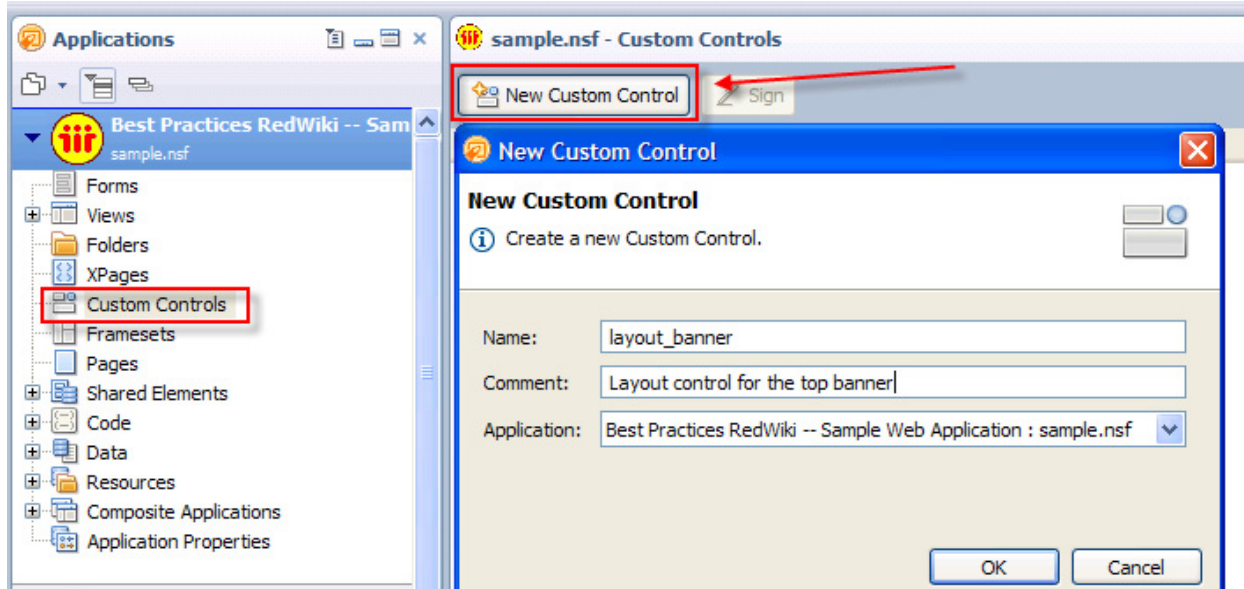
- You can make changes in the content control without worrying about any unintended effect on the overall layout.
- Developers experienced with CSS framework and themes can focus on layout controls and those experienced with core functionality can focus on the content controls.
- Use a naming convention and custom control category to distinguish layout controls from the content controls. We are going to name the layout custom controls starting with “**layout_**” and use “**Layout Custom Controls**” to display them in Custom Controls palette.
- Since the content of the layout control can potentially change, instead of hard coding the content, we are going to use **Editable Area** control to act as a place holder for the content. Editable Areas are regions of the custom control that can be modified when a custom control is placed in an XPage. For example, you can add other controls within an Editable Area of the custom control in an XPage. This allows you to use the same custom control but modify its content based on the XPage or compute the content based on a criteria – such as query string parameter.

Step 5.1 : Creating layout custom control for banner area

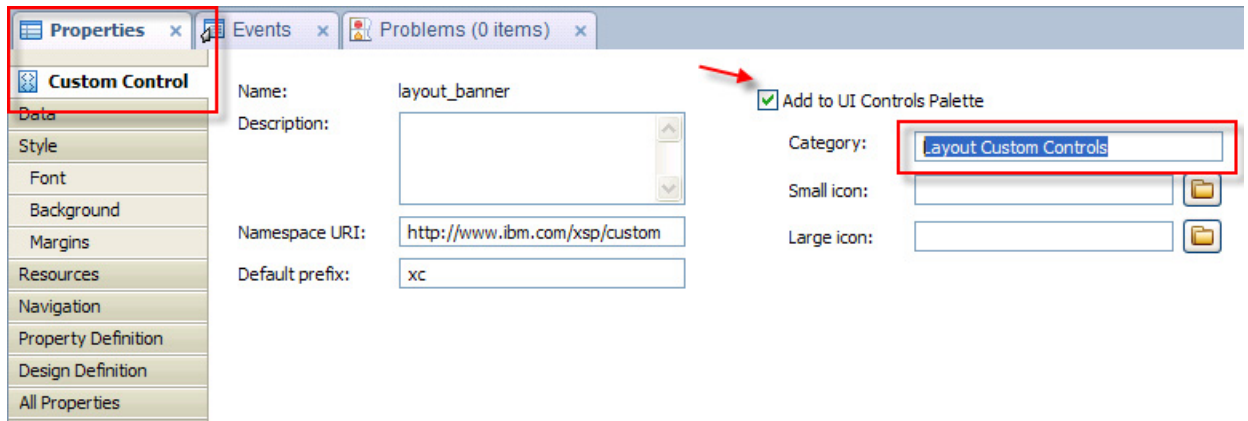
In this section we will build the layout custom control for the top banner. Top banner contains logo, global navigation and utility links (Help, About, Logout, etc.) In this custom control, we are not adding any content. Instead, this is just a layout control, for positioning and styling, with an Editable Area which acts as a place holder for the contents.



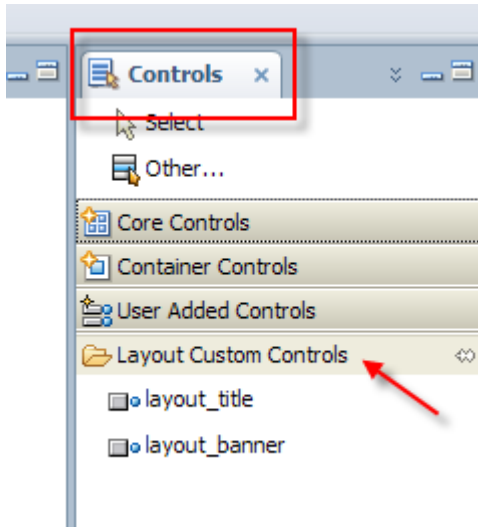
1. Create a new custom control and name it as **layout_banner**.



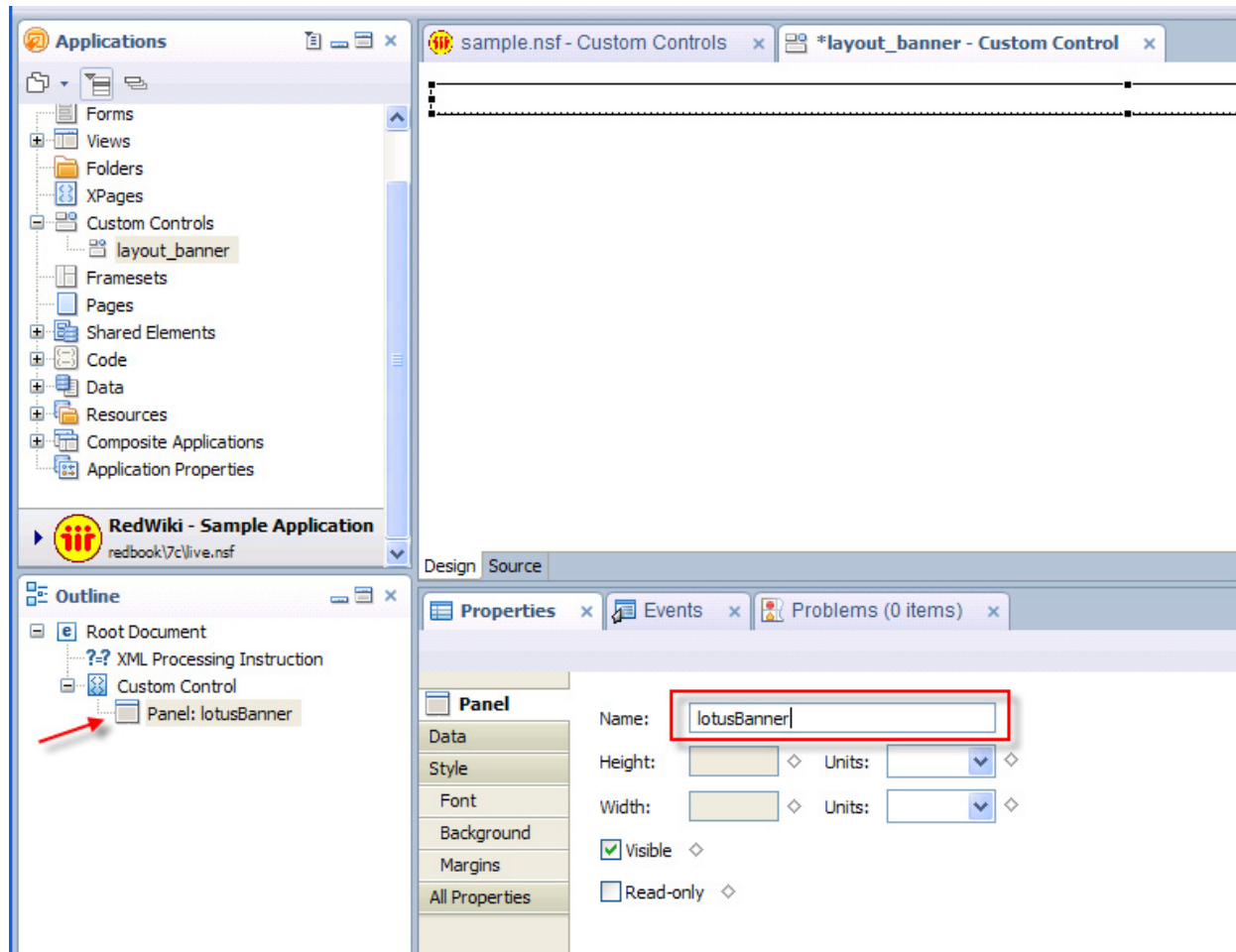
2. Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Layout Custom Controls**” as the category.



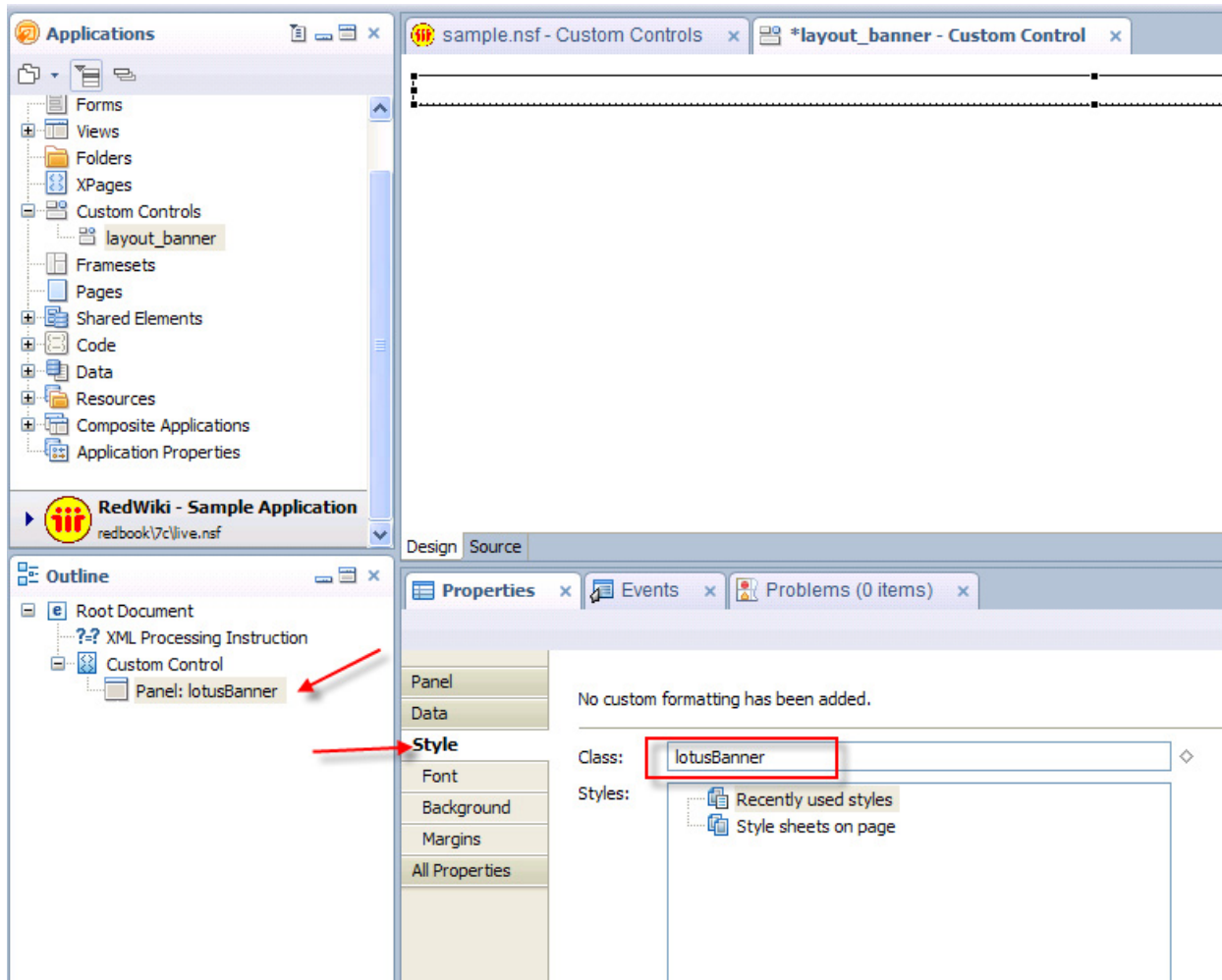
3. Save the custom control by pressing **Ctrl-S**. This adds a new category to the controls palette and moves this custom control under the category “**Layout Custom Controls**”. Categorization allows you to separate custom controls within the **Controls** palette, which makes it easier to find them as the total number of custom controls grow.



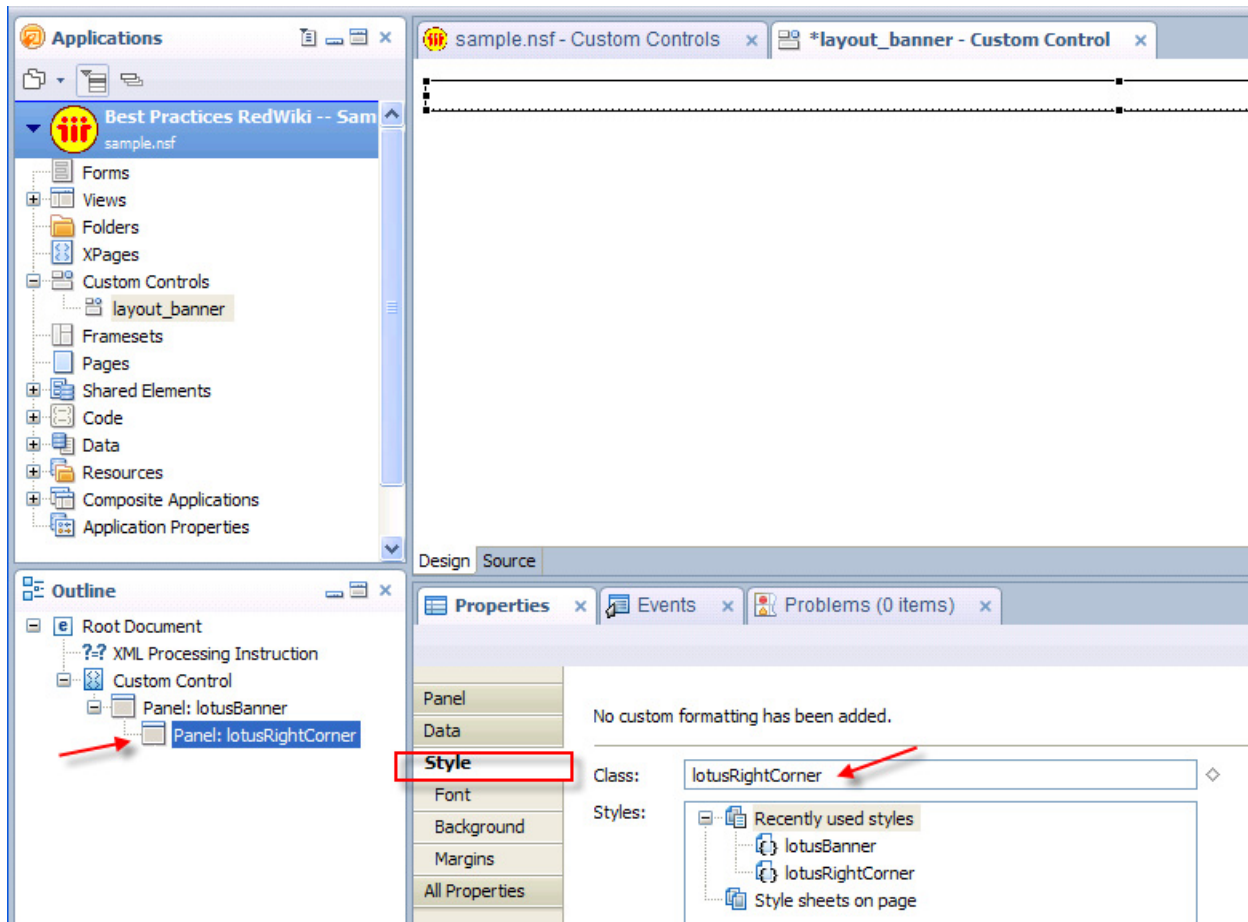
4. From the control palette, under the **Container Controls** section, drag and drop **Panel** control to the top-left on editor.
5. From the outline palette, select the newly created panel and enter “**lotusBanner**” as the panel name.



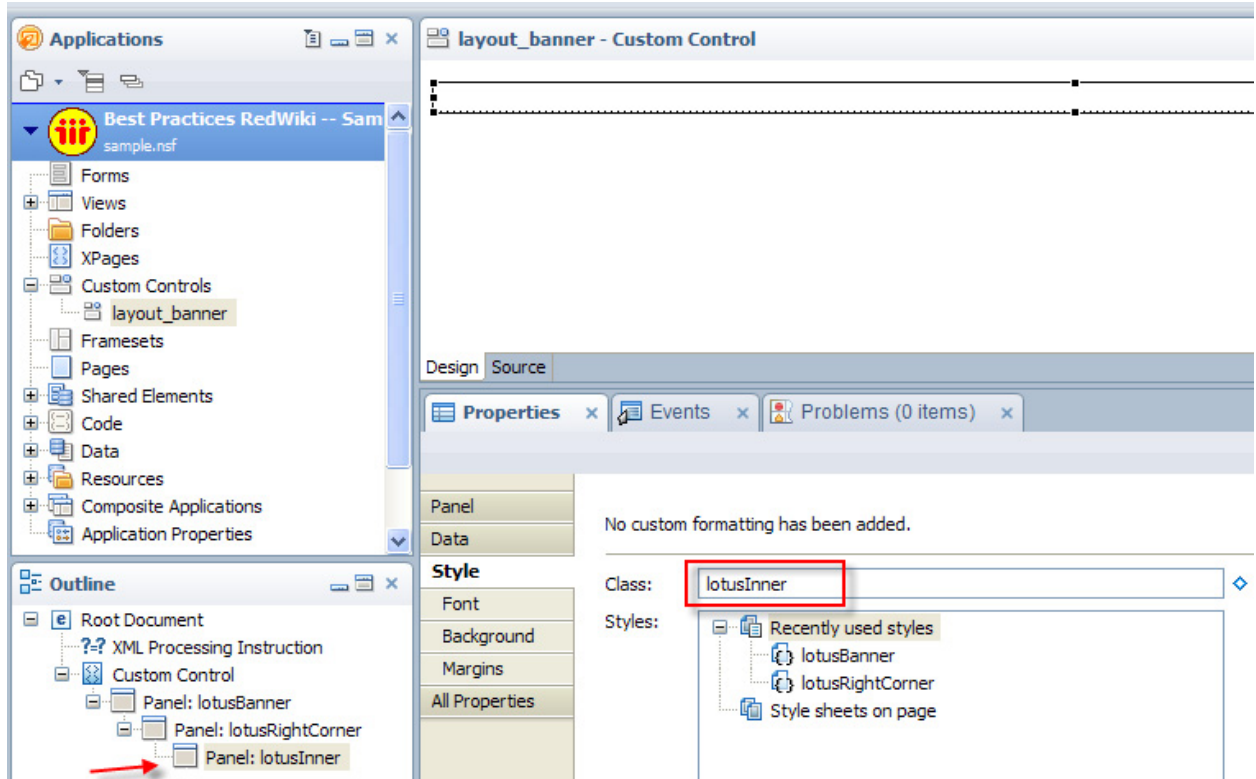
6. From the Properties palette, select “**Style**” tab and enter “**lotusBanner**” as CSS style class.



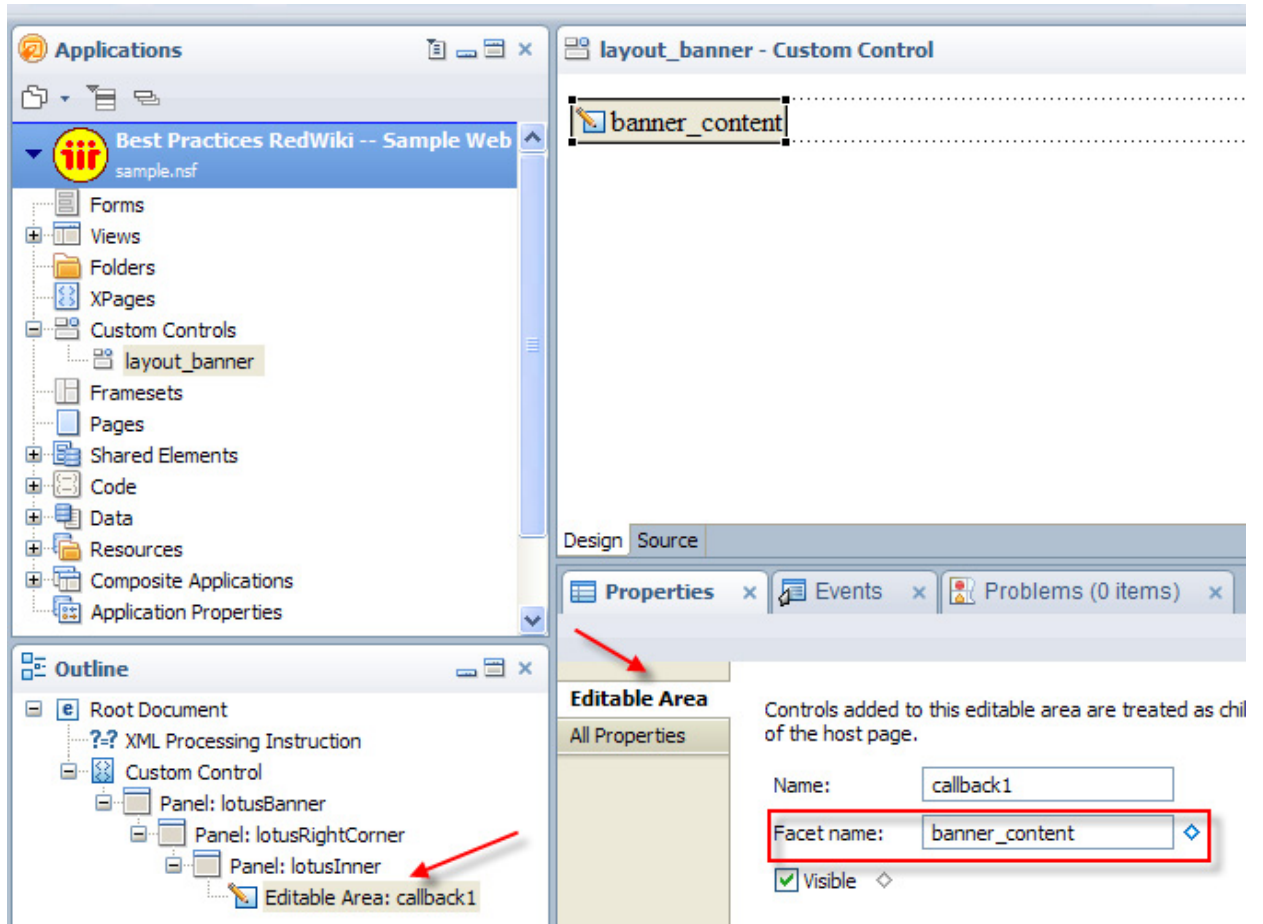
7. From the control palette, drag **Panel** control and drop it within the “**lotusBanner**” panel created earlier. Enter “**lotusRightCorner**” both as control name and CSS style class.



8. From the control palette, drag **Panel** control and drop it within the “**lotusRightCorner**” panel created above. Enter “**lotusInner**” both as control name and CSS style class.



9. From the core controls palette, drag the **Editable Area** control and drop it over the “**lotusInner**” panel created in last step. Enter “**banner_content**” as the facet name. Save the custom control by pressing Ctrl-S.

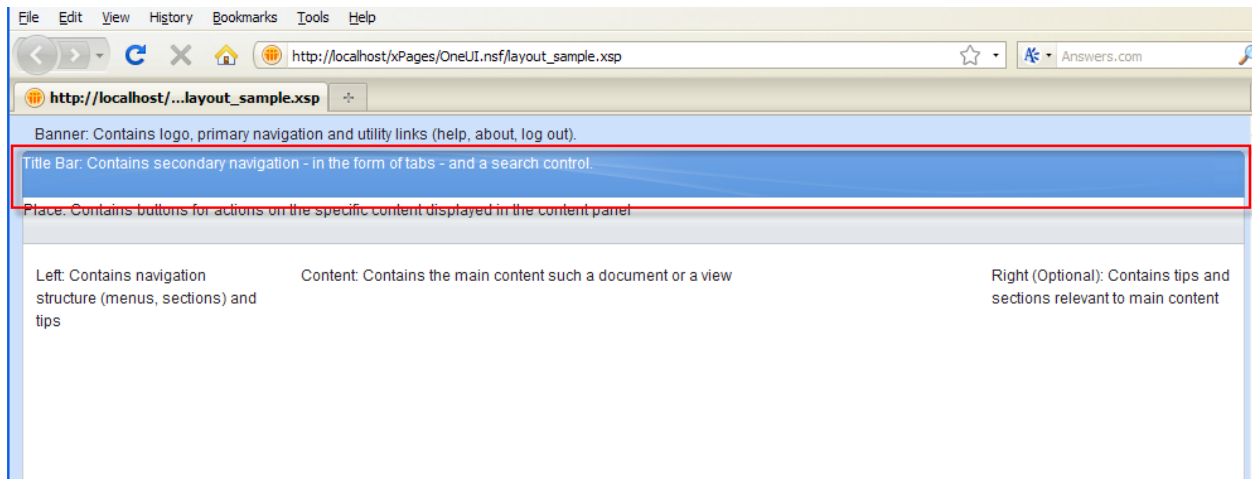


10. Click on "Source" tab in xPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

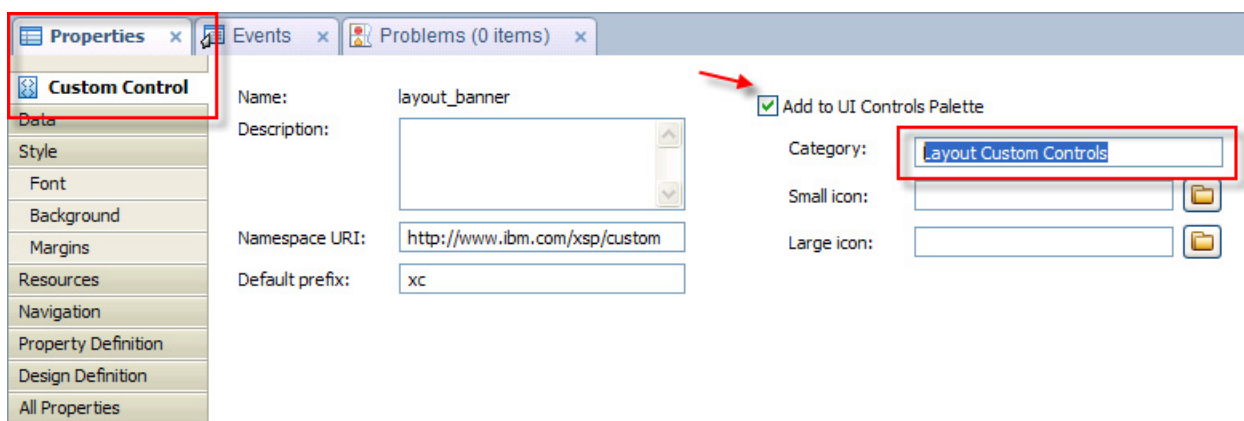
```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel id="lotusBanner" styleClass="lotusBanner">
    <xp:panel styleClass="lotusRightCorner" id="lotusRightCorner">
      <xp:panel id="lotusInner" styleClass="lotusInner">
        <xp:callback facetName="banner_content"
id="callback1"></xp:callback>
      </xp:panel>
    </xp:panel>
  </xp:panel>
</xp:view>
```


Step 5.2 : Creating layout custom control for title bar

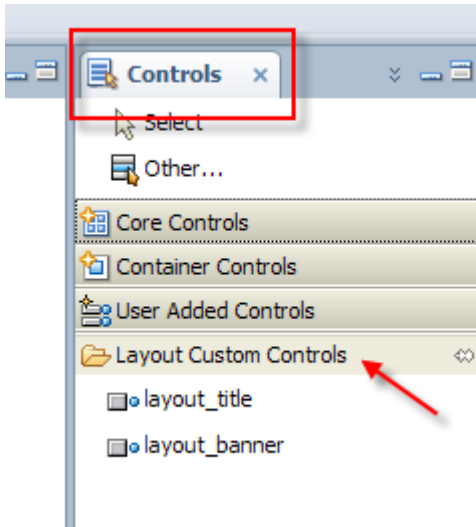
In this section we will build the layout custom control for the title bar. Title bar is the section right below banner area. It contains secondary navigation (in forms of tabs) and search control. In this custom control, we are not adding any content. Instead, this is just a layout control, for positioning and styling, with an Editable Area, which acts as a place holder for the contents.



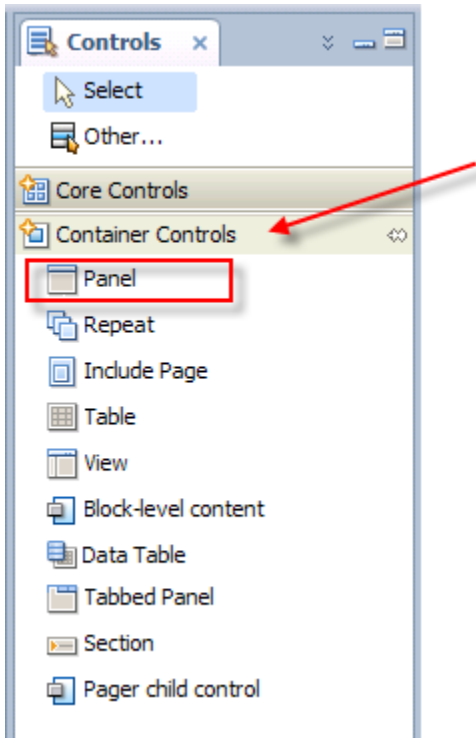
1. From the “**Custom Control**” click on “**New Custom Control**” button and enter “**layout_title**” as the name. Click OK to save.
2. Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Layout Custom Controls**” as the category.



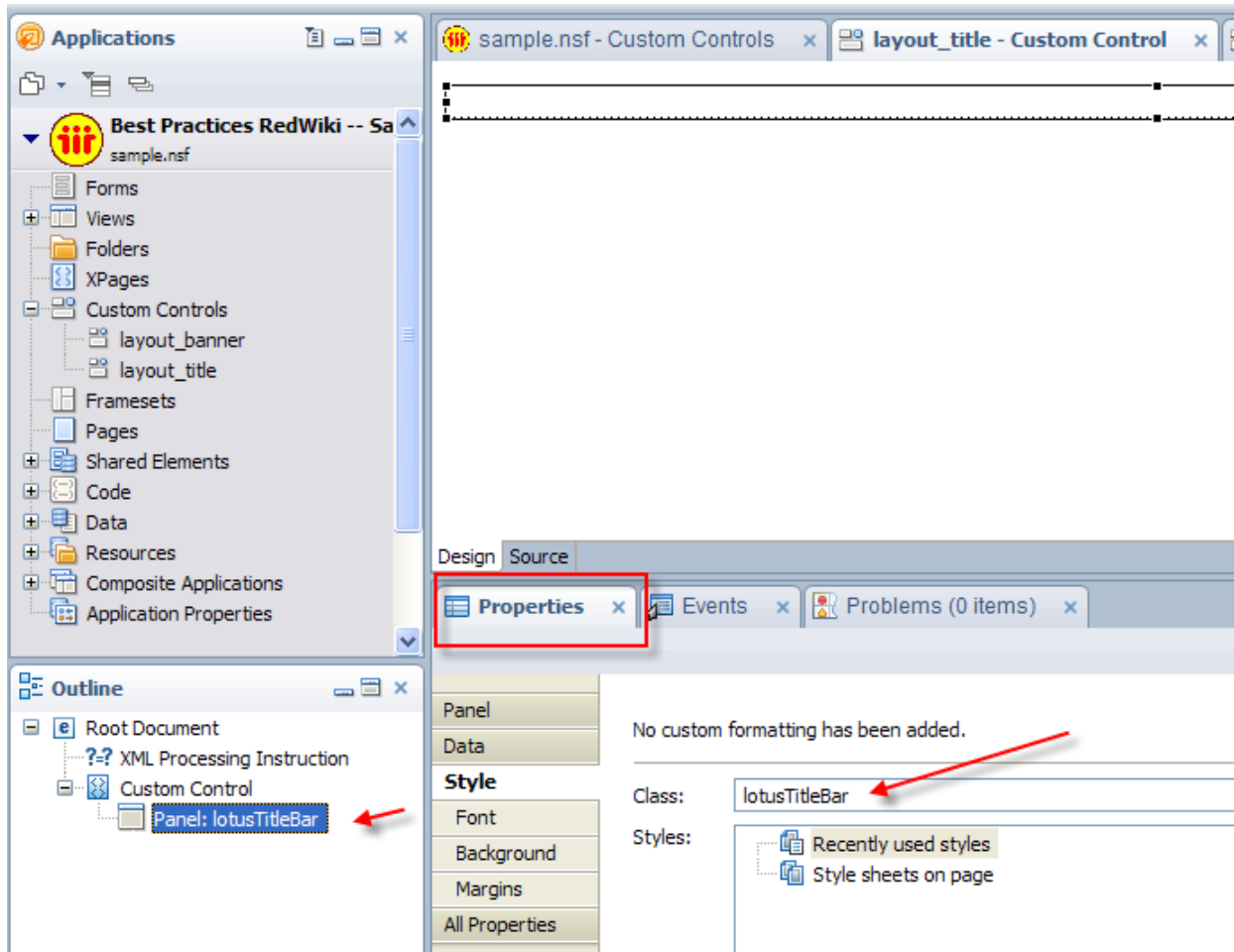
3. Save the custom control by pressing Ctrl-S. This moves this custom control under the category “**Layout Custom Controls**”.



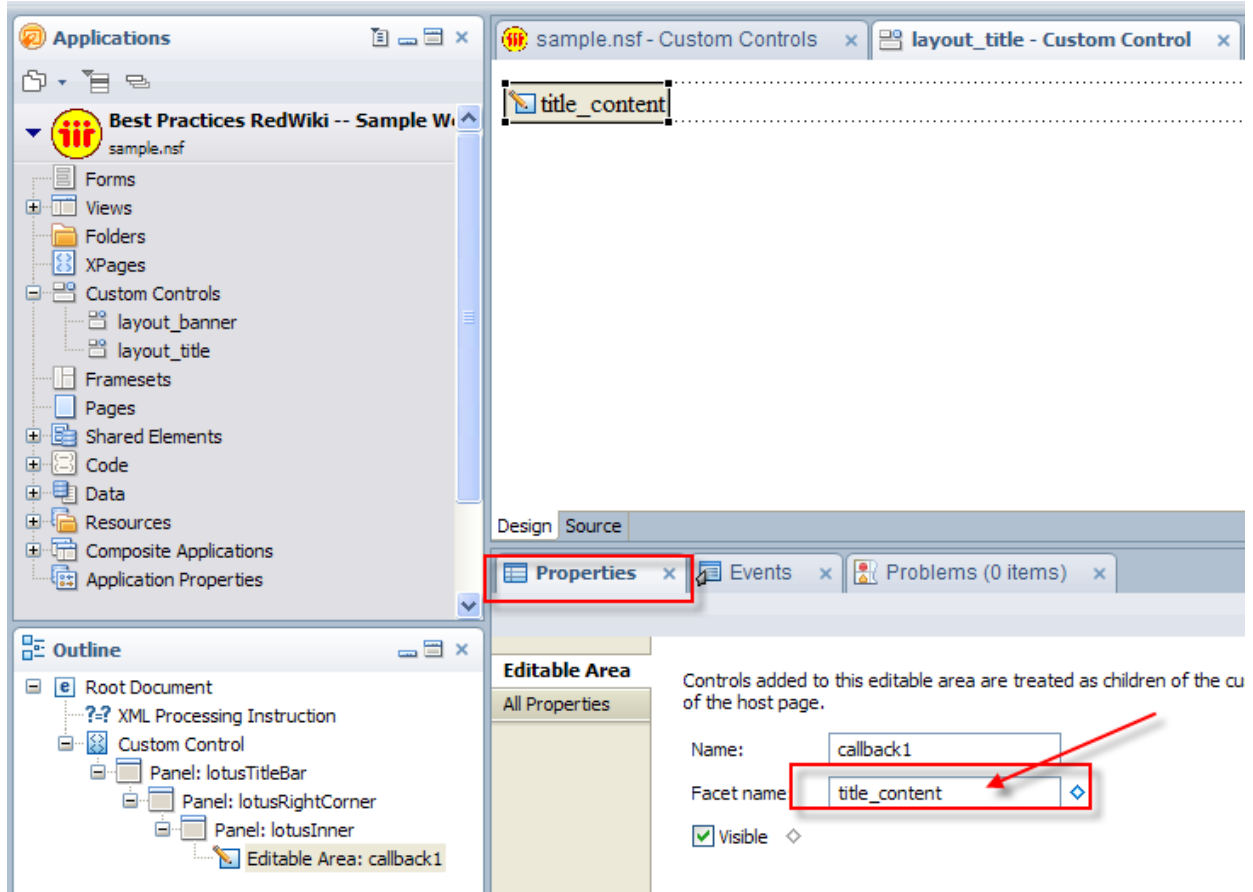
4. From the control palette, under the **Container Controls** section, drag and drop **Panel** control to the top-left on custom control editor.



5. From the outline palette, select the newly created panel and enter "**lotusTitleBar**" both as the panel name and as CSS style class.



6. From the control palette, drag **Panel** control and drop it within the “**lotusTitleBar**” panel created above. Enter “**lotusRightCorner**” both as control name and css style class.
7. From the control palette, drag Panel control and drop it within the “**lotusRightCorner**” panel created above. Enter “**lotusInner**” both as control name and css style class.
8. From the core controls palette, drag the **Editable Area** control and drop it over the “**lotusInner**” panel created in last step. Enter “**titleBar_content**” as the facet name. Save the custom control by pressing Ctrl-S.

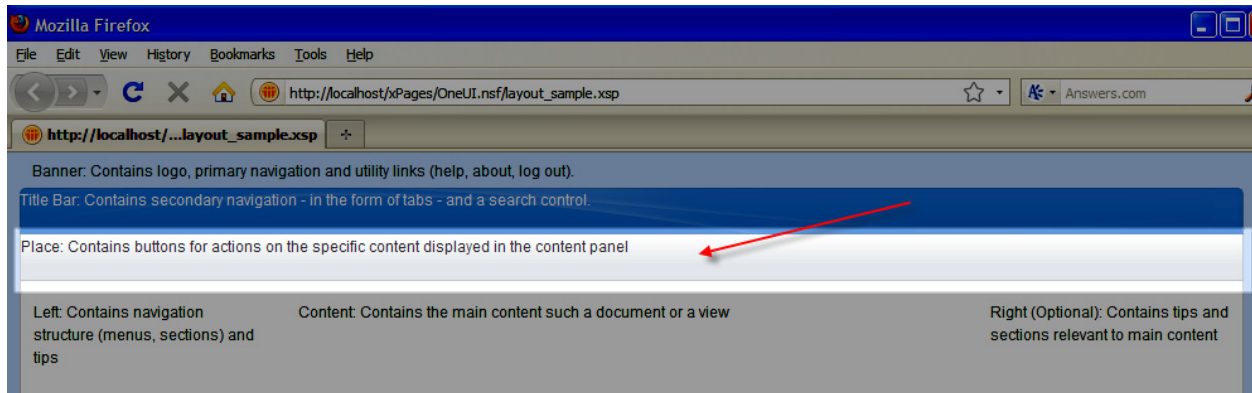


9. Click on “Source” tab in XPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel id="lotusTitleBar" styleClass="lotusTitleBar">
    <xp:panel styleClass="lotusRightCorner" id="lotusRightCorner">
      <xp:panel id="lotusInner" styleClass="lotusInner">
        <xp:callback facetName="title_content"
id="callback1"></xp:callback>
      </xp:panel>
    </xp:panel>
  </xp:panel>
</xp:view>
```

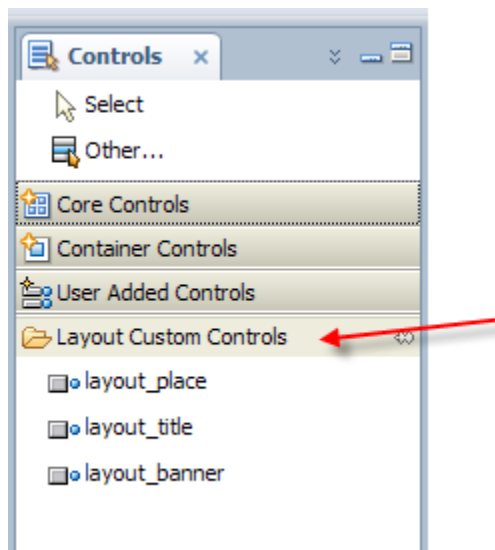
Step 5.3 : Creating layout custom control for place bar

In this section we will build the layout custom control for the place bar. Place bar is the section right below the title bar. It contains actions/buttons for the content displayed in the content panel. In this custom control, we are not adding any content. Instead, this is just a layout control, for positioning and styling, with an Editable Area, which acts as a place holder for the contents.

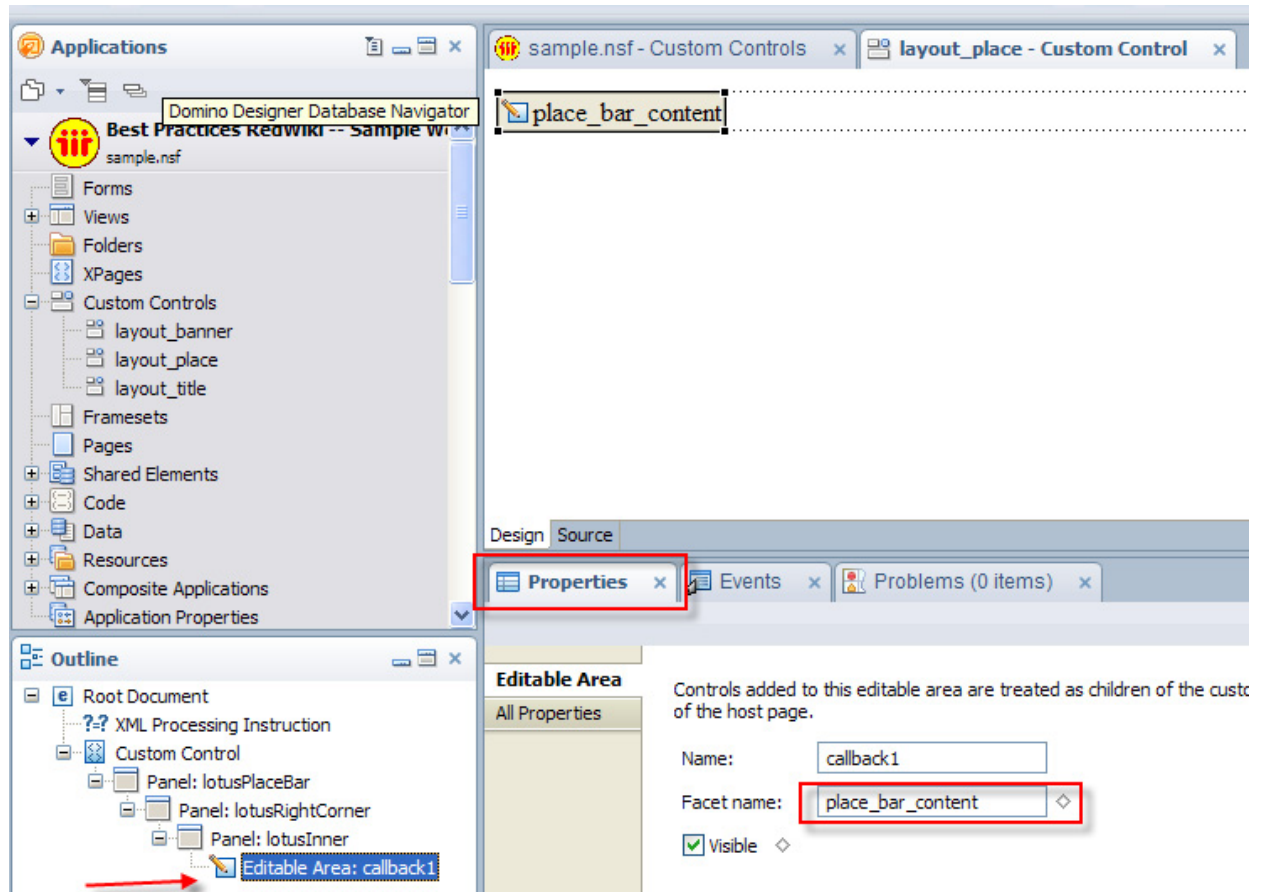


1. From the “**Custom Control**” click on “**New Custom Control**” button and enter “**layout_place**” as the name. Click **OK** to save.

10. Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Layout Custom Controls**” as the category. This moves this custom control under the category “**Layout Custom Controls**”.



2. From the control palette, under the **Container Controls** section, drag and drop **Panel** control to the top-left on custom control editor.
3. From the outline palette, select the newly created panel and enter “**lotusPlaceBar**” both as the panel name and as CSS style class.
4. From the control palette, drag Panel control and drop it within the “**lotusPlaceBar**” panel created above. Enter “**lotusRightCorner**” both as control name and css style class.
5. From the control palette, drag Panel control and drop it within the “**lotusRightCorner**” panel created above. Enter “**lotusInner**” both as control name and css style class.
6. From the core controls palette, drag the **Editable Area** control and drop it over the “**lotusInner**” panel created in last step. Enter “**place_bar_content**” as the facet name. Save the custom control by pressing **Ctrl-S**.

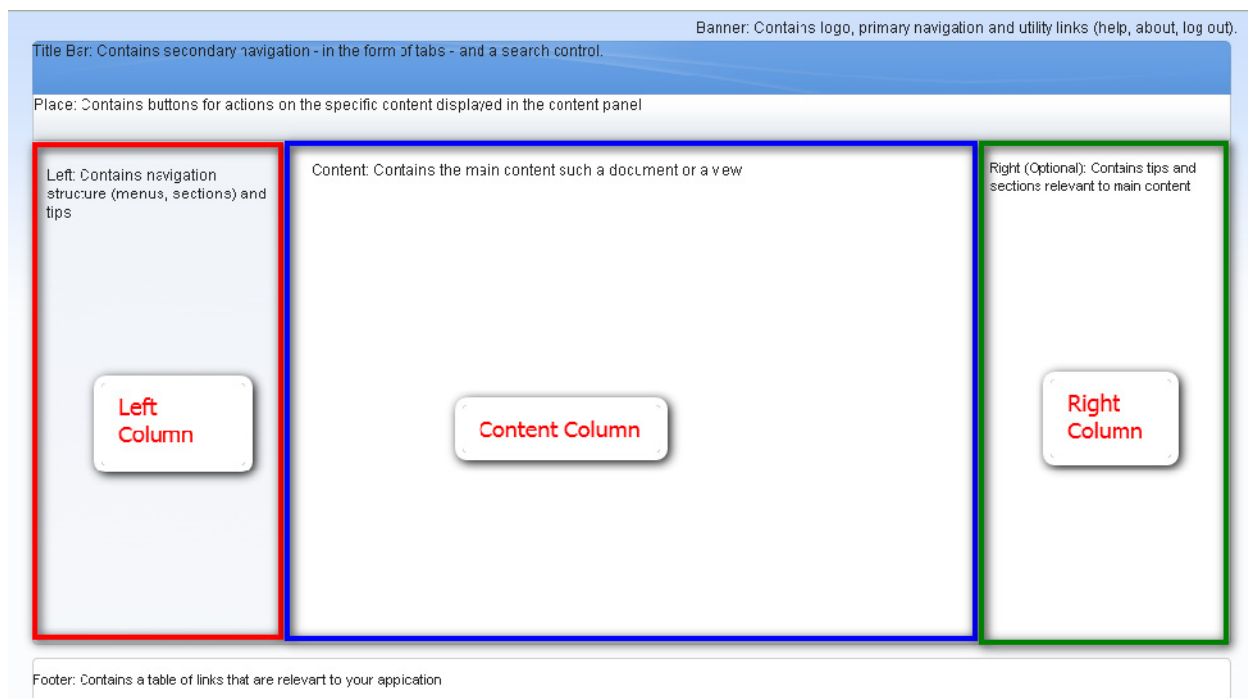


- Click on “Source” tab in XPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel id="lotusPlaceBar" styleClass="lotusPlaceBar">
    <xp:panel id="lotusRightCorner" styleClass="lotusRightCorner">
      <xp:panel styleClass="lotusInner" id="lotusInner">
        <xp:callback facetName="place_bar_content"
id="callback1"></xp:callback>
      </xp:panel>
    </xp:panel>
  </xp:panel>
</xp:view>
```

Step 5.4 : Creating layout custom control for left column

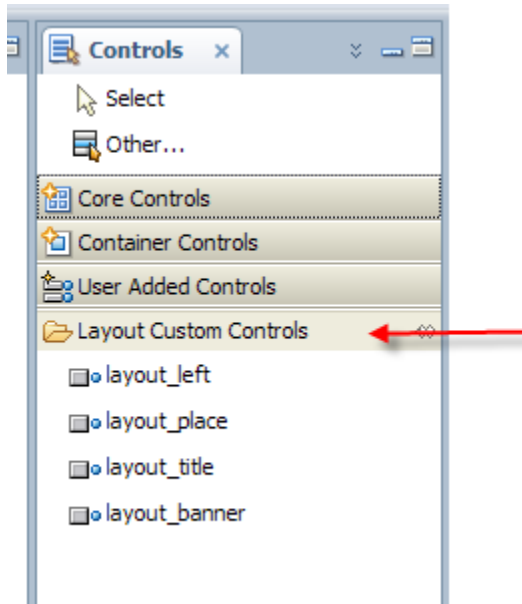
One UI divides the main panel into three sections: left, content and right. Since the layout stays the same but content can change from one web page to another, we are going to implement each one of these three as a layout custom control.



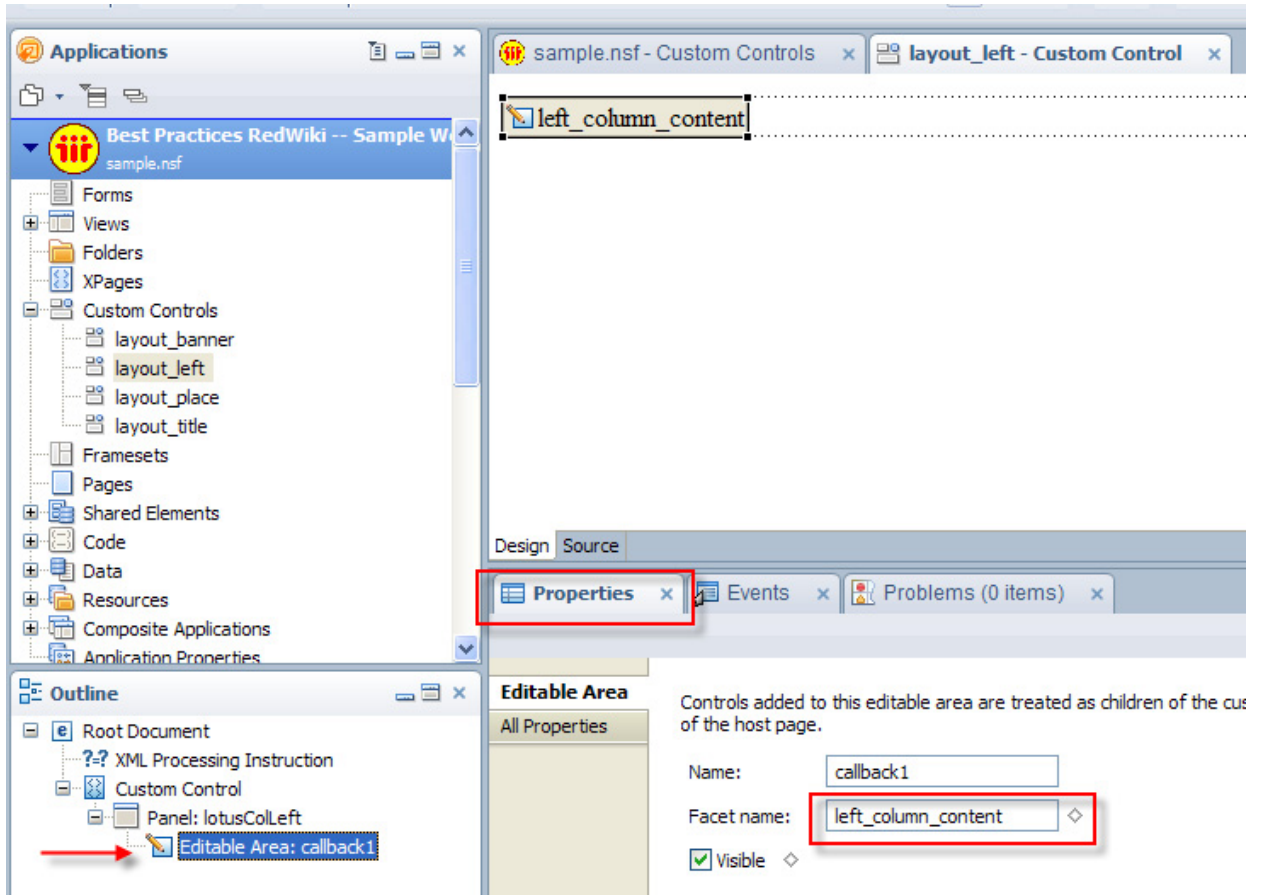
In this section we will build the layout custom control for the left column. This column contains the navigation structure, such as menus and sections, for the content displayed in the content column. In this custom control, we are not adding any content. Instead, this is just a layout control, for positioning and styling, with an Editable Area which acts as a place holder for the content.

- Click on “New Custom Control” button and enter “**layout_left**” as the name. Click **OK**.

2. Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Layout Custom Controls”** as the category. This moves this custom control under the category **“Layout Custom Controls”**.



3. From the control palette, under the Container Controls section, drag and drop Panel control to the top-left on custom control editor.
4. From the outline palette, select the newly created panel and enter “**lotusColLeft**” both as the panel name and as CSS style class.
5. From the core controls palette, drag the **Editable Area** control and drop it over the “**lotusColLeft**” panel created in last step. Enter “**left_column_content**” as the facet name. Save the custom control by pressing **Ctrl-S**.

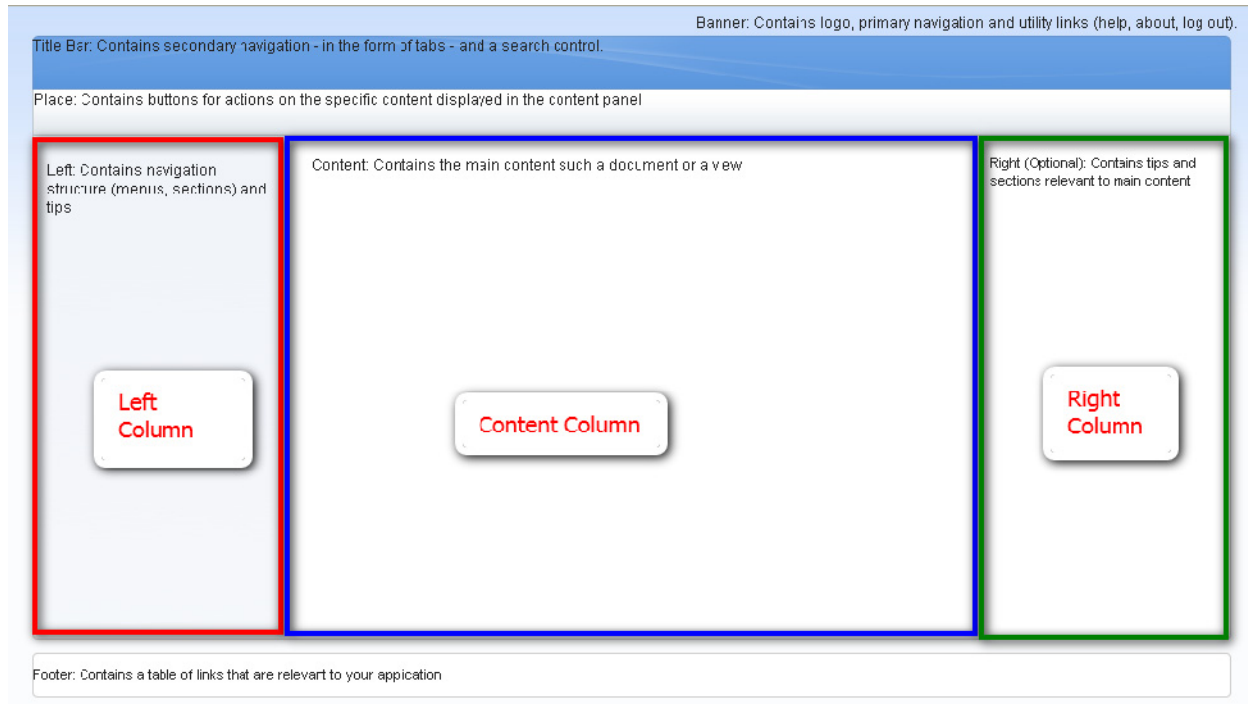


- Click on **"Source"** tab in xPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel id="lotusColLeft" styleClass="lotusColLeft">
    <xp:callback facetName="left_column_content"
id="callback1"></xp:callback>
  </xp:panel>
</xp:view>
```

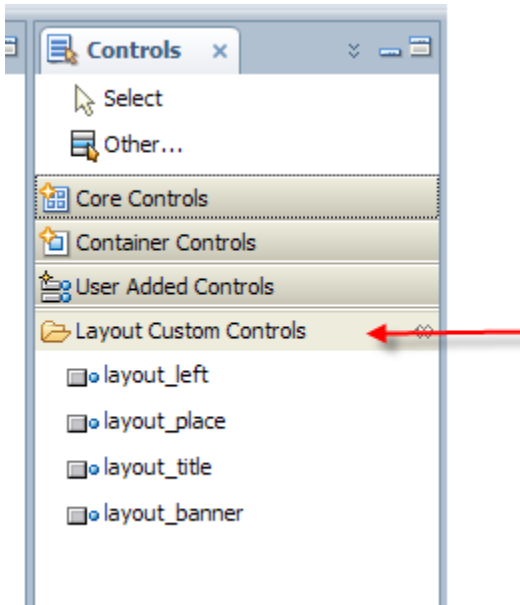
Step 5.5 : Creating layout custom control for right column

One UI divides the main panel into three sections: left, content and right. Since the layout stays the same but content can change from one web page to another, we are going to implement each one of these three as a layout custom control.

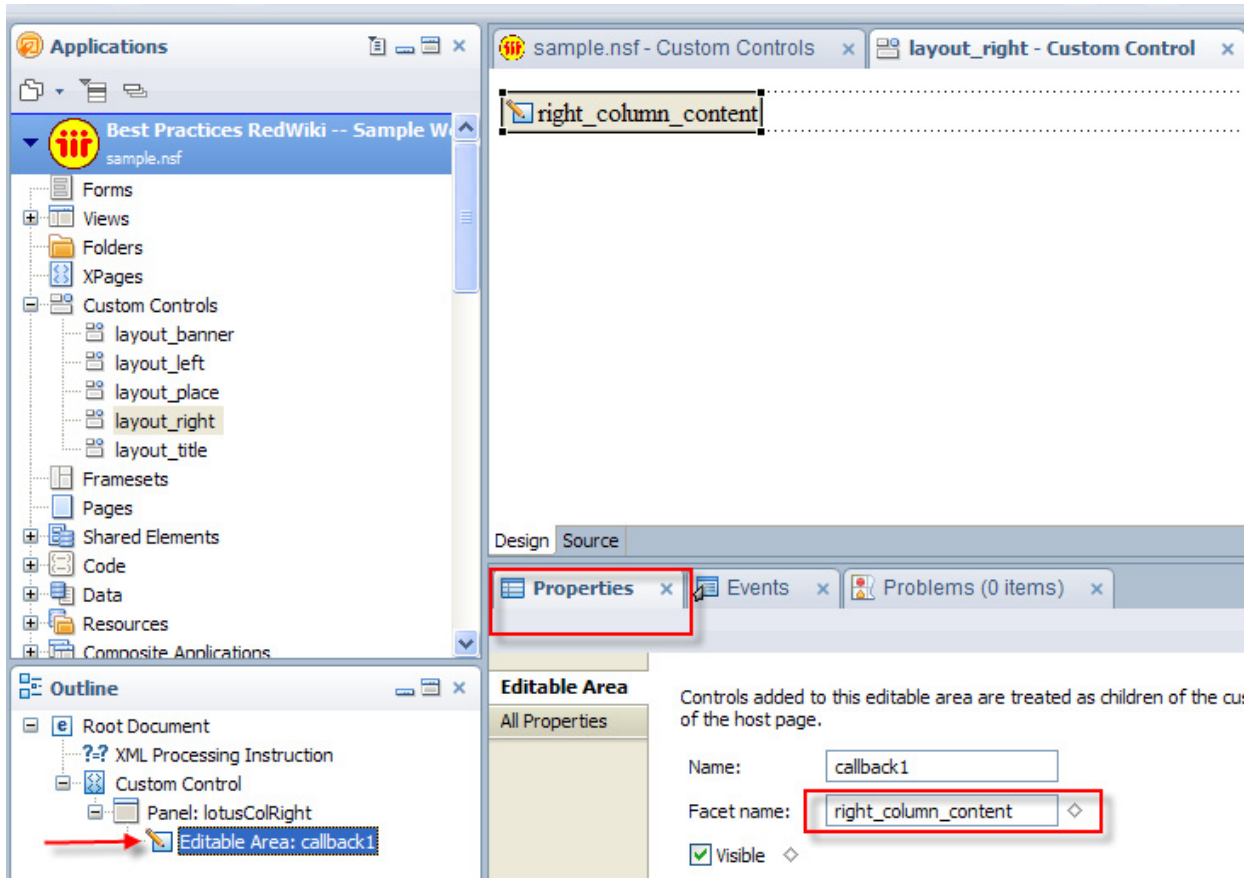


In this section we will build the layout custom control for the right column. This column contains the tips and sections relevant to the content displayed in the content column. In this custom control, we are not adding any content. Instead, this is just a layout control, for positioning and styling, with an Editable Area which acts as a place holder for the content.

1. Click on **"New Custom Control"** button and enter **"layout_right"** as the name. Click **OK**.
2. Under the Properties tab, make sure Custom Control is selected and check **"Add to UI Controls Palette"** and enter **"Layout Custom Controls"** as the category. This moves this custom control under the category **"Layout Custom Controls"**.



3. From the control palette, under the **Container Controls** section, drag and drop **Panel** control to the top-left on custom control editor.
4. From the outline palette, select the newly created panel and enter "**lotusColRight**" both as the panel name and as CSS style class.
5. From the core controls palette, drag the **Editable Area** control and drop it over the "**lotusColRight**" panel created in last step. Enter "**right_column_content**" as the facet name. Save the custom control by pressing **Ctrl-S**.

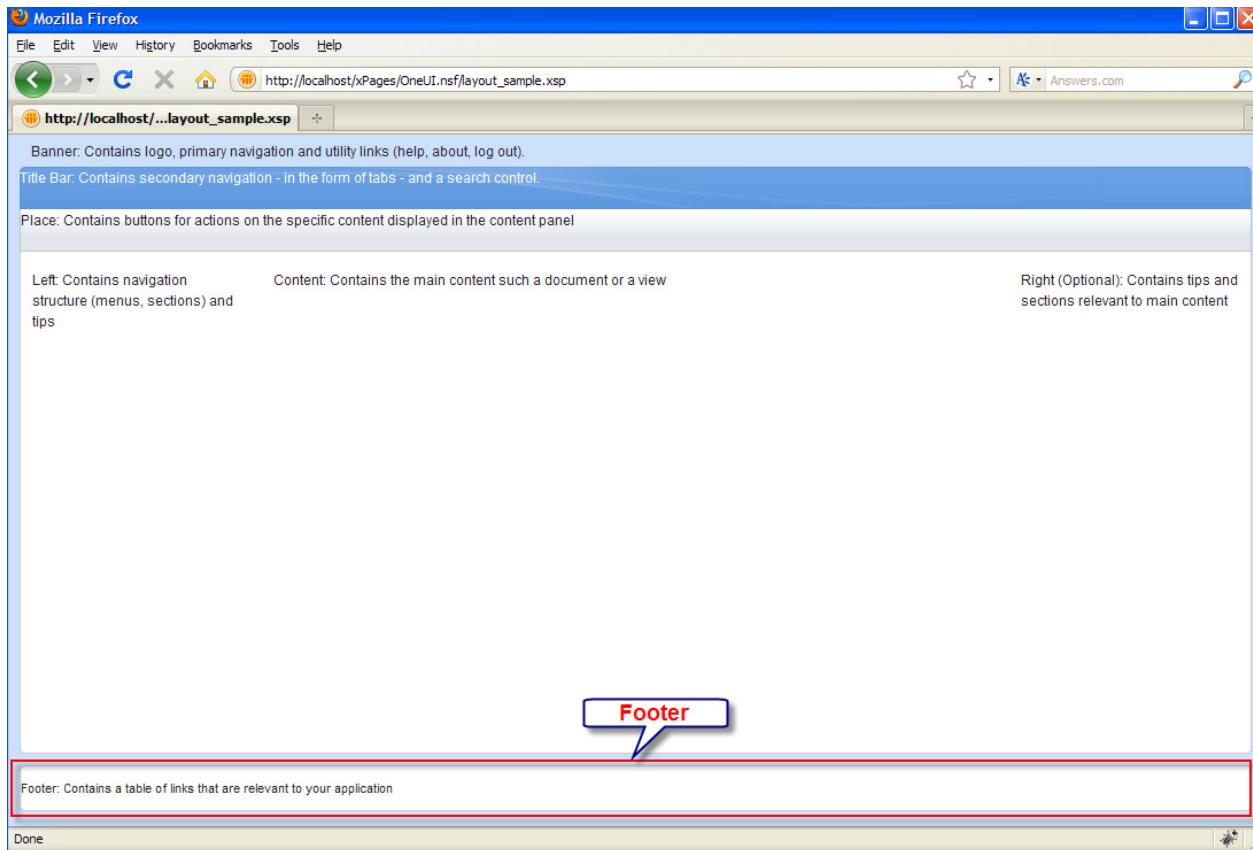


6. Click on **"Source"** tab in xPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

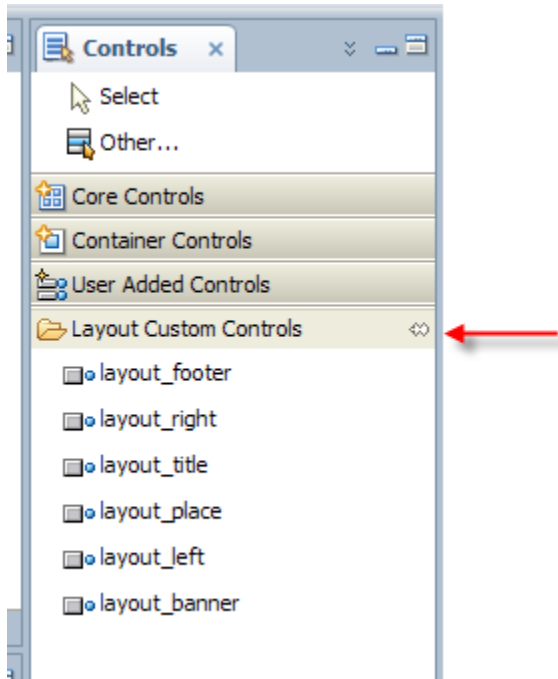
```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel styleClass="lotusColRight" id="lotusColRight">
    <xp:callback facetName="right_column_content" id="callback1">
    </xp:callback>
  </xp:panel>
</xp:view>
```

Step 5.6: Creating layout custom control for footer area

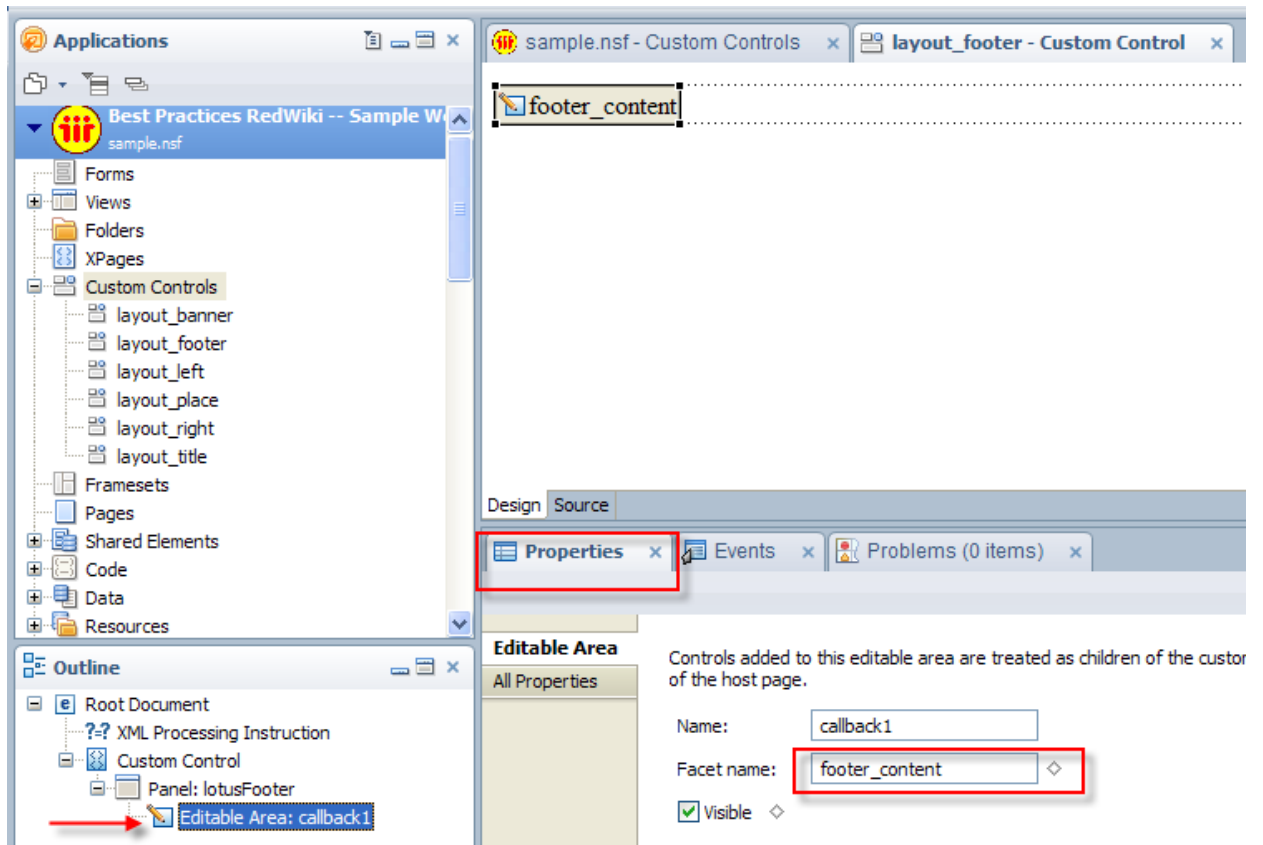
In this section we will build the layout custom control for the footer area. This area can contain any copyright and legal verbiage along with some relevant links that are applicable across all pages. In this custom control, we are not adding any content. Instead, this is just a layout control, for positioning and styling, with an Editable Area which acts as a place holder for the content.



1. Click on **“New Custom Control”** button and enter **“layout_footer”** as the name. Click **OK**.
2. Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Layout Custom Controls”** as the category and save changes. This moves this custom control under the category **“Layout Custom Controls”**.



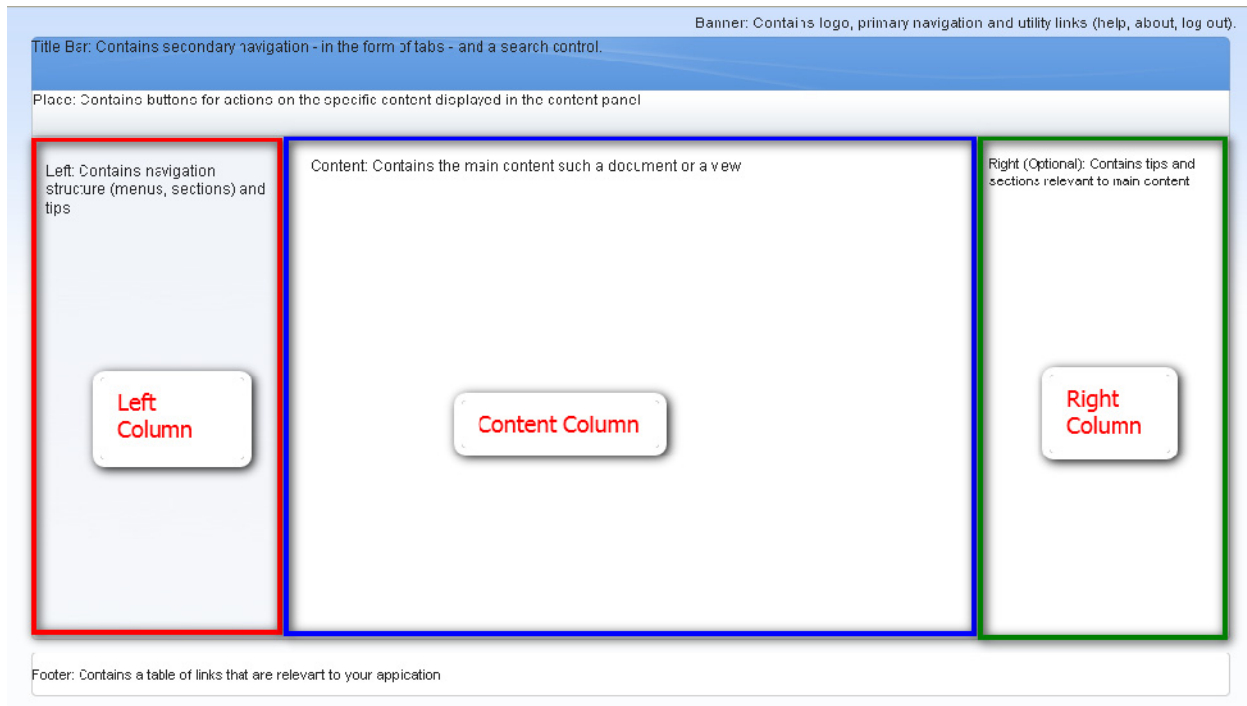
3. From the control palette, under the **Container Controls** section, drag and drop **Panel** control to the top-left on custom control editor.
4. From the outline palette, select the newly created panel and enter "**lotusFooter**" both as the panel name and as CSS style class.
5. From the core controls palette, drag the **Editable Area** control and drop it over the "**lotusFooter**" panel created in last step. Enter "**footer_content**" as the facet name. Save the custom control by pressing **Ctrl-S**.



6. Click on “Source” tab in xPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

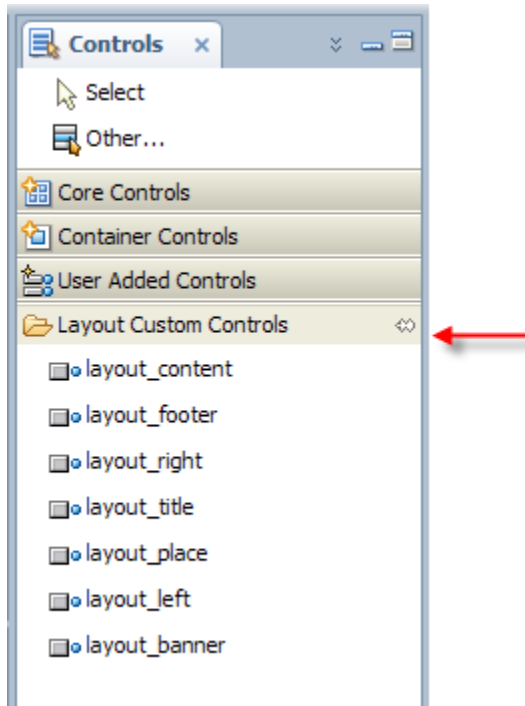
```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel styleClass="lotusFooter" id="lotusFooter">
    <xp:callback facetName="footer_content" id="callback1">
    </xp:callback>
  </xp:panel>
</xp:view>
```


Step 5.7: Creating layout custom control for content column

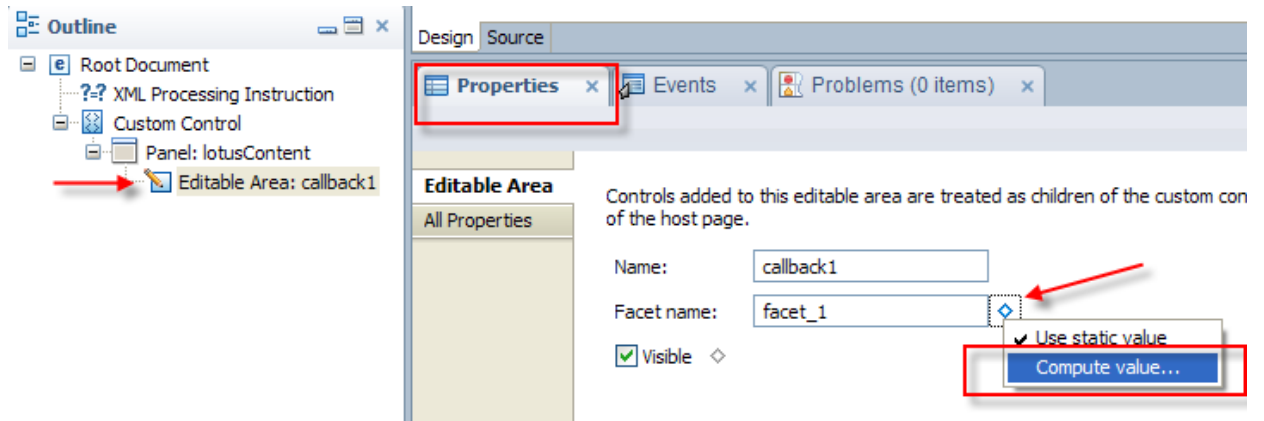


In this section we will build the layout custom control for the content column. This column contains the main content such as views, forms, search results, etc. In this custom control, we are not adding any content. Instead, this is just a layout control, for positioning and styling, with an Editable Area which acts as a place holder for the content. In case of content column, the actual contents change frequently based on what user has selected from the left menu or any other link they have clicked within the application. Therefore, we are going to use a technique which allows the content to be selected **dynamically** based on a formula – in this case query string parameter.

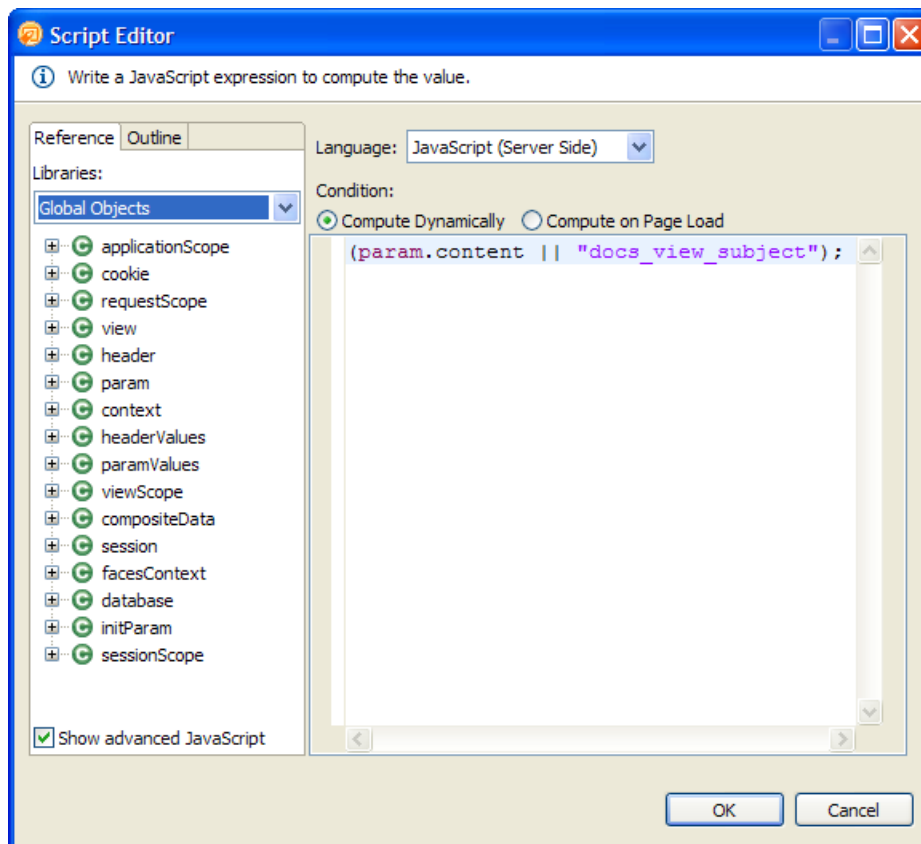
1. Click on **“New Custom Control”** button and enter **“layout_content”** as the name. Click OK.
2. Under the Properties tab, make sure **Custom Control** is selected and check **“Add to UI Controls Palette”** and enter **“Layout Custom Controls”** as the category. This moves this custom control under the category **“Layout Custom Controls”**.



3. From the control palette, under the **Container Controls** section, drag and drop **Panel** control to the top-left on custom control editor.
4. From the outline palette, select the newly created panel and enter "**lotusContent**" both as the panel name and as CSS style class.
5. From the core controls palette, drag the **Editable Area** control and drop it over the "**lotusContent**" panel created in last step.
6. From the outline palette, select the editable area created in the last step. From the properties palette, click on diamond-shaped icon to the right of "**Facet name**" property and select "**computed**" from the drop down. This allows the property to be computed -- as opposed to a static hard coded value.



7. When you select “**Computed Value..**” in the last step, it opens a script editor. Make sure the language is selected as “**JavaScript (Server Side)**” and condition radio button is selected as “**Computed Dynamically**”. Enter `(param.content || "docs_view_subject");` and click **OK**. This makes the facet name computed dynamically based on the value of the query string parameter named “content”. If there is no such parameter, it will use “docs_view_subject” facet name as default value. **This technique allows us to include a custom control dynamically** within another control based on the value of query string parameter.



- Click on “Source” tab in xPages editor and press Ctr-Shift-F to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel id="lotusContent" styleClass="lotusContent">
    <xp:callback id="callback1">
      <xp:this.facetName><![CDATA[#{javascript:(param.content ||
"docs_view_subject");}]]></xp:this.facetName>
    </xp:callback>
  </xp:panel>
</xp:view>
```

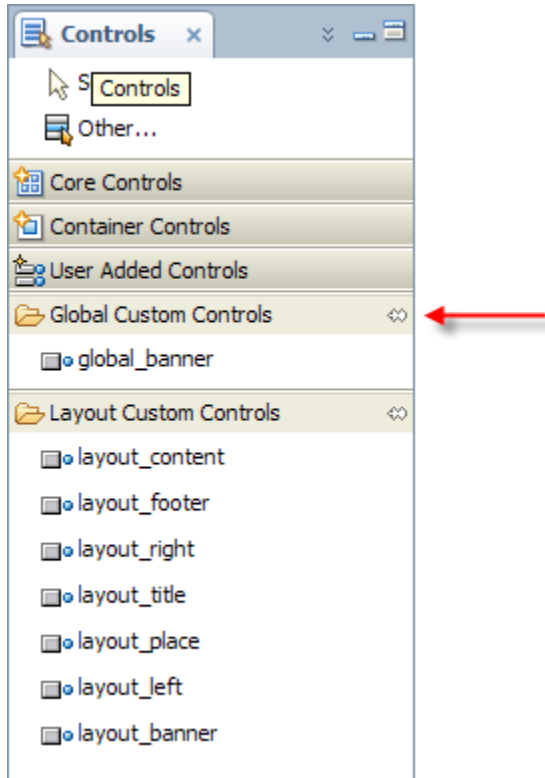
Step 6 – Sample 2: Developing custom controls for global content

Layout controls developed earlier are only for positioning and high-level styling. Instead of containing any content they are placeholders for content. In this section, we are going to develop the custom controls for global content. Global content are those areas which are neither the layout-only areas nor specific to a particular page/section of the web application. In the context of One UI Framework, this includes global navigation on the top banner, utility links, search control and footer content. These custom controls are going to be included within the layout controls, but they are not the layout controls themselves – they do include content other than panels (which turn into <div> at run time) and Editable Areas (which acts as placeholder where the content controls are added within the layout controls).

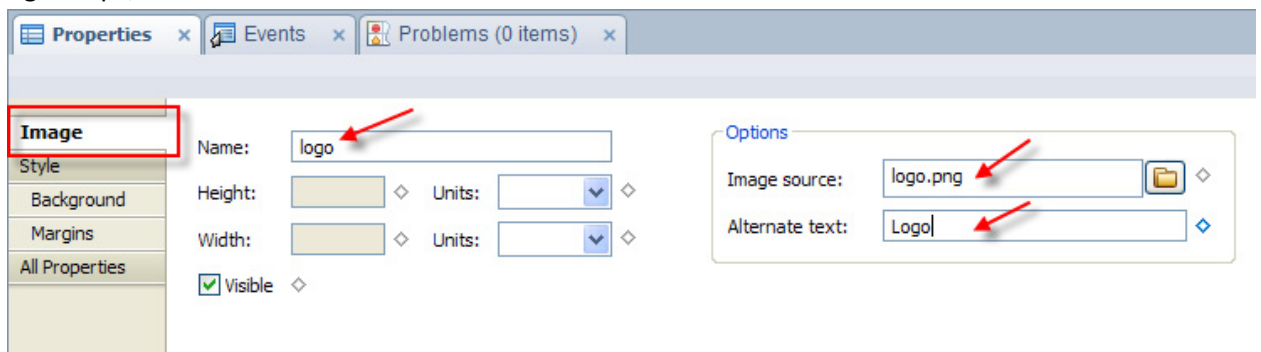
Step 6.1 : Creating custom control for global navigation

In this section, we are going to develop a custom control to hold the content for the top banner. This banner contains application logo, global links to navigate to other applications/sections and utility links for common tasks such as Log Out, Help, About, Contact. In One UI Framework, utility links and global links are implemented using unordered list HTML element, with each link wrapped within the list HTML element. In order to implement them in custom controls, we are using **Tabbed Panel** control, which gets translated at runtime into , with each tab as at run time. In addition, **Tabbed Panel** control provides all the benefits of XPages controls such as drag and drop, visual property setting, properties can be dynamically computed, etc. **Tabbed Panel** also provides useful styling options such as **startTabStyleClass**, **endTabStyleClass**, **selectedTabStyleClass** and ability to dynamically compute the selected tab – all very useful for web applications.

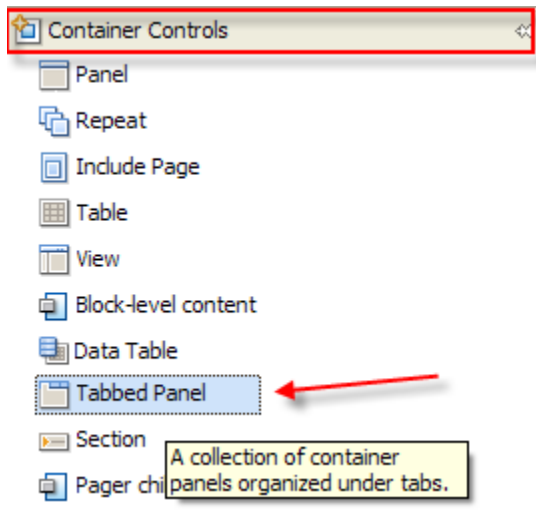
- Click on “New Custom Control” button and enter “**global_banner**” as the name. Click **OK**.
- Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Global Custom Controls**” as the category. This moves this custom control under the category “**Global Custom Controls**”.



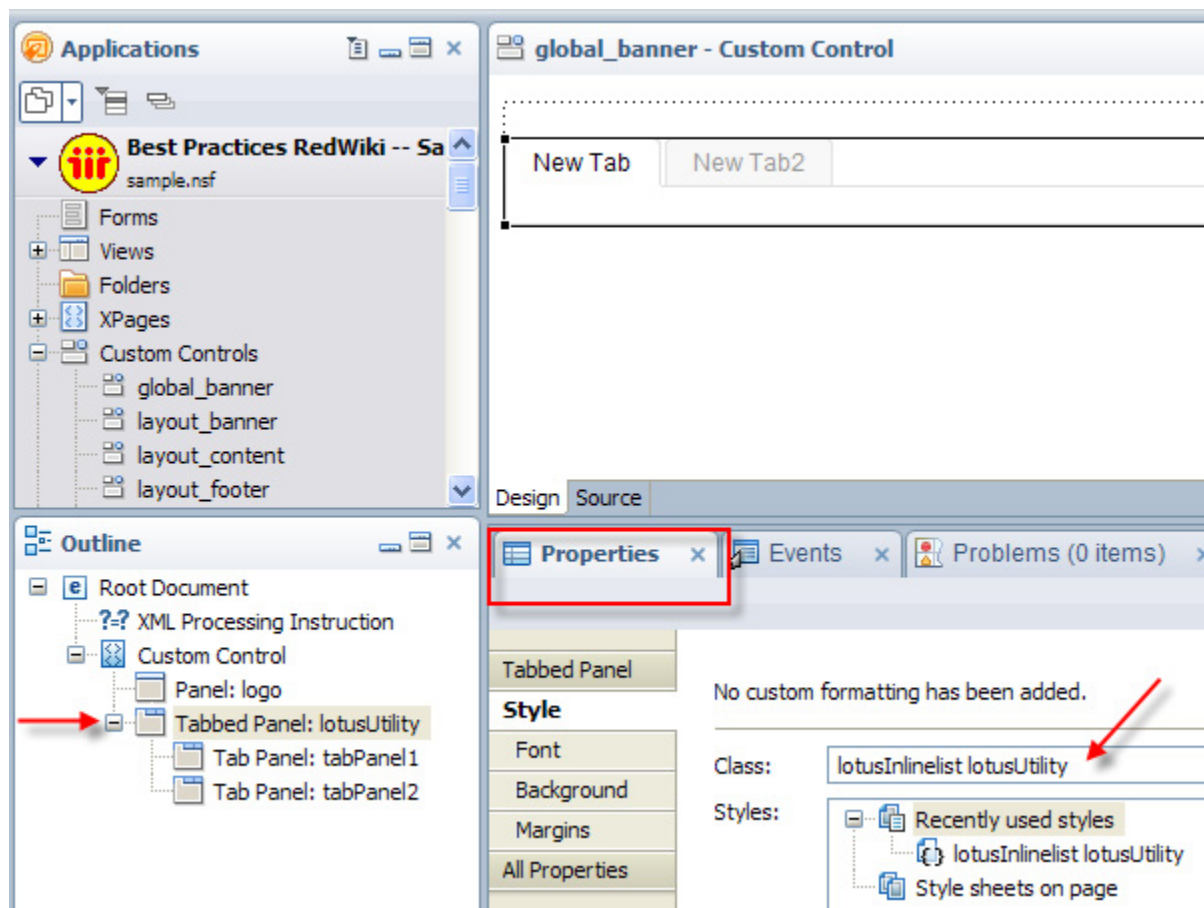
- From the control palette, drag and drop an **Image** control to the top-left. Enter the following properties: **Name:** logo, **Image Source:** logo.png, **Alternate text:** Logo, **Style:** float:left; margin-right:20px;



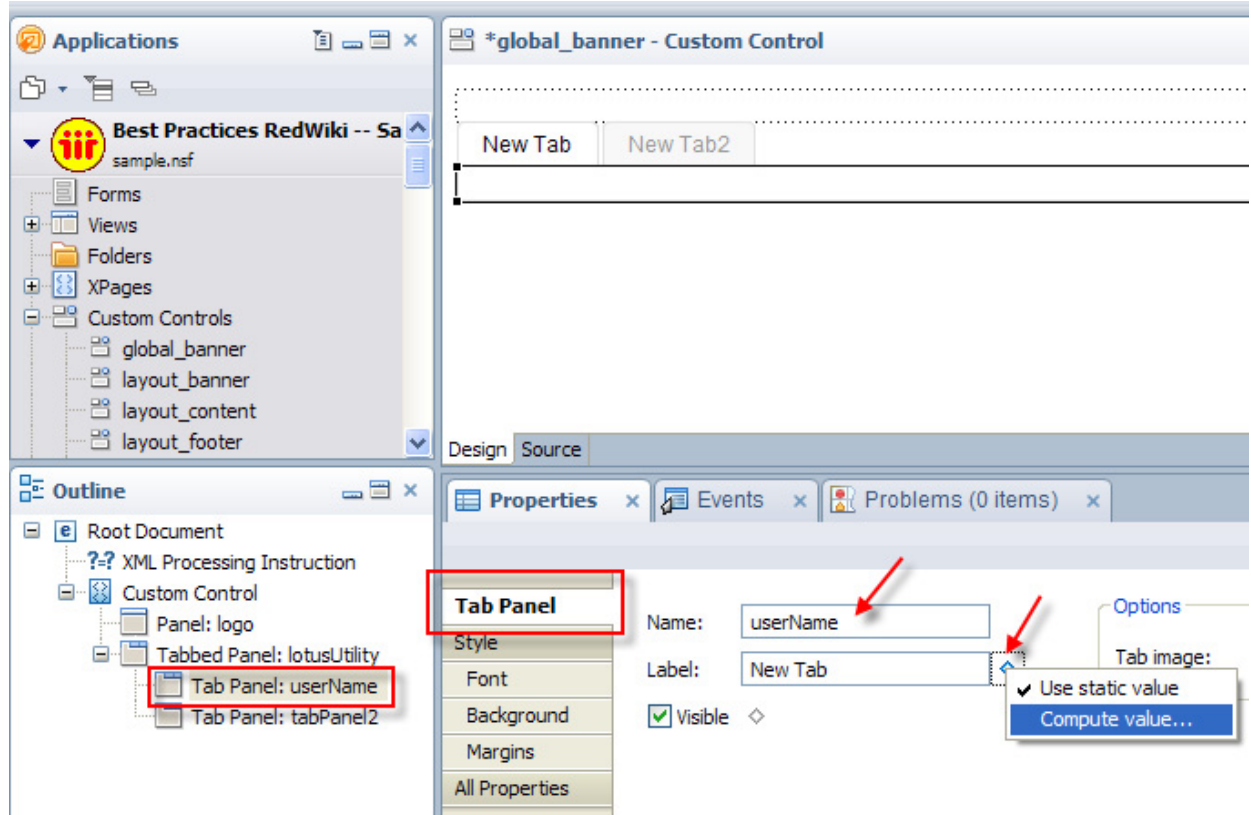
- From the control palette, under the **Container Controls** section, drag and drop **Tabbed Panel** control below the “logo” panel created in the last step.



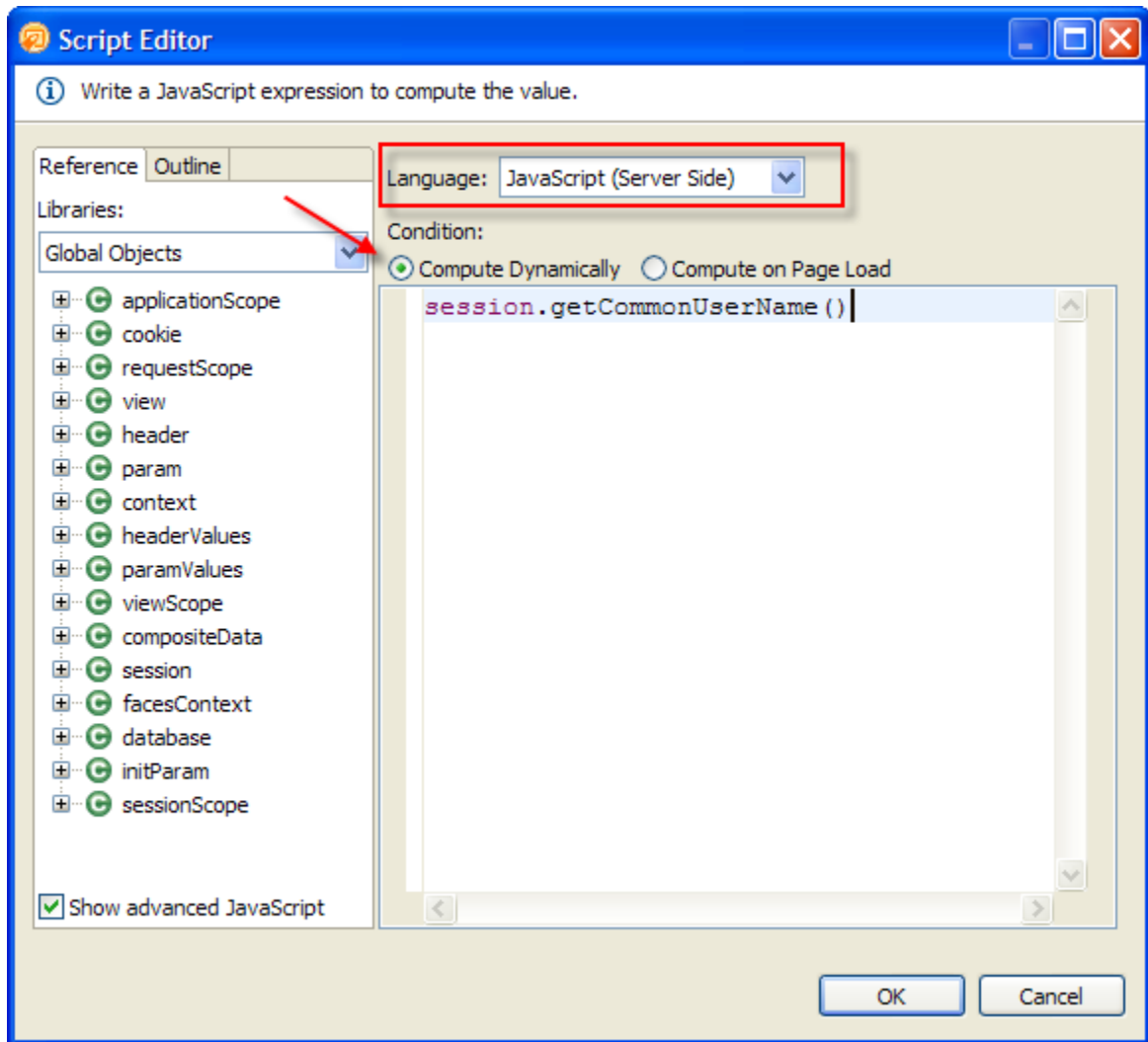
- From the outline palette, select the newly created Tabbed Panel and enter “**lotusUtility**” as its name and “**lotusInlinelist lotusUtility**” as CSS style class.



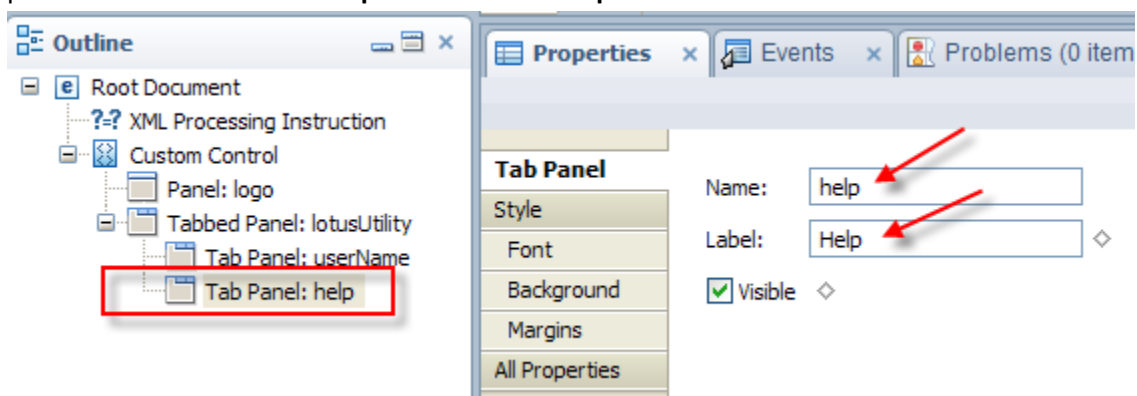
6. From the outline palette, expand **Tabbed Panel** and select the first tab panel. From the properties palette, change its **Name** property to “**userName**”. Click on the diamond icon next to the **Label** property and select “**Computed value...**”.



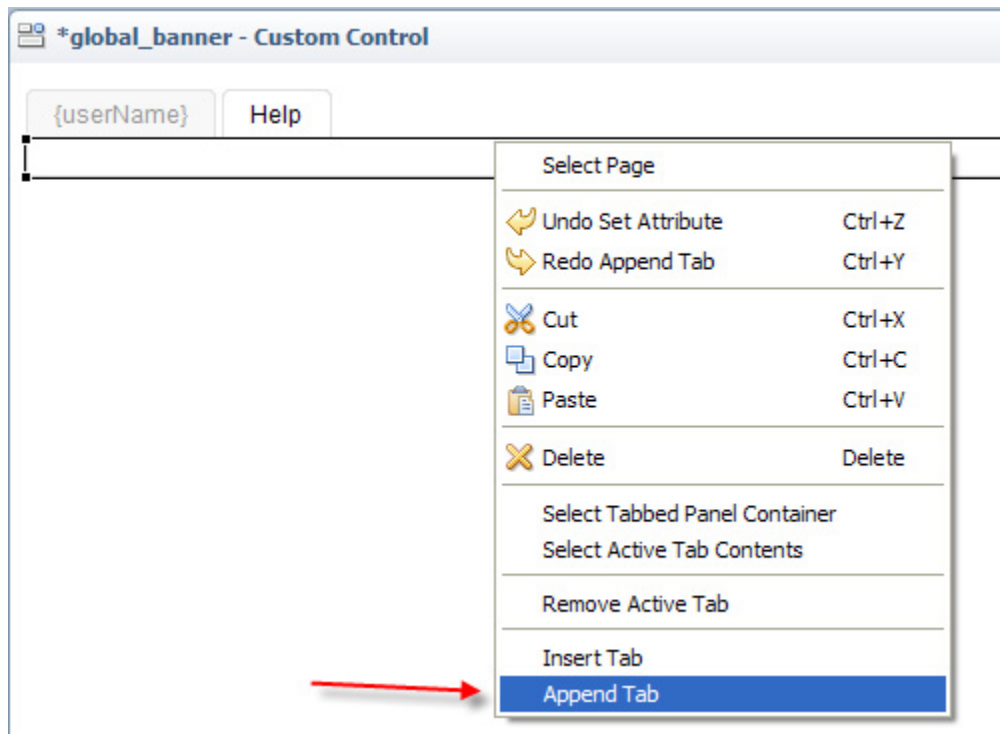
9. When you click on “**Computed Value**” in the last step, it opens a script editor. Enter “**session.getCommonUserName()**” in script editor and click OK. This will display the logged-in user’s common name in this tab at run time.



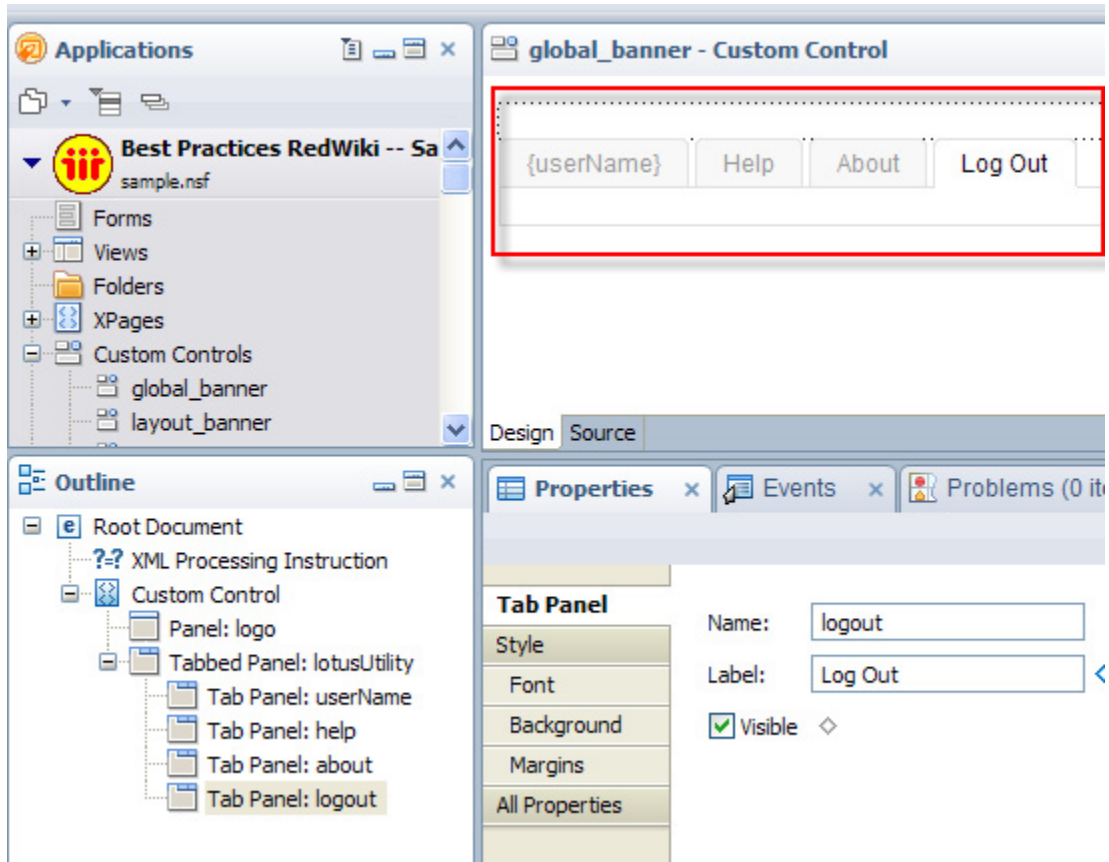
7. From the outline palette, click on the second Tab Panel to select it and from the properties palette enter its name as **“help”** and label as **“Help”**.



8. On the custom control editor in design mode, right click the **Tabbed Panel** and select **“Append Tab”**. This appends a new tab. Name this new tab as **“about”** and enter its label property as **“About”**.



9. On the custom control editor in design mode, right click the Tabbed Panel and select “**Append Tab**”. This appends a new tab. Name this new tab as “**logout**” and enter its label property as “**Log Out**”.



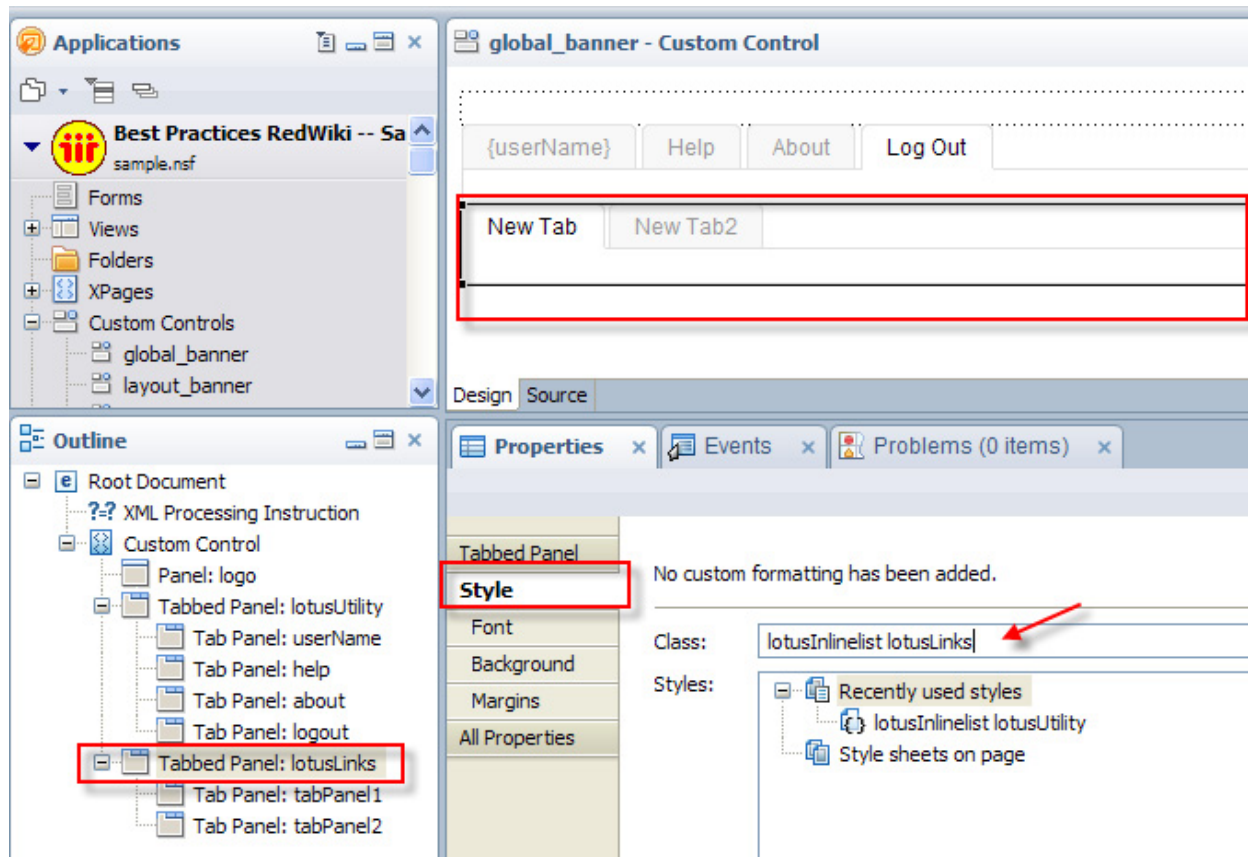
10. Click on "Source" tab in xPages editor and press Ctr-Shift-F to format the source code and save the changes. You should see the following code:

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

    <xp:image url="logo.png" id="logo"
        style="float:left; margin-right:20px;" alt="Logo">
    </xp:image>

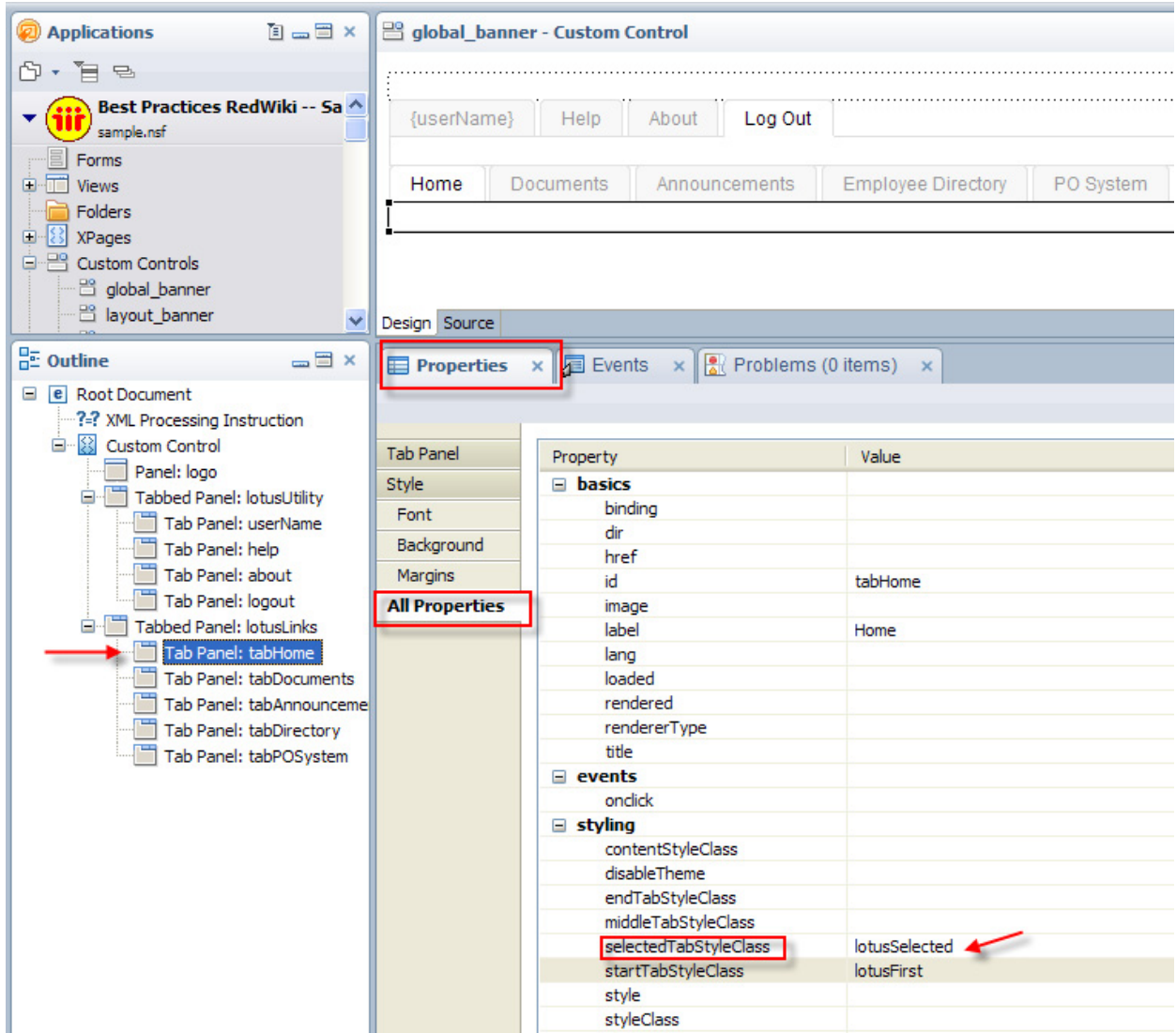
    <xp:tabbedPanel id="lotusUtility" styleClass="lotusInlinelist
lotusUtility">
        <xp:tabPanel label="{javascript:session.getCommonUserName()}"
            id="userName" styleClass="userName">
        </xp:tabPanel>
        <xp:tabPanel id="help" label="Help"></xp:tabPanel>
        <xp:tabPanel id="about" label="About"></xp:tabPanel>
        <xp:tabPanel label="Log Out" id="logout"></xp:tabPanel>
    </xp:tabbedPanel>
</xp:view>
```

11. We have created a tabbed panel for utility links. Now we need to create a tabbed panel for the application links. Switch to design mode and drag another **Tabbed Panel** control and drop it below the first tabbed panel. Enter “**lotusLinks**” as its name and “**lotusInlinelist lotusLinks**” as CSS class.

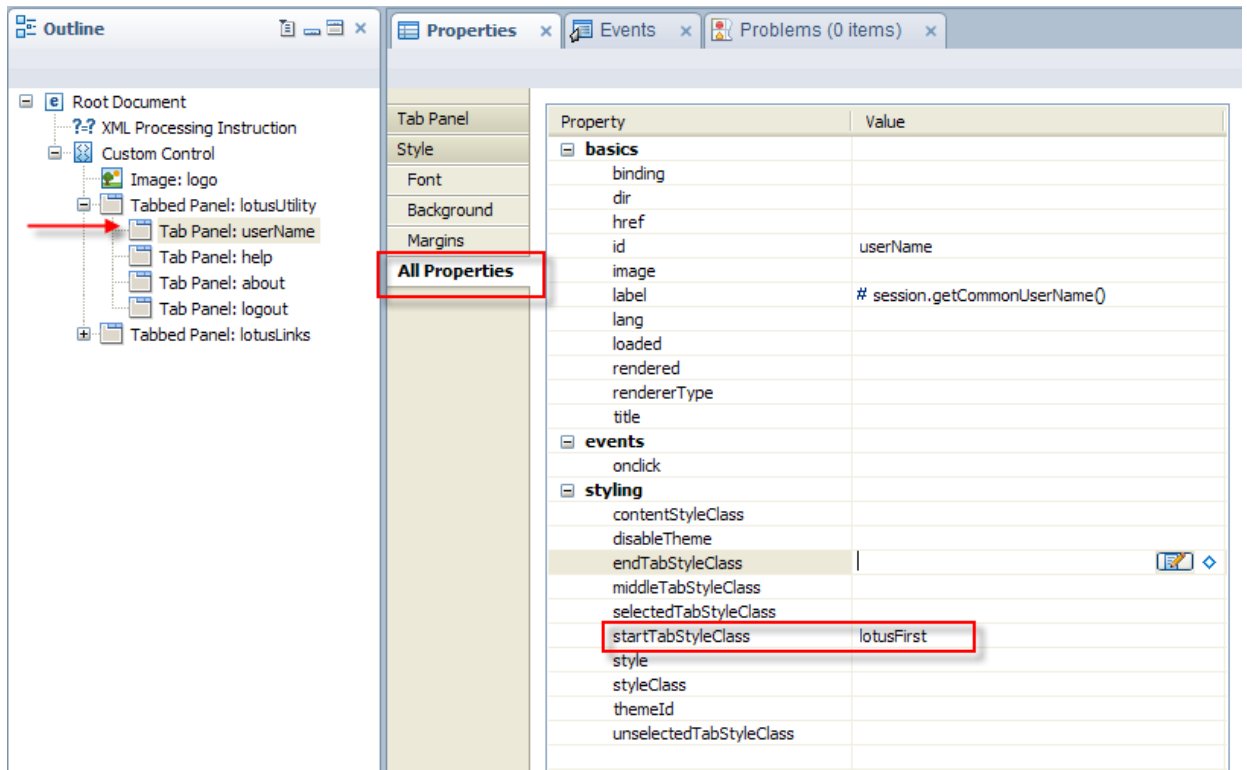


12. From the custom control editor in design mode, right click the “lotusLinks” Tabbed Panel and select “Append Tab”. This appends a new tab. Append two more tabs so that we have five tabs within this tabbed panel. From the outline palette, starting from the first tab, select each Tab Panel one by one, and enter the following property values:

Tab	Name	Label	startTabStyleClass	selectedTabStyleClass
1	tabHome	Home	lotusFirst	lotusSelected
2	tabDocuments	Documents		lotusSelected
3	tabAnnouncements	Announcements		lotusSelected
4	tabDirectory	Employee Directory		lotusSelected
5	tabPOSystem	PO System		lotusSelected



- For the first tab in the "lotusLinks" and "lotusUtility" tabbed panel, enter "lotusFirst" as "startTabStyleClass" property value. This needs to be done only on the first tab.



14. Click on “Source” tab in xPages editor and press Ctrl-Shift-F to format the source code and save the changes. You should see the following code:

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

    <xp:image url="logo.png" id="logo" style="float:left; margin-right:20px;"
        alt="Logo">
    </xp:image>

    <xp:tabbedPanel id="lotusUtility" styleClass="lotusInlinelist
lotusUtility">
        <xp:tabPanel label="#{javascript:session.getUserName()}"
            id="userName" startTabStyleClass="lotusFirst">
        </xp:tabPanel>
        <xp:tabPanel id="help" label="Help"></xp:tabPanel>
        <xp:tabPanel id="about" label="About"></xp:tabPanel>
        <xp:tabPanel label="Log Out" id="logout"></xp:tabPanel>
    </xp:tabbedPanel>

    <xp:tabbedPanel id="lotusLinks" styleClass="lotusInlinelist
lotusLinks">
        <xp:tabPanel label="Home" id="tabHome"
            selectedTabStyleClass="lotusSelected"
startTabStyleClass="lotusFirst">
        </xp:tabPanel>
        <xp:tabPanel label="Documents" id="tabDocuments"
            selectedTabStyleClass="lotusSelected">
```

```

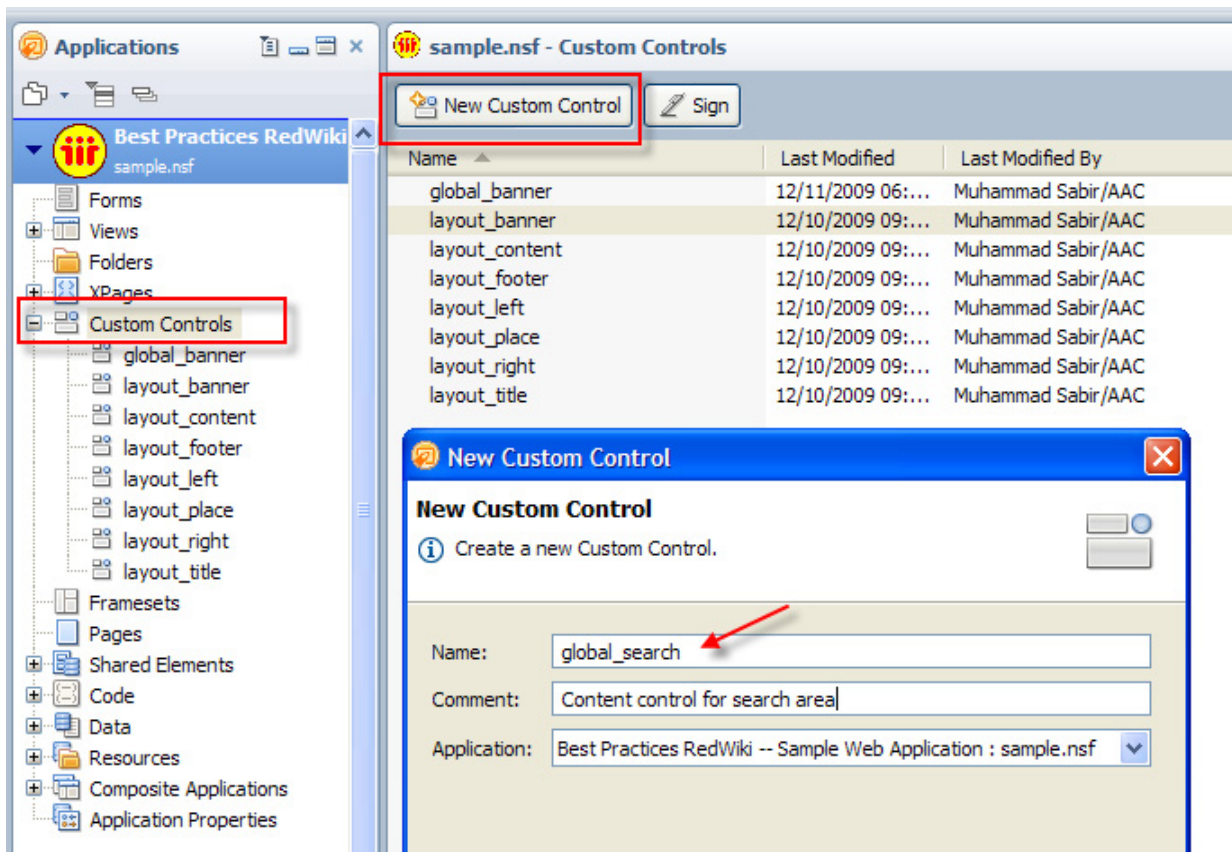
</xp:tabPanel>
<xp:tabPanel label="Announcements" id="tabAnnouncements"
  selectedTabStyleClass="lotusSelected">
</xp:tabPanel>
<xp:tabPanel label="Employee Directory" id="tabDirectory"
  selectedTabStyleClass="lotusSelected">
</xp:tabPanel>
<xp:tabPanel label="PO System" id="tabPOSystem"
  selectedTabStyleClass="lotusSelected">
</xp:tabPanel>
</xp:tabbedPanel>
</xp:view>

```

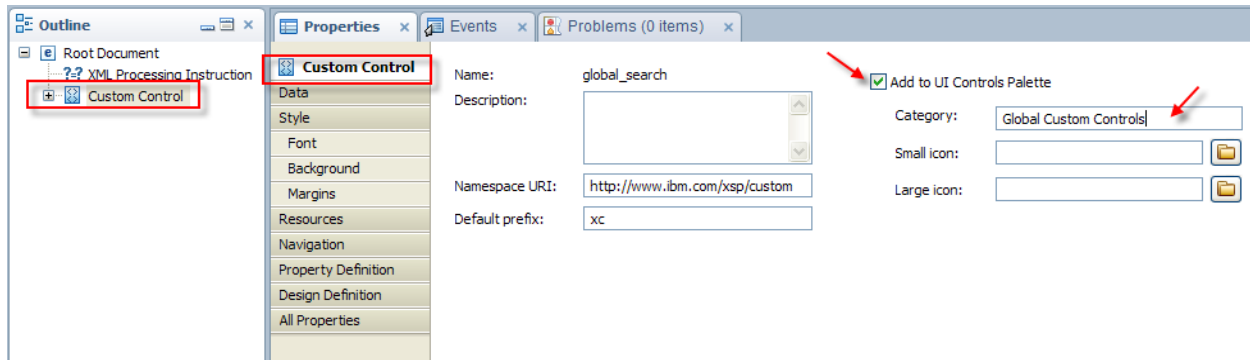
Step 6.2: Creating custom control for search input

We want to implement search feature in our application so that user can search for the content they are looking for. Since the search feature can optionally be included in any xPage, it makes sense to implement this as a custom control. In One UI framework, search control is included within the title bar. We implemented title bar as just a layout control and did not include any content related elements in it; therefore, we are implementing the search control as a separate custom control.

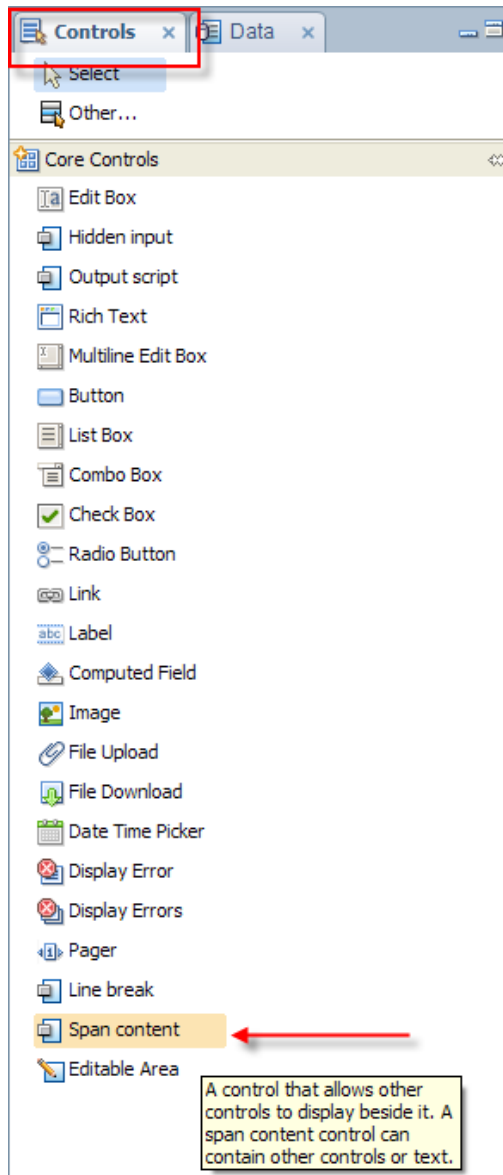
1. Click on “New Custom Control” button and enter “**global_search**” as control name. Click **OK**.



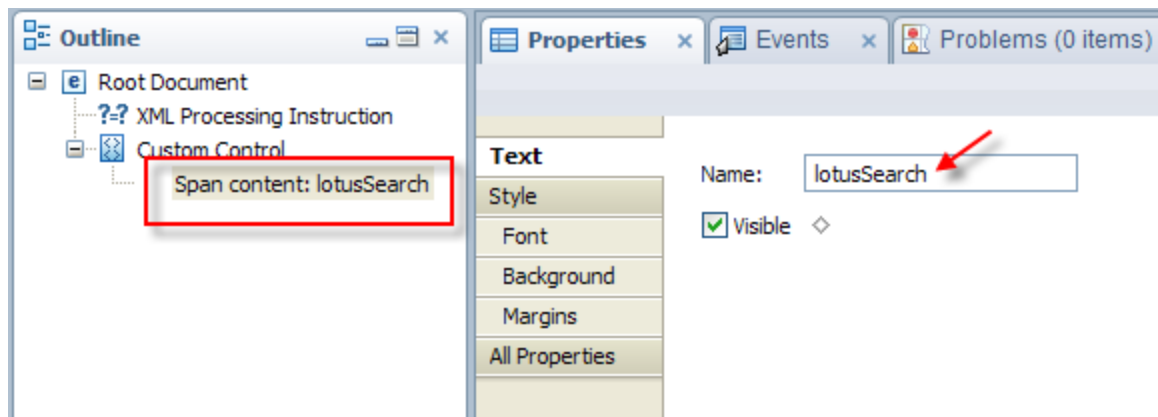
- Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Global Custom Controls”** as the category. This moves this custom control under the category **“Global Custom Controls”**.

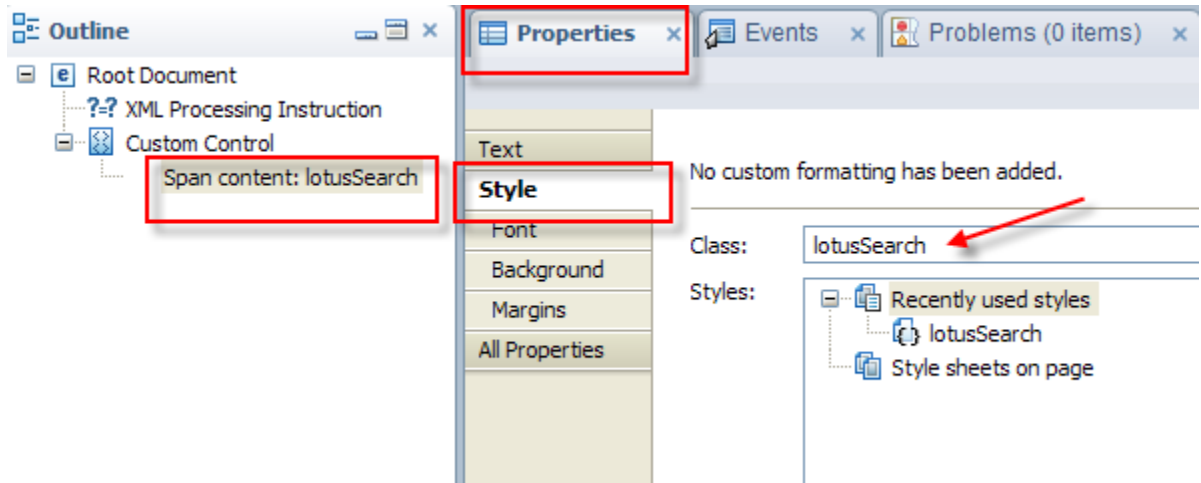


- From the core controls section, drag and drop **“Span Content”** control to the top-left in design pane. This control translates to **** html tag at runtime.

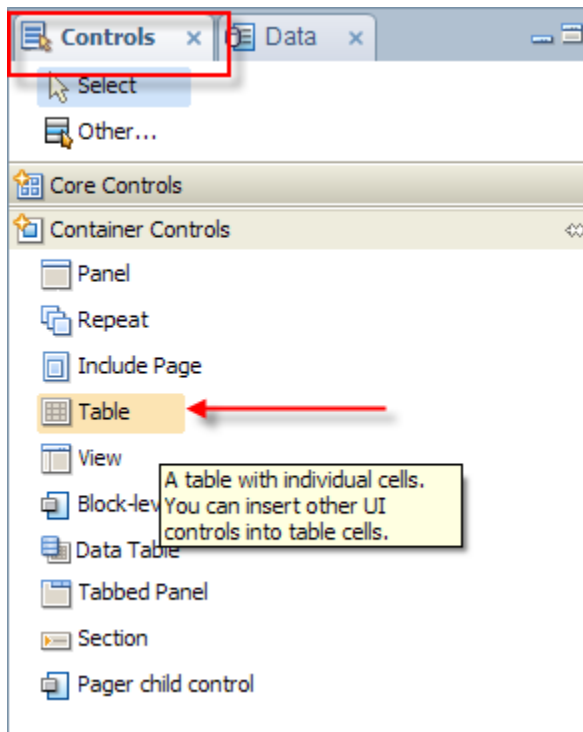


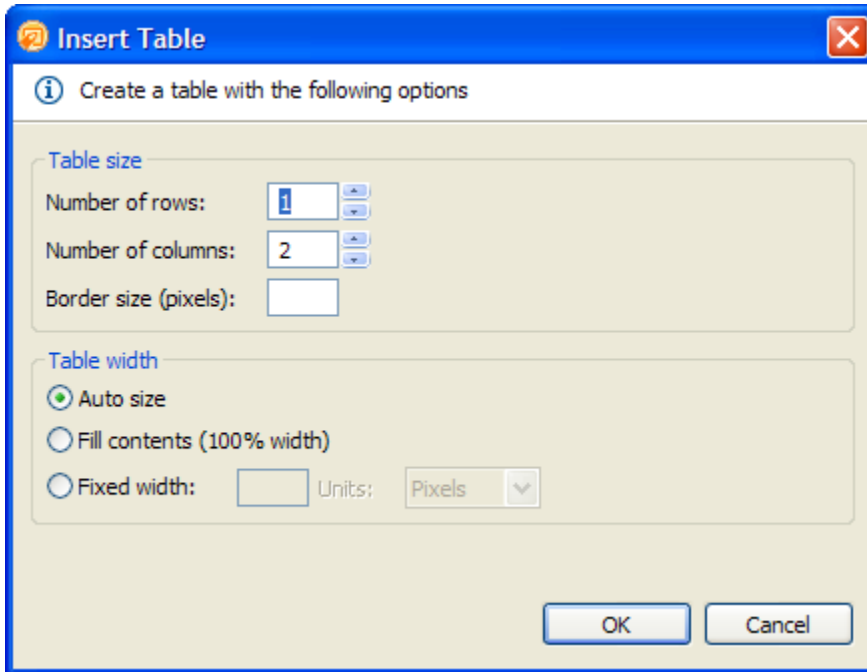
4. Enter "lotusSearch" both as name and style class.



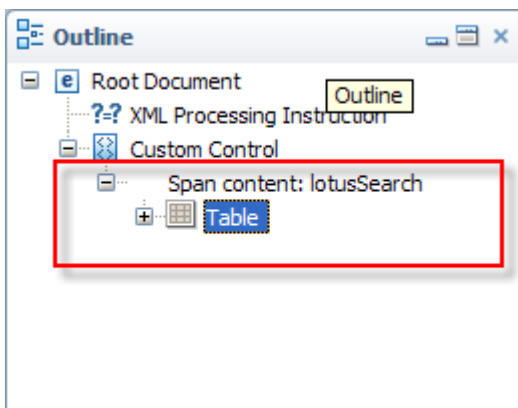
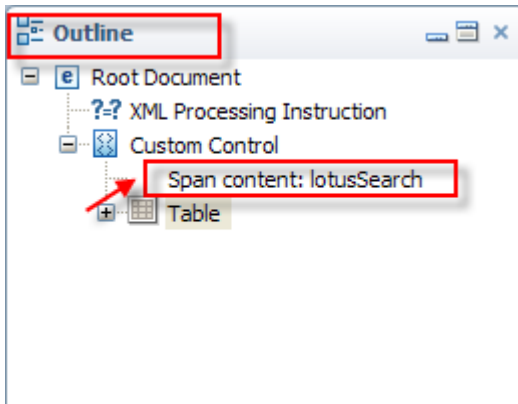


5. From the container controls, drag and drop a **Table** control to the design pane. Enter “**number of columns**” as **1** and “**number of rows**” as **2**. Make sure table width is set to “Auto Size”.

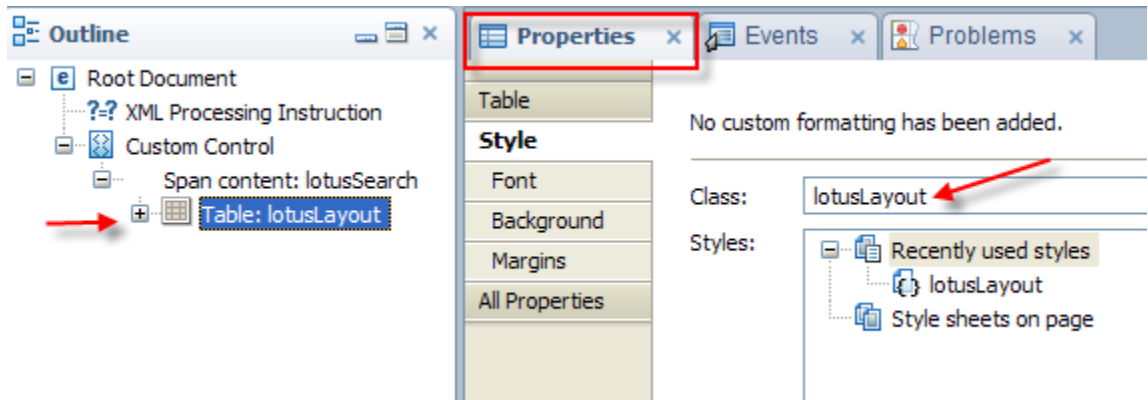




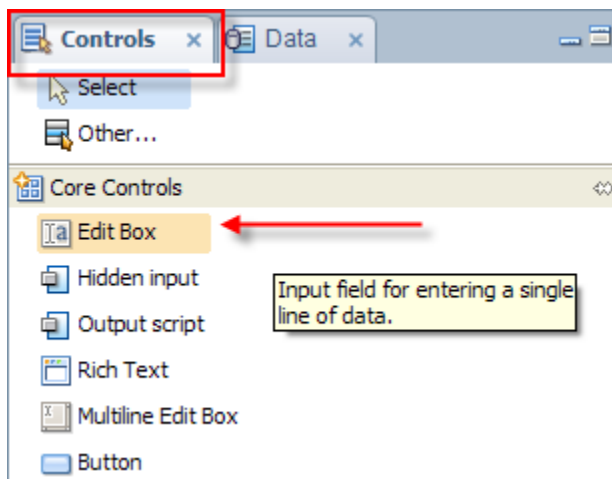
6. From the outline palette, drag and move the table control to “**lotusSearch**” span content.



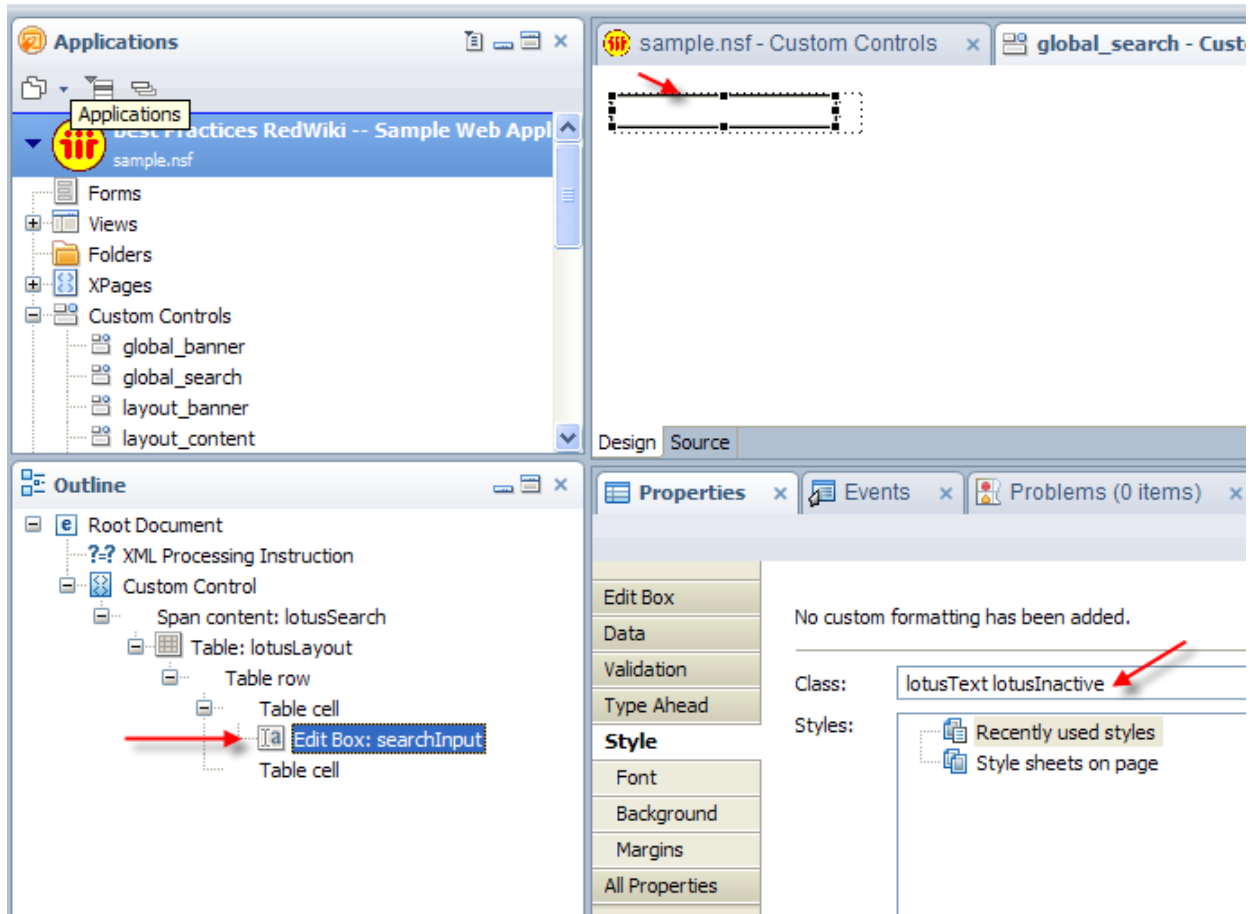
7. Enter “**lotusLayout**” both as table name and its CSS class property.



8. From the core controls, drag and drop an “**Edit Box**” control to the first cell on the table



9. Enter “**searchInput**” as the edit box name and “**lotusText lotusInactive**” as its CSS class property

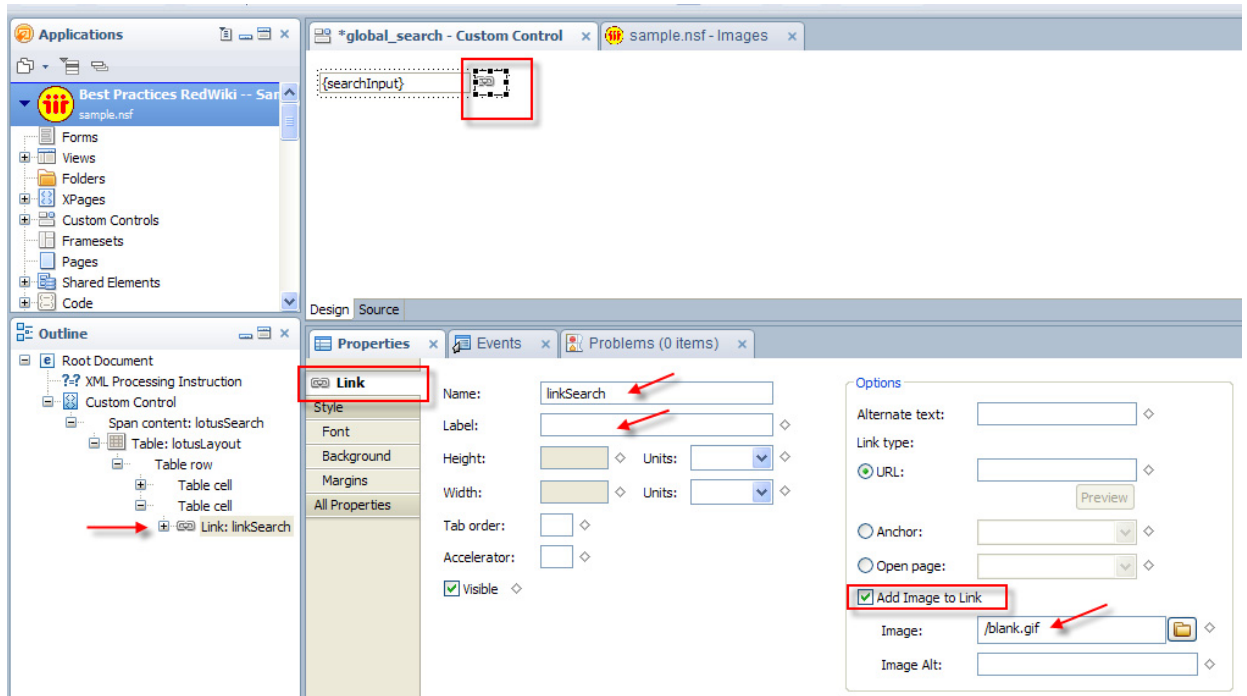


10. Select the “**searchInput**” edit box, select “**All Properties**” tab from the properties palette. Enter the following properties:
- title=Enter Keywords
 - defaultValue=”Enter Keywords”
 - onFocus=this.value=”
 - value=sessionScope.searchValue (computed value)

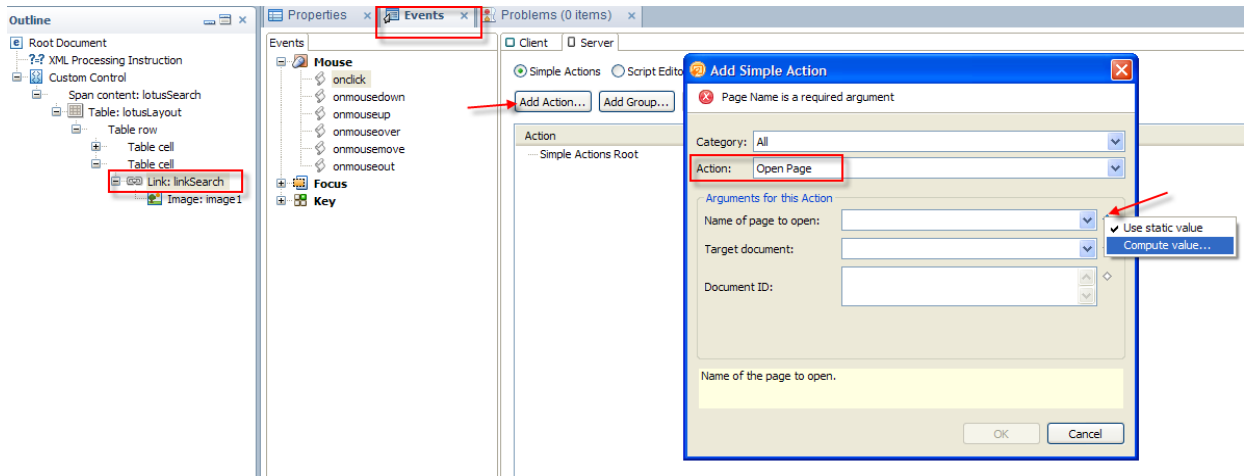
The screenshot shows the Domino Properties palette with the 'All Properties' section selected. The 'basics' and 'data' sections are expanded. The 'value' property is set to '# sessionScope.searchValue'. A context menu is open over the 'value' property, showing 'Use static value' and 'Compute value...' options. Red arrows point to the 'Enter Keywords' text, the 'value' property, and the 'Compute value...' option.

Property	Value
basics	
accesskey	
autocomplete	
binding	
dir	
disabled	
disableModifiedFlag	
htmlFilter	
htmlFilterIn	
id	searchInput
immediate	
lang	
loaded	
maxlength	
multipleSeparator	
multipleTrim	
password	
readonly	
redisplay	
rendered	
rendererType	
required	
size	
tabindex	
title	Enter Keywords
data	
converter	
defaultValue	Enter Keywords
disableClientSideValidation	
validator	
validators	
value	# sessionScope.searchValue
valueChangeListener	
valueChangeListeners	
events	
onblur	
onchange	
onclick	
ondblclick	
onfocus	this.value = "
onkeydown	

- From the core controls, drag and drop a **Link** control to the second cell in the table. From the properties palette, enter "**linkSearch**" as its name and make the label property as blank. Check "**Add Image to the Link**" box under "**Options**" section on the right. Click on the folder icon next to the **Image** field and select "**blank.gif**" from the image list.

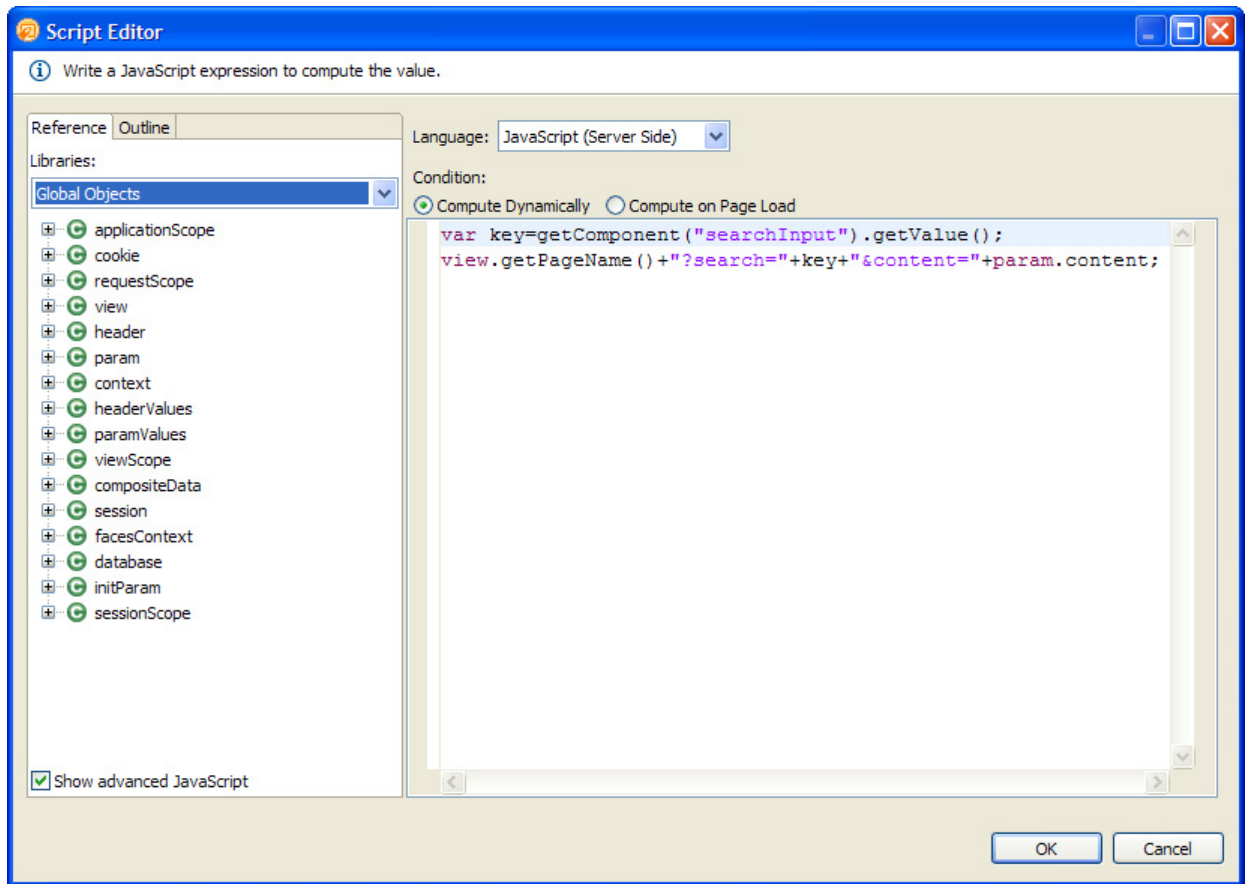


12. Click on the events tab and select “onclick” event. Click “Add Action” and select “Open Page” action from the drop down. Click on the diamond icon in front of the “Name of Page to Open” and select “computed”



13. Enter the following **Server Side JavaScript** code in the script editor pop-up and click **OK**.

```
var key=getComponent("searchInput").getValue();
view.getPageName()+"?search="+key+"&content="+param.content;
```



15. Click on "Source" tab in XPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

    <xp:span id="lotusSearch" styleClass="lotusSearch">
        <xp:table styleClass="lotusLayout" id="lotusLayout">
            <xp:tr>
                <xp:td>
                    <xp:inputText id="searchInput"
styleClass="lotusText lotusInactive"
                    title="Enter Keywords"
defaultValue="Enter Keywords" value="#{sessionScope.searchValue}"
                    onfocus="this.value=''">
                </xp:inputText>
            </xp:td>
            <xp:td>
                <xp:span styleClass="lotusBtnImg" title="Submit
search" id="lotusBtnImg">
                    <xp:link escape="true" id="linkSearch">
                        <xp:image
styleClass="lotusSearchButton" id="image1" url="/blank.gif"></xp:image>
                    <xp:eventHandler event="onclick"
submit="true" refreshMode="complete">
```

```

        <xp:this.action>
            <xp:openPage>

        <xp:this.name><![CDATA[#{javascript:var
key=getComponent("searchInput").getValue();
view.getPageName()+"?search="+key+"&content="+param.content;
}]]></xp:this.name>

    </xp:openPage>

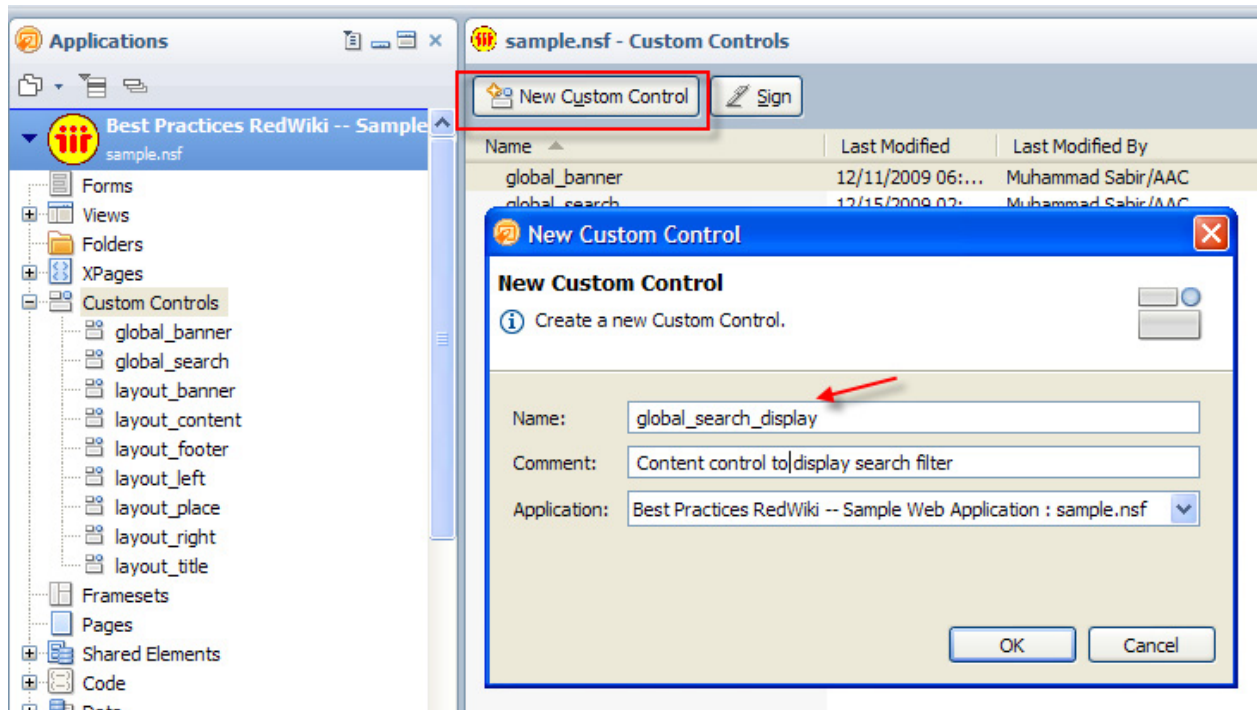
</xp:this.action></xp:eventHandler></xp:link>
    </xp:span>
</xp:td>
</xp:tr>
</xp:table>
</xp:span>
</xp:view>

```

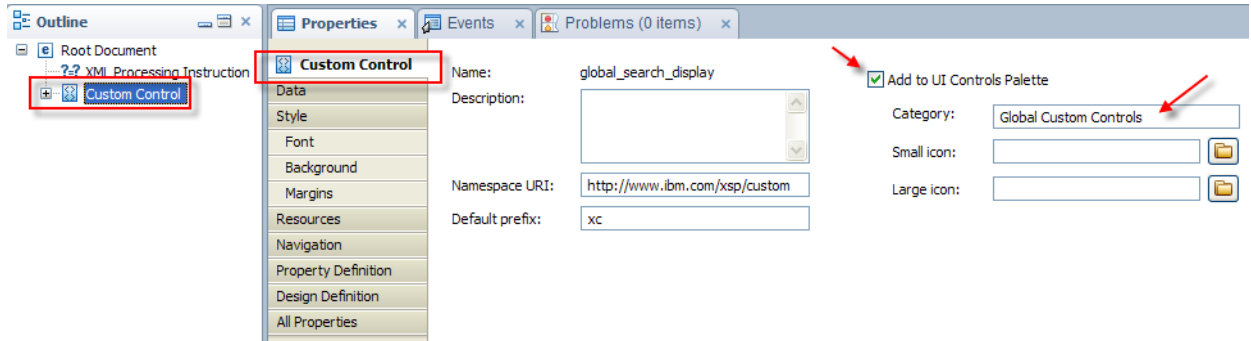
Step 6.3: Creating custom control to indicate search filter

The sample application provides a search feature where user can enter search criteria and the search results will be displayed on an xPage. The application needs to indicate that the web page is displaying filtered view by listing those documents that match the search criteria. Also, users should be able to clear the search filter to view all documents. Since this feature is applicable to various XPages, we are going to implement it using a custom control.

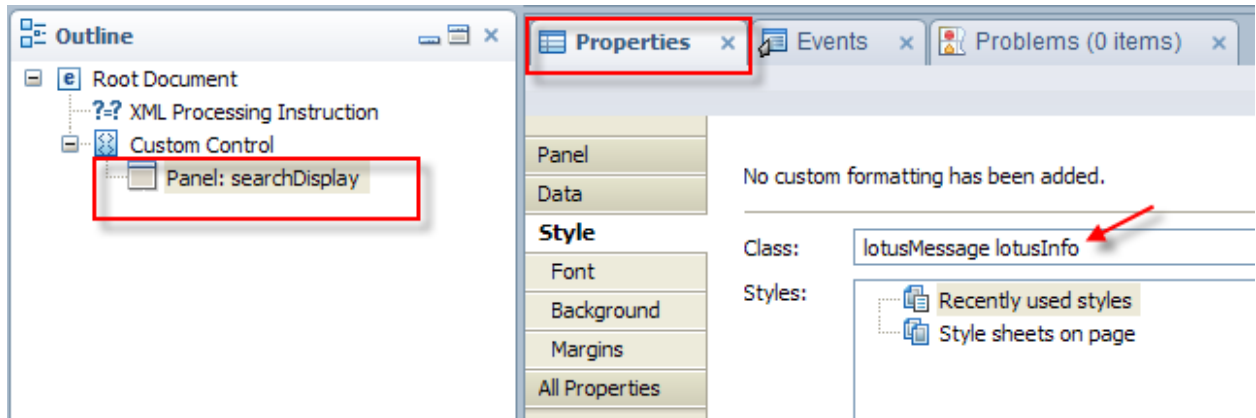
1. From the custom controls list, click on “**New Custom Control**” and enter control name as “**global_search_display**”. Click OK



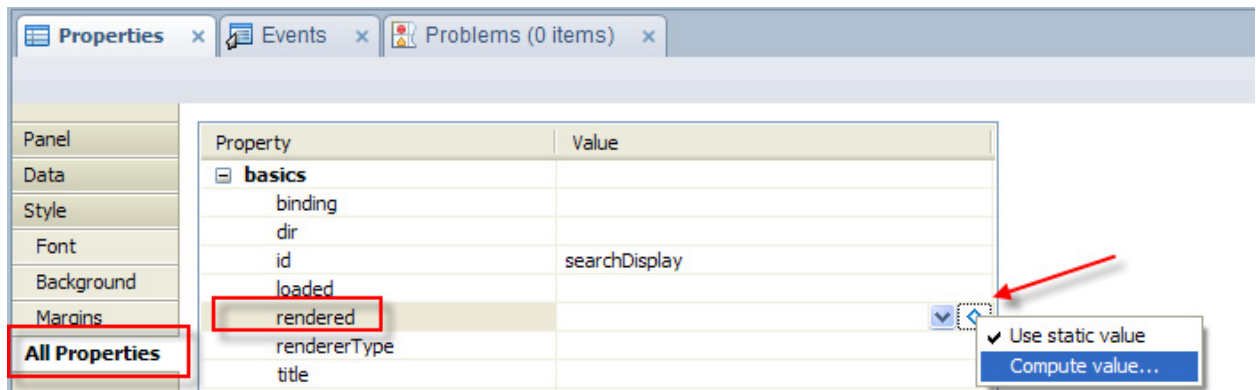
- Under the Properties tab, make sure Custom Control is selected and check **"Add to UI Controls Palette"** and enter **"Global Custom Controls"** as the category. This moves this custom control under the category **"Global Custom Controls"**.



- From the container controls, drag and drop a **Panel** control to design pane. From the properties palette, enter name as **"searchDisplay"** and CSS style class as **"lotusMessage lotusInfo"**

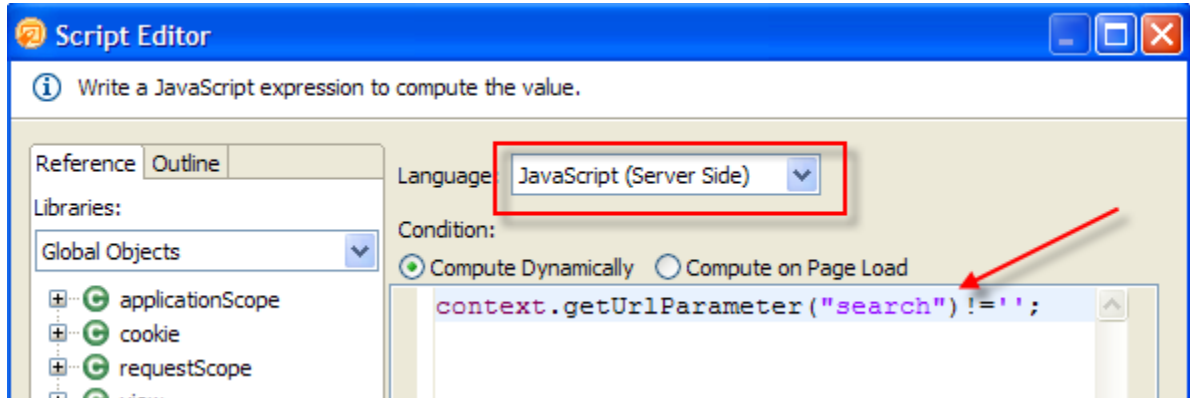


- Click on **"All Properties"** tab and select **"Computed Value.."** from the diamond icon in front of the rendered property.

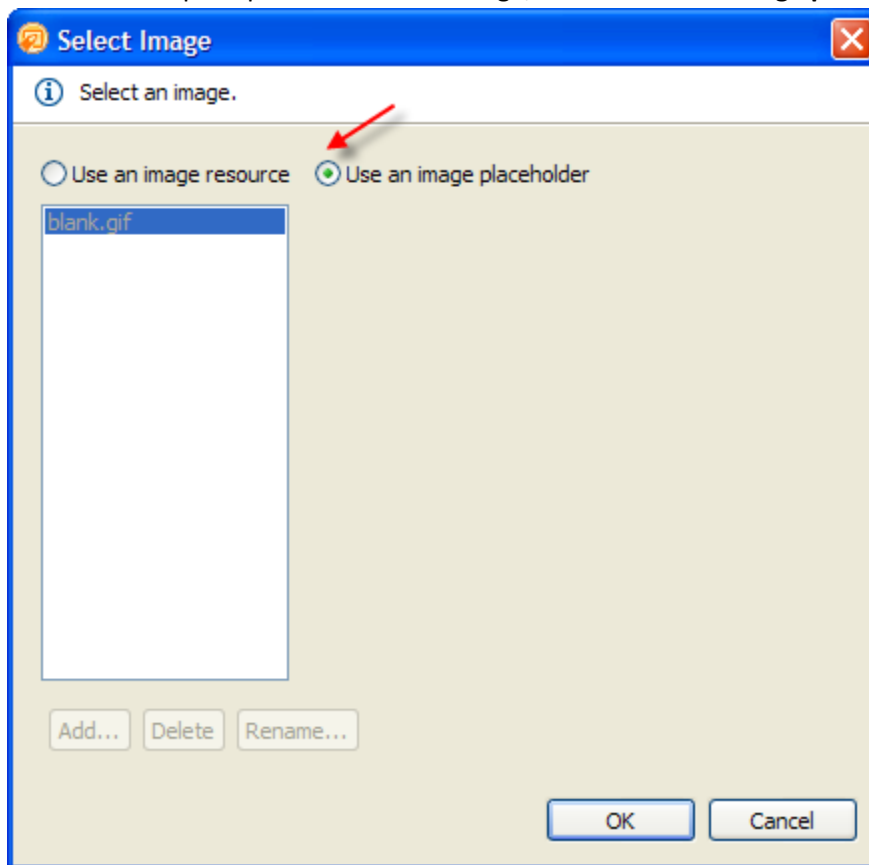


- From the pop-up script editor, enter the following server side JavaScript. This is done to hide this panel if there is no search filter applied to a given view:

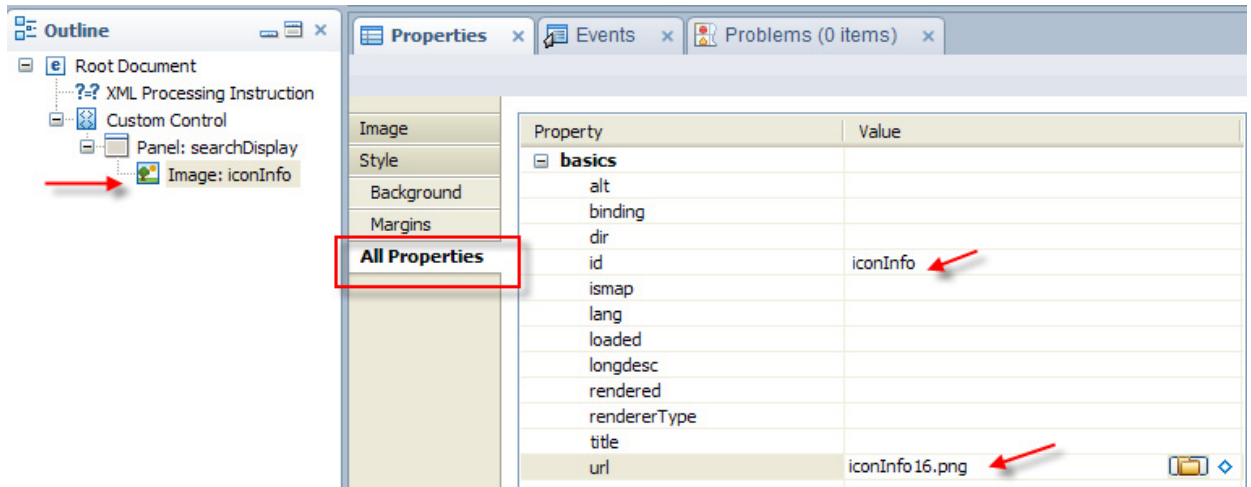
context.getUrlParameter("search")!="";



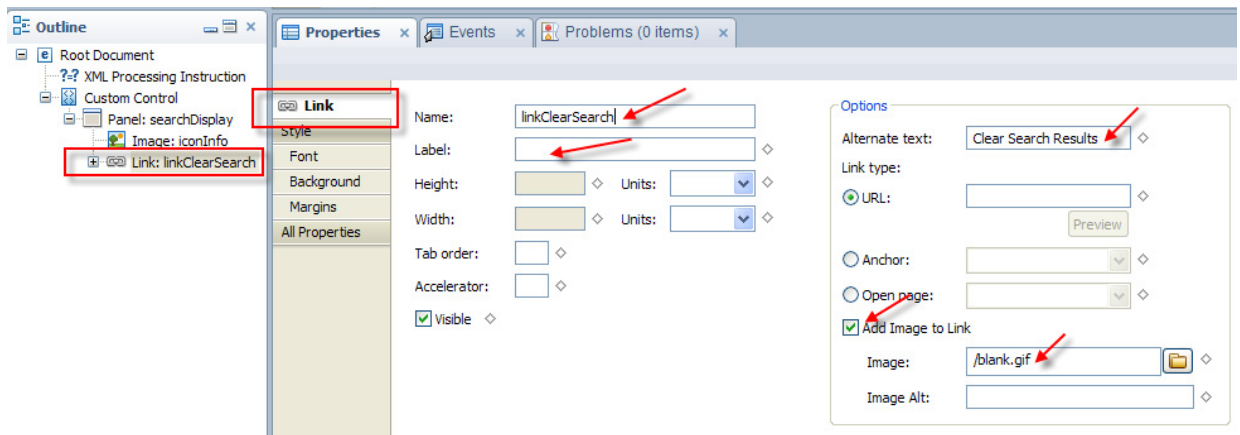
- From the core controls, drag and drop an image control to the “searchDisplay” panel created earlier. When prompted to select an image, select “use and image place holder” option



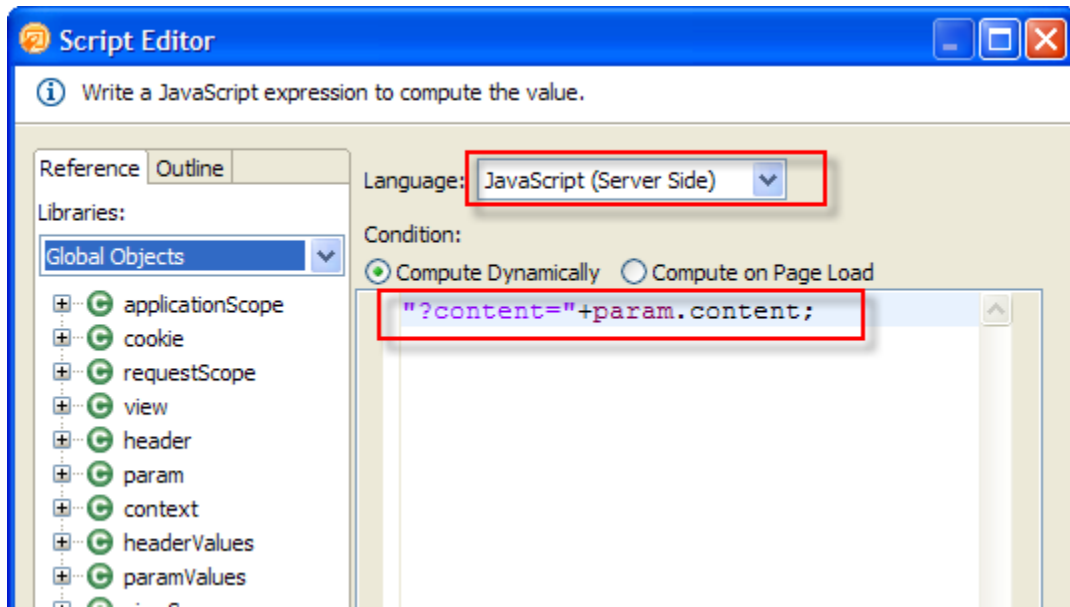
- Name the image as “**iconInfo**” and enter its url property as “**iconInfo16.png**”.



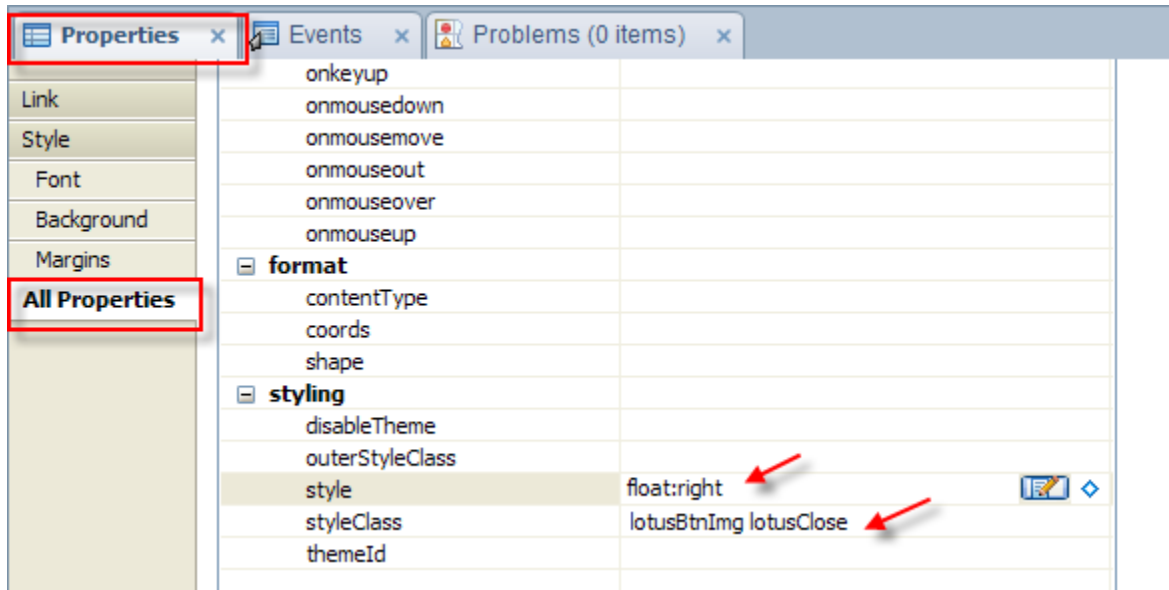
- From the core controls, drag and drop a **Link** control to the right of **iconInfo** control created earlier. From the properties palette, enter name as “**linkClearSearch**” and alternate text as “**Clear Search Results**”. Remove label text so that it is a blank field. Under the options section, click on “**Add Image to Link**” and select **blank.gif**



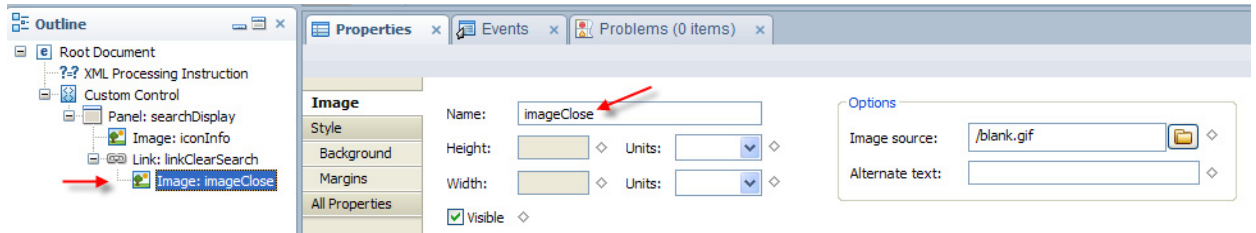
- Select “**computed**” for URL and enter the following code:
`"?content="+param.content;`



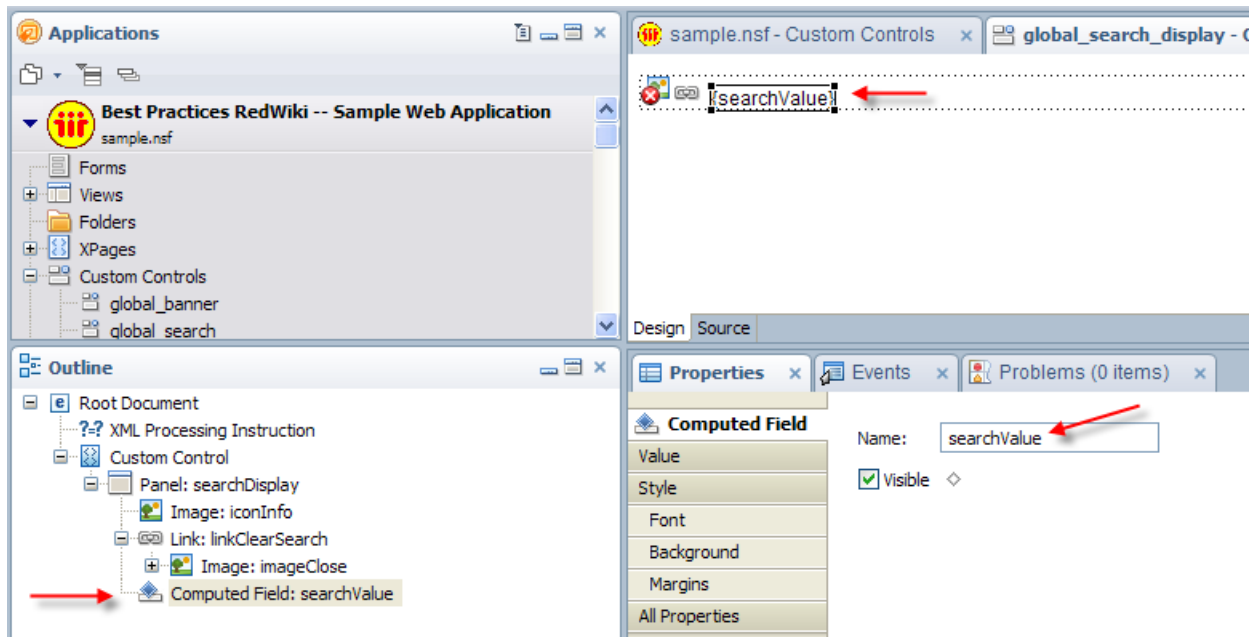
10. Click **All Properties** tab and enter **"lotusBtnImg lotusClose"** as **style class** and **"float:right"** as **style** property value.



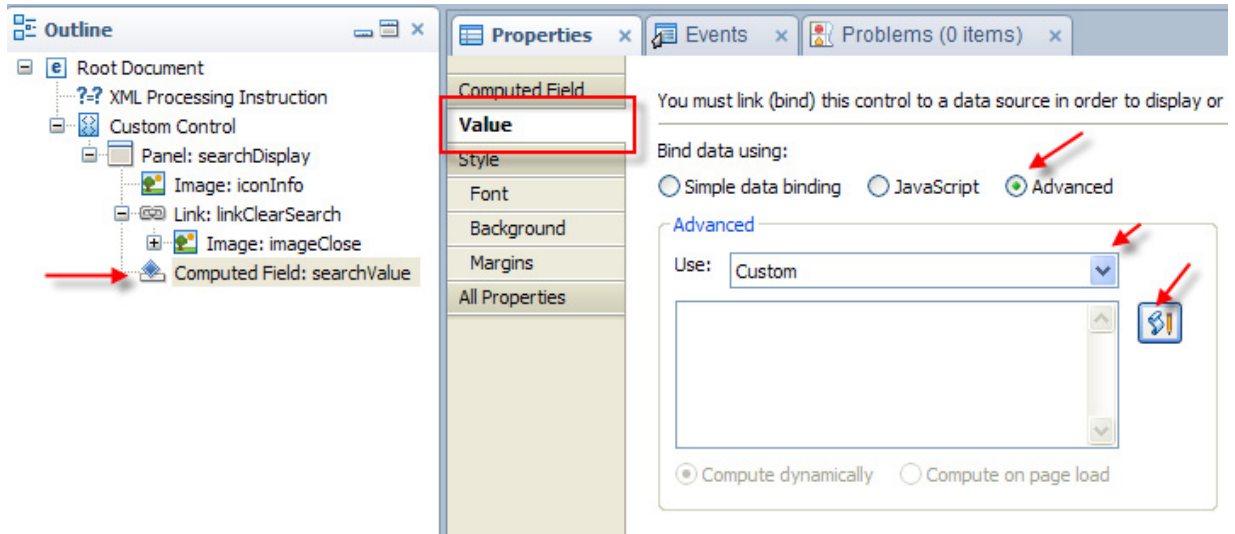
11. From the outline palette, expand **"linkClearSearch"** and select the image control. Name this image as **"imageClose"**.



12. From the core controls, drag and drop a **Computed Field** control to the right of **linkClearSearch** link created earlier. Enter its name property as “**searchValue**”.

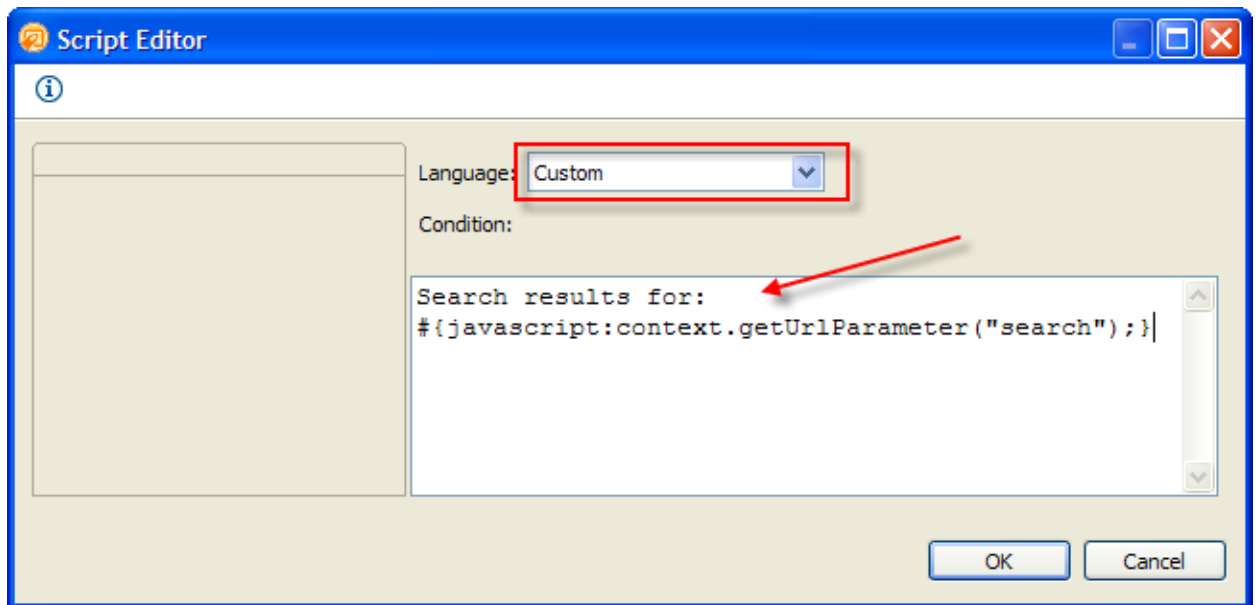


13. From the properties palette, click on the **Value** tab and select “**Advanced**” data binding option.



14. Select on the editor icon to open the JavaScript editor and enter the following code:

Search results for: #{javascript:context.getUrlParameter("search");}



15. Click on "Source" tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel id="searchDisplay" styleClass="lotusMessage lotusInfo">
```

```

    <xp:this.rendered><![CDATA[#{javascript:context.getUrlParameter("search")!='';}]]></xp:this.rendered>
    <xp:image url="iconInfo16.png" id="iconInfo"></xp:image>
    <xp:link escape="true" id="linkClearSearch" title="Clear Search
Results"
        style="float:right" styleClass="lotusBtnImg lotusClose">

    <xp:this.value><![CDATA[#{javascript:"?content="+param.content;}]]></xp:
:this.value><xp:image id="imageClose" url="/blank.gif">
    </xp:image>
    </xp:link>
    <xp:text escape="true" id="searchValue">
        <xp:this.value><![CDATA[Search results for:
#{javascript:context.getUrlParameter("search");}]]></xp:this.value>
    </xp:text>
    </xp:panel>
</xp:view>

```

Step 6.4 : Creating custom control for tool tip

Dojo is one of the most popular open source DHTML toolkits which provide out-of-the-box UI elements for web application. Domino 8.5.1 provides a built-in support for dojo. Dojo libraries are installed in (data)\domino\js\dojo-x.x.x\dojo\ (where x.x.x is the dojo version)

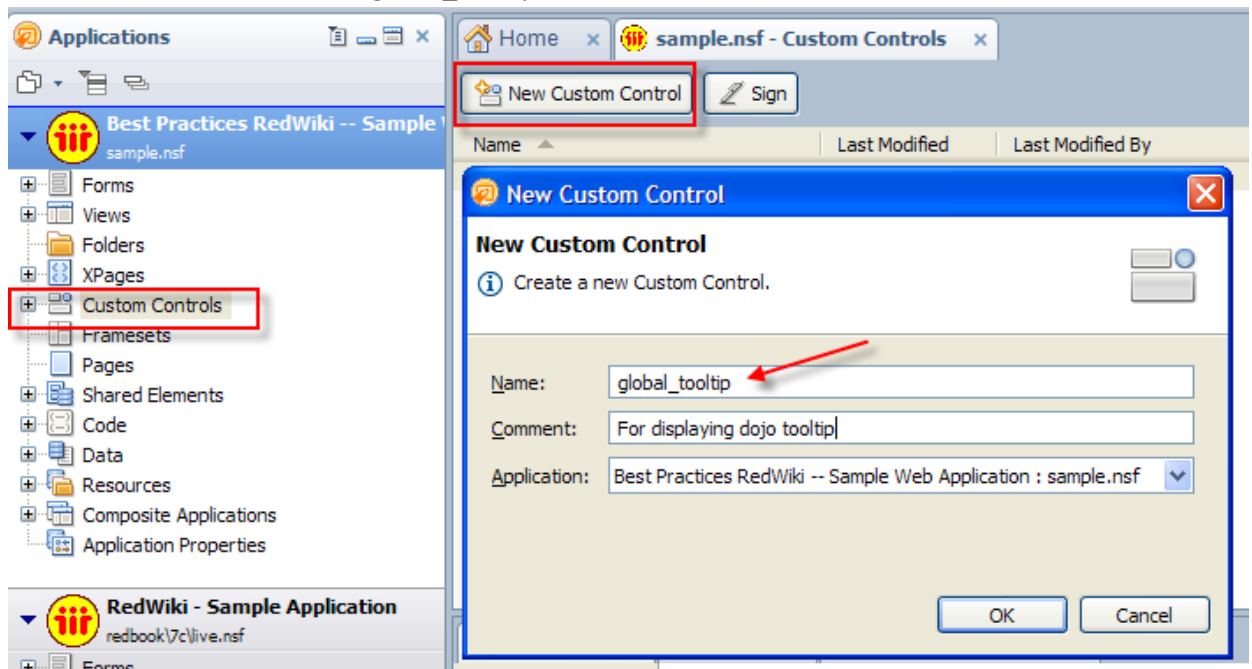
Below are the links for additional find out more information about Dojo:

http://www-10.lotus.com/ldd/ddwiki.nsf/dx/Client_Side_JavaScript_Libraries_in_XPages.htm

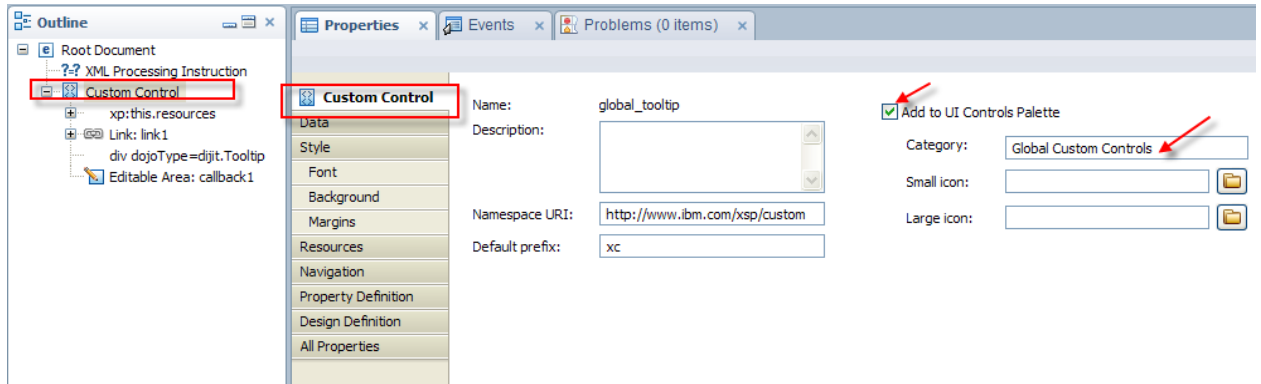
<http://www.dojotoolkit.org/>

We are going to use dojo tool tip and multiple places within the application, so it makes sense to implement this as a separate custom control which can be reused. This also serves as an example to incorporate dojo within XPages application.

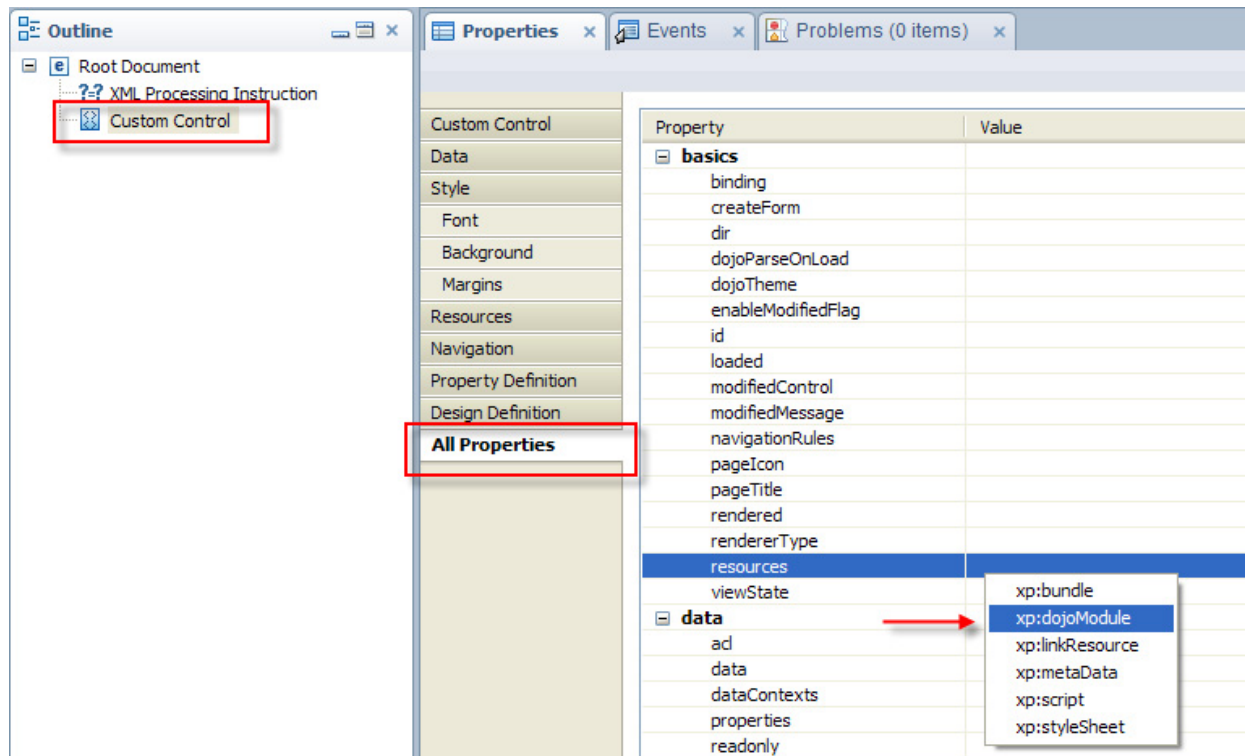
1. Create a new custom control “global_tooltip”



2. Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Global Custom Controls**” as the category. This moves this custom control under the category “**Global Custom Controls**”.



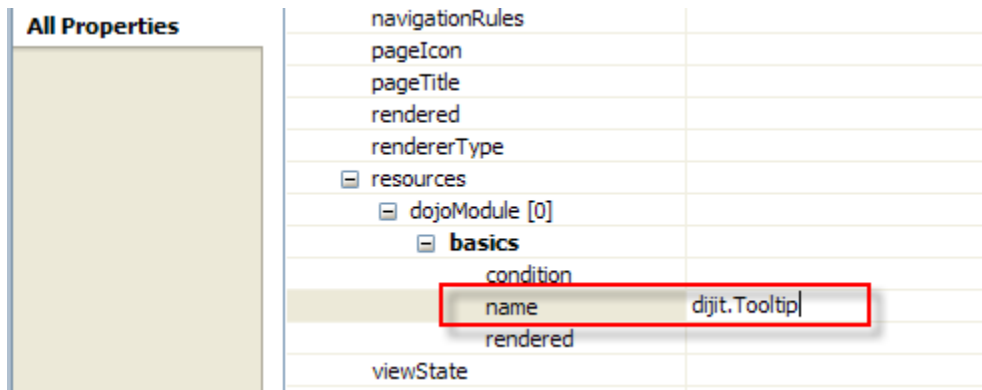
- Under “All Properties”, click on + sign in “resources” property and select “xp:dojoModule” from the drop down.



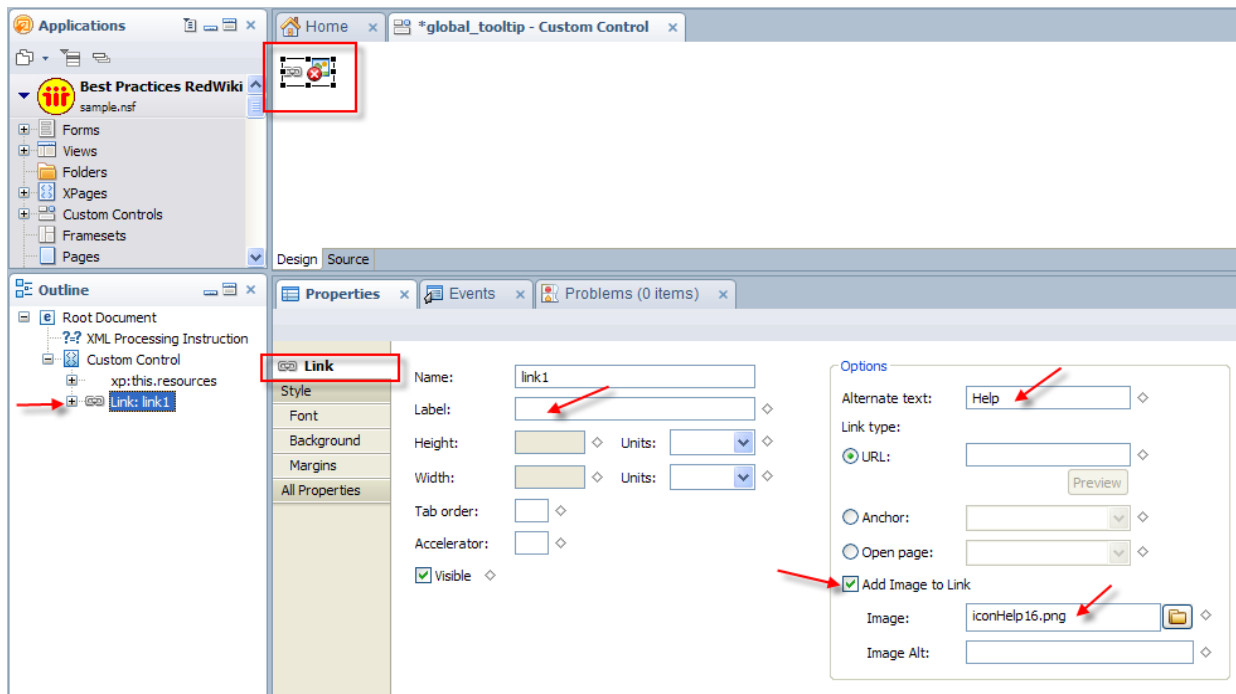
- Enter “dijit.Tooltip” as the **dojoModule** name. This will add the following code to the source:

```
<xp:this.resources>
  <xp:dojoModule name="dijit.Tooltip"></xp:dojoModule>
</xp:this.resources>
```

This code makes sure that the required dojo resources are referenced and loaded.



5. Drag and drop as **Link** control to the editor. Remove any text in the Label property to leave it blank as shown in the screenshot. Enter **“Help”** as **“Alternate Text”** property. Check **“Add Image to Link”** checkbox and enter **“iconhelp16.png”** as **“Image”** property.



6. Switch to the **Source** tab and enter the following code below the link code -- just above `</xp:view>` tag:

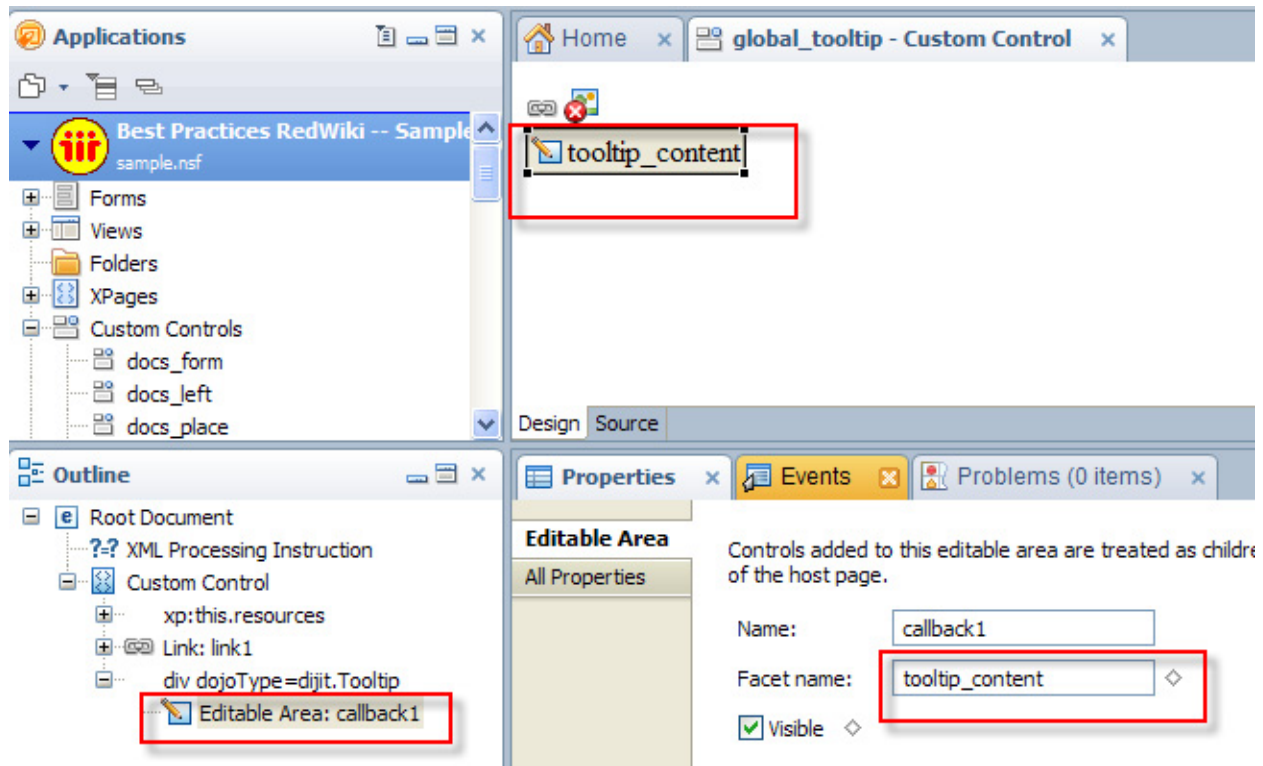
```
<div dojoType="dijit.Tooltip" connectId="#{id:link1}"></div>
```

This code creates a dojo tool tip and sets up its **connectId** attribute. ConnectId attribute indicates which HTML element at run time does activate the dojo tooltip. In this case we are associating the tooltip with the link element created earlier – so whenever someone clicks on a

link, a tooltip is activated (becomes visible). Since the actual id of the element can change at run time, we are using `#{id:elementId}` – which gets translated to the real element id at runtime. Refer to dojo documentation for additional details about dojo tooltip.



7. Drag an **Editable Area** control and drop it within the div element created above. If it does not get dropped to within the div element, just move it div control from the outline pallet. Enter **"tooltip_content"** as **Facet Name**. The editable area allows us to change the actual content of the tooltip when this custom control is added to an XPage.



8. Click on "Source" tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

  <xp:this.resources>
    <xp:dojoModule name="dijit.Tooltip"></xp:dojoModule>
  </xp:this.resources>

  <xp:link escape="true" id="link1" title="Help">
    <xp:image id="image1" url="iconHelp16.png"></xp:image>
  </xp:link>

  <div dojoType="dijit.Tooltip" connectId="#{id:link1}">
    <xp:callback facetName="tooltip_content"
id="callback1"></xp:callback>
  </div>

</xp:view>
```

Step 7 – Sample 2: Creating forms and views

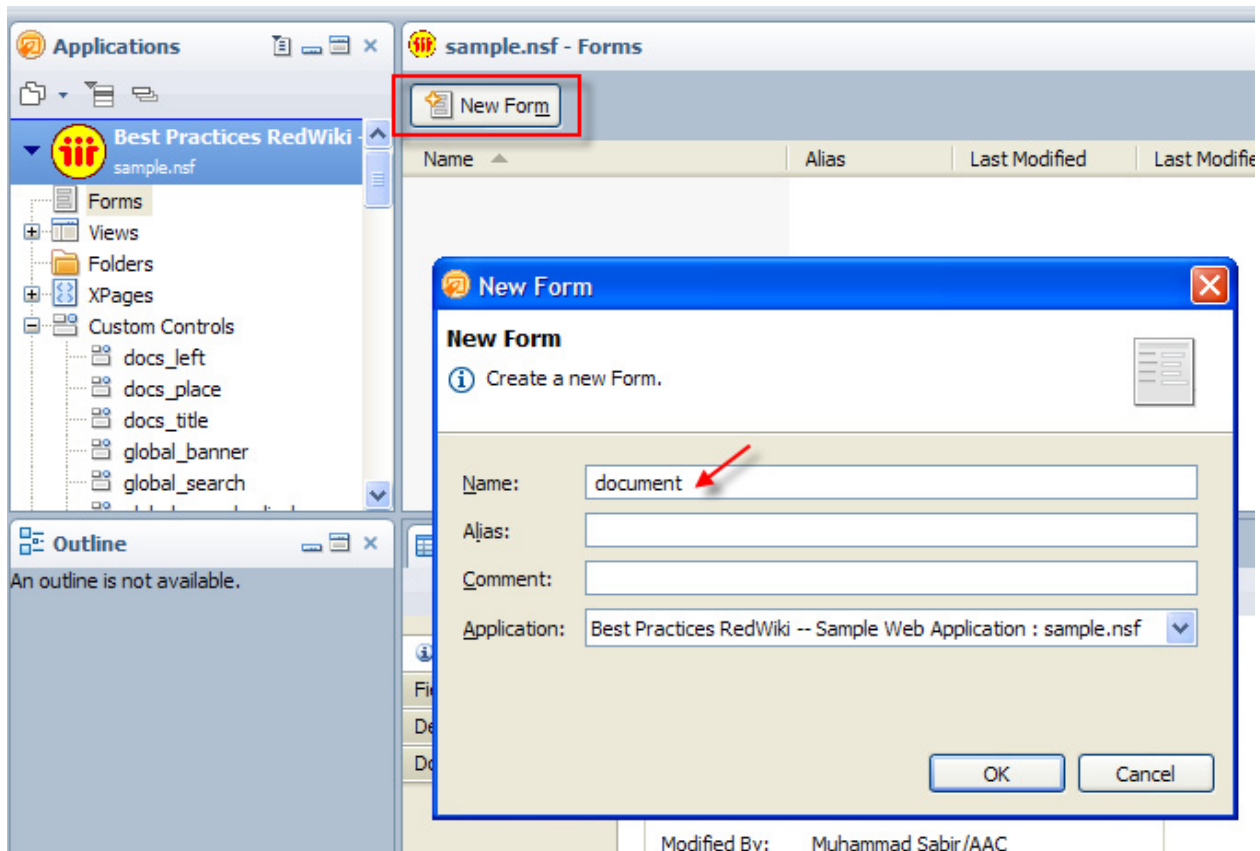
So far we have built themes, layout custom controls and global custom control. All of this work was to build a framework and layout with place holders for content. None of this was specific to any particular web application that we want to build. The work up to this point can be captured and saved so that you can apply it to various applications to provide the consistent look and feel across different applications.

From this point onward, most of the work we will do is specific to the sample application. In this section, we are going to create forms and views that will be used by XPages or custom controls in later sections.

Step 7.1: Creating a document form

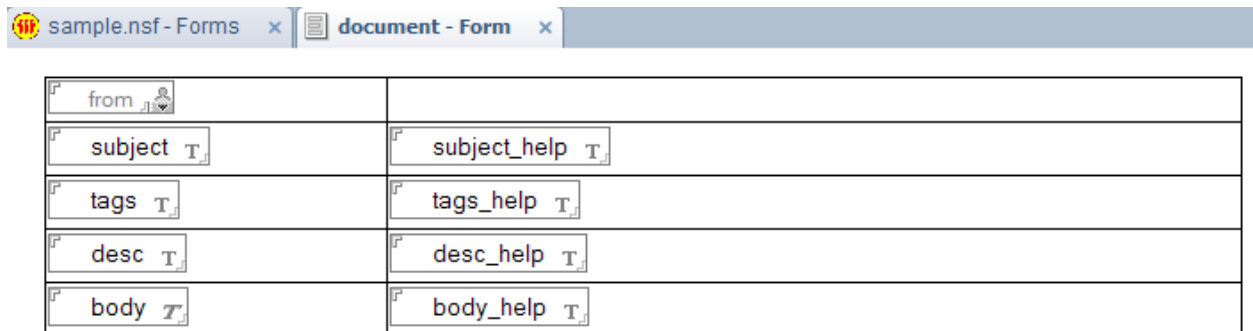
In XPages based application, the look and feel of a Domino form element is irrelevant. A form in an Xpage application is similar to a table in relational database application where it is used to define fields and associated data types.

1. Click on **“New Form”** button and enter **“document”** as the form name.



2. Create the fields based on the information in the following table. Since most developers are familiar with this process, we have not specified every single step here. All the fields ending with “_help” contain the tooltip text for a particular field. This will become clear in later parts of this exercise.

Field Name	Type	Value
From	Names, computed when composed	@Name([CN];@UserName)
Subject	Text, editable	
Subject_help	Text, computed	Use any characters except these: / : * ? \ " < >
Tags	Text, editable, allow multiple values	
Tag_help	Text, computed	Tags are keywords you can add to files to make the files easier to find later. Separate multiple tags with a space.
Desc	Text, editable	
Desc_help	Text, computed	Type a brief description about the attached document
Body	Rich Text	
Body_help	Text, computed	Files must be smaller than 10 MB



The screenshot shows a web browser window with two tabs: "sample.nsf - Forms" and "document - Form". The "document - Form" tab is active, displaying a form with the following fields:

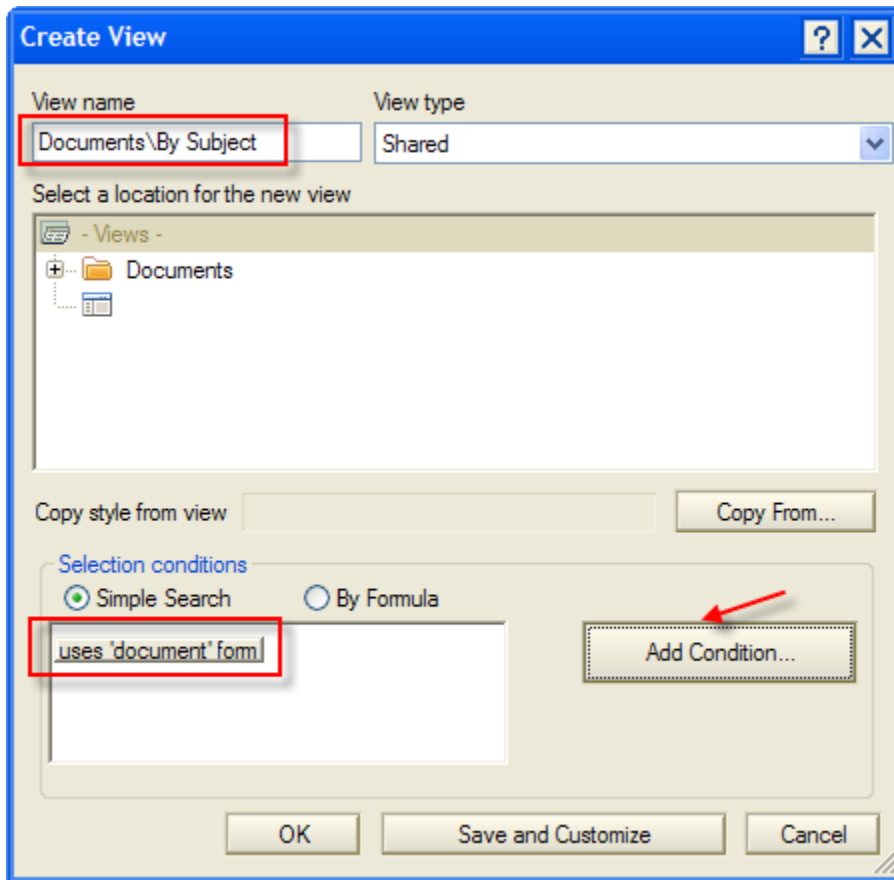
from	
subject	subject_help
tags	tags_help
desc	desc_help
body	body_help

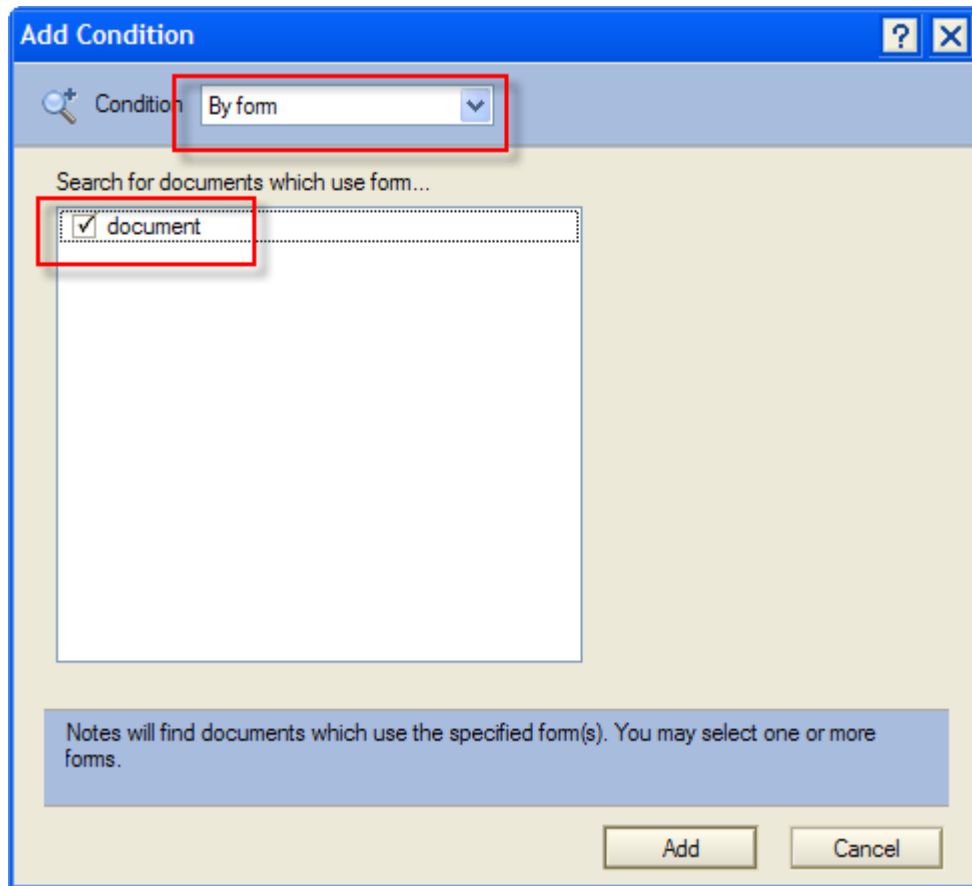
Step 7.2 : Creating views

In XPages based application, the look and feel of a Domino view element is irrelevant. View in an Xpage application is similar to a query in relational database application where it is used to query the data saved in the tables (documents in this case).

Step 7.2.1: Creating “Document\By Subject” view

1. Create a new shared view named as “**Documents\By Subject**”. Click “**Add Condition**” button and select “**By Form**” and check the “**document**” form. This will include only those documents created with “**document**” form.

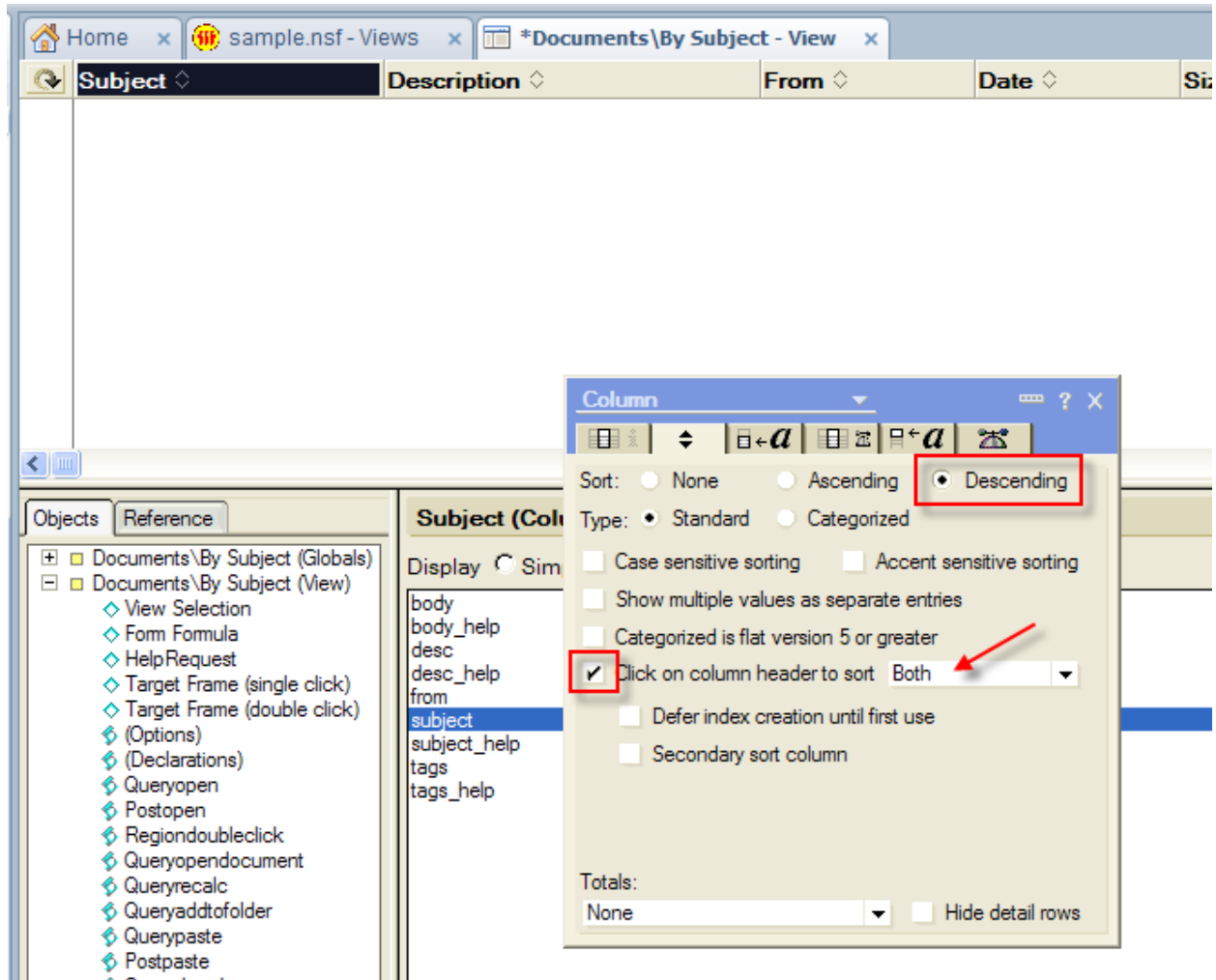




2. Create columns based on the information in the following table.

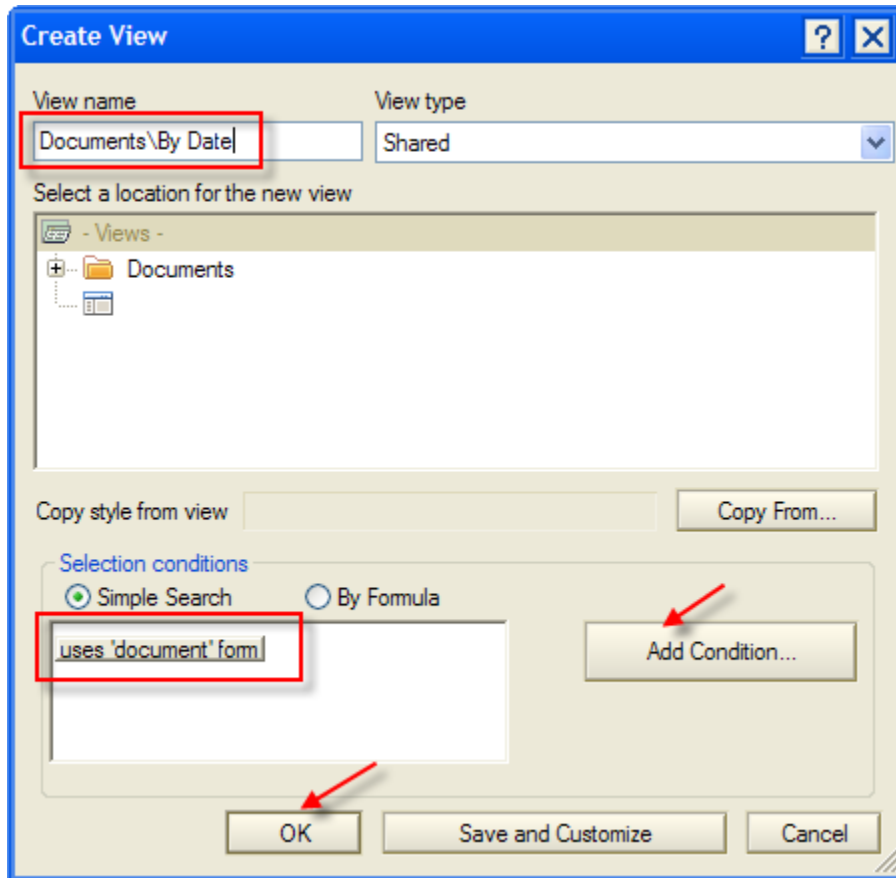
Column	Title	Type	Value
1	Subject	Field	Subject
2	Description	Field	Desc
3	From	Field	From
4	Date	Simple Action	Creation Date
5	Size	Simple Action	Attachment Lengths

3. Sort the first column in **ascending** order and select "Click **column header to sort both ways**". This allows column sorting when it is added to a custom control or xPage.



Step 7.2.2: Creating “Document\By Date” view

1. Create a new shared view named as “**Documents\By Date**”. Click “**Add Condition**” button and select “**By Form**” and check the “**document**” form. This will include only those documents created with “**document**” form.

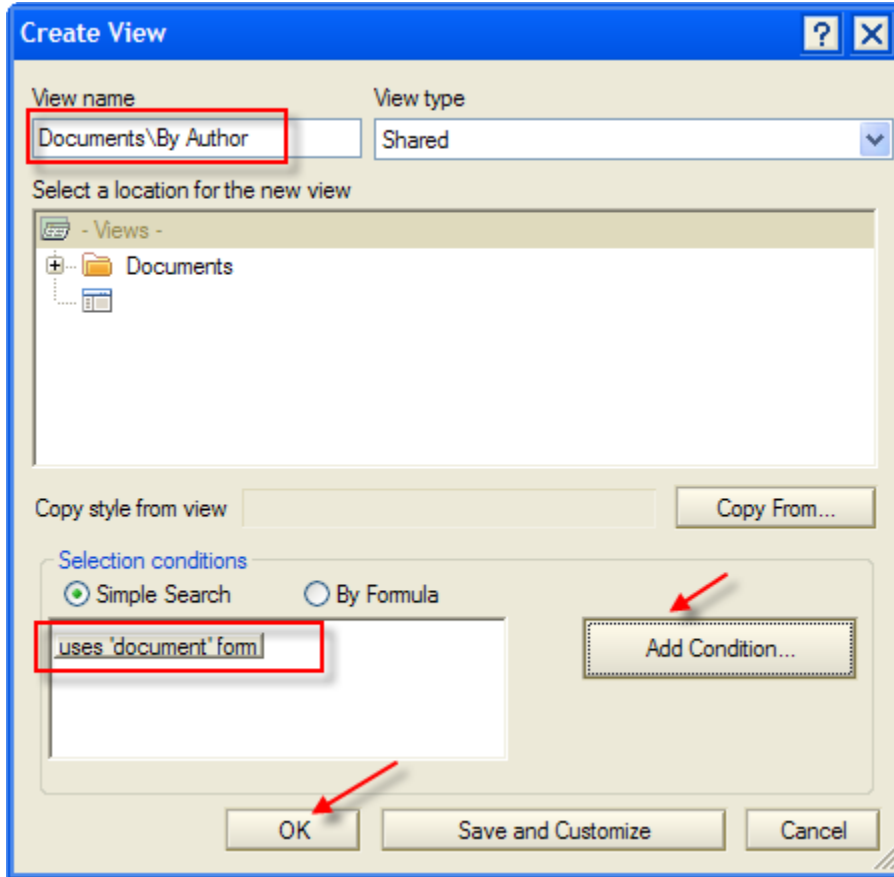


2. Create columns based on the information in the following table.

Column	Title	Type	Value
1	Date	Formula (categorized and sorted in descending order)	@Date(@Created)
2	Subject	Field	Subject
3	Description	Field	Desc
4	From	Field	From
5	Size	Simple Action	Attachment Lengths

Step 7.2.3: Creating “Document\By Author” view

1. Create a new shared view named as “**Documents\By Author**”. Click “**Add Condition**” button and select “**By Form**” and check the “**document**” form. This will include only those documents created with “**document**” form.

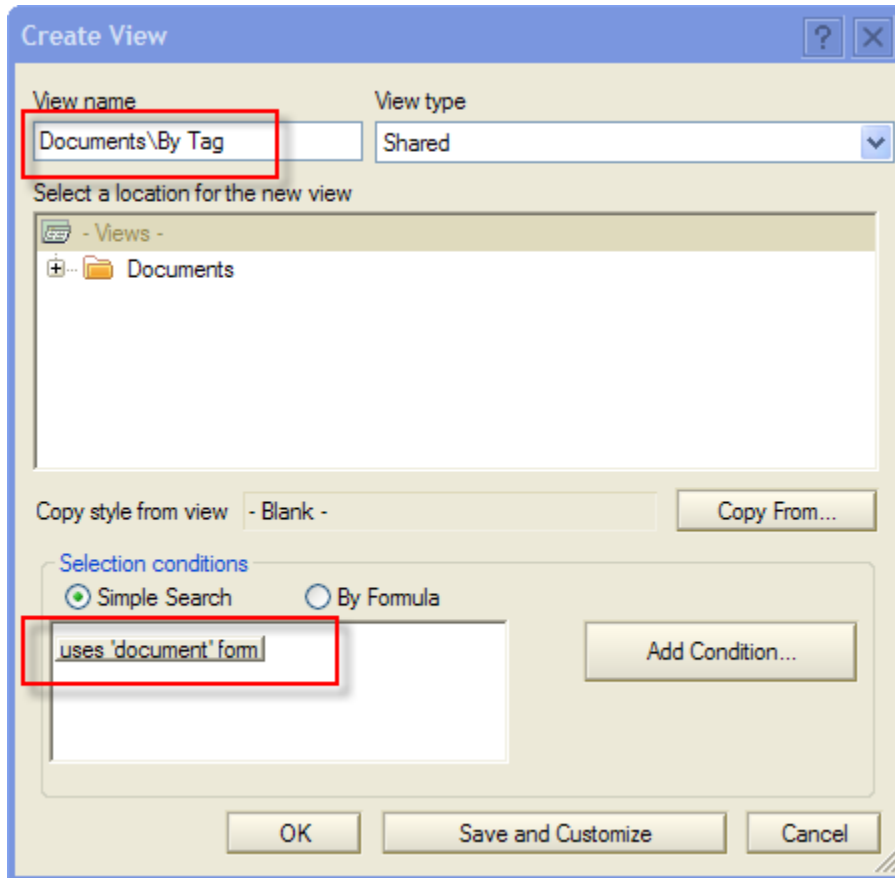


2. Create columns based on the information in the following table.

Column	Title	Type	Value
1	From	Field (Categorized and sorted)	From
2	Date	Simple Action	Creation Date
3	Subject	Field	Subject
4	Description	Field	Desc
5	Size	Simple Action	Attachment Lengths

Step 7.2.4: Creating “Document\By Tag” view

1. Create a new shared view named as “**Documents\By Tag**”. Click “**Add Condition**” button and select “**By Form**” and check the “**document**” form. This will include only those documents created with “**document**” form.

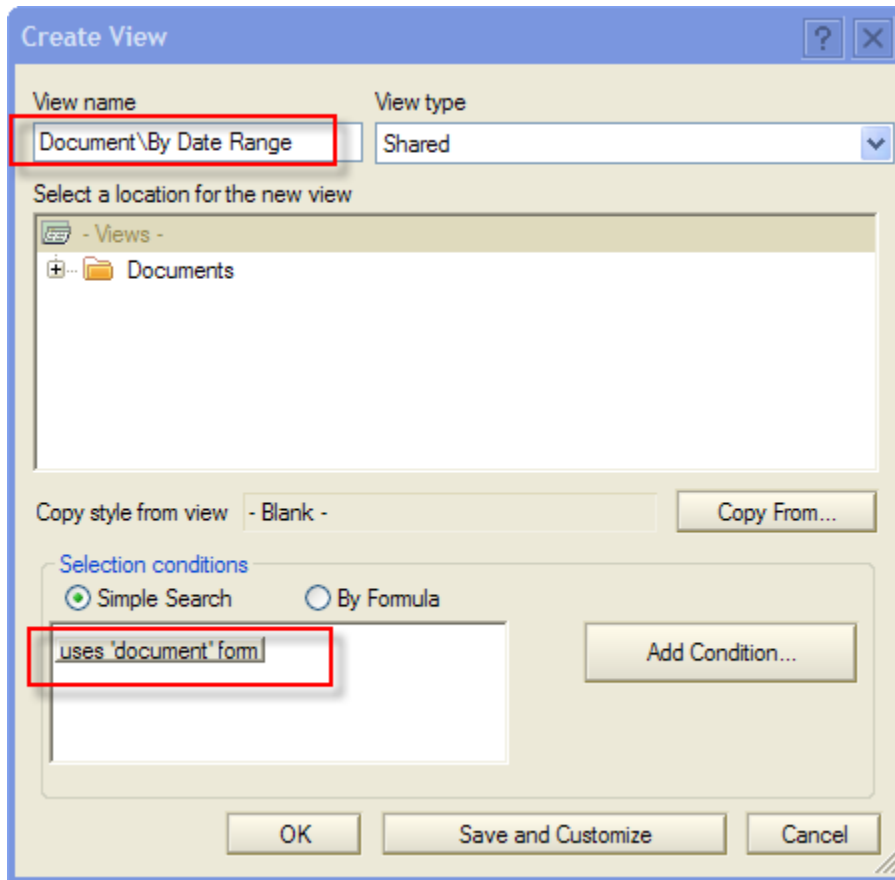


2. Create columns based on the information in the following table.

Column	Title	Type	Value
1	Tag	Field(categorized and sorted in descending order)	Tag
2	Subject	Field	Subject
3	Description	Field	Desc
4	From	Field	From
5	Size	Simple Action	Attachment Lengths

Step 7.2.5 : Creating “Document\By Date Range” view

1. Create a new shared view named as “**Documents\By Author**”. Click “**Add Condition**” button and select “**By Form**” and check the “**document**” form. This will include only those documents created with “**document**” form



2. Create columns based on the information in the following table.

Column	Title	Type	Value
1		Formula (Categorized and sorted in descending order)	<pre> week:=@If(@Adjust(@Created; 0;0;7;0;0;0) > @Today;"week";"""); today:=@If(@Date(@Created)=@Today;"today";"""); month:=@If(@Month(@Created)=@Month(@Today);"mont h";"""); year:=@If(@Year(@Created)=@Year(@Today);"year";"""); list:="all+"+today+"+"+week+"+"+month+"+"+year; @Explode(list;"+") </pre>
2	Date	Simple Action	Creation Date
3	Subject	Field	Subject
4	Description	Field	Desc
5	From	Field	From
6	Size	Simple Action	Attachment Lengths

Please note that the first column has formula to categorize documents based on the date they were created. This formula works but it is not an efficient one for production use. Refer to this document for better option for using date based formula within Domino views:

<http://www-01.ibm.com/support/docview.wss?uid=swg27003557>

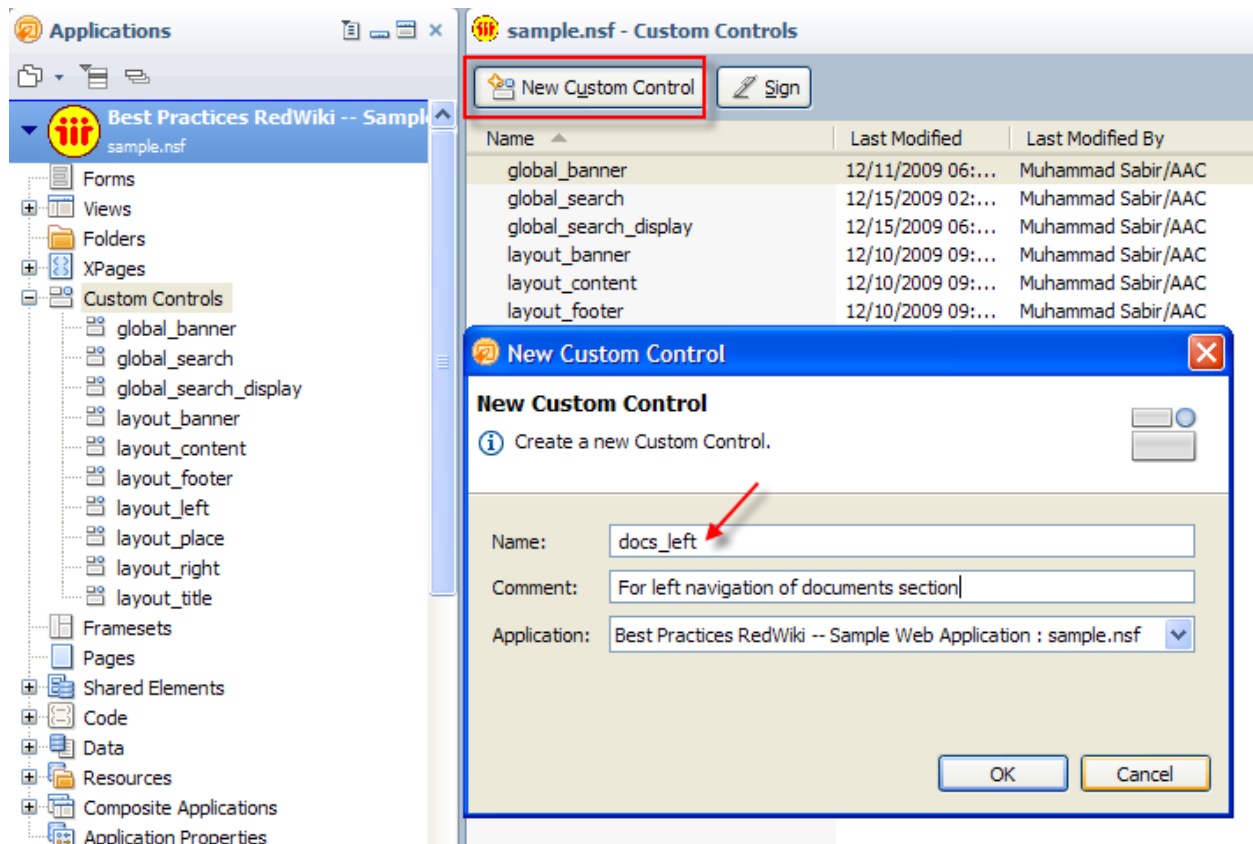
Step 8 – Sample 2: Developing custom controls for specific content

We have already created custom controls for the layout and global content. In this section we will develop custom controls for specific sections of the application. For this sample application, we will build a documents section where users can upload, search, download and view a list of documents. Functionality like this is fairly common within corporate Intranets; therefore we are using an Intranet context for global navigation – even though we are only developing “documents” section of the intranet.

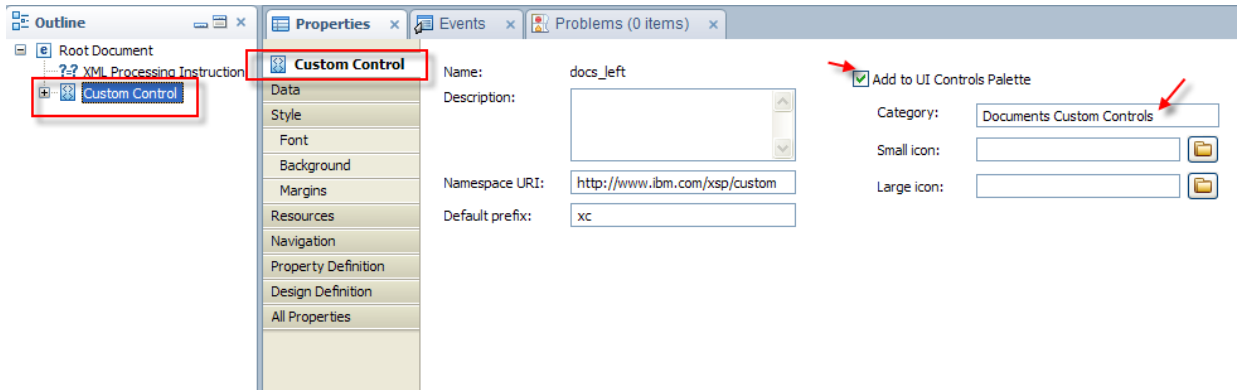
Step 8.2 : Creating content control for left navigation

In this section we will build a custom control for left navigation of documents section of the sample application.

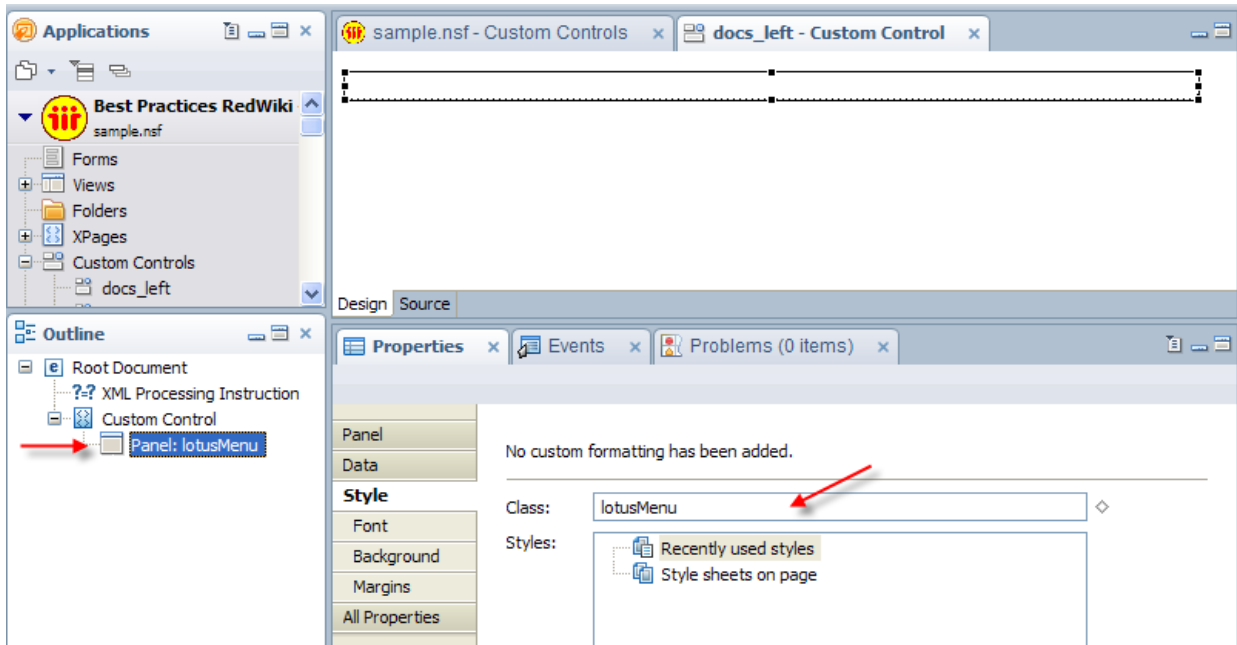
1. Create new custom control and name it “docs_left”.



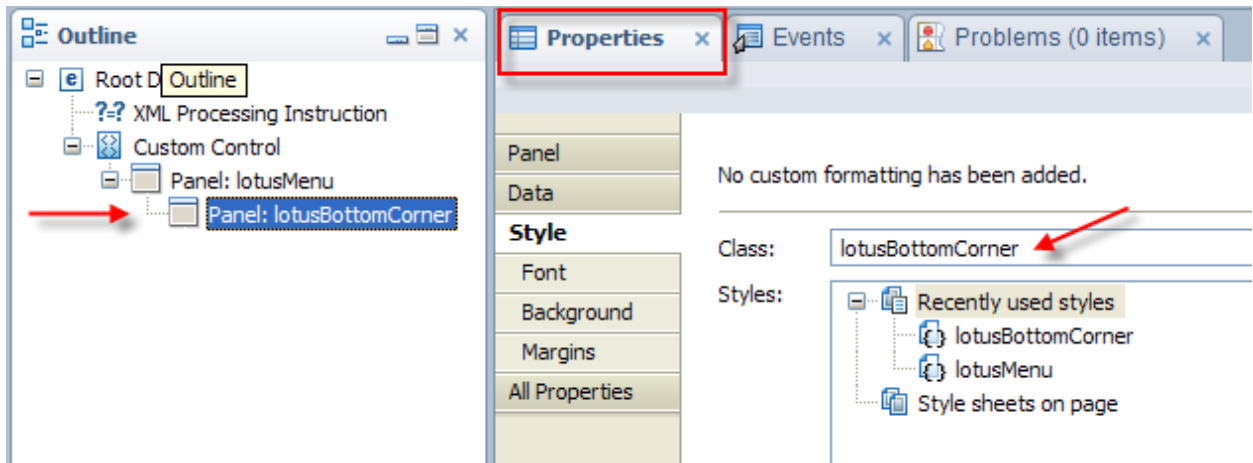
- Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Documents Custom Controls”** as the category. This moves this custom control under the category **“Documents Custom Controls”**.



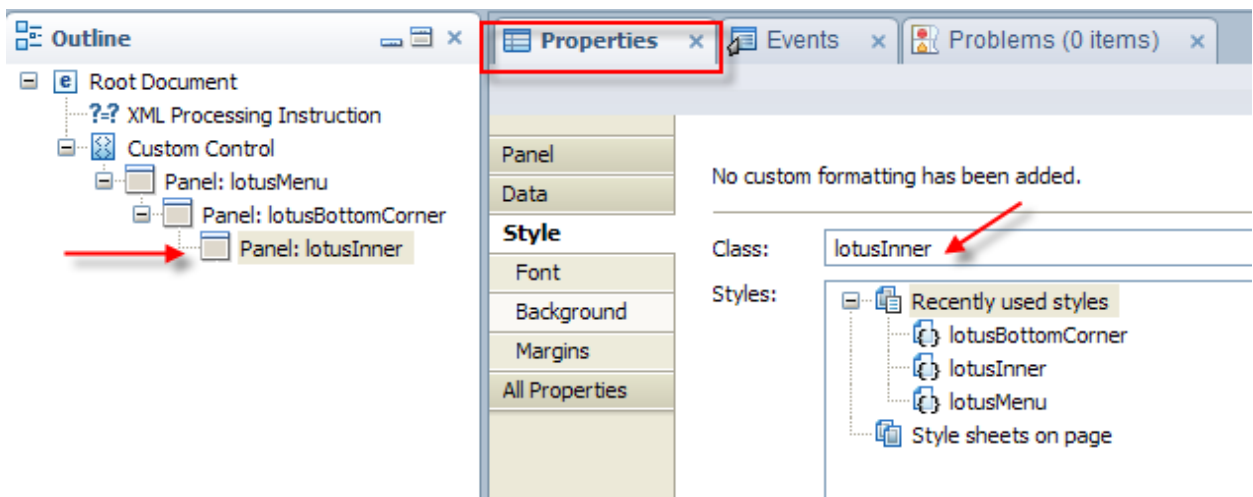
- Drag and drop a **Panel** control. Enter **“lotusMenu”** both as its name and CSS style class.



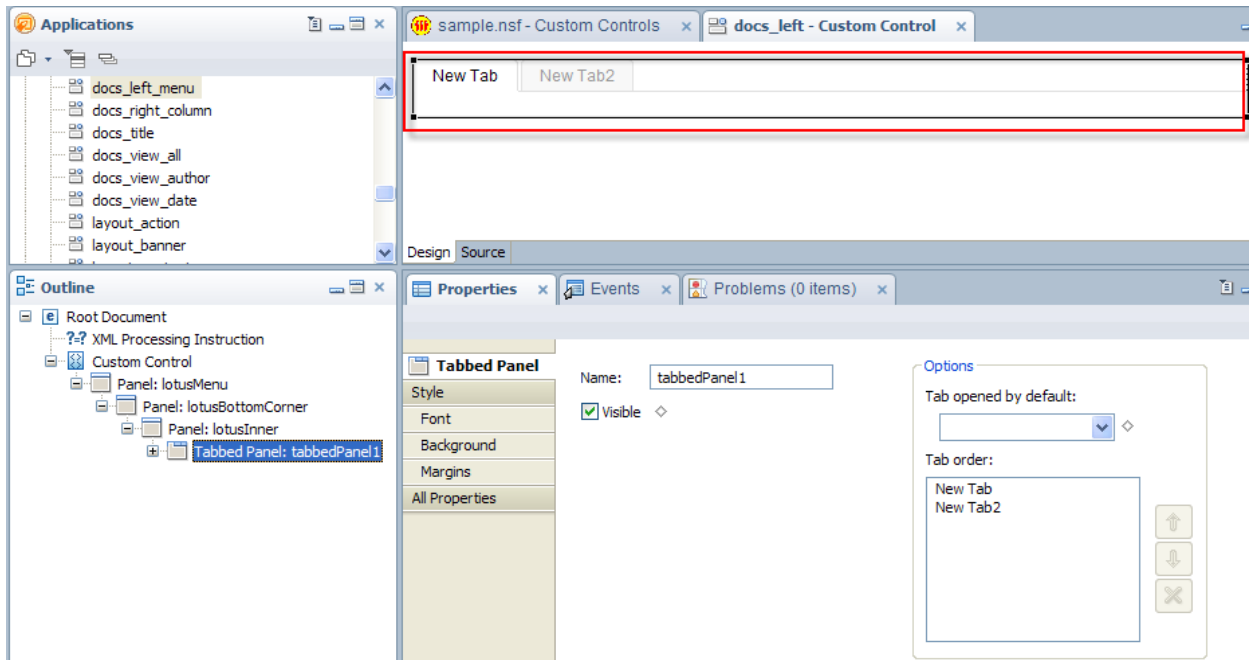
- Drag a **Panel** control and drop it within the **“lotusMenu”** panel created in last step. Enter **“lotusBottomCorner”** both as its name and CSS style class.



5. Drag a **Panel** control and drop it within the “**lotusBottomCorner**” panel created in last step. Enter “**lotusInner**” both as its name and CSS style class.

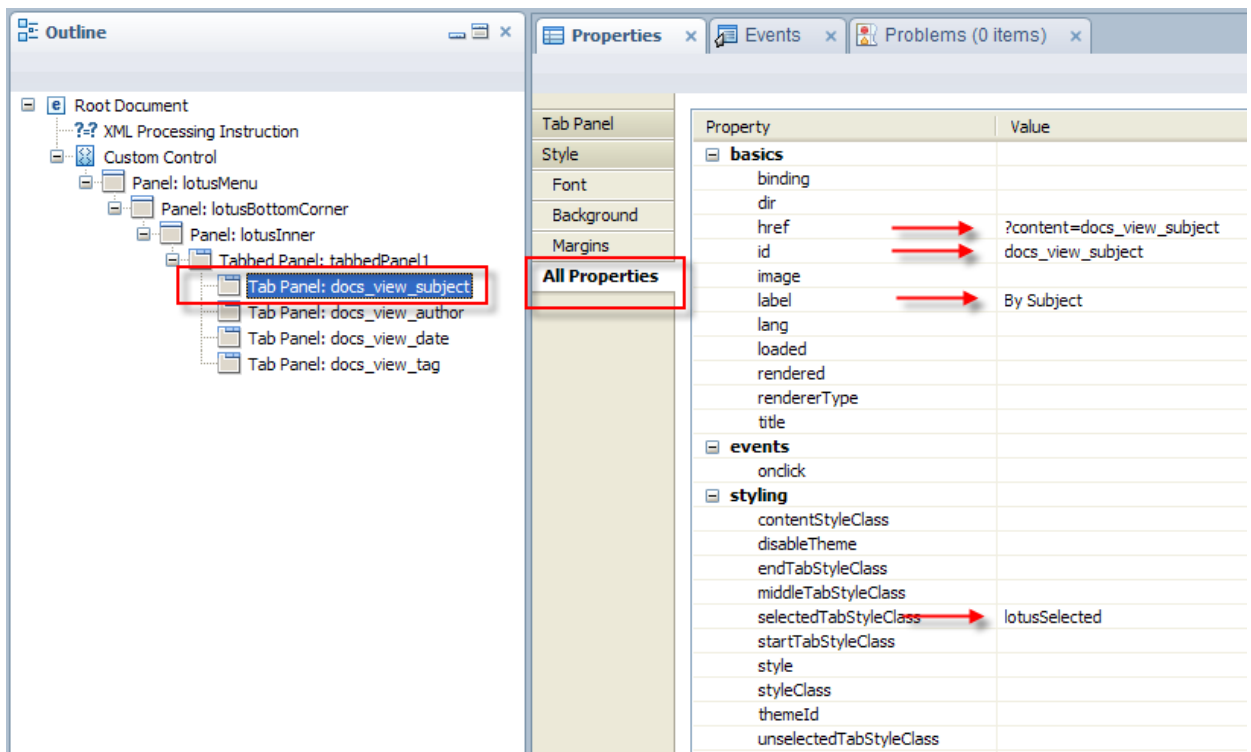
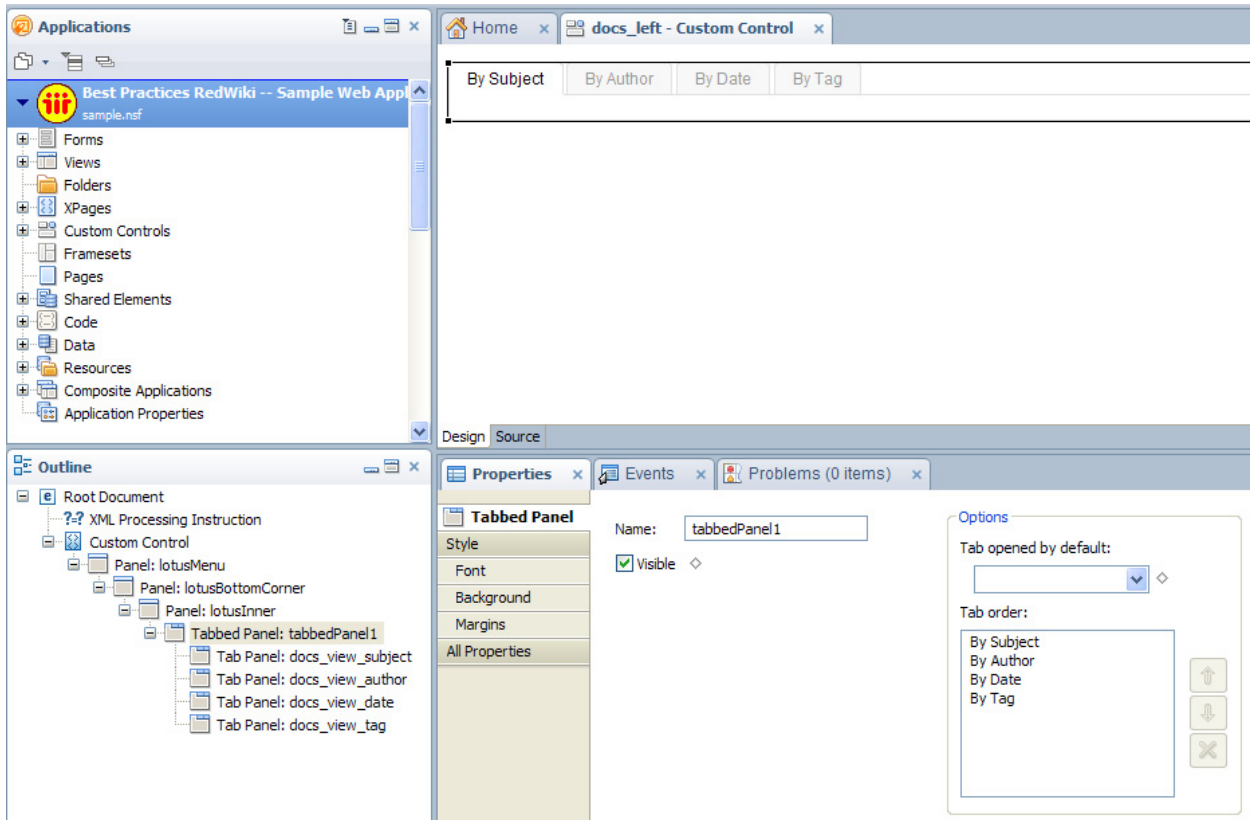


6. Drag a **Tabbed Panel** control and drop it within the “**lotusInner**” panel created in last step.



7. Add two additional tabs by right clicking on **Tabbed Panel** and selecting "Append Row". For each tab panel, click on "All Properties" and set the following properties:

id	Label	href	selectedTabStyleClass
docs_view_subject	By Subject	?content= docs_view_subject	lotusSelected
docs_view_author	By Author	?content= docs_view_author	lotusSelected
docs_view_date	By Date	?content= docs_view_date	lotusSelected
docs_view_tag	By Tag	?content= docs_view_tag	lotusSelected

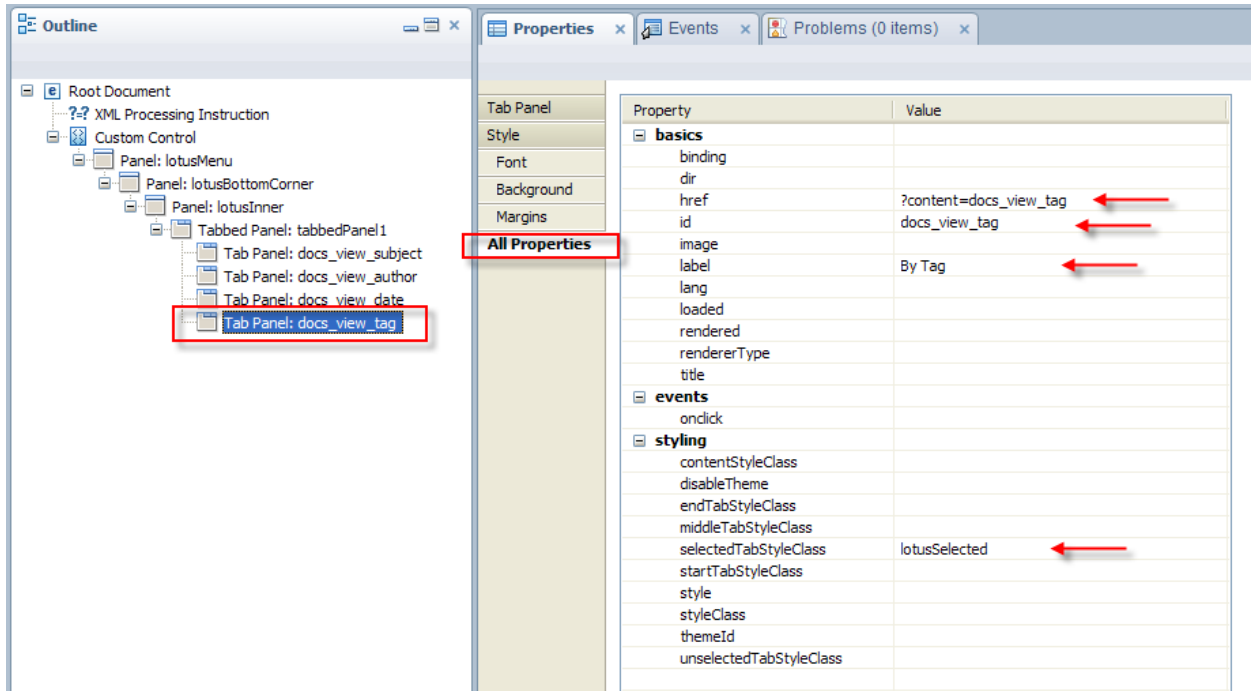


The screenshot shows the Domino Designer interface with the Outline and Properties windows. The Outline window displays a tree structure of the document, with 'Tab Panel: docs_view_author' selected. The Properties window shows the 'All Properties' tab, listing various properties for the selected tab panel. Red arrows highlight specific values in the Properties window:

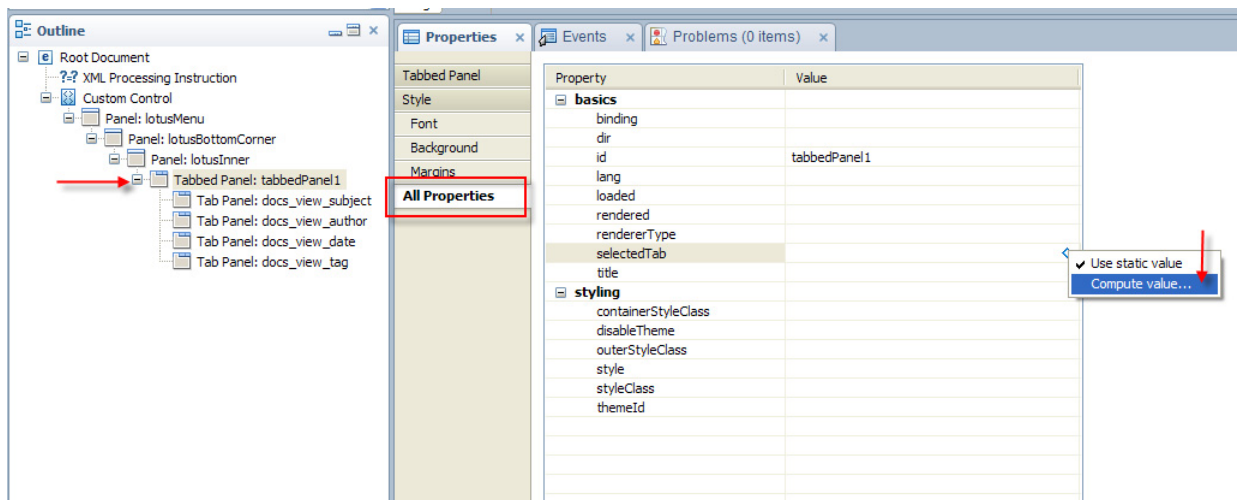
- dir**: ?content=docs_view_author
- id**: docs_view_author
- label**: By Author
- selectedTabStyleClass**: lotusSelected

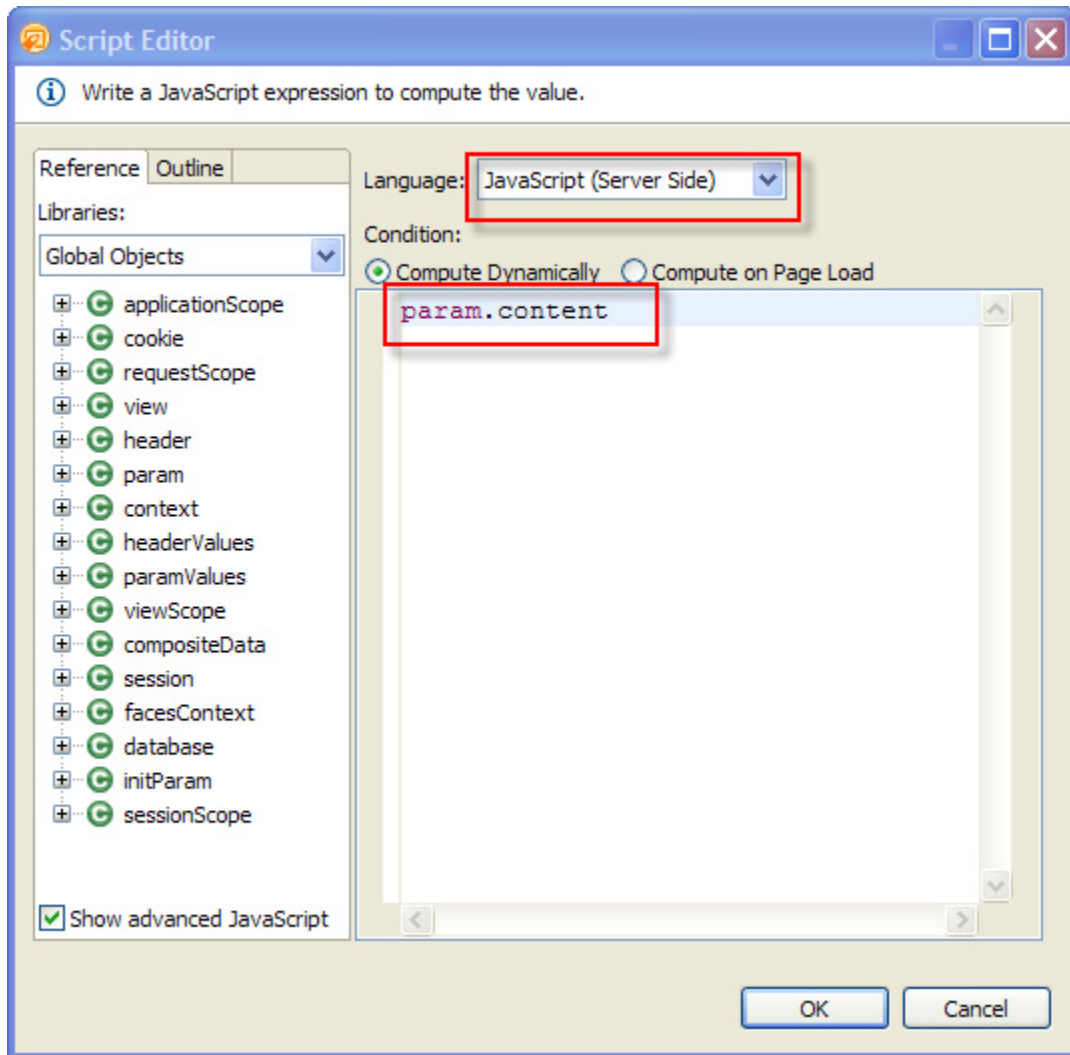
The screenshot shows the Domino Designer interface with the Outline and Properties windows. The Outline window displays a tree structure of the document, with 'Tab Panel: docs_view_date' selected. The Properties window shows the 'All Properties' tab, listing various properties for the selected tab panel. Red arrows highlight specific values in the Properties window:

- dir**: ?content=docs_view_date
- id**: docs_view_date
- label**: By Date
- selectedTabStyleClass**: lotusSelected



8. Select tabbed panel from the outline palette and select “**computed value..**” for **selectedTab** property. Enter “**param.content**” in pop-up JavaScript editor. This will dynamically assign the selected tab, based on the value of “content” query string parameter.





9. Click on **"Source"** tab in editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
  <xp:panel id="lotusMenu" styleClass="lotusMenu">
    <xp:panel styleClass="lotusBottomCorner" id="lotusBottomCorner">
      <xp:panel styleClass="lotusInner" id="lotusInner">
        <xp:tabbedPanel id="tabbedPanel1">

          <xp:this.selectedTab><![CDATA[#{javascript:(param.content)}]]></xp:this.s
          electedTab>

          <xp:tabPanel label="By Subject"
            id="docs_view_subject"
            href="?content=docs_view_subject"
            selectedTabStyleClass="lotusSelected">
          </xp:tabPanel>
          <xp:tabPanel label="By Author"
            id="docs_view_author"
            href="?content=docs_view_author"
            selectedTabStyleClass="lotusSelected">
```

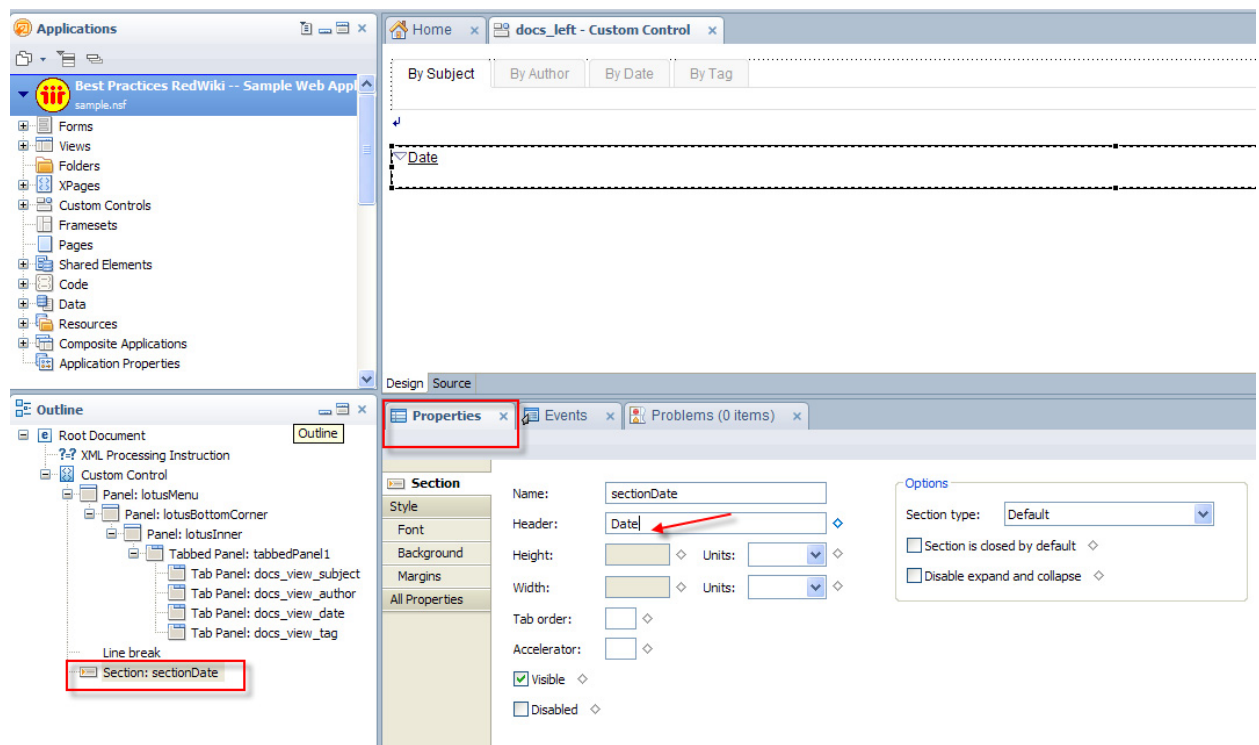
```

        </xp:tabPanel>
        <xp:tabPanel label="By Date"
                    href="?content=docs_view_date"
                    selectedTabStyleClass="lotusSelected">
        </xp:tabPanel>
        <xp:tabPanel label="By Tag" id="docs_view_tag"
                    href="?content=docs_view_tag"
                    selectedTabStyleClass="lotusSelected">
        </xp:tabPanel>
        </xp:tabbedPanel>
    </xp:panel>
</xp:panel>
</xp:panel>
</xp:view>

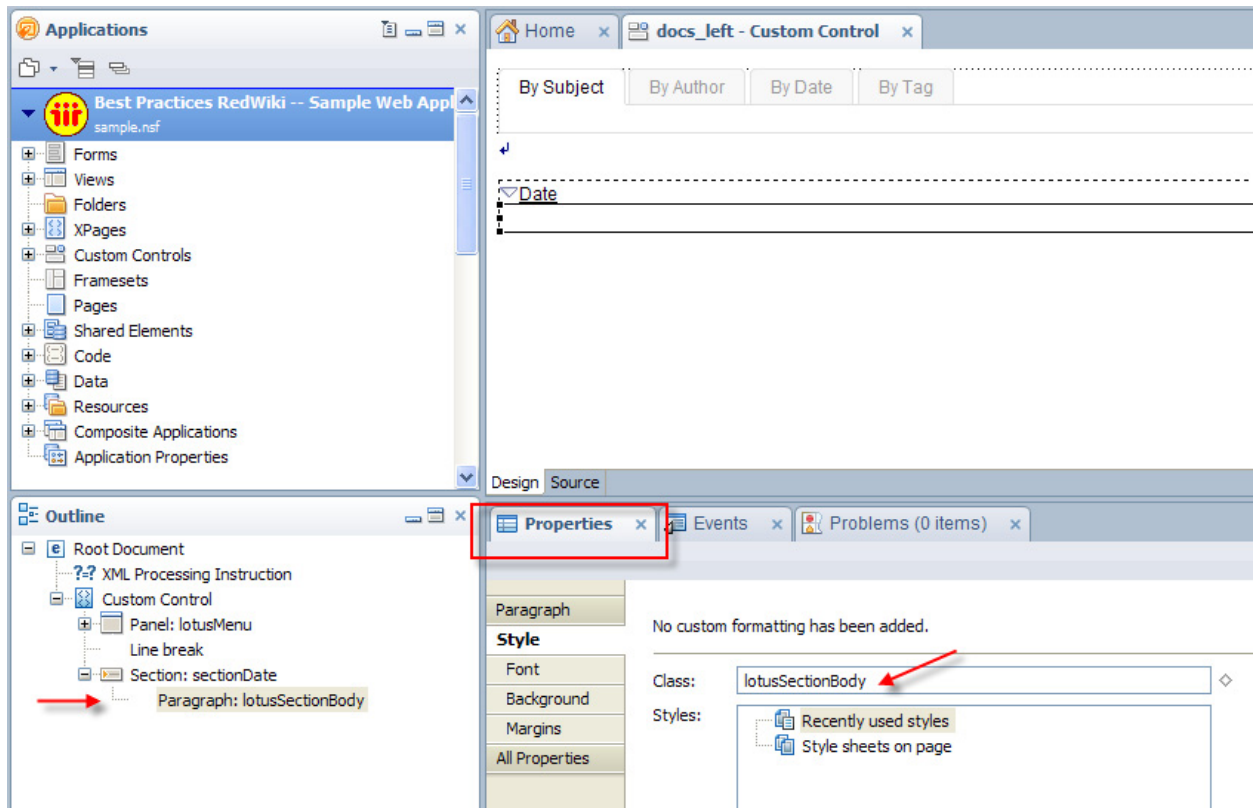
```

10. Drag a **Section** controls from the **Container** controls are drop it below the tabbed panel. You can enter an extras line break to separate it. Enter the following property values:

- Name=sectionDate
- Header=Date

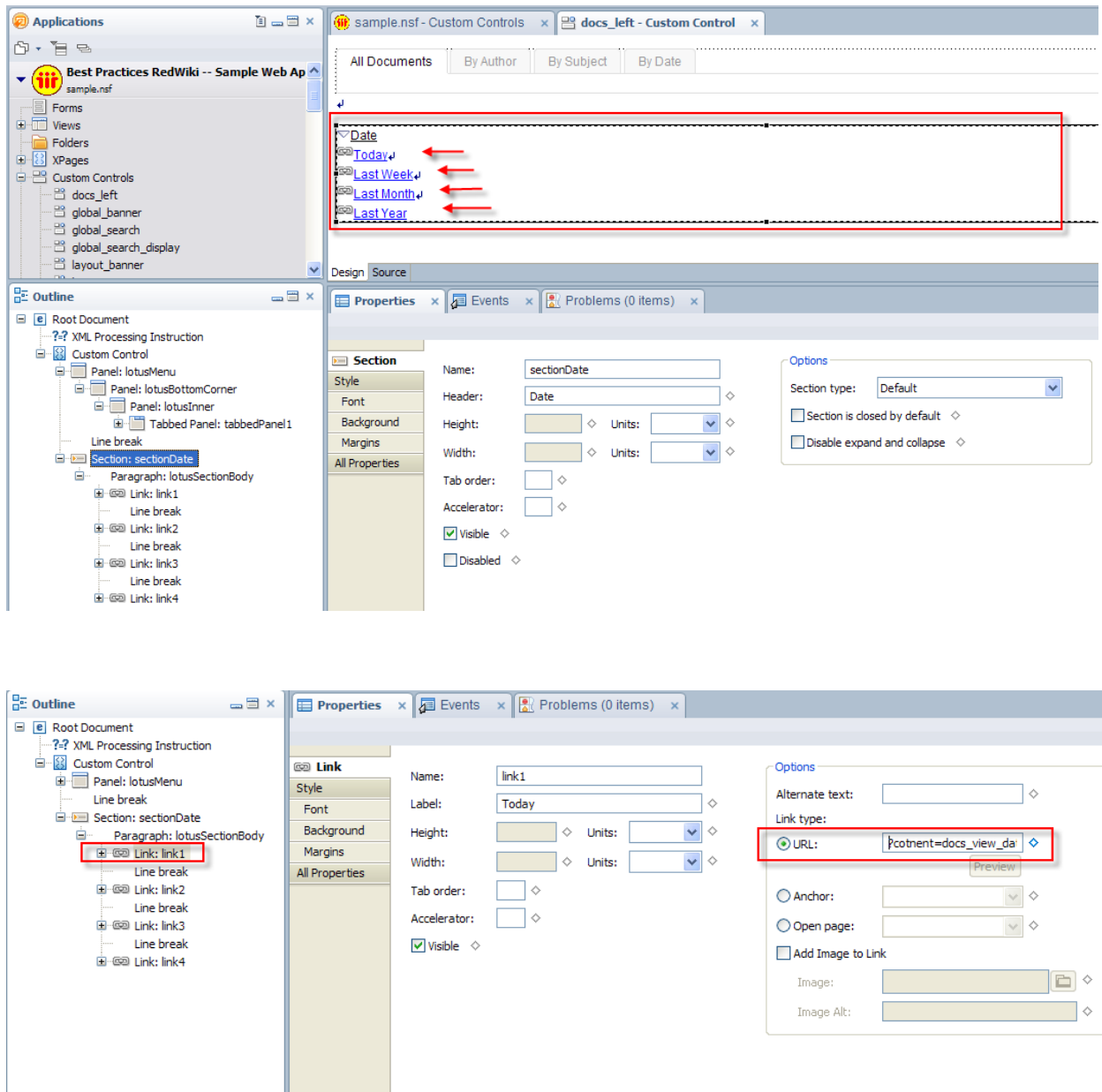


11. Drag a **Paragraph** control from “User Added Controls” and drop it within the section created in last step. Enter “**lotusSectionBody**” both as its name and CSS style class.



12. Drag and drop four **Link** controls within the “lotusSectionBody” paragraph control added in the last step. Enter a line break after each link so that each link displays in a new line. Enter the following properties for these four links:

Link Label	URL
Today	?cotnent=docs_view_date_byCat&viewCat=today
Last Week	?cotnent=docs_view_date_byCat&viewCat=week
Last Month	?cotnent=docs_view_date_byCat&viewCat=month
Last Year	?cotnent=docs_view_date_byCat&viewCat=year



13. Click on "Source" tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

  <xp:panel id="lotusMenu" styleClass="lotusMenu">
    <xp:panel styleClass="lotusBottomCorner"
      id="lotusBottomCorner">
```



```

<xp:panel styleClass="lotusInner" id="lotusInner">
  <xp:tabbedPanel id="tabbedPanel1"
    selectedTab="#{javascript:param.content}">

    <xp:tabPanel label="By Subject"
      id="docs_view_subject"
href="?content=docs_view_subject"
      selectedTabStyleClass="lotusSelected">
    </xp:tabPanel>
    <xp:tabPanel label="By Author"
id="docs_view_author"
      href="?content=docs_view_author"
      selectedTabStyleClass="lotusSelected">
    </xp:tabPanel>
    <xp:tabPanel label="By Date"
id="docs_view_date"
      href="?content=docs_view_date"
      selectedTabStyleClass="lotusSelected">
    </xp:tabPanel>
    <xp:tabPanel label="By Tag" id="docs_view_tag"
      href="?content=docs_view_tag"
      selectedTabStyleClass="lotusSelected">
    </xp:tabPanel>
  </xp:tabbedPanel>
</xp:panel>
</xp:panel>
</xp:panel>
<xp:br></xp:br>

<xp:section id="sectionDate" header="Date">
  <xp:div id="lotusSectionBody" styleClass="lotusSectionBody">
    <xp:link escape="true" text="Today" id="link1">

      <xp:this.value><![CDATA[?content=docs_view_date_byCat&viewCat=today]]><
/ xp:this.value>
    </xp:link>
    <xp:br />
    <xp:link escape="true" text="Last Week" id="link2">

      <xp:this.value><![CDATA[?content=docs_view_date_byCat&viewCat=week]]></
xp:this.value>
    </xp:link>
    <xp:br />
    <xp:link escape="true" text="Last Month" id="link3">

      <xp:this.value><![CDATA[?content=docs_view_date_byCat&viewCat=month]]><
/ xp:this.value>
    </xp:link>
    <xp:br />
    <xp:link escape="true" text="Last Year" id="link4">

      <xp:this.value><![CDATA[?content=docs_view_date_byCat&viewCat=year]]></
xp:this.value>
    </xp:link>
  </xp:div>
</xp:section>

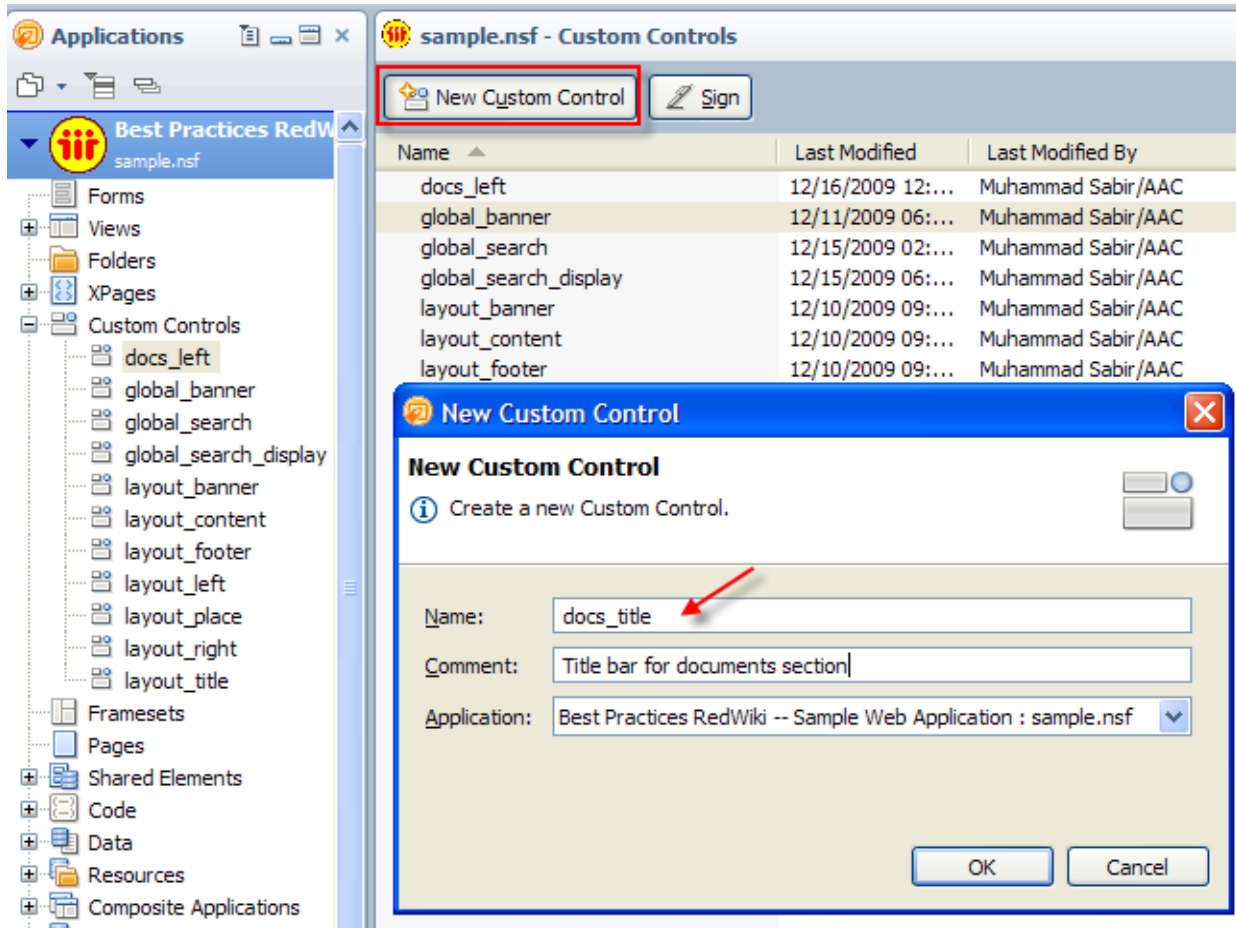
```

```
</xp:view>
```

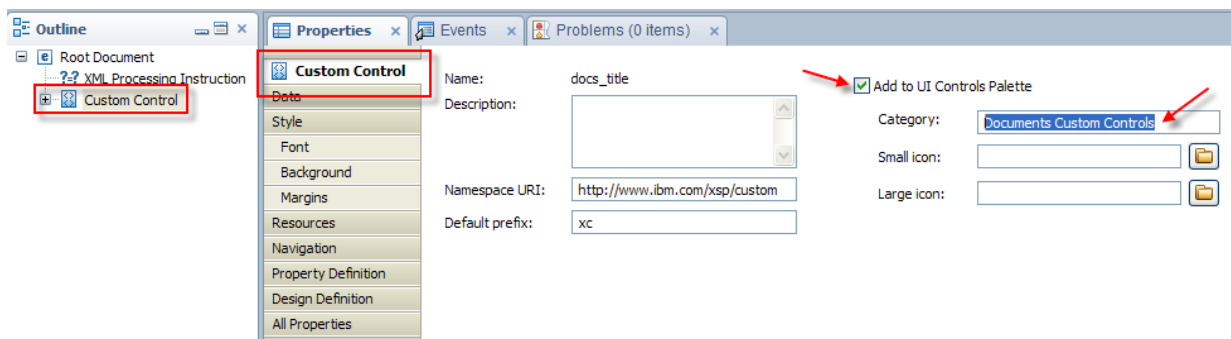
Step 8.3: Creating a custom control for title bar of document section

In this section we will build a custom control for the title bar content for documents section.

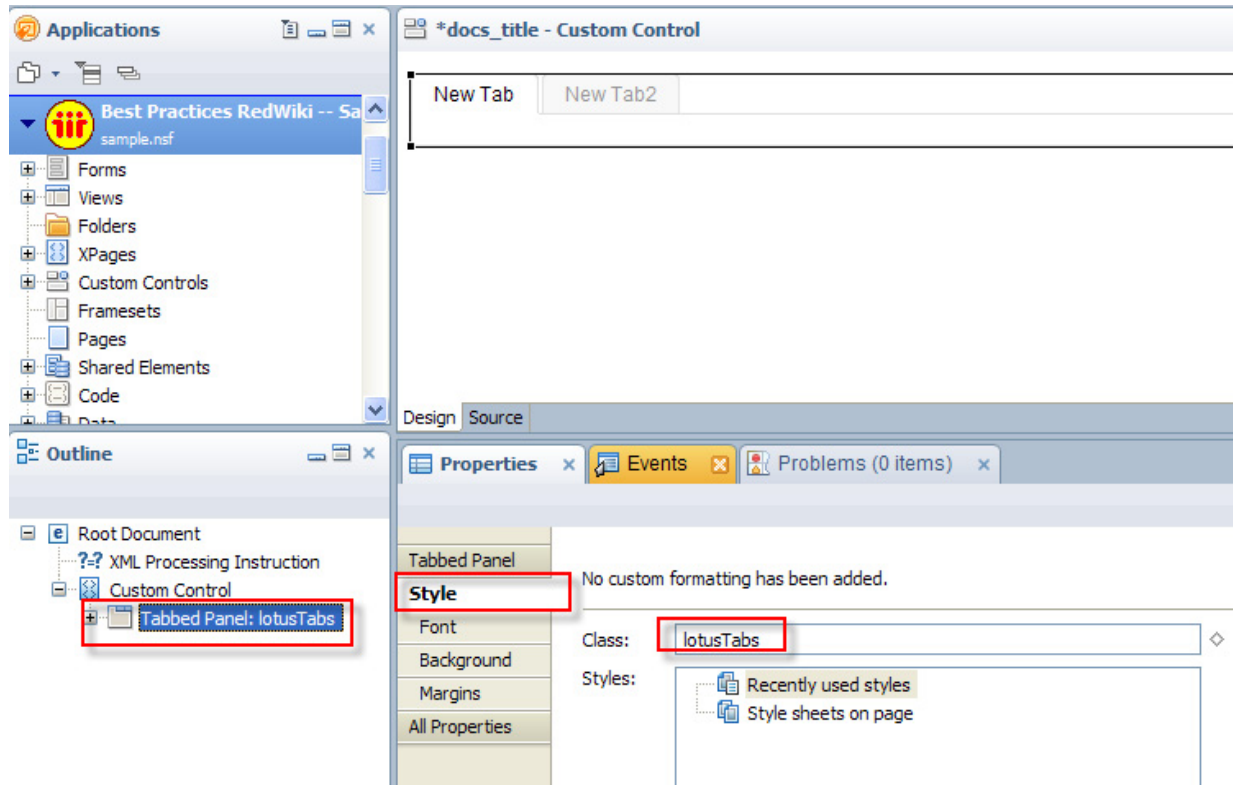
1. Create new custom control and name it “**docs_title**”.



2. Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Documents Custom Controls**” as the category. This moves this custom control under the category “**Documents Custom Controls**”.



3. Drag and drop a **Tabbed Panel** control. Enter “lotusTab” both as its name and CSS style class.



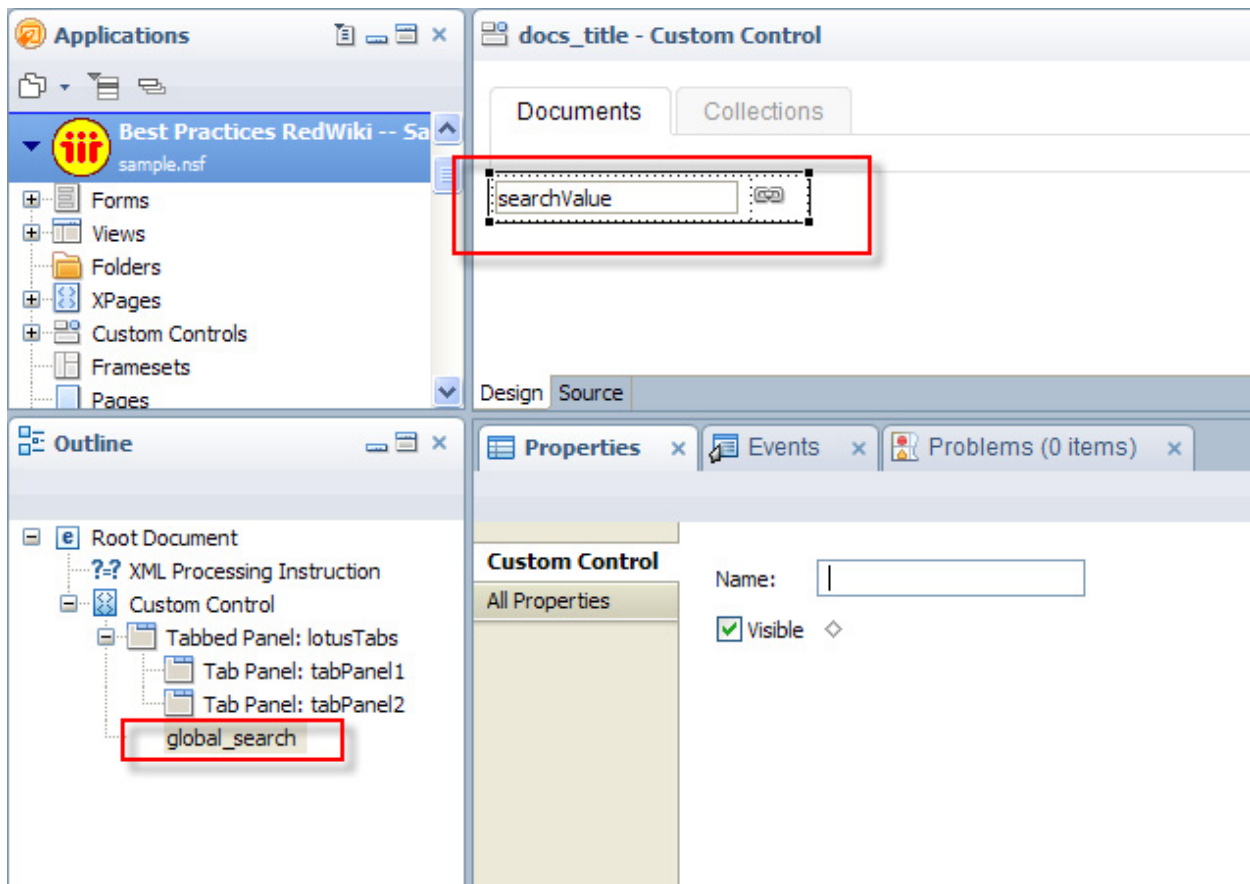
4. Label the first tab as “**Documents**” and second one as “**Collections**”. Enter “**lotusSelected**” as “**selectedTabStyleClass**” for each tab. We are not adding the collections feature in the sample application. This is for display only to show how we can implement One UI title bar using xPages.

The screenshot shows the Domino Designer interface with the following components:

- Applications:** A sidebar on the left showing the project structure for 'Best Practices RedWiki -- Sa' (sample.nsf). It includes folders for Forms, Views, Folders, XPages, Custom Controls, Framesets, and Pages.
- Outline:** A sidebar on the left showing the document structure. It includes a 'Root Document' with an 'XML Processing Instruction' and a 'Custom Control' containing a 'Tabbed Panel: lotusTabs' with three sub-items: 'Tab Panel: tabPanel1', 'Tab Panel: tabPanel2', and 'Tab Panel: tabPanel3'.
- docs_title - Custom Control:** The main design view showing a custom control with two tabs: 'Documents' and 'Collections'. The 'Documents' tab is selected and highlighted with a red box.
- Properties:** A panel on the right showing the properties of the selected 'Documents' tab. It includes a 'Tab Panel' section with 'Style', 'Font', 'Background', and 'Margins' properties. Below this is the 'All Properties' section, which is highlighted with a red box. It contains a table of properties and values:

Property	Value
basics	
binding	
dir	
href	
id	tabPanel1
image	
label	Documents
lang	
loaded	
rendered	
rendererType	
title	
events	
onclick	
styling	
contentStyleClass	
disableTheme	
endTabStyleClass	
middleTabStyleClass	
selectedTabStyleClass	lotusSelected
startTabStyleClass	
style	
styleClass	
themeId	
unselectedTabStyleClass	

5. Drag and drop “**global_search**” custom control below the tabbed panel.



6. Click on “**Source**” tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

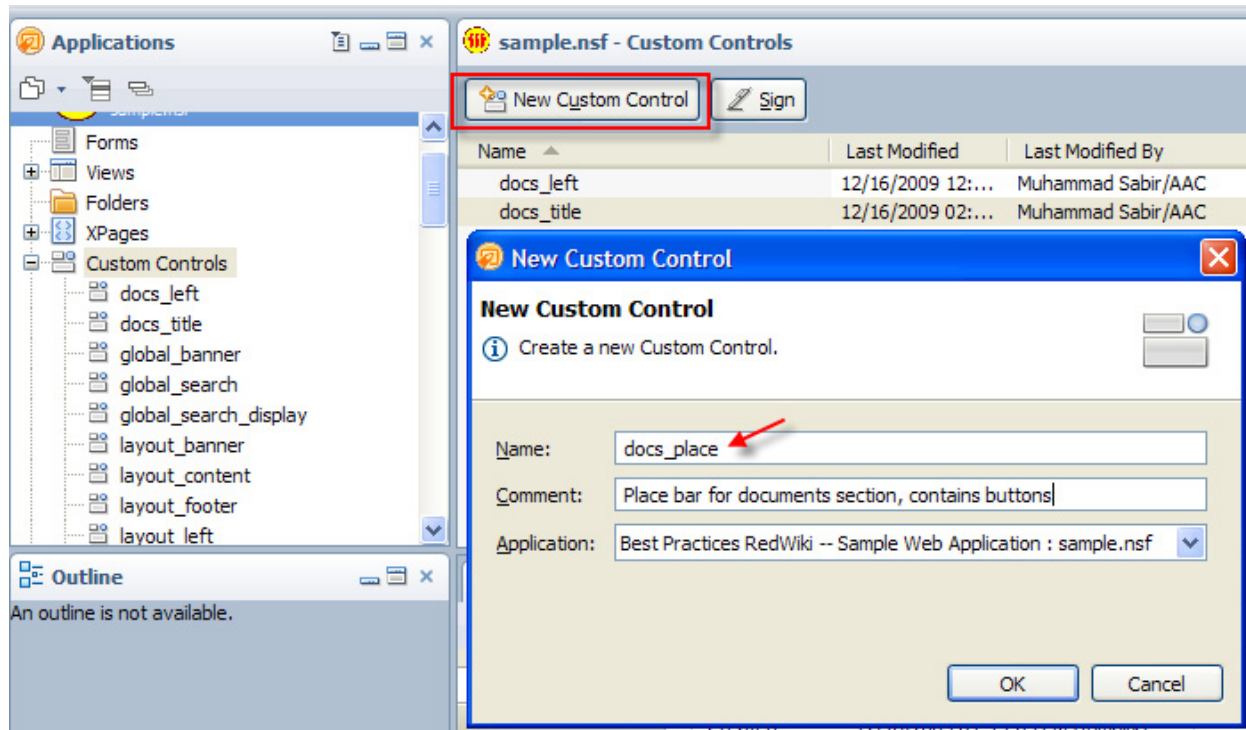
  <xp:tabbedPanel id="lotusTabs" styleClass="lotusTabs">
    <xp:tabPanel label="Documents" id="tabPanel1"
      selectedTabStyleClass="lotusSelected">
    </xp:tabPanel>
    <xp:tabPanel label="Collections" id="tabPanel2"
      selectedTabStyleClass="lotusSelected">
    </xp:tabPanel>
  </xp:tabbedPanel>
  <xc:global_search></xc:global_search>

</xp:view>
```

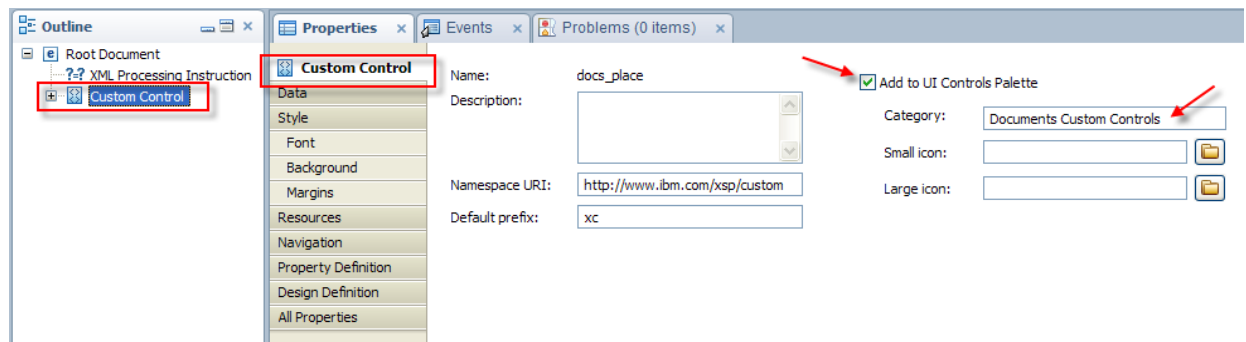
Step 8.4: Creating a custom control for place bar of document section

In this section we will build a custom control for the place bar content for documents section. We are going to add buttons to upload and delete documents.

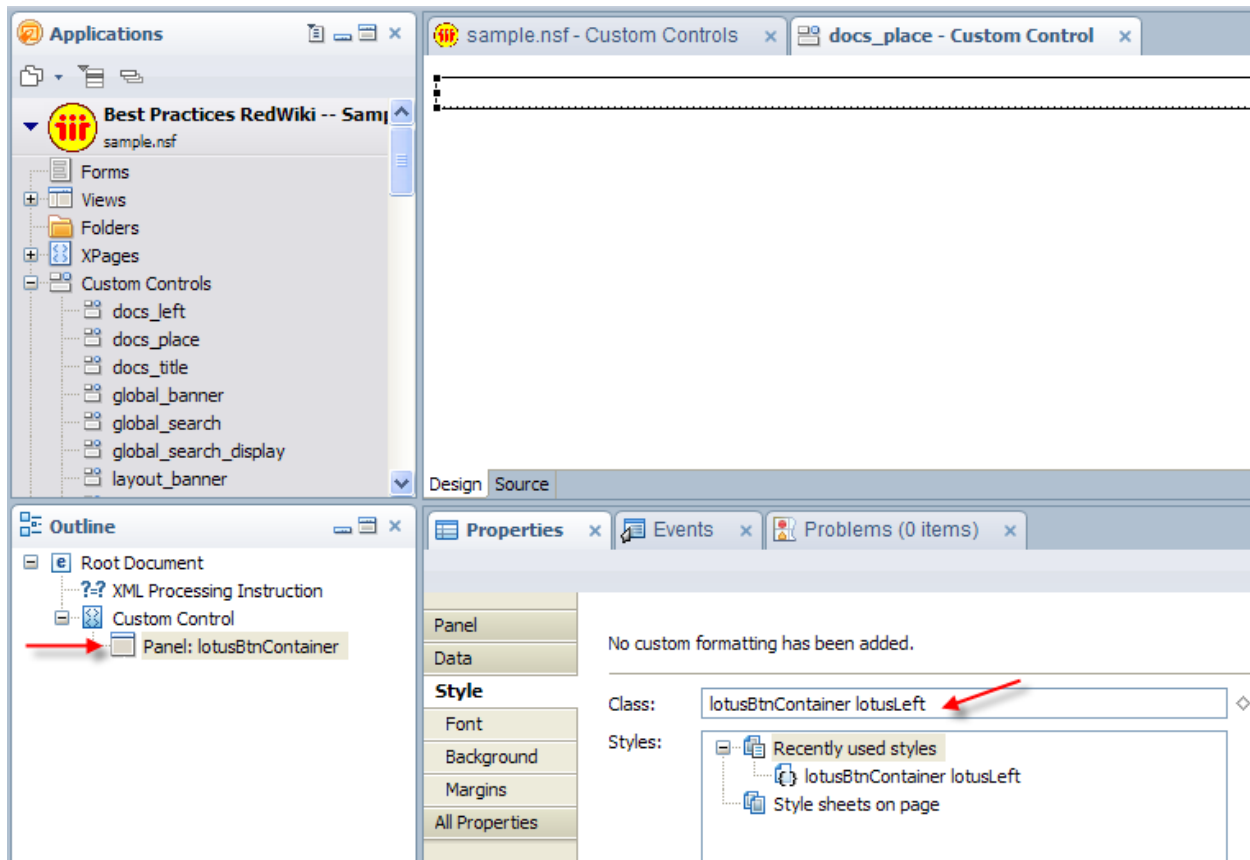
1. Create a new custom control “**docs_place**”.



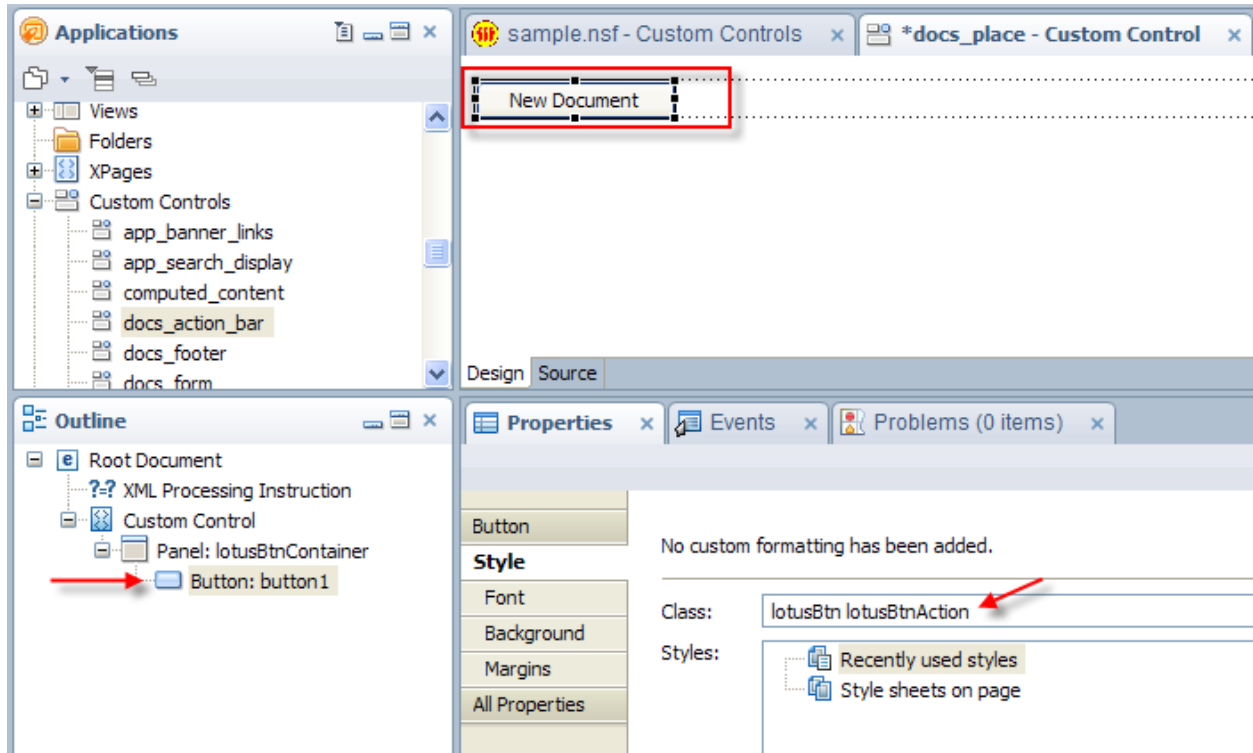
2. Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Documents Custom Controls**” as the category. This moves this custom control under the category “**Documents Custom Controls**”.



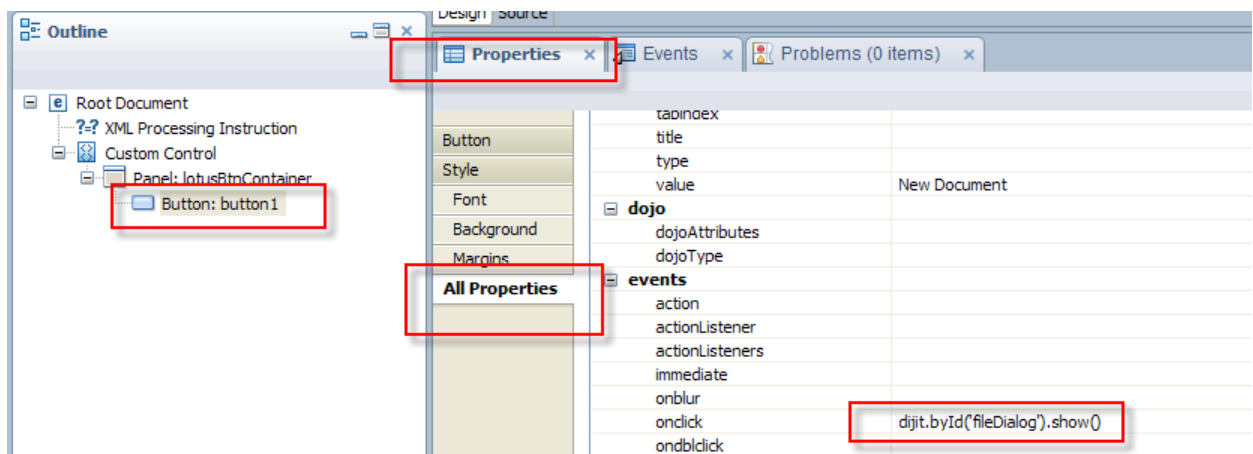
3. Drag and drop a **Panel** control. Enter “**lotusBtnContainer**” as its name and “**lotusBtnContainer lotusLeft**” as CSS style class.



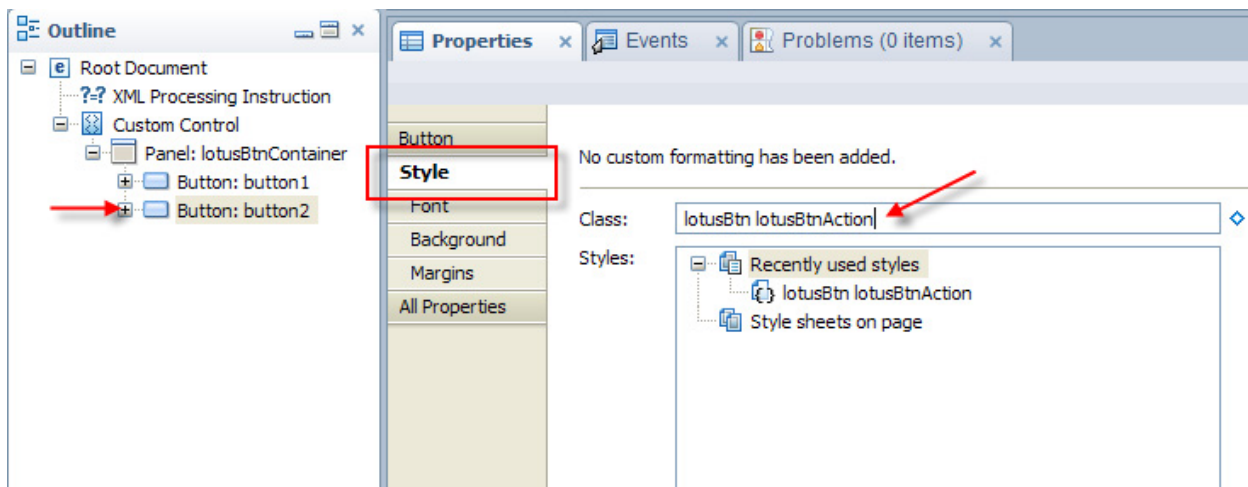
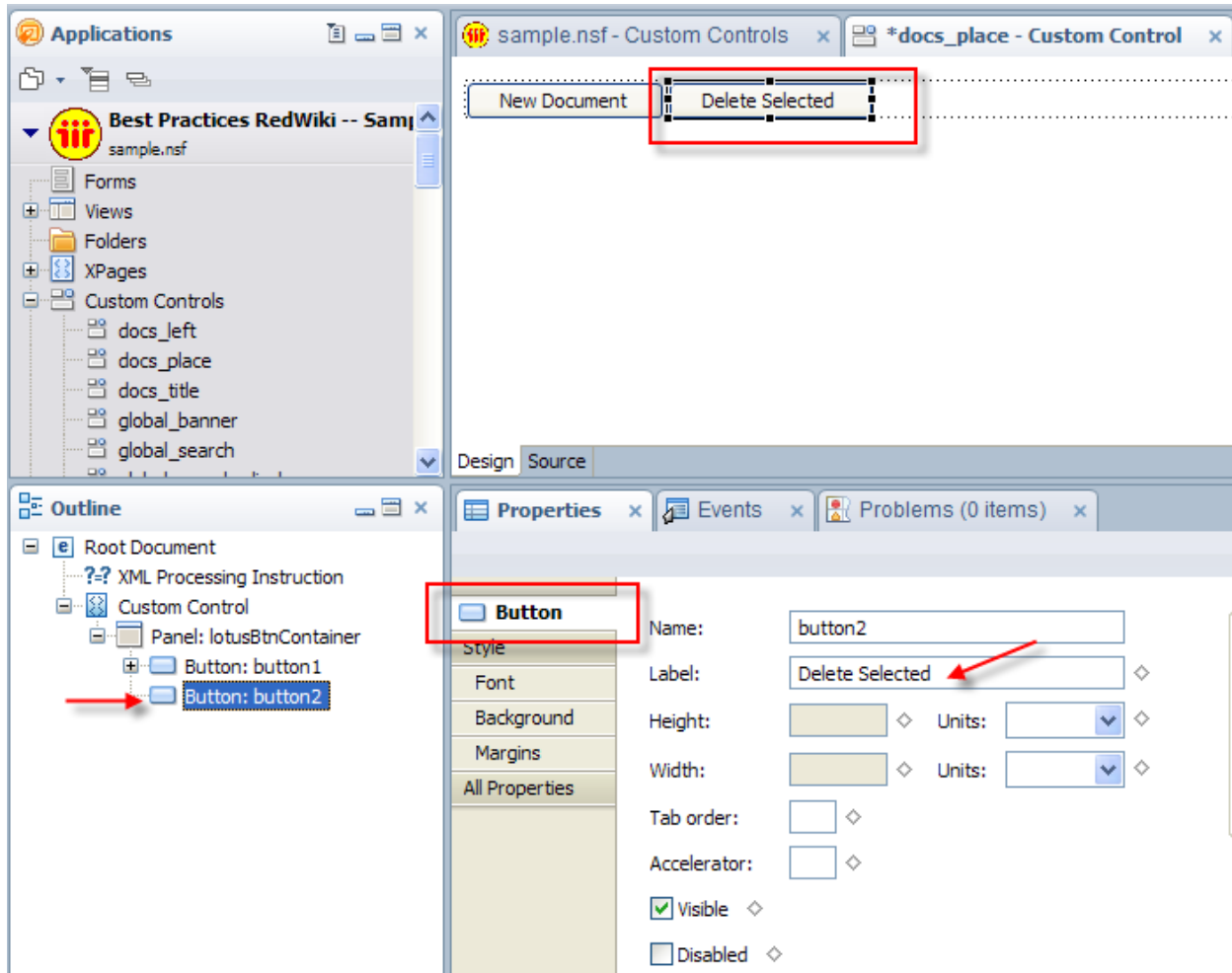
4. Drag and drop a **Button** control. Enter “**New Document**” as **Label** and “**lotusBtn lotusBtnAction**” as CSS style class.



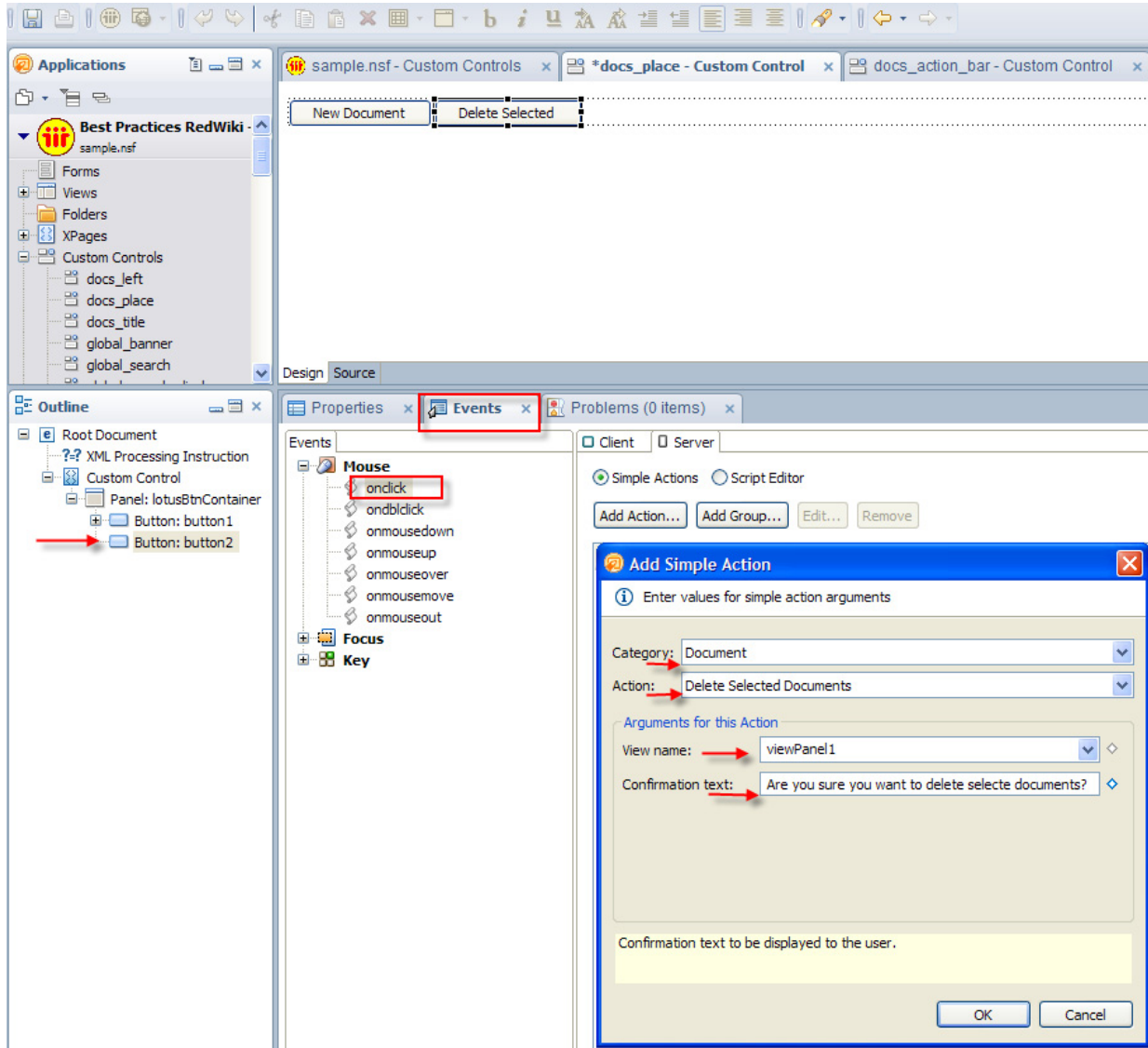
5. Under “**All Properties**”, enter the following code for “**onclick**” event: **dijit.byId('fileDialog').show()**. This code will display the file upload form as dojo dialog. For more information on using a dojo dialog, refer to dojo documentation.



6. Drag and drop a **Button** control. Enter “Delete Selected” as **Label** and “lotusBtn lotusBtnAction” as CSS style class.



7. Make sure the second button (Delete Selected), is selected. Click on **Events** tab and select **Mouse>onclick** event. Click **"Add Action"**, select **"Document"** category, select **"Delete Selected Documents"** action, and enter the following confirmation text: **"Are you sure you want to delete selected documents?"**



8. Click on **"Source"** tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

  <xp:panel styleClass="lotusBtnContainer lotusLeft"
    id="lotusBtnContainer">
```

```

<xp:button value="New Document" id="button1"
  styleClass="lotusBtn lotusBtnAction"
  onclick="diigit.byId('fileDialog').show()">
</xp:button>

<xp:button value="Delete Selected" id="button2"
  styleClass="lotusBtn lotusBtnAction">
  <xp:eventHandler event="onclick" submit="true"
    refreshMode="complete">
    <xp:this.action>
      <xp:deleteSelectedDocuments view="viewPanel1"
        message="Are you sure you want to delete
selecte documents?">
      </xp:deleteSelectedDocuments>
    </xp:this.action>
  </xp:eventHandler>
</xp:button>
</xp:panel>

</xp:view>

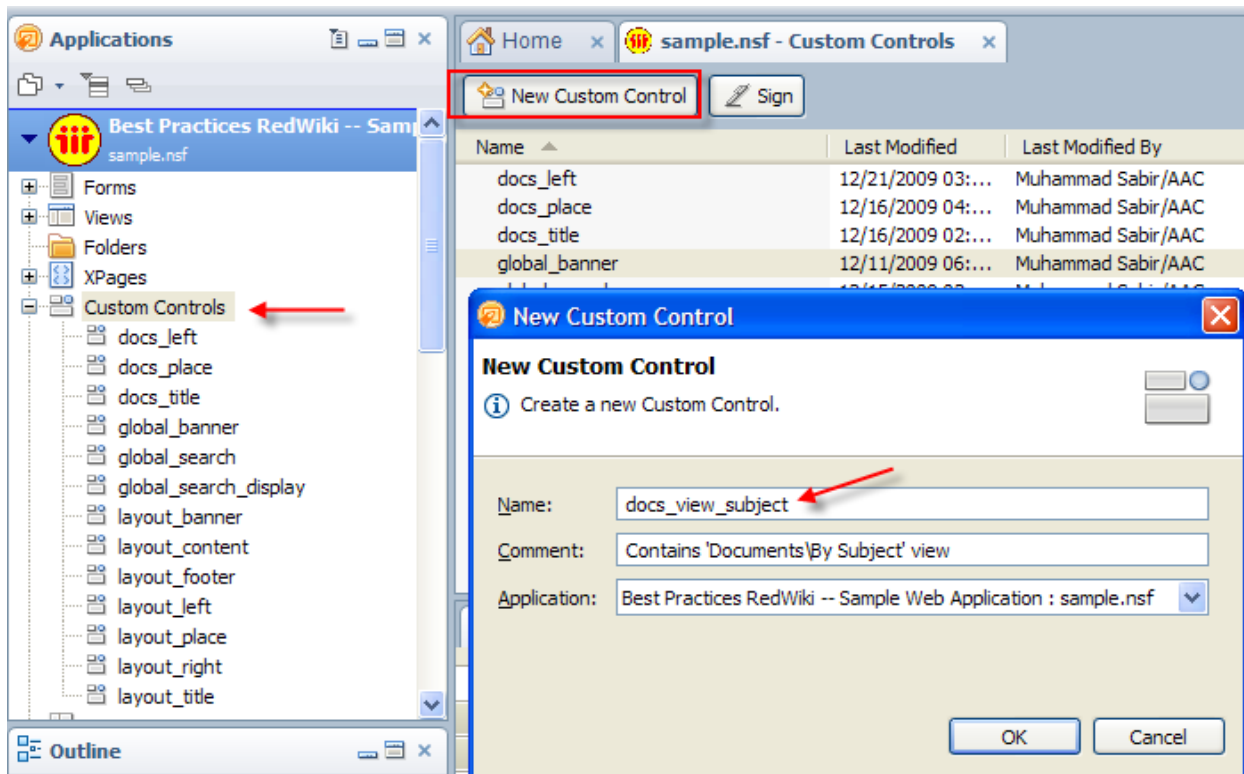
```

Step 8.5: Creating a custom controls for document related views

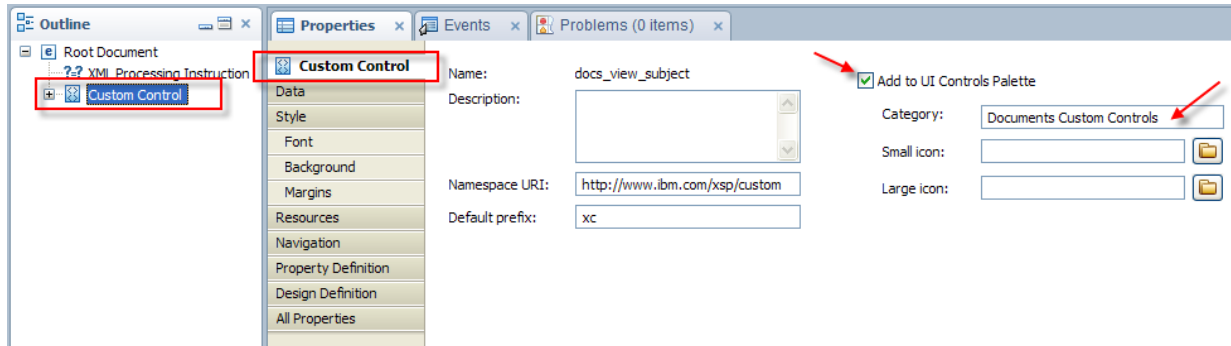
In this section we are going create custom controls for various views within the application. These custom controls are going to be dynamically added to the XPage based on query string parameter.

Step 8.5.1: Creating custom control for "Document by Subject" view

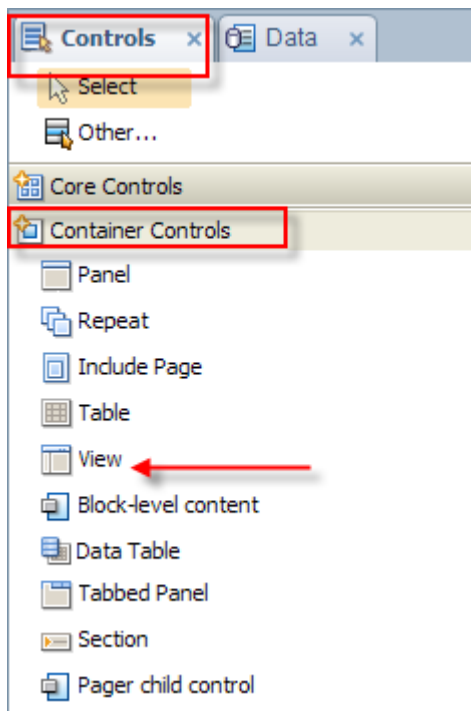
1. Create a new custom control "docs_place".



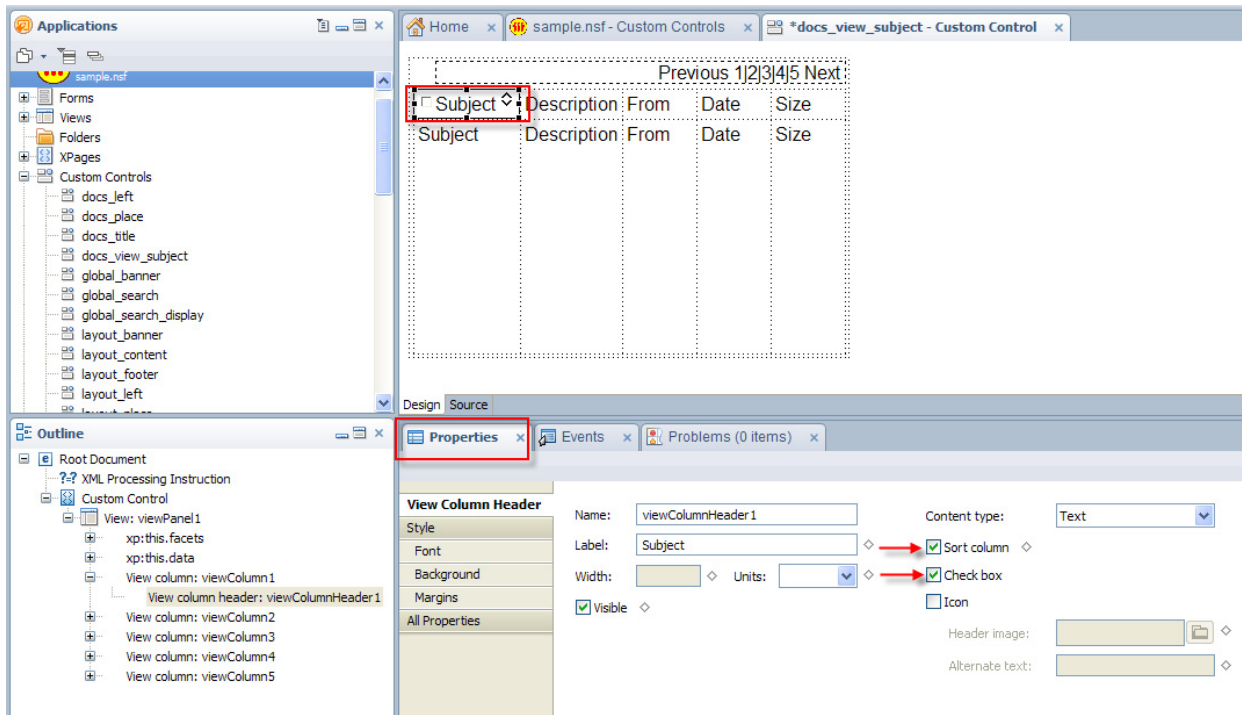
2. Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Documents Custom Controls”** as the category. This moves this custom control under the category **“Documents Custom Controls”**.



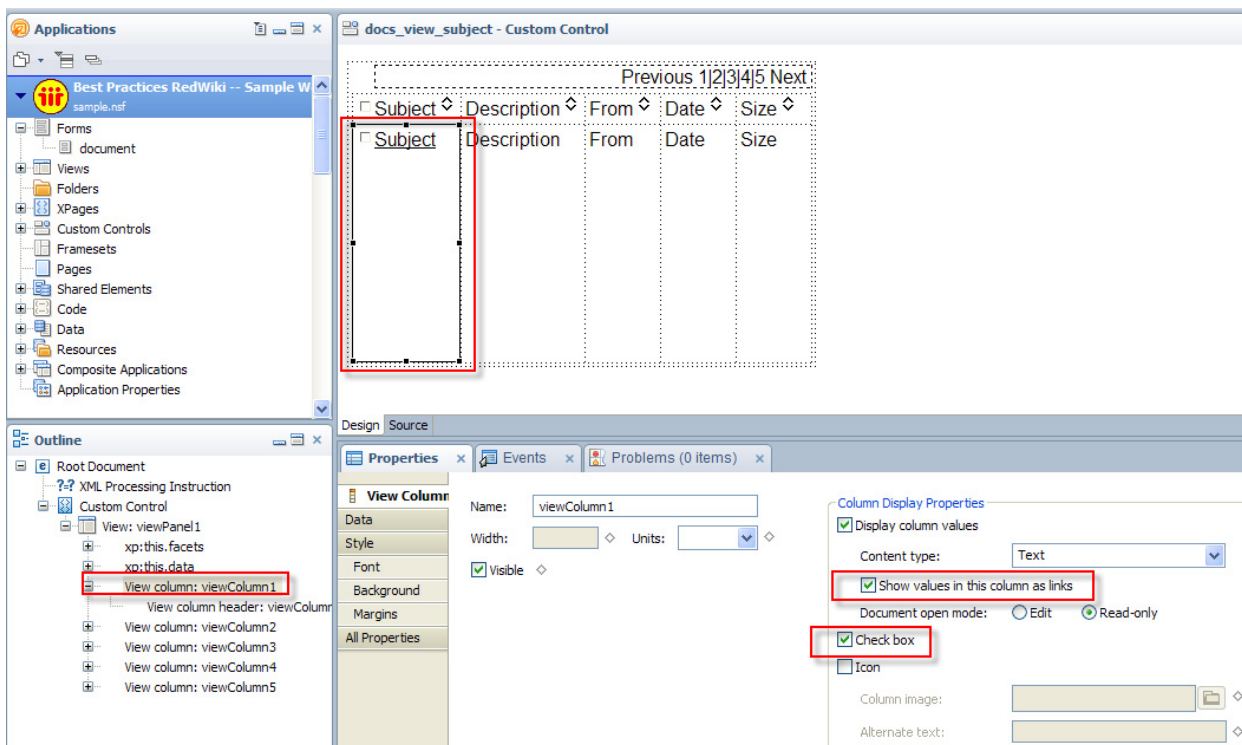
3. From the **Container Controls**, drag and drop a **“View”** control to the editor.



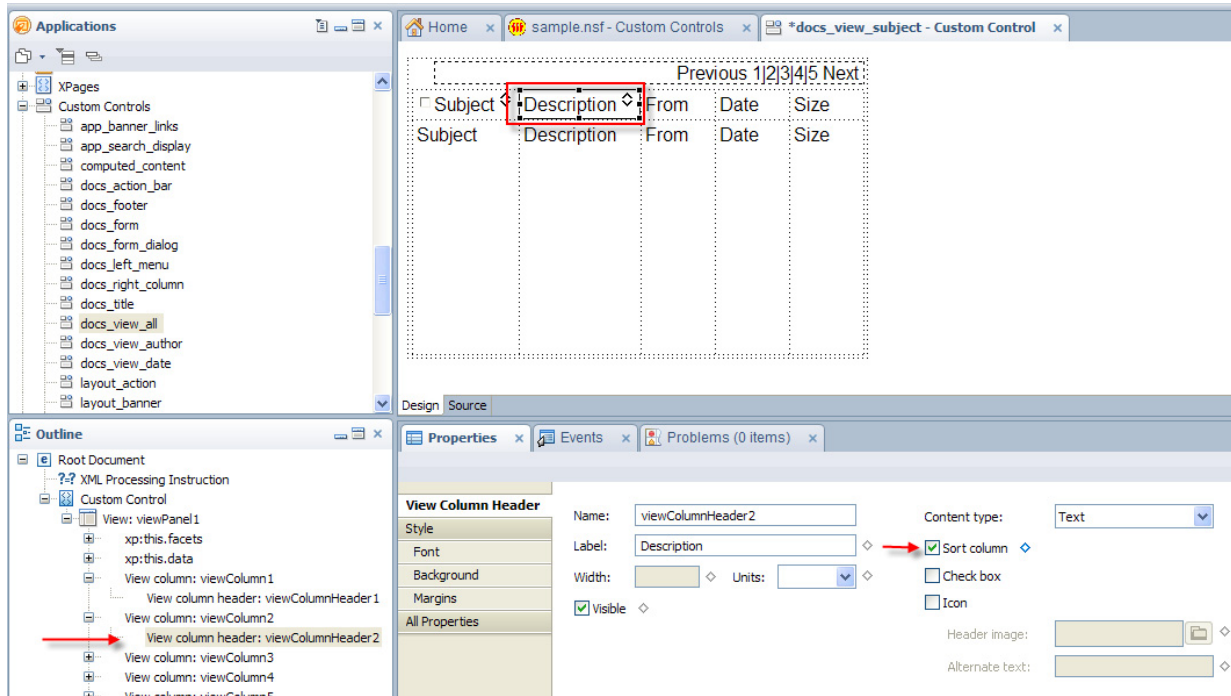
4. Select the “**Subject**” column header. Check “**Check box**” and “**Sort column**” properties.



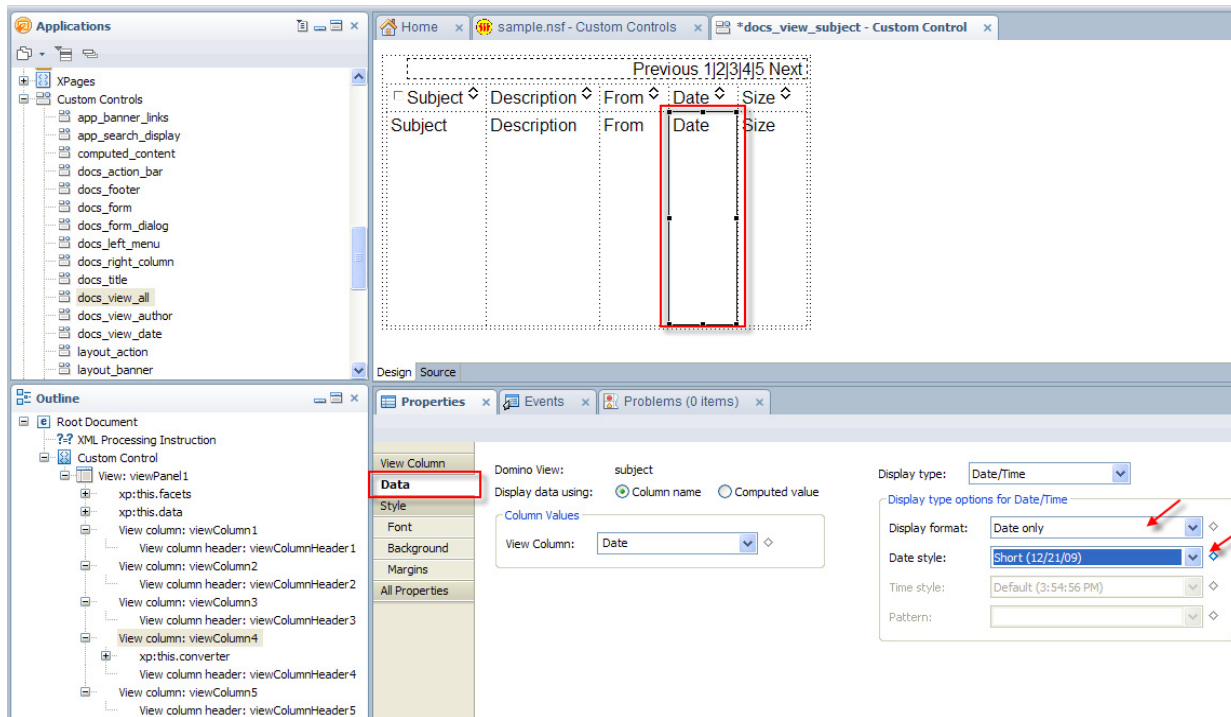
5. Select “**Subject**” column. Under “**Column Display Properties**”, check “**Show values in columns as links**” and “**Checkbox**” options. Select “**Document Mode**” as “**Read Only**”



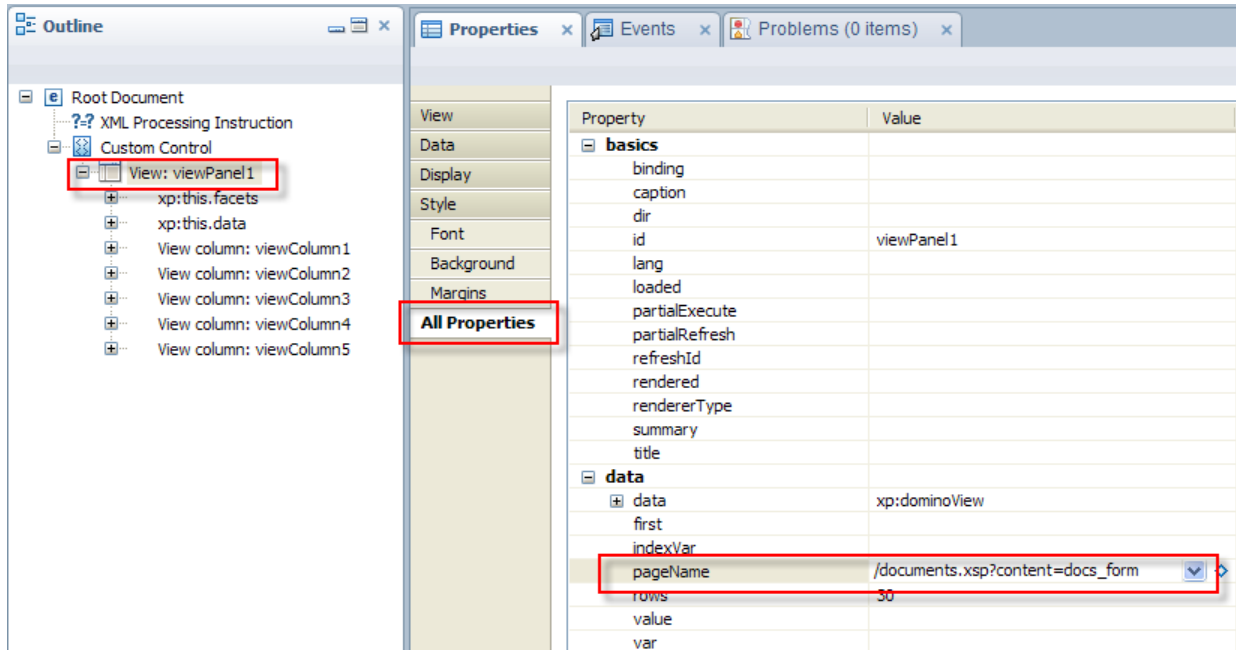
6. Select each column header and check **"Sort column"** option. This allows users to sort each column by clicking on the respective twisty.



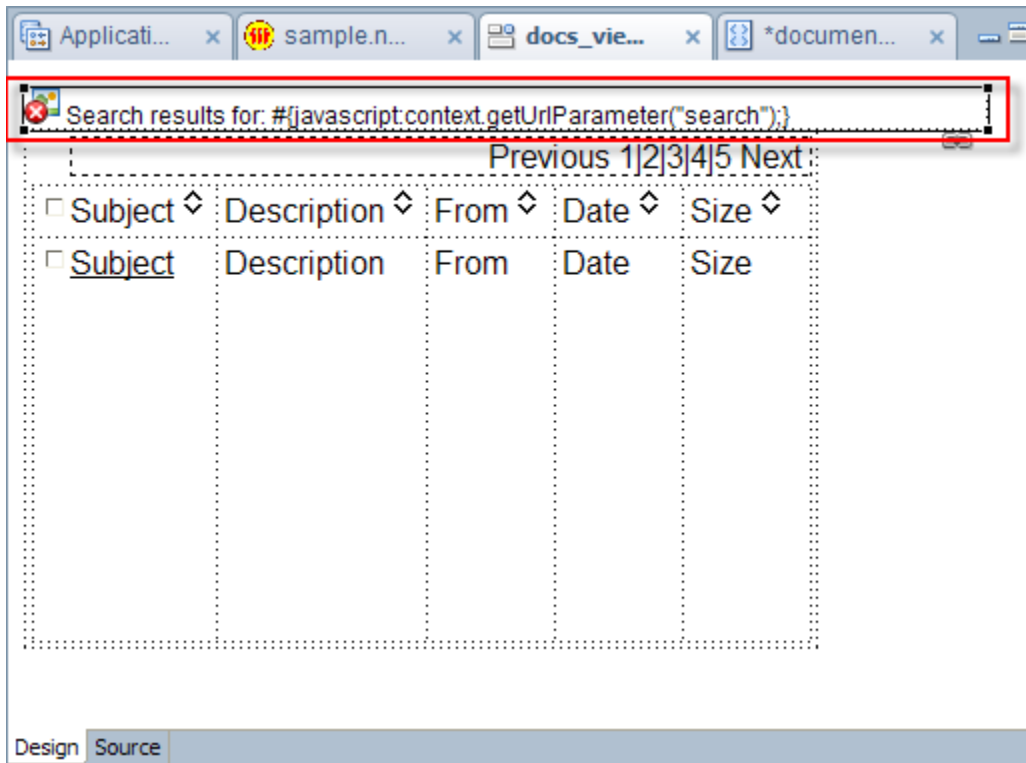
7. Select **"Date"** column, click on **"Date"** tab, under **"display type options for date/time"**, select **"Display Format"** as **"Date Only"** and **"Date style"** as **"short"**.



8. From outline palette, select **viewPanel1** and enter **“/documents.xsp?content=docs_form”** as **“pageName”** property. This will use **“documents”** XPage to display the document when user clicks on a link within this view. It also adds a **“content”** query string parameter when a link is clicked. Within documents.xsp, custom control in the main content area is computed based on the query string parameter. In this case when a link is clicked to open a document, it will display **“docs_form”** custom control in the main content area of the XPage.



9. Drag and drop a **“global_search_display”** custom control to the top-left (before the view control).



10. Click on **"Source"** tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core"
  xmlns:xc="http://www.ibm.com/xsp/custom">
  <xc:global_search_display></xc:global_search_display>
  <xp:viewPanel rows="30" id="viewPanell1"
    pageName="/documents.xsp?content=docs_form">

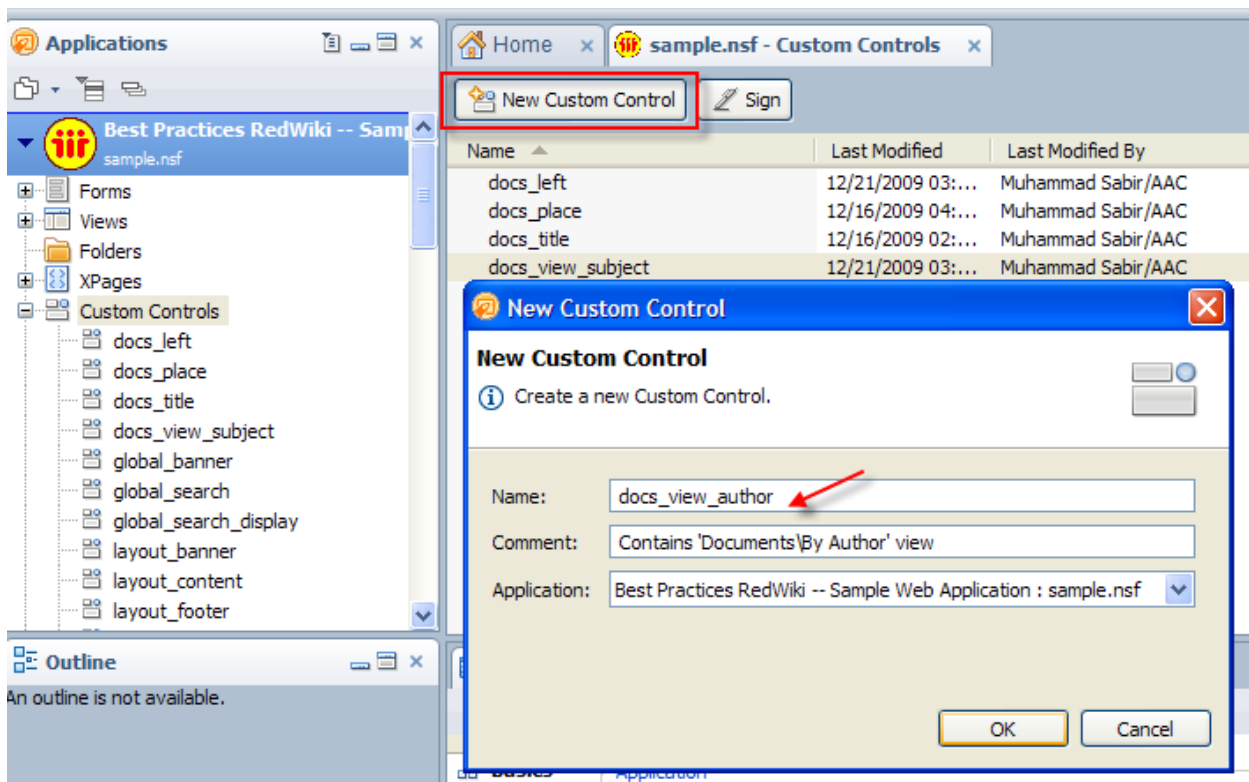
    <xp:this.facets>
      <xp:pager partialRefresh="true" layout="Previous Group
Next"
        xp:key="headerPager" id="pager1">
      </xp:pager>
    </xp:this.facets>
    <xp:this.data>
      <xp:dominoView var="subject"
viewName="subject"></xp:dominoView>
      <xp:viewColumn columnName="Subject" id="viewColumn1"
        showCheckbox="true" displayAs="link"
openDocAsReadOnly="true">
        <xp:viewColumnHeader value="Subject" id="viewColumnHeader1"
          sortable="true" showCheckbox="true">
        </xp:viewColumnHeader>
      </xp:viewColumn>
    </xp:this.data>
  </xp:viewPanel>
</xp:view>
```



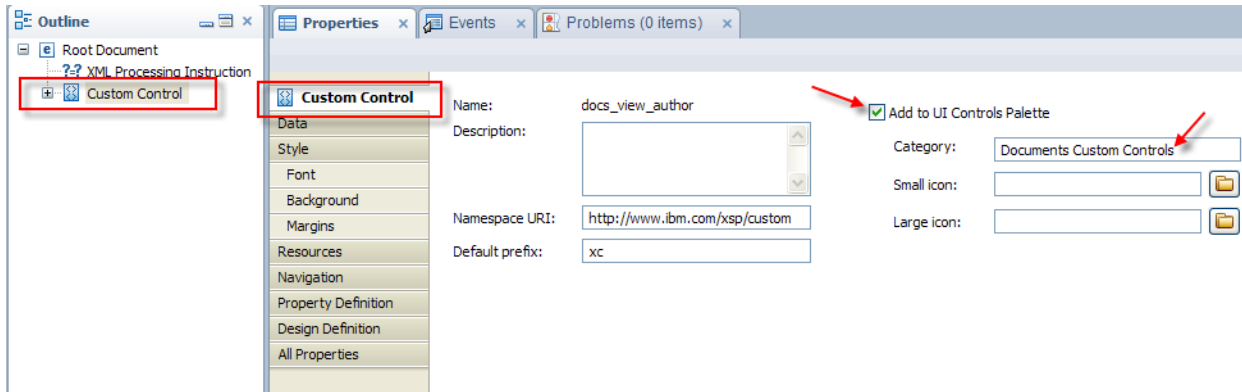
```
<xp:viewColumn columnName="Description" id="viewColumn2">
  <xp:viewColumnHeader value="Description"
    id="viewColumnHeader2" sortable="true">
  </xp:viewColumnHeader>
</xp:viewColumn>
<xp:viewColumn columnName="From" id="viewColumn3">
  <xp:viewColumnHeader value="From" id="viewColumnHeader3"
    sortable="true">
  </xp:viewColumnHeader>
</xp:viewColumn>
<xp:viewColumn columnName="Date" id="viewColumn4">
  <xp:viewColumnHeader value="Date" id="viewColumnHeader4"
    sortable="true">
  </xp:viewColumnHeader>
</xp:viewColumn>
<xp:viewColumn columnName="Size" id="viewColumn5">
  <xp:viewColumnHeader value="Size" id="viewColumnHeader5"
    sortable="true">
  </xp:viewColumnHeader>
</xp:viewColumn>
</xp:viewPanel>
</xp:view>
```

Step 8.5.2: Creating custom control for “Document by Author” view

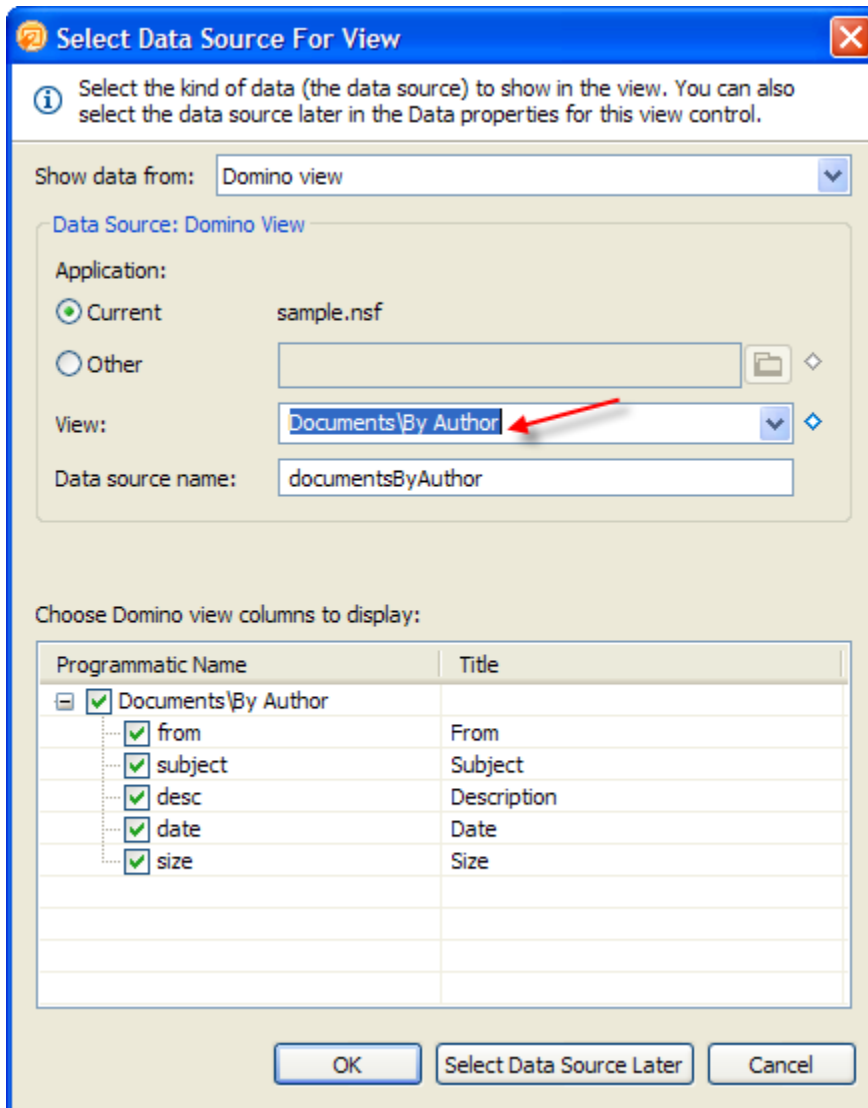
1. Create a new custom control “docs_view_author”.



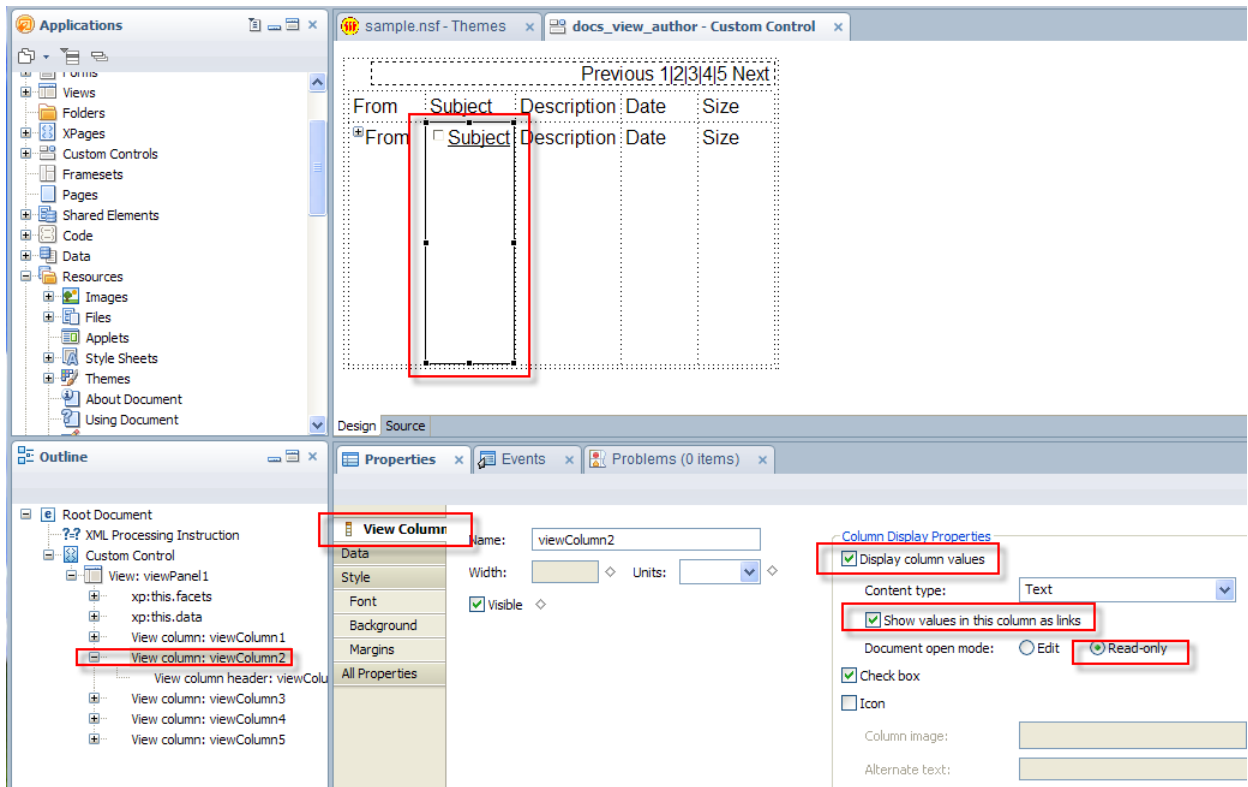
- Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Documents Custom Controls”** as the category. This moves this custom control under the category **“Documents Custom Controls”**.



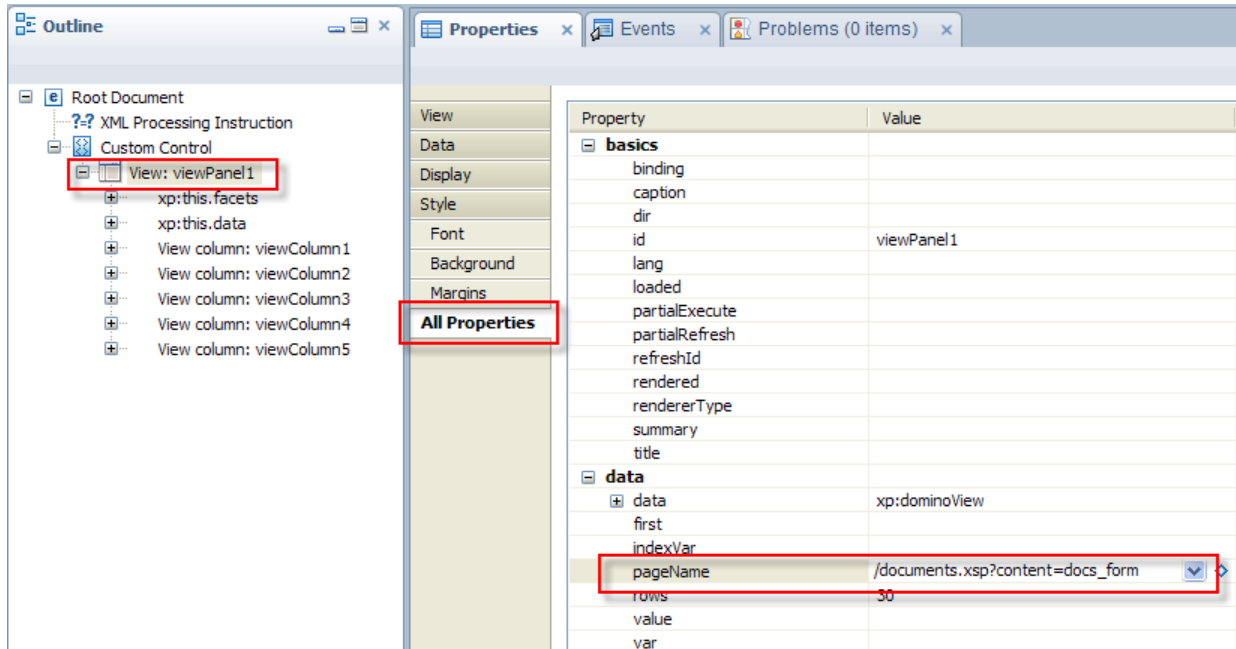
- From the **Container Controls**, drag and drop a **“View”** control to the editor. Select **“Documents\By Author”** view when prompted.



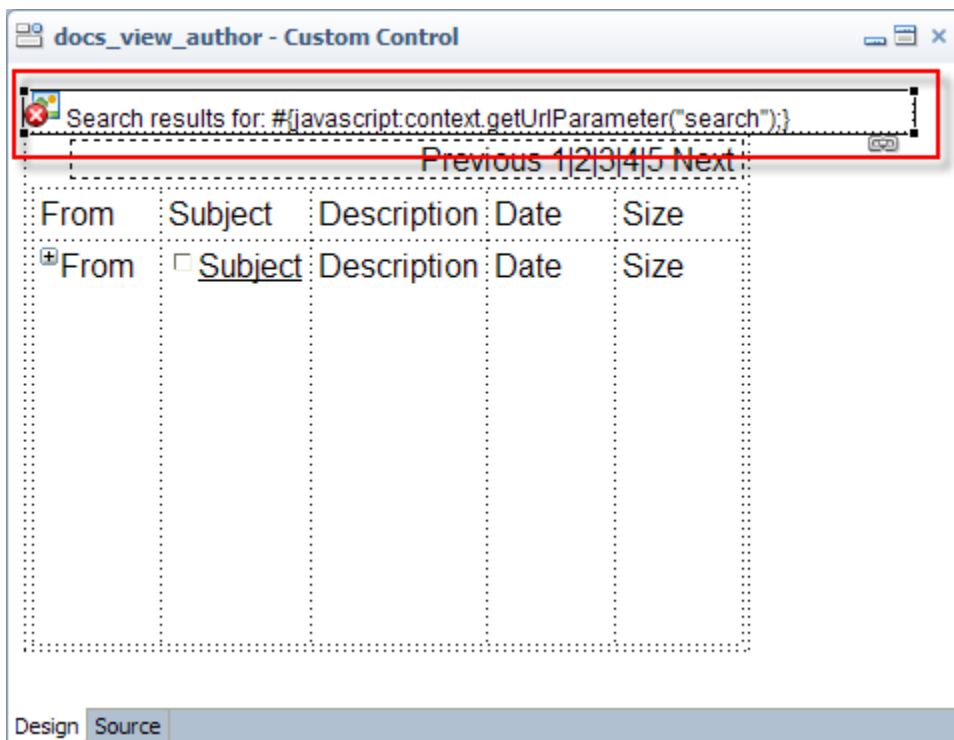
4. Select **“Subject”** column. Under **“Column Display Properties”**, check **“Show values in columns as links”** and **“Checkbox”** options. Select **“Document Mode”** as **“Read Only”**



5. From outline palette, select **viewPanel1** and enter `"/documents.xsp?content=docs_form"` as **"pageName"** property. This will use **"documents"** XPage to display the document when user clicks on a link within this view. It also adds a **"content"** query string parameter when a link is clicked. Within documents.xsp, custom control in the main content area is computed based on the query string parameter. In this case when a link is clicked to open a document, it will display **"docs_form"** custom control in the main content area of the XPage.



6. Drag and drop a “**global_search_display**” custom control to the top-left (before the view control).



7. Click on “**Source**” tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core"
  xmlns:xc="http://www.ibm.com/xsp/custom">

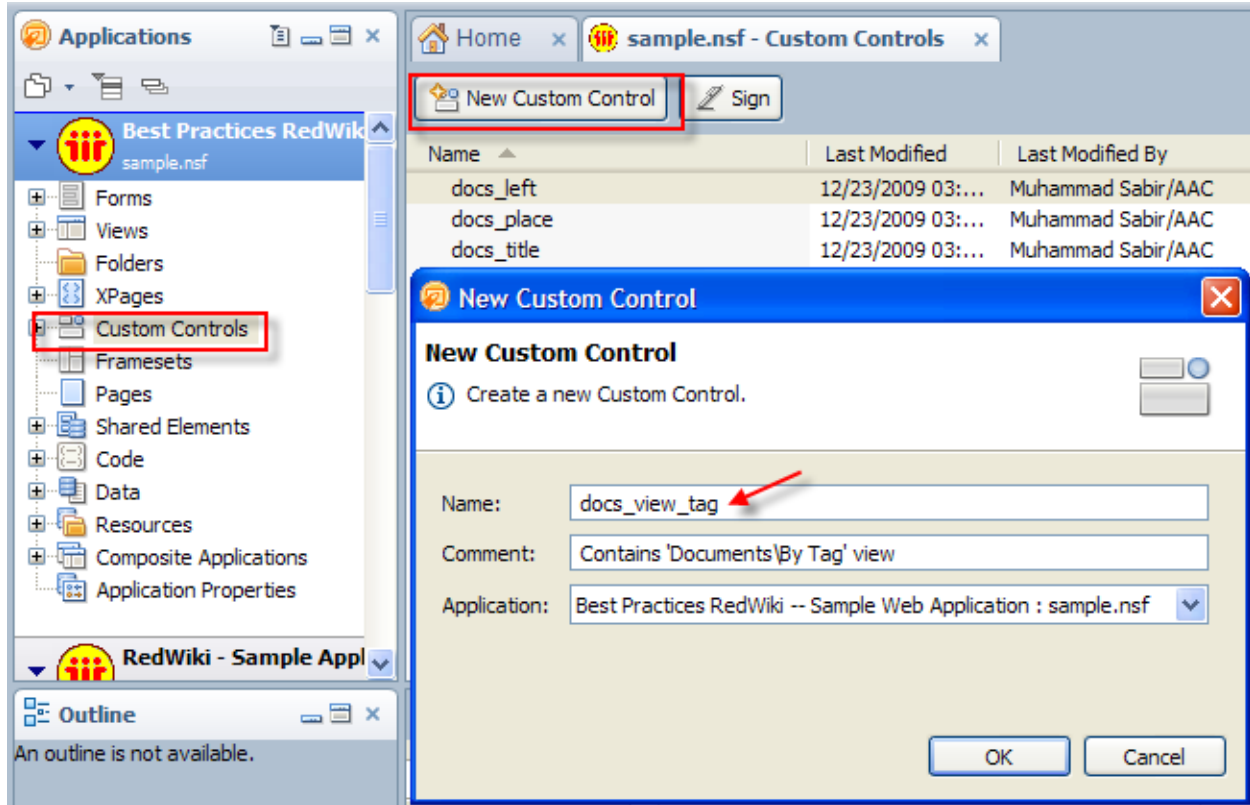
  <xc:global_search_display></xc:global_search_display>
  <xp:viewPanel rows="30" id="viewPanel1"
    pageName="/documents.xsp?content=docs_form">
    <xp:this.facets>
      <xp:pager partialRefresh="true" layout="Previous Group
Next "
        xp:key="headerPager" id="pager1">
        </xp:pager>
      </xp:this.facets>
      <xp:this.data>
        <xp:dominoView var="documentsByAuthor"
          viewName="Documents\By Author">
          </xp:dominoView>
        </xp:this.data>

        <xp:viewColumn columnName="From" id="viewColumn1">
          <xp:viewColumnHeader value="From"
id="viewColumnHeader1"></xp:viewColumnHeader>
        </xp:viewColumn>
        <xp:viewColumn columnName="Subject" id="viewColumn2">
          displayAs="link" openDocAsReadonly="true"
showCheckbox="true">
          <xp:viewColumnHeader value="Subject"
id="viewColumnHeader2">
          </xp:viewColumnHeader>
        </xp:viewColumn>
        <xp:viewColumn columnName="Description" id="viewColumn3">
          <xp:viewColumnHeader value="Description"
id="viewColumnHeader3">
          </xp:viewColumnHeader>
        </xp:viewColumn>
        <xp:viewColumn columnName="Date" id="viewColumn4">
          <xp:viewColumnHeader value="Date"
id="viewColumnHeader4"></xp:viewColumnHeader>
        </xp:viewColumn>
        <xp:viewColumn columnName="Size" id="viewColumn5">
          <xp:viewColumnHeader value="Size"
id="viewColumnHeader5"></xp:viewColumnHeader>
        </xp:viewColumn>
      </xp:viewPanel>
    </xp:view>
  </xp:view>

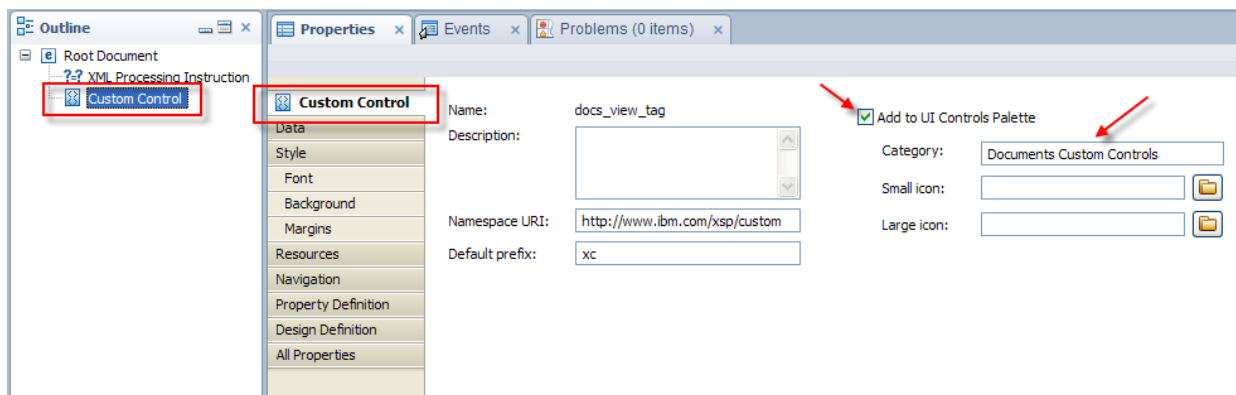
```

Step 8.5.3: Creating custom control for “Document by Tag” view

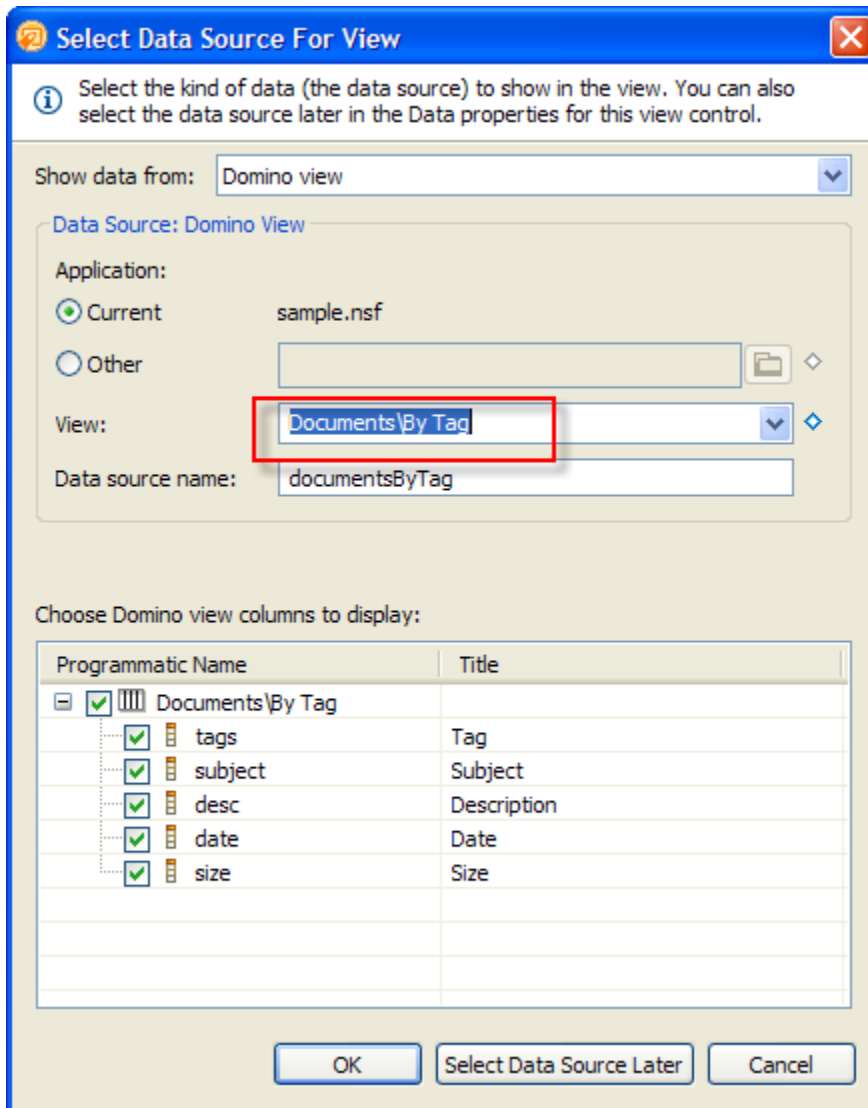
1. Create a new custom control “docs_view_tag”.



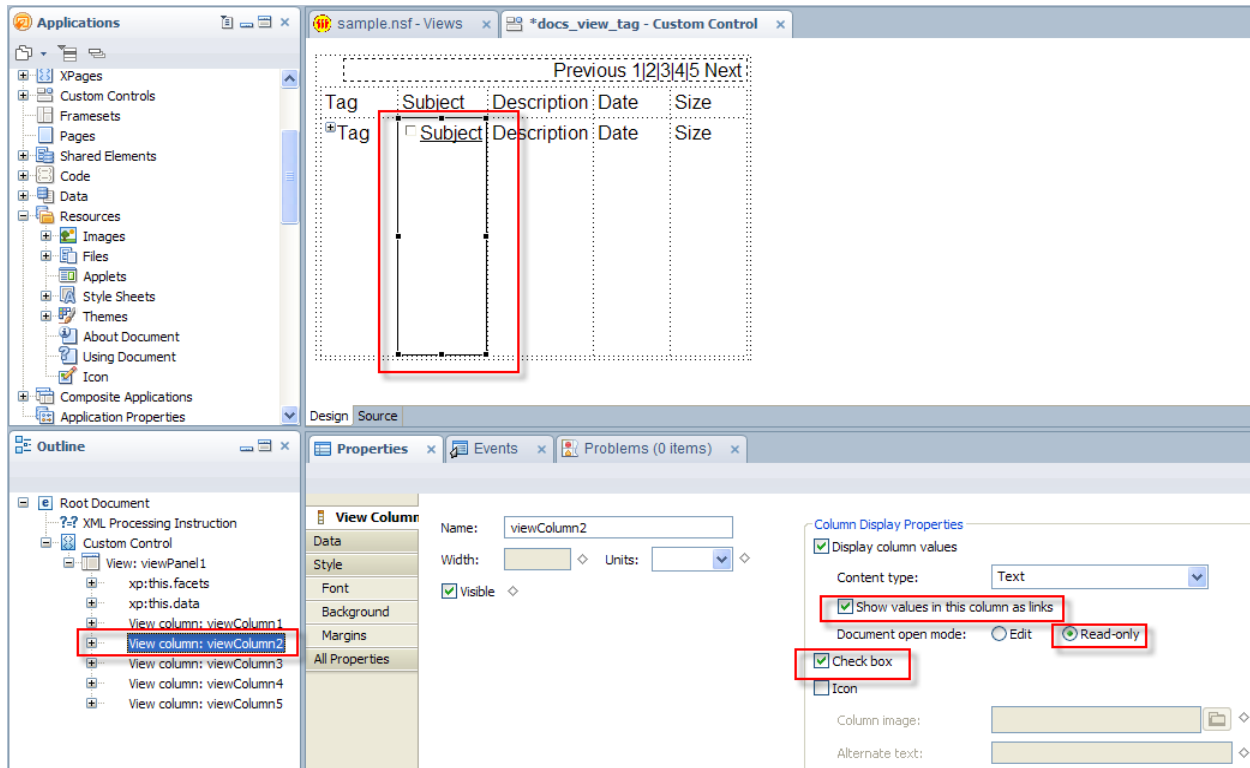
2. Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Documents Custom Controls”** as the category. This moves this custom control under the category **“Documents Custom Controls”**.



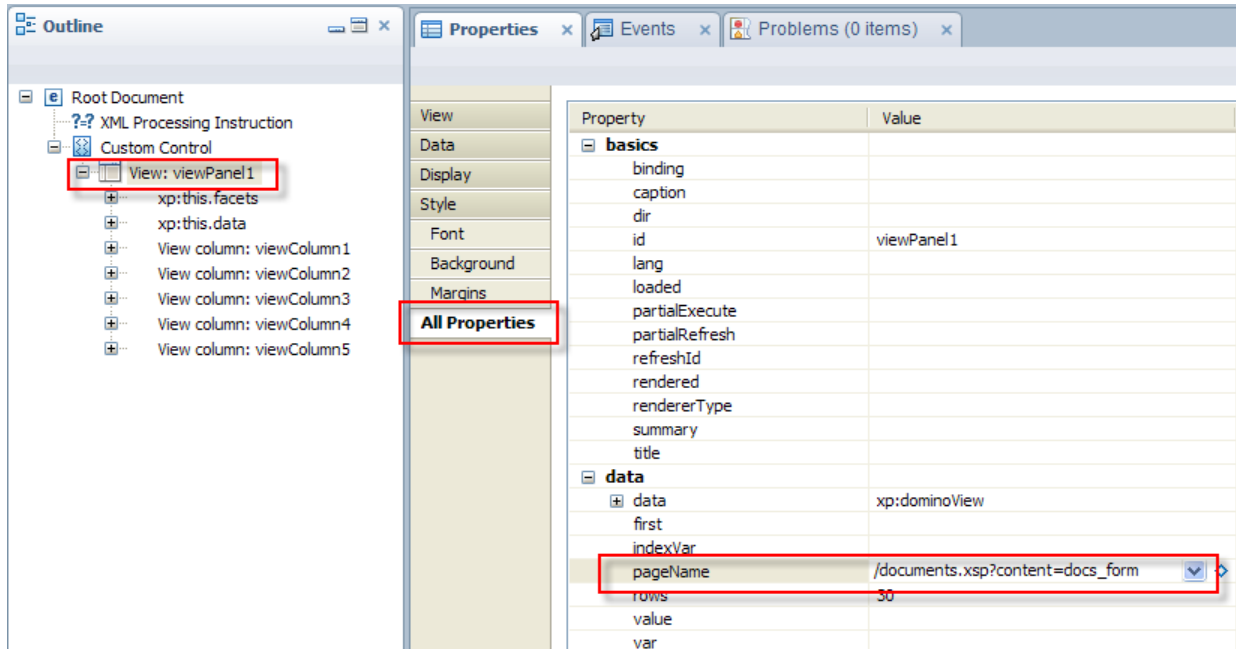
3. From the **Container Controls**, drag and drop a **“View”** control to the editor. Select **“Documents\By Tag”** view when prompted.



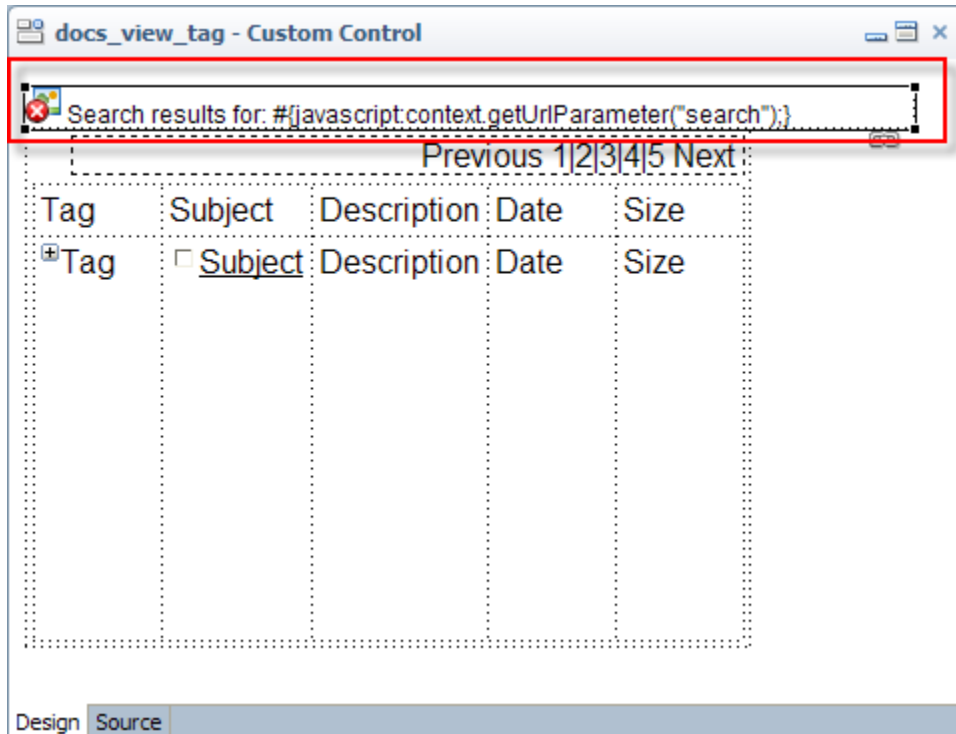
4. Select **“Subject”** column. Under **“Column Display Properties”**, check **“Show values in columns as links”** and **“Checkbox”** options. Select **“Document Mode”** as **“Read Only”**



5. From outline palette, select **viewPanel1** and enter **“/documents.xsp?content=docs_form”** as **“pageName”** property. This will use **“documents”** XPage to display the document when user clicks on a link within this view. It also adds a **“content”** query string parameter when a link is clicked. Within documents.xsp, custom control in the main content area is computed based on the query string parameter. In this case when a link is clicked to open a document, it will display **“docs_form”** custom control in the main content area of the XPage.



6. Drag and drop a “**global_search_display**” custom control to the top-left (before the view control).



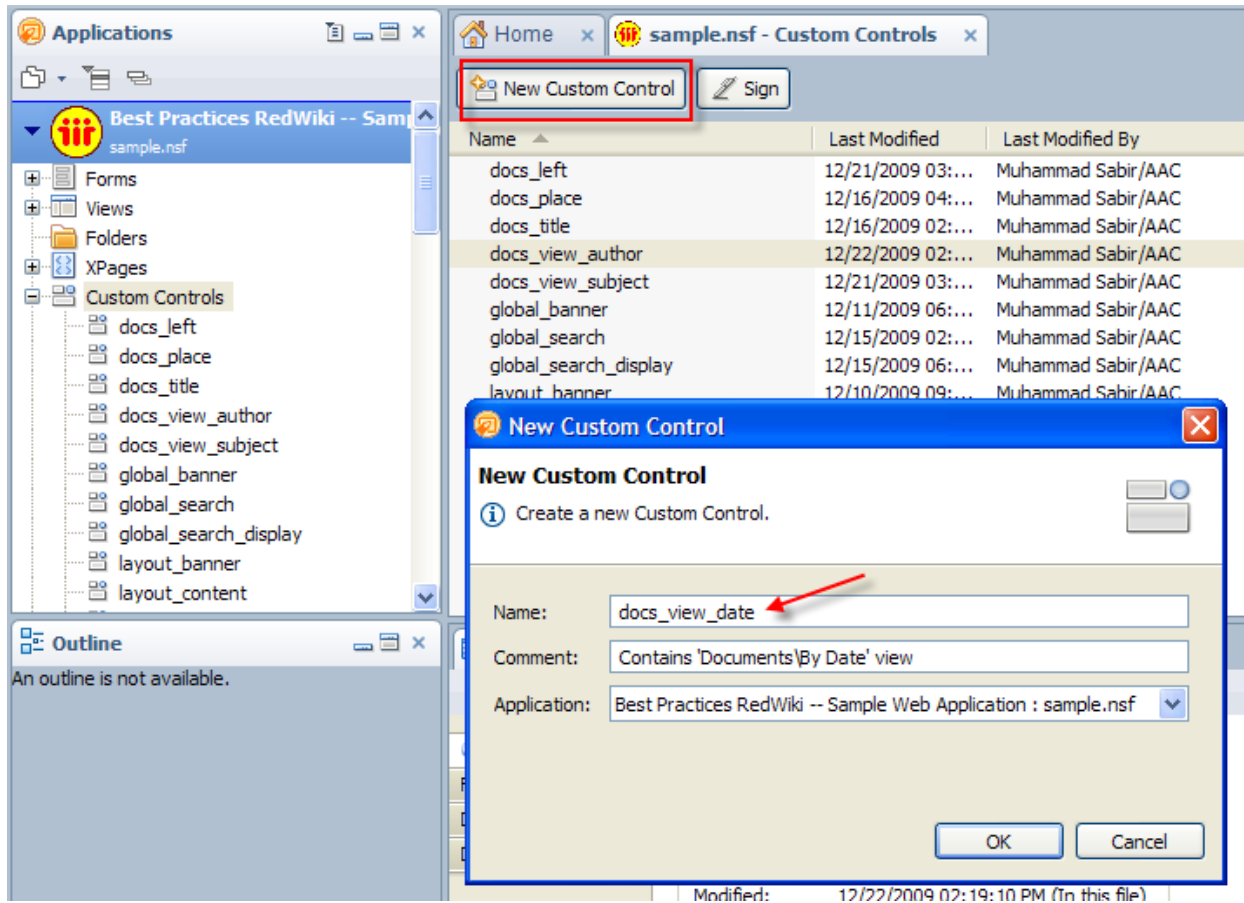
- Click on **“Source”** tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

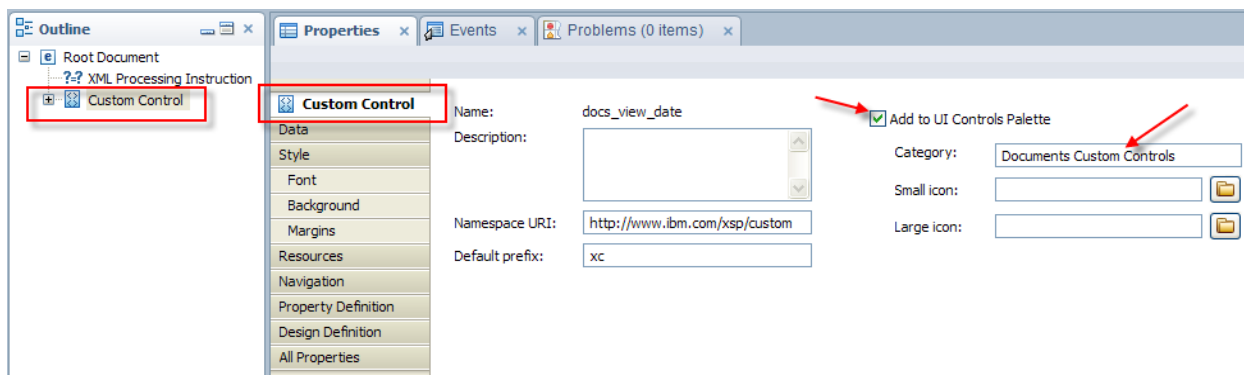
  <xc:global_search_display></xc:global_search_display>
  <xp:viewPanel rows="30" id="viewPanel1"
    pageName="/documents.xsp?content=docs_form">
    <xp:this.facets>
      <xp:pager partialRefresh="true" layout="Previous Group
Next"
        xp:key="headerPager" id="pager1">
      </xp:pager>
    </xp:this.facets>
    <xp:this.data>
      <xp:dominoView var="documentsByTag"
        viewName="Documents\By Tag">
      </xp:dominoView>
    </xp:this.data>
    <xp:viewColumn columnName="Tag" id="viewColumn1">
      <xp:viewColumnHeader value="Tag"
id="viewColumnHeader1"></xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="Subject" id="viewColumn2"
      displayAs="link" openDocAsReadonly="true"
showCheckbox="true">
      <xp:viewColumnHeader value="Subject"
id="viewColumnHeader2">
      </xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="Description" id="viewColumn3">
      <xp:viewColumnHeader value="Description"
id="viewColumnHeader3">
      </xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="Date" id="viewColumn4">
      <xp:viewColumnHeader value="Date"
id="viewColumnHeader4"></xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="Size" id="viewColumn5">
      <xp:viewColumnHeader value="Size"
id="viewColumnHeader5"></xp:viewColumnHeader>
    </xp:viewColumn>
  </xp:viewPanel>
</xp:view>
```

Step 8.5.4: Creating custom control for “Document by Date” view

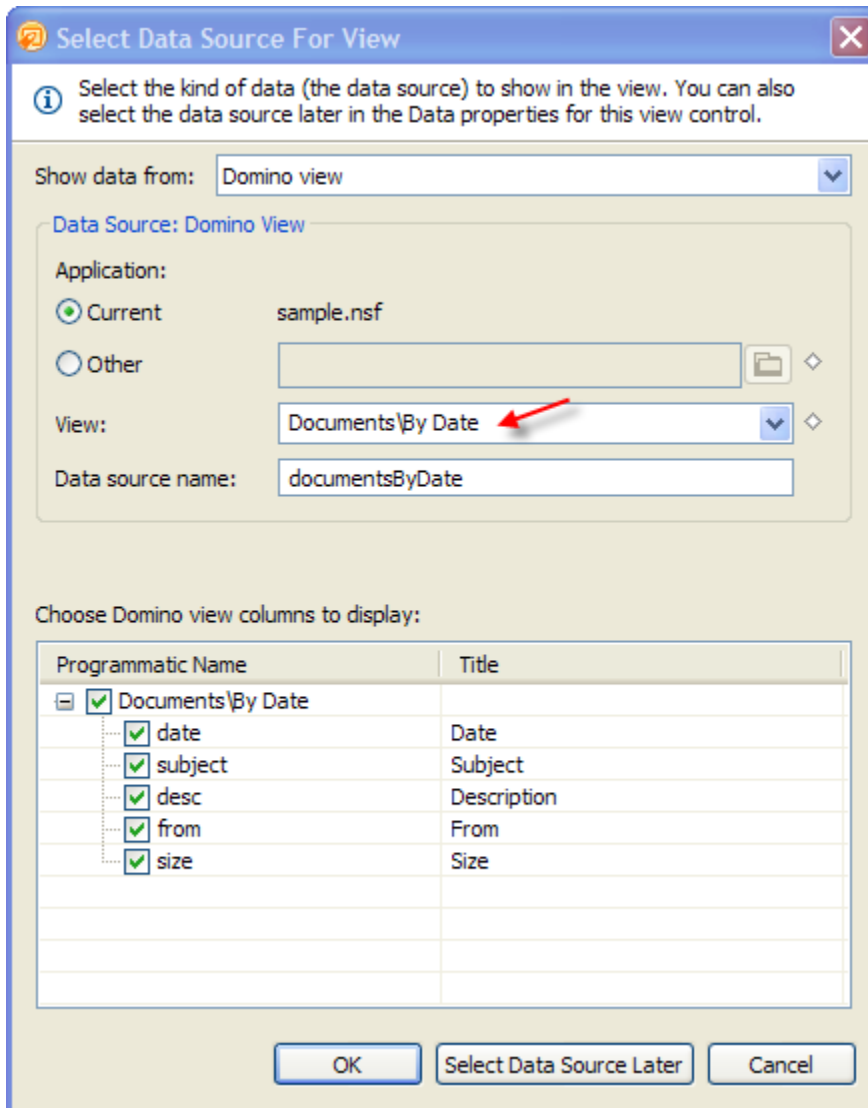
- Create a new custom control **“docs_view_date”**.



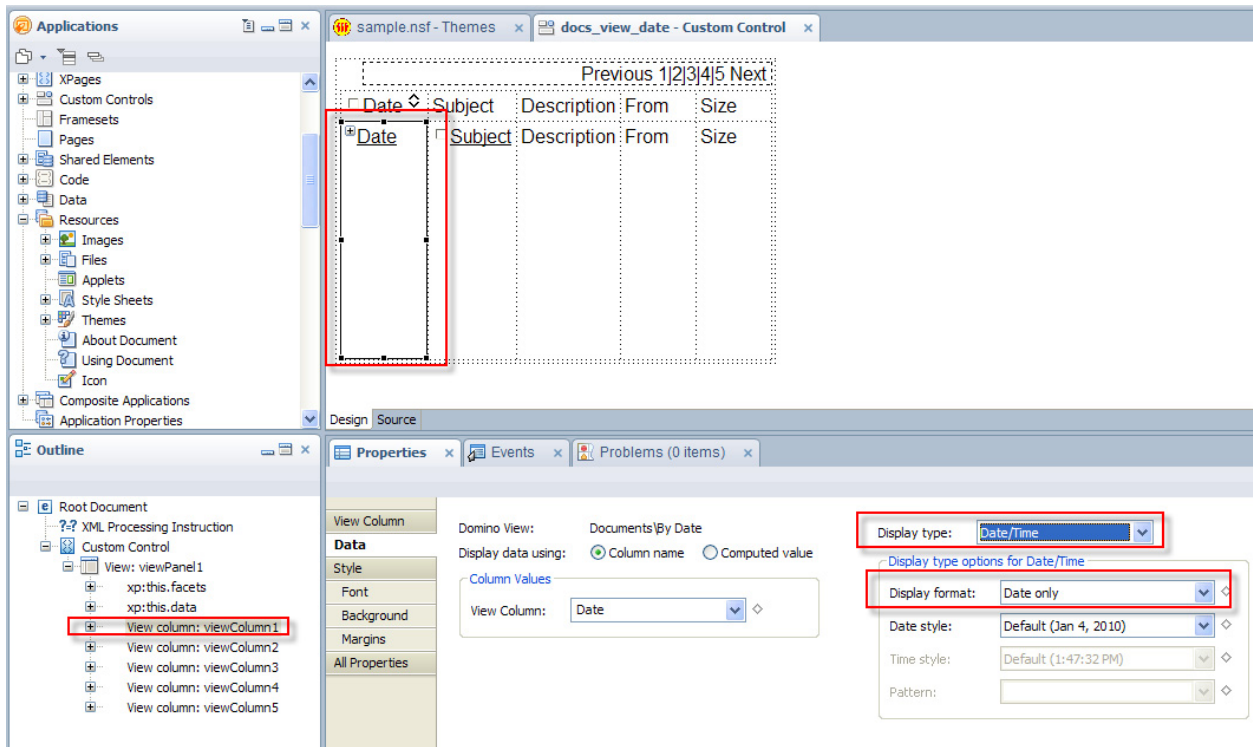
- Under the Properties tab, make sure Custom Control is selected and check **“Add to UI Controls Palette”** and enter **“Documents Custom Controls”** as the category. This moves this custom control under the category **“Documents Custom Controls”**.



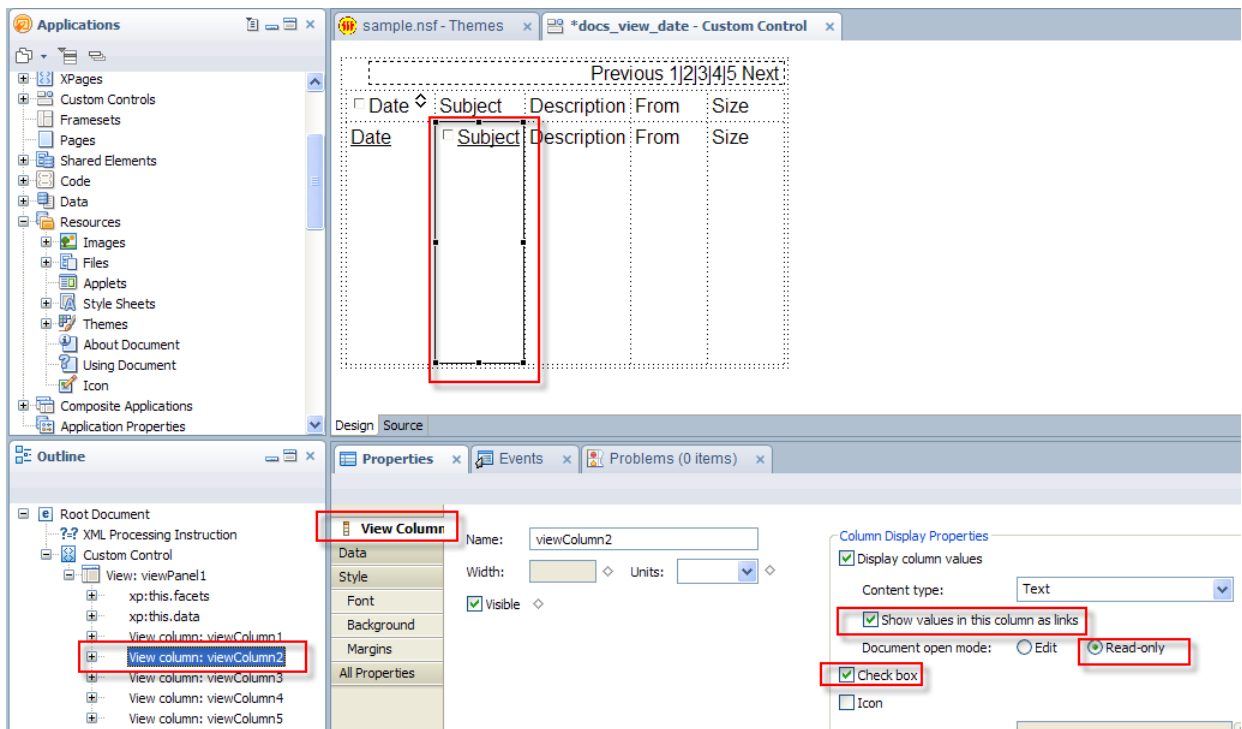
- From the **Container Controls**, drag and drop a **“View”** control to the editor. Select **“Documents\By Date”** view when prompted.



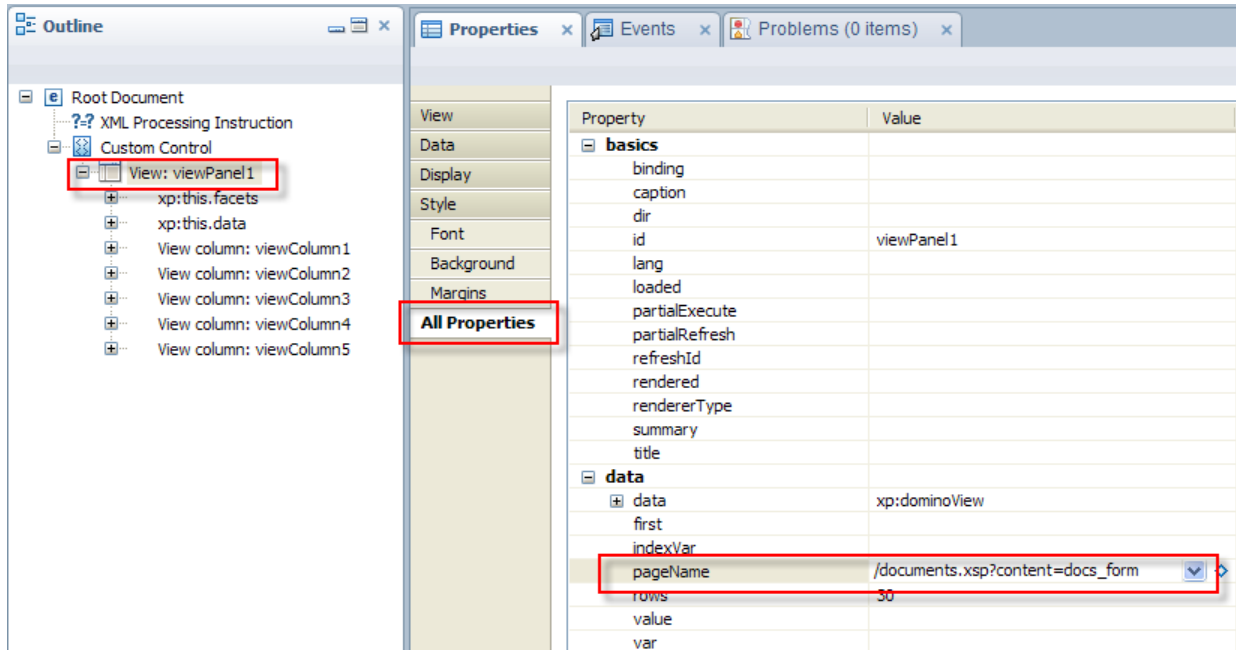
4. Select **"Date"** column, click on **"Date"** tab, under **"display type options for date/time"**, select **"Display Format"** as **"Date Only"** and **"Date style"** as **"default"**.



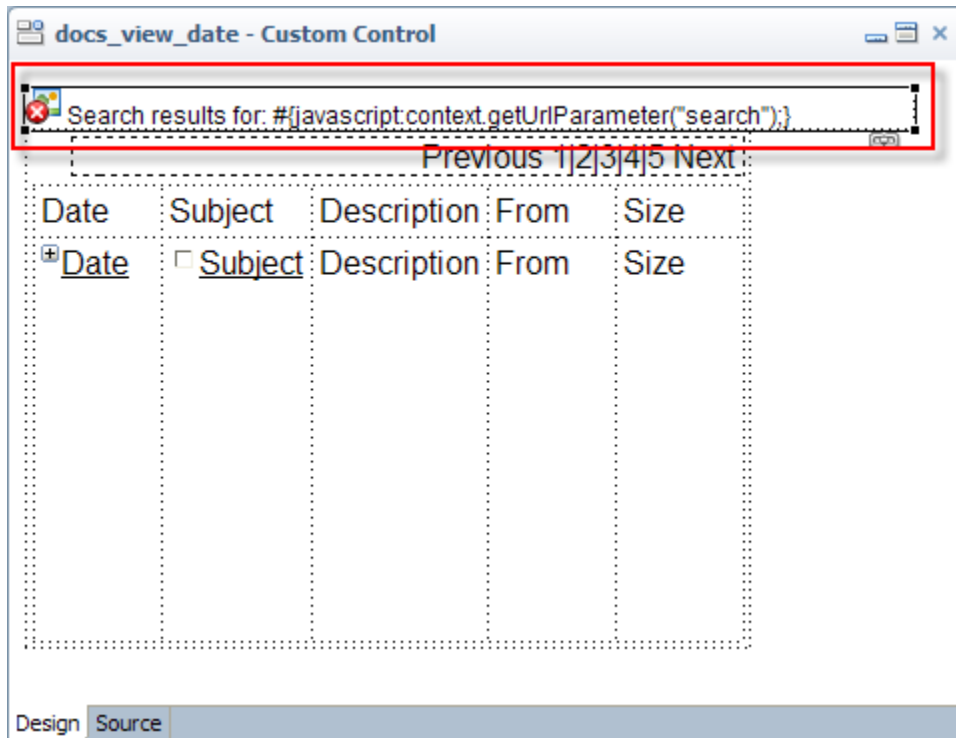
5. Select “Subject” column. Under “Column Display Properties”, check “Show values in columns as links” and “Checkbox” options. Select “Document Mode” as “Read Only”



6. From outline palette, select **viewPanel1** and enter **“/documents.xsp?content=docs_form”** as **“pageName”** property. This will use **“documents”** XPage to display the document when user clicks on a link within this view. It also adds a **“content”** query string parameter when a link is clicked. Within documents.xsp, custom control in the main content area is computed based on the query string parameter. In this case when a link is clicked to open a document, it will display **“docs_form”** custom control in the main content area of the XPage.



7. Drag and drop a **“global_search_display”** custom control to the top-left (before the view control).



8. Click on **"Source"** tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

  <xc:global_search_display></xc:global_search_display>
  <xp:viewPanel rows="30" id="viewPanel1"
    pageName="/documents.xsp?content=docs_form">
    <xp:this.facets>
      <xp:pager partialRefresh="true" layout="Previous Group
Next"
        xp:key="headerPager" id="pager1">
      </xp:pager>
    </xp:this.facets>
    <xp:this.data>
      <xp:dominoView var="documentsByDate"
        viewName="Documents\By Date">
      </xp:dominoView>
    </xp:this.data>
    <xp:viewColumn columnName="Date" id="viewColumn1"
      displayAs="link" openDocAsReadonly="true">
      <xp:this.converter>
        <xp:convertDateTime type="date"></xp:convertDateTime>
      </xp:this.converter>
      <xp:viewColumnHeader value="Date" id="viewColumnHeader1">
      </xp:viewColumnHeader>
```



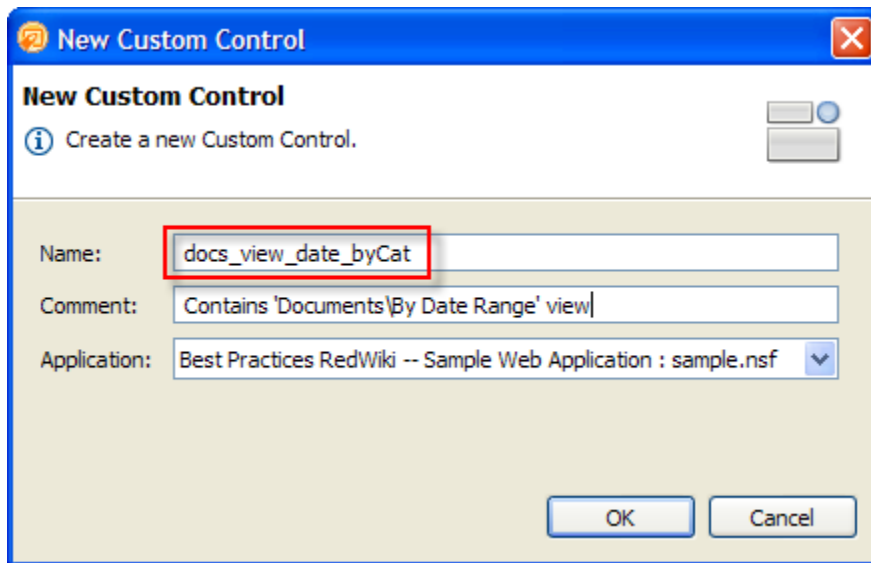
```

</xp:viewColumn>
<xp:viewColumn columnName="Subject" id="viewColumn2"
  showCheckbox="true" displayAs="link"
openDocAsReadonly="true">
  <xp:viewColumnHeader value="Subject"
    id="viewColumnHeader2">
  </xp:viewColumnHeader>
</xp:viewColumn>
<xp:viewColumn columnName="Description" id="viewColumn3">
  <xp:viewColumnHeader value="Description"
    id="viewColumnHeader3">
  </xp:viewColumnHeader>
</xp:viewColumn>
<xp:viewColumn columnName="From" id="viewColumn4">
  <xp:viewColumnHeader value="From"
id="viewColumnHeader4"></xp:viewColumnHeader>
</xp:viewColumn>
<xp:viewColumn columnName="Size" id="viewColumn5">
  <xp:viewColumnHeader value="Size"
id="viewColumnHeader5"></xp:viewColumnHeader>
</xp:viewColumn>
</xp:viewPanel>
</xp:view>

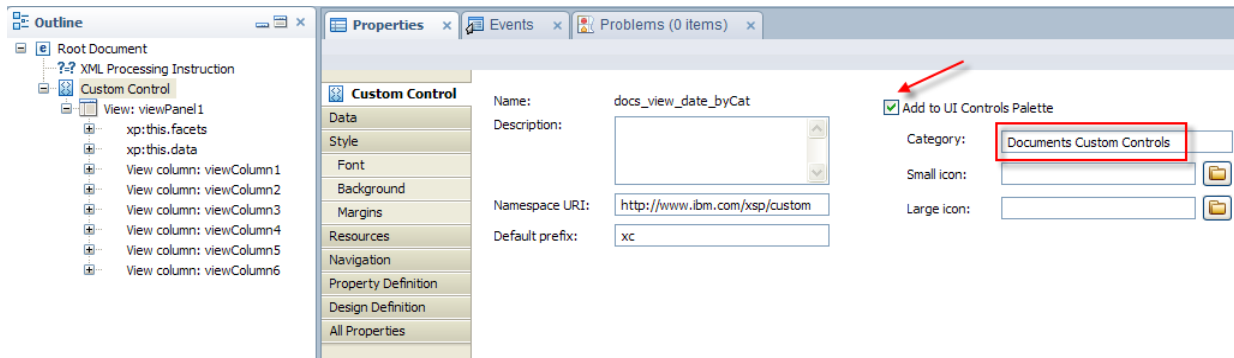
```

Step 8.5.4: Creating custom control for “Document by Date Range” view

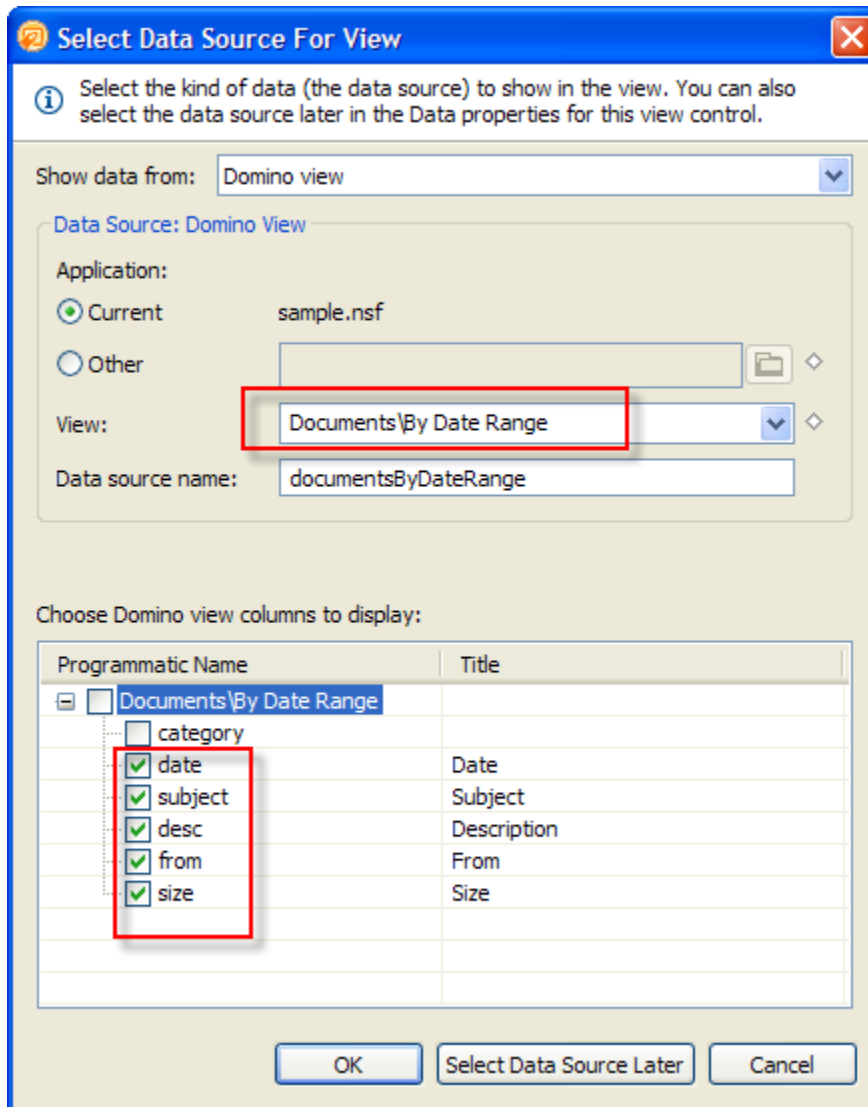
1. Create a new custom control “docs_view_date_byCat”.



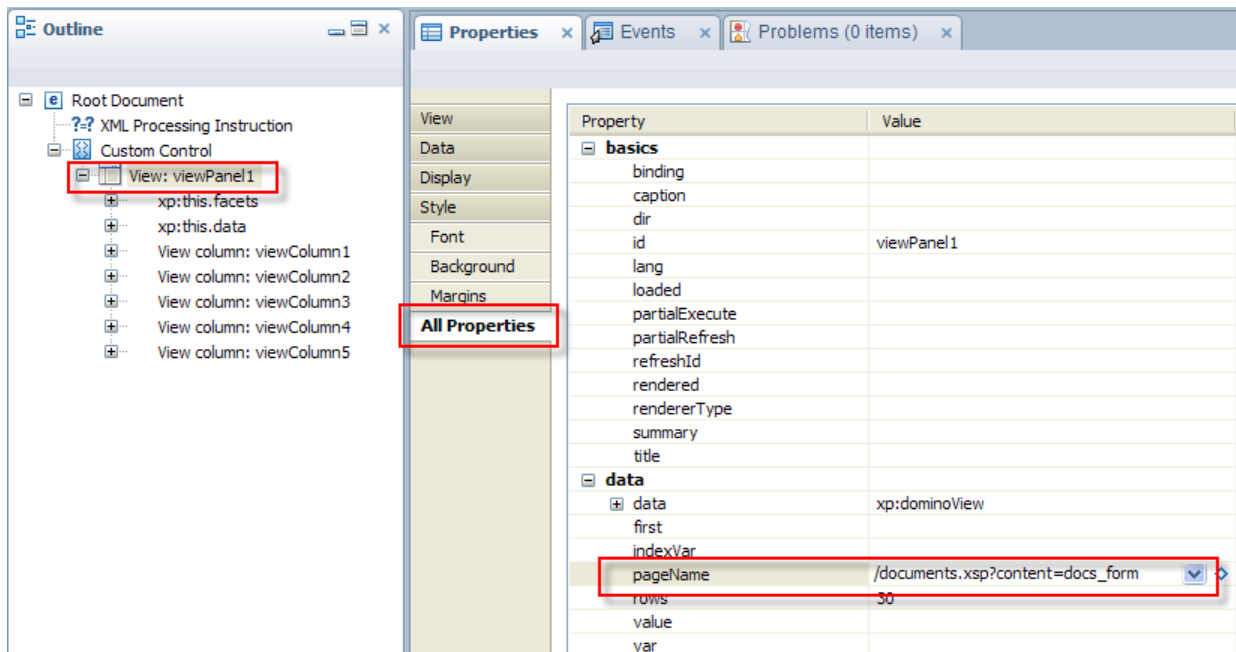
2. Under the Properties tab, make sure Custom Control is selected and check “**Add to UI Controls Palette**” and enter “**Documents Custom Controls**” as the category. This moves this custom control under the category “**Documents Custom Controls**”.



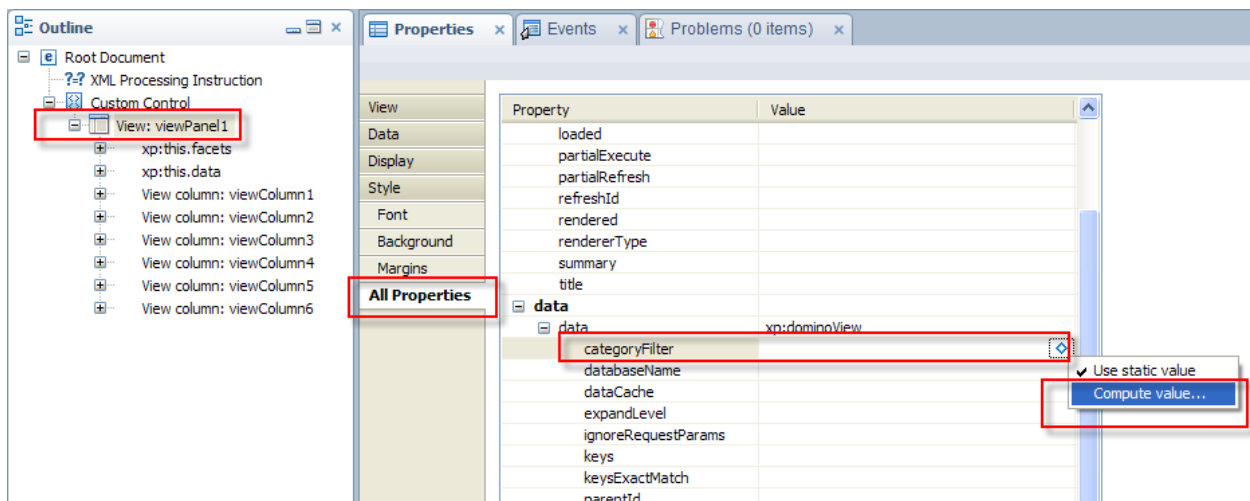
- From the **Container Controls**, drag and drop a “**View**” control to the editor. Select “**Documents\By Date Range**” view when prompted. Check all columns except the first one – category.

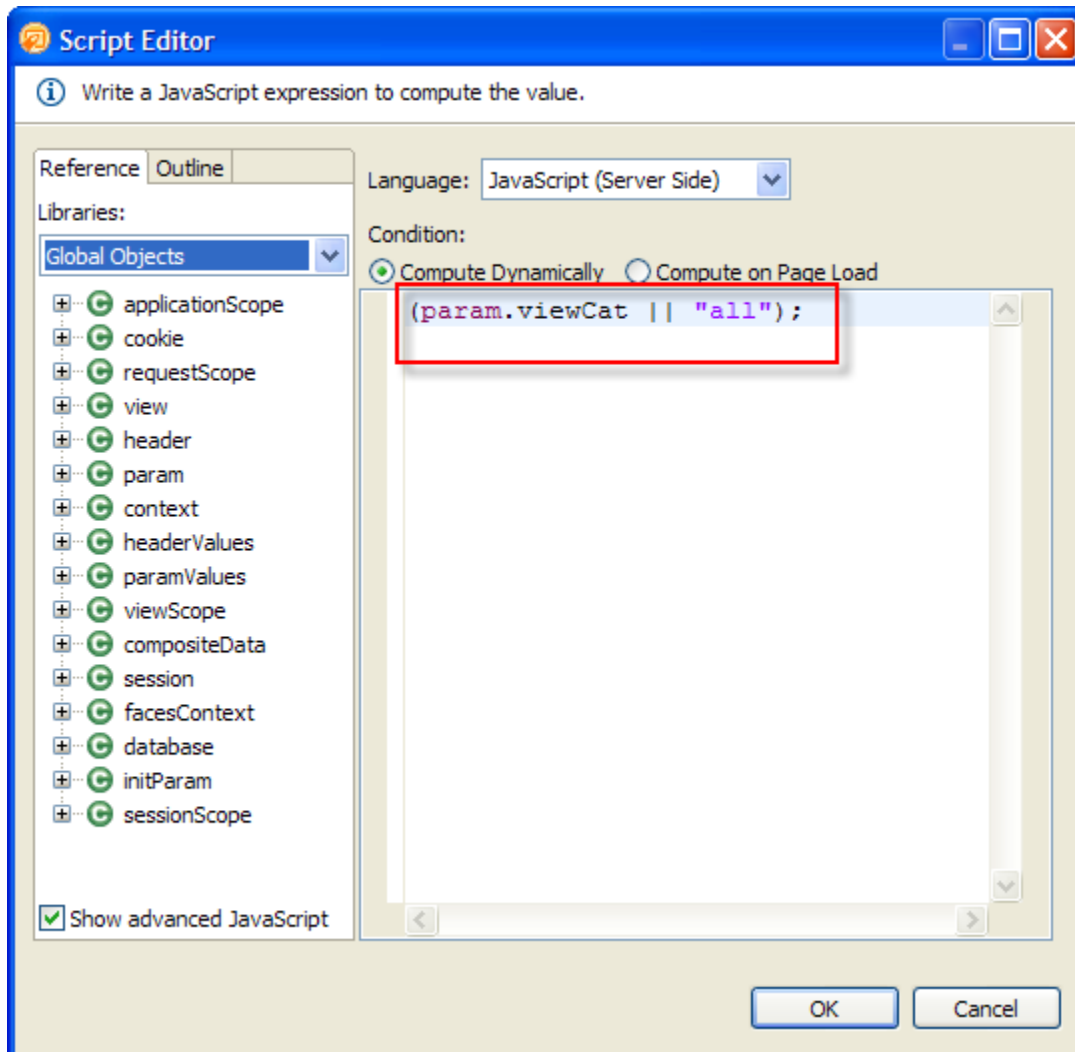


- From outline palette, select **viewPanel1** and enter **“/documents.xsp?content=docs_form”** as **“pageName”** property. This will use **“documents”** XPage to display the document when user clicks on a link within this view. It also adds a **“content”** query string parameter when a link is clicked. Within documents.xsp, custom control in the main content area is computed based on the query string parameter. In this case when a link is clicked to open a document, it will display **“docs_form”** custom control in the main content area of the XPage.

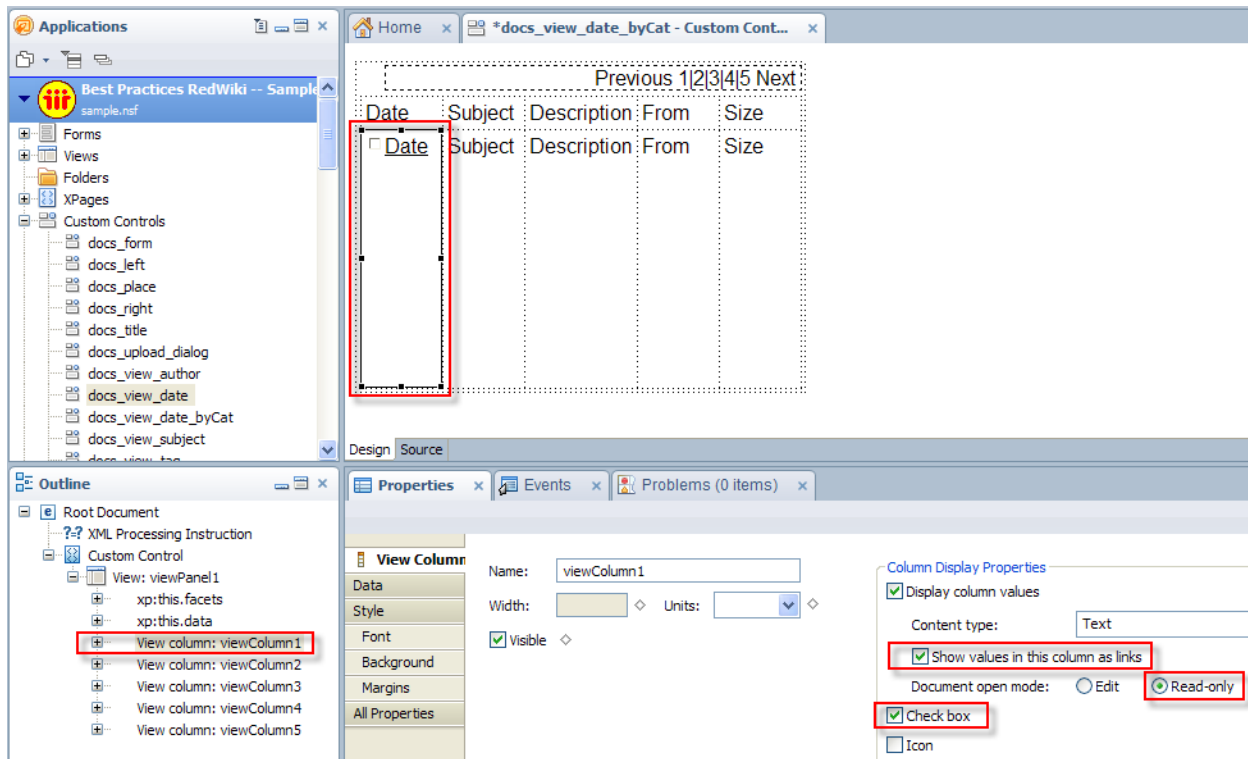


- From outline palette, select **viewPanel1**. Select **“All Properties”** and expand data section. Select **“computed value..”** for **“categoryFilter”** and enter **(param.viewCat || “all”)** in the JavaScript pop-up editor. This will assign limit the view content to single category based on **“viewCat”** query string parameter.

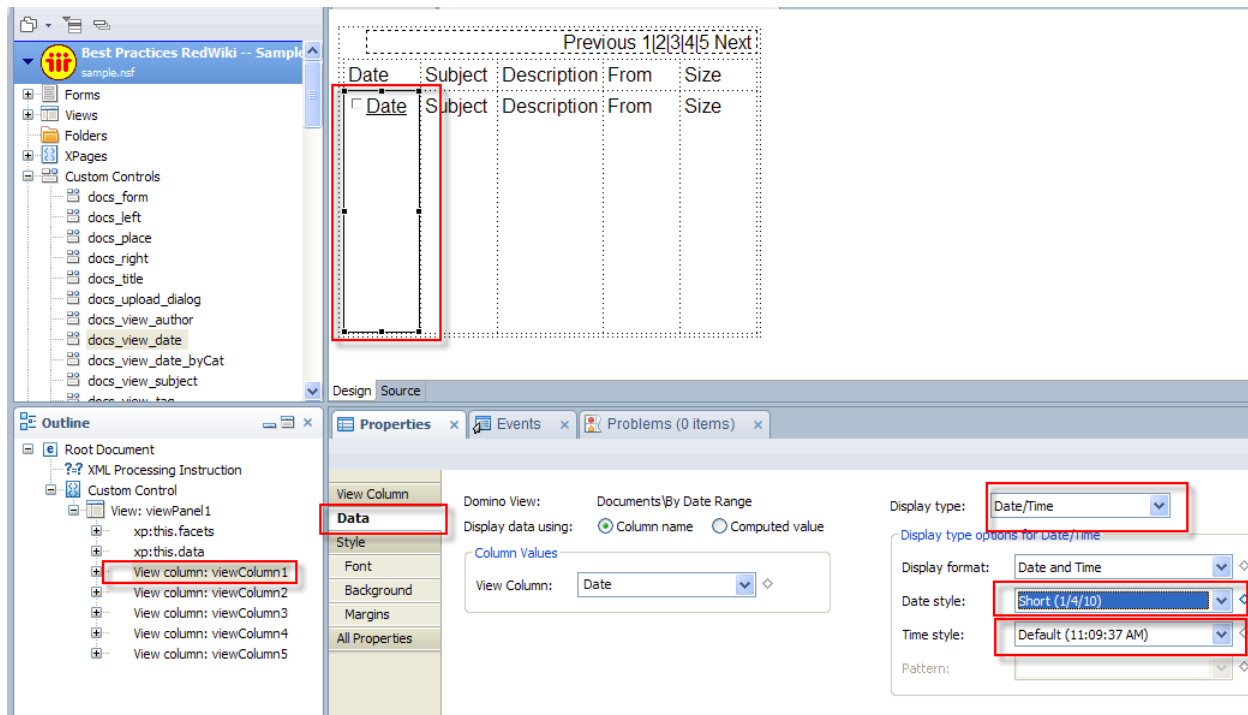




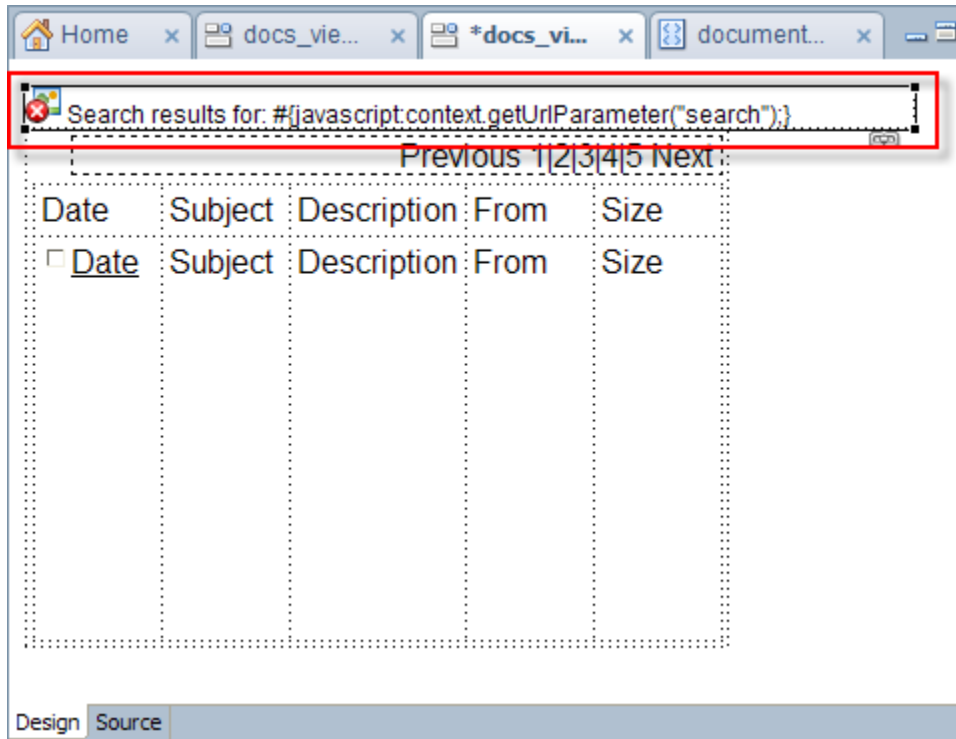
6. Select **"Date"** column. Under **"Column Display Properties"**, check **"Show values in columns as links"** and **"Checkbox"** options. Select **"Document Mode"** as **"Read Only"**



7. Select **"Date"** column, click on **"Date"** tab, under **"display type options for date/time"**, select **"Display Format"** as **"Date and Time"** and **"Date style"** as **"short"** and **"Time style"** as **"default"**.



8. Drag and drop a “**global_search_display**” custom control to the top-left (before the view control).



9. Click on “**Source**” tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core"
  xmlns:xc="http://www.ibm.com/xsp/custom">
  <xc:global_search_display></xc:global_search_display>
  <xp:viewPanel rows="30" id="viewPanel1"
    pageName="/documents.xsp?content=docs_form">
    <xp:this.facets>
      <xp:pager partialRefresh="true" layout="Previous Group
Next"
        xp:key="headerPager" id="pager1">
      </xp:pager>
    </xp:this.facets>
    <xp:this.data>
      <xp:dominoView var="documentsByDateRange"
        viewName="Documents\By Date Range">

      <xp:this.categoryFilter><![CDATA[#(javascript:(param.viewCat ||
"all"))]]></xp:this.categoryFilter>
      </xp:dominoView>
    </xp:this.data>
    <xp:viewColumn columnName="Date" id="viewColumn1"
      displayAs="link" openDocAsReadOnly="true"
showCheckbox="true">
```

```

        <xp:this.converter>
            <xp:convertDateTime type="both"
dateStyle="short"></xp:convertDateTime>
        </xp:this.converter>
        <xp:viewColumnHeader value="Date"
id="viewColumnHeader1"></xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="Subject" id="viewColumn2">
        <xp:viewColumnHeader value="Subject"
            id="viewColumnHeader2">
        </xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="Description" id="viewColumn3">
        <xp:viewColumnHeader value="Description"
            id="viewColumnHeader3">
        </xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="From" id="viewColumn4">
        <xp:viewColumnHeader value="From"
id="viewColumnHeader4"></xp:viewColumnHeader>
    </xp:viewColumn>
    <xp:viewColumn columnName="Size" id="viewColumn5">
        <xp:viewColumnHeader value="Size"
id="viewColumnHeader5"></xp:viewColumnHeader>
    </xp:viewColumn>
</xp:viewPanel>
</xp:view>

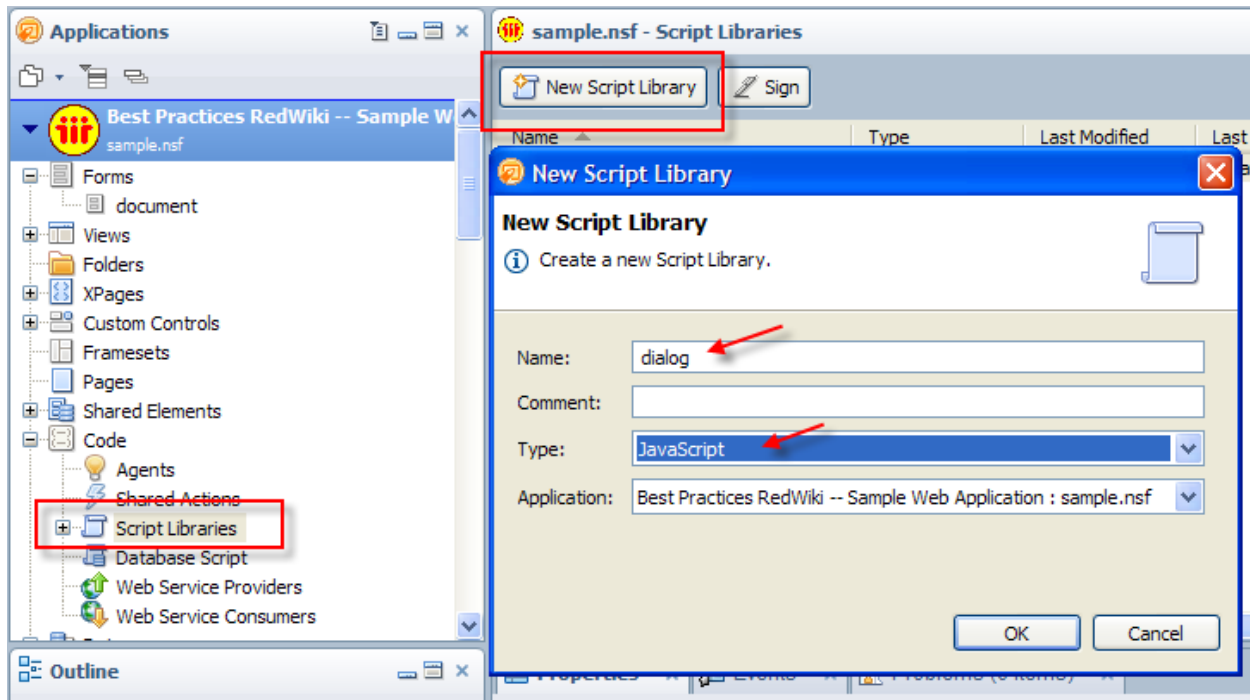
```

Step 8.6: Creating a custom controls for document forms

In the last section, we created custom controls for views. In this section we are going to create custom controls for document forms – one for upload and one viewing/editing the document. Upload form is separate from view/edit form to show an example of dojo dialog (used for upload form).

8.6.1: Creating custom control for document upload

1. Before we build the upload custom control, we need to create a JavaScript library for function called from the upload form. From the application palette, expand the code node and click on Script Libraries. Click **"New Script Library"**, name it **"dialog"** (without .js) and select type as **"JavaScript"**.



2. Enter the following code. This code basically creates a dojo dialog based on the id of <div> element passed to it. It also appends the dialog to the underlying form – so that fields can be saved when the form is submitted. We are going to call this function from the custom control to create a dojo dialog for uploading. Refer to dojo document for additional information about specifics of dojo dialog and the code below.

```
function dialog_create(id, title1) {

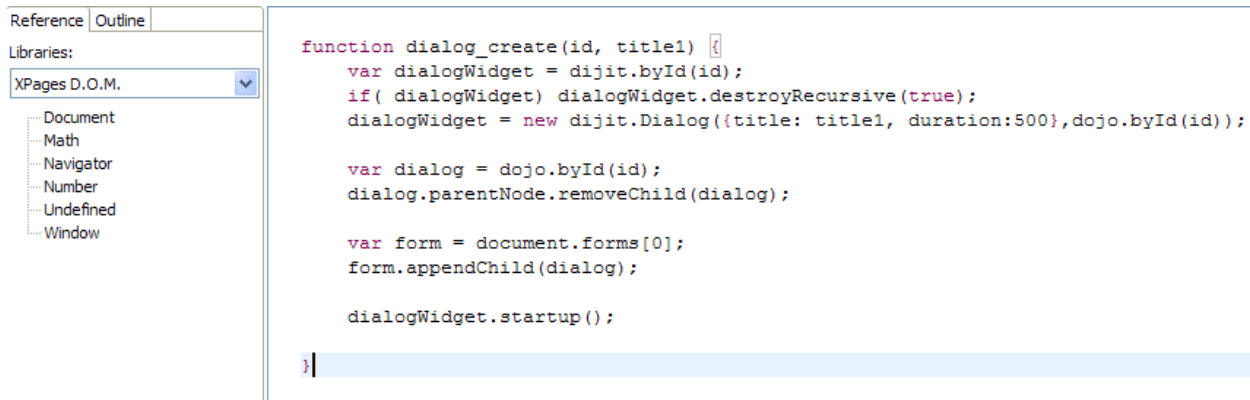
    var dialogWidget = dijit.byId(id);
    if( dialogWidget) dialogWidget.destroyRecursive(true);
    dialogWidget = new dijit.Dialog({title: title1,
duration:500},dojo.byId(id));

    var dialog = dojo.byId(id);
    dialog.parentNode.removeChild(dialog);

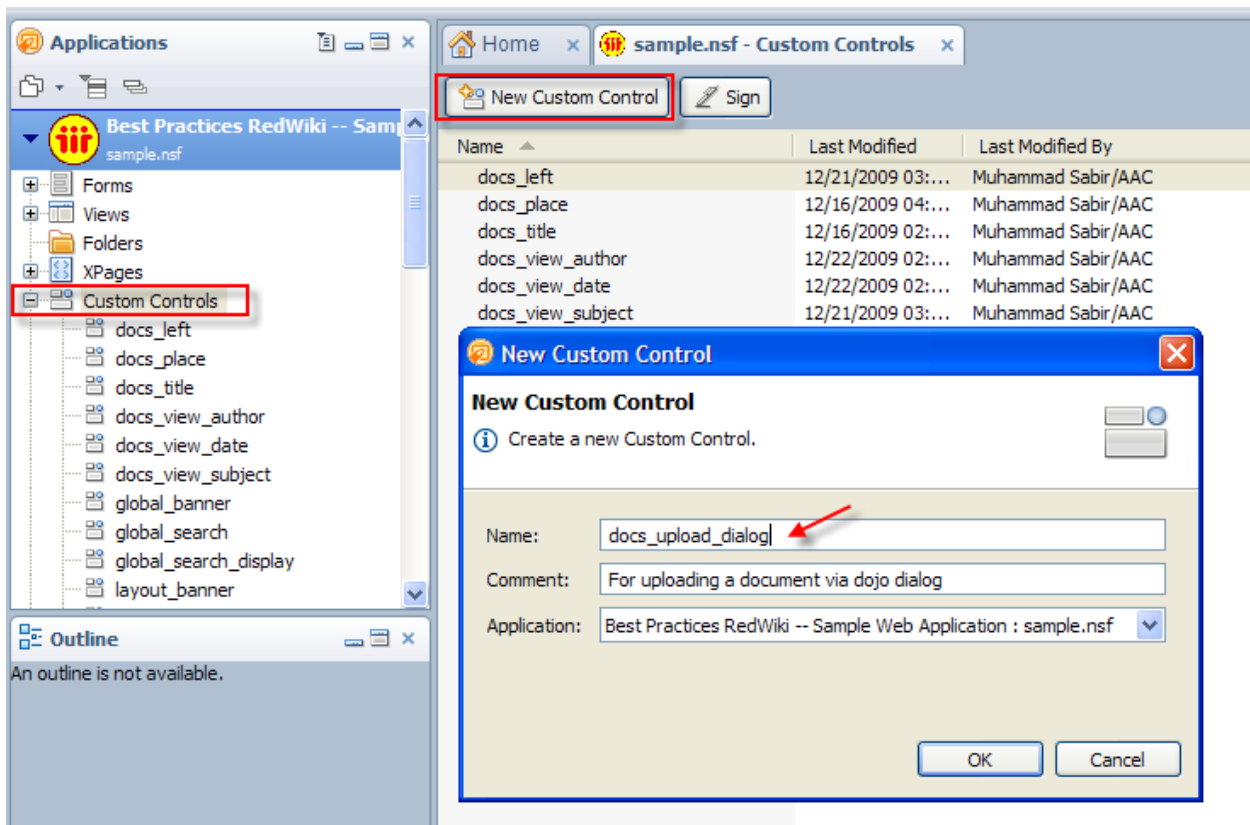
    var form = document.forms[0];
    form.appendChild(dialog);

    dialogWidget.startup();

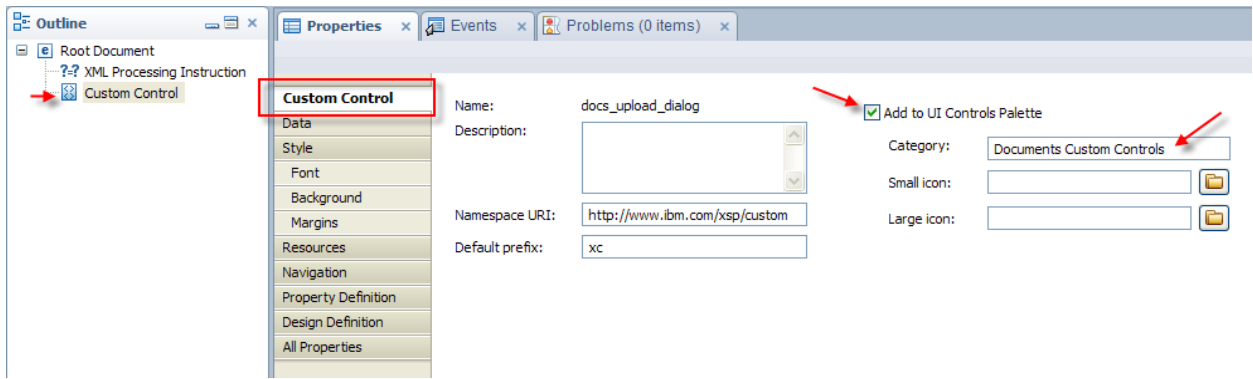
}
```

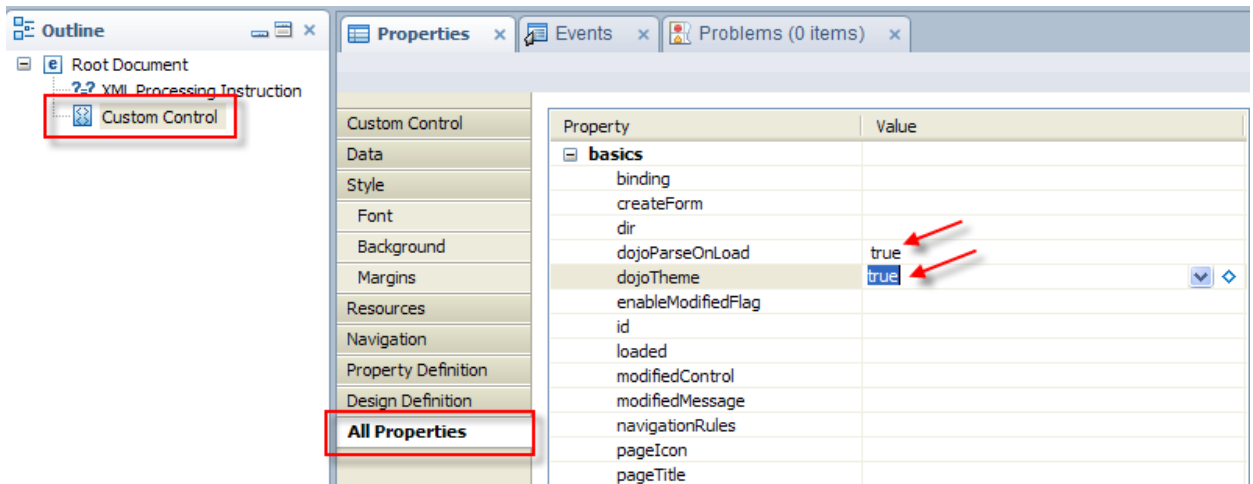
3. Create a new custom control “docs_view_date_byCat”.



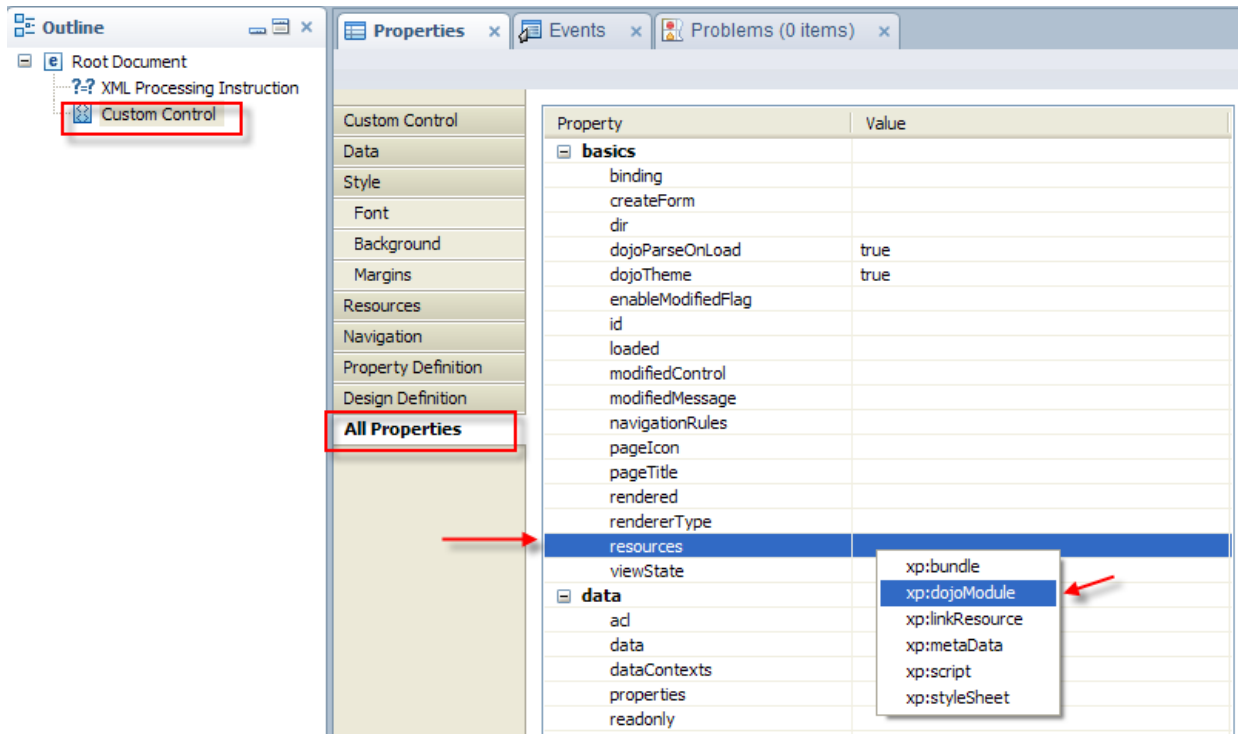
4. Under the Properties tab, make sure Custom Control is selected and check “Add to UI Controls Palette” and enter “Documents Custom Controls” as the category. This moves this custom control under the category “Documents Custom Controls”.



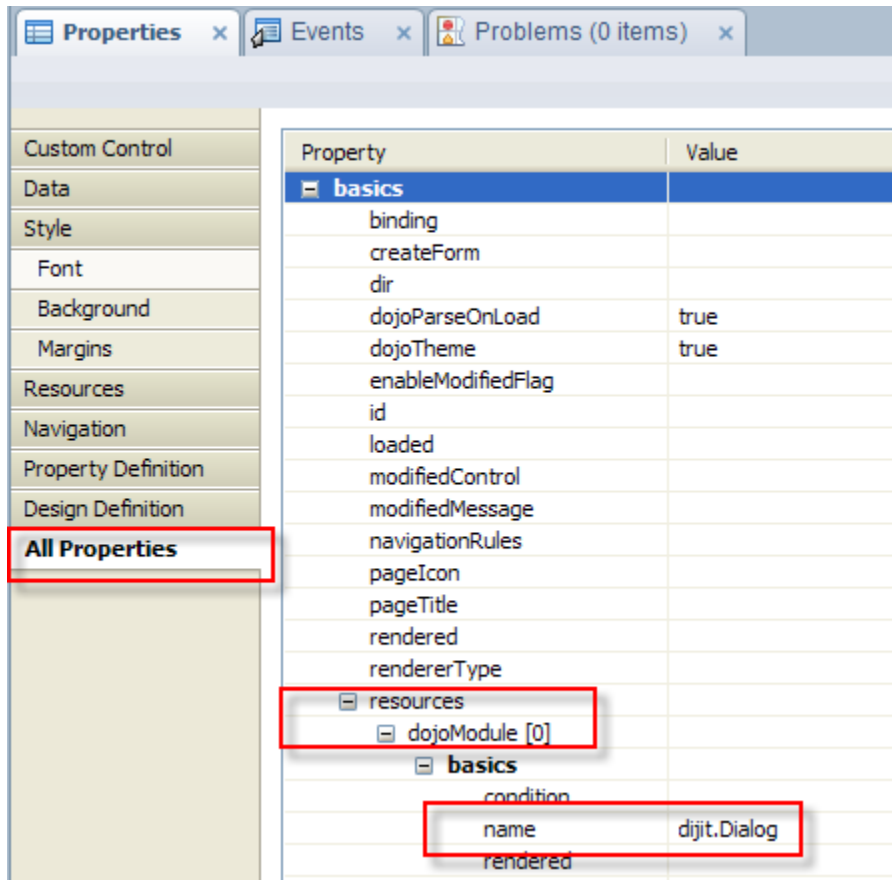
5. Select “**dojoParseOnLoad**” and “**dojoTheme**” to **true**.



6. From custom control properties, click on “All Properties” tab. Select “**dojoParseOnLoad**” and “**dojoTheme**” to **true**. This will trigger dojo parse on load and inject dojo related resources to this custom control.



- Expand the “resources” property and add a “dojoModule” named “dijit.Dialog”.



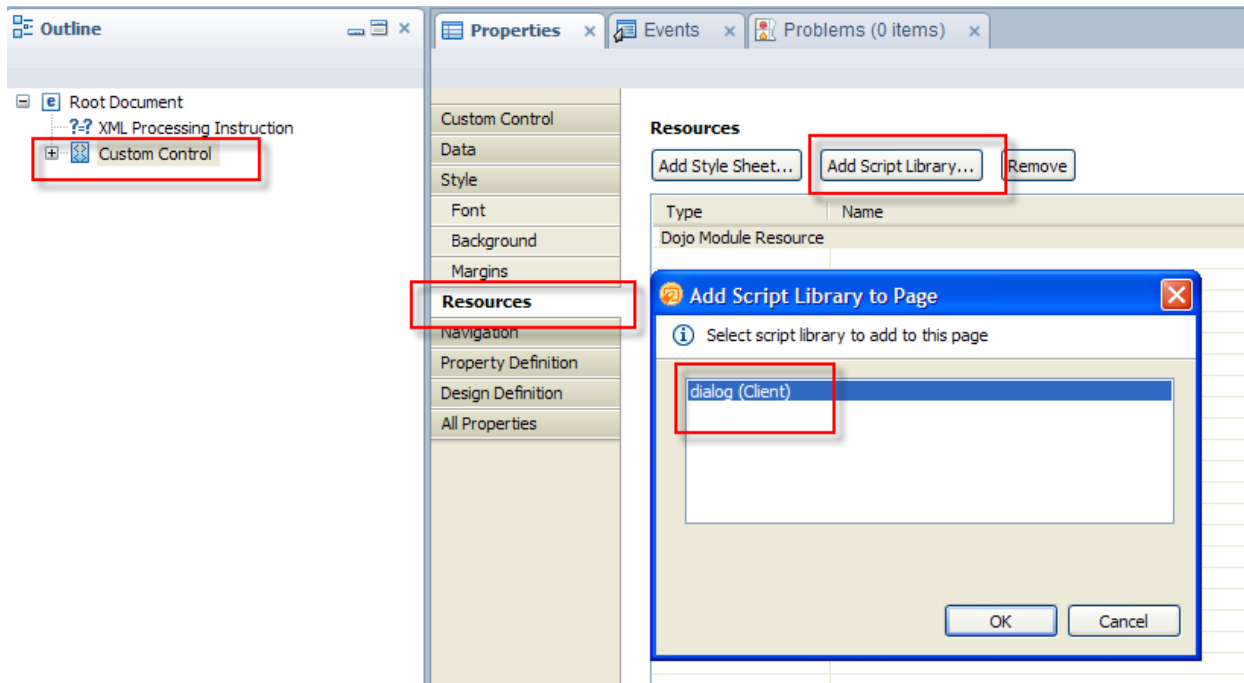
This is what the source code should look like:

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

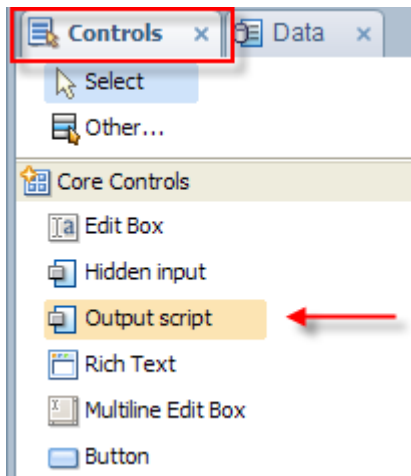
  <xp:this.resources>
    <xp:dojoModule name="dijit.Dialog"></xp:dojoModule>
  </xp:this.resources>

</xp:view>
```

8. From custom control properties, click on **Resources** tab and click **Add Script Library** and select "dialog (Client)" library. This will add the selected JavaScript library to this custom control.



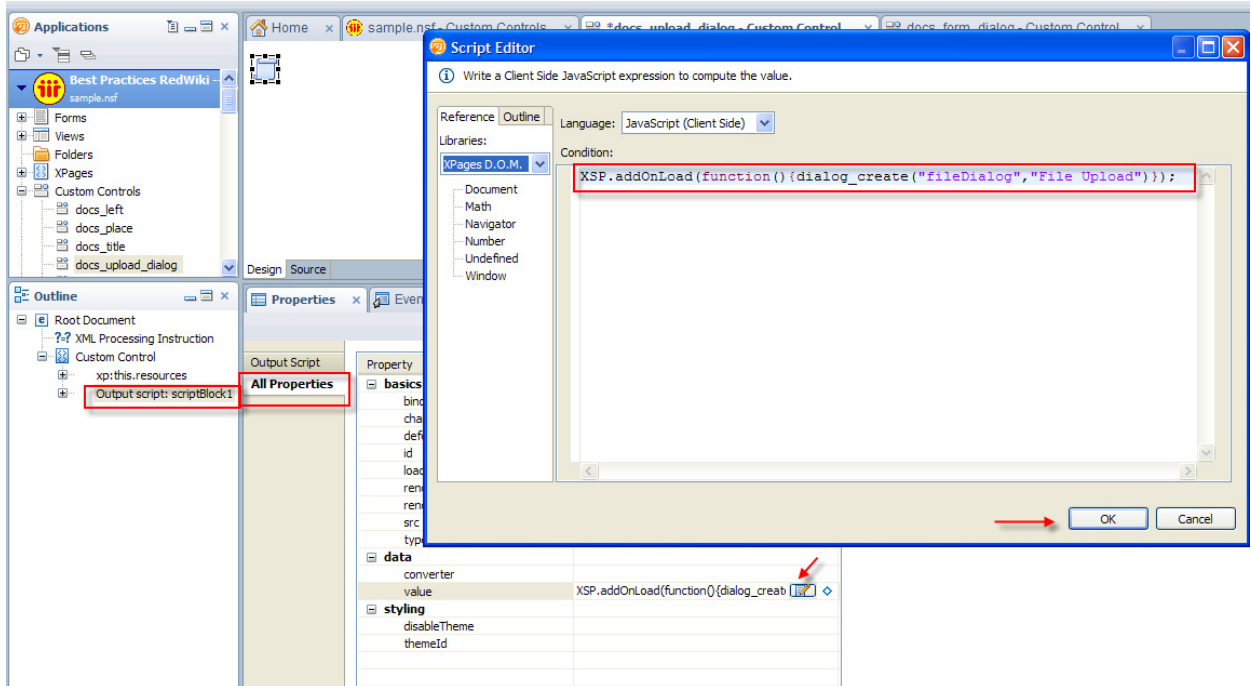
9. From Core Controls, drag and drop an “**Output script**” control.



10. Select output script control in the outline palette, and click on “**All Properties**” tab. Click on the icon in front of the value property and enter the following code in pop-up JavaScript editor:

```
XSP.addOnLoad(function(){dialog_create("fileDialog","File Upload")})
```

This will call “**dialog_create**” function when this custom control is loaded. This function creates the dojo dialog -- but it is hidden until the “**show**” function on the dojo dialog is called when “New Document” button is clicked.



This is what the source code should look like so far:

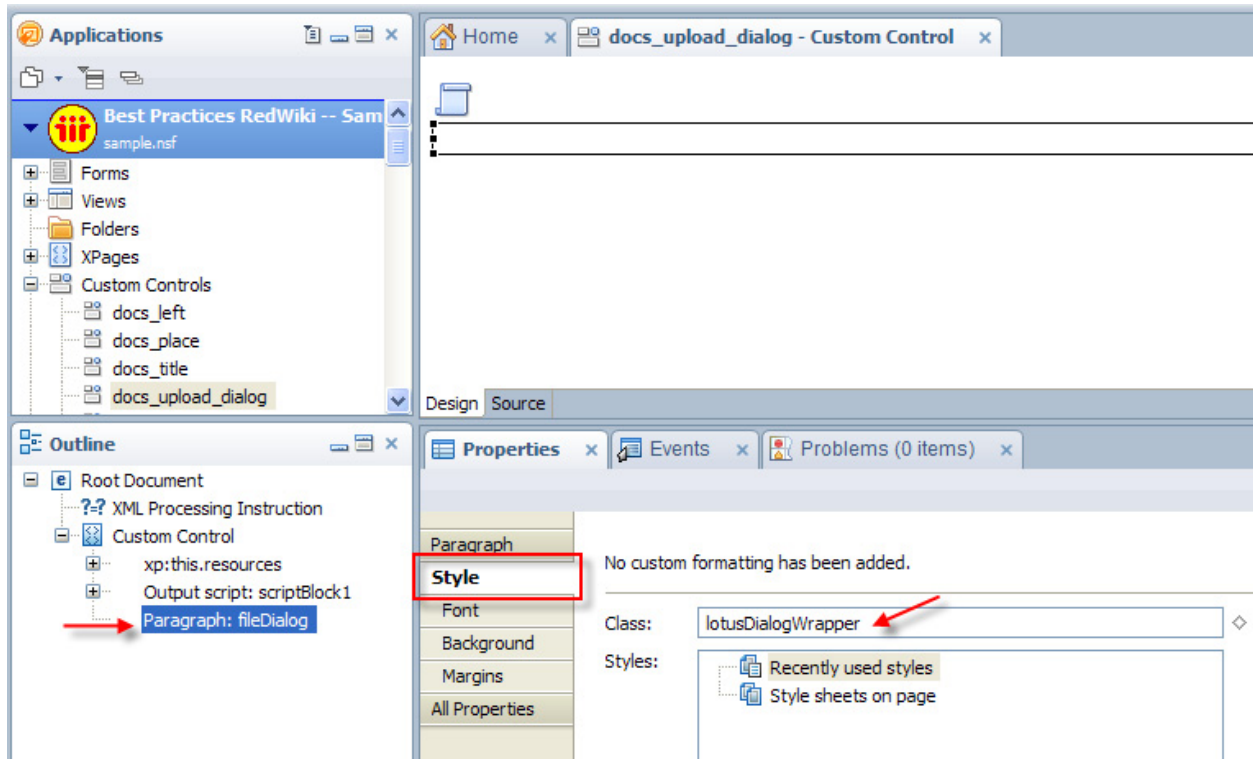
```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core" dojoParseOnLoad="true"
  dojoTheme="true">

  <xp:this.resources>
    <xp:dojoModule name="dijit.Dialog"></xp:dojoModule>
    <xp:script src="/dialog.js" clientSide="true"></xp:script>
  </xp:this.resources>

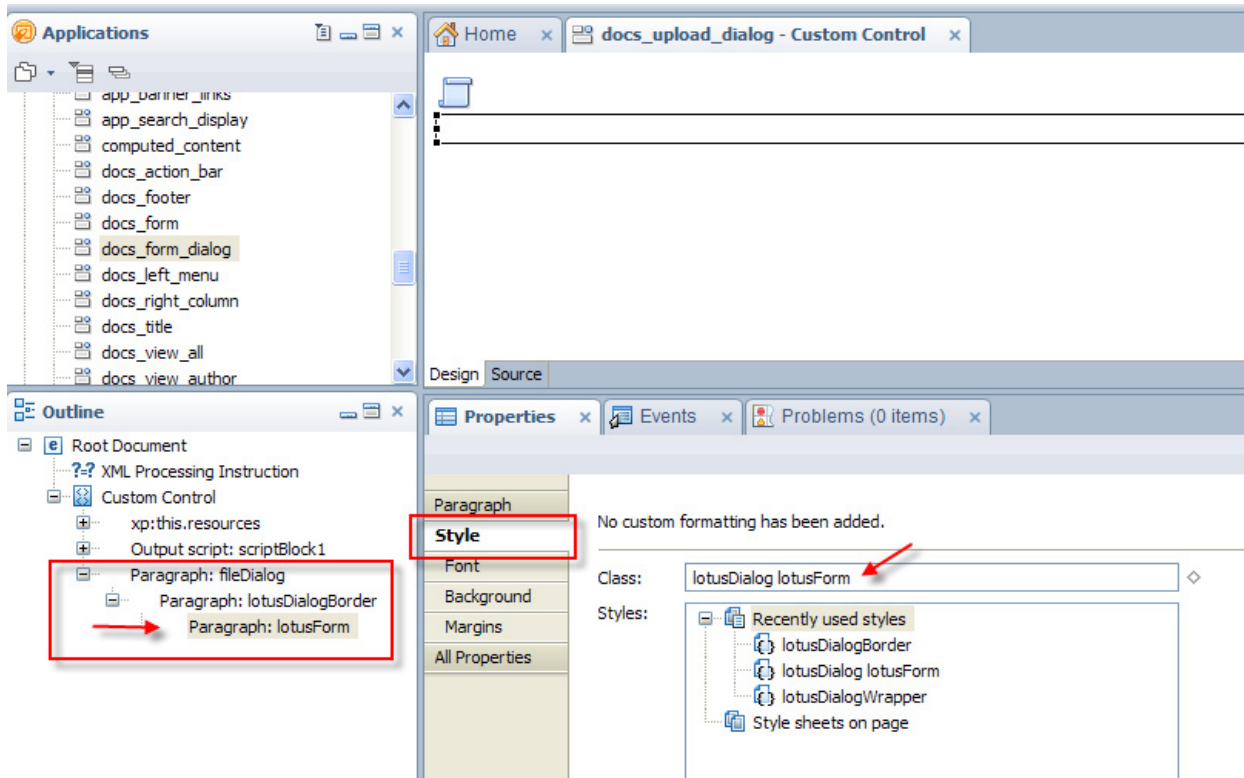
  <xp:scriptBlock id="scriptBlock1">
    <xp:this.value><![CDATA[XSP.addOnLoad(function(){dialog_create("fileDialog","File Upload");});]]></xp:this.value>
  </xp:scriptBlock>

</xp:view>
```

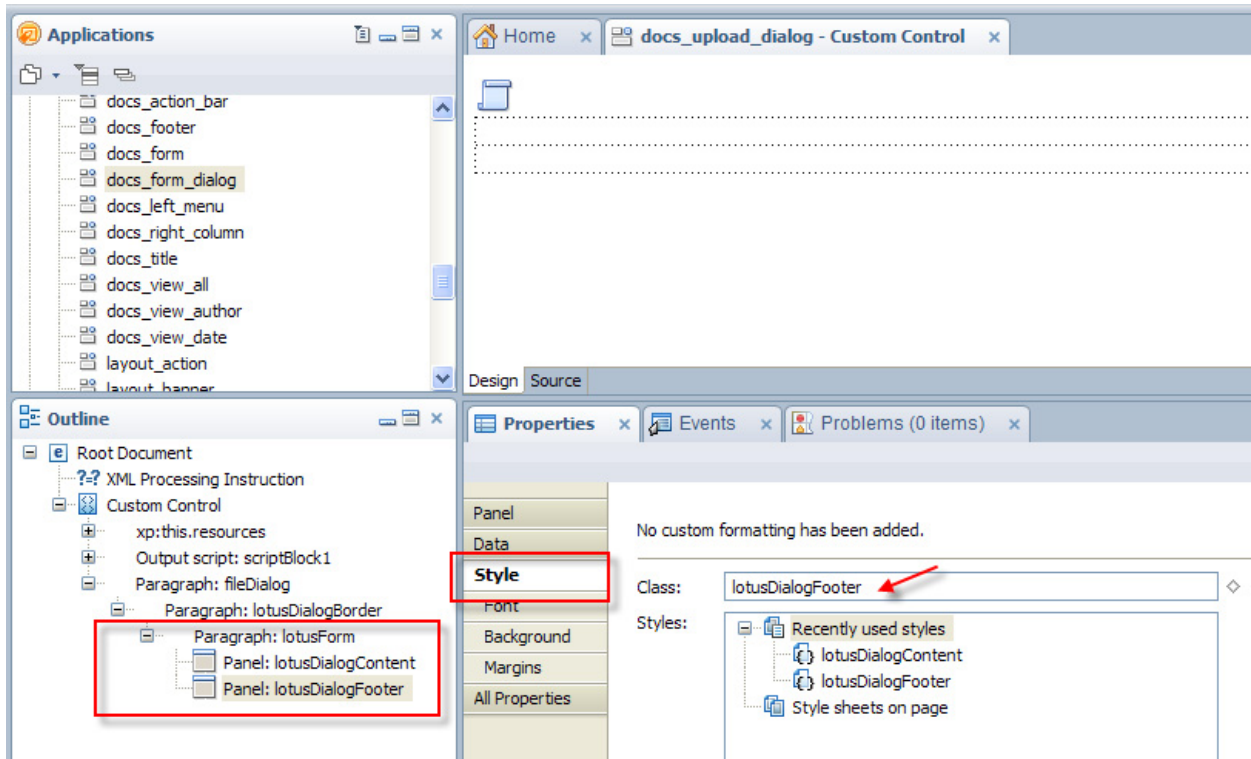
11. From the “User Added Controls”, drag and drop a “Paragraph” control. Name it as “fileDialog” and enter “lotusDialogWrapper” as its CSS style class.



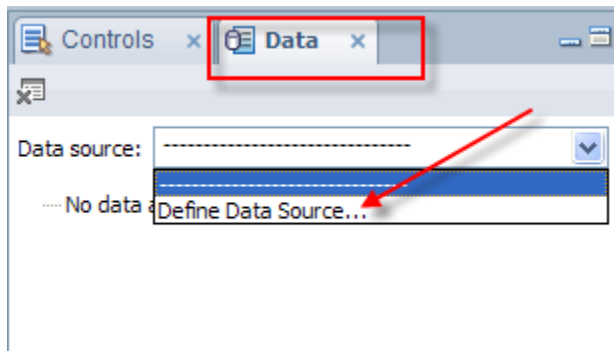
12. Drag a “**Paragraph**” control again and drop it over the last paragraph control. Enter “**lotusDialogBorder**” both as its name and CSS style class.
13. Drag a “**Paragraph**” control again and drop it over the last paragraph control. Enter “**lotusForm**” as its name and “**lotusDialog lotusForm**” as CSS style class. Now we should have three paragraph controls as shown in the screenshot below (note the sequence in the outline palette)



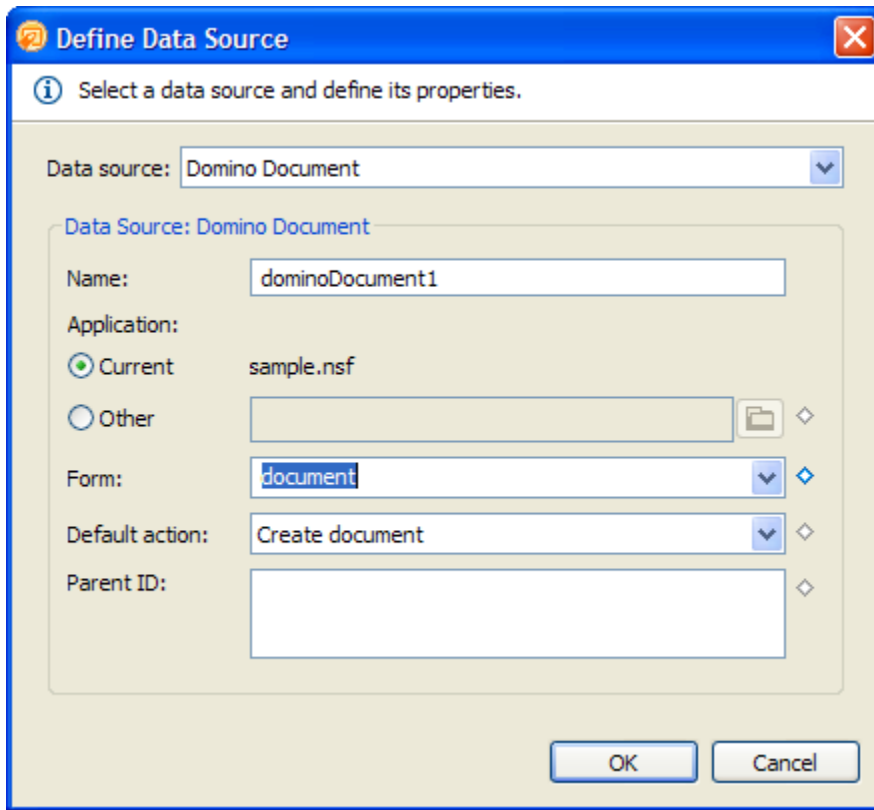
14. Drag and drop two Panel controls within the “**lotusDialogBorder**” control created in the last step. Enter “**lotusDialogContent**” both as name and CSS style class for the first panel. Enter “**lotusDialogFooter**” both as name and CSS style class for the 2nd panel.



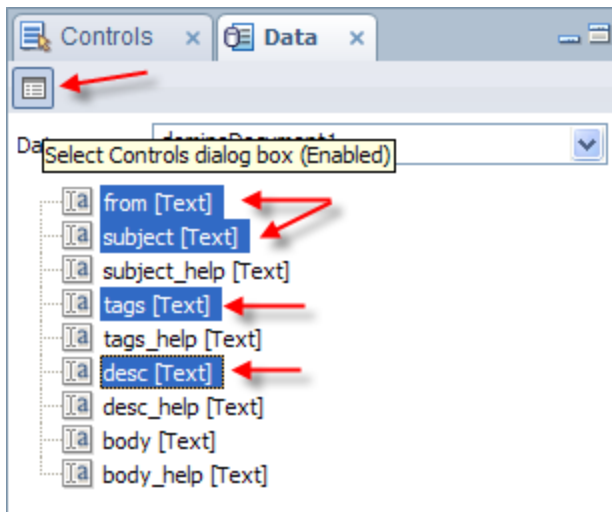
15. Click on the “**Data**” palette (next to the controls palette), and select “**Define Data Source...**” from the drop down.



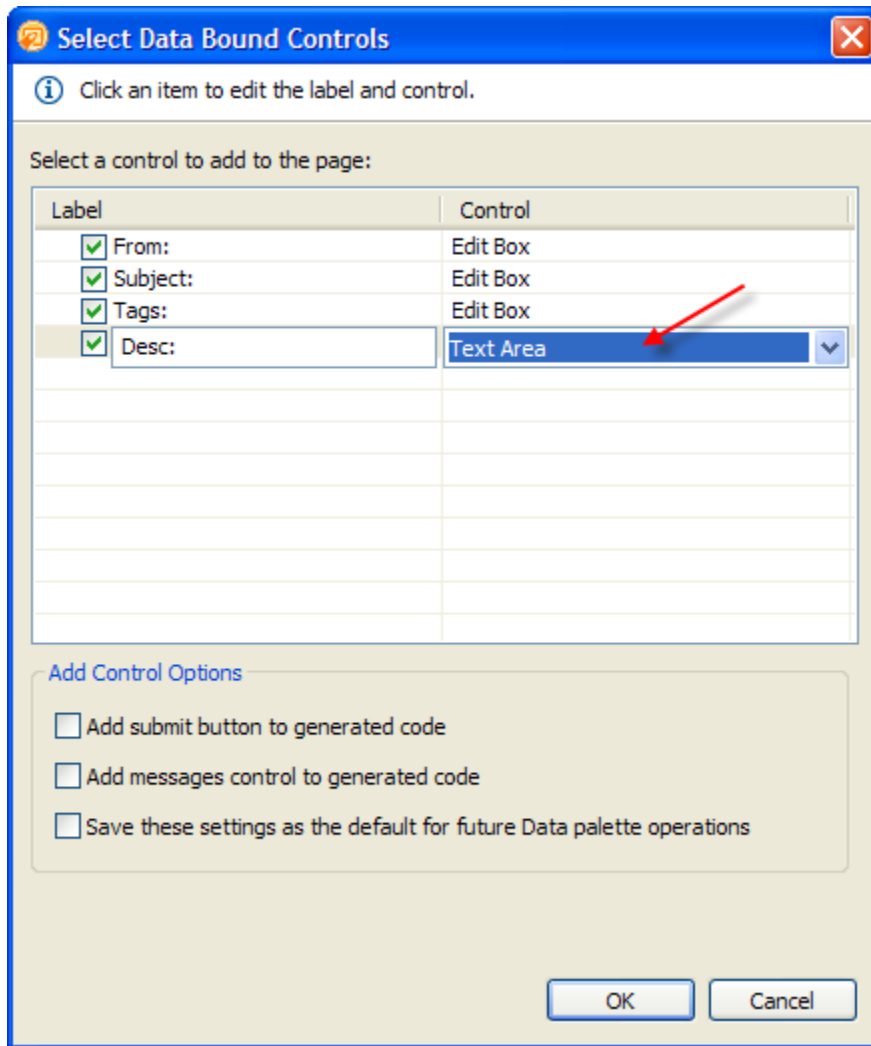
16. Select “**Domino Document**” as data source. Select “**document**” as form and “**Create document**” as default action.



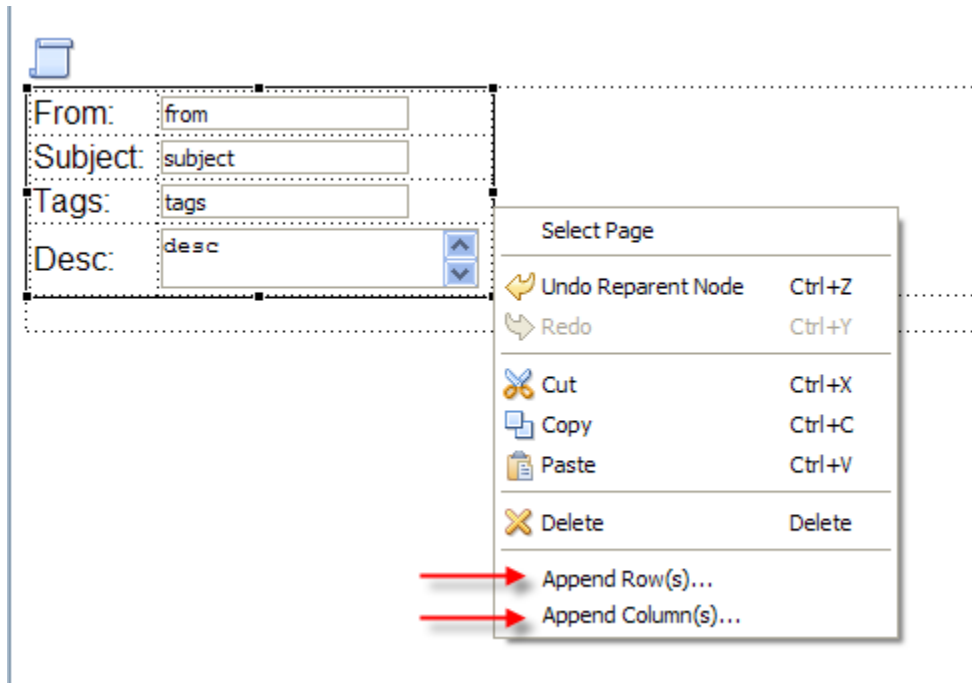
17. Make sure that “**Select Controls dial box**” is enabled (as shown in the screenshot). Select the following data fields: from, subject, tags, desc and drag; and drop them to “lotusDialogContent” panel.



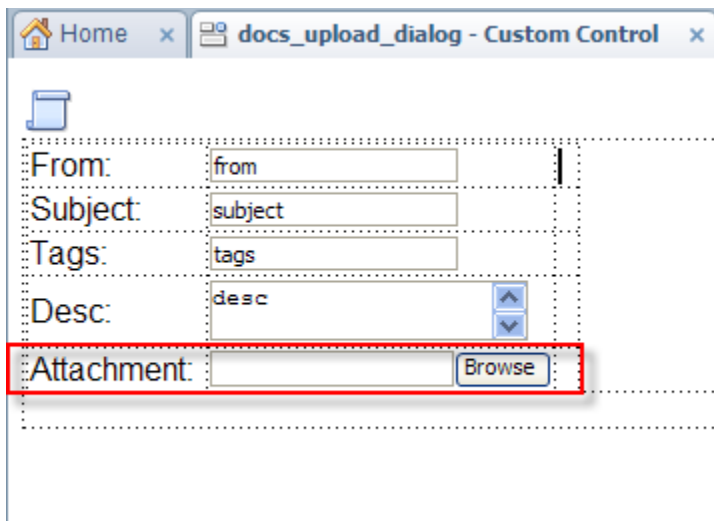
18. Keep the control type to “**Edit Box**” for all fields except “Desc” – change its control type to “**Text Area**”. Click OK. This will create a table with four rows with field labels and controls bound to the respective fields.



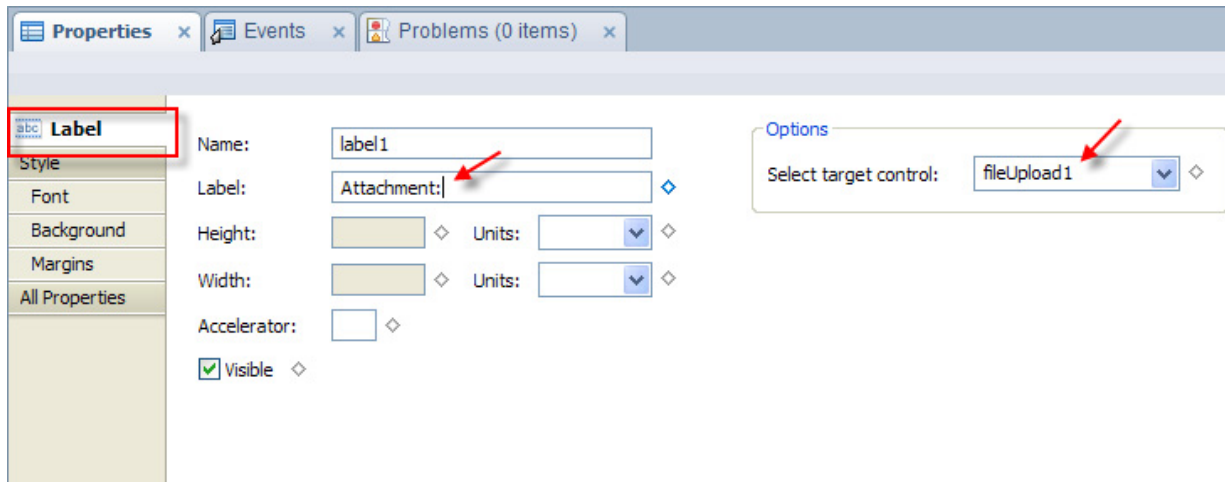
19. Right-click the newly created table and **append a column and a row** by selecting respective options.



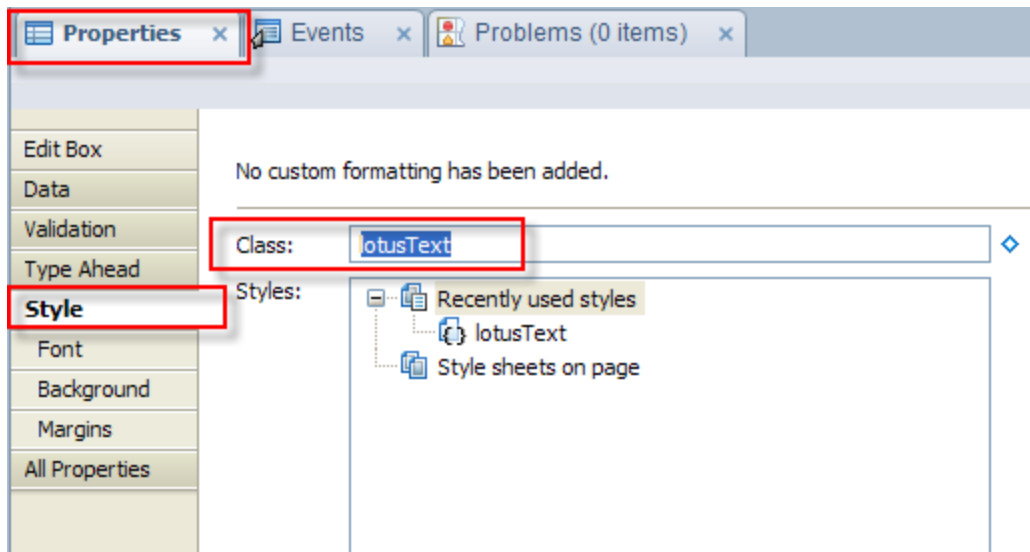
20. From the core controls, drag a **"File Download"** control and drop it over to 2nd column in the new row.



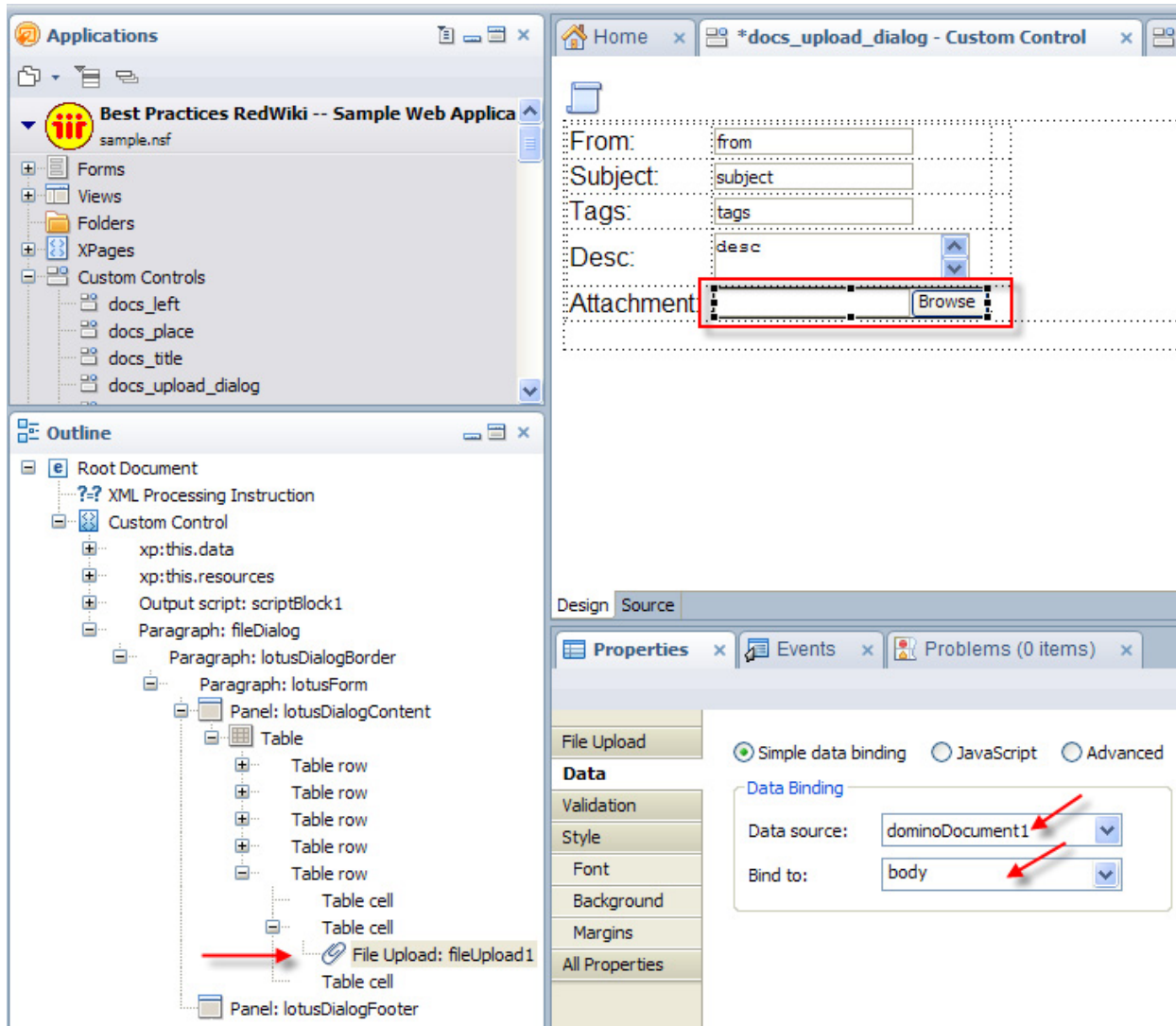
21. Drag a **"Label"** control and drop it to the first column. Enter **"Attachment:"** as label property and select **"fileUpload1"** as target control. This will associate the label with the download control created in last step.



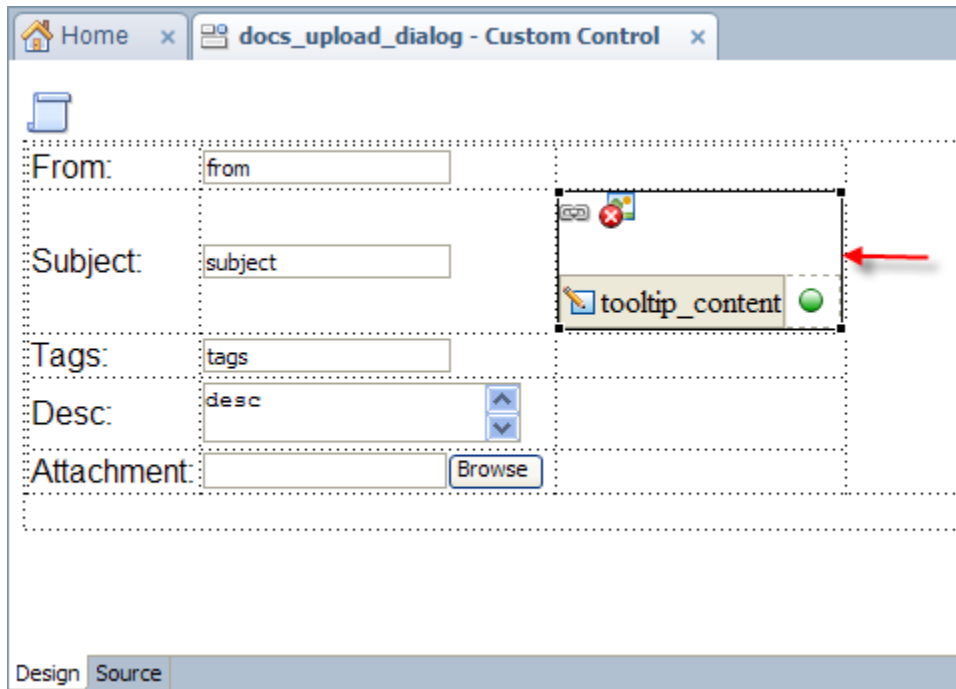
22. Select each one of the five input fields, and enter “**lotusText**” as CSS style class.



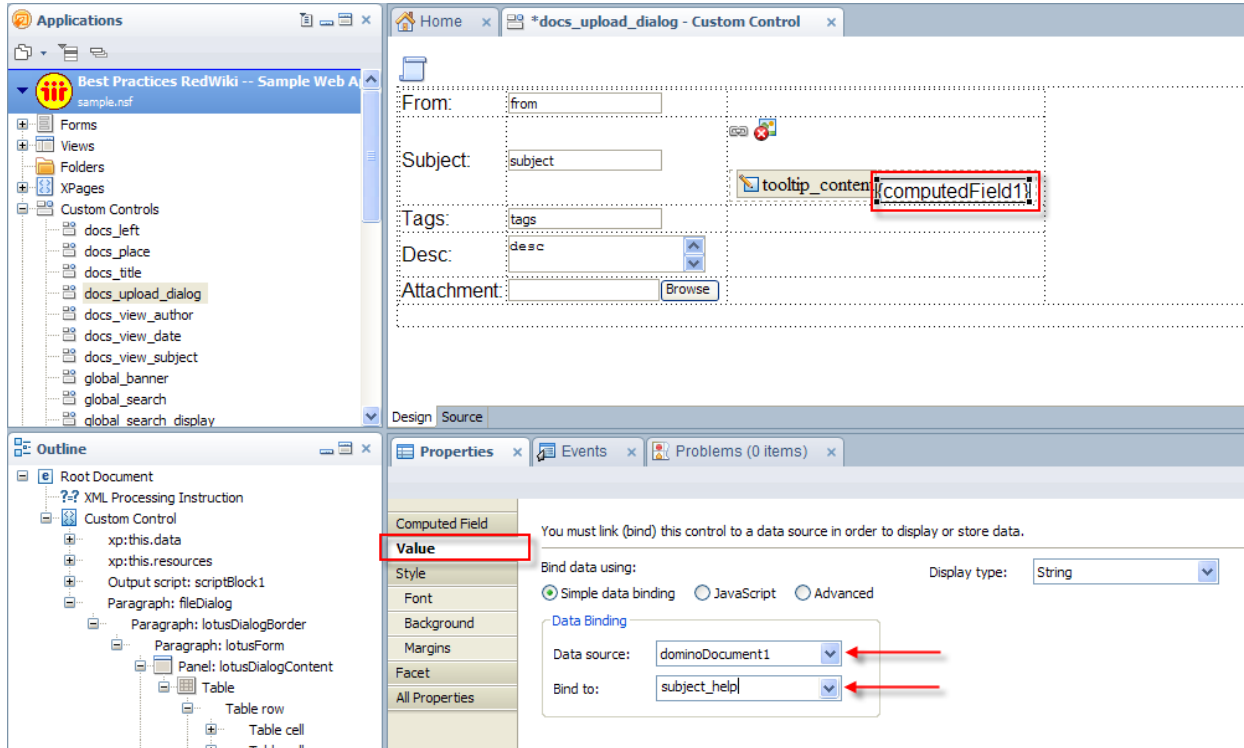
23. Select “**fileUpload1**” control in the outline palette. Select “**data**” tab in the properties palette. Select “**simple data binding**” option and bind this control to the “**body**” field of the document. Since “**body**” is rich text field, it can be used to save attachments.



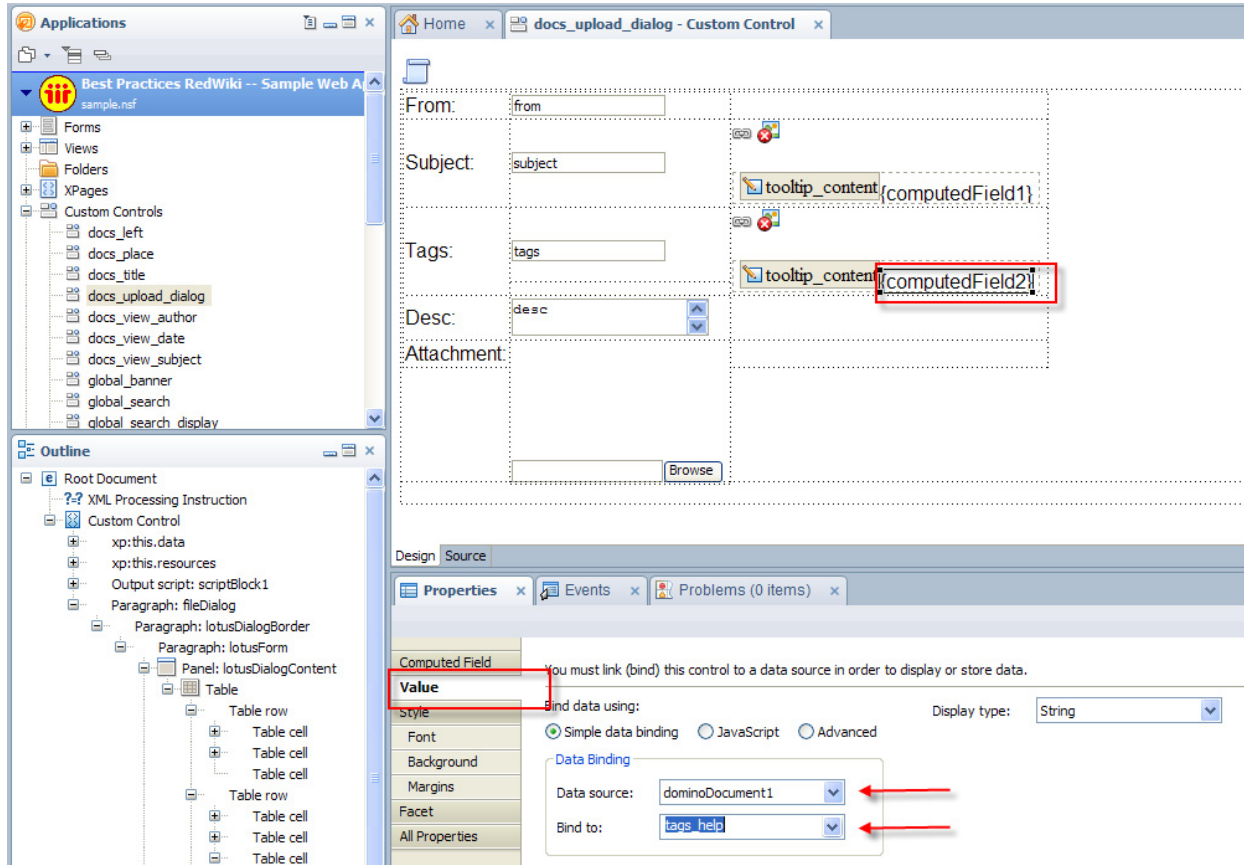
24. Drag the “**global_tooltip**” custom control and drop it to the 3rd column on 2nd row (subject line).



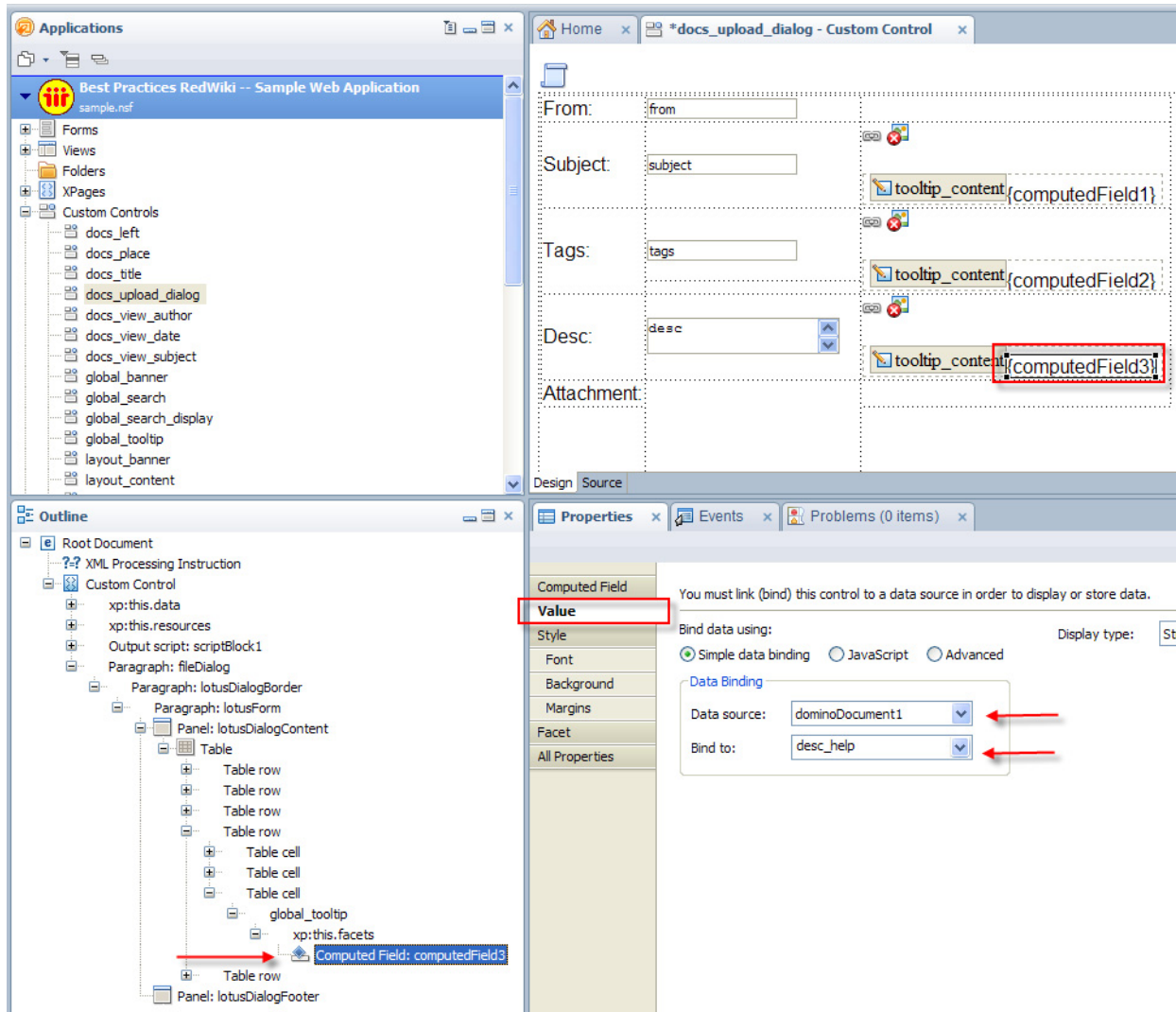
25. Drag a **“Computed Field”** control and drop it over to the dynamic area (indicated by green circle) within the **“global_tooltip”**. Bind the value property of the computed field with **“subject_help”** field of the **dominoDocument1** data source – as shown in the screenshot.



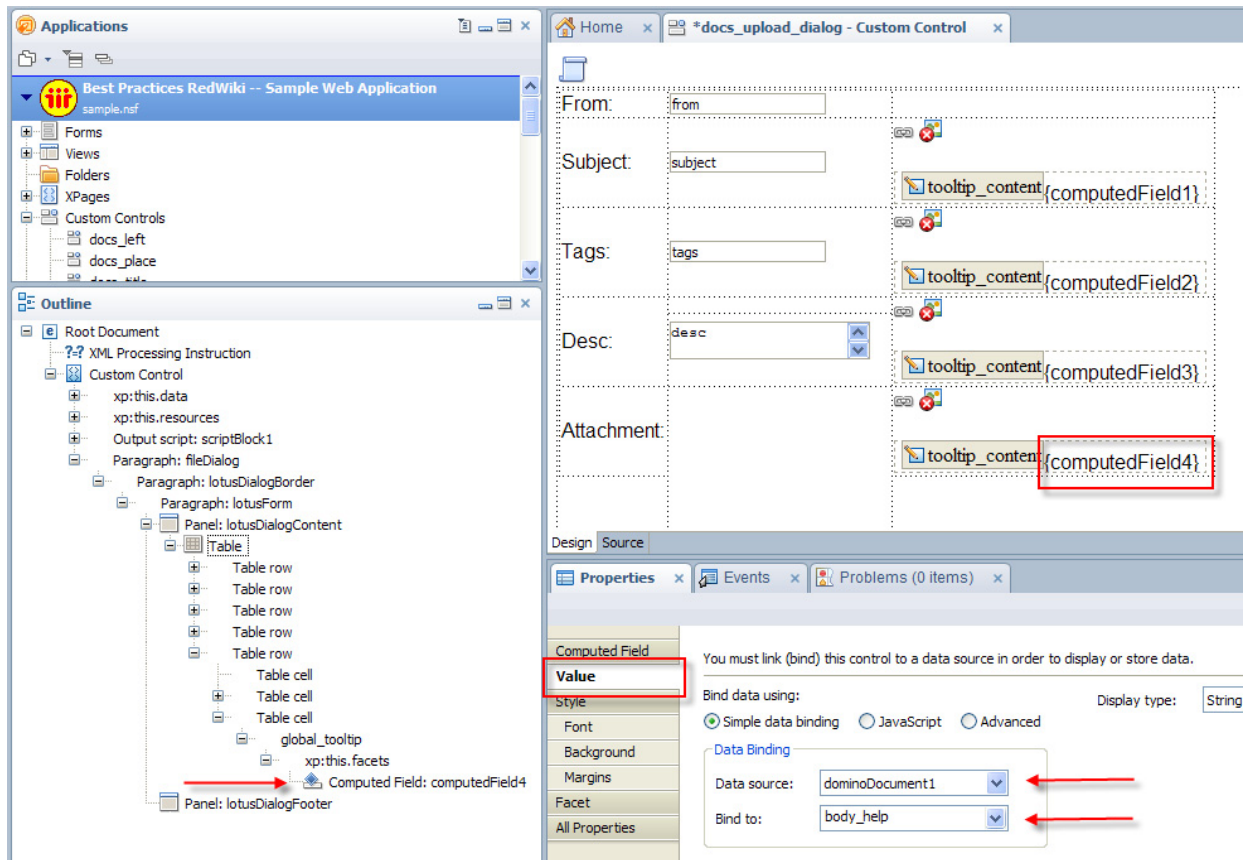
26. Similar to 2nd row, drop a “**global_tooltip**” control to 3rd column of 3rd row (tags). Drop a computed field over the tool tip control and bind it to “tags_help” field.



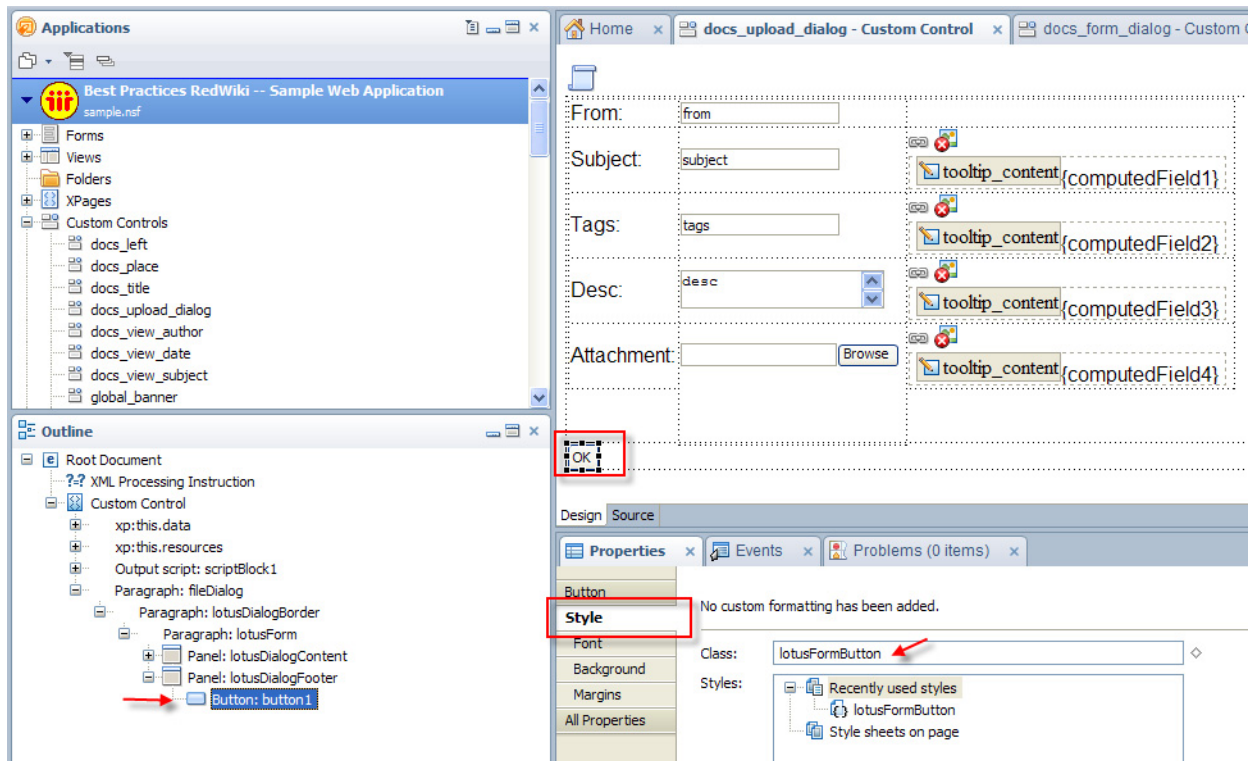
27. Similar to 3rd row, drop a “**global_tooltip**” control to 3rd column of 4th row (desc). Drop a computed field over the tool tip control and bind it to “desc_help” field.



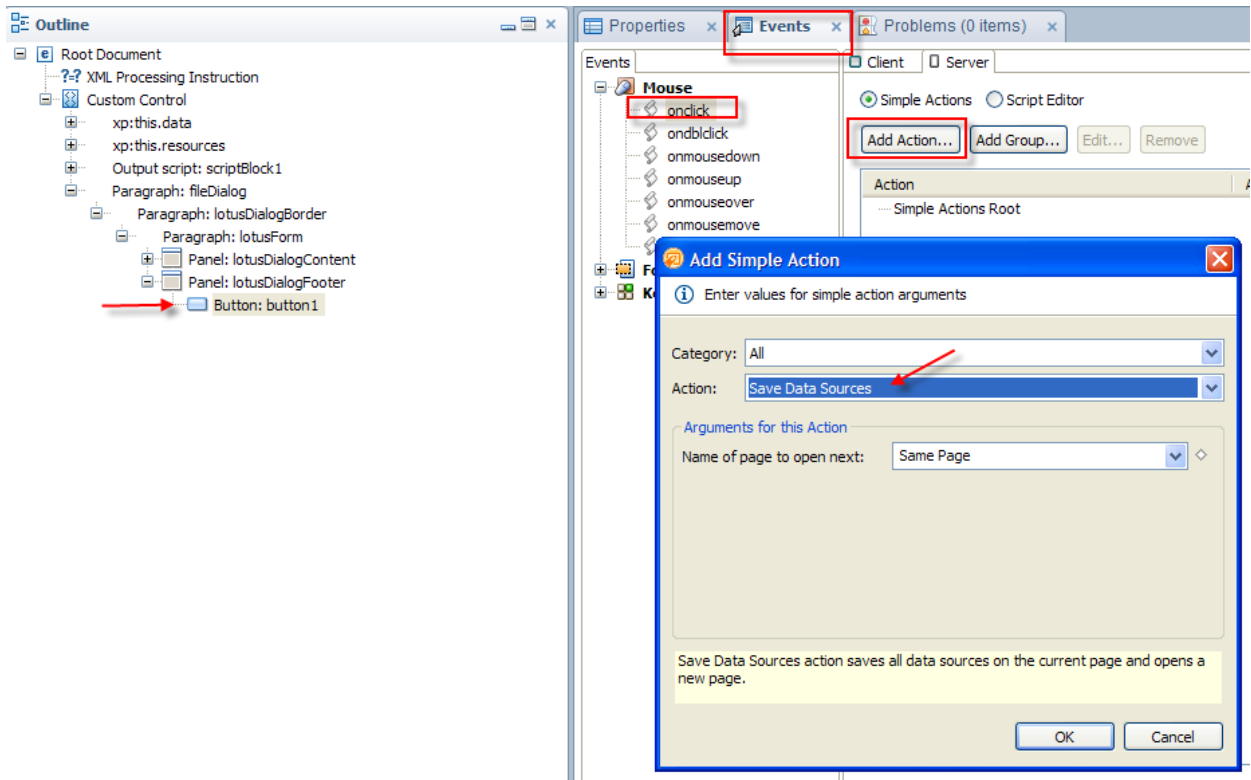
28. Similar to 4th row, drop a “**global_tooltip**” control to 3rd column of 5th row (attachment). Drop a computed field over the tool tip control and bind it to “body_help” field.



29. Drag and drop a Button control to the “**lotusDialogFooter**” panel. Label is as “OK” and enter “lotusFormButton” as CSS style class.



30. Click on “**events**” tab, select “**onclick**” event and click “**Add Action..**” button. Select “**Save Data Source**” action. This will save the form when OK button is clicked.



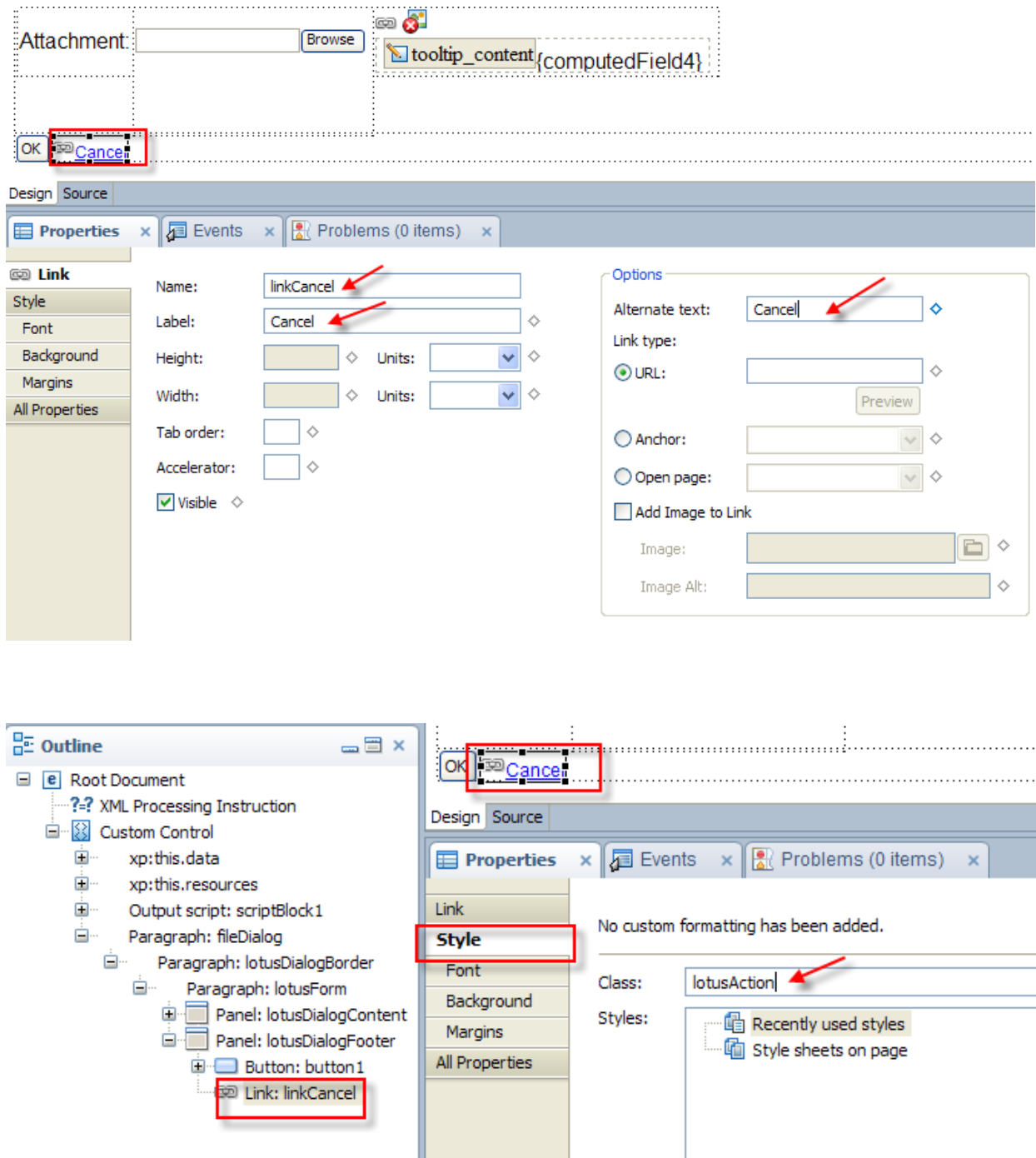
31. Drag a “**Link**” control and drop it to the right of “**OK**” button. Set the following properties:

linkName=linkCancel

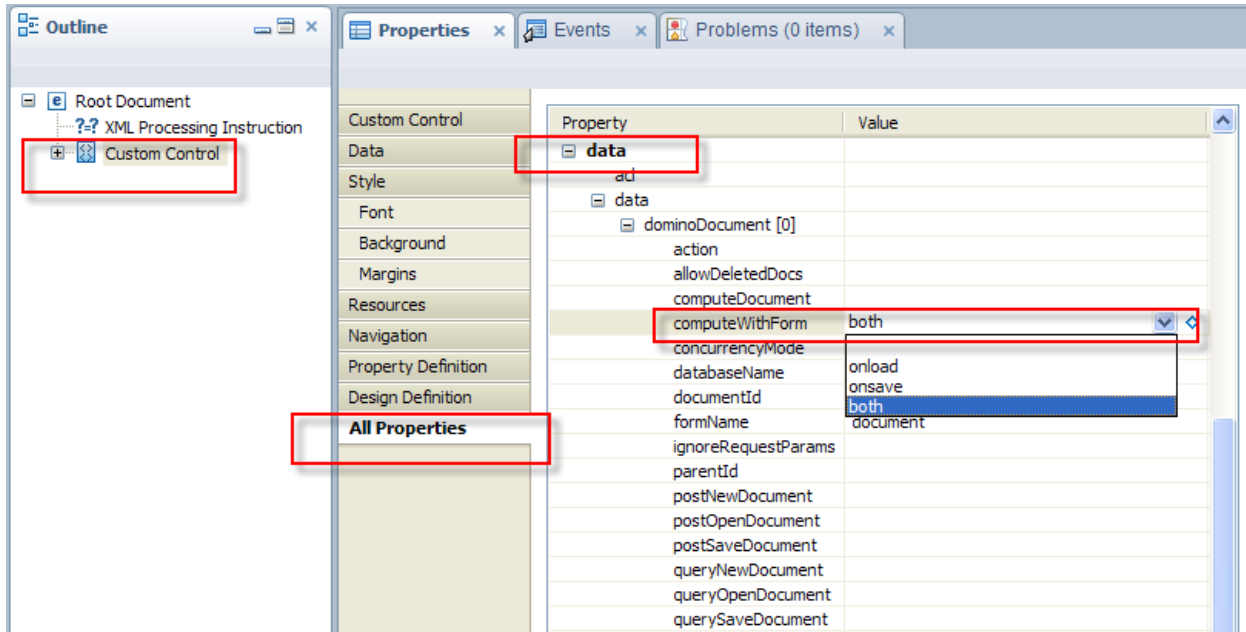
Label=Cancel

Alternate text=Cancel

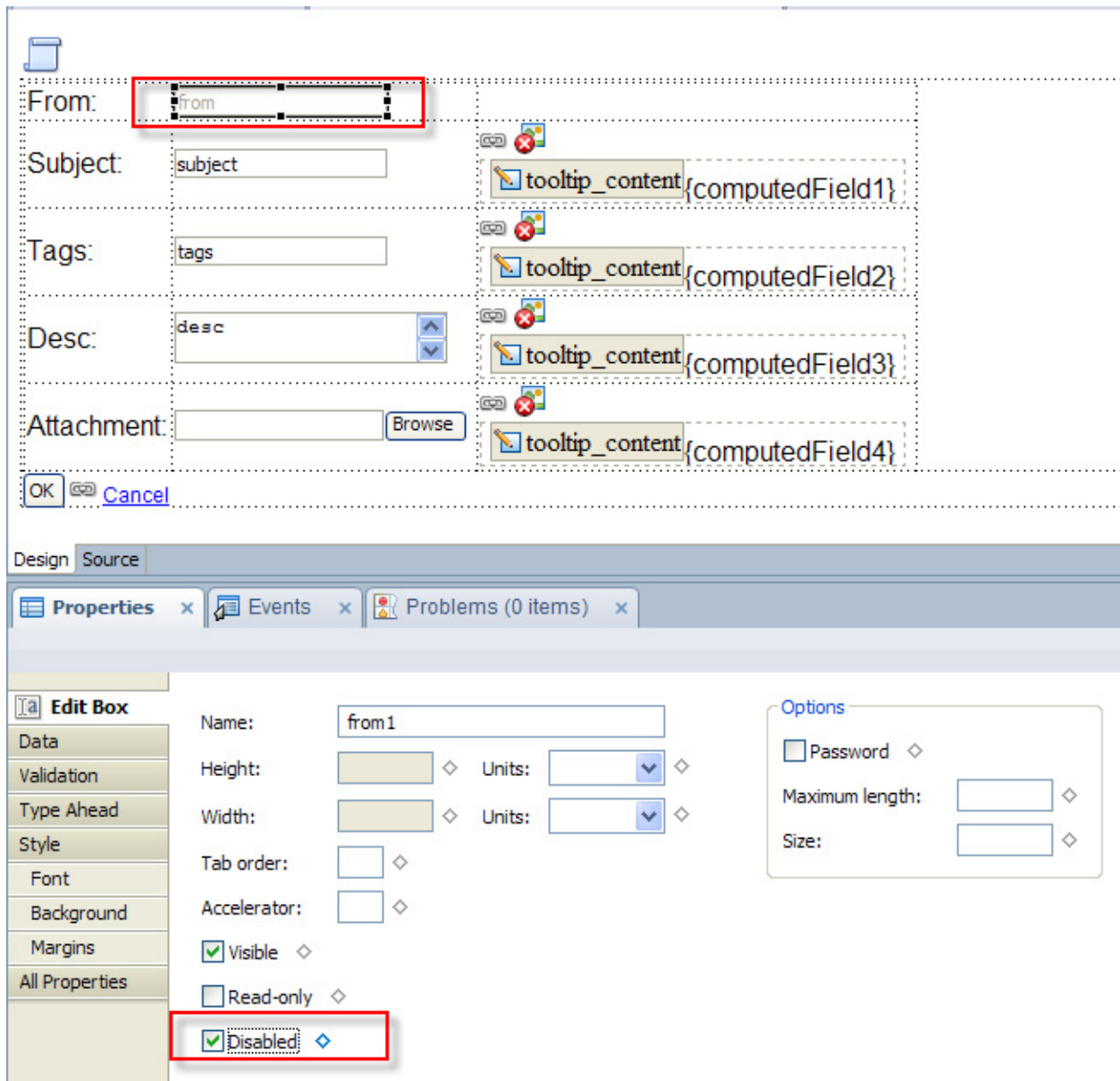
CSS Style Class=lotusAction



32. Select the custom control from the outline palette and select **both** as **computeWithForm** property. Since we have the computed fields in the form, this will make sure the fields displayed in the documents are computed on load as well on save.



33. Select “**form1**” edit box control and check the “**Disabled**” property. Since this field is computed, we don’t want users to be able to edit it.



34. Switch to the source editor and look for `<xp:div id="fileDialog" .. >`. Change it from `<xp:div>` to `<div>` tag. Make sure to change the corresponding end tag as well. In our JavaScript, we are calling the id of the `<div>` element as "fileDialog". If we use `<xp:div>`, the id will change at run time. `<xp:div>` provides advantages over `<div>`, such as visual and dynamic properties and all the other advantages that XPages controls offer. In this specific case, we are not taking advantage of any of those features, so it is easier to just change it from `<xp:div>` to `<div>` than to change JavaScript code. Note that `<div>` element uses "class" attribute – as opposed to "styleClass" used in `<xp:div>`



35. Click on **"Source"** tab in XPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

  <xp:this.data>
    <xp:dominoDocument var="dominoDocument1" formName="document"
      computeWithForm="both">
    </xp:dominoDocument>
  </xp:this.data>

  <xp:this.resources>
    <xp:dojoModule name="dijit.Dialog"></xp:dojoModule>
    <xp:script src="/dialog.js" clientSide="true"></xp:script>
  </xp:this.resources>

  <xp:scriptBlock id="scriptBlock1">
    <xp:this.value><![CDATA[XSP.addOnLoad(function(){dialog_create("fileDia
log","File Upload")});]]></xp:this.value>
  </xp:scriptBlock>

  <div id="fileDialog" class="lotusDialogWrapper">
    <xp:div id="lotusDialogBorder" styleClass="lotusDialogBorder">
      <xp:div id="lotusForm" styleClass="lotusDialog lotusForm">
        <xp:panel id="lotusDialogContent"
styleClass="lotusDialogContent">

          <xp:table>
            <xp:tr>
              <xp:td>
                <xp:label value="From:"
id="from_Label1" for="from1">
              </xp:label>
            </xp:td>
          </xp:table>
        </xp:panel>
      </xp:div>
    </xp:div>
  </div>
</xp:view>
```



```

        <xp:td styleClass="lotusText">
            <xp:inputText
value="#{dominoDocument1.from}" id="from1" disabled="true"
styleClass="lotusText">
                </xp:inputText>
            </xp:td>
        <xp:td></xp:td>
    </xp:tr>
    <xp:tr>
        <xp:td>
            <xp:label value="Subject:"
id="subject_Label1" for="subject1">
                </xp:label>
            </xp:td>
        <xp:td>
            <xp:inputText
value="#{dominoDocument1.subject}"
styleClass="lotusText">
                id="subject1"
            </xp:inputText>
        </xp:td>
        <xp:td>
            <xc:global_tooltip>
                <xp:this.facets>
                    <xp:text
escape="true" id="computedField1" xp:key="tooltip_content"
value="#{dominoDocument1.subject_help}"
                    </xp:text>
                </xp:this.facets>
            </xc:global_tooltip>
        </xp:td>
    </xp:tr>
    <xp:tr>
        <xp:td>
            <xp:label value="Tags:"
id="tags_Label1" for="tags1">
                </xp:label>
            </xp:td>
        <xp:td>
            <xp:inputText
value="#{dominoDocument1.tags}" id="tags1" styleClass="lotusText">
                </xp:inputText>
            </xp:td>
        <xp:td>
            <xc:global_tooltip>
                <xp:this.facets>
                    <xp:text
escape="true" id="computedField2" xp:key="tooltip_content"
value="#{dominoDocument1.tags_help}"
                    </xp:text>
                </xp:this.facets>
            </xc:global_tooltip>
        </xp:td>
    </xp:tr>
</xp:tr>

```

```

id="desc_Label1" for="desc1">
    <xp:td>
        <xp:label value="Desc:"
        </xp:label>
    </xp:td>
    <xp:td>
        <xp:inputTextarea
            id="desc1"
        </xp:inputTextarea>
    </xp:td>
    <xp:td>
        <xc:global_tooltip>
            <xp:this.facets>
                <xp:text
escape="true" id="computedField3" xp:key="tooltip_content"
            value="#{dominoDocument1.desc_help}"
            </xp:text>
            </xp:this.facets>
        </xc:global_tooltip>
    </xp:td>
</xp:tr>
<xp:tr>
    <xp:td>
        <xp:label value="Attachment:"
id="label1" for="fileUpload1"></xp:label></xp:td>
    <xp:td>
        <xp:fileUpload
id="fileUpload1" value="#{dominoDocument1.body}" styleClass="lotusText">
        </xp:fileUpload>
    </xp:td>
    <xp:td>
        <xc:global_tooltip>
            <xp:this.facets>
                <xp:text
escape="true" id="computedField4" xp:key="tooltip_content"
            value="#{dominoDocument1.body_help}"
            </xp:text>
            </xp:this.facets>
        </xc:global_tooltip>
    </xp:td>
</xp:tr>
</xp:table>
</xp:panel>
<xp:panel styleClass="lotusDialogFooter"
id="lotusDialogFooter">
    <xp:button value="OK" id="button1"
styleClass="lotusFormButton">
        <xp:eventHandler event="onclick"
submit="true"
            refreshMode="complete">
                <xp:this.action>
                    <xp:save></xp:save>

```

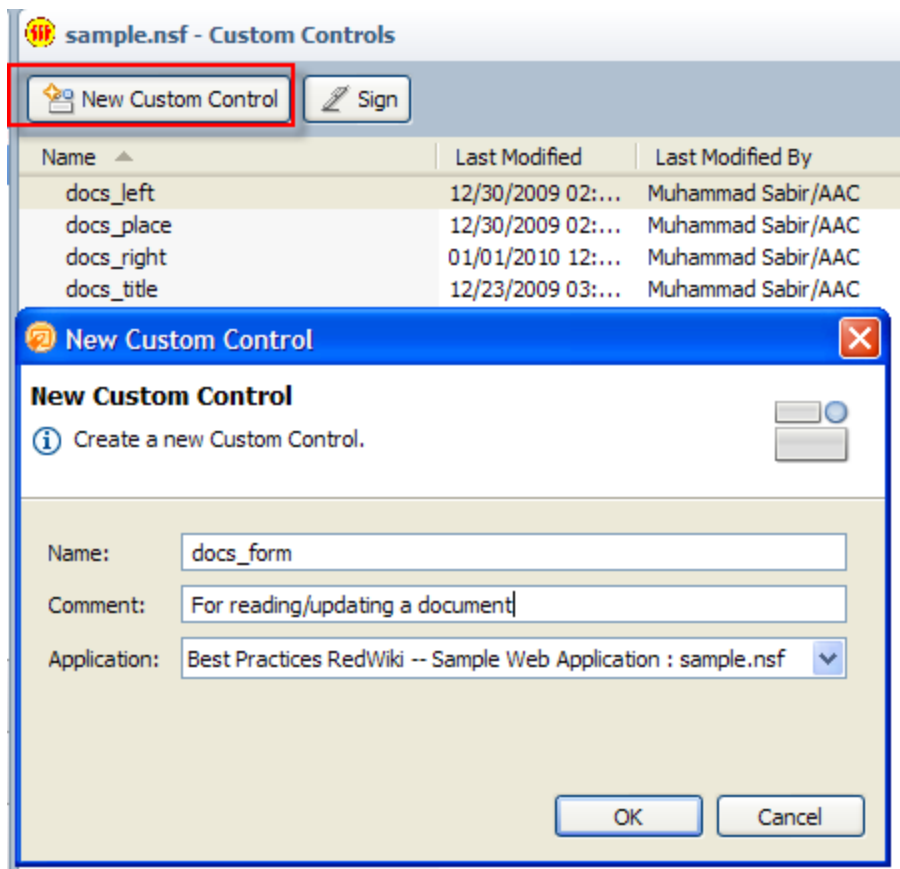
```

        </xp:this.action>
    </xp:EventHandler>
</xp:button>
<xp:link escape="true" text="Cancel"
id="linkCancel" title="Cancel"
        styleClass="lotusAction"
onclick="diigit.byId('fileDialog').hide();">
    </xp:link>
</xp:panel>
</xp:div>
</xp:div>
</div>
</xp:view>

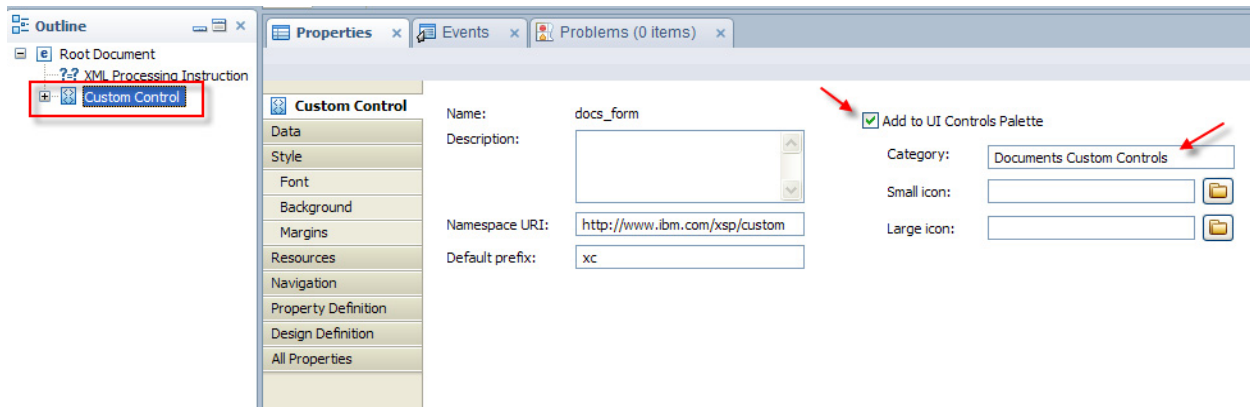
```

8.6.3: Creating custom control for document read and edit

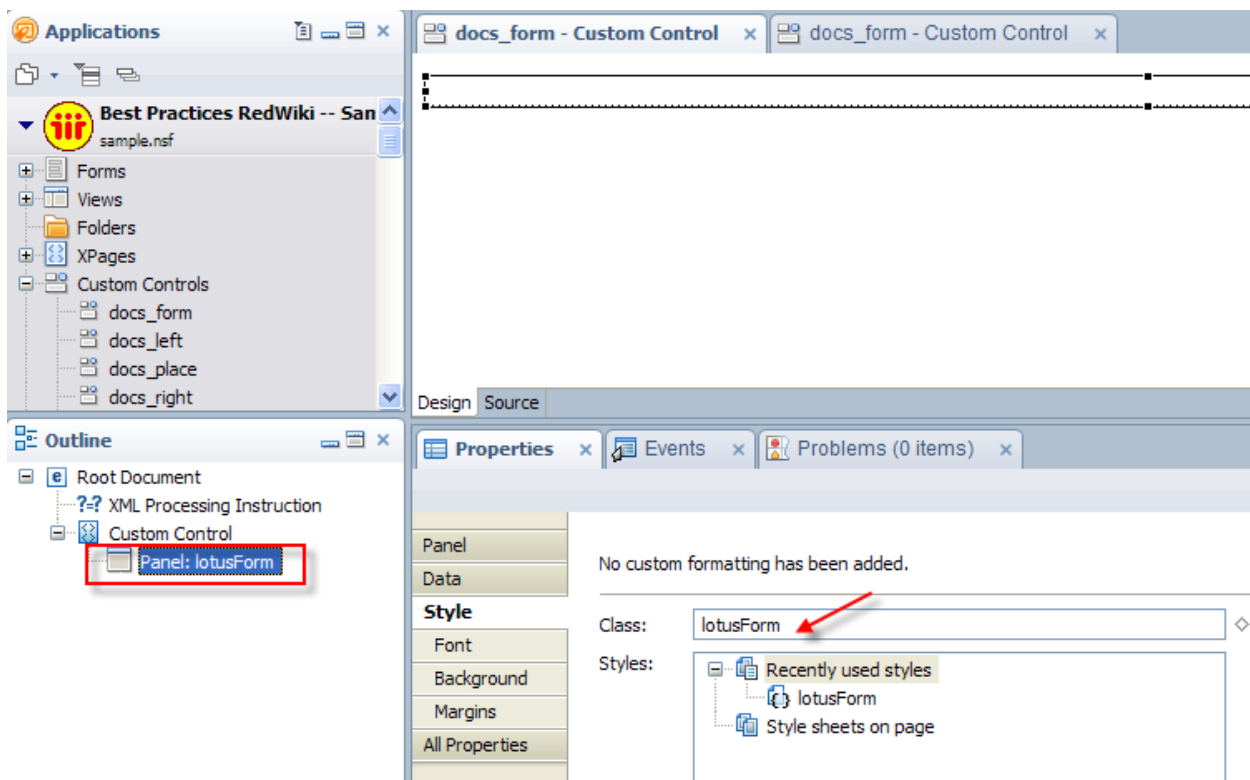
1. Create a new custom control named "docs_form".



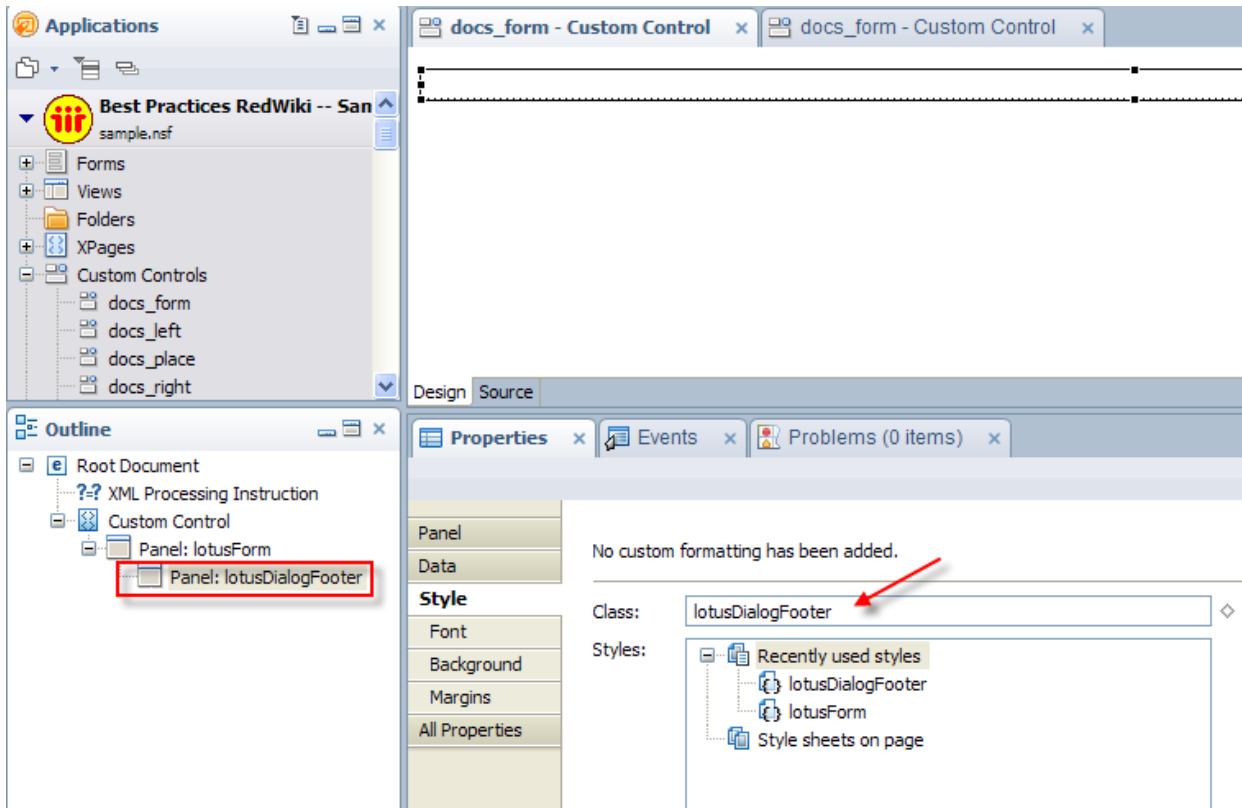
2. Under the Properties tab, make sure Custom Control is selected and check **"Add to UI Controls Palette"** and enter **"Documents Custom Controls"** as the category. This moves this custom control under the category **"Documents Custom Controls"**.



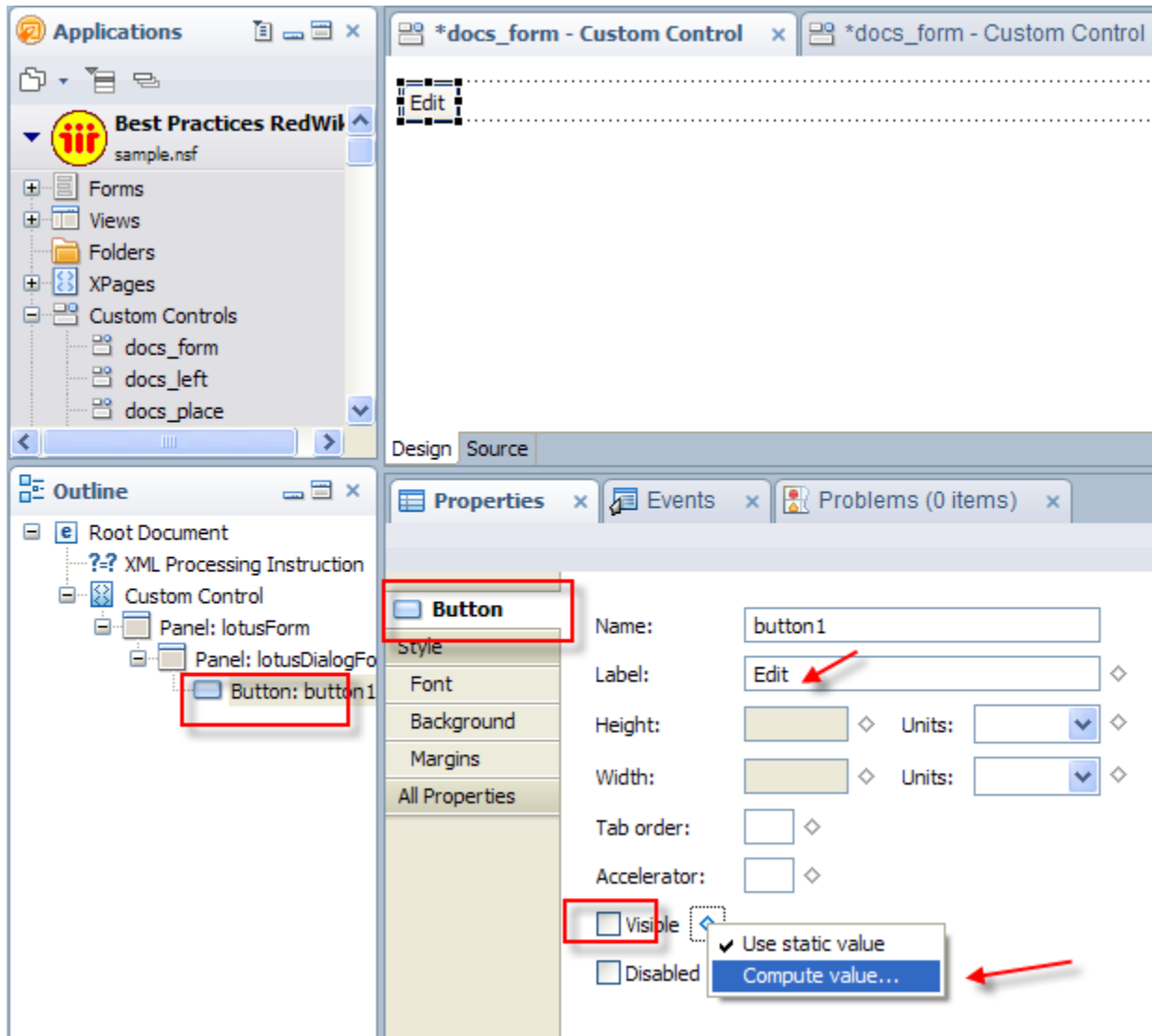
3. Drag and drop a **Panel** control. Enter “**lotusForm**” both as name and as CSS style class.

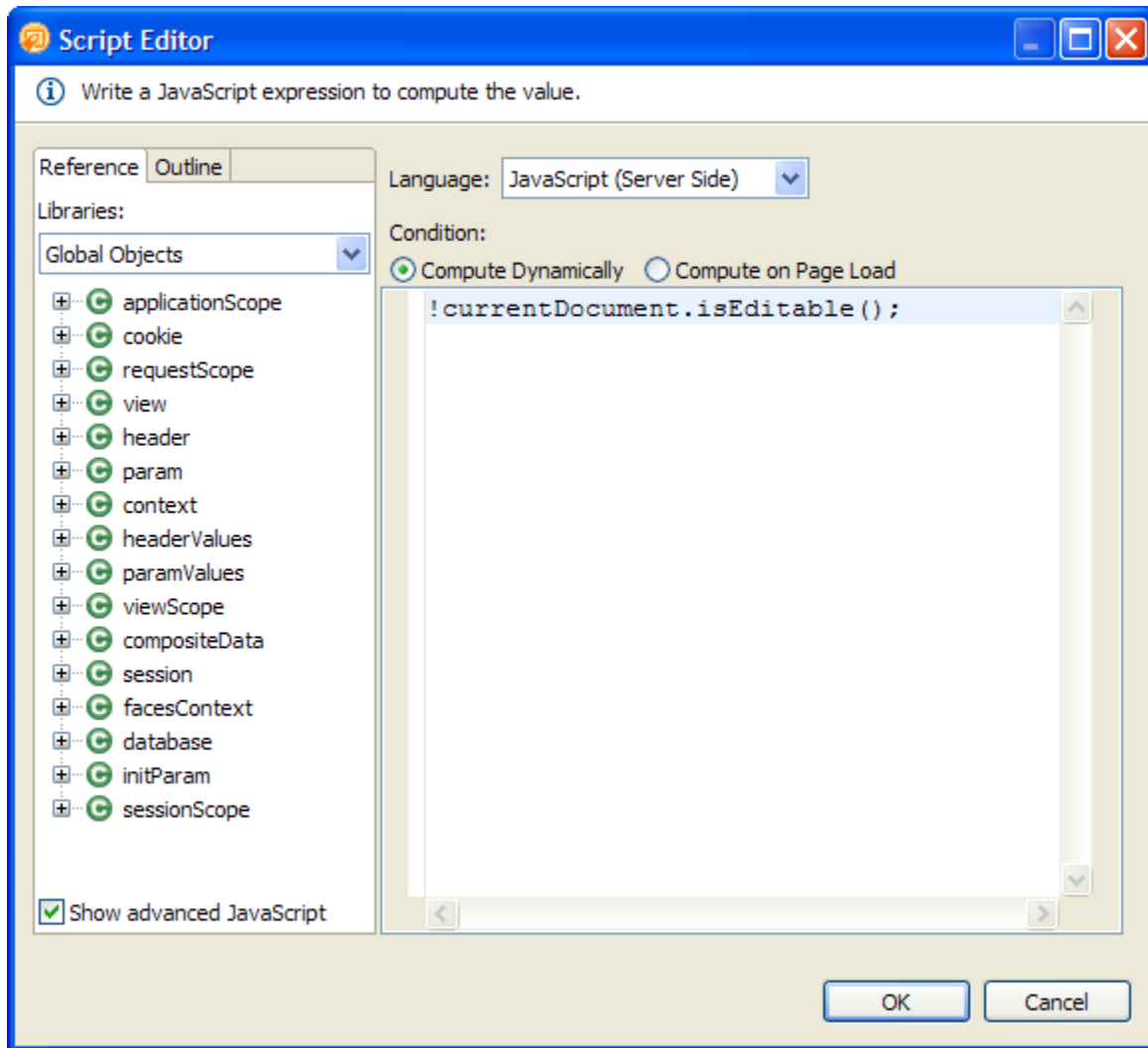


4. Drag a **Panel** control and drop it within the “**lotusForm**” panel. Enter “**lotusDialogFooter**” both as name and as CSS style class.

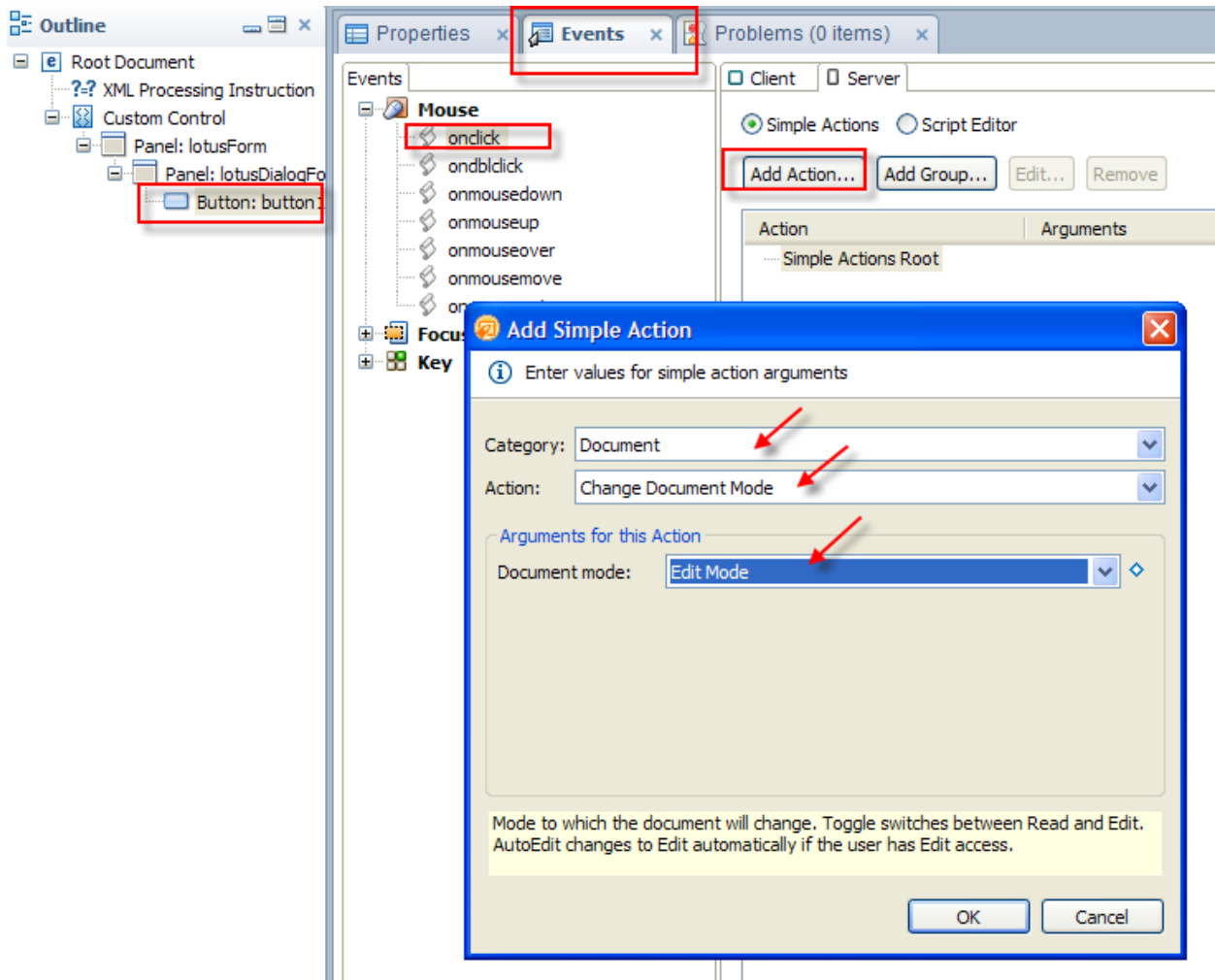


5. Drag a **Button** control and drop it on “**lotusDialogFooter**” panel. Enter the following properties:
Label=Edit
Visible= Computed = !currentDocument.isEditable();
CSS style class=lotusFormButton

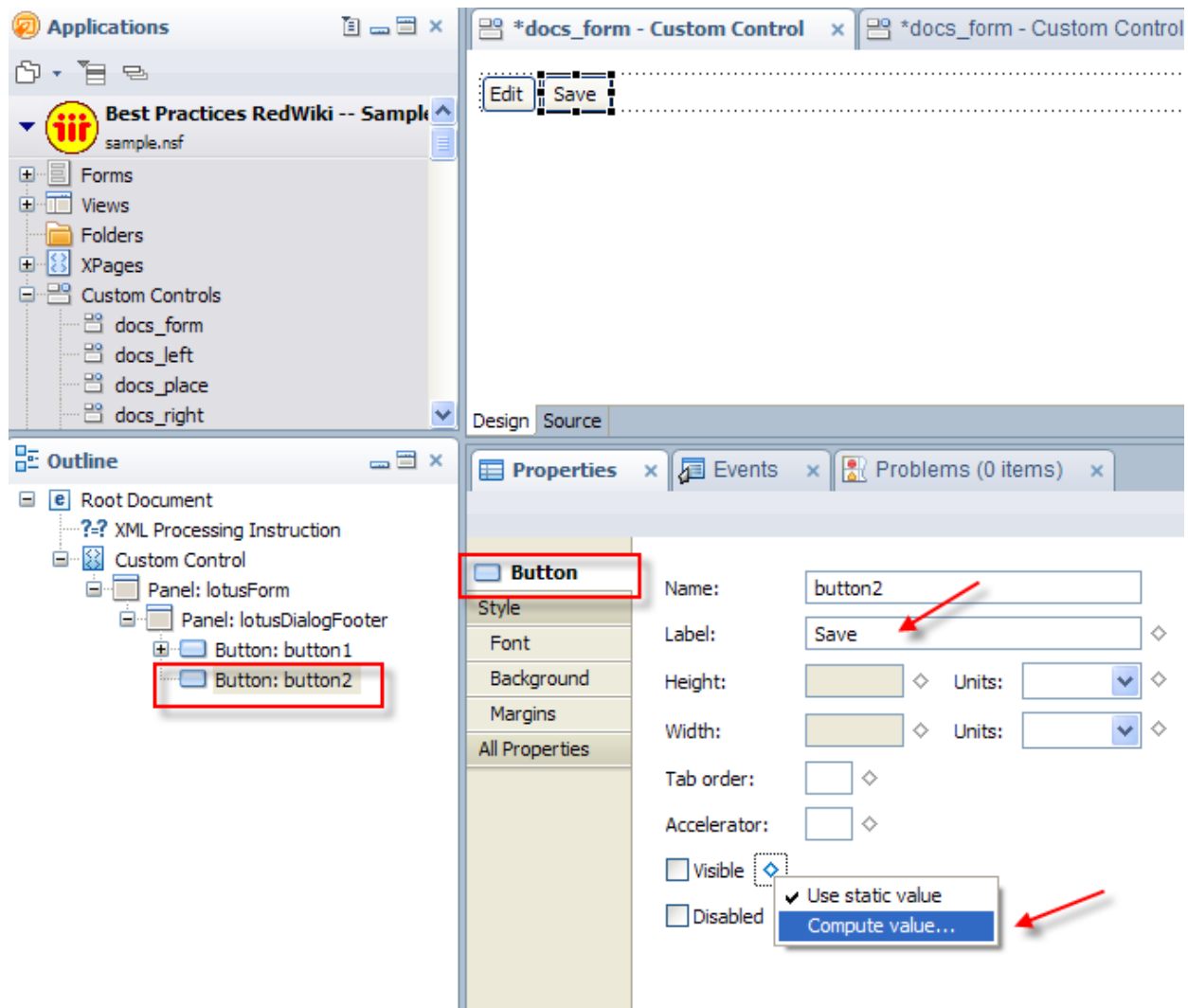


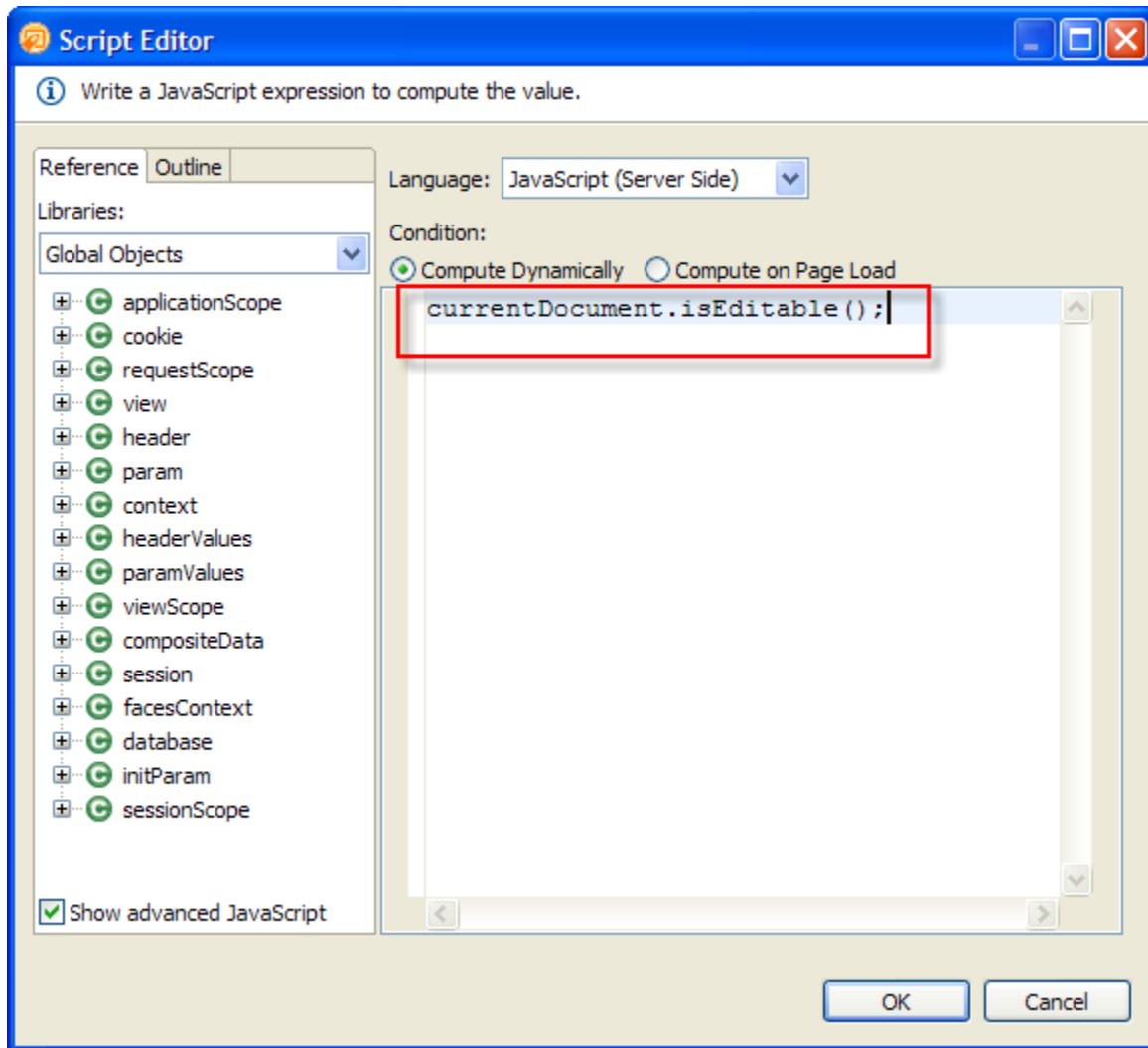


6. Select **Events** tab for **"Edit"** button and click **"Add Action"** for **"onclick"** event. Select category as **"Document"**, action as **"Change Document Mode"** and mode as **"Edit Mode"**. This will edit the document when the button is clicked.

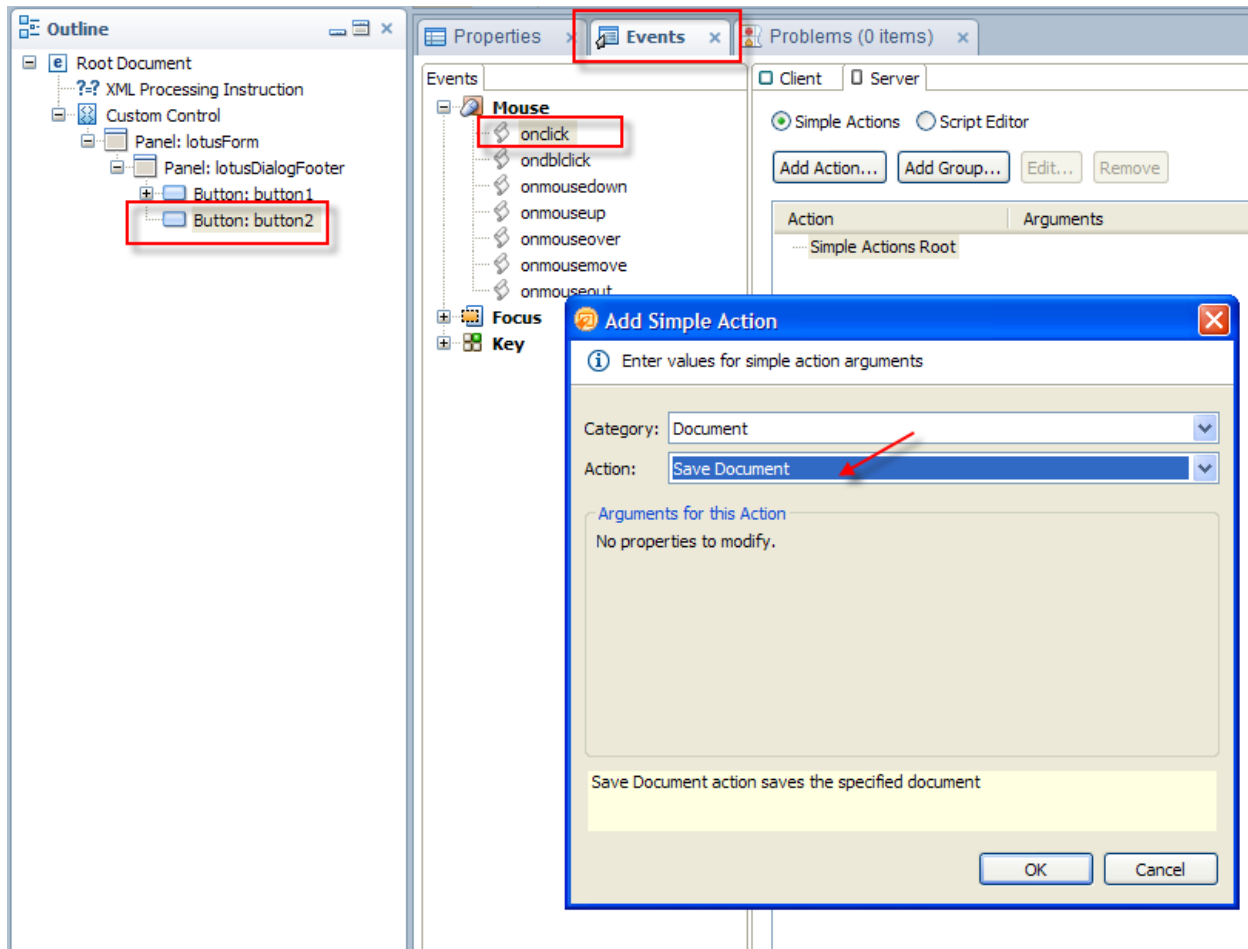


7. Drag a **Button** control and drop it to the right of "Edit" button. Enter the following properties:
Label=Save
Visible= Computed = `currentDocument.isEditable();`
CSS style class=lotusFormButton

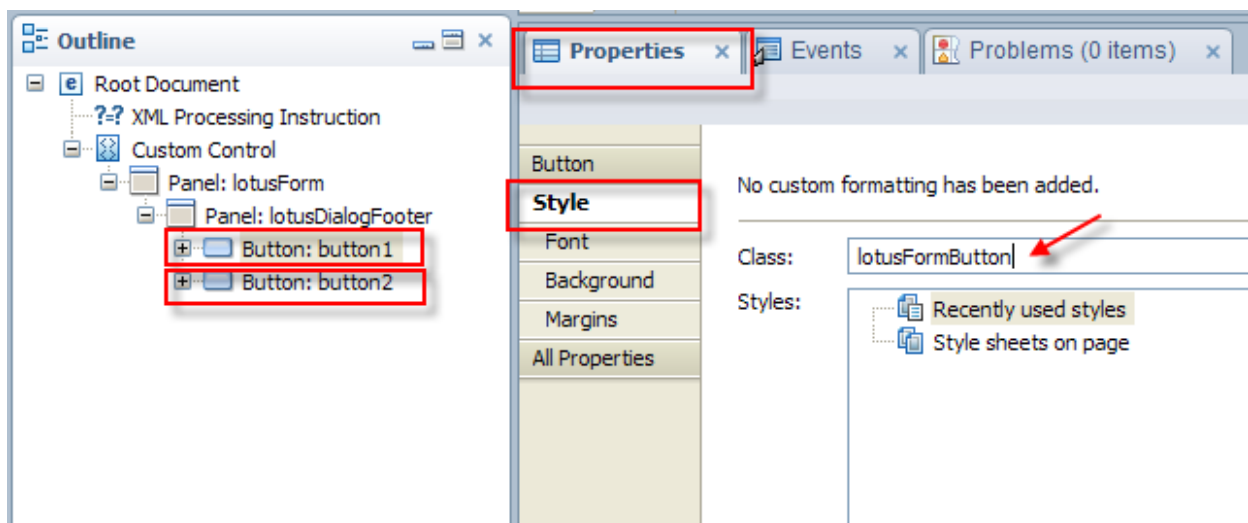




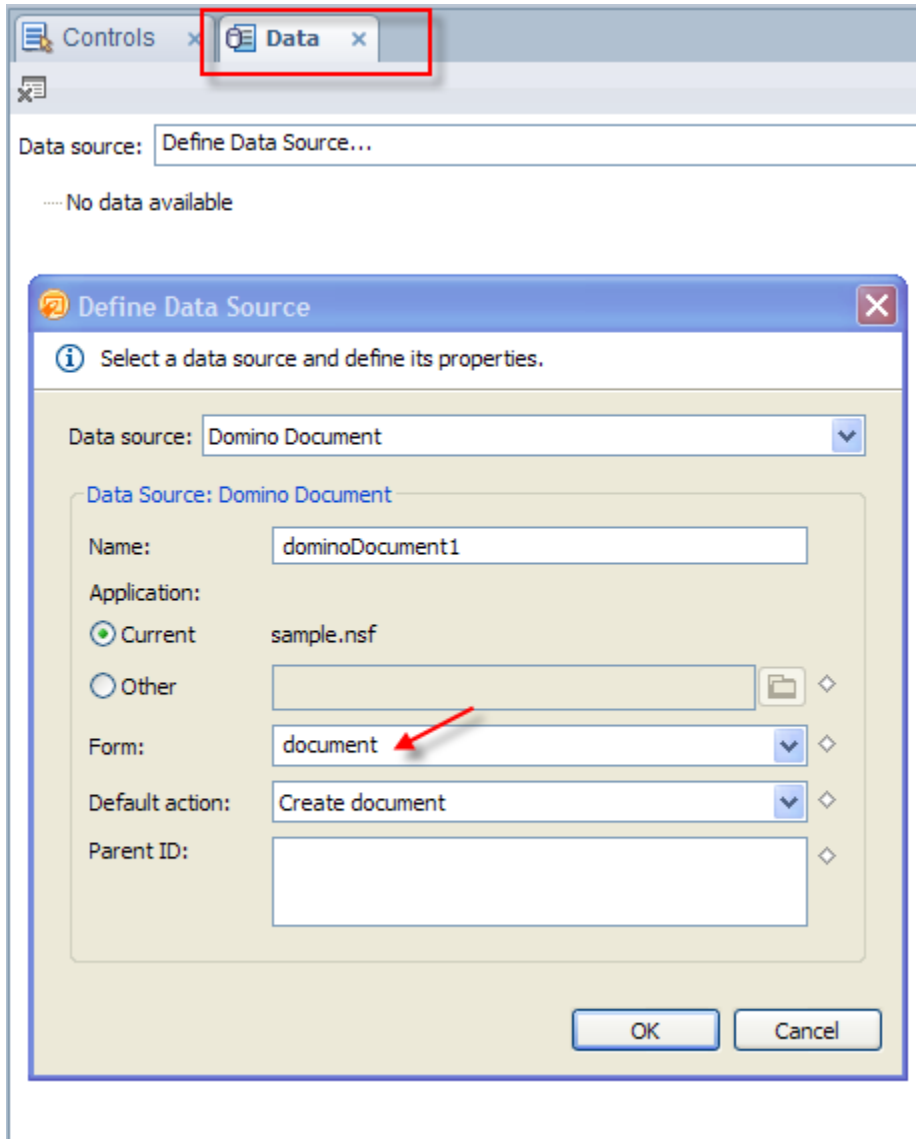
8. Select **Events** tab for **“Save”** button and click **“Add Action”** for **“onclick”** event. Select category as **“Document”**, action as **“Save Document”**. This will save the document when the button is clicked.



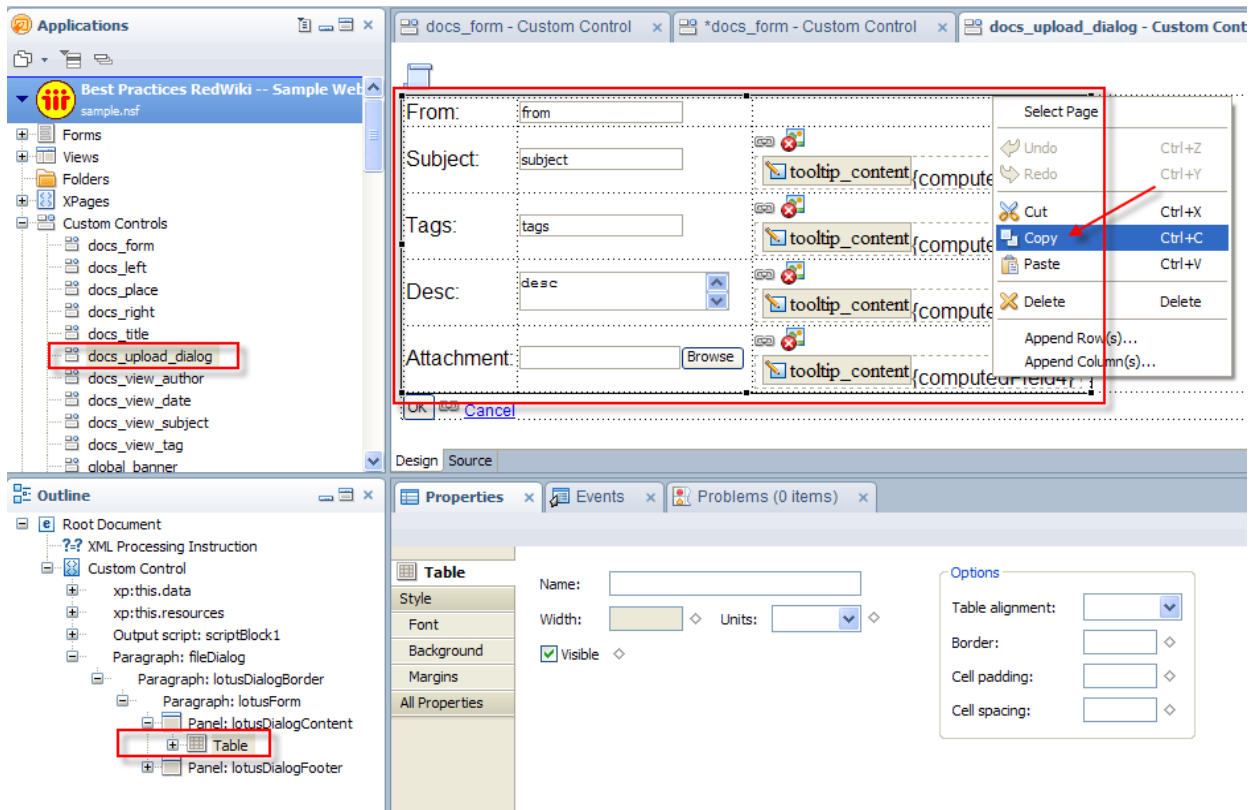
9. For each of the two buttons, make sure the CSS style class is set as “**lotusFormButton**”.



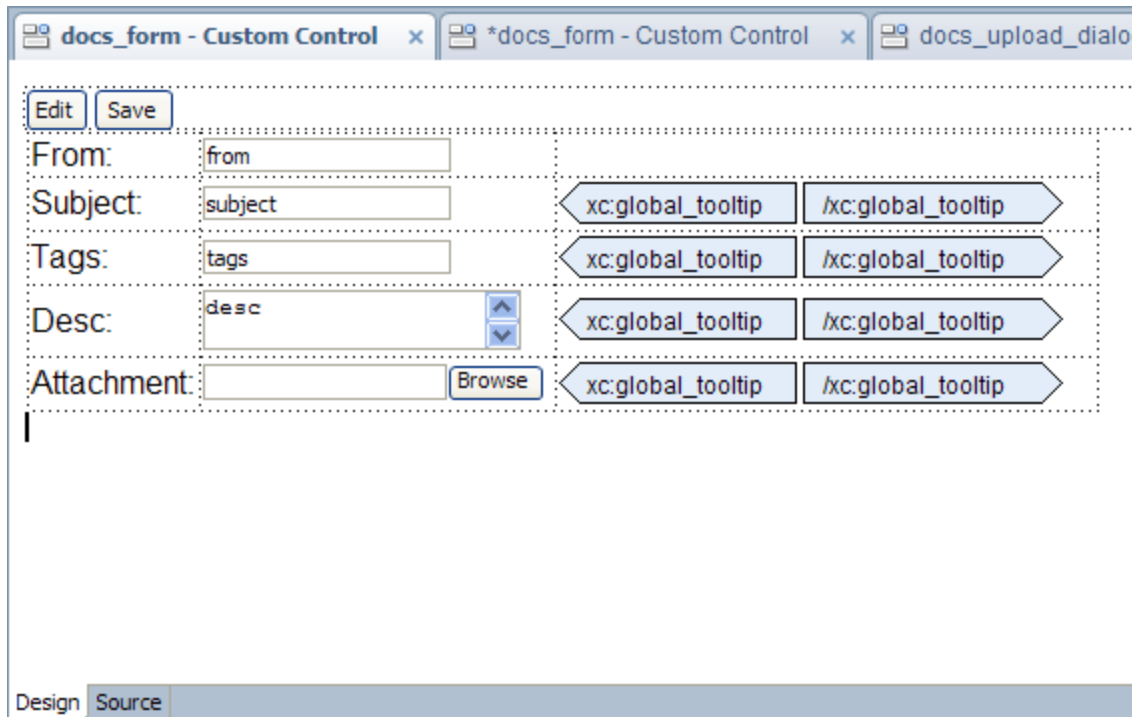
- Click on the **"Data"** palette (next to the controls palette), and select **"Define Data Source..."** from the drop down. Select **"Domino Document"** as data source. Select **"document"** as form and **"Create document"** as default action.



- Open **"docs_upload_dialog"** custom control created earlier. Select the table contained within **"lotusDialogContent"** panel. Right click and select copy. We are going to copy this table over to **"docs_form"** custom control that we are creating in this step.



12. Paste copied table to “docs_form” below “lotusForm” panel.



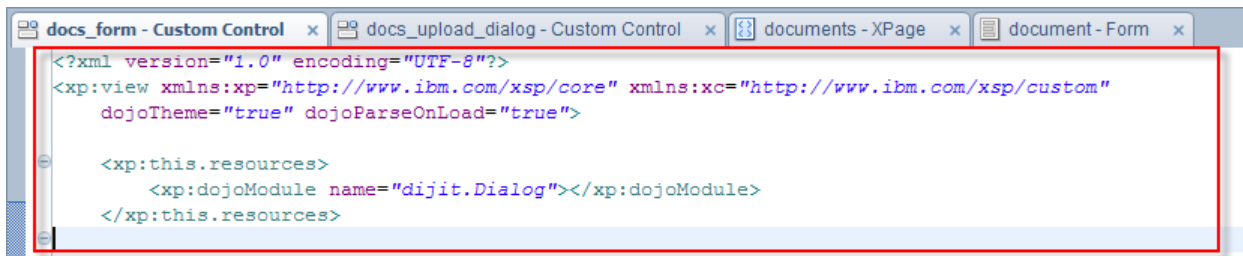
13. Since “xc” name space has not been referenced, design view does not display the “global_tooltip” custom control. Switch to source tab and replace this:

```
<xp:view xmlns:xp="http://www.ibm.com/xsp/core">
```

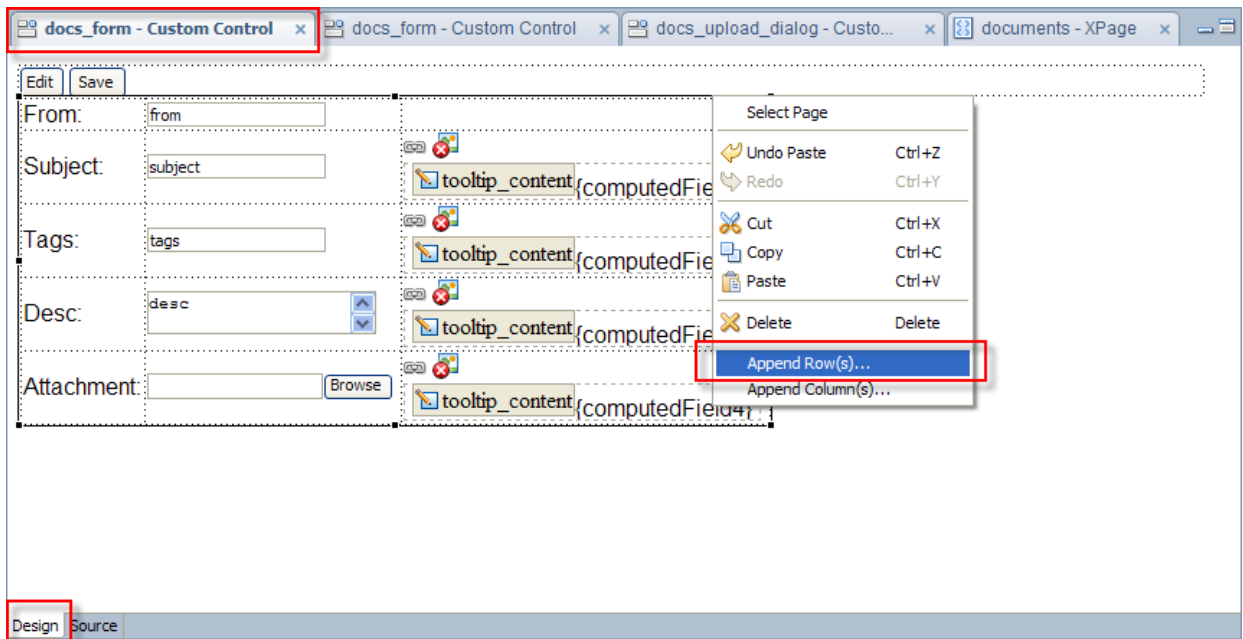
With this:

```
<xp:view xmlns:xp="http://www.ibm.com/xsp/core"
  xmlns:xc="http://www.ibm.com/xsp/custom" dojoTheme="true"
  dojoParseOnLoad="true" >

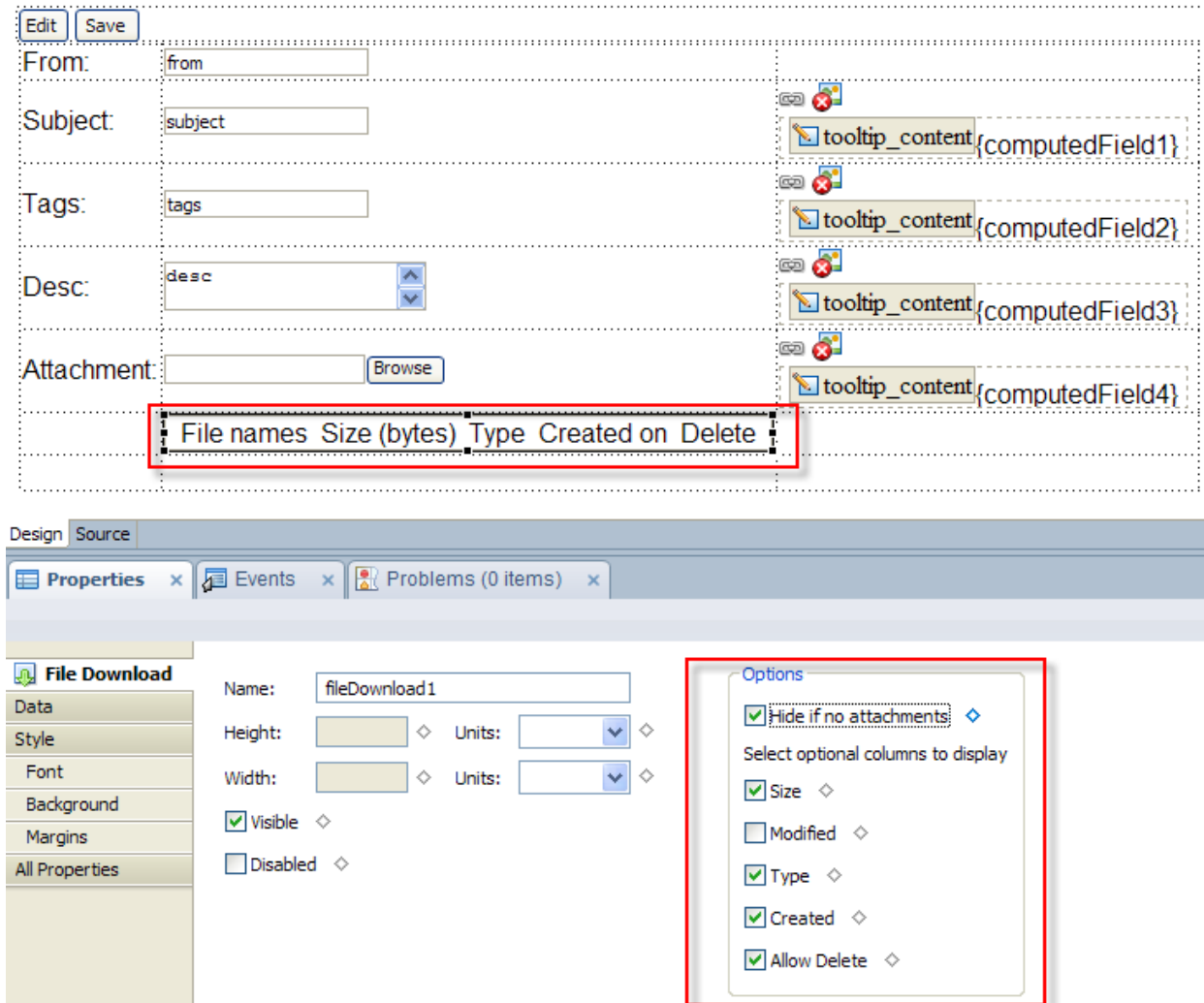
  <xp:this.resources>
    <xp:dojoModule name="dijit.Tooltip"></xp:dojoModule>
  </xp:this.resources>
```



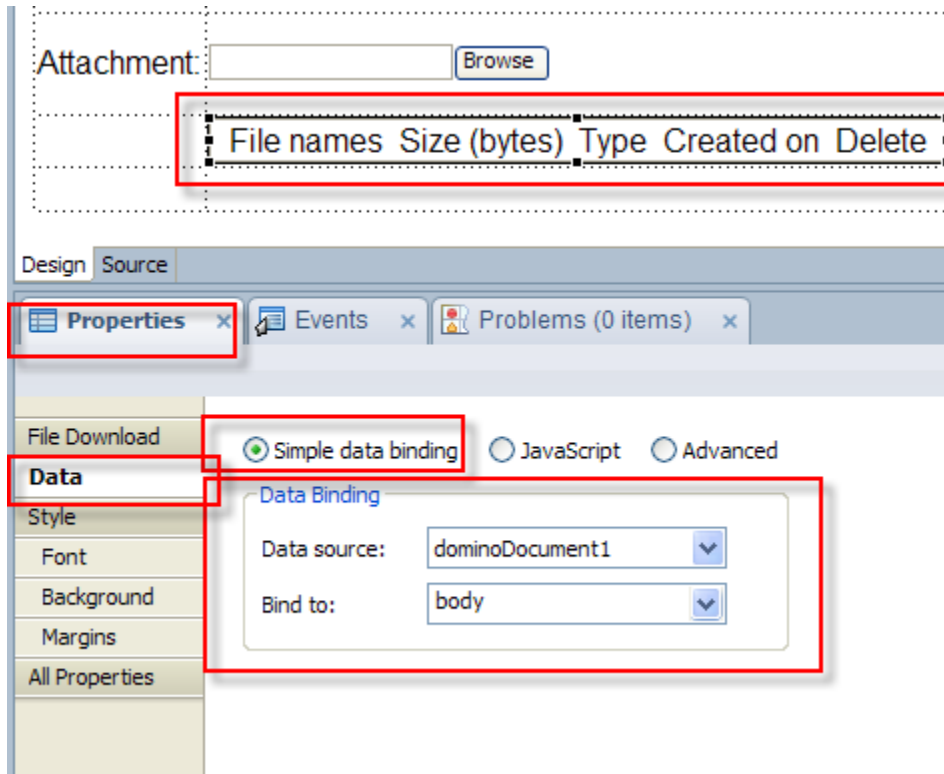
14. Switch back to the design tab, select table control and append two rows.



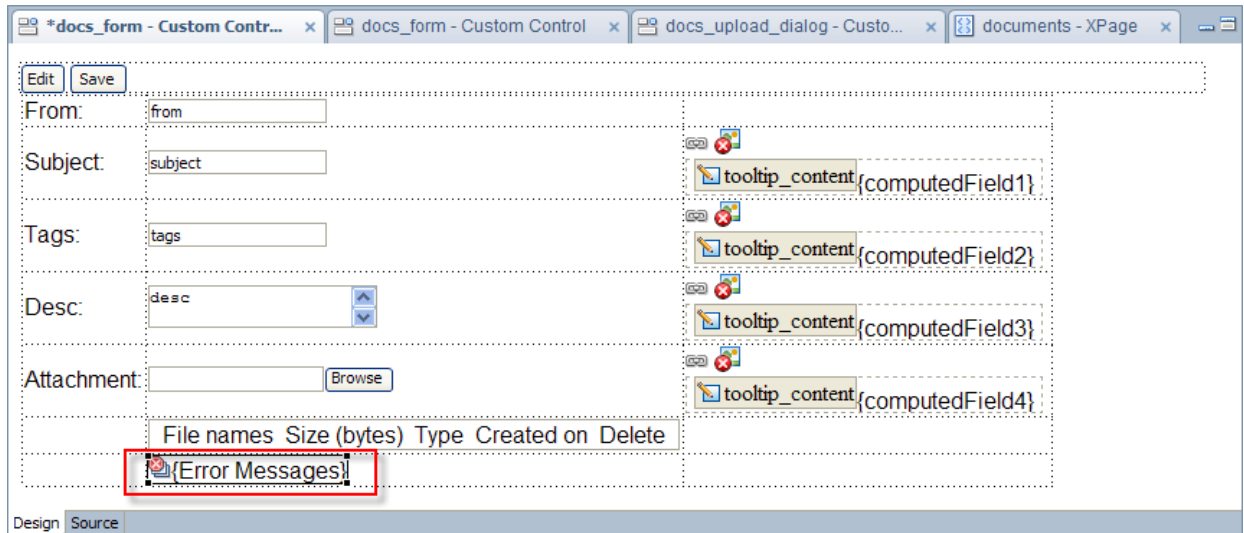
15. Drag “**File Download**” control from the core controls and drop it to the middle column in the second last row, as shown in the screenshot. From the properties palette, check the following properties under options section: **Hide if no attachment**, **Size**, **Type**, **Created**, and **Allow Delete**.



16. Select “**File Download**” control, and click on “**Data**” tab in the properties tab. Bind this control to the body field of “**dominoDocument1**” data source. This will display the files attached within the body field.



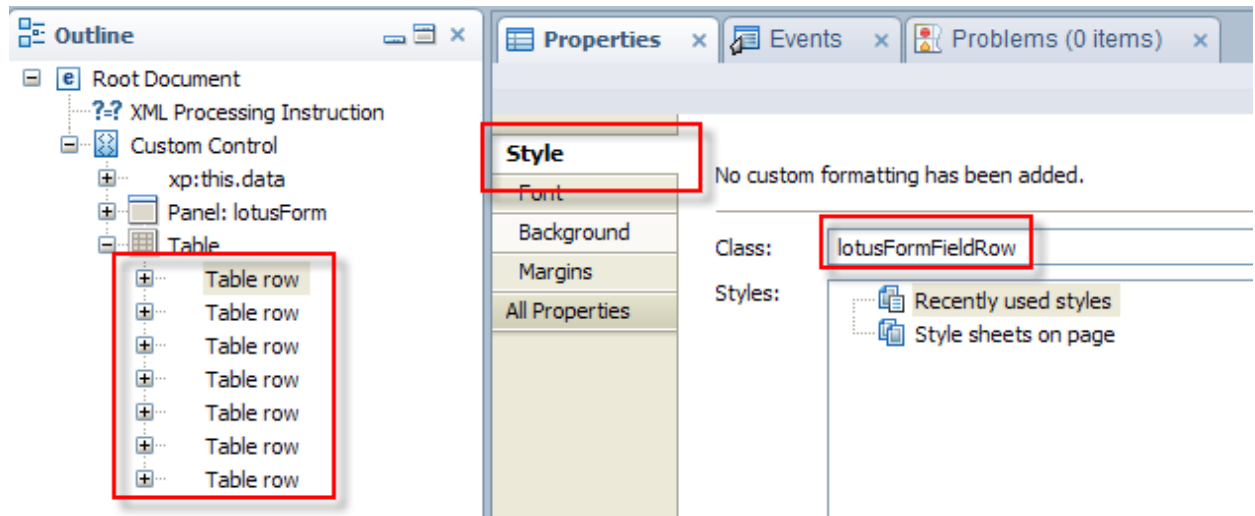
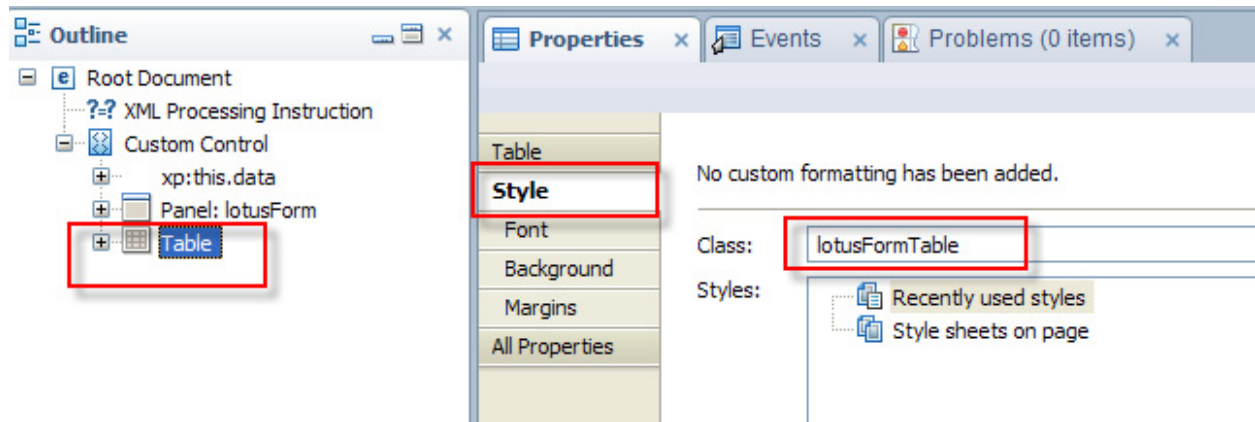
17. From core controls, drag “**Display Errors**” control and drop it to the middle column in the last row.

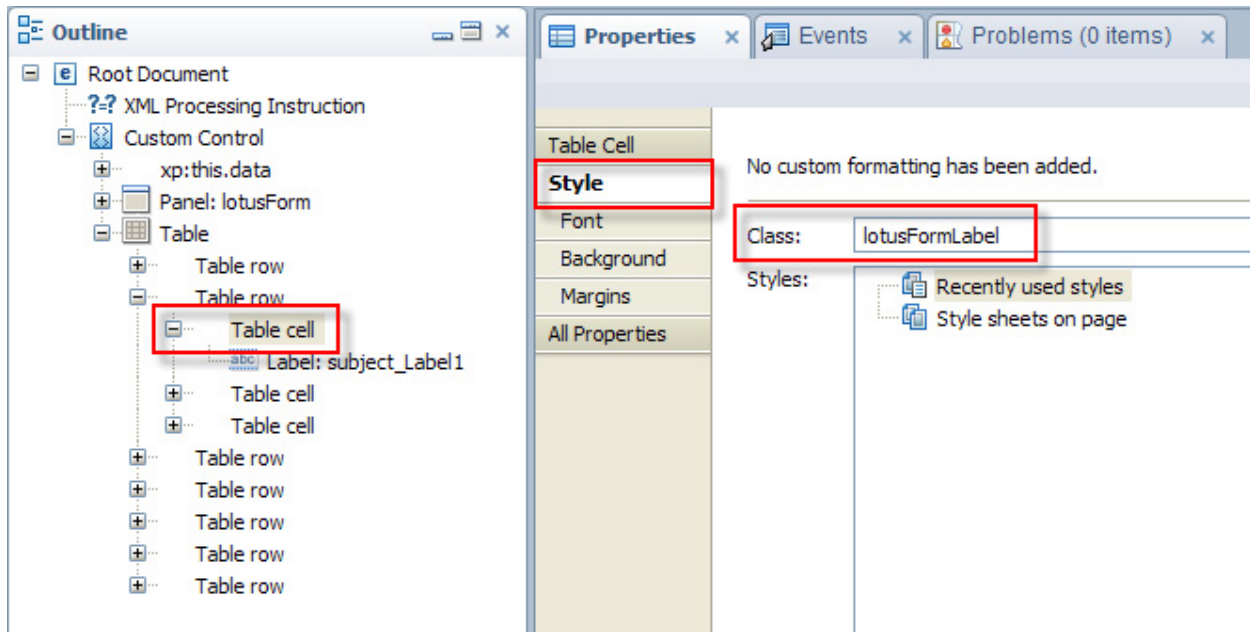


18. Within the table control, set the following CSS style properties.

Element	CSS Style Class
Table	lotusFormTable
Table Rows (each one of them)	lotusFormFieldRow
Table Cell (Only those that contain labels)	lotusFormLabel

--	--





19. Click on **"Source"** tab in XPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

    <xp:this.data>
        <xp:dominoDocument var="dominoDocument1"
formName="document"></xp:dominoDocument>
    </xp:this.data>

    <xp:panel id="lotusForm" styleClass="lotusForm">
        <xp:panel styleClass="lotusDialogFooter" id="lotusDialogFooter">
            <xp:button value="Edit" id="button1"

                rendered="#{javascript:currentDocument.isEditable();}"
styleClass="lotusFormButton">
                <xp:eventHandler event="onclick" submit="true"
                    refreshMode="complete">
                    <xp:this.action>
                        <xp:changeDocumentMode
mode="edit"></xp:changeDocumentMode>
                    </xp:this.action>
                </xp:eventHandler>
            </xp:button>
            <xp:button id="button2" value="Save"

                rendered="#{javascript:currentDocument.isEditable();}"
styleClass="lotusFormButton">
                <xp:eventHandler event="onclick" submit="true"
```

```

        refreshMode="complete">
        <xp:this.action>
            <xp:saveDocument></xp:saveDocument>
        </xp:this.action>
    </xp:eventHandler>
</xp:button>
</xp:panel>
</xp:panel>

<xp:table styleClass="lotusFormTable">
    <xp:tr styleClass="lotusFormFieldRow">
        <xp:td styleClass="lotusFormLabel">
            <xp:label value="From:" id="from_Label1" for="from1">
            </xp:label>
        </xp:td>
        <xp:td>
            <xp:inputText value="#{dominoDocument1.from}"
id="from1"
                styleClass="lotusText">
            </xp:inputText>
        </xp:td>
        <xp:td></xp:td>
    </xp:tr>
    <xp:tr styleClass="lotusFormFieldRow">
        <xp:td styleClass="lotusFormLabel">
            <xp:label value="Subject:" id="subject_Label1"
for="subject1">
            </xp:label>
        </xp:td>
        <xp:td>
            <xp:inputText value="#{dominoDocument1.subject}"
id="subject1"
                styleClass="lotusText">
            </xp:inputText>
        </xp:td>
        <xp:td>
            <xc:global_tooltip>
                <xp:this.facets>
                    <xp:text escape="true"
id="computedField1" xp:key="tooltip_content"
                    value="#{dominoDocument1.subject_help}"
                    </xp:text>
                </xp:this.facets>
            </xc:global_tooltip>
        </xp:td>
    </xp:tr>
    <xp:tr styleClass="lotusFormFieldRow">
        <xp:td styleClass="lotusFormLabel">
            <xp:label value="Tags:" id="tags_Label1" for="tags1">
            </xp:label>
        </xp:td>
        <xp:td>
            <xp:inputText value="#{dominoDocument1.tags}"
id="tags1"
                styleClass="lotusText">

```

```

        </xp:inputText>
    </xp:td>
    <xp:td>
        <xc:global_tooltip>
            <xp:this.facets>
                <xp:text escape="true"
id="computedField2" xp:key="tooltip_content"

                value="#{dominoDocument1.tags_help}">
            </xp:text>
        </xp:this.facets>
    </xc:global_tooltip>
    </xp:td>
</xp:tr>
<xp:tr styleClass="lotusFormFieldRow">
    <xp:td styleClass="lotusFormLabel">
        <xp:label value="Desc:" id="desc_Label1" for="desc1">
        </xp:label>
    </xp:td>
    <xp:td>
        <xp:inputTextarea value="#{dominoDocument1.desc}"
            id="desc1" styleClass="lotusText">
        </xp:inputTextarea>
    </xp:td>
    <xp:td>
        <xc:global_tooltip>
            <xp:this.facets>
                <xp:text escape="true"
id="computedField3" xp:key="tooltip_content"

                value="#{dominoDocument1.desc_help}">
            </xp:text>
        </xp:this.facets>
    </xc:global_tooltip>
    </xp:td>
</xp:tr>
<xp:tr styleClass="lotusFormFieldRow">
    <xp:td styleClass="lotusFormLabel">
        <xp:label value="Attachment:" id="attachment_label"
for="fileUpload1">
        </xp:label>
    </xp:td>
    <xp:td>
        <xp:fileUpload id="fileUpload1"
value="#{dominoDocument1.body}"
            styleClass="lotusText">
        </xp:fileUpload>
    </xp:td>
    <xp:td>
        <xc:global_tooltip>
            <xp:this.facets>
                <xp:text escape="true"
id="computedField4" xp:key="tooltip_content"

                value="#{dominoDocument1.body_help}">
            </xp:text>
        </xp:this.facets>
    </xc:global_tooltip>
    </xp:td>
</xp:tr>

```

```

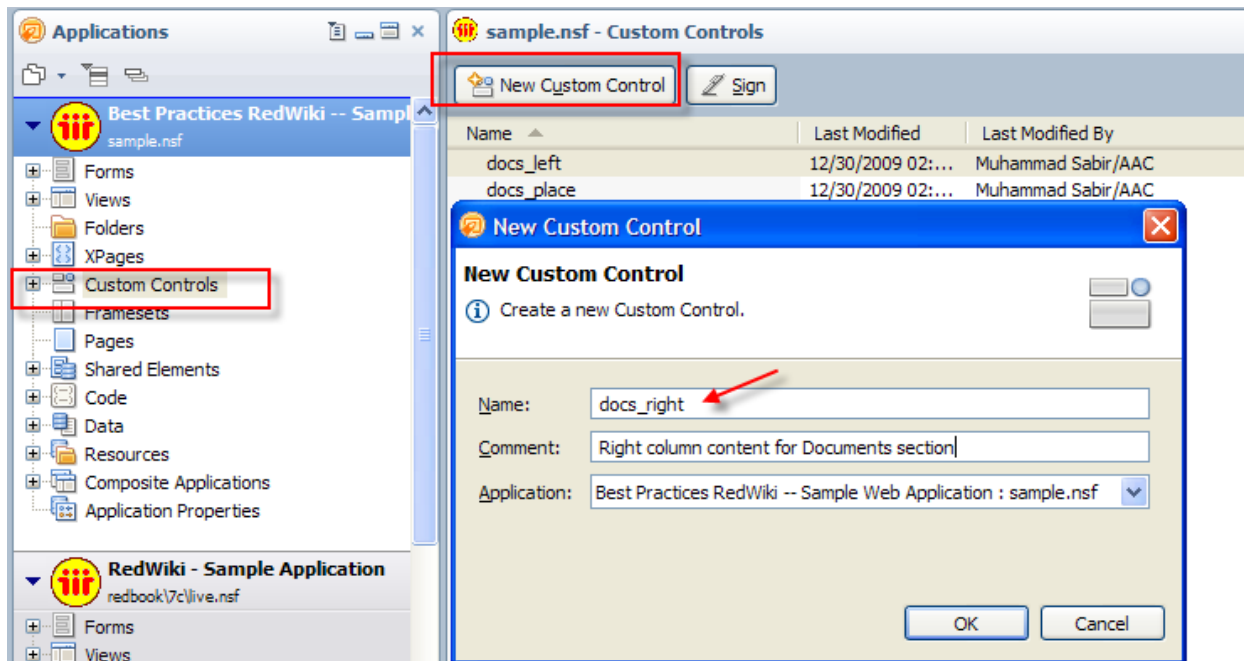
        </xc:global_tooltip>
    </xp:td>
</xp:tr>
<xp:tr styleClass="lotusFormFieldRow">
    <xp:td></xp:td>
    <xp:td>
        <xp:fileDownload rows="30" id="fileDownload1"
            displayLastModified="false" allowDelete="true"
hideWhen="true"

        value="#{dominoDocument1.body}"></xp:fileDownload>
    </xp:td>
    <xp:td></xp:td>
</xp:tr>
<xp:tr styleClass="lotusFormFieldRow">
    <xp:td></xp:td>
    <xp:td>
        <xp:messages id="messages1"></xp:messages>
    </xp:td>
    <xp:td></xp:td>
</xp:tr>
</xp:table>
</xp:view>

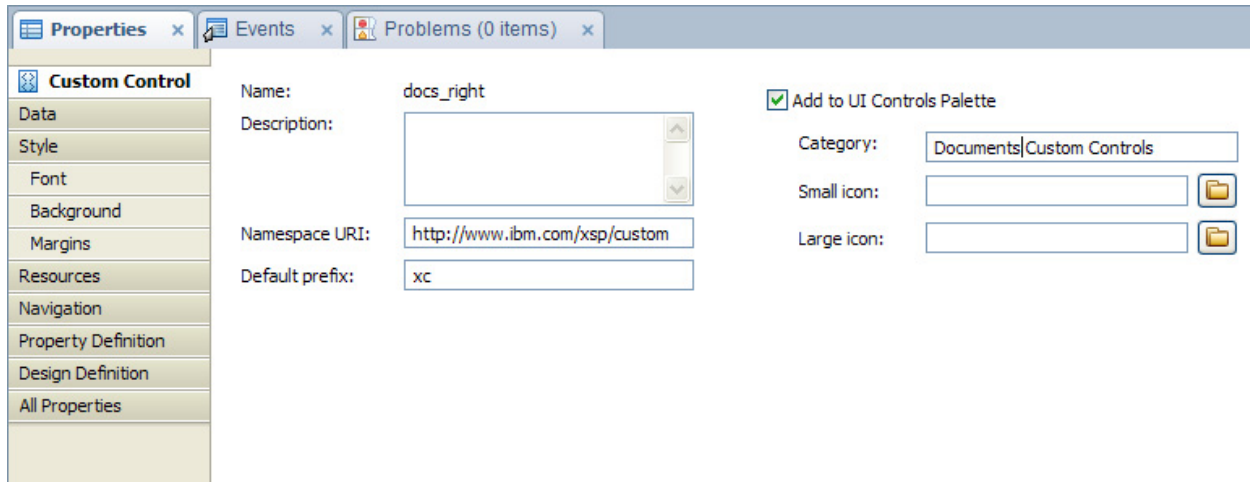
```

8.6.4: Creating custom control for right column content

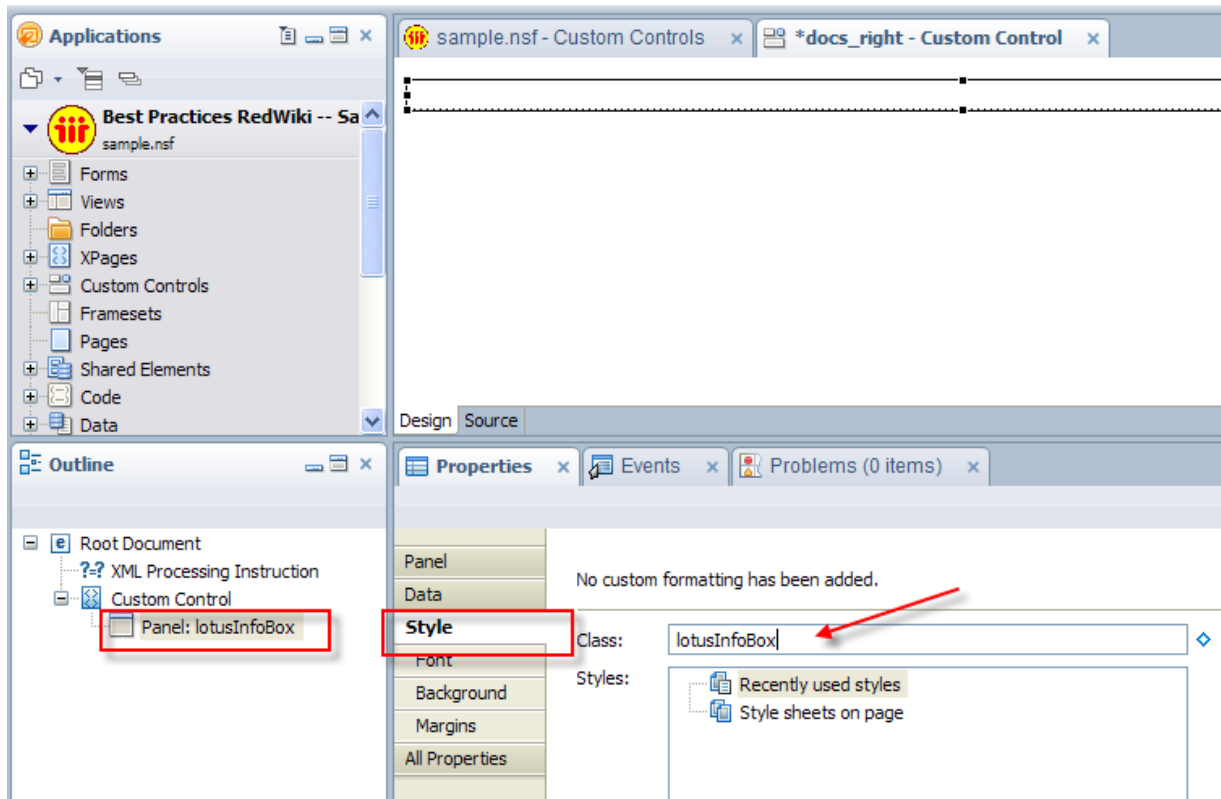
1. Create a new custom control "docs_right".



- Under the Properties tab, make sure Custom Control is selected and check **"Add to UI Controls Palette"** and enter **"Documents Custom Controls"** as the category. This moves this custom control under the category **"Documents Custom Controls"**.

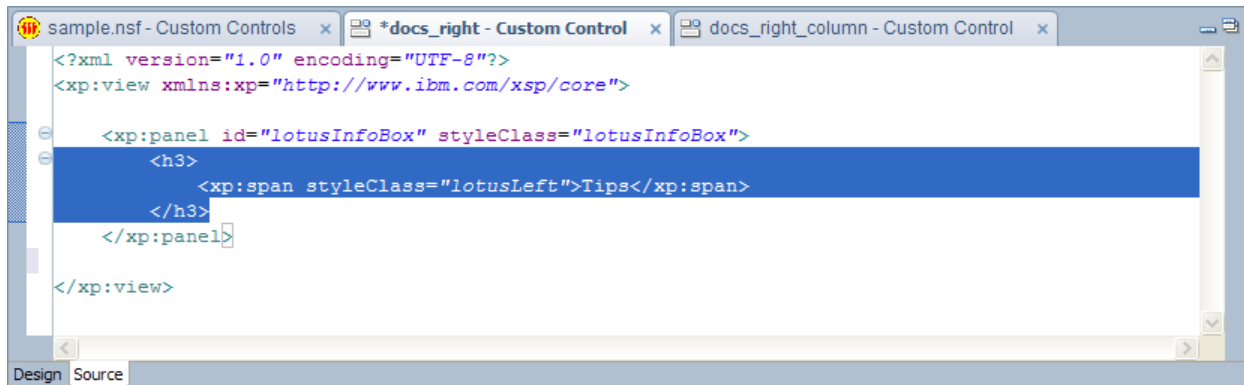


- Drag and drop a **"Panel"** control. Enter **"lotusInfoBox"** both as its name and CSS style class.



- Switch to **"Source"** tab and enter the following code within the **"lotusInfoBox"** panel:

```
<h3>
  <xp:span styleClass="lotusLeft">Tips</xp:span>
</h3>
```



5. Drag and drop a **Link** control within **<h3>** tag as shown in the screenshot. Set the following properties:

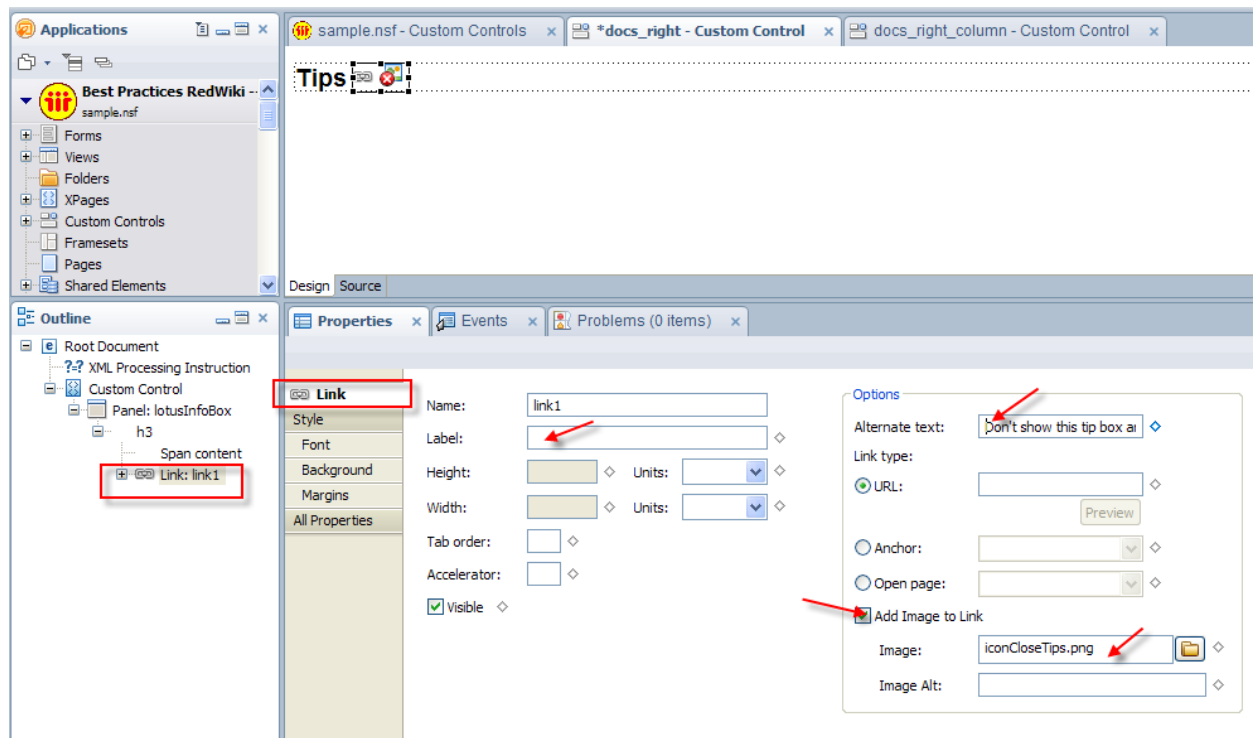
Label: (Remove the text to make it blank)

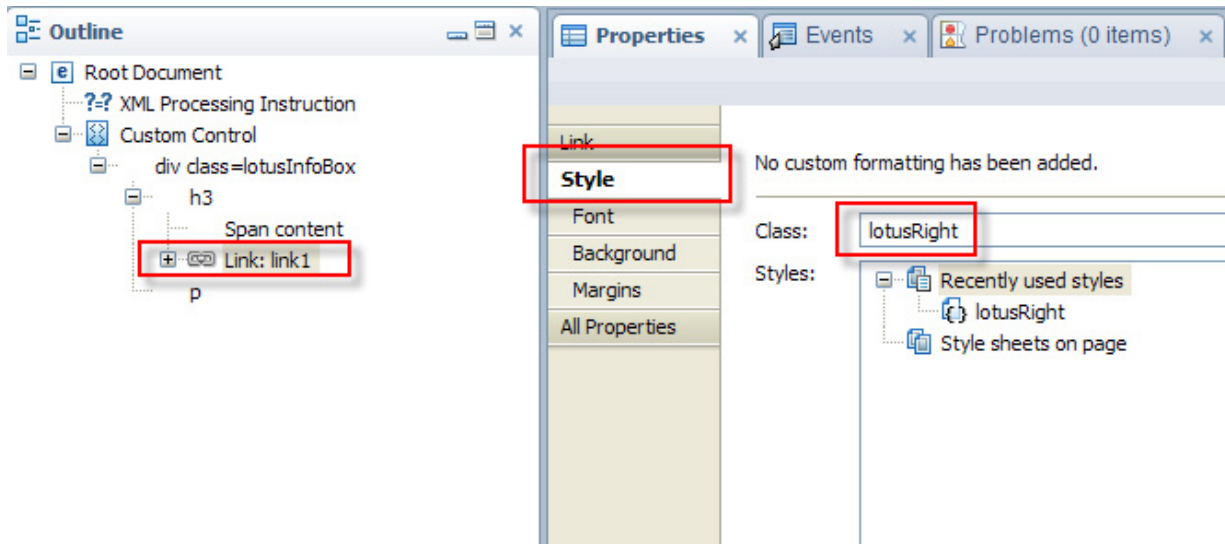
Alternate Text: Don't show this tip box anymore

Add to Image Link: (checked)

Image: iconCloseTips.png

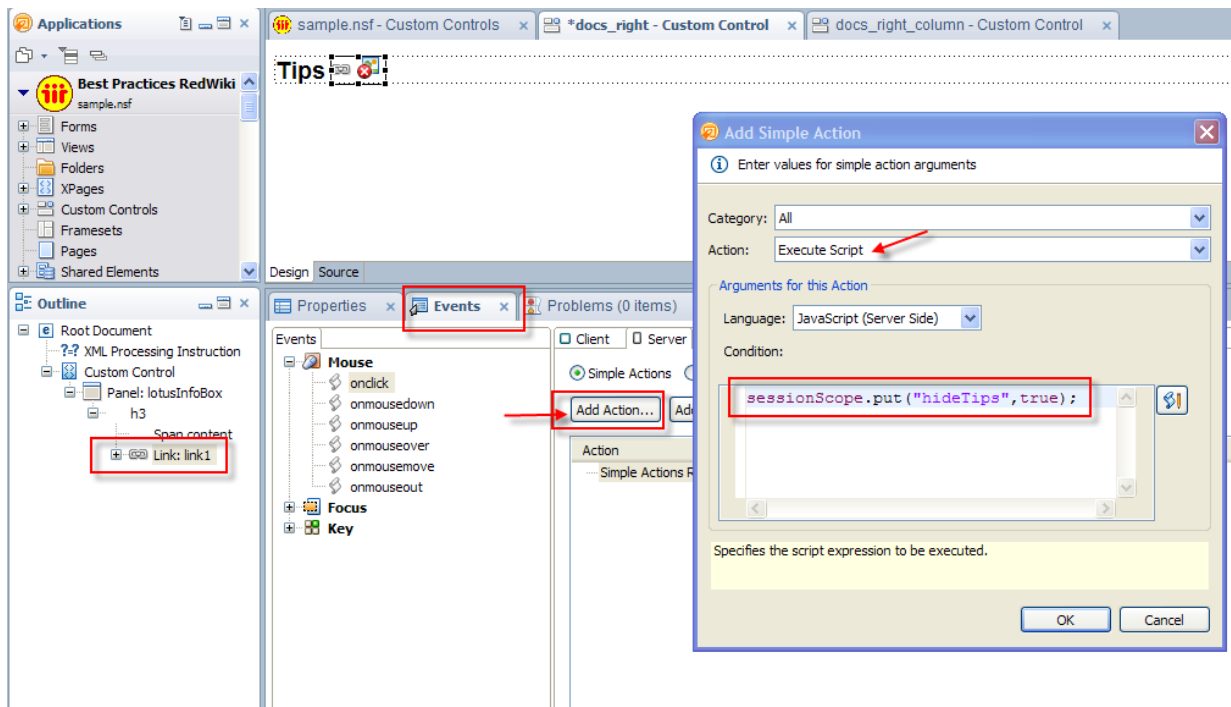
CSS Style Class: lotusRight



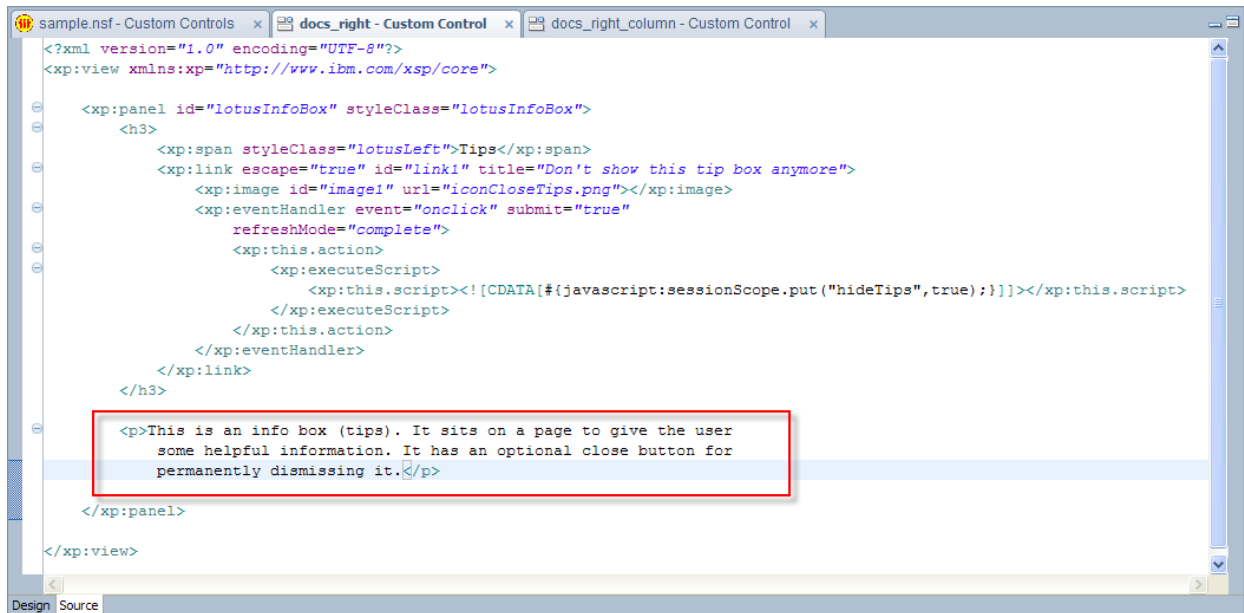


6. For **link1** control, select “**Events**” tab, and select “**onclick**” event. Click “**Add Action**” button and add the following code: `sessionScope.put("hideTips",true);`

This will set “**hideTips**” session scope variable to **true** when this link is clicked.



7. Switch to “**Source**” tab and enter the following code below `</h3>` tag:
`<p>Click on a document link to view and download it. To upload a document, click on ‘New Document’</p>`



8. Click on **"Source"** tab in XPages editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

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

    <xp:panel id="lotusInfoBox" styleClass="lotusInfoBox">
        <h3>
            <xp:span styleClass="lotusLeft">Tips</xp:span>
            <xp:link escape="true" id="link1" title="Don't show this
tip box anymore" styleClass="lotusRight">
                <xp:image id="image1"
url="iconCloseTips.png"></xp:image>
                <xp:eventHandler event="onclick" submit="true"
                    refreshMode="complete">
                    <xp:this.action>
                        <xp:executeScript>

                            <xp:this.script><![CDATA[#{javascript:sessionScope.put("hideTips",true)
;}}]></xp:this.script>

                        </xp:executeScript>
                    </xp:this.action>
                </xp:eventHandler>
            </xp:link>
        </h3>

        <p>This is an info box (tips). It sits on a page to give the user
            some helpful information. It has an optional close button
            for permanently dismissing it.</p>

    </xp:panel>
```

```
</xp:view>
```

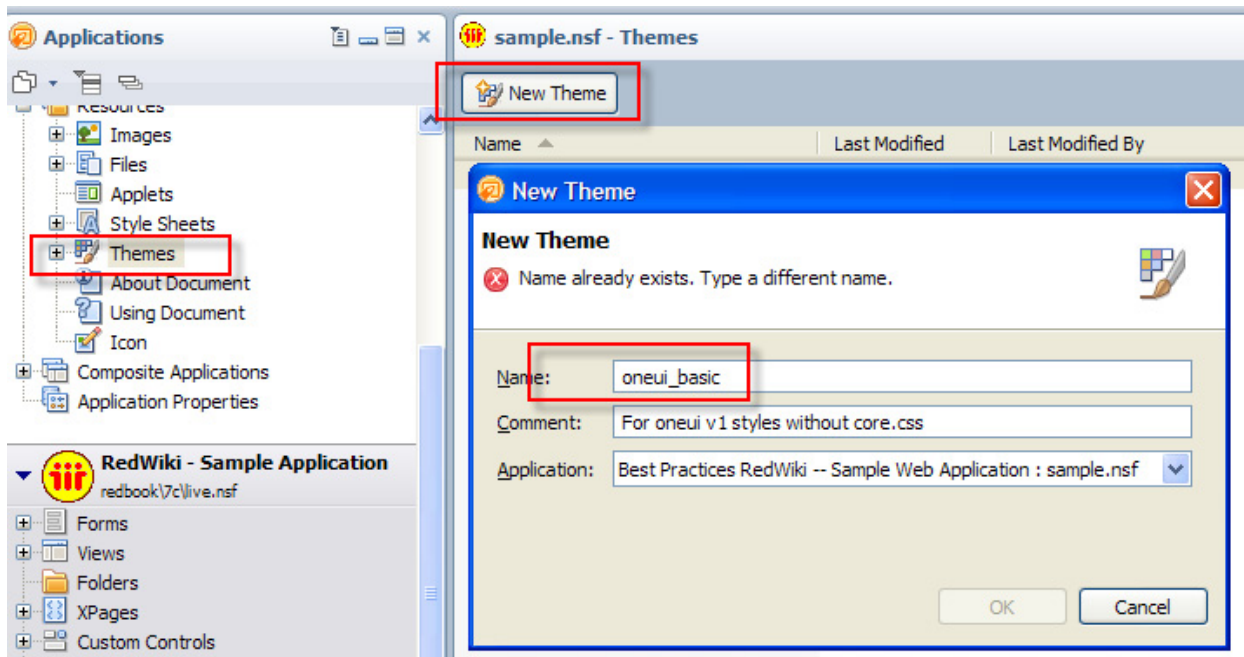
Step 9 – Sample 2: Creating the XPage for documents section

So far, we have been building the custom controls. In this section we are going to build the XPage to hold these custom controls in a specific order. Since we are including the main content dynamically (based on query string parameter), we will have only one XPage for the entire “document” section within the sample Intranet application.

9.1: Updating themes for better styling

Before we actually build the XPage, we need to update our themes for better styling. Note that we used version 2 of One UI. Since we did not extend themes from version 1, we don’t have some of the styling classes available – especially those related with the view and table display. If we simply extend version 1, all referenced CSS files get included in the theme (which is not what we want). Therefore, we will just copy-paste One UI version 1 theme but remove the core CSS file (core.css) reference from it.

1. Create a new theme and name it as “**oneui_basic**”.



2. One UI theme is already included in Domino server and designer. You can find it by searching for oneui.theme document under the install folder. By default it is located at the following location in Domino designer:

C:\Program Files\IBM\Lotus\Domino\xsp\nsf\themes\oneui.theme

- Open the theme document and remove all the XML content before this comment:

```
<!--
  TMG: The following xsp.css contains XSP Specific overrides and
  extensions to the above linked OneUI library.
-->
```

The screenshot shows a Notepad++ window titled "C:\Program Files\IBM\Lotus\Domino\xsp\nsf\themes\oneui.theme - Notepad++". The XML content is as follows:

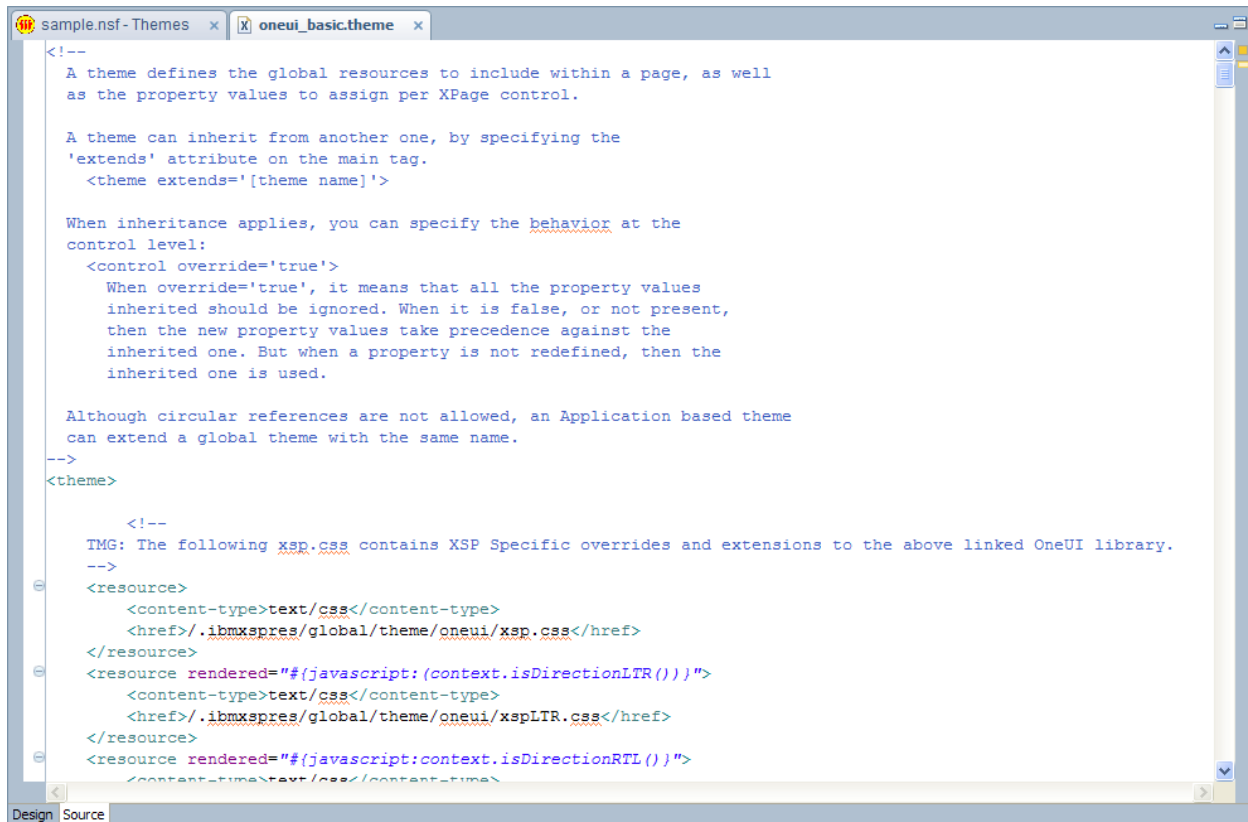
```

31 <!--
32 TMG: IBM OneUI R2v1.6 - The following three .css are "out-of-the-box" from the OneUI library.
33 THEREFORE DO NOT MODIFY THESE FILES - INSTEAD EXTEND/OVERRIDE USING XSP.CSS BELOW
34 -->
35 <resource>
36   <content-type>text/css</content-type>
37   <href>/ibmxxspres/global/theme/oneui/core.css</href>
38 </resource>
39 <resource rendered="#{javascript:context.isDirectionLTR()}">
40   <content-type>text/css</content-type>
41   <href>/ibmxxspres/global/theme/oneui/coreLTR.css</href>
42 </resource>
43 <resource rendered="#{javascript:context.isDirectionRTL()}">
44   <content-type>text/css</content-type>
45   <href>/ibmxxspres/global/theme/oneui/coreRTL.css</href>
46 </resource>
47
48 <resource>
49   <content-type>text/css</content-type>
50   <href>/ibmxxspres/global/theme/oneui/defaultTheme.css</href>
51 </resource>
52 <resource rendered="#{javascript:(context.isDirectionLTR())}">
53   <content-type>text/css</content-type>
54   <href>/ibmxxspres/global/theme/oneui/defaultThemeLTR.css</href>
55 </resource>
56 <resource rendered="#{javascript:context.isDirectionRTL()}">
57   <content-type>text/css</content-type>
58   <href>/ibmxxspres/global/theme/oneui/defaultThemeRTL.css</href>
59 </resource>
60
61 <resource rendered="#{javascript:context.getUserAgent().isIE(0,6) == true}">
62   <content-type>text/css</content-type>
63   <href>/ibmxxspres/global/theme/oneui/ie hacks.css</href>
64 </resource>
65 <resource rendered="#{javascript:(context.getUserAgent().isIE(0,6) == true &amp; (context.isDirectionLTR()))}">
66   <content-type>text/css</content-type>
67   <href>/ibmxxspres/global/theme/oneui/ie hacksLTR.css</href>
68 </resource>
69 <resource rendered="#{javascript:(context.isDirectionRTL() &amp; context.getUserAgent().isIE(0,6))}">
70   <content-type>text/css</content-type>
71   <href>/ibmxxspres/global/theme/oneui/ie hacksRTL.css</href>
72 </resource>
73
74 <!--
75 TMG: The following xsp.css contains XSP Specific overrides and extensions to the above linked OneUI library.
76 -->
77 <resource>
78   <content-type>text/css</content-type>
79   <href>/ibmxxspres/global/theme/oneui/xsp.css</href>
80 </resource>

```

A red box highlights the content from line 35 to line 72. A callout bubble points to this box with the text "Remove all of these".

- After removing the One UI version 1 specific content, copy-paste the XML to oneui_basic theme document.



- Below is the code that we need to copy over. As you can see it, it is specific to extensions and controls, but we have removed the version 1 related CSS resources.

```

<!--
A theme defines the global resources to include within a page, as well
as the property values to assign per XPage control.

A theme can inherit from another one, by specifying the
'extends' attribute on the main tag.
  <theme extends='[theme name] '>

When inheritance applies, you can specify the behavior at the
control level:
  <control override='true'>
    When override='true', it means that all the property values
    inherited should be ignored. When it is false, or not present,
    then the new property values take precedence against the
    inherited one. But when a property is not redefined, then the
    inherited one is used.

Although circular references are not allowed, an Application based theme
can extend a global theme with the same name.
-->
<theme>

  <!--

```

TMG: The following xsp.css contains XSP Specific overrides and extensions to the above linked OneUI library.

```
-->
<resource>
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xsp.css</href>
</resource>
<resource rendered="#{javascript:(context.isDirectionLTR())}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspLTR.css</href>
</resource>
<resource rendered="#{javascript:context.isDirectionRTL()}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspRTL.css</href>
</resource>

<!-- IE Specific -->
<resource rendered="#{javascript:context.getUserAgent().isIE(0,6)}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspIE06.css</href>
</resource>
<resource rendered="#{javascript:context.getUserAgent().isIE(7,8)}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspIE78.css</href>
</resource>
<resource rendered="#{javascript:(context.isDirectionRTL() & amp; context.getUserAgent().isIE())}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspIERTL.css</href>
</resource>

<!-- FireFox Specific -->
<resource rendered="#{javascript:context.getUserAgent().isFirefox()}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspFF.css</href>
</resource>

<!-- Safari Specific -->
<resource rendered="#{javascript:context.getUserAgent().isSafari()}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspSF.css</href>
</resource>

<!-- RCP Specific -->
<resource rendered="#{javascript:context.isRunningContext('Notes')}">
  <content-type>text/css</content-type>
  <href>/ibmxxspres/global/theme/oneui/xspRCP.css</href>
</resource>
<!--
=====
Global controls
=====
-->

<!-- View Root (Page Body) -->
<control>
  <name>ViewRoot</name>
```

```

        <property mode="concat">
            <name>styleClass</name>
            <value>xspView tundra</value>
        </property>
    </control>

<!-- Form -->
<control>
    <name>Form</name>
    <property>
        <name>styleClass</name>
        <value>lotusForm</value>
    </property>
</control>

<!--
=====
    Ouput text based controls
=====
-->

<!-- Basic Text -->
<control>
    <name>Text</name>
    <property>
        <name>styleClass</name>
        <value>xspText</value>
    </property>
</control>

<!-- Computed field -->
<control>
    <name>Text.ComputedField</name>
    <property>
        <name>styleClass</name>
        <value>xspTextComputedField</value>
    </property>
</control>

<!-- Label -->
<control>
    <name>Text.Label</name>
    <property>
        <name>styleClass</name>
        <value>xspTextLabel</value>
    </property>
</control>

<!-- View Title -->
<control>
    <name>Text.ViewTitle</name>
    <property>
        <name>styleClass</name>
        <value>xspTextViewTitle</value>
    </property>
</control>

```

```

<!--
=====
      Input text based controls
=====
-->

<!-- Basic Input Text -->
<control>
  <name>InputField</name>
  <property>
    <name>styleClass</name>
    <value>xspInputField</value>
  </property>
</control>

<!-- Standard Edit Box -->
<control>
  <name>InputField.EditBox</name>
  <property>
    <name>styleClass</name>
    <value>xspInputFieldEditBox</value>
  </property>
</control>

<!-- Password Edit Box -->
<control>
  <name>InputField.Secret</name>
  <property>
    <name>styleClass</name>
    <value>xspInputFieldSecret</value>
  </property>
</control>

<!-- Date Time Picker -->
<control>
  <name>InputField.DateTimePicker</name>
  <property mode="concat">
    <name>styleClass</name>
    <value>xspInputFieldDateTimePicker</value>
  </property>
</control>

<!-- TextArea -->
<control>
  <name>InputField.TextArea</name>
  <property>
    <name>styleClass</name>
    <value>xspInputFieldTextArea</value>
  </property>
</control>

<!-- Rich Text -->
<control>
  <name>InputField.RichText</name>
  <property mode="concat">
    <name>styleClass</name>
    <value>domino-richtext xspInputFieldRichText</value>
  </property>
</control>

```

```

        </property>
    </control>

    <!-- Hidden Field -->
    <control>
        <name>InputField.Hidden</name>
        <property>
            <name>styleClass</name>
            <value>xspInputFieldHidden</value>
        </property>
    </control>

    <!-- File Upload -->
    <control>
        <name>InputField.FileUpload</name>
        <property>
            <name>styleClass</name>
            <value>xspInputFieldFileUpload</value>
        </property>
    </control>

    <!--
=====
        File Download
=====
-->

    <!-- File Download Table -->
    <control>
        <name>DataTable.FileDownload</name>
        <property>
            <name>styleClass</name>
            <value>xspDataTableFileDownload</value>
        </property>
        <property>
            <name>typeStyleClass</name>
            <value>xspDataTableFileDownloadType</value>
        </property>
        <property>
            <name>sizeStyleClass</name>
            <value>xspDataTableFileDownloadSize</value>
        </property>
        <property>
            <name>nameStyleClass</name>
            <value>xspDataTableFileDownloadName</value>
        </property>
        <property>
            <name>createdStyleClass</name>
            <value>xspDataTableFileDownloadCreated</value>
        </property>
        <property>
            <name>modifiedStyleClass</name>
            <value>xspDataTableFileDownloadModified</value>
        </property>
        <property>
            <name>deleteStyleClass</name>
            <value>xspDataTableFileDownloadDelete</value>

```



```

        </property>
        <property>
        <name>captionStyleClass</name>
        <value>xspDataTableFileDownloadCaption</value>
        </property>
    </control>

<!-- File Download Link -->
<control>
    <name>Link.FileDownload</name>
    <property>
        <name>styleClass</name>
        <value>xspLinkFileDownload</value>
    </property>
</control>

<!--
=====
                Link controls
=====
-->

<!-- Basic Link -->
<control>
    <name>Link</name>
    <property>
        <name>styleClass</name>
        <value>xspLink</value>
    </property>
</control>

<!--
=====
                Button controls
=====
-->

<!-- Basic Button -->
<control>
    <name>Button</name>
    <property>
        <name>styleClass</name>
        <value>xspButtonNative</value>
    </property>
</control>

<!-- Command Button -->
<control>
    <name>Button.Command</name>
    <property>
        <name>styleClass</name>
        <value>lotusFormButton</value>
        <!-- default button falls back to Button.Command
stylekitfamily -->
        <!-- <value>xspButtonCommand</value> -->
    </property>
</control>

```

```

<!-- Submit Button -->
<control>
  <name>Button.Submit</name>
  <property>
    <name>styleClass</name>
    <value>xspButtonSubmit</value>
  </property>
</control>

<!-- Cancel Button -->
<control>
  <name>Button.Cancel</name>
  <property>
    <name>styleClass</name>
    <value>xspButtonCancel</value>
  </property>
</control>

<!--
=====
      CheckBox controls
=====
-->

<!-- Basic CheckBox -->
<control>
  <name>CheckBox</name>
  <property>
    <name>styleClass</name>
    <value>xspCheckBox</value>
  </property>
</control>

<!--
=====
      RadioButton controls
=====
-->

<!-- Basic RadioButton -->
<control>
  <name>RadioButton</name>
  <property>
    <name>styleClass</name>
    <value>xspRadioButton</value>
  </property>
</control>

<!--
=====
      ListBox controls
=====
-->

<!-- Basic ListBox -->
<control>

```

```

        <name>ListBox</name>
        <property>
            <name>styleClass</name>
            <value>xspListBox</value>
        </property>
    </control>

    <!--
=====
        ComboBox controls
=====
-->

    <!-- Basic ComboBox -->
    <control>
        <name>ComboBox</name>
        <property>
            <name>styleClass</name>
            <value>xspComboBox</value>
        </property>
    </control>

    <!--
=====
        Image controls
=====
-->

    <!-- Basic Image -->
    <control>
        <name>Image</name>
        <property>
            <name>styleClass</name>
            <value>xspImage</value>
        </property>
    </control>

    <!--
=====
        Message controls
=====
-->

    <!-- Basic Message -->
    <control>
        <name>Message</name>
        <property>
            <name>styleClass</name>
            <value>xspMessage</value>
        </property>
    </control>

    <!-- Basic Messages -->
    <control>
        <name>Messages</name>
        <property>
            <name>styleClass</name>
```

```

        <value>xspMessages</value>
    </property>
</control>

<!--
=====
    Panel controls
=====
-->

<!-- Basic Panel -->
<!--
<control>
    <name>Panel</name>
    <property>
        <name>styleClass</name>
        <value>xspPanel</value>
    </property>
</control>
-->

<!--
=====
    Section controls
=====
-->

<!-- Basic Section
<control>
    <name>Section</name>
    <property>
        <name>styleClass</name>
        <value>xspSection</value>
    </property>
</control>
-->

<!--
=====
    DataTable controls
    A table column has some virtual properties when dealing with
themes:
        startStyleClass: the style class to apply to the first
column
        middleStyleClass: the style class to apply to the column
that are neither first or last
        endStyleClass: the style class to apply to the last column
    Those properties take priority over the styleClass when defined
in a theme
=====
-->

<!-- Basic DataTable -->
<control>
    <name>DataTable</name>
    <property>
        <name>styleClass</name>

```

```

        <value>lotusTable</value>
    </property>
</property>
    <name>unreadMarksClass</name>
    <value>xspDataTableRowUnread</value>
</property>
</property>
    <name>readMarksClass</name>
    <value>xspDataTableRowRead</value>
</property>
</property>
    <name>captionStyleClass</name>
    <value>xspDataTableCaption</value>
</property>
</control>

<!-- Basic Table Column -->
<control>
    <name>Column</name>
    <property>
        <name>styleClass</name>
        <value>xspColumn</value>
    </property>
</control>

<!--
=====
View controls
=====
-->

<!-- ===== View Table ===== -->
<!-- View DataTable -->
<control>
    <name>DataTable.ViewPanel</name>
    <property>
        <name>viewStyleClass</name>
        <value>xspDataTableViewPanel</value>
    </property>
    <property>
        <name>dataTableStyleClass</name>
        <value>xspDataTable</value>
    </property>
    <property>
        <name>headerEndStyleClass</name>
        <value>xspDataTableViewPanelHeaderEnd</value>
    </property>
    <property>
        <name>headerStartStyleClass</name>
        <value>xspDataTableViewPanelHeaderStart</value>
    </property>
    <property>
        <name>footerStartStyleClass</name>
        <value>xspDataTableViewPanelFooterStart</value>
    </property>
    <property>
        <name>footerEndStyleClass</name>

```

```

        <value>xspDataTableViewPanelFooterEnd</value>
    </property>
</property>
    <name>footerStyleClass</name>
    <value>xspDataTableViewPanelFooterMiddle</value>
</property>
</property>
    <name>headerStyleClass</name>
    <value>xspDataTableViewPanelHeaderMiddle</value>
</property>
</property>
    <name>unreadMarksClass</name>
    <value>xspDataTableRowUnread</value>
</property>
</property>
    <name>readMarksClass</name>
    <value>xspDataTableRowRead</value>
</property>
</property>
    <name>captionStyleClass</name>
    <value>xspDataTableViewPanelCaption</value>
</property>
</control>

<!-- ===== View Column ===== -->
>

<!-- View Column -->
<control>
    <name>Column.View</name>
    <property>
        <name>startStyleClass</name>
        <value>xspColumnViewStart</value>
    </property>
    <property>
        <name>middleStyleClass</name>
        <value>xspColumnViewMiddle</value>
    </property>
    <property>
        <name>endStyleClass</name>
        <value>xspColumnViewEnd</value>
    </property>
    <property>
        <name>numericStyleClass</name>
        <value>xspColumnViewNumeric</value>
    </property>
    <property>
        <name>viewCheckboxStyle</name>
        <value>margin:0px 2px;height:12px;</value>
    </property>
</control>

<!-- View Column text -->
<control>
    <name>Text.ViewColumn</name>
    <property>
        <name>styleClass</name>

```

```

        <value>xspTextViewColumn</value>
    </property>
</control>

<!-- View Column computed text -->
<control>
    <name>Text.ViewColumnComputed</name>
    <property>
        <name>styleClass</name>
        <value>xspTextViewColumnComputed</value>
    </property>
</control>

<!-- View Column link -->
<control>
    <name>Link.ViewColumn</name>
    <property>
        <name>styleClass</name>
        <value>xspLinkViewColumn</value>
    </property>
</control>

<!-- View Column image -->
<control>
    <name>Image.ViewColumn</name>
    <property>
        <name>styleClass</name>
        <value>xspImageViewColumn</value>
    </property>
</control>

<!-- View Column checkbox -->
<control>
    <name>CheckBox.ViewColumn</name>
    <property>
        <name>styleClass</name>
        <value>xspCheckBoxViewColumn</value>
    </property>
</control>

<!-- ===== View Column Header
===== -->

<!-- View Column Header -->
<control>
    <name>Panel.ViewColumnHeader</name>
    <property>
        <name>styleClass</name>
        <value>xspPanelViewColumnHeader</value>
    </property>
</control>

<!-- View Column Header text -->
<control>
    <name>Text.ViewColumnHeader</name>
    <property>
        <name>styleClass</name>

```

```

        <value>xspTableHead</value>
    </property>
</property>
    <name>style</name>
    <value>white-space:nowrap;</value>
</property>
</property>
    <name>checkboxStyle</name>
    <value>margin:0px 2px;height:12px;</value>
</property>
</control>

<!-- View Column Header link -->
<control>
    <name>Link.ViewColumnHeader</name>
    <property>
        <name>styleClass</name>
        <value>xspLinkViewColumnHeader</value>
    </property>
</control>

<!-- View Column Header checkbox -->
<control>
    <name>CheckBox.ViewColumnHeader</name>
    <property>
        <name>styleClass</name>
        <value>xspCheckBoxViewColumnHeader</value>
    </property>
</control>

<!-- View Column Header image -->
<control>
    <name>Image.ViewColumnHeader</name>
    <property>
        <name>styleClass</name>
        <value>xspImageViewColumnHeader</value>
    </property>
</control>

<!-- View Column Header icon image -->
<control>
    <name>Image.ViewColumnHeaderIcon</name>
    <property>
        <name>styleClass</name>
        <value>xspImageViewColumnHeaderIcon</value>
    </property>
</control>

<!-- View Column Header sort image -->
<control>
    <name>Image.ViewColumnHeaderSort</name>
    <property>
        <name>styleClass</name>
        <value>xspImageViewColumnHeaderSort</value>
    </property>
</control>

```



```

<!--
=====
    Tab panel controls
    A tab has some virtual properties when dealing with themes:
        startStyleClass: the style class to apply to the first tab
        middleStyleClass: the style class to apply to the tab that
are neither first or last
        endStyleClass: the style class to apply to the tab column
    Those properties take priority over the styleClass when defined
in a theme
=====
-->

<!-- Tabbed Panel
<control>
    <name>TabbedPanel</name>
    <property>
        <name>styleClass</name>
        <value>lotusTabs</value>
    </property>
    <property>
        <name>separatorStyleClass</name>
        <value>xspTabbedPanelContentSeparator</value>
    </property>
    <property>
        <name>containerStyleClass</name>
        <value>lotusTabContainer</value>
    </property>
    <property>
        <name>outerStyleClass</name>
        <value>xspTabbedPanelOuter</value>
    </property>
</control>
-->
<!-- Tab
<control>
    <name>Tab.TabbedPanel</name>
    <property>
        <name>styleClass</name>
        <value>xspTabTabbedPanel</value>
    </property>
    <property>
        <name>selectedTabStyleClass</name>
        <value>lotusSelected</value>
    </property>
    <property>
        <name>unselectedTabStyleClass</name>
        <value>xspUnselectedTab</value>
    </property>
    <property>
        <name>startTabStyleClass</name>
        <value>xspStartTab</value>
    </property>
    <property>
        <name>middleTabStyleClass</name>
        <value>xspMiddleTab</value>
    </property>

```

```

        <property>
            <name>endTabStyleClass</name>
            <value>xspEndTab</value>
        </property>
    </control>
    -->

<!-- ===== Pager / PagerControl
===== -->

<!-- Pager -->
<control>
    <name>Pager</name>
    <property>
        <name>outerStyleClass</name>
        <value>xspPagerContainer</value>
    </property>
    <property>
        <name>styleClass</name>
        <value>xspPager</value>
    </property>
</control>

<!-- PagerControl -->
<control>
    <name>PagerControl.Pager</name>
    <property>
        <name>styleClass</name>
        <value>xspPagerNav</value>
    </property>
</control>

<control>
    <name>PagerControl.Pager.First</name>
    <property>
        <name>styleClass</name>
        <value>xspPagerNav xspFirst</value>
    </property>
</control>

<control>
    <name>PagerControl.Pager.Previous</name>
    <property>
        <name>styleClass</name>
        <value>xspPagerNav xspPrevious</value>
    </property>
</control>

<control>
    <name>PagerControl.Pager.Next</name>
    <property>
        <name>styleClass</name>
        <value>xspPagerNav xspNext</value>
    </property>
</control>

<control>

```

```

        <name>PagerControl.Pager.Last</name>
        <property>
            <name>styleClass</name>
            <value>xspPagerNav xspLast</value>
        </property>
    </control>

    <control>
        <name>PagerControl.Pager.Group</name>
        <property>
            <name>styleClass</name>
            <value>xspPagerNav xspGroup</value>
        </property>
        <property>
            <name>currentStyleClass</name>
            <value>xspCurrentItem</value>
        </property>
        <property>
            <name>firstStyleClass</name>
            <value>xspFirstItem</value>
        </property>
        <property>
            <name>lastStyleClass</name>
            <value>xspLastItem</value>
        </property>
    </control>

    <control>
        <name>PagerControl.Pager.Status</name>
        <property>
            <name>styleClass</name>
            <value>xspPagerNav xspStatus</value>
        </property>
    </control>

    <control>
        <name>PagerControl.Pager.Goto</name>
        <property>
            <name>styleClass</name>
            <value>xspPagerNav xspGoto</value>
        </property>
    </control>

    <control>
        <name>PagerControl.Pager.Separator</name>
        <property>
            <name>styleClass</name>
            <value>xspPagerNav xspSeparator</value>
        </property>
    </control>

    <!--
=====
    HTML passthrough controls
=====
-->
    <!--

```

```

<control>
  <name>HtmlDiv</name>
  <property>
    <name>styleClass</name>
    <value>xspHtmlDiv</value>
  </property>
</control>

<control>
  <name>HtmlBr</name>
  <property>
    <name>styleClass</name>
    <value>xspHtmlBr</value>
  </property>
</control>

<control>
  <name>HtmlP</name>
  <property>
    <name>styleClass</name>
    <value>xspHtmlP</value>
  </property>
</control>

<control>
  <name>HtmlSpan</name>
  <property>
    <name>styleClass</name>
    <value>xspHtmlSpan</value>
  </property>
</control>

<control>
  <name>HtmlTable</name>
  <property>
    <name>styleClass</name>
    <value>xspHtmlTable</value>
  </property>
  <property>
    <name>captionStyleClass</name>
    <value>xspHtmlTableCaption</value>
  </property>
</control>

<control>
  <name>HtmlTd</name>
  <property>
    <name>styleClass</name>
    <value>xspHtmlTd</value>
  </property>
</control>

<control>
  <name>HtmlTr</name>
  <property>
    <name>styleClass</name>
    <value>xspHtmlTr</value>
  </property>
</control>

```

```

        </property>
    </control>
-->

<!--
=====
        Misc
=====
-->

<!-- Client Script tag -->
<control>
    <name>Script</name>
    <property>
        <name>styleClass</name>
        <value>xspScript</value>
    </property>
</control>

</theme>

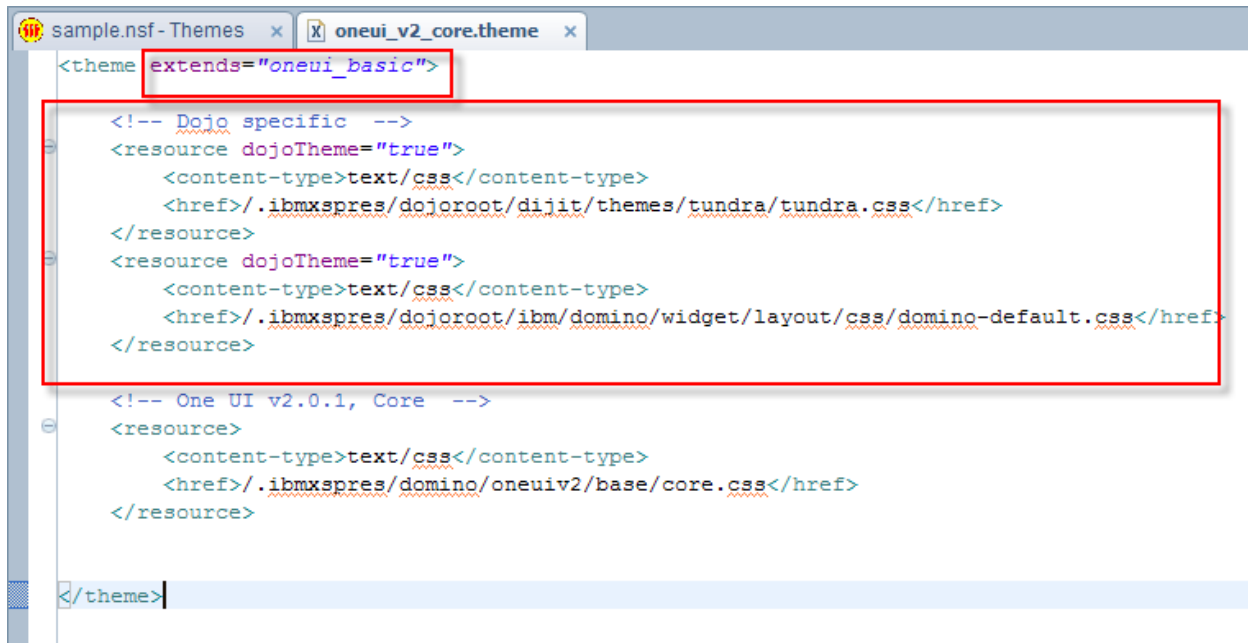
```

6. Save “**oneui_basic**” theme created in last step. Now we need to make sure that our version 2 theme extends the resources from “**oneui_basic**” theme. Open “**oneui_v2_core**” theme and enter **extends=“oneui_basic”** as the attribute for <theme> tag. We need to add dojo related resources as well. Enter the following code after the theme tag:

```

<!-- Dojo specific -->
<resource dojoTheme="true">
    <content-type>text/css</content-type>
    <href>/.ibmxspres/dojoroot/dijit/themes/tundra/tundra.css</href>
</resource>
<resource dojoTheme="true">
    <content-type>text/css</content-type>
    <href>/.ibmxspres/dojoroot/ibm/domino/widget/layout/css/domino-
default.css</href>
</resource>

```



7. This is what “oneui_v2_core” theme document should look like:

```

<theme extends="oneui_basic">

  <!-- Dojo specific -->
  <resource dojoTheme="true">
    <content-type>text/css</content-type>
    <href>/.ibmxspres/dojo/dojo/dijit/themes/tundra/tundra.css</href>
  </resource>
  <resource dojoTheme="true">
    <content-type>text/css</content-type>
    <href>/.ibmxspres/dojo/dojo/ibm/domino/widget/layout/css/domino-
default.css</href>
  </resource>

  <!-- One UI v2.0.1, Core -->
  <resource>
    <content-type>text/css</content-type>
    <href>/.ibmxspres/dojo/dojo/oneui2/base/core.css</href>
  </resource>

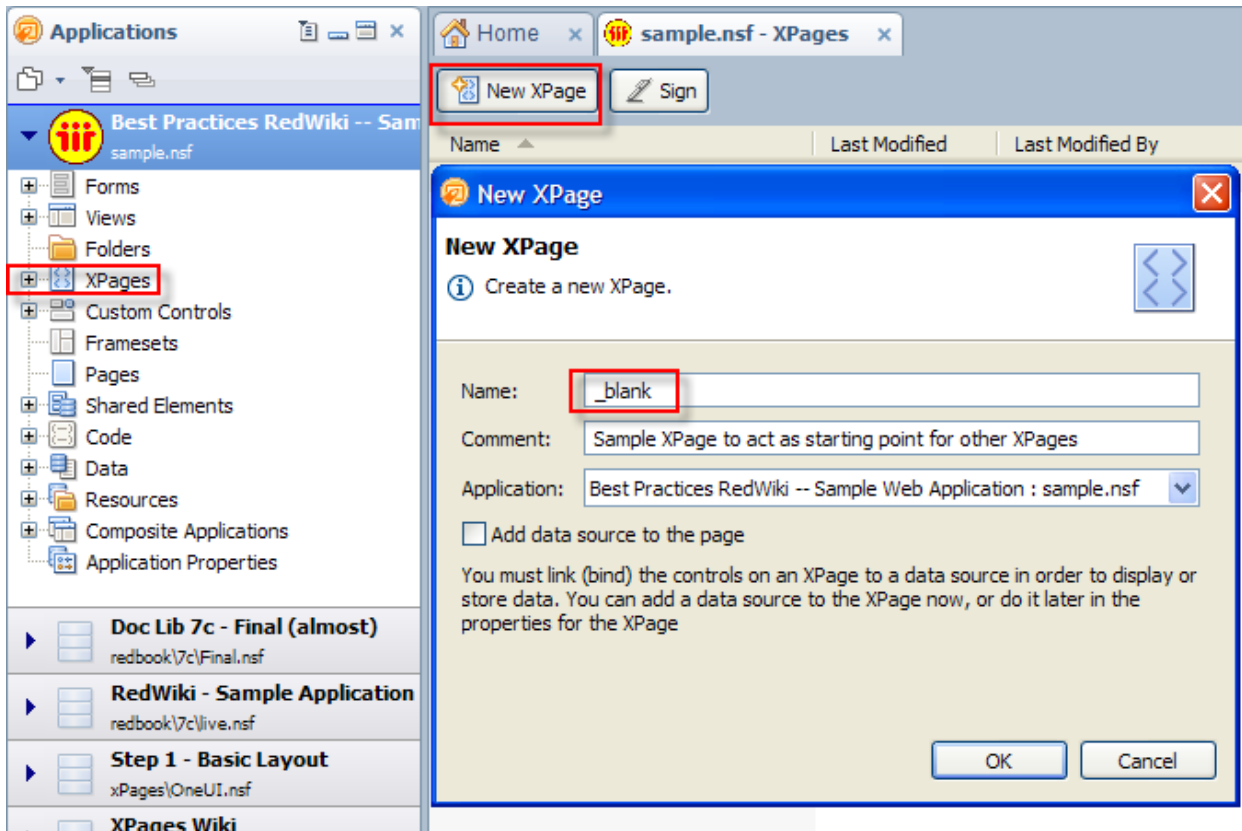
</theme>

```

Step 9.2: Creating a sample/bank XPage

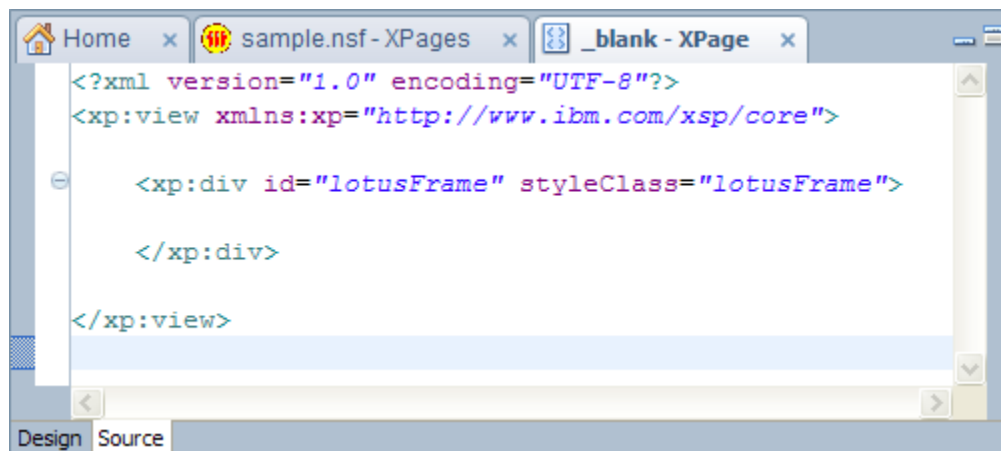
In this section, we are going to build a sample blank page with all the layout custom controls in the order they are supposed to be. This acts as starting point to build additional XPages by just copying this sample page.

1. Click on “**New XPage**” button and name the XPage as “**_blank**”.

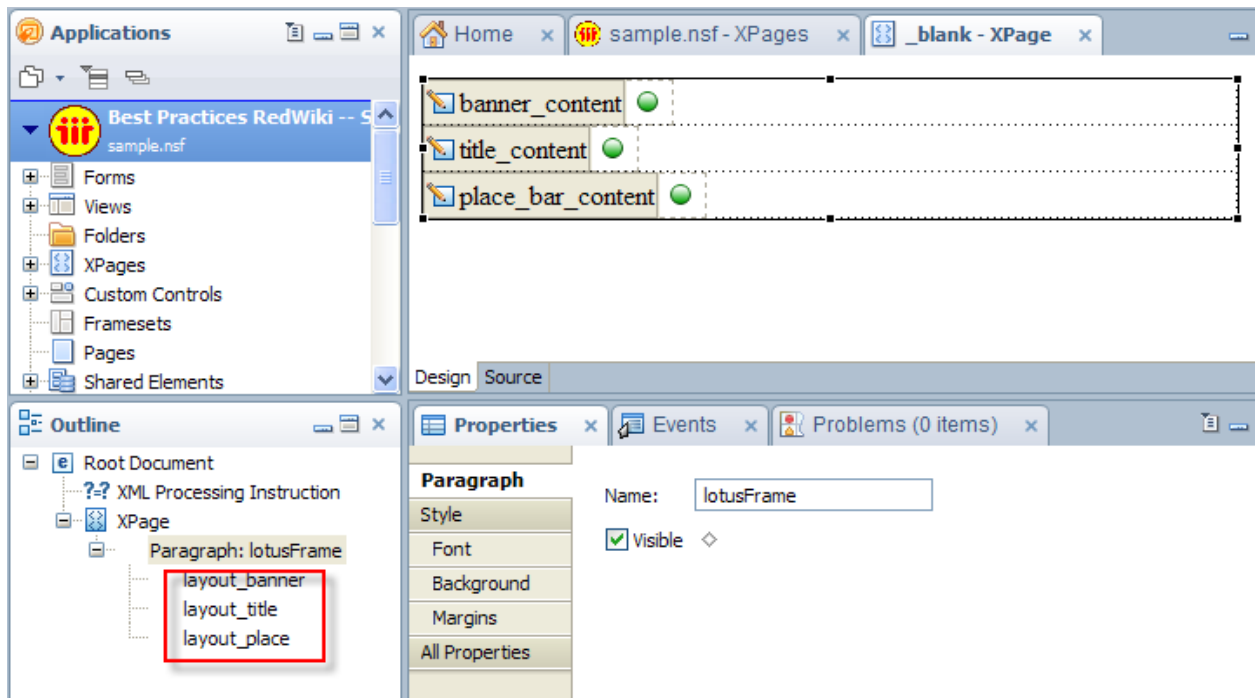


2. Switch to the Source tab and enter the following code. Note that in One UI, all elements are contained within the parent <DIV> container named “lotusFrame”.

```
<xp:div id="lotusFrame" styleClass="lotusFrame"></xp:div>
```

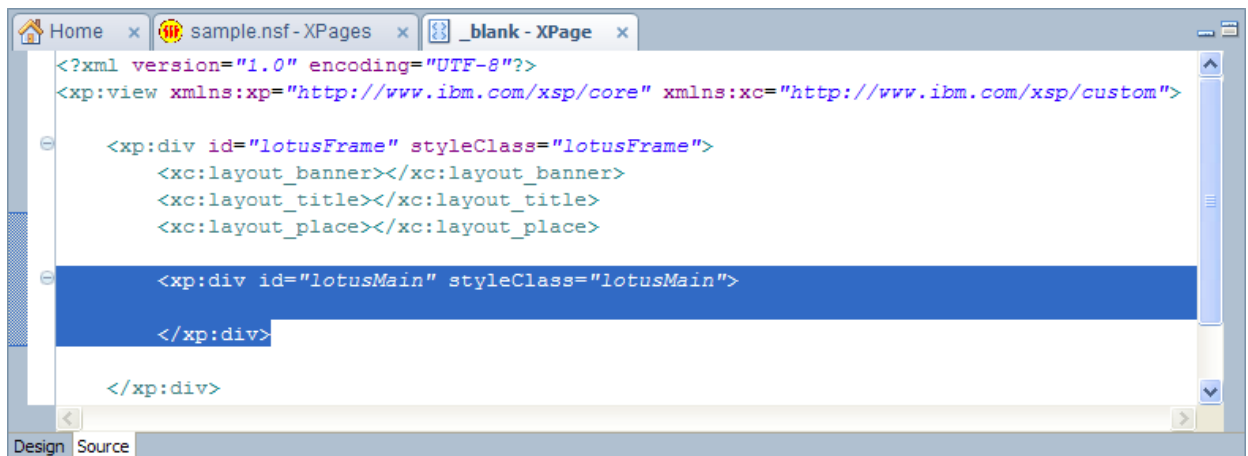


- Switch back to Design tab. Drag and drop the custom controls within the lotusFrame <DIV> element in the following order: layout_banner, layout_title, layout_place, as shown in the screenshot below.

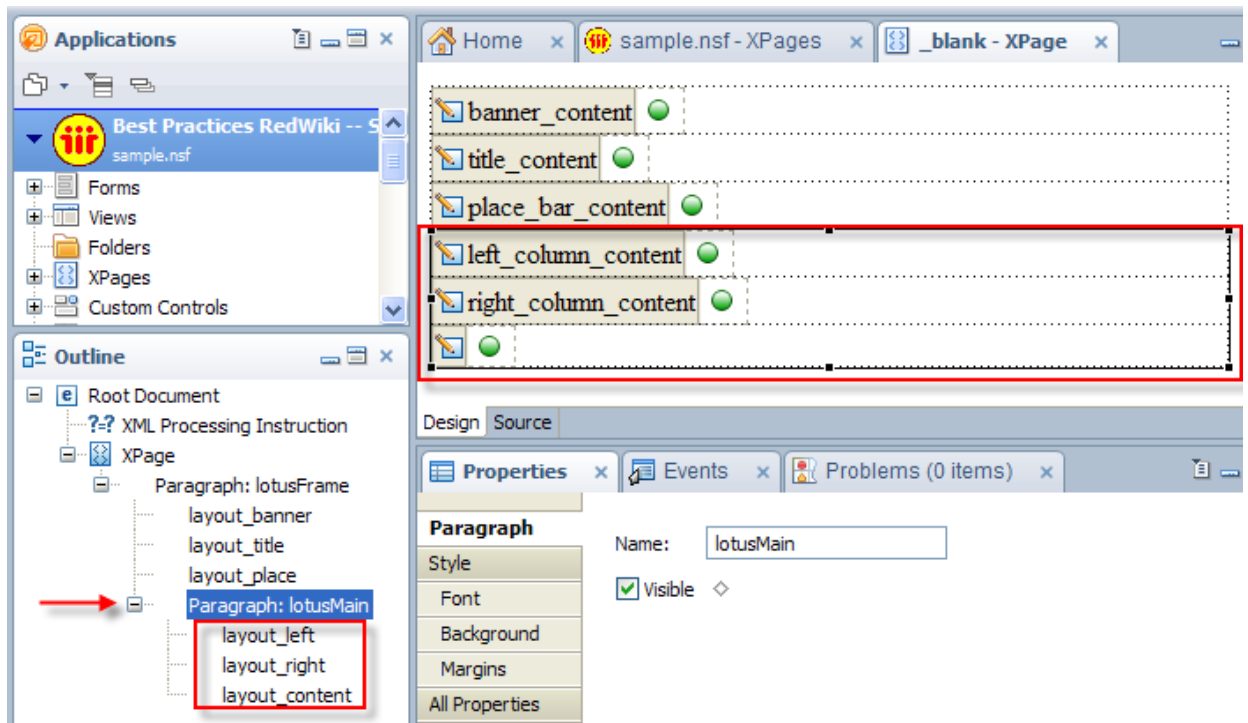


- Switch back to the Source tab and enter the following code above </xp:div>. This one creates the <DIV> element for the main content area – which is further subdivided into three columns: left, content, right. Please refer to step 4 for further details on One UI sections:

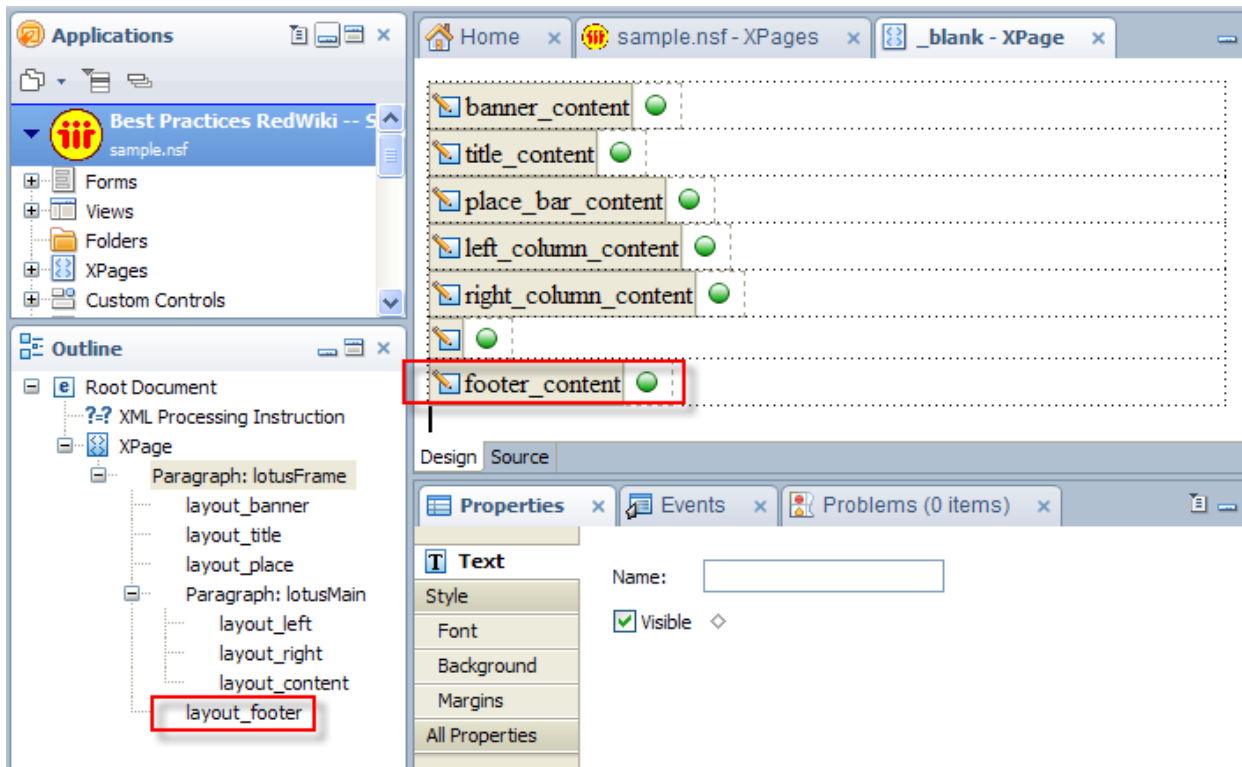
```
<xp:div id="lotusMain" styleClass="lotusMain">
</xp:div>
```



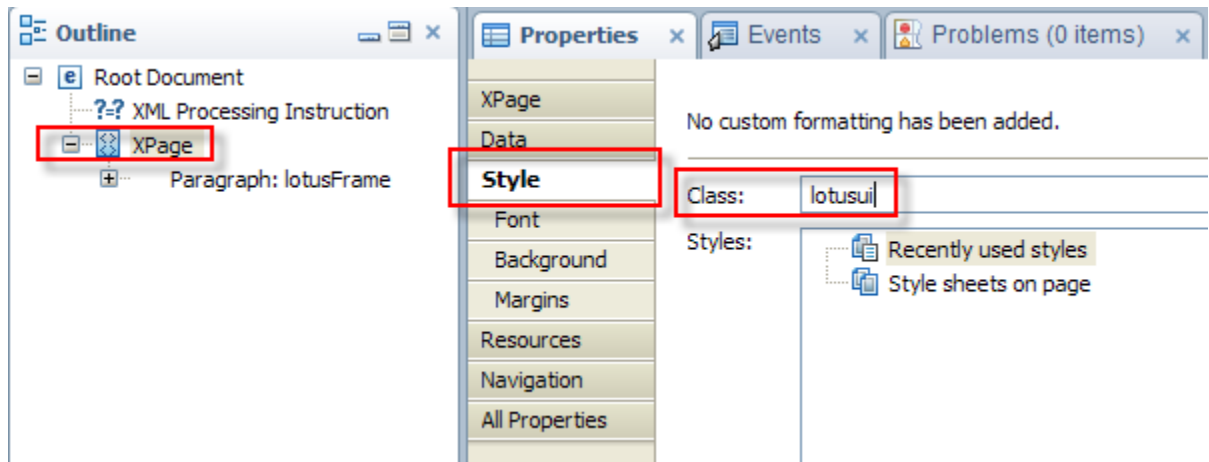
- Switch back to Design tab. Drag and drop custom controls within the "lotusMain" <DIV> element in the following order: **layout_left**, **layout_right**, **layout_content**.



6. Drag and drop a “lotus_footer” custom control below the “lotusMain” <DIV> element.



7. Select the XPage form the Outline palette, and enter “lotusui” as its CSS style class.



8. Click on “Source” tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core"
xmlns:xc="http://www.ibm.com/xsp/custom"
styleClass="lotusui">

    <xp:div id="lotusFrame" styleClass="lotusFrame">
        <xc:layout_banner></xc:layout_banner>
        <xc:layout_title></xc:layout_title>
        <xc:layout_place></xc:layout_place>

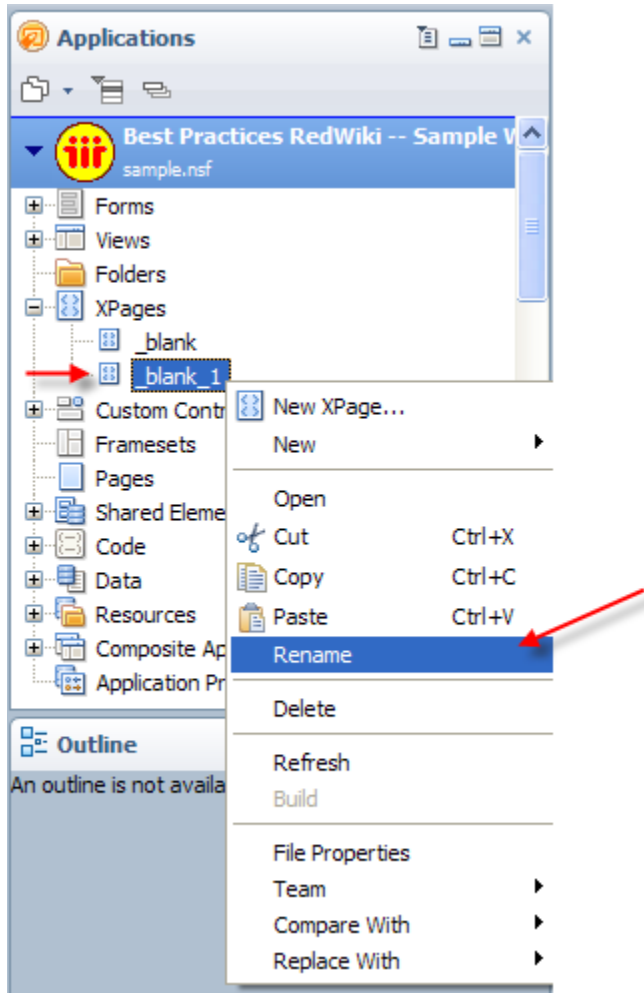
        <xp:div id="lotusMain" styleClass="lotusMain">
            <xc:layout_left></xc:layout_left>
            <xc:layout_right></xc:layout_right>
            <xc:layout_content></xc:layout_content>
        </xp:div>

        <xc:layout_footer></xc:layout_footer>

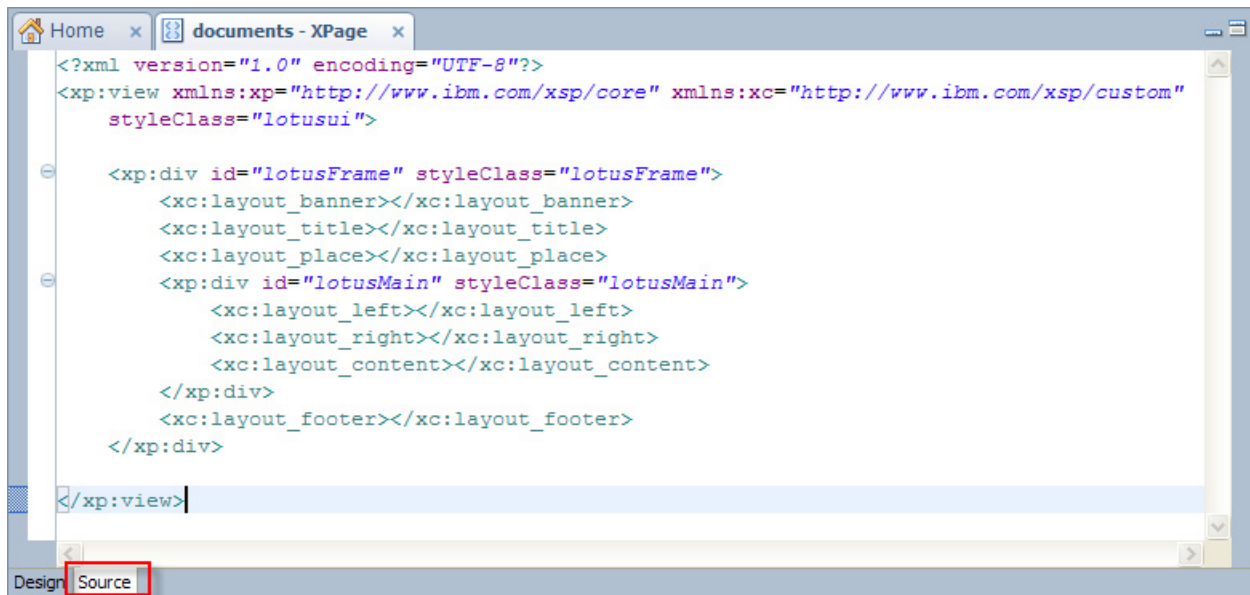
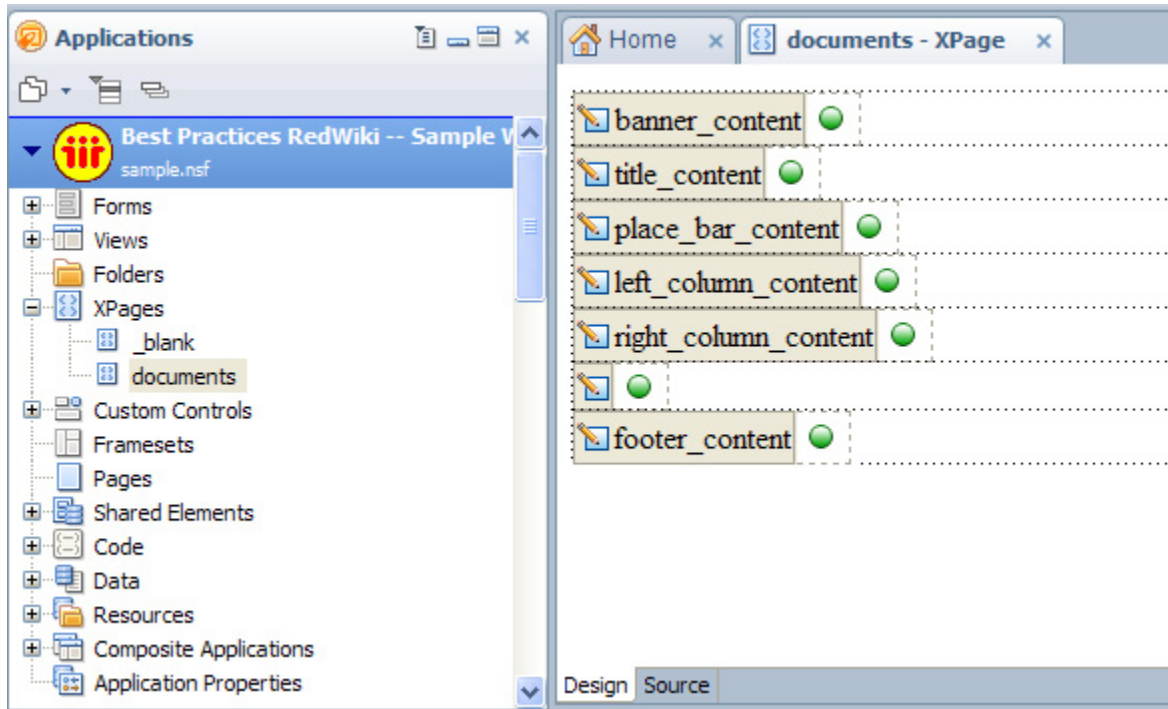
    </xp:div>
</xp:view>
```

Step 9.3: Creating an XPage for documents section

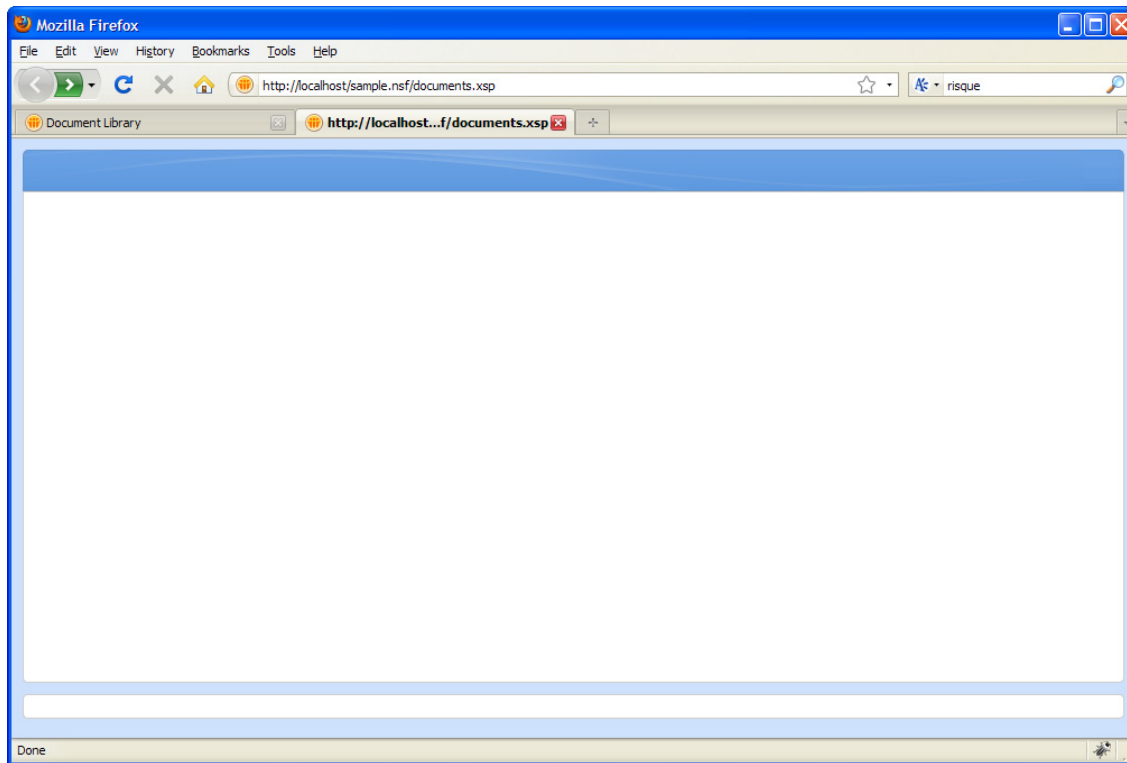
1. The purpose of “**_blank**” XPage was to act as a starting point. We are going to use it in this section to create an XPage for documents section of the sample application. Copy the “**_blank**” XPage and rename the new copy as “**documents**”.



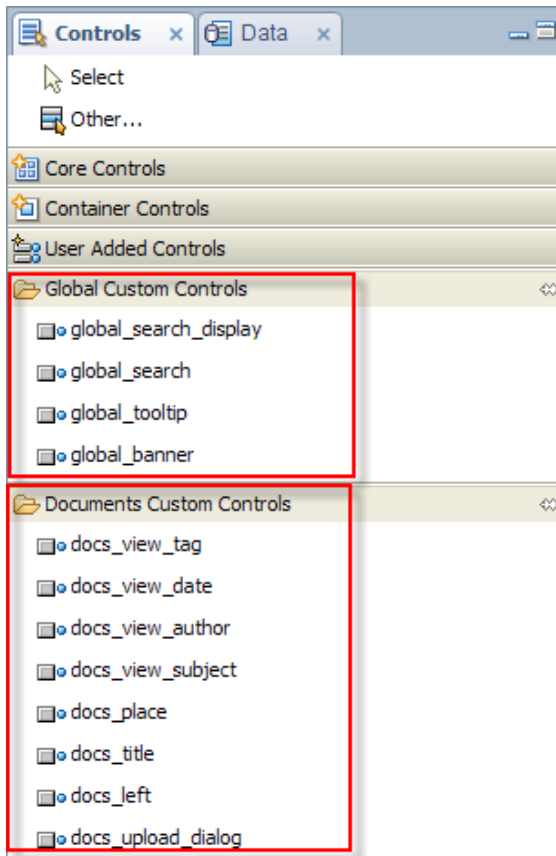
2. Open “documents” XPage and take a look at it in design and source mode. In this XPage, we have placeholders for content. We are going to drop specific content custom controls in these placeholders (layout custom controls).



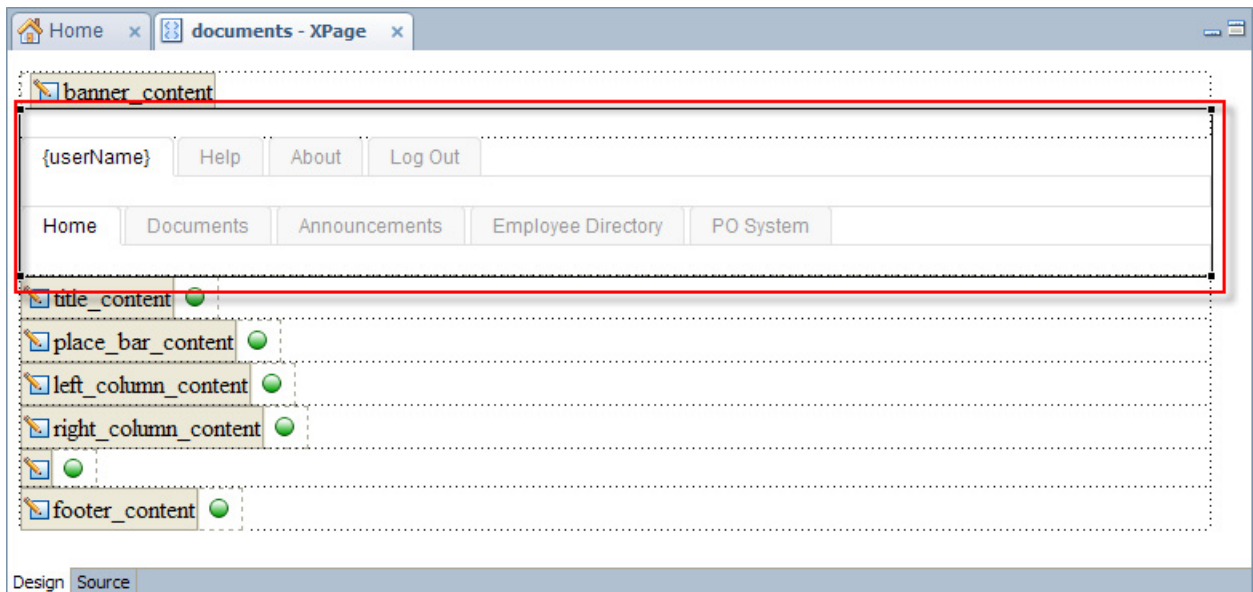
3. Preview the XPage in a web browser and you will see a blank web page with some basic layout.

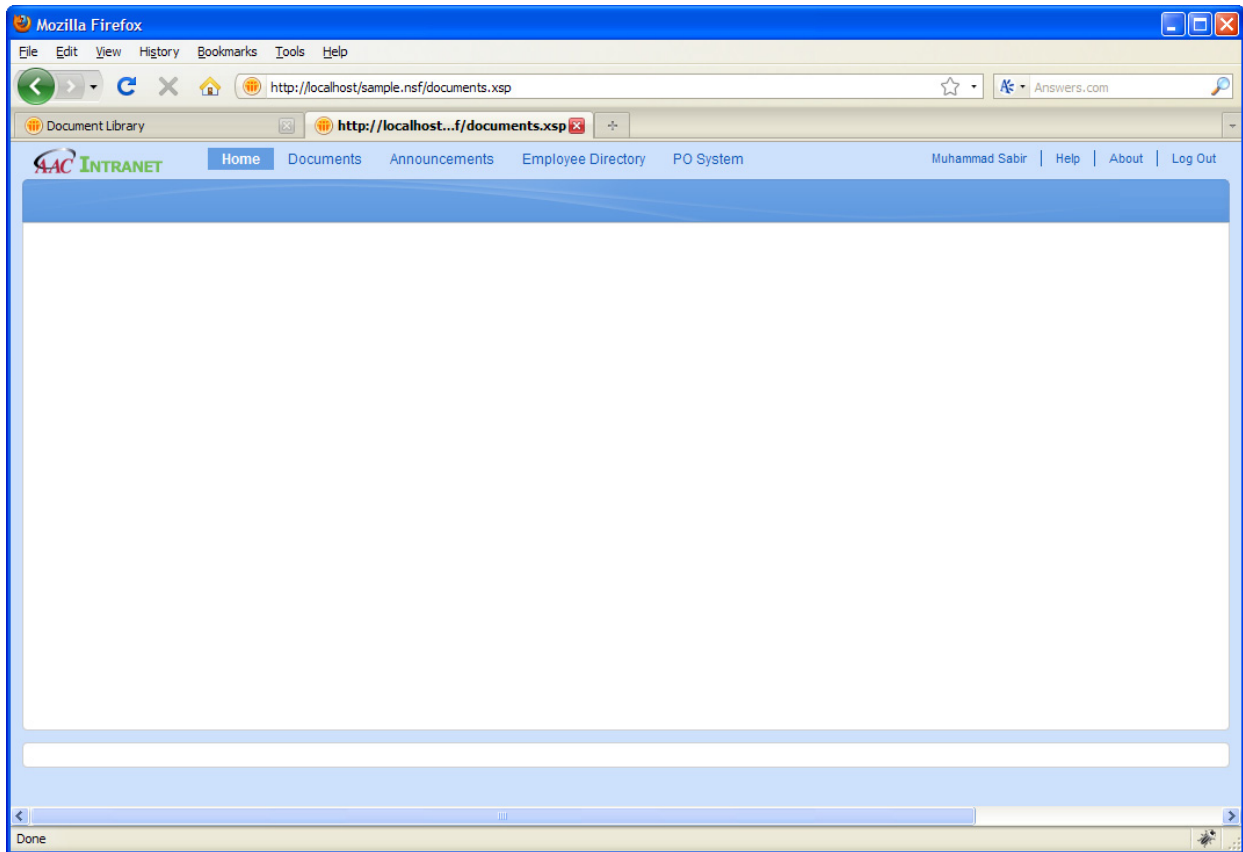


4. We will be using the custom controls we developed earlier and add them to this XPage. Take a look at **“Global Custom Controls”** and **“Documents Custom Controls”** categories in the controls palette.

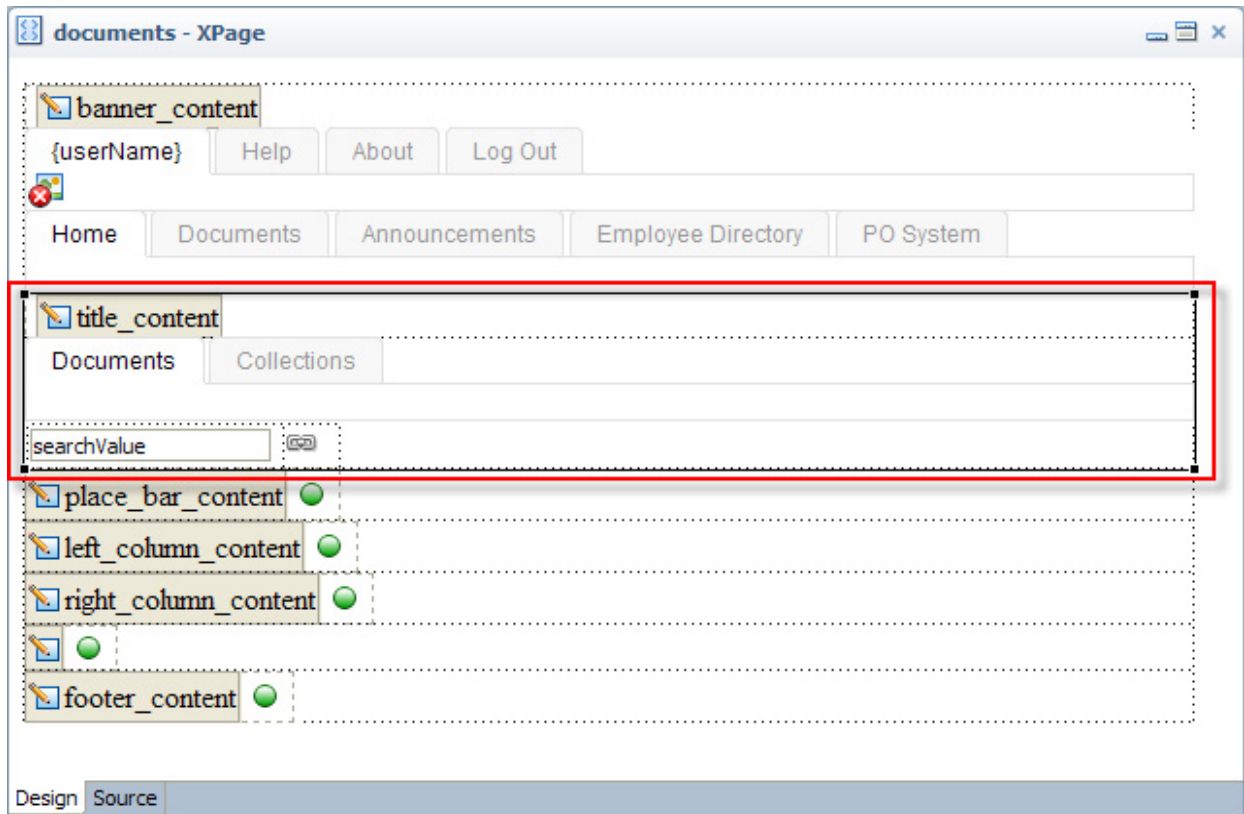


5. From **"Global Custom Controls"**, drag **"global_banner"** control and drop it over **"banner_content"** layout area (indicated by green circle). Preview the XPage in web browser and you should see the banner appearing on the top as shown in the screenshot.

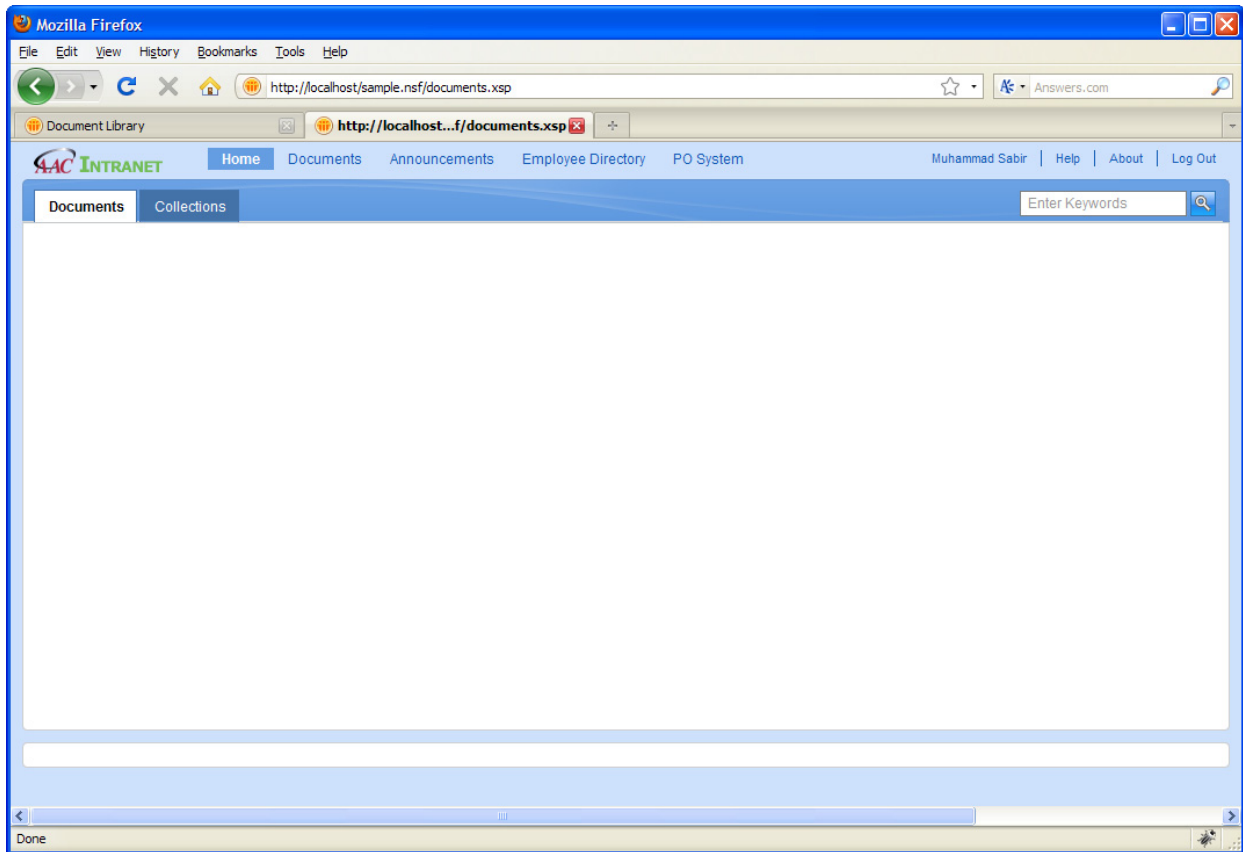




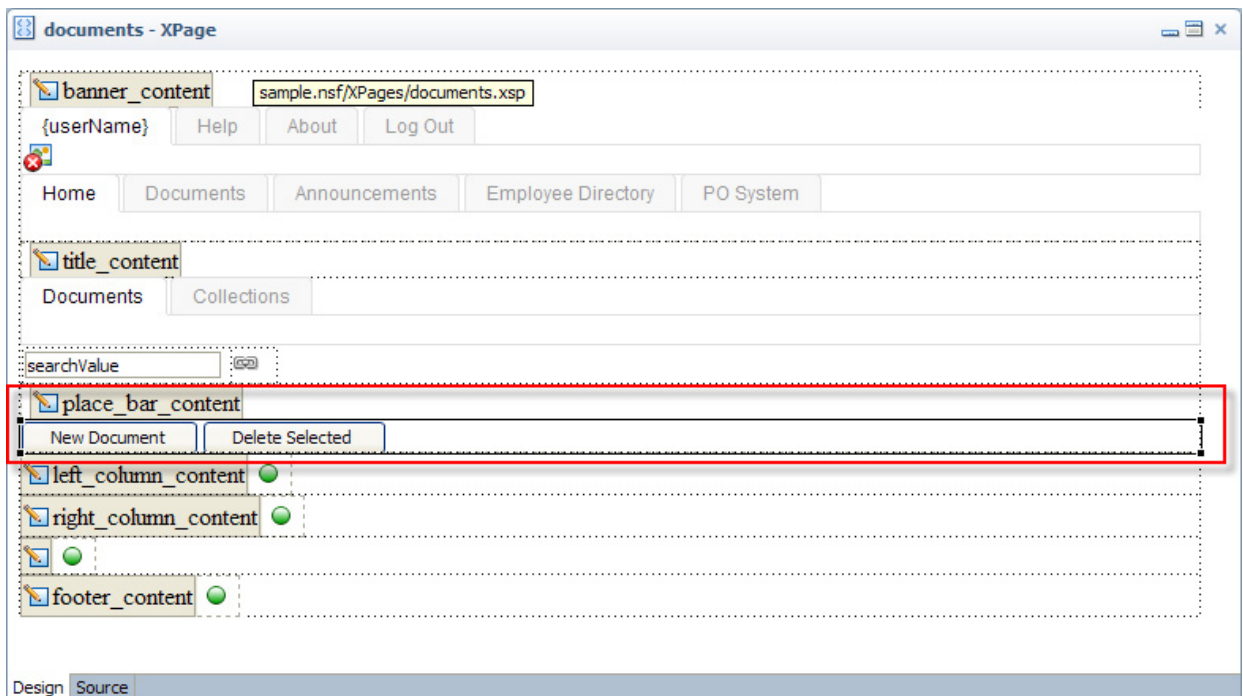
6. From “**Documents Custom Controls**”, drag “**docs_title**” control and drop it over “**title_content**” layout area (indicated by green circle).

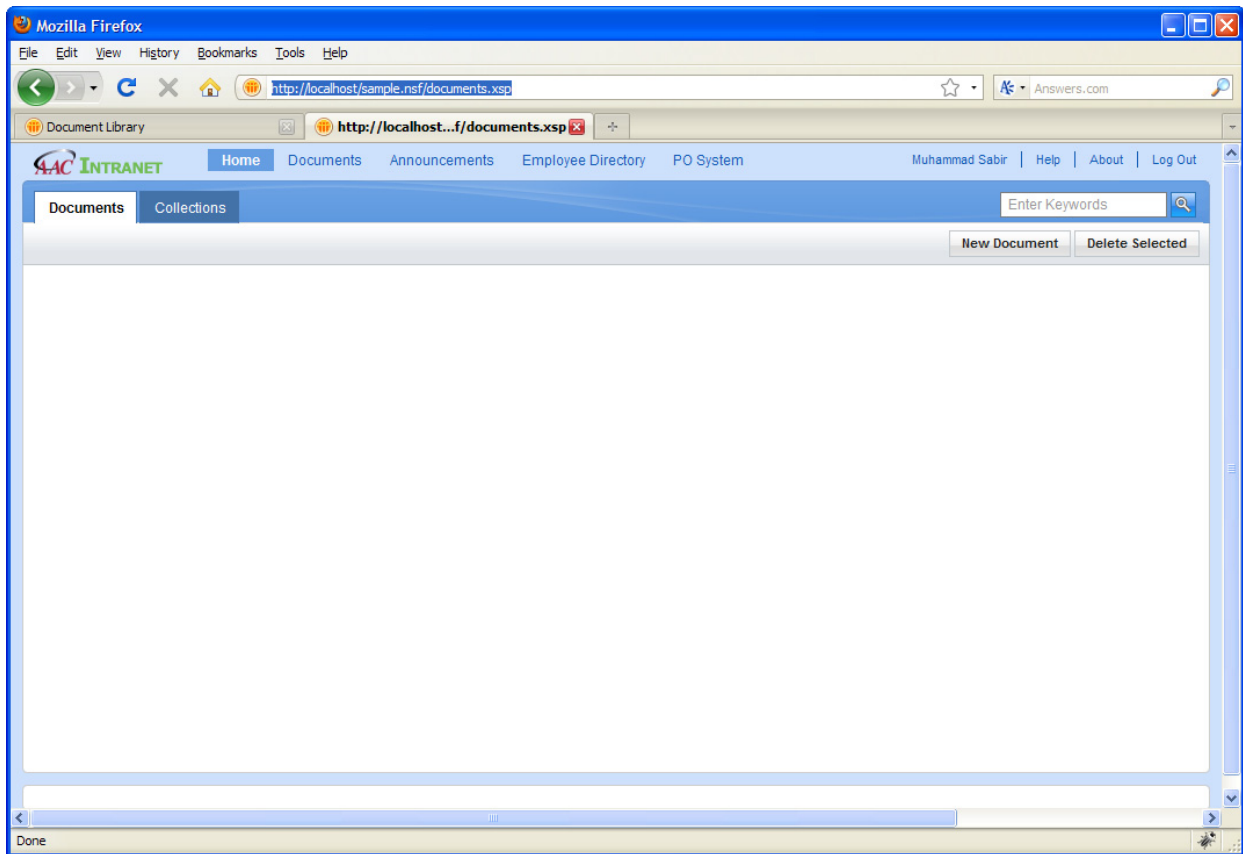


7. Preview the XPage in web browser. You should see a title bar below the banner area, with tabs on the left and search control on the right.



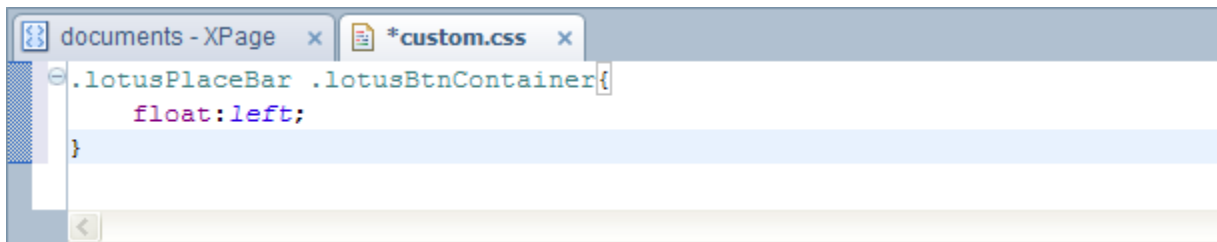
8. From “Documents Custom Controls”, drag “docs_place” control and drop it over “place_bar_content” layout area (indicated by green circle) and preview it in web browser.

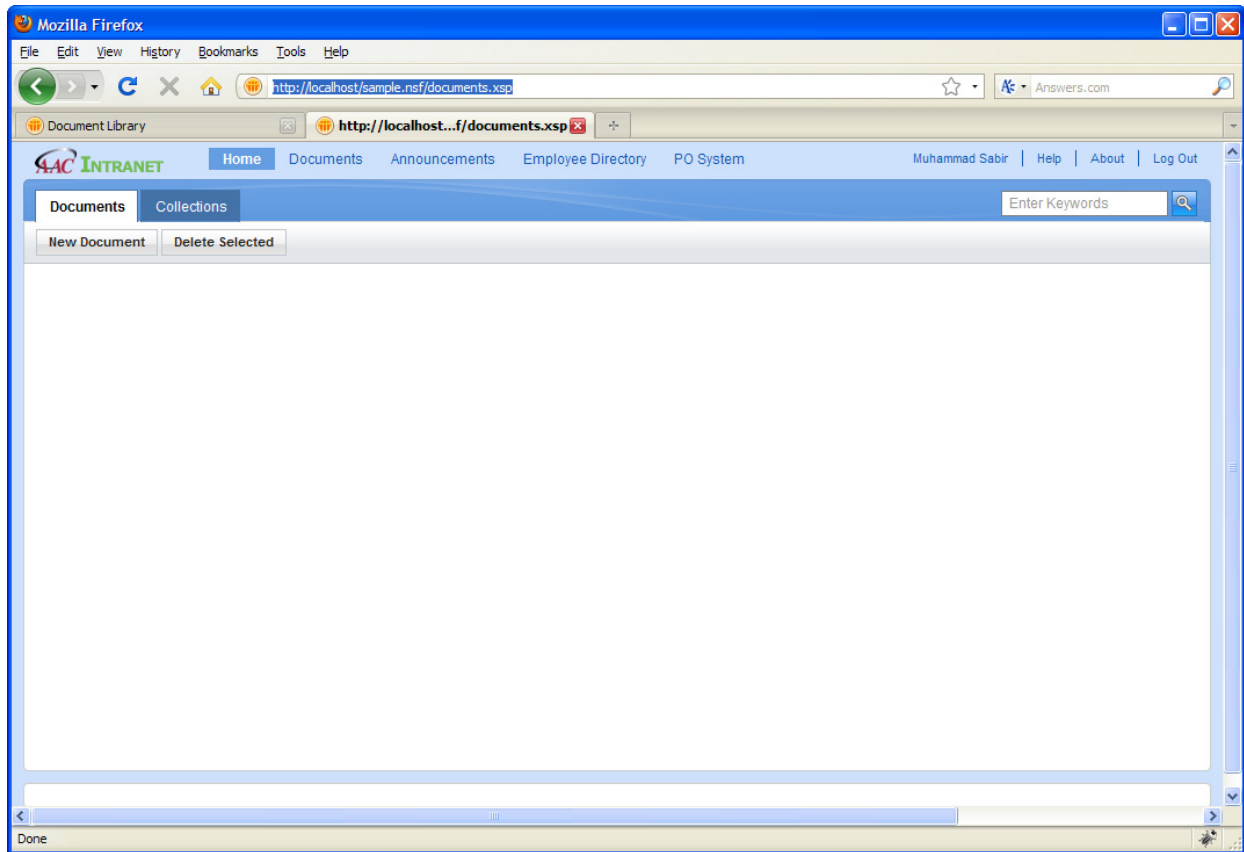




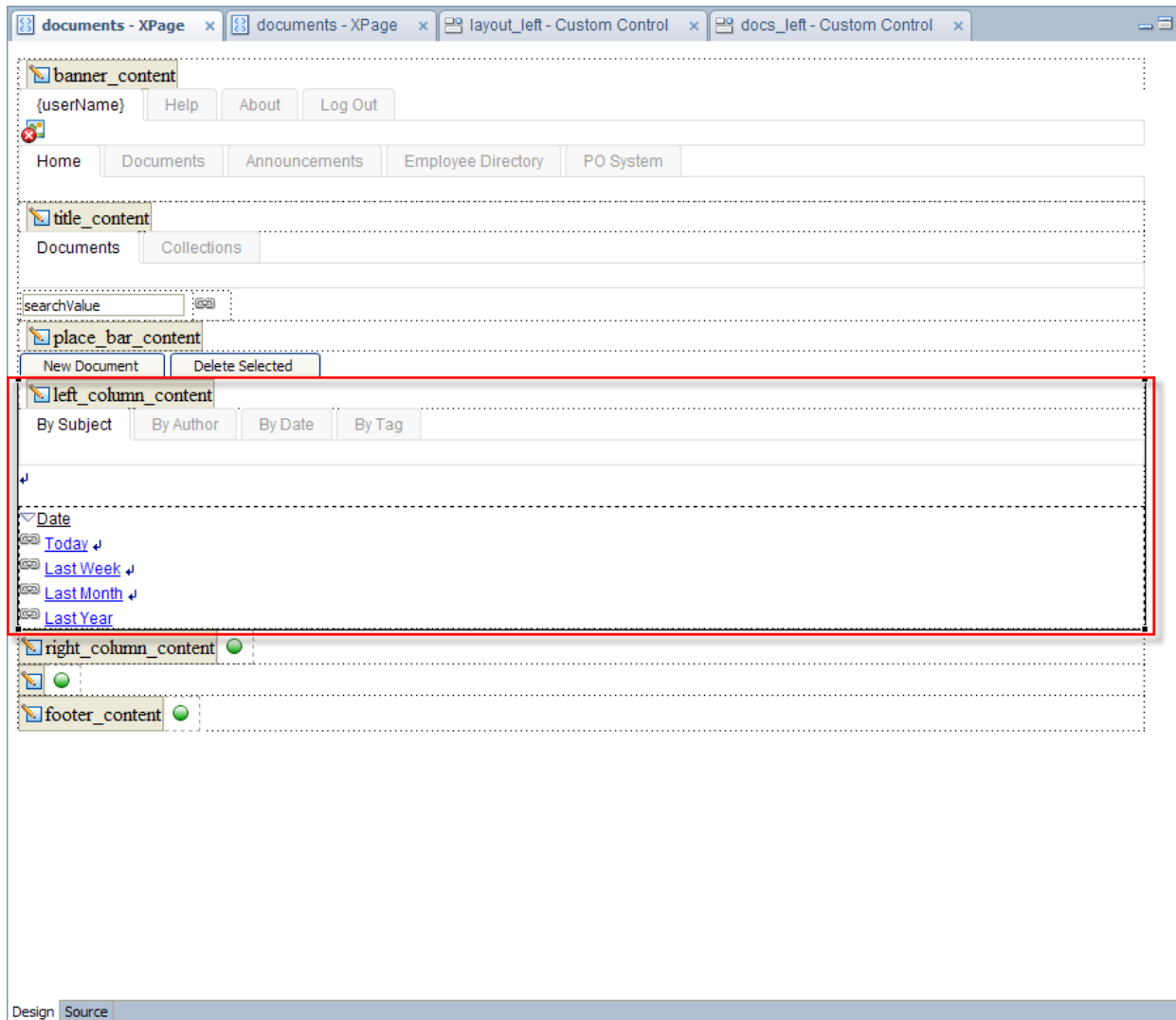
9. Notice that buttons on the place bar appear on the right. To move them on left, enter the following code in custom.css and save the changes. Refresh browser screen and the buttons should not be left aligned as expected.

```
.lotusPlaceBar .lotusBtnContainer{  
    float:left;  
}
```

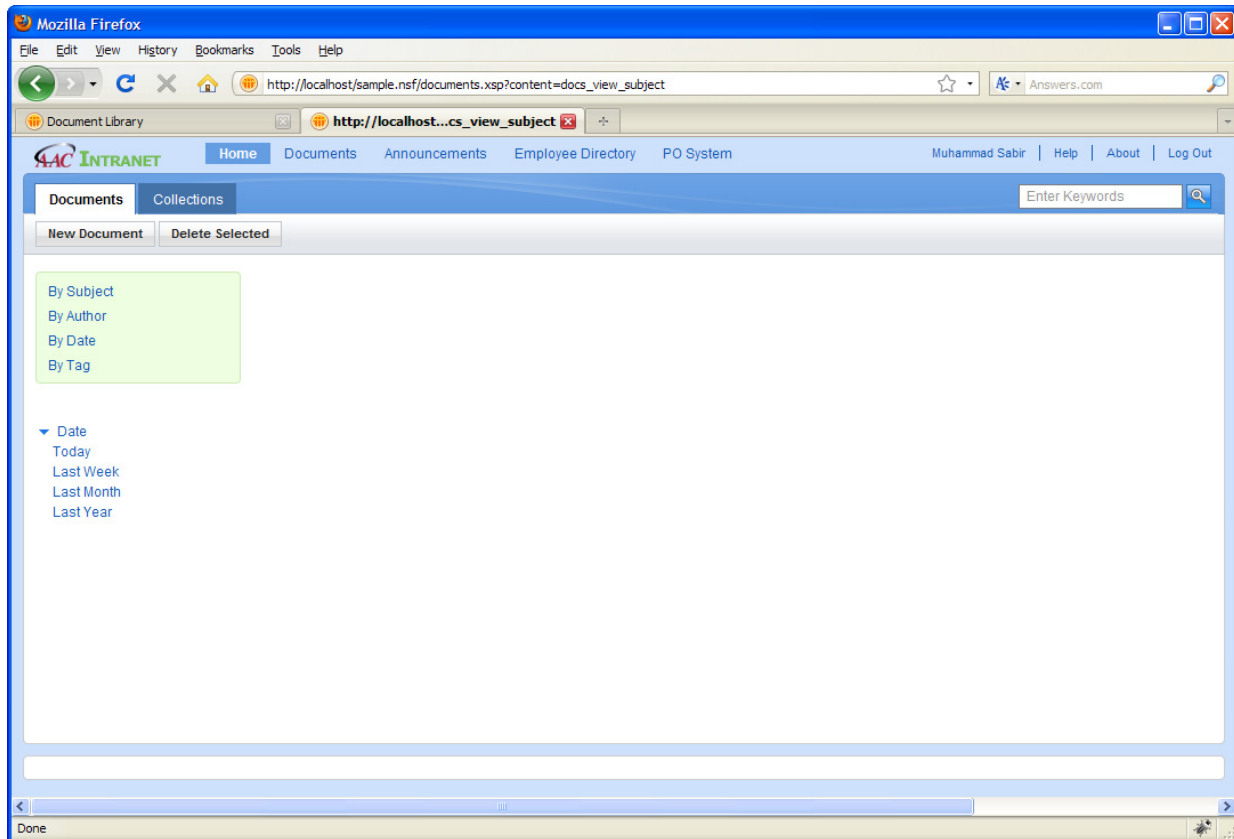




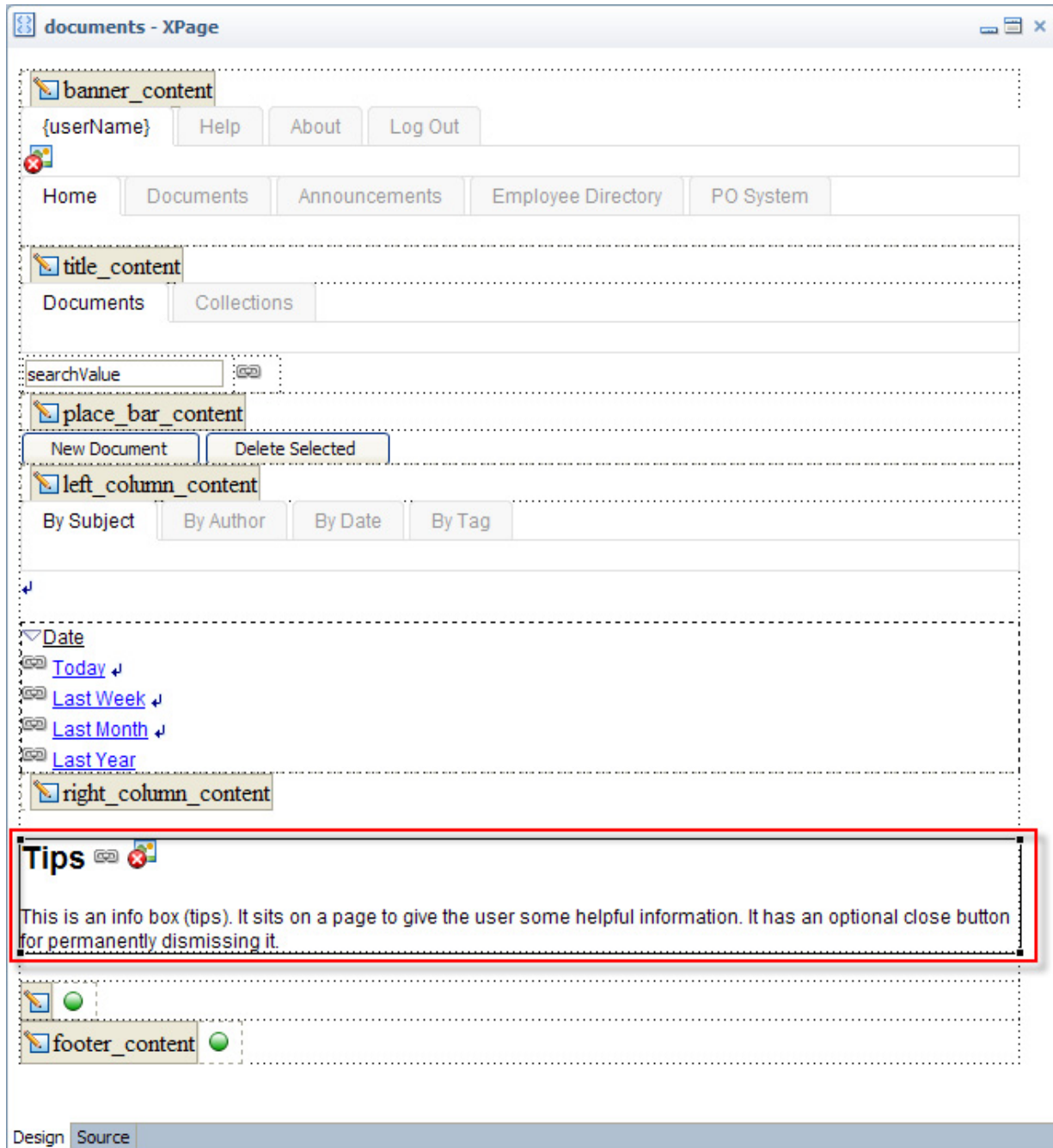
10. From “**Documents Custom Controls**”, drag “**docs_left**” control and drop it over “**left_column_content**” layout area.



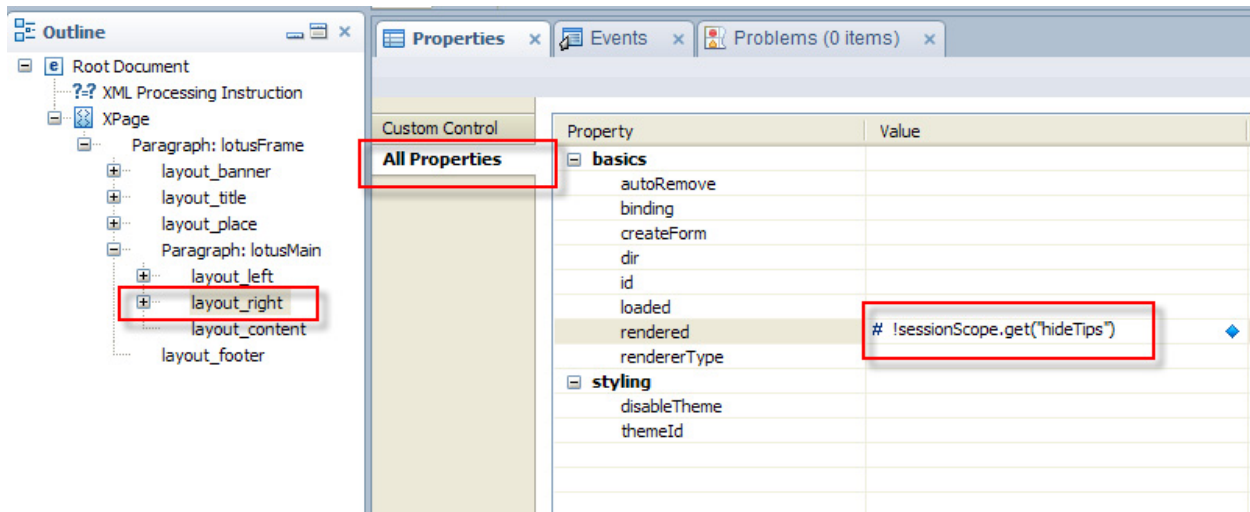
11. Preview the XPage in web browser. You should see a left navigation menu added to the page.



12. From **“Documents Custom Controls”**, drag **“docs_right”** control and drop it over **“right_column_content”** layout area.

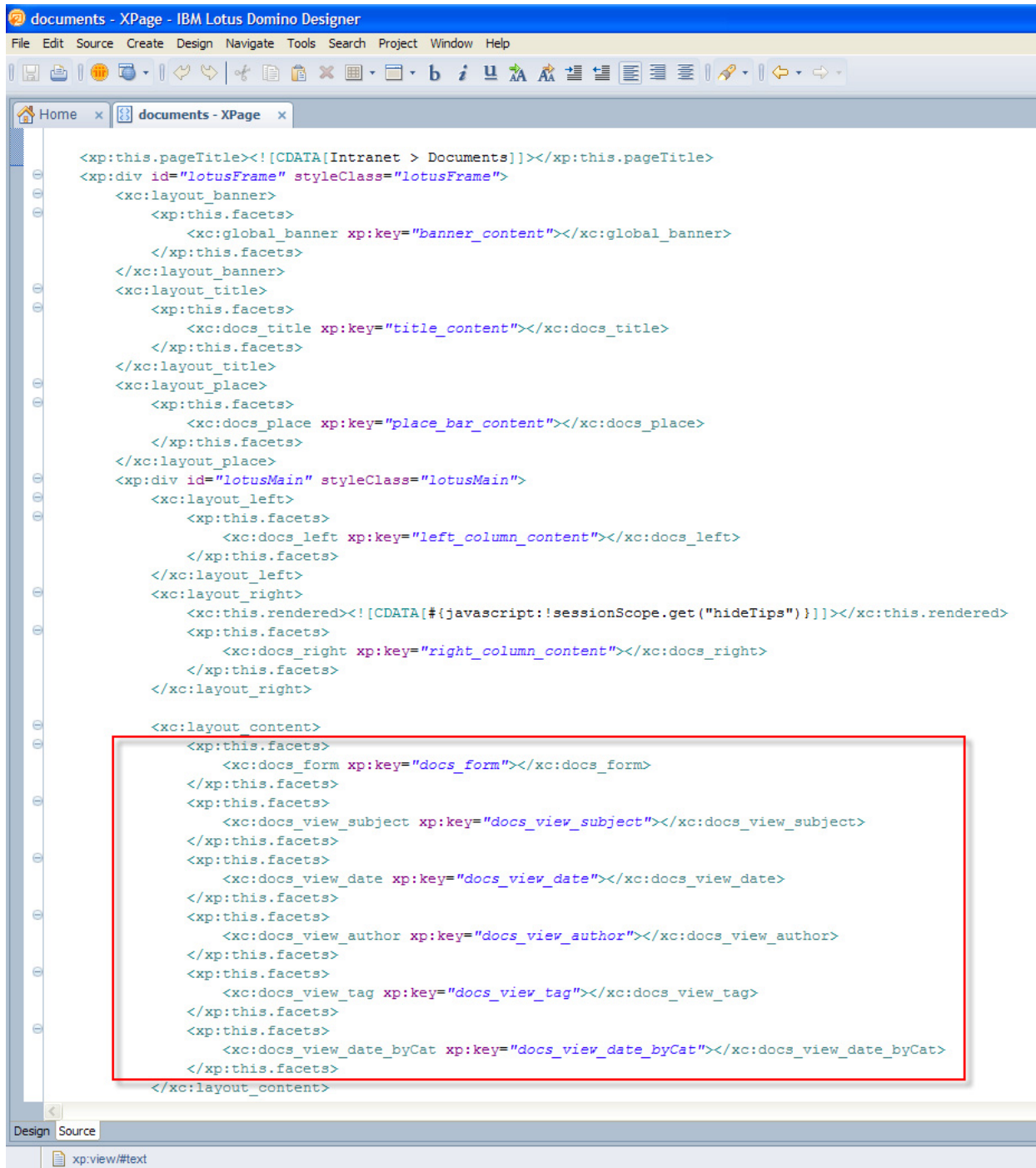


13. Select "**layout_right**" control in the outline palette. Select "**All Properties**" tab and enter the following code for "**rendered**" property: `sessionScope.get("hideTips")`. This will display or hide the control based on "**hideTips**" session scope variable.



14. Now we have built the layout and content, except for the main column in the center – which is actually implemented via “**layout_content**” custom control. Rest of the page stays static but the content of this section change based on the link user has clicked within the application. The link actually sets up the “**content**” query string parameter and based on the value of this parameter a control is dynamically included in the XPage. In order to make it work, we need to include all possible custom controls that can potentially be included in this section. Switch to the source mode and enter the following code within **<xc:layout_content>** tag:

```
<xp:this.facets>
  <xc:docs_form xp:key="docs_form"></xc:docs_form>
</xp:this.facets>
<xp:this.facets>
  <xc:docs_view_subject
xp:key="docs_view_subject"></xc:docs_view_subject>
</xp:this.facets>
<xp:this.facets>
  <xc:docs_view_date xp:key="docs_view_date"></xc:docs_view_date>
</xp:this.facets>
<xp:this.facets>
  <xc:docs_view_author
xp:key="docs_view_author"></xc:docs_view_author>
</xp:this.facets>
<xp:this.facets>
  <xc:docs_view_tag xp:key="docs_view_tag"></xc:docs_view_tag>
</xp:this.facets>
<xp:this.facets>
  <xc:docs_view_date_byCat
xp:key="docs_view_date_byCat"></xc:docs_view_date_byCat>
</xp:this.facets>
```



- Click on "Source" tab in **XPages** editor and press **Ctrl-Shift-F** to format the source code and save the changes. You should see the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core"
  xmlns:xc="http://www.ibm.com/xsp/custom" styleClass="lotusui">

```



```

<xp:this.pageTitle><![CDATA[Intranet > Documents]]></xp:this.pageTitle>
<xp:div id="lotusFrame" styleClass="lotusFrame">
  <xc:layout_banner>
    <xp:this.facets>
      <xc:global_banner
xp:key="banner_content"></xc:global_banner>
    </xp:this.facets>
  </xc:layout_banner>
  <xc:layout_title>
    <xp:this.facets>
      <xc:docs_title
xp:key="title_content"></xc:docs_title>
    </xp:this.facets>
  </xc:layout_title>
  <xc:layout_place>
    <xp:this.facets>
      <xc:docs_place
xp:key="place_bar_content"></xc:docs_place>
    </xp:this.facets>
  </xc:layout_place>
  <xp:div id="lotusMain" styleClass="lotusMain">
    <xc:layout_left>
      <xp:this.facets>
        <xc:docs_left
xp:key="left_column_content"></xc:docs_left>
      </xp:this.facets>
    </xc:layout_left>
    <xc:layout_right>

    <xc:this.rendered><![CDATA[#{javascript:!sessionScope.get("hideTips")}]]
]></xc:this.rendered>
      <xp:this.facets>
        <xc:docs_right
xp:key="right_column_content"></xc:docs_right>
      </xp:this.facets>
    </xc:layout_right>

    <xc:layout_content>
      <xp:this.facets>
        <xc:docs_form
xp:key="docs_form"></xc:docs_form>
      </xp:this.facets>
      <xp:this.facets>
        <xc:docs_view_subject
xp:key="docs_view_subject"></xc:docs_view_subject>
      </xp:this.facets>
      <xp:this.facets>
        <xc:docs_view_date
xp:key="docs_view_date"></xc:docs_view_date>
      </xp:this.facets>
      <xp:this.facets>
        <xc:docs_view_author
xp:key="docs_view_author"></xc:docs_view_author>
      </xp:this.facets>
      <xp:this.facets>
        <xc:docs_view_tag
xp:key="docs_view_tag"></xc:docs_view_tag>

```

```

        </xp:this.facets>
        <xp:this.facets>
            <xc:docs_view_date_byCat
xp:key="docs_view_date_byCat"></xc:docs_view_date_byCat>
        </xp:this.facets>
    </xc:layout_content>

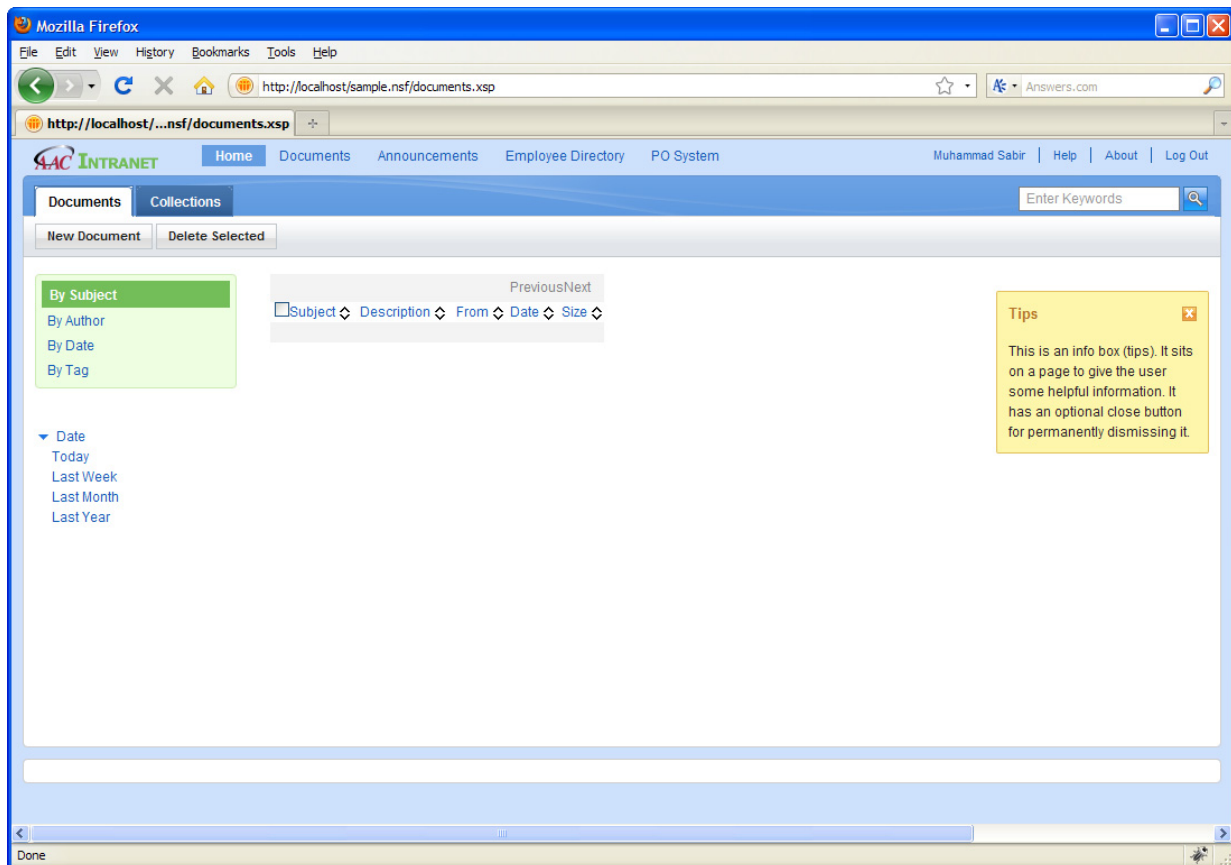
    <xc:docs_upload_dialog></xc:docs_upload_dialog>
</xp:div>
<xc:layout_footer></xc:layout_footer>
</xp:div>

</xp:view>

```

15. Save changes and preview the page in web browser. If there is no content query sting, it will load **"docs_view_subject"** custom control. This was explained earlier in the section where we developed the **"layout_content"** custom control. The **"Facet Name"** property for that control was set to be computed with the following formula: (param.content || "docs_view_subject").

Note you can hide the tips box on the right by clicking the close button on the right of the box – marked with x sign.

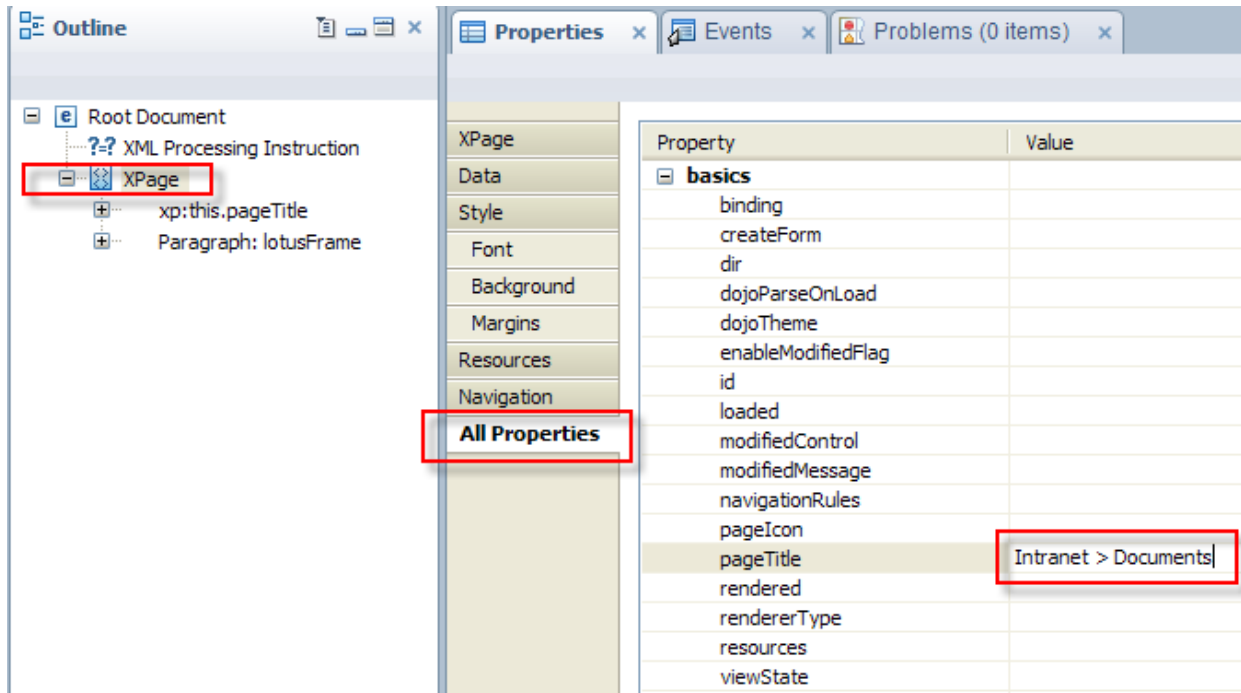


16. Note that the table does not expand fully to the content area (at least in firefox browser). Also the spacing between column headers is a bit small. Enter the following CSS in custom.css:

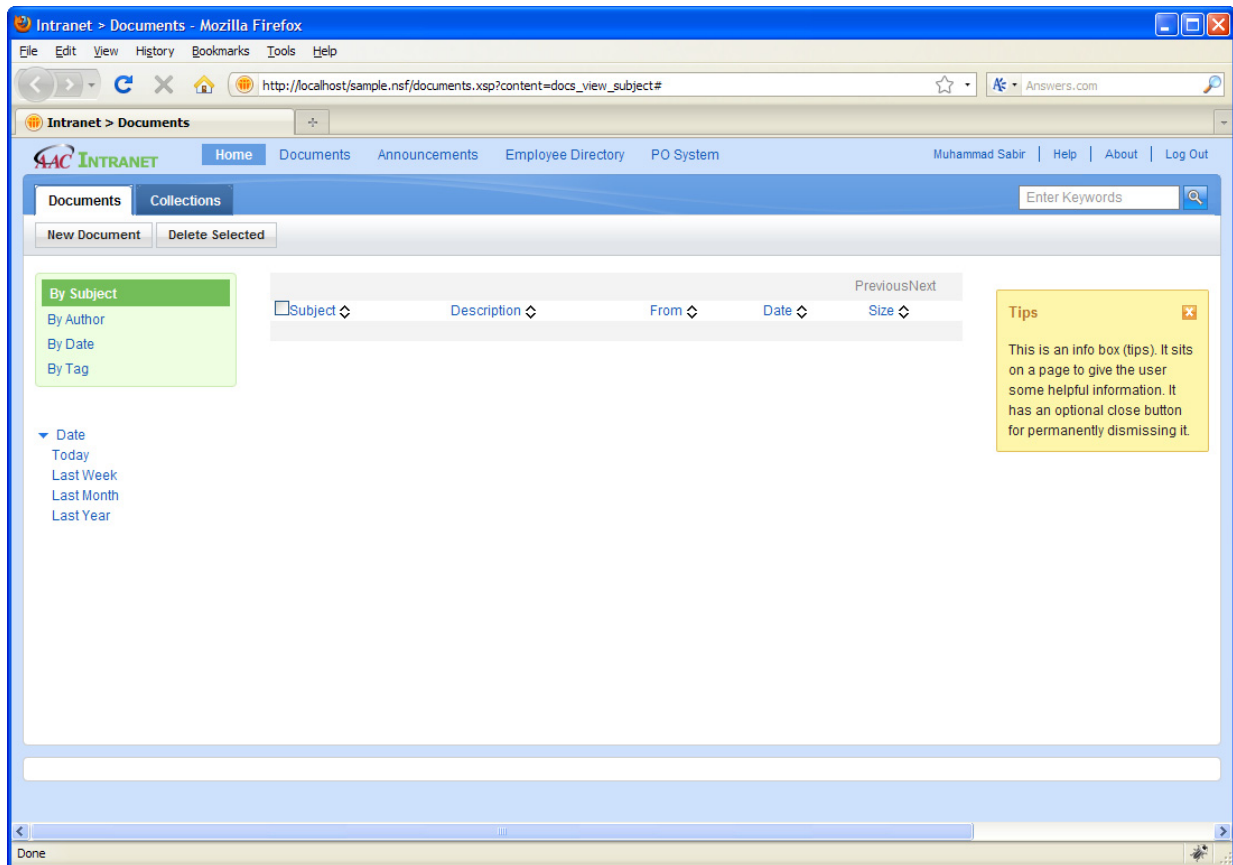
```
.xspDataTableViewPanel {  
    width:100%;  
}  
  
.xspColumnViewStart{  
    padding-right:10px;  
}  
  
.xspColumnViewMiddle{  
    padding-right:10px;  
}
```



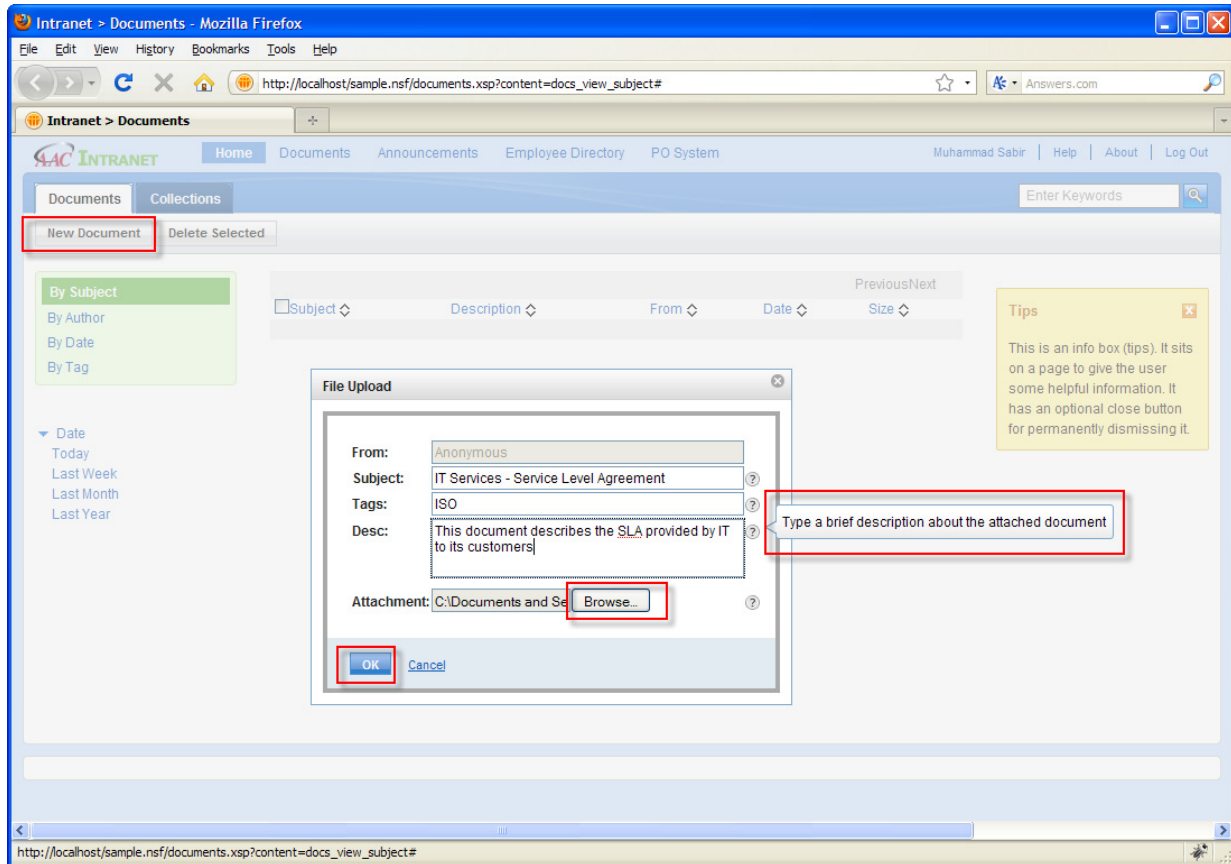
17. Set "pageTitle" property of XPage as "Intranet > Documents".



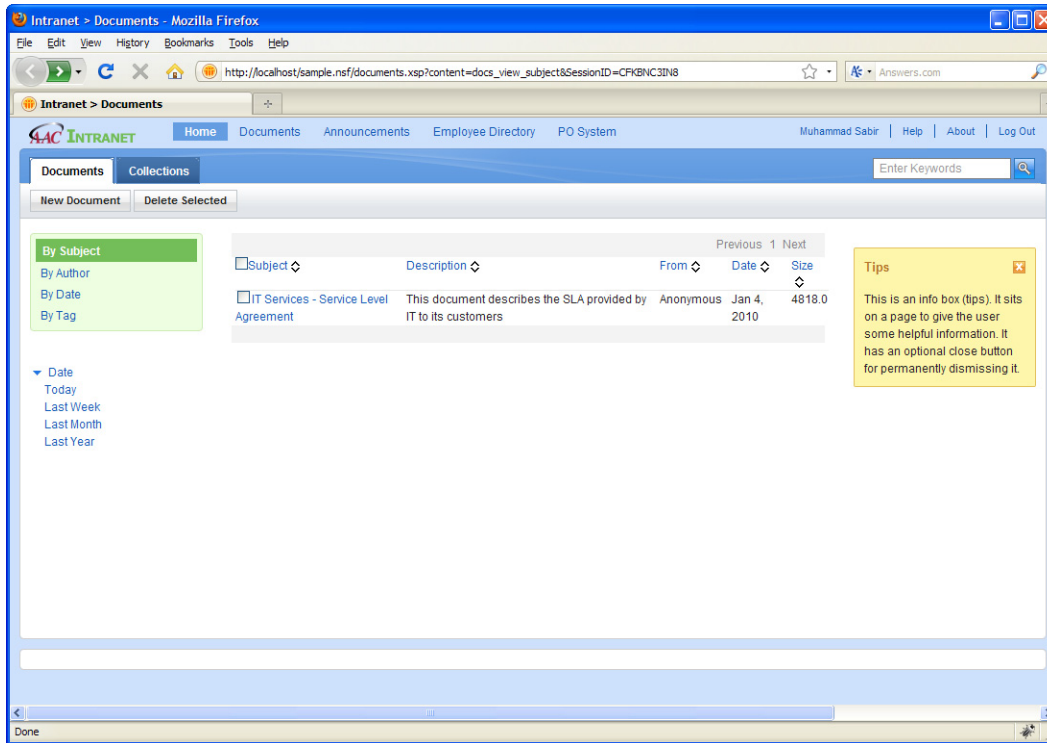
18. Preview XPage in the browser. Note the improved formatting for table and its columns.



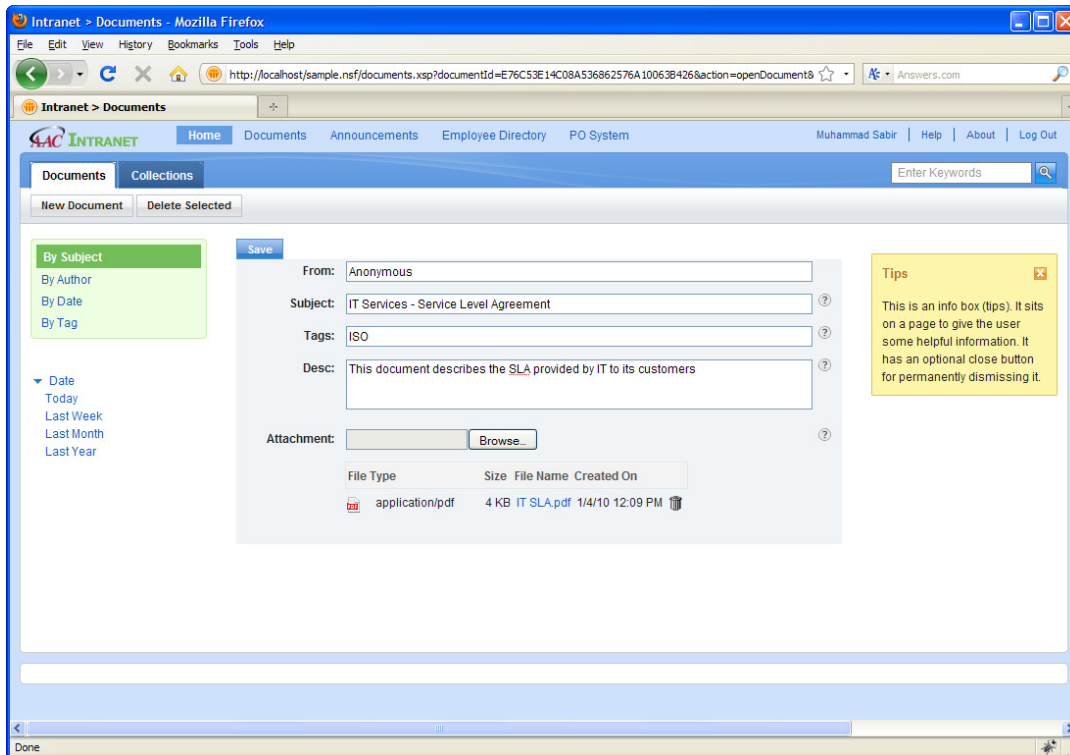
16. Click on “New Document” and file upload dialog will appear. Fill up the form and click OK to save changed. Note that you can click on ? icon to view the tooltip related to each field.



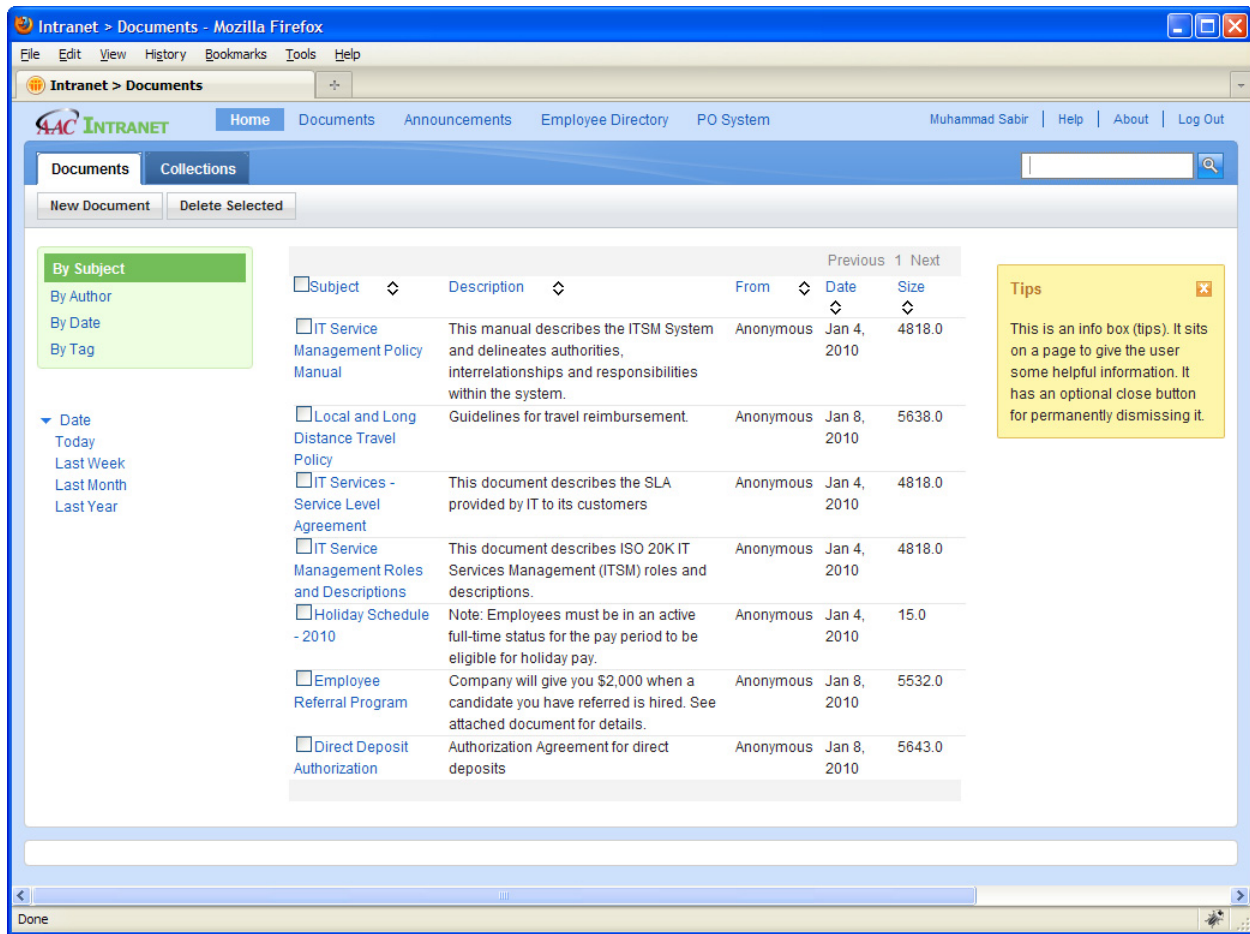
17. Once a document is uploaded, it gets listed.



15. Click on any document link to view/edit update the document.



16. Click on the links on the left menu to change the view. Note the “**content**” query string variable changes as you click on any link.



17. To search for a document, enter the search criteria in search control and click on submit button to the right of the search input. You should see the matching documents. Click on "x" button to clear the search filter.

