XML Data-Binding in Backbase 3.1

Intended Audience

This tutorial is intended for developers who want to learn how to best use the data-binding features of Backbase 3.1, and is specifically aimed at people doing data-binding for the first time. Basic knowledge of XML and BXML is assumed.

Introduction

Data-binding is a technique frequently used in many different programming environments. Because of this, the term is overloaded and means different things to different developers. We will therefore start this tutorial by giving a short definition of data-binding.

On an abstract level, data-binding means connecting data from a server-side data source to a client-side UI control. A good data-binding mechanism provides for an easy way to achieve this connection and will free the developer from writing plumbing code.

Concretely, a data-binding involves executing two tasks:

1. Loading and storing the data: The data is physically located on the server and needs to be transferred between the client and the server.

2. Transformation to required format: The data is stored in some format in the data source. Displaying data in the UI control requires the data to be compatible with the client-side presentation format.

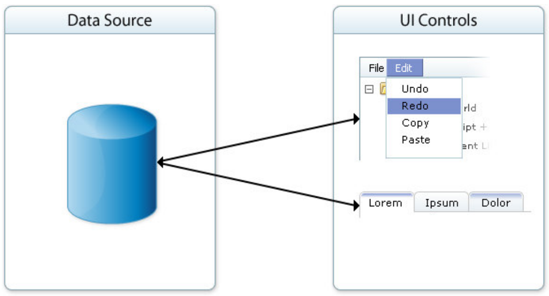


Figure 1 "Data-binding Tasks"

An essential question when looking at different data-binding approaches is where the two tasks (1) loading / storing and (2) transformation are executed, on the client or on the server.

Backbase is an extremely flexible Ajax engine that lets developers use different approaches. As shown in the figure below, there are two basic strategies:

1. Executing both loading / storing and transformation on the server,

2. Executing loading / storing on the server but transformation on the client.

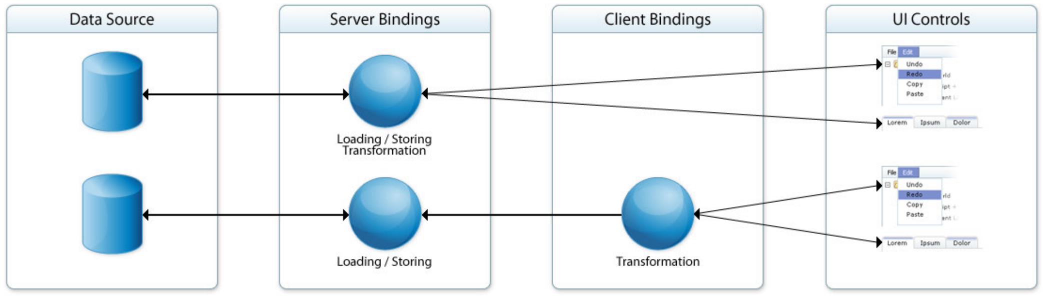


Figure 2 "Data Binding Approaches"

The article will describe three strategies for data-binding by explaining the logic for loading and storing and by showing how to transform data into BXML. Two of the data-binding strategies involve doing the transformation client-side. The third one is completely server-side.

1. XML Data-binding with the Browser XSLT Engine

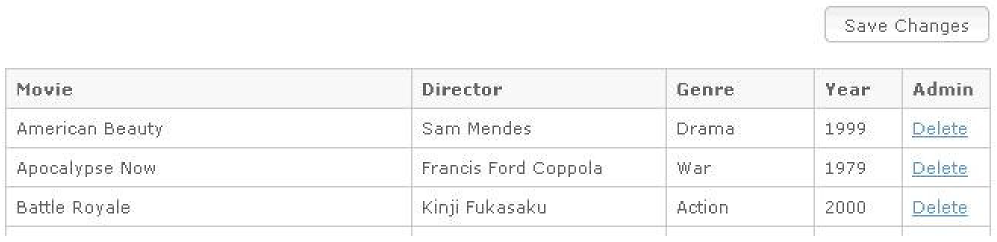
2. XML Data-binding with the BPC XSLT Engine

3. XML Data-binding with the PHP XSLT Engine

The article will not only describe how to use the above three strategies but also highlight the advantages of each method. As of version 3.1.1 you can find complete code templates for each approach in the data-binding starterkit.

The Sample Application

The sample application is a table with movie data that end users can edit in the browser. The screenshot below shows the application and the data-binding from the end user's perspective. Some data is displayed, it can be edited, and the changes can be saved.



Save Changes

Movie	Director	Genre	Year	Admin
American Beauty	Sam Mendes	Drama	1999	Delete
Apocalypse Now	Francis Ford Coppola	War	1979	Delete
Battle Royale	Kinji Fukasaku	Action	2000	Delete

Figure 3 "Example of data display and editing"

The Data-binding Loop

When data is stored on the server, displayed and edited in the client, and then saved back on the server, it will be ready to go through this process again and again. This is the data-binding loop. The picture below illustrates this loop in a very general way.

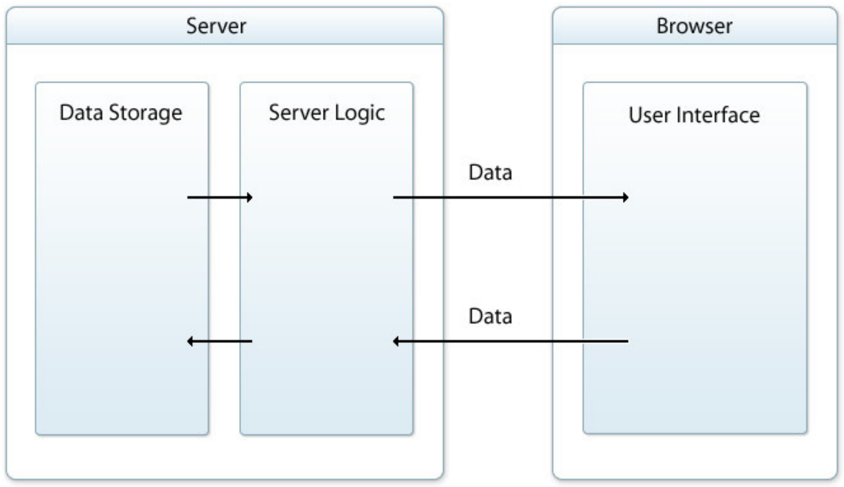


Figure 4 "Data-binding Overview"

Because the data is usually stored on the server in a different format from how it is managed and displayed in the user interface, transformations between storage and display formats are necessary.

In this article it is assumed that the server logic code already has the data available as raw XML. In many environments a conversion from a SQL managed database storage to XML format will be needed to get to this point. However, this step depends heavily on the particular server environment in use, and handling this lies not within the scope of this article. In our examples, we have worked with data that is stored as a simple XML file on the server, so no additional conversions are needed.

XML Data-binding with the Browser XSLT Engine

Overview

In our first case we will have a look at using the XSLT transformation engine available in all browsers currently supported by Backbase. When using this approach, it means the data is transferred from the server to the browser as raw XML, as illustrated in the picture below.

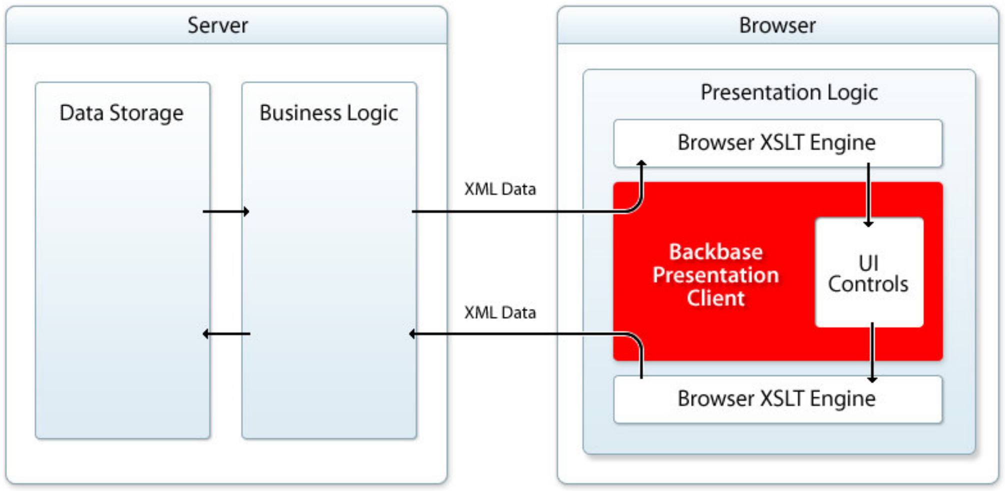


Figure 5 "XML data-binding with the browser XSLT engine"

When the XML data arrives in the browser, the Backbase Presentation Client (BPC) passes it through the browser XSLT engine to transform it to BXML. The data can now be inserted into the BXML-Tree, and the BPC will automatically take care of updating the interface with this new information.

Fast & loosely coupled

The biggest advantage of using the browser transformation engine is performance; it is very fast because it uses the browser's native XSLT implementation. When transforming large amounts of data, this becomes important. Also, by doing the transformation in the browser, the server can stay agnostic of what the data is used for. This allows for a flexible and loosely coupled services architecture.

Cross-browser compatibility

A drawback of this approach is that the developer needs to explicitly deal with potential browser incompatibilities. For example, the XSLT implementation in Internet Explorer 5.0 on Windows is based on an unfinished version of the W3C XSLT specification. Some essential changes were made to this specification after Internet Explorer 5.0 was released. This means that normal stylesheets are likely to fail in that browser, and you might have to feed it alternative stylesheets tailored to its quirks. This adds complexity to the development and maintenance of your applications.

The cross-browser compatibility issue goes further than that though: if Backbase starts supporting new browsers in the future, you will have to deal with their particular XSLT implementations. These might not be powerful or flexible enough to be used in dynamically updated Rich Internet Applications.

Using the browser XSLT engine: taking XML from the server

When loading an XSL stylesheet and XML data that will be transformed by the browser engine, the code to set up the transformation will look a lot like this:

```
<!-- Load data and store as XML Document in a variable -->
<s:variable b:name="filmdata-source" b:scope="global"/>
<s:task b:action="load"
    b:url="data/filmdata.xml"
    b:destination="$filmdata-source"/>
<s:task b:action="string2xml" b:variable="$filmdata-source"/>

<!-- Load stylesheet and store as XML Document in a variable -->
<s:variable b:name="stylesheet" b:scope="global"/>
<s:task b:action="load"
    b:url="data/xml2bxml_standard.xsl"
    b:destination="$stylesheet"/>
<s:task b:action="string2xml" b:variable="$stylesheet"/>

<!-- Transform data, move output directly to BXML-Tree -->
<s:task b:action="xsl-transform"
    b:stylesheet="$stylesheet"
    b:datasource="$filmdata-source"
    b:destination="id('filmdisplay-container')"/>
```

Note the conversion of the loaded files from a string data type to an XML Document data type. The name of the command that triggers the transformation is xsl-transform and the output goes directly to a location in the BXML-Tree specified with a simple XPath.

Note also the global scoping of all variables in this example. By storing the stylesheet and XML data in these variables, they can be reused at a later point without downloading them again.

Using the browser XSLT engine: putting the changes back in XML format

When preparing the changed data for transformation back to raw XML, the code looks like this:

```
<!-- Take changed data and store as XML Document in a variable -->
<s:variable b:name="filmdata-delta"
    b:select="id('delta-bxml-container')"
    b:scope="global"/>
<s:task b:action="bxml2string" b:variable="$filmdata-delta"/>
<s:task b:action="string2xml" b:variable="$filmdata-delta"/>

<!-- Load stylesheet and store as XML Document in a variable -->
<s:variable b:name="stylesheet" b:scope="global"/>
<s:task b:action="load"
    b:url="data/bxml2xml_standard.xsl"
    b:destination="$stylesheet"/>
<s:task b:action="string2xml" b:variable="$stylesheet"/>

<!-- Do the transformation, place output in a variable -->
<s:variable b:name="delta-xml" b:scope="global"/>
<s:task b:action="xsl-transform"
    b:stylesheet="$stylesheet"
    b:datasource="$filmdata-delta"
    b:destination="$delta-xml"/>
```

The process of loading the stylesheet is the same as in the earlier example, but the preparations for the input data are different. It needs to be converted from the BXML data type to the XML Document data type. This does not mean that it has already been transformed to the clean XML that the server wants, the tag and attribute names and their structure are not affected by these data type conversion routines. It's just a procedure for making the xsl-transform command recognize its input as something it can handle.

XML Data-binding with the BPC XSLT Engine

Overview

In our second case we will look at how to use the XSLT transformation engine offered by the Backbase Presentation Client (BPC). When you compare this process with the browser engine approach, you will notice that again the data is transferred as raw XML, but this time the transformation step between XML and BXML is contained inside the BPC.

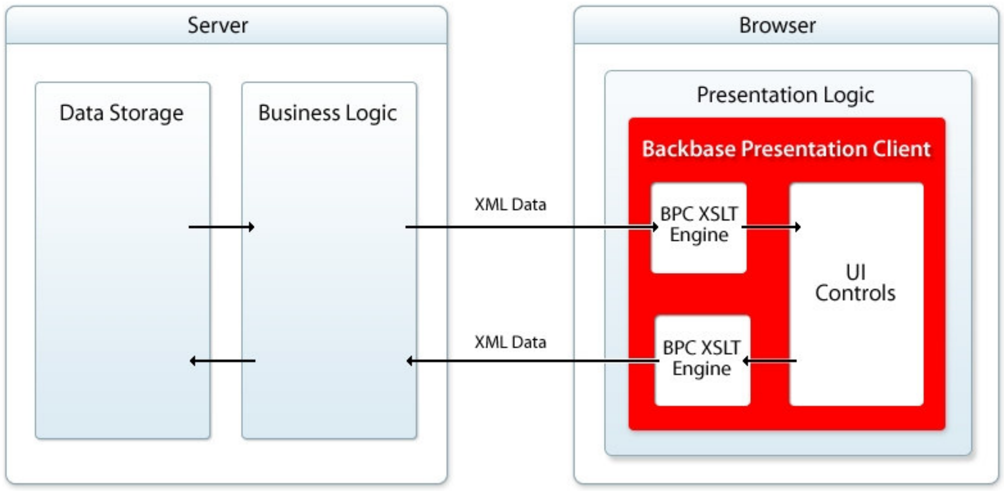


Figure 6 "XML data-binding with the BPC XSLT engine"

When the XML data arrives in the browser, the Backbase Presentation Client transforms it to BXML with its integrated XSLT engine. Before sending the data that was changed by the user back to the server, the BPC XSLT engine is used again to transform it from BXML to XML format.

Loosely coupled, flexible & integrated, XPath 2.0, cross-browser

Like in the case of using the browser XSLT engine, the data is transferred from the server to the browser as raw XML. This allows for a loosely coupled enterprise application architecture based on platform-independent XML communication.

By using the BPC XSLT engine you can take advantage of its excellent integration features to increase development productivity. You can start using variables in some powerful ways. For example, it becomes easy to set up a paging interface that shows (and transforms) only a range of records from a large dataset at a time, say numbers 101 to 120.

That same selection principle can be taken to a higher level by combining several user controls that access the same data source. This is perfect for interface patterns like the one used in Mozilla Thunderbird, where you have a folder category view, an email list view and an email detail view. The data displayed in the list and detail views need only be accessed and transformed based on what the user selects. Setting up this selective transformation pattern is easy with the integrated variables that can be used in the BPC XSLT engine.

Backbase 3.1 supports a large part of XPath 2.0, a significant improvement over XPath 1.0. The most notable additions are more XPath functions and more operators. This, together with the other features of the BPC XSLT engine, further strengthens its flexibility and level of control over transformations.
Finally, a big advantage of the BPC XSLT engine is that it is cross-browser compatible. It will work in all browsers without a problem.

Performance limits and stylesheet namespacing

The BPC XSLT engine has a limit to the amount of data it can reasonably process at a time. You do not want to transform too much data at once; otherwise the user interface might start responding slowly to changes.

Another characteristic that should be mentioned is that stylesheets written for the BPC engine use slightly different namespacing prefixes. Instead of the xsl namespace, it uses the s namespace.

Using the BPC XSLT engine: taking XML from the server

The preparations for doing a transformation with the BPC engine look like this:

```
<!-- Load data and store as string in a variable -->
<s:variable b:name="filmdata_string" b:scope="global"/>
<s:task b:action="load"
    b:url="data/filmdata.xml"
    b:destination="$filmdata_string"/>

<!-- Remove XML declaration from string if present -->
<s:if b:test="contains($filmdata_string, '?&gt;')">
    <s:task b:action="assign"
        b:target="$filmdata_string"
        b:select="substring-after($filmdata_string, '?&gt;')"/>
</s:if>

<!-- Move data to Data Island -->
<s:task b:action="assign"
    b:target="$filmdata_string"
    b:select="concat(
                    '&lt;s:xml b:name=&quot;filmdata_xml&quot; ',
                    'xmlns:s=&quot;http://www.backbase.com/s&quot; ',
                    'xmlns:b=&quot;http://www.backbase.com/b&quot;&gt;',
                    $filmdata_string,
                    '&lt;/s:xml&gt;'
                    )"/>
<s:task b:action="move" b:destination="." b:source="$filmdata_string"/>

<!-- Load stylesheet to BXML-Tree, where it becomes a Data Island -->
<s:task b:action="load"
    b:url="data/xml2bxml_bpc.xsl"
    b:destination="."/>
```

```
<!-- Transform the data and output the result directly to the BXML-Tree -->
<s:task b:action="transform"
    b:stylesheet="$xml2bxml"
    b:datasource="$filmdata_xml"
    b:destination="id('filmdisplay-container')"/>
```

What's happening here is that the raw XML is first loaded into a variable, at which point its data type is String. Then two things need to be taken care of:

1. It is possible that the loaded XML document starts with an XML declaration. If it does, this is stripped off because it would give a conflict when inserted into the BXML-Tree.

2. The string must then be wrapped inside an s:xml tag, and this s:xml tag be moved into the BXML-Tree.

What this accomplishes is that the XML data now has a special abstract data type that we will refer to in this article as a BXML Data Island. Although it has been moved into the BXML-Tree, it is not processed as deeply as a normal node in the BXML-Tree, which improves performance.

You may already have spotted the presence of a b:name attribute on the s:xml tag: `b:name=&quot;filmdata_xml&quot;`. This is the variable name that is used to refer to the Data Island when it is used as input for the transformation.
Some of these steps will be simplified in the next Backbase update.

The XSL stylesheet for the BPC XSLT engine becomes a Data Island as well. Because it doesn't need to be checked for an XML declaration and wrapped in an s:xml tag, it can be loaded directly into the BXML-Tree. The s:stylesheet tag in the XSL file also has an identifier: `b:name="xml2bxml"`. This identifier is used in the transform command, and the output of the transformation goes directly to the BXML-Tree.

Using the BPC XSLT engine: putting the changes back in XML format

The reverse process for transforming the changed data from BXML to XML format is simple and direct. The reference identifier to the stylesheet works because the b:name of the s:stylesheet element in the bxml2xml_bpc.xsl file has the value bxml2xml.

```
<!-- Load stylesheet to BXML-Tree, where it becomes a Data Island -->
<s:task b:action="load"
    b:url="data/bxml2xml_bpc.xsl"
    b:destination="."/>

<!-- Do transformation using XPaths to source and destination BXML nodes -->
<s:task b:action="transform"
    b:stylesheet="$bxml2xml"
    b:datasource="id('delta-bxml-container')"
    b:destination="id('delta-xml-container')"/>
```

For sending the changed data back to the server, the load command is used. The b:data attribute contains the XML:

```
<s:variable b:name="delta-xml" b:select="id('delta-xml-container')/*"/>
<s:task b:action="bxml2string" b:variable="$delta-xml"/>
<s:variable b:name="response"/>
<s:task b:action="load"
    b:method="POST"
    b:data="{concat('xmldata=', $delta-xml)}"
    b:destination="$response"
    b:url="save_xml.php"/>
```

XML Data-binding with the PHP XSLT Engine

Overview

Instead of doing the transformations between XML and BXML in the browser, you can do this on the server. As illustrated below, this means that the data will be transferred as BXML, and little processing needs to be done in the browser.

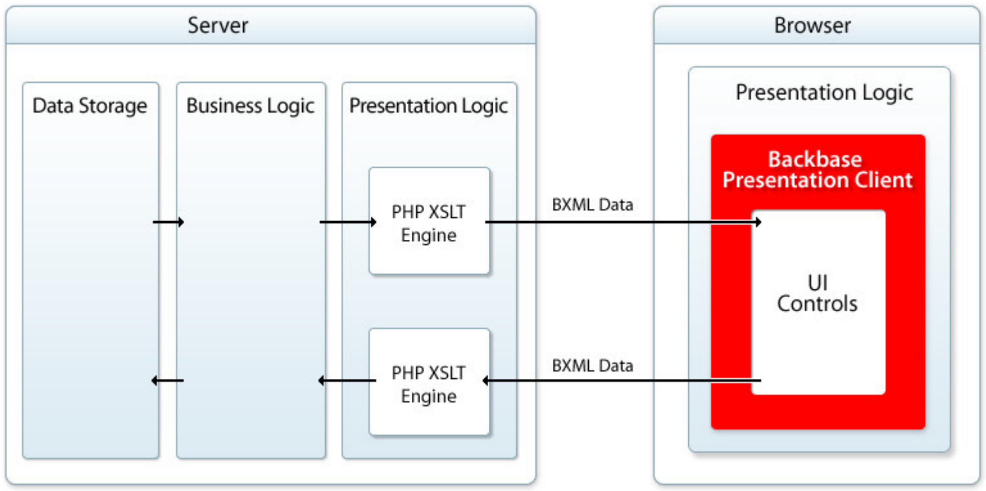


Figure 7 "XML data-binding with the PHP XSLT engine"

When the BXML data arrives in the browser, it can be immediately inserted in the BXML-Tree.

High performance

The advantage of delivering the data from a server straight to the Backbase Presentation Client in BXML format is optimum client side performance. The received data is displayed immediately and the client system's processing and memory resources are taxed less. This advantage is balanced with the drawback that more server roundtrips will be needed, because every transformation requires server activity. The server load will be higher than with client side transformations. This should be considered if you want to scale up the amount of simultaneous users.

Communication flexibility and maintenance

A server communicating in BXML can't communicate with other enterprise services quite as easily as when it is outputting and consuming raw XML. Also, because part of the presentation logic is done on the server, care must be taken not to mix it up with the business logic. This helps minimize maintenance costs later on.

Using the PHP XSLT engine: taking BXML from the server

The BXML code required to load the data into the BXML-Tree is very simple:

```
<!-- Load the BXML data directly into the BXML-Tree -->
<s:task b:action="load"
    b:url="load_bxml.php"
    b:destination="id('filmdisplay-container')"/>
```

Using the PHP XSLT engine: sending the changes back in BXML format

Sending the changed data back in BXML format is also very easy. With the send command you can specify an XPath to the BXML node containing the changed data:

```
<!-- Send the changed data to the server in BXML format -->
<s:variable b:name="response"/>
<s:task b:action="send"
    b:source="id('delta-container')"
    b:destination="$response"
    b:url="save_bxml.php"/>
```

When using the send command, remember that it wraps a b:backbase node around the BXML you are sending. This is done to ensure that the sent data has a single root element, and correctly declared namespaces. Take this into account when manipulating the data on the server.

If the server sends back some information to confirm how it has handled the saving action, this information will become available immediately in the response variable.

Security

Security is always an issue when manipulating data on the server and allowing user input. Every good RIA performs client side validation of the user input, but this is mainly a service provided to the user to improve the interaction flow. A browser client is ultimately controlled by the person running it on his computer, and potentially everything can be changed or faked. This is why people can cause accidental or intentional damage if they manage to send malformed input to the server that does not get validated there. This is why server side validation must always be performed regardless of whether client side validation is implemented in your RIA. In the examples and templates in this article no server side validation is done, but you will have to add it in real-world deployments.

Conclusion

By using client side XSLT transformations to do your XML data-binding, you get a clean, flexible and easy to maintain architecture, especially when using the BPC XSLT engine. If you have large datasets or a heavy interface that already burdens the browser client, it is wise to use server side XSLT transformations. In that case, performance and responsiveness in the browser will be better.
Also, these approaches can be combined if necessary. In the examples above the same XSLT engine was used each time for going from XML to BXML and back, but this is not a necessity.

Basically, with Backbase 3.1 you have a lot of flexibility right out of the box, and you can always choose an approach that suits your project's needs.

Download complete code

A complete and working code template for each approach and this article in PDF format can be found in the starter kit directory of your Backbase installation, e.g. C:\backbase\3_1_1\starterkits\xml_databinding_3_1.

As a pre-requisite, the templates require PHP 5 with XSLT enabled in order to run completely and allow for saving the changed data.