



# XML Application Developer's Guide

---



VERSION 5

**Borland®**  
**JBuilder™**

Borland Software Corporation  
100 Enterprise Way, Scotts Valley, CA 95066-3249  
[www.borland.com](http://www.borland.com)

Refer to the file DEPLOY.TXT located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997, 2001 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Other product names are trademarks or registered trademarks of their respective holders.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JB5XML 1E0R0501

0102030405-9 8 7 6 5 4 3 2 1

PDF

# Contents

## Chapter 1

### **Introduction 1-1**

Contacting Borland developer support . . . . .	1-2
Online resources . . . . .	1-2
World Wide Web . . . . .	1-2
Borland newsgroups . . . . .	1-3
Usenet newsgroups . . . . .	1-3
Reporting bugs . . . . .	1-3
Documentation conventions . . . . .	1-4
Macintosh conventions . . . . .	1-5

## Chapter 2

### **Using JBuilder's XML features 2-1**

Overview . . . . .	2-1
Creation and validation of XML . . . . .	2-2
Creating XML-related documents . . . . .	2-2
DTD To XML wizard . . . . .	2-2
XML To DTD wizard . . . . .	2-4
Viewing XML documents . . . . .	2-5
JBuilder's XML viewer . . . . .	2-5
Validating XML documents . . . . .	2-7
Presentation of XML . . . . .	2-9
Cocoon XML publishing framework . . . . .	2-9
Transforming XML documents . . . . .	2-12
Applying internal stylesheets . . . . .	2-13
Applying external stylesheets . . . . .	2-13
Setting transform trace options . . . . .	2-15
XML configurations . . . . .	2-16
XML resources . . . . .	2-17
Programmatic manipulation of XML . . . . .	2-17
Creating a SAX handler . . . . .	2-18
Databinding . . . . .	2-20
BorlandXML . . . . .	2-20
Castor . . . . .	2-22
Interface to business data in databases . . . . .	2-23

## Chapter 3

### **Using JBuilder's XML database components 3-1**

Using the template-based components . . . . .	3-2
Setting properties for the template beans . . . . .	3-2
Using the component's customizer . . . . .	3-2
Using the Inspector . . . . .	3-8
XML query document . . . . .	3-8

Using the model-based components . . . . .	3-9
XML-DBMS . . . . .	3-10
JBuilder and XML-DBMS . . . . .	3-11
XML-DBMS wizard . . . . .	3-11
Setting properties for the model-based components . . . . .	3-15
Using the component's customizer . . . . .	3-15
Using the Inspector . . . . .	3-19

## Chapter 4

### **Tutorial: Validating and transforming XML documents 4-1**

Overview . . . . .	4-1
Step 1: Creating an XML document from a DTD . . . . .	4-2
Step 2: Editing the generated XML document with the data . . . . .	4-3
Step 3: Validating the XML document . . . . .	4-4
Step 4: Associating stylesheets with the document . . . . .	4-5
Step 5: Transforming the document using stylesheets . . . . .	4-7
Step 6: Setting transform trace options . . . . .	4-8

## Chapter 5

### **Tutorial: Creating a SAX Handler for parsing XML documents 5-1**

Overview . . . . .	5-1
Step 1: Using the SAX Handler wizard . . . . .	5-2
Step 2: Editing the SAX parser . . . . .	5-3
Step 3: Running the program . . . . .	5-5
Step 4: Adding attributes . . . . .	5-6
Source code for MySaxParser.java . . . . .	5-8

## Chapter 6

### **Tutorial: DTD databinding with BorlandXML 6-1**

Overview . . . . .	6-1
Step 1: Generating Java classes from a DTD . . . . .	6-2
Step 2: Unmarshalling the data . . . . .	6-4
Step 3: Adding an employee . . . . .	6-5
Step 4: Modifying an employee . . . . .	6-6
Step 5: Running the completed application . . . . .	6-7

<b>Chapter 7</b>	
<b>Tutorial: Schema databinding with Castor</b>	<b>7-1</b>
Overview . . . . .	7-1
Step 1: Generating Java classes from a schema. . . . .	7-2
Step 2: Unmarshalling the data. . . . .	7-4
Step 3: Adding an employee . . . . .	7-5
Step 4: Modifying the new employee data . . .	7-6
Step 5: Running the completed application. . .	7-7

<b>Chapter 8</b>	
<b>Tutorial: Transferring data with the model-based XML database components</b>	<b>8-1</b>
Getting started . . . . .	8-2
Creating the map and SQL script files. . . . .	8-3
Entering JDBC connection information . . .	8-4
Testing the connection . . . . .	8-5
Specifying the file names . . . . .	8-5
Creating the database table(s) . . . . .	8-6
Working with the sample test application. . .	8-8
Using XMLDBMSTable's customizer. . . . .	8-8
Selecting a JDBC connection. . . . .	8-9
Transferring data from an XML document to the database table . . . . .	8-9

Transferring data from a database table to an XML document . . . . .	8-10
Using XMLDBMSQuery's customizer . . .	8-14
Selecting a JDBC connection . . . . .	8-14
Transferring data with a SQL statement . . . . .	8-14
Map files for the XMLDBMSQuery component . . . . .	8-15

<b>Chapter 9</b>	
<b>Tutorial: Transferring data with the template-based XML database components</b>	<b>9-1</b>
Getting started . . . . .	9-2
Working with the sample test application . . .	9-2
Using XTable's customizer . . . . .	9-3
Entering JDBC connection information . . . . .	9-3
Transferring data from the database table to an XML document . . . . .	9-4
Using XQuery's customizer . . . . .	9-6
Selecting a JDBC connection . . . . .	9-7
Transferring data with a SQL statement . . . . .	9-7

<b>Index</b>	<b>I-1</b>
--------------	------------

# Introduction

XML support is a feature of JBuilder Professional and Enterprise.

The *XML Application Developer's Guide* explains how to use JBuilder's XML features and contains the following chapters:

- Chapter 2, "Using JBuilder's XML features"

Explains how to use JBuilder's XML features for creating, validating, and presenting XML documents.

This is a feature of JBuilder Enterprise.

Also includes "Programmatic manipulation of XML" on page 2-17, which explains how to create a SAX parser and manipulate your XML data programmatically using several databinding solutions.

This is a feature of JBuilder Enterprise.

- Chapter 3, "Using JBuilder's XML database components"

Explains how to use the XML model and template bean components for database queries and transfer of data between XML documents and databases.

- Tutorials

- Chapter 4, "Tutorial: Validating and transforming XML documents"

These tutorials are available in JBuilder Enterprise.

- Chapter 5, "Tutorial: Creating a SAX Handler for parsing XML documents"
- Chapter 6, "Tutorial: DTD databinding with BorlandXML"
- Chapter 7, "Tutorial: Schema databinding with Castor"
- Chapter 8, "Tutorial: Transferring data with the model-based XML database components"
- Chapter 9, "Tutorial: Transferring data with the template-based XML database components"

## Contacting Borland developer support

---

Borland offers a variety of support options. These include free services on the Internet, where you can search our extensive information base and connect with other users of Borland products. In addition, you can choose from several categories of support, ranging from support on installation of the Borland product to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at <http://www.borland.com/devsupport/>, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

### Online resources

---

You can get information from any of these online sources:

<b>World Wide Web</b>	<a href="http://www.borland.com/">http://www.borland.com/</a>
<b>FTP</b>	<a href="ftp.borland.com">ftp.borland.com</a> Technical documents available by anonymous ftp.
<b>Listserv</b>	To subscribe to electronic newsletters, use the online form at: <a href="http://www.borland.com/contact/listserv.html">http://www.borland.com/contact/listserv.html</a> or, for Borland's international listserver, <a href="http://www.borland.com/contact/intlist.html">http://www.borland.com/contact/intlist.html</a>

### World Wide Web

---

Check [www.borland.com](http://www.borland.com) regularly. The JBuilder Product Team will post white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- <http://www.borland.com/jbuilder/> (updated software and other files)
- <http://www.borland.com/techpubs/jbuilder/> (updated documentation and other files)
- <http://community.borland.com/> (contains our web-based news magazine for developers)

## Borland newsgroups

---

You can register JBuilder and participate in many threaded discussion groups devoted to JBuilder.

You can find user-supported newsgroups for JBuilder and other Borland products at <http://www.borland.com/newsgroups/>

## Usenet newsgroups

---

The following Usenet groups are devoted to Java and related programming issues:

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

**Note** These newsgroups are maintained by users and are not official Borland sites.

## Reporting bugs

---

If you find what you think may be a bug in the software, please report it in the JBuilder Developer Support page at <http://www.borland.com/devsupport/jbuilder/>. From this site, you can also submit a feature request or view a list of bugs that have already been reported.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) with the JBuilder documentation, you may email [jgpubs@borland.com](mailto:jgpubs@borland.com). This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input, because it helps us to improve our product.

# Documentation conventions

---

The Borland documentation for JBuilder uses the typefaces and symbols described in the table below to indicate special text.

**Table 1.1**    Typeface and symbol conventions

Typeface	Meaning
Monospace type	Monospaced type represents the following: <ul style="list-style-type: none"><li>• text as it appears onscreen</li><li>• anything you must type, such as “Enter Hello World in the Title field of the Application wizard.”</li><li>• file names</li><li>• path names</li><li>• directory and folder names</li><li>• commands, such as SET PATH, CLASSPATH</li><li>• Java code</li><li>• Java data types, such as boolean, int, and long.</li><li>• Java identifiers, such as names of variables, classes, interfaces, components, properties, methods, and events</li><li>• package names</li><li>• argument names</li><li>• field names</li><li>• Java keywords, such as void and static</li></ul>
<b>Bold</b>	Bold is used for java tools, bmj (Borland Make for Java), bcj (Borland Compiler for Java), and compiler options. For example: <b>javac, bmj, -classpath</b> .
<i>Italics</i>	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
<i>Keycaps</i>	This typeface indicates a key on your keyboard. For example, “Press <i>Esc</i> to exit a menu.”
[ ]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.
< >	Angle brackets in text or syntax listings indicate a variable string; type in a string appropriate for your code. Do not type the angle brackets. Angle brackets are also used for HTML tags.
...	In code examples, an ellipsis indicates code that is missing from the example. On a button, an ellipsis indicates that the button links to a selection dialog.



JBuilder is available on multiple platforms. See the table below for a description of platforms and directory conventions used in the documentation.

**Table 1.2** Platform conventions and directories

Item	Meaning
Paths	All paths in the documentation are indicated with a forward slash (/). For the Windows platform, use a backslash (\).
Home directory	The location of the home directory varies by platform. <ul style="list-style-type: none"> <li>• For UNIX and Linux, the home directory can vary. For example, it could be <code>/user/[username]</code> or <code>/home/[username]</code></li> <li>• For Windows 95/98, the home directory is <code>C:\Windows</code></li> <li>• For Windows NT, the home directory is <code>C:\Winnt\Profiles\[username]</code></li> <li>• For Windows 2000, the home directory is <code>C:\Documents and Settings\[username]</code></li> </ul>
<code>.jbuilder5</code> directory	The <code>.jbuilder5</code> directory, where JBuilder settings are stored, is located in the home directory.
<code>jbproject</code> directory	The <code>jbproject</code> directory, which contains project, class, and source files, is located in the home directory. JBuilder saves files to this default path.
Screen shots	Screen shots reflect JBuilder's Metal Look & Feel on various platforms.

## Macintosh conventions

---

JBuilder is designed to support Macintosh OS X so seamlessly that JBuilder will have the look and feel of a native application. The Macintosh platform has conventions of appearance and style that vary from JBuilder's own; where that happens, JBuilder supports the Mac look and feel. This means that there are some variations between what JBuilder looks like on the Mac and how it is presented in the documentation. For instance, this documentation uses the word "directory" where Mac uses the word "folder." For further information on Macintosh OS X paths, terminology, and UI conventions, please consult the documentation that comes with your OS X installation.



# Using JBuilder's XML features

## Overview

---

These are features of  
JBuilder Professional and  
Enterprise.

JBuilder provides several features and incorporates various tools to provide support for the Extensible Markup Language (XML). XML is a platform-independent method of structuring information. Because XML separates the content of a document from the structure, it can be a useful means of exchanging data. For example, XML can be used to transfer data between databases and Java programs. Also, because content and structure are separate, stylesheets can be applied to display the same content in different formats, such as Portable Document Format (PDF), HTML for display in a Web browser, and so on.

In working with XML, JBuilder separates functionality into several layers:

- Creation and validation of XML documents
- Presentation of XML documents
- Programmatic manipulation of XML documents
- Interface to business data in databases

These are features of  
JBuilder Enterprise.

**See also**

World Wide Web Consortium (W3C) at <http://www.w3.org/>  
The XML Cover Pages at  
<http://www.oasis-open.org/cover/sgml-xml.html> (or  
<http://xml.coverpages.org/>)  
XML.org at <http://xml.org/>  
xmlinfo at <http://www.xmlinfo.com/>

## Creation and validation of XML

---

Creation and validation  
are features of JBuilder  
Professional and  
Enterprise.

JBuilder provides a variety of features that allow you to create, edit, view, and validate your XML documents without ever leaving the development environment. You can use wizards to quickly create XML-related documents, view them in the XML viewer in a collapsible tree view, edit the text in JBuilder's editor which supports XML syntax highlighting, find errors, and finally, validate documents.

For a tutorial on creating and validating XML documents, see Chapter 4, "Tutorial: Validating and transforming XML documents."

### Creating XML-related documents

---

JBuilder provides wizards for creating several XML-related documents within the IDE:

- DTD To XML wizard
- XML To DTD wizard

These wizards are available from the right-click menu in the project pane and from the XML page of the object gallery (File | New).

**Tip** You can also create empty XML-related documents as follows, and the editor will recognize the file type and provide syntax highlighting:

- 1 Choose File | Open File.
- 2 Enter a file name and extension, such as `.dtd`, `.xml`, `.xsl`, and `.xsd`, in the File Name field.
- 3 Enter text in the file.
- 4 Save the file.
- 5 Add the file to the project with the Add To Project button.

### DTD To XML wizard

The DTD To XML wizard is a quick way to create an XML document from an existing DTD. This wizard creates an XML template from the DTD with `pcdata` placeholders for content that you replace with your own content.

To use the DTD To XML wizard,

- 1 Right-click the DTD file in the project pane and choose Generate XML. This will automatically enter the DTD file name in the Input DTD File field of the wizard.
- 2 Select the root element from the Root Element drop-down list.
- 3 Accept the default file name in the Output XML File field or click the ellipsis button to enter a file name for the XML document.

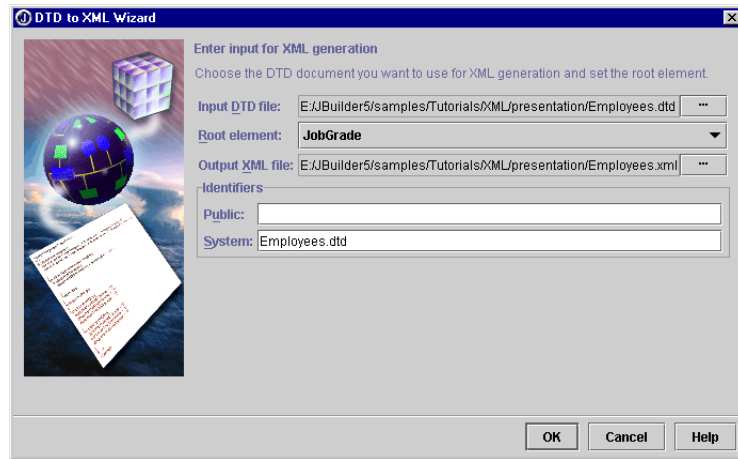
#### 4 Optional: Enter any identifiers for the DOCTYPE declaration.

- Public: enter the URI for the specified standards library.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML3.2 Final//EN">
```

- System: enter the name of the DTD file. This generates the DOCTYPE declaration. For example:

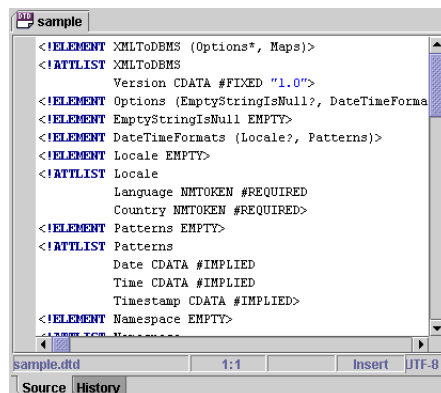
```
<!DOCTYPE root SYSTEM "Employees.dtd">
```

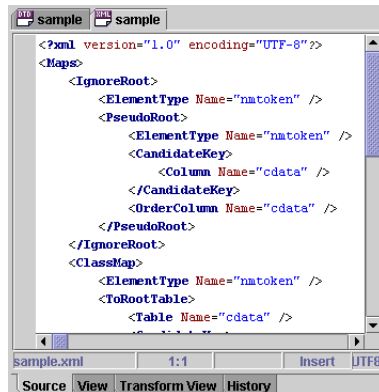


- 5 Click OK to close the wizard. The XML document is added to the project and appears in the project pane.

The wizard also handles attributes and converts the ATTLIST definitions in the DTD into attributes in the XML document.

**Figure 2.1** DTD with ATTLIST definitions



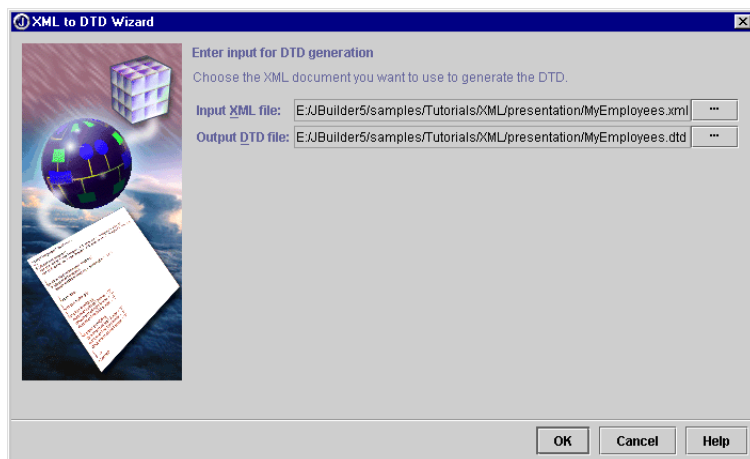
**Figure 2.2** XML created by the wizard

## XML To DTD wizard

The XML To DTD wizard is a quick way to create a DTD from an existing XML document.

To use the XML To DTD wizard,

- 1 Right-click the XML file in the project pane and choose Generate DTD to open the XML To DTD wizard. This will automatically enter the XML file name in the Input XML File field of the wizard.
- 2 Accept the default file name in the Output DTD File field or click the ellipsis button to enter a different file name for the XML document.



- 3 Click OK to close the wizard. The DTD is added to the project and appears in the project pane.

**Important** If attributes are included in the XML document, the XML To DTD wizard generates ATTLIST definitions for them in the DTD. See the “DTD To XML wizard” on page 2-2 for examples of attributes.

## Viewing XML documents

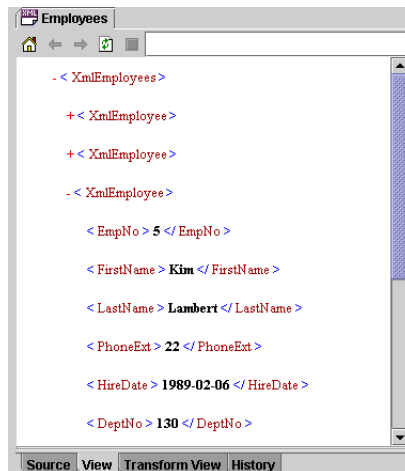
JBuilder provides an XML viewer to view your XML documents so you never need to leave the development environment. You can view XML using a user-defined stylesheet, JBuilder's default stylesheet, or without a stylesheet. JBuilder's XML viewer, which has JavaScript support, displays JBuilder's default stylesheet as a collapsible tree view.

### JBuilder's XML viewer

You can view an XML document in JBuilder by opening the XML document and selecting the View tab in the content pane. If the View tab is not available, you need to enable it on the XML page of the IDE Options dialog box (Tools | IDE Options).

If a CSS stylesheet is not available, JBuilder applies a default XSLT stylesheet that displays the document in a collapsible tree view. Note that the View tab ignores XSL stylesheets. For applying stylesheets, see "Transforming XML documents" on page 2-12.

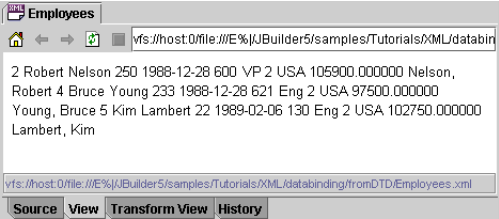
**Figure 2.3** XML view with default stylesheet



**Note** You can expand and collapse the tree view by clicking (+) symbols and the minus (-).

When the Apply Default Stylesheet option is turned off, you can view your XML document without any style. You can disable it on the XML page of the IDE Options dialog box.

Figure 2.4 XML view without a stylesheet



If your XML file contains a Cascading Style Sheet (CSS), JBuilder's XML viewer renders the document using that stylesheet.

For example, if you want to render the following XML with a stylesheet directly instead of transforming it, you can create a CSS file as shown and reference it in the XML document as follows:

```
<?xml-stylesheet type="text/css" href="cd_catalog.css"?>
```

Figure 2.5 XML document

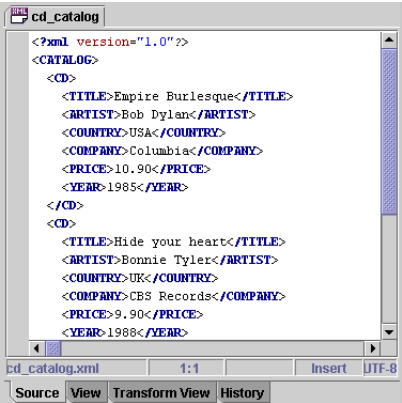
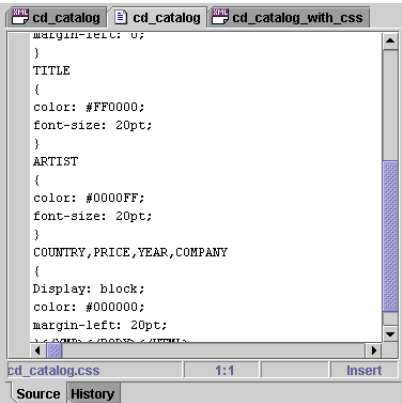


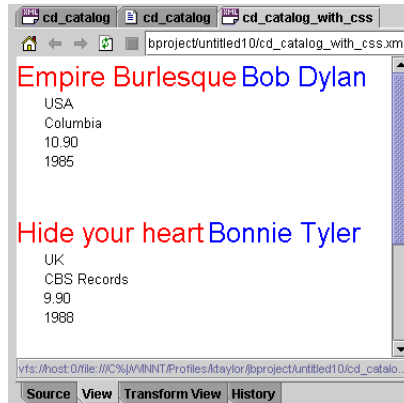
Figure 2.6 Cascading stylesheet source





The result of this is shown in the following image:

**Figure 2.7** XML document with cascading stylesheet applied



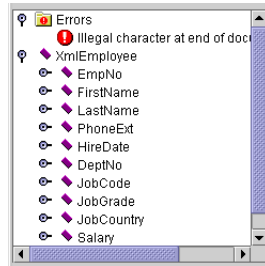
## Validating XML documents

In XML, there are two types of validation: well-formedness and grammatical validity. For a document to be well formed, it must follow the XML rules for the physical document structure and syntax. For example, all XML documents must have a root element. Also if the document has an internal DTD, all the entities must be declared. A well-formed document is not checked against an external DTD.

In contrast, a valid XML document is a well-formed document that also conforms to the stricter rules specified in the Document Type Definition (DTD) or schema. The DTD describes a document's structure and specifies which element types are allowed and defines the properties for each element.

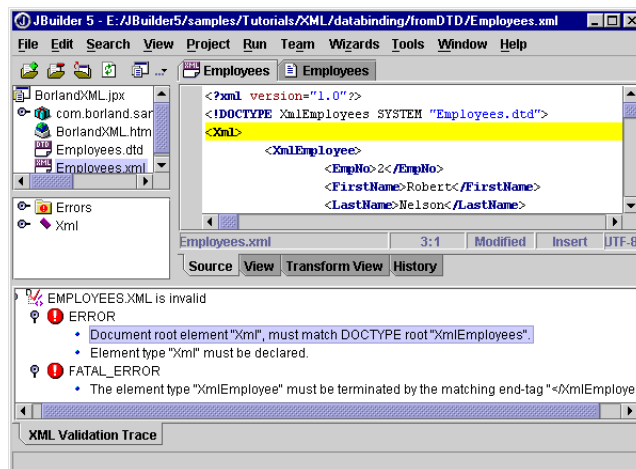
JBuilder integrates the Xerces parser to provide XML parsing for validating XML documents. For information about Xerces, see the Xerces documentation and samples available in the `extras` directory of the JBuilder full installation or visit the Apache web site at <http://xml.apache.org/>.

When viewing an open XML document in JBuilder, the structure pane displays the structure of the document. If the document isn't well formed, the structure pane displays an `Errors` folder that contains error messages. Use these messages to correct the errors in a document's structure. Click an error message in the structure pane to highlight it in the source code and double-click to move the cursor focus to the editor.

**Figure 2.8** Errors folder in structure pane

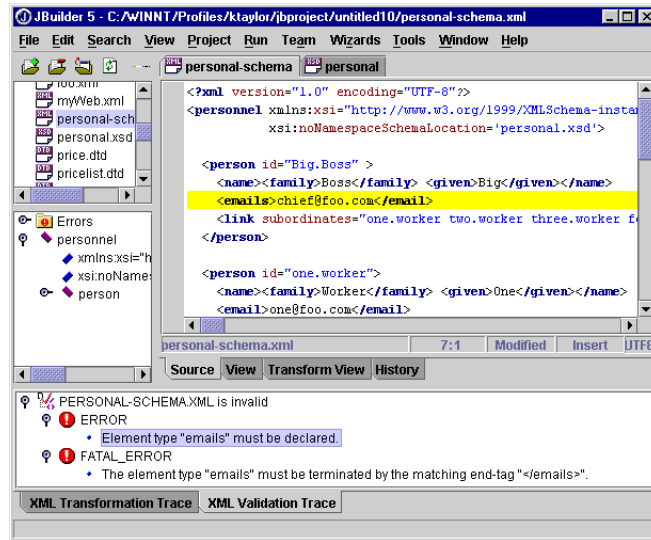
JBuilder can also validate the grammar of the XML in your document against the definitions in the DTD. With the XML document displayed in the content pane, right-click the XML file in the project pane and choose **Validate**. If the document is valid, a dialog box displays with a message that the document is valid. If the document has errors, the results are reported on an XML Validation Trace page in the message pane. Click an error message to highlight the error in the source code. Double-click a message to move the cursor focus to the source code.

The message pane displays both types of error messages: well formed and valid. If the DTD is missing, the document is considered invalid and a message displays in the message pane. After fixing the errors, re-validate the document to verify that it is valid.

**Figure 2.9** XML validation errors using DTD

JBuilder also supports validation of schema (XSD) files. As with DTDs, right-click the schema file in the project pane and choose **Validate**. Errors appear in the structure pane and/or the message pane. If a schema file is not available, a message displays in the message pane. If the schema is valid, a dialog box appears declaring it valid.

Figure 2.10 XML validation errors using schema



## Presentation of XML

Presentation is a feature of JBuilder Professional and Enterprise.

JBuilder provides tools for performing the tasks of presentation of XML documents:

- Cocoon as the presentation layer
- Validation of XML documents
- Transformation of XML documents

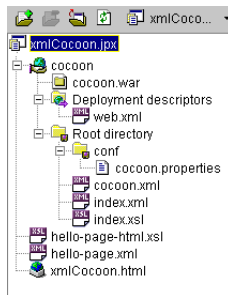
## Cocoon XML publishing framework

Cocoon, part of the Apache XML project, is integrated into JBuilder. It is a servlet-based, Java publishing framework for XML that allows separation of content, style, and logic and uses XSL transformation to merge them. Cocoon can also use logic sheets, Extensible Server Pages (XSP), to deliver dynamic content embedded with program logic written in Java. The Cocoon model divides web content into:

- XML creation: XML files are created by content owners who need to understand DTDs but don't need to know about processing.
- XML processing: the XML file is processed according to logic sheets. Logic is separate from the content.
- XSL rendering: the XML document is rendered by applying a stylesheet to it and formatting it according to the resource type (PDF, HTML, WML, XHTML, etc).

When you run the Cocoon Web Application wizard, Cocoon is configured to use the version of Cocoon bundled with JBuilder. Use the Cocoon Web Application wizard on the XML page of the object gallery (File | New | XML) to set up Cocoon after you start a new project:

- 1 Create a project using JBuilder's Project wizard (File | New Project).
- 2 Choose File | New and choose the XML tab of the object gallery.
- 3 Double-click the Cocoon Web Application icon to open the Cocoon Web Application wizard.
- 4 Accept the default Cocoon base.
- 5 Accept the Generate WAR option if you want to create a WAR file.
- 6 Click OK to close the wizard and generate the Cocoon files.
- 7 Select the project file in the project pane, right-click, and choose Make Project to generate the WAR file.
- 8 Expand the cocoon node in the project pane to see the Cocoon files generated by the wizard:



- cocoon.war - a web archive file
- web.xml - a web application deployment descriptor
- cocoon.properties - a properties file
- cocoon.xml - a configuration file
- index.xml - sample xml file
- index.xsl - sample stylesheet

You can edit most of these files directly in the editor if you want to make changes later without running the wizard again.

- 9 Add your existing XML and XSL files to the project using the Add To Project button on the project pane toolbar.

For more information on `web.xml` and the editor for the deployment descriptor, see the "Deployment descriptors" topics in "Working with WebApps and WAR files" and "Deploying your web application" in the *Web Application Developer's Guide*.

Open Cocoon's sample file, `index.xml`, and notice that it uses `index.xsl` as a stylesheet.

Figure 2.11 XML source code for index.xml

```
<?xml version="1.0"?>
<?xml-stylesheet href="index.xsl" type="text/xsl"?>
<?cocoon-process type="xslt"?>

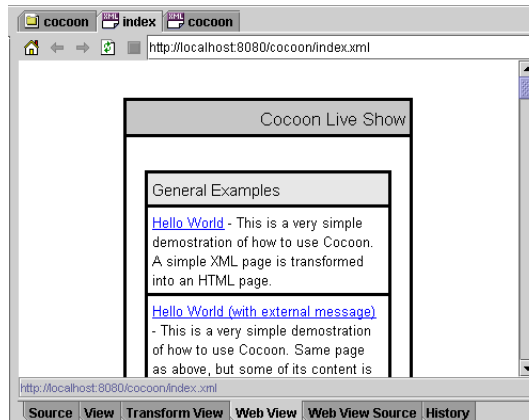
<samples>
  <group name="General Examples">
    <sample name="Hello World" url="hello/hello-
      This is a very simple demonstration of how t
      is transformed into an HTML page.
    </sample>
    <sample name="Hello World (with external mes
      This is a very simple demonstration of how t
      but some of its content is included using X
    </sample>
    <sample name="Hello World (with imported sty
      This is a very simple demonstration of how t
      but its stylesheet is an extention of the p
      of its properties.
    </sample>
  </group>
</samples>
```

Figure 2.12 Stylesheet source code for index.xsl

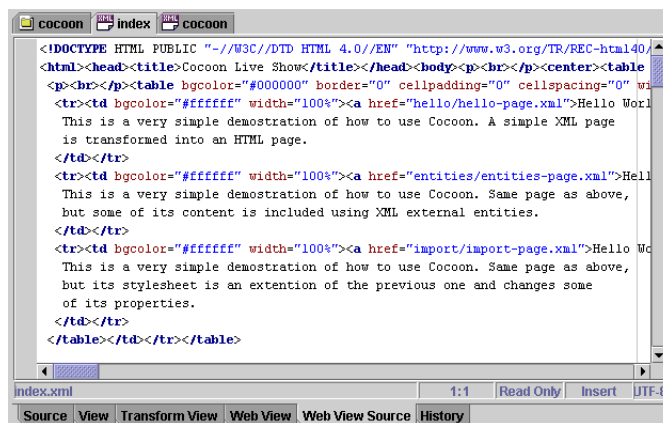
```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3
  <xsl:template match="samples">
    <xsl:processing-instruction name="cocoon-format">type=
    <html>
    <head>
      <title>Cocoon Live Show</title>
    </head>
    <body>
      <p><br /></p>
      <center>
        <table border="0" width="60%" bgcolor="#000000" cell
        <tr>
          <td width="100%">
            <table border="0" width="100%" cellpadding="4">
              <tr>
```

To run Cocoon, right-click the cocoon node in the project pane and choose Cocoon Run on the pop-up menu. Cocoon launches the currently configured servlet engine and inserts itself in the servlet environment, using information in the web.xml and cocoon.properties files the Cocoon Web Application wizard generated. You can modify cocoon.properties to add XSP (Extensible Server Pages) libraries and individual resources to each logic sheet.

Now, choose the Web View tab to see the Cocoon sample with the stylesheet applied.

**Figure 2.13** Web view of index.xml

To see the source code for the web view, choose the Web View Source tab.

**Figure 2.14** Web view source of index.xml

For complete information about using Cocoon, see the Cocoon documentation and samples in the `cocoon` directory of your JBuilder installation or visit the Apache web site at <http://xml.apache.org/cocoon/index.html>.

## Transforming XML documents

The process of converting an XML document to any other kind of document is called XML transformation. JBuilder incorporates Xalan as the stylesheet processor for transformation of XML documents and uses stylesheets written in Extensible Style Language Transformations (XSLT) for transformation. An XSL stylesheet contains instructions for

transforming XML documents from one document type into another document type (XML, HTML, PFD, WML, or other).

For information about Xalan, see the Xalan documentation and samples available in the `extras` directory of the full JBuilder installation or visit the Apache web site at <http://xml.apache.org/>.

## Applying internal stylesheets

To apply a stylesheet to an XML document, choose the XML file's Transform View tab in the content pane. If the document contains an XSLT processing instruction and just a single stylesheet, the stylesheet is applied to the XML document. If a tree view displays instead, press the Default Stylesheet button on the transform view toolbar to disable the tree view. The transformed document, held in a temporary buffer, displays on the Transform View tab of the content pane with the stylesheet applied. A Transform View Source tab also displays, so you can view the source code for that transformation.



If you want to apply another internal stylesheet listed in the stylesheet instruction in the document, choose it from the stylesheet drop-down list on the transform view's toolbar.

**Figure 2.15** Transform view toolbar



**Table 2.1** Transform view toolbar buttons

Button	Description
Default stylesheet	Applies the default JBuilder stylesheet, which is a collapsible tree view.
Refresh	Refreshes the view.
Set trace options	Opens Set Transform Trace Options dialog box where you set traces for the application process.
Add Stylesheets	Opens the Configure Node Stylesheets dialog box where you can associate stylesheets with a document.

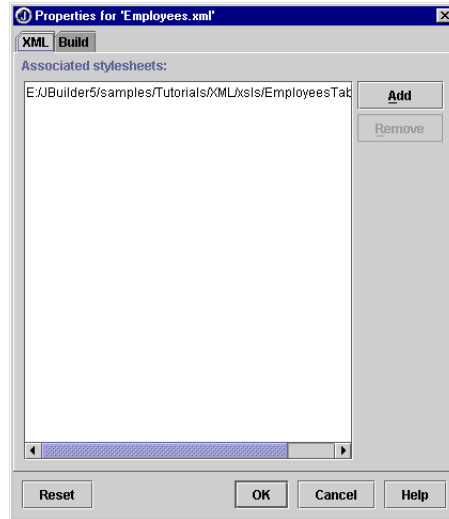
## Applying external stylesheets

You can also apply external stylesheets to a document. First, you need to associate them with the XML document. There are several ways to add and remove external stylesheets associated with a document:

- Right-click the XML document in the project pane and choose Properties.



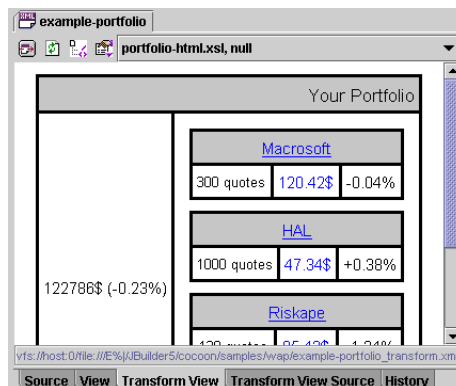
- Click the Add Stylesheets button on the transform view toolbar.



Then, use the Add and Remove buttons to add and remove selected stylesheets. After the stylesheets are associated with the document, they appear in the stylesheet drop-down list along with the internal stylesheets on the transform view toolbar.

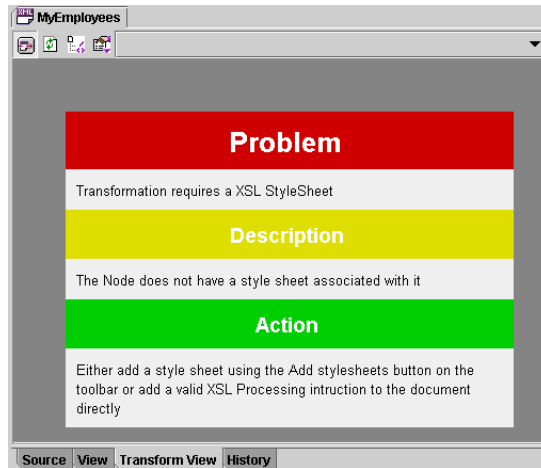
Next, choose the Transform View tab and select an external stylesheet from the drop-down list to apply it. If the document displays in a tree view, choose the Default Stylesheet button on the transform view toolbar to disable it.

**Figure 2.16** Transform view with external stylesheet applied

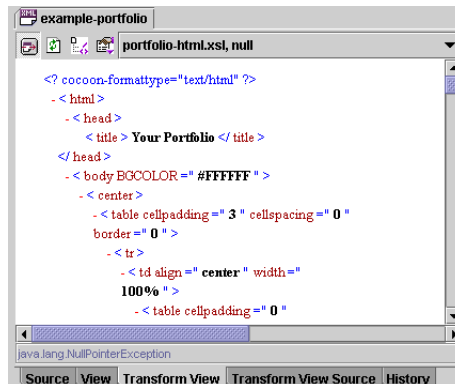


If a stylesheet is not available, a message displays in the transform view indicating that a stylesheet is not associated with the document.



**Figure 2.17** Transform view without a stylesheet

To display the results of the transformation in a tree view using JBuilder's default stylesheet, choose the Default Stylesheet button on the transform view's toolbar. This is useful if the output of a transformation is another XML document without a stylesheet.

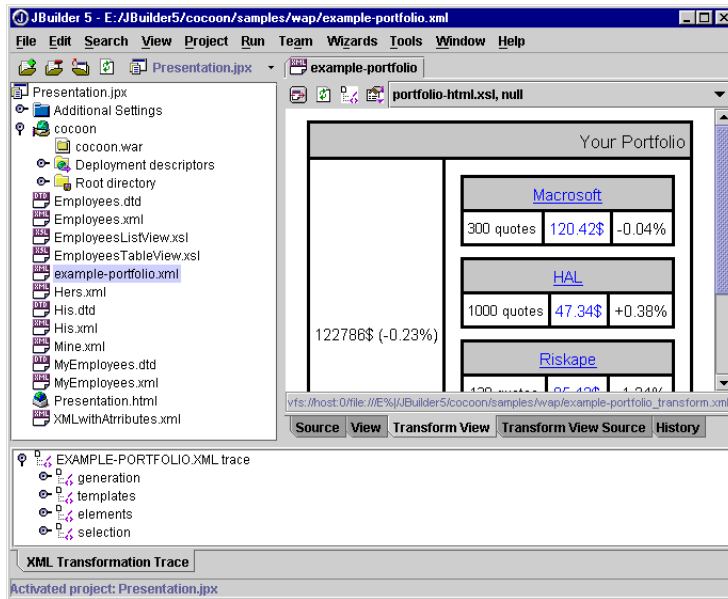
**Figure 2.18** Transform view with default stylesheet tree view

## Setting transform trace options

You can set transform trace options so that when a transformation occurs, you can see a trace of the application process. These options include Generation, Templates, Elements, and Selections. To enable tracing, choose Tools | IDE Options, choose the XML tab, and check the trace options you want. You can also set these options by choosing the Set Trace Options button on the transform view's toolbar. The traces appear in the message pane. Clicking on a trace highlights the corresponding source



code. Double-clicking a trace changes the focus to the source code in the editor so you can begin editing.

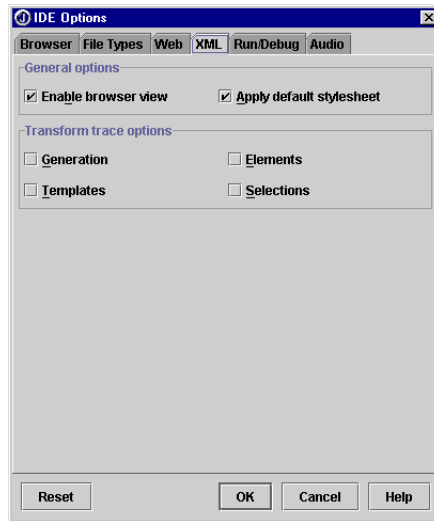


## XML configurations

This is a feature of JBuilder Professional and Enterprise.

You can set XML configurations in the IDE Options dialog box. Choose Tools | IDE Options and click the XML tab to set the following options:

- General options
  - Enable Browser View: enables JBuilder's XML viewer. When this option is enabled, a View tab is available in the content pane.
  - Apply Default Stylesheet: JBuilder's default stylesheet displays an XML document in a tree view.
- Transform Trace options: set transform trace options so that after a transformation occurs, you can follow the sequence in which the various stylesheet elements were applied. The trace options include:
  - Generation
  - Templates
  - Elements
  - Selections



## XML resources

---

Additional XML resources are included in the full JBuilder install in the `extras` directory: Xerces, Xalan, Castor, and Borland XML. Documentation, Javadoc, and samples are also included.

## Programmatic manipulation of XML

---

Programmatic manipulation is a feature of JBuilder Enterprise.

XML is typically manipulated programmatically either through parsers or through a more specific databinding solution JBuilder supports both approaches and provides tools for both:

- A SAX wizard and library definitions for DOM and JAXP.
- BorlandXML for generating Java sources from DTD
- Castor for generating Java sources from Schema

Pre-defined libraries, which are bundled with JBuilder, can be added to your project: JDOM, JAXP, Xerces, BorlandXML, Castor, and so on. You can add these to your project in the Project Properties dialog box. Choose Project | Project Properties and choose the Paths page. Choose the Required Libraries tab and add the libraries. Once the libraries are added, JBuilder's CodeInsight has access to them and can display context-sensitive pop-up windows within the editor that show accessible data members and methods, classes, parameters expected for the method being coded, as well as drilling down into source code.

## Creating a SAX handler

---

This is a feature of JBuilder Enterprise.

SAX, the Simple API for XML, is a standard interface for event-based XML parsing. There are two types of XML APIs: tree-based APIs and event-based APIs.

A tree-based API, which compiles an XML document into an internal tree structure, allows an application to navigate the tree. This tree-based API is currently being standardized as a Document Object Model (DOM).

SAX, an event-based API, reports parsing events directly to the application through callbacks. The application implements handlers to deal with the different events, similar to event handling in a graphical user interface.

For example, an event-based API looks at this document:

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

<page>
  <title>Event-based example</title>
  <content>Hello, world!</content>
</page>
```

and breaks it into these events:

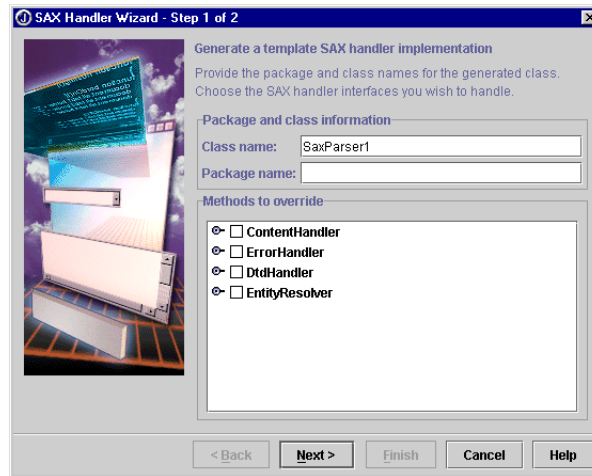
```
start document
start element: page
start element: title
characters: Event-based example
end element: title
start element: content
characters: Hello, world!
end element: content
end element: page
end document
```

JBuilder makes it easier to use SAX to manipulate your XML programmatically. The SAX Handler wizard creates a SAX parser implementation template that includes just the methods you want to implement to parse your XML.

To use the SAX Handler wizard,

- 1 Choose File | New to open the object gallery, click the XML tab, and double-click the SAX Handler wizard icon to open the wizard.

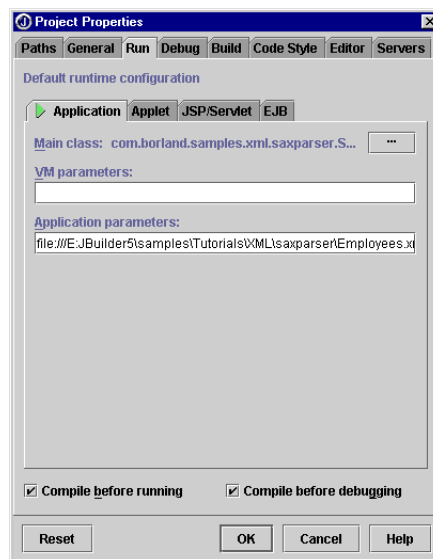
- 2 Specify the name of the class and package names or accept the default names.



- 3 Select the interfaces and methods you want to override and click Next.
- 4 Select the SAX Parser options you want and click Finish.

The wizard generates a class that implements a SAX parser. You must then fill in the method bodies with the code to complete the implementation.

- 5 Configure the Run page of Project Properties by selecting the main class to run and specifying the XML file to parse in the Application Parameters field.



For information about SAX, visit  
<http://www.megginson.com/SAX/index.html>.

For a tutorial on the SAX Handler wizard, see Chapter 5, "Tutorial: Creating a SAX Handler for parsing XML documents."

## Databinding

---

This is a feature of  
JBuilder Enterprise.

Databinding is a means of accessing data and manipulating it, then sending the revised data back to the database or displaying it with an XML document. The XML document can be used as the transfer mechanism between the database and the application. This transfer is done by binding a Java object to an XML document. The databinding is implemented by generating Java classes to represent the constraints contained in a grammar, such as in a DTD or an XML schema. You can then use these classes to create XML documents that comply to the grammar, read XML documents that comply to the grammar, and validate XML documents against the grammar as changes are made to them.

JBuilder offers several databinding solutions: BorlandXML and open-source Castor. BorlandXML works with DTD files, while Castor generates Java classes from schema files (.xsd).

**See also** "The XML Databinding Specification" at  
<http://www.oasis-open.org/cover/xmlDataBinding.html>

### BorlandXML

BorlandXML provides a databinding mechanism that hides the details of XML and reduces code complexity with ease of maintenance.

BorlandXML is a template-based programmable class generator used to generate JavaBean classes from a Document Type Definition (DTD). You then use the simple JavaBean programming convention to manipulate XML data without worrying about the XML details.

BorlandXML uses DTDs in a two-step process to generate Java classes. In the first step, BorlandXML generates a class model file from a DTD. The class model file is an XML file with .bom extension. This file describes a high-level structure of the target classes and provides a way to customize these classes. In the second step, BorlandXML generates Java classes from the .bom file (class model XML file).

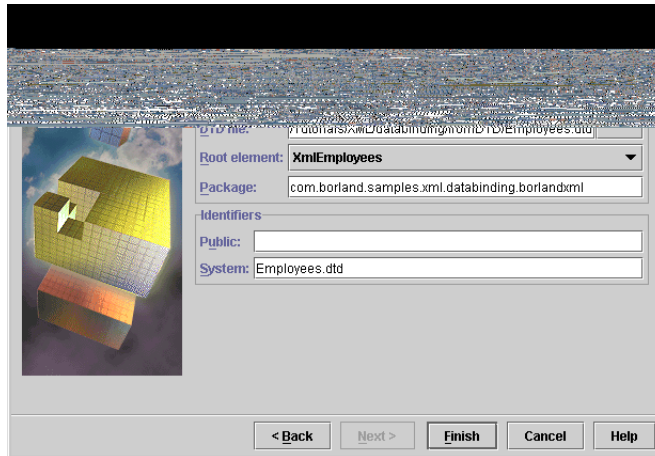
BorlandXML supports several features:

- JavaBean manipulation: manipulates a bean to construct an XML document or access data in the document.
- Marshalling and unmarshalling: conversion between Java and XML.
  - Marshalling: writes out an XML document from JavaBean objects - Java to XML.
  - Unmarshalling: reads an XML document into JavaBean objects - XML to Java.

- Document validation: validates JavaBean objects before marshalling objects to XML or after unmarshalling an XML document back to JavaBean objects.
- PCDATA customization: allows PCDATA to be customized to support different primitive data types, such as Integer and Long, and to support customized property names.
- Variable names: allows generated variable names for elements and attributes to have customized prefix and suffix.

To generate Java classes from a DTD using the Databinding wizard,

- 1 Right-click the DTD file in the project pane and choose Generate Java to open the Databinding wizard. By doing this, the DTD File field in the wizard is automatically filled in with the file name. The Databinding wizard is also available on the XML tab of the object gallery (File | New).
- 2 Select BorlandXML as the Databinding Type, which is DTD-based only, and click Next.
- 3 Fill in the required fields, such as the name and location of the DTD being used, the root element, and the package name.
- 4 Enter a PUBLIC or SYSTEM identifier which is inserted into the DOCTYPE declaration.



- 5 Click Finish.
- 6 Expand the generated package node in the project pane to see the .java files generated by the wizard.

For a tutorial on databinding with BorlandXML, see Chapter 6, “Tutorial: DTD databinding with BorlandXML.”

## Castor

Castor is an XML databinding framework that maps an instance of an XML schema to an object model which represents the data. This object model includes a set of classes and types as well as descriptors which are used to obtain information about a class and its fields.

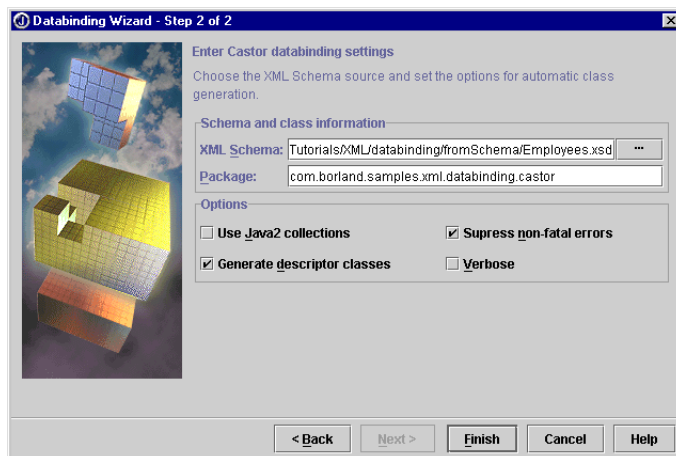
Castor uses a marshalling framework that includes a set of `ClassDescriptors` and `FieldDescriptors` to describe how an `Object` should be marshalled and unmarshalled from XML.

For those not familiar with the terms “marshal” and “unmarshal”, it’s simply the act of converting a stream (sequence of bytes) of data to and from an `Object`. The act of “marshalling” consists of converting an `Object` to to a stream, and “unmarshalling” from a stream to an `Object`.

Castor uses schema to create Java classes instead of DTDs. Schemas (XSD), more robust and flexible, have several advantages over DTDs. Schemas are XML documents, unlike DTDs which contain non-XML syntax. Schemas also support namespaces, which are required to avoid naming conflicts, and offer more extensive data type and inheritance support.

To generate Java classes from an XML schema, use the Databinding wizard as follows:

- 1 Right-click the schema file (XSD) in the project pane and choose Generate Java to open the Databinding wizard. By doing this, the XML Schema File field in the wizard is automatically filled in with the file name. The Databinding wizard is also available on the XML tab of the object gallery (File | New).
- 2 Select Castor as the Databinding Type, which supports XML schemas, and click Next.
- 3 Fill in the required fields, such as the package name, and specify the options you want.





- 4 Click Finish.
- 5 Expand the generated package node in the project pane to see the .java files generated by the wizard.

**Note** By default, Castor's marshaller writes XML documents without indentation, because indentation inflates the size of the generated XML documents. To turn indentation on, modify the Castor properties file with the following content: `org.exolab.castor.indent=true`. There are also other properties in this file that you may want to modify. The `castor.properties` file is created automatically by the Databinding wizard in the source directory of the project.

For a tutorial on databinding with Castor, see Chapter 7, "Tutorial: Schema databinding with Castor."

Castor samples and documentation are provided in the `extras` directory of the JBuilder full install or visit the Castor web site at <http://castor.exolab.org>.

## Interface to business data in databases

---

Interfacing to business data is a feature of JBuilder Enterprise.

XML database support in JBuilder falls into two categories - model-based and template-based. The model-based solution uses a map document that determines how the data transfers between an XML structure and the database metadata. The model-based components, `XMLDBMSTable` and `XMLDBMSQuery`, are implemented using XML-DBMS, an Open Source XML middleware that is bundled with JBuilder.

The template-based solution works with a template, a set of rules. The template-based components, `XTable` and `XQuery`, are very flexible as there is no predefined relationship between the XML document and the set of database metadata you are querying.

For more information on XML database components, see Chapter 3, "Using JBuilder's XML database components."

**See also** XML-DBMS at <http://www.rpbouret.com/xmldbms/>



## Using JBuilder's XML database components

This is a feature of JBuilder Enterprise.

JBuilder's XML database support is available through a set of components on the XML page of the component palette. The runtime code for the beans is provided as part of a redistributable library in Xbeans.jar.

The XBeans library consists of two types of XML database components

- Template-based components
- Model-based components

To use template-based components, you supply an SQL statement, and the component generates an appropriate XML document. The SQL you provide serves as the template that is replaced in the XML document as the result of applying the template. The template-based solution is very flexible as there is no predefined relationship between the XML document and the set of database metadata you are querying. Although template-based components are flexible in getting data out of a database and into an XML document, the format of the XML document is flat and relatively simple. In addition, the template-based components can generate HTML documents based on default style sheets or on a custom style sheet provided by the user.

Model-based components use a map document that determines how the data transfers between an XML structure and the database metadata. Because the user specifies a map between an element in the XML document to a particular table or column in a database, deeply nested XML documents can be transferred to and from a set of database tables. The model-based components are implemented using XML-DBMS, an Open Source XML middleware that is bundled with JBuilder.

## Using the template-based components

---

The two template-based components are `XTable` and `XQuery`, the first and second XML components on the JBuilder component palette.

For a tutorial about using the template-based XML components, see Chapter 9, “Tutorial: Transferring data with the template-based XML database components.”

To begin working with these components, select either of them on the XML page of the component palette and drop it in the UI Designer or in the structure pane to add the component to your application.

### Setting properties for the template beans

---

There are three ways to set the properties of the two template-based components:

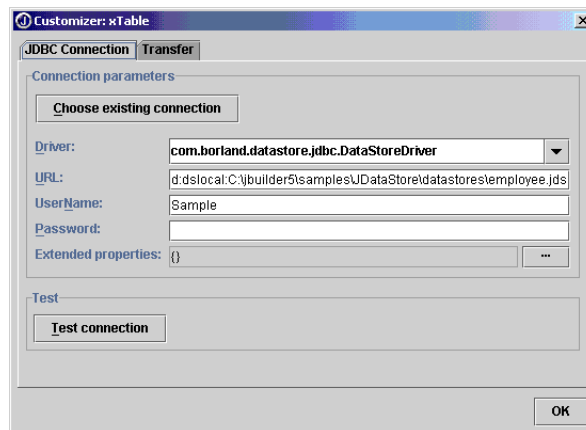
- Using the component’s customizer
- Using the Inspector
- Using an XML query document

#### Using the component’s customizer

Each XML database component has its own customizer. Using a component’s customizer is the easiest way to set the component’s properties. You can even test your JDBC connection, perform the transfer to view the generated document, and see the Document Object Model (DOM).

To display a component’s customizer, right-click the component in the structure pane and choose Customizer on the pop-up menu.

This is the customizer for `XTable`:



## JDBC Connection

The JDBC Connection page lets you specify the JDBC connection to the database that contains the data you want to use to create an XML document. It contains these fields:

Driver	Specify the JDBC driver to use from the drop-down list. Those drivers displayed in black are drivers you have installed. Drivers shown in red are not available on your system.
URL	Specify the URL to the data source that contains the information you want to use to create an XML document. When you click in the field, it displays the pattern you must use to specify the URL depending on your choice of JDBC driver.
User Name	Enter the user name for the data source, if any.
Password	Enter the data source password, if one is required.
Extended Properties	Add any extended properties you need. Click the ... button to display the Extended Properties dialog box you use to add new properties.

If you already have one or more connections defined within JBuilder to data sources, click the Choose Existing Connection button and select the connection you want. Most of the Connection Properties are then filled in automatically for you.

To test to see if your JDBC connection is correct, click the Test Connection button. The customizer reports whether the connection was successful or failed.

Once you have a successful connection, click the Transfer tab.

The screenshot shows the 'Customizer: xQuery' dialog box with the 'Transfer' tab selected. The 'JDBC Connection' tab is also visible. The 'Transfer information' section includes fields for 'Query File', 'Output File' (set to 'xQueryOut.html'), and 'XSL File', each with a browse button (...). The 'Column format' section has radio buttons for 'As Elements' (selected) and 'As Attributes'. The 'Output format' section has radio buttons for 'XML' and 'HTML' (selected). The 'Element names' section has text boxes for 'Document' (set to 'XmlEmployees') and 'Row' (set to 'XmlEmployee'). There is an 'Ignore Nulls' checkbox. The 'SQL' field contains the query 'select \* from "XmlEmployee"'. The 'DefaultParams' field is empty. At the bottom, there are 'Transfer', 'View HTML', and 'Reset' buttons, and an 'OK' button at the very bottom right.

## Transfer

Query File	An XML query document. Using an XML query document is optional. If you use an XML query document, you won't be filling in any of the other fields in the customizer except the Output File name and optionally the XSL File's name as the query document will specify your property settings. For more information about creating and using an XML query document, see "XML query document" on page 3-8.
Output File	Specify the name of the XML or HTML file you want to generate.
XSL File	Specify the name of the XSL style sheet file you want used to transform the output file, if any. If no file is specified, a default style sheet is generated and placed in the same directory as the output file. The name of the XSL file generated is <code>JBuilderDefault.xsl</code> . The XSL file can be copied and then modified to create a more custom presentation. If you want to edit the XSL file, make sure the XSL File name property is set to point to the modified file. Note that JBuilder won't override a previously existing default stylesheet.
Column Format	Specify whether you want the columns of the data source to appear as elements or as attributes in the generated XML file.
Output Format	Specify whether you want the generated file to be in XML or HTML format.
Element Names	Specify a name for the Document element and another for the Row element.
Ignore Nulls	Check this check box if you want nulls to be ignored in your XML output. If this check box remains unchecked, "null" will be used as a placeholder.
Table Name	Specify the name of the table that contains the data you are interested in.

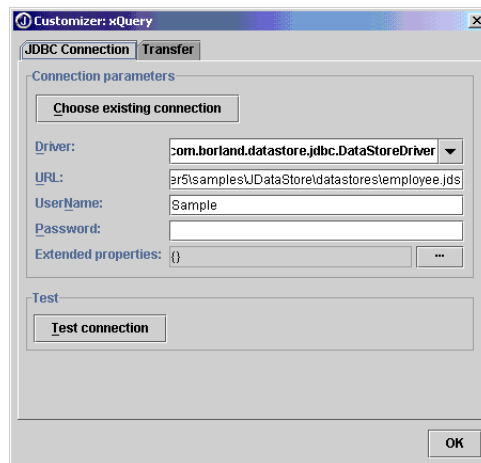
**Keys**

Specify the key(s) that identifies the row(s) in the table you want to become part of the generated XML document. To specify a key, click the Add button. In the string array property editor that appears, click the Add button to add an item to the array. Change the name of the added item to a column in the table. Continue adding keys with the property editor until you've added all the keys you want. If you specify a table name but don't specify any keys, all the rows of the table will be returned.

**DefaultParams**

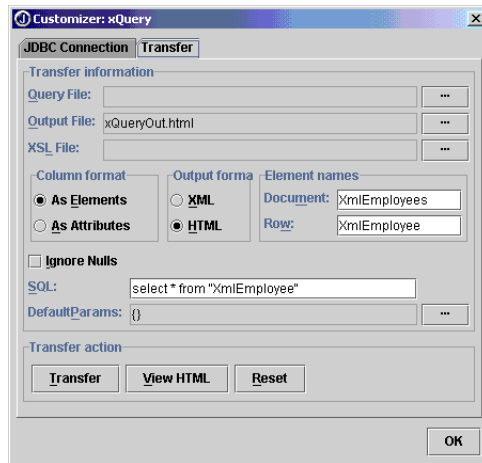
Use this field to specify a default parameter for a parameterized query. If you specified a value for the Key field, you must specify a default parameter for the column or columns specified as a key. Click the Add button to add any default parameters to your query. In the Default Params dialog box that appears, click the Add button to add a default parameter. In the new blank line that is added, specify the name of the parameter as Param Name, and the value of the parameter as the Param Value. For example, if the key is EMP\_NO, specify EMP\_NO as the Param Name and specify the value to be found in the EMP\_NO column. Remember to put single quotes around any string values. For more information about adding default parameters, see "Specifying parameters" on page 3-6.

The customizer for the `xQuery` looks very similar. Like `xTable`, it has a JDBC Connection page:



Fill this page in as you would for `xTable` and test your connection.

The Transfer page of the customizer for `xQuery` differs from that of `xTable` in that it has an SQL field that replaces the Table Name and Keys fields:



In the SQL field, you can specify any SQL statement. If your SQL statement is a parameterized query, you must specify a default parameter for each parameterized variable.

### Specifying parameters

If the query you are using is a parameterized query, you must specify a value for the default parameter before generating the XML or HTML file. During runtime, you can override the default parameter value with another parameter value. If no parameter is supplied at runtime, the default parameter is used.

To see how to use parameters and default parameters, look at a sample query:

```
Select emp_name from employee where emp_no = :emp_no
```

Let's say the table `Employee` contains the following entries:

emp_no	emp_name
1	Tom
2	Dick

There are two ways to provide the parameter `:emp_no`. You can use a default parameter and/or a parameter supplied at runtime. These are the possibilities

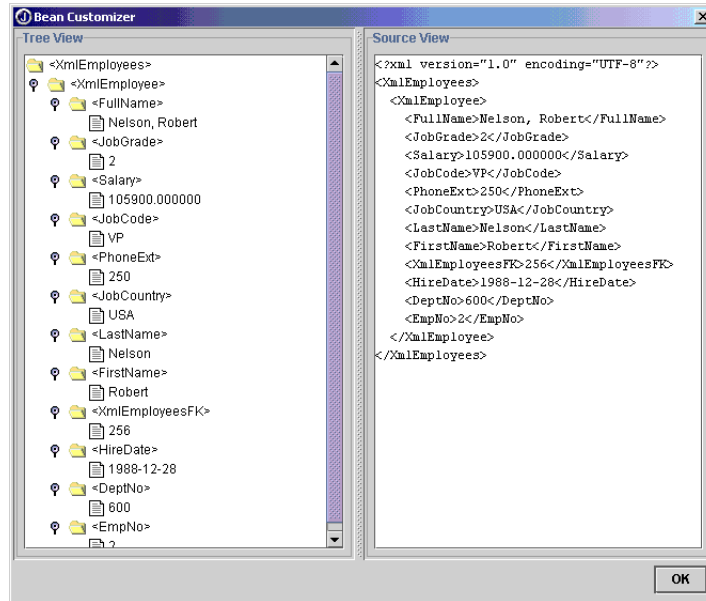
- No parameters of any kind are specified. The result: the query returns an error.
- defaultParams set to `:emp_no = 1` and no runtime parameters specified. The result: the query returns Tom.
- defaultParams set to `:emp_no = 1` and a runtime parameter set to `:emp_no = 2`. The result: the query returns Dick.



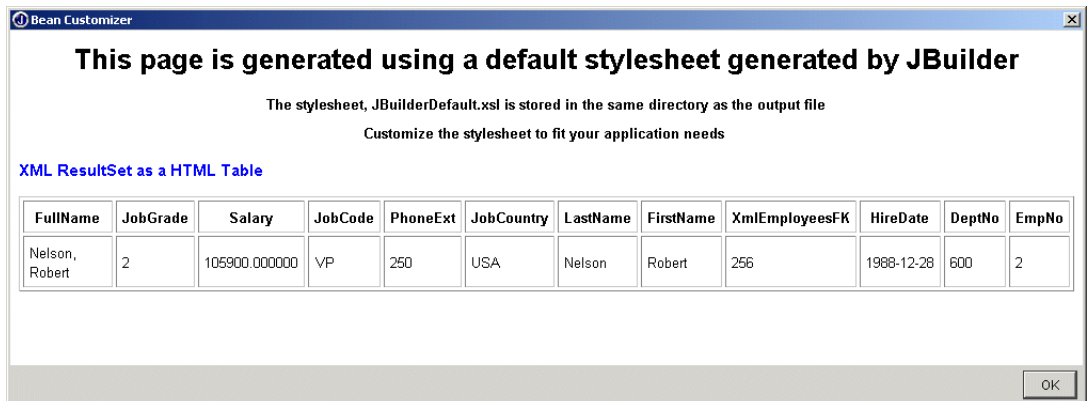
In other words, if no parameter is specified, the default parameter is used. If a parameter is specified at runtime, it is used instead and the default parameter value is ignored. The parameter names are case-sensitive.

### Transferring to XML or HTML

To see what the results of your property settings will be, click the Transfer button. If you chose to create an XML file, you can click the View DOM button to see the Document Object Model (DOM):



If you selected to generate an HTML file, you can click the View HTML file to view the resulting HTML:



## Using the Inspector

You can also set these properties in the designer's Inspector. To open the Inspector,

- Choose the Design tab in the content pane. The Inspector displays to the right of the designer.
- Click the field to the right of a property and enter the appropriate information.

## XML query document

Another way to set the connection and transfer options is through an XML query document. You create an XML query document and specify it as the value of the Query File field in the component's customizer or in the Inspector. Here is a sample query document for a XTable:

```
<Query>
  <Options
    OutputType=XML"
    ColumnFormat="AsElements"
    IgnoreNulls="True"
    DocumentElement="MyDoc"
    RowElement="YourRow">

  <Connection
    Url="jdbc:odbc:foodb"
    Driver="sun.jdbc.odbc.JdbcOdbcDriver"
    User="me"
    Password="ok">

  <Params>
    <Param Name=":Part" Default="ab-c">
    <Param Name=":Number" Default="2">
  </Params>

  <table name="LINES">
    <Key Name="Number">
    <Key Name="Part">
  </table>
</Query>
```

The above query should return the following XML document:

```
<MyDoc>
  <YourRow>
    <col1>some data</col1>
    <col2>some data</col2>
    <Number>2</Number>
    <Part>ab-c</Part>
  </YourRow>
```

```

<YourRow>
  <col1>some other data</col1>
  <col2>some other data</col2>
  <Number>2</Number>
  <Part>ab-c</Part>
</YourRow>
</MyDoc>

```

**Note** If the column format of the query document is “AsAttributes”, then col1, col2, Number and Part would be attributes to YourRow.

Here is a sample query document for a XQuery:

```

<Query>
  <Options
    OutputType="XML"
    ColumnFormat="AsElements"
    IgnoreNulls="True"
    DocumentElement="MyDoc"
    RowElement="YourRow">

  <Connection
    Url="jdbc:odbc:foodb"
    Driver="sun.jdbc.odbc.JdbcOdbcDriver"
    User="me"
    Password="ok">

  <Params>
    <Param Name=":Part" Default="ab-c">
    <Param Name=":Number" Default="2">
  </Params>

  <Sql Value="SELECT * FROM LINES where Number >= :Number AND Number <= :Number"/>
  <!--The above should use CDATA section or escape with lt/gt entity references -->
</Query>

```

For a tutorial about using the XTable and XQuery components, see Chapter 9, “Tutorial: Transferring data with the template-based XML database components.”

## Using the model-based components

---

JBuilder uses XML-DBMS in the model-based components. XML-DBMS, which is middleware for transferring data between XML documents and relational databases, uses object-relational mapping to map objects to the database. XML-DBMS is redistributed with JBuilder and is located in the XML directory on the JBuilder CD. You can find XML-DBMS documentation in the `jbuilder5\extras\xmldbms\docs` directory.

JBuilder provides two beans to actually transfer XML-DBMS data: `XMLDBMSTable` and `XMLDBMSQuery`, which are the third and fourth beans on the component palette above JBuilder’s designer. The `XMLDBMSTable` uses a

specified table and keys to serve as the select criteria for the transfer, while the `XMLDBMSQuery` works on results of an SQL query.

For a tutorial about using the model-based XML components, see Chapter 8, “Tutorial: Transferring data with the model-based XML database components.”

To drop a bean in your application, choose the Design tab and click the XML tab on the component palette. Choose a bean and drop it in the designer.

## XML-DBMS

---

The XML-DBMS solution consists of the following:

- A relational database with a JDBC driver
- An XML document for input/output of data
- An XML map document which defines the mapping between the database and the XML document
- A library with a set of API methods to transfer data between the database and the XML document

At the core of XML-DBMS is the mapping document specified in XML. This is defined by a mapping language and is documented as part of the XML-DBMS distribution. See the XML-DBMS documentation and the sources for more information.

The main elements of the mapping language include:

### **ClassMap**

The `ClassMap` is the root of the mapping. A `ClassMap` maps a database table to XML elements which contain other elements (element content model). In addition, a `ClassMap` nests `PropertyMaps` and `RelatedClassMap`.

### **PropertyMap**

The `PropertyMap` maps PCDATA-only elements and single-value attributes to specific columns in a database table.

### **RelatedClassMap**

`RelatedClassMap` maps interclass relationships. This is done by referring to another `ElementType` (for example, `ClassMap`) declared elsewhere and specifying the basis for the relationship. The map specifies the primary key and foreign key used in the relationship as well as which of the two

tables hold the primary key. Note that the identifier `CandidateKey` is used to represent a primary key.

In addition, key generation is supported. There are scenarios in which the keys are actual business data such as `CustNo` or `EmpNo`. In others, keys must be created just for the purpose of linking. This is possible by using a `generate attribute` as part of the respective key definition.

An optional `orderColumn` with auto key generation, if necessary, is also supported as part of the mapping.

### MiscMaps and Options

In addition to the above mappings, there are a few others to handle nulls, to ignore a root element which does not have any corresponding data in the database but just serves as a grouping element, and date and time formats.

## JBuilder and XML-DBMS

---

JBuilder provides the following XML-DBMS support:

- XML-DBMS wizard
- `XModelBean`: base class for `XMLDBMSTable` and `XMLDBMSQuery`
- `XMLDBMSTable`: transfers data based on a table and key
- `XMLDBMSQuery`: transfers data based on a result set as defined by an SQL query

### XML-DBMS wizard

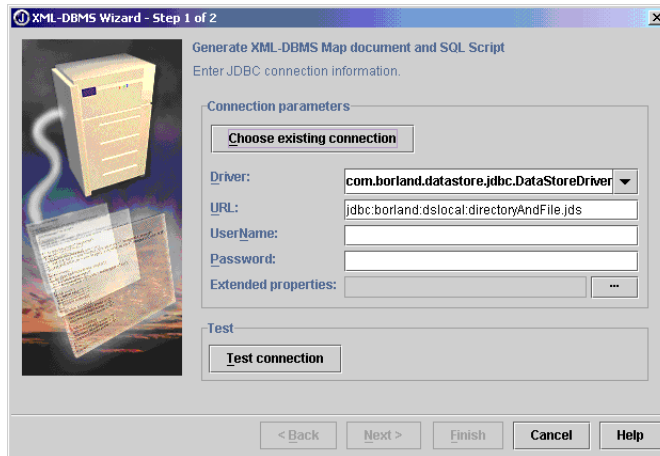
---

JBuilder's XML-DBMS wizard is part of the model/map-based solution that uses the `Map_Factory_DTD` API in XML-DBMS. Given a DTD, the wizard generates a template map document and SQL script file for creating the metadata. In all but the simplest cases, the map document merely serves as a starting point to create the required mapping. The SQL script, a set of `Create Table` statements, also must be modified because XML-DBMS doesn't regenerate the SQL scripts from the modified map document.

Currently, XML-DBMS doesn't support creating a map file from a database schema. If you are starting with an existing database, you must create the map file manually. If you have the XML document, you can open it, right-click it and generate the DTD. Then you can use the generated DTD to generate the map file and edit it to match the database schema.

To use the XML-DBMS wizard,

- 1 Select File | New and click the object gallery's XML tab.
- 2 Double-click the XML-DBMS icon.



The first page lets you specify the JDBC connection to the database that contains the data you want to use to create an XML document. It contains these fields:

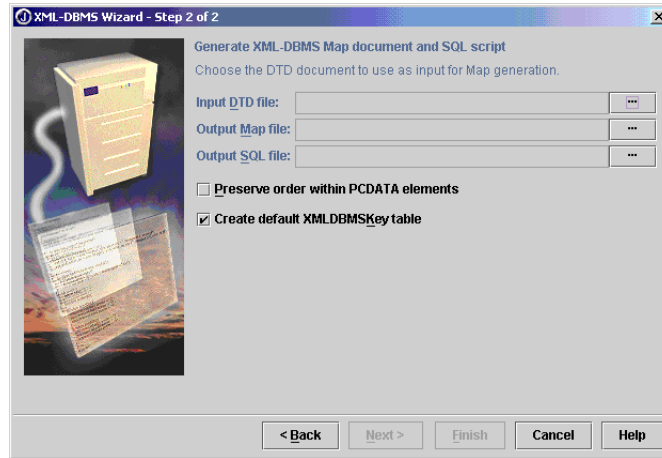
Driver	Specify the JDBC driver to use from the drop-down list. Those drivers displayed in black are drivers you have installed. Drivers shown in red are not available on your system.
URL	Specify the URL to the data source that contains the information you want to use to create an XML document. When you click in the field, it displays the pattern you must use to specify the URL depending on your choice of JDBC driver.
User Name	Enter the user name for the data source, if one is required.
Password	Enter the data source password, if one is required.
Extended Properties	Add any extended properties you need. Click the ... button to display the Extended Properties dialog box you use to add new properties.

If you already have one or more connections defined within JBuilder to data sources, click the Choose Existing Connection button and select

the connection you want. Most of the Connection Properties are then filled in automatically for you.

To test to see if your JDBC connection is correct, click the Test Connection button. The customizer reports whether the connection was successful or failed.

**3** Once you have a successful connection, click Next.



You use this page to specify the DTD you are using to generate the Map and the SQL script to create the database table. Fill in these fields:

DTD File	Specify an existing DTD file.
Output directory	Accept the default name or change it as you like.
MAP File	Specify the name of the Map file you want generated.
SQL Script File	Specify the name of the SQL script file you want generated.

**4** Click OK to close the wizard. The wizard generates the map and SQL files and adds them to your project.

For example, suppose you have a DTD, `request.dtd`:

```
<!ELEMENT request (req_name, parameter*)>
<!ELEMENT parameter (para_name, type, value)>
<!ELEMENT req_name (#PCDATA)>
<!ELEMENT para_name (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT value (#PCDATA)>
```

The XML-DBMS wizard would generate this request.map file:

```
<?xml version='1.0' ?>
<!DOCTYPE XMLToDBMS SYSTEM "xmldbms.dtd" >

<XMLToDBMS Version="1.0">
  <Options>
  </Options>
  <Maps>
    <ClassMap>
      <ElementType Name="request"/>
      <ToRootTable>
        <Table Name="request"/>
        <CandidateKey Generate="Yes">
          <Column Name="requestPK"/>
        </CandidateKey>
      </ToRootTable>
      <PropertyMap>
        <ElementType Name="req_name"/>
        <ToColumn>
          <Column Name="req_name"/>
        </ToColumn>
      </PropertyMap>
      <RelatedClass KeyInParentTable="Candidate">
        <ElementType Name="parameter"/>
        <CandidateKey Generate="Yes">
          <Column Name="requestPK"/>
        </CandidateKey>
        <ForeignKey>
          <Column Name="requestFK"/>
        </ForeignKey>
      </RelatedClass>
    </ClassMap>
    <ClassMap>
      <ElementType Name="parameter"/>
      <ToClassTable>
        <Table Name="parameter"/>
      </ToClassTable>
      <PropertyMap>
        <ElementType Name="para_name"/>
        <ToColumn>
          <Column Name="para_name"/>
        </ToColumn>
      </PropertyMap>
      <PropertyMap>
        <ElementType Name="type"/>
        <ToColumn>
          <Column Name="type"/>
        </ToColumn>
      </PropertyMap>
      <PropertyMap>
        <ElementType Name="value"/>
        <ToColumn>
          <Column Name="value"/>
        </ToColumn>
      </PropertyMap>
    </ClassMap>
  </Maps>
</XMLToDBMS>
```



The XML-DBMS wizard would create the following `request.sql` file:

```
CREATE TABLE "request" ("req_name" VARCHAR(255), "requestPK" INTEGER);
CREATE TABLE "parameter" ("para_name" VARCHAR(255), "type" VARCHAR(255),
                           "requestFK" INTEGER, "value" VARCHAR(255));
CREATE TABLE XMLDBMSKey (HighKey Integer);
INSERT INTO XMLDBMSKey VALUES (0);
```

Once you have a map file and an SQL script file, you can modify them as you wish. For example, while an element name might be “HireDate”, you know the column name is actually “Date\_Hired”. You can make that change by editing the map file directly. Usually the SQL script file is just a starting point for creating the type of table you want, so you often need to edit it also.

When you have your SQL script file to your liking, execute it to create the database tables. A simple way to do this is to copy the SQL statements to the Database Pilot and click the Execute button. For information about using Database Pilot, see “Database Application Developer’s Guide: Database Pilot.” For specific information about executing SQL statements, see “Executing SQL statements” topic in the Database administration tasks chapter in the *Database Application Developer’s Guide*.

## Setting properties for the model-based components

When you have the XML file, the map file, and the database tables, you are ready to use the model beans to transfer data back and forth between the XML file and the table.

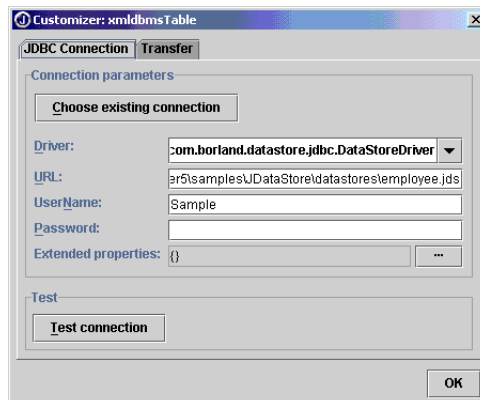
There are two ways to set the properties for a model bean:

- Using the component’s customizer
- Using the Inspector

### Using the component’s customizer

To display a component’s customizer, right-click the component in the structure pane and choose Customizer on the context menu.

This is the customizer for `XMLDBMSTable`:



## JDBC Connection

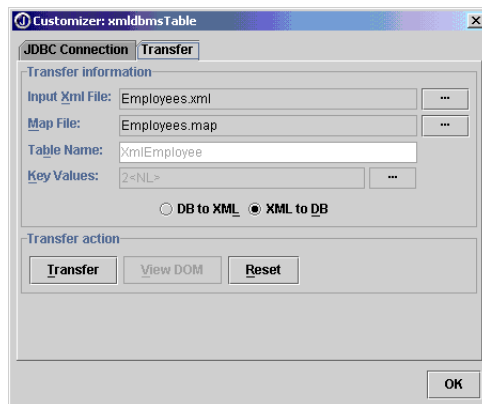
The JDBC Connection page lets you specify the JDBC connection to the database that contains the data you want to use to create an XML document. It contains these fields:

Driver	Specify the JDBC driver to use from the drop-down list. Those drivers displayed in black are drivers you have installed. Drivers shown in red are not available on your system.
URL	Specify the URL to the data source that contains the information you want to use to create an XML document. When you click in the field, it displays the pattern you must use to specify the URL depending on your choice of JDBC driver.
User Name	Enter the user name for the data source, if one is required.
Password	Enter the data source password, if one is required.
Extended Properties	Add any extended properties you need. Click the ... button to display the Extended Properties dialog box you use to add new properties.

If you already have one or more connections defined within JBuilder to data sources, click the Choose Existing Connection button and select the connection you want. Most of the Connection Properties are then filled in automatically for you.

To test to see if your JDBC connection is correct, click the Test Connection button. The customizer reports whether the connection was successful or failed.

Once you have a successful connection, click the Transfer tab.



## Transfer

Use the second page of the wizard to specify whether you are transferring data from an XML document to the database and from the database to an XML document, and to fill in the required information to make the transfer possible.

To transfer data from an XML file to the database file, follow these steps:

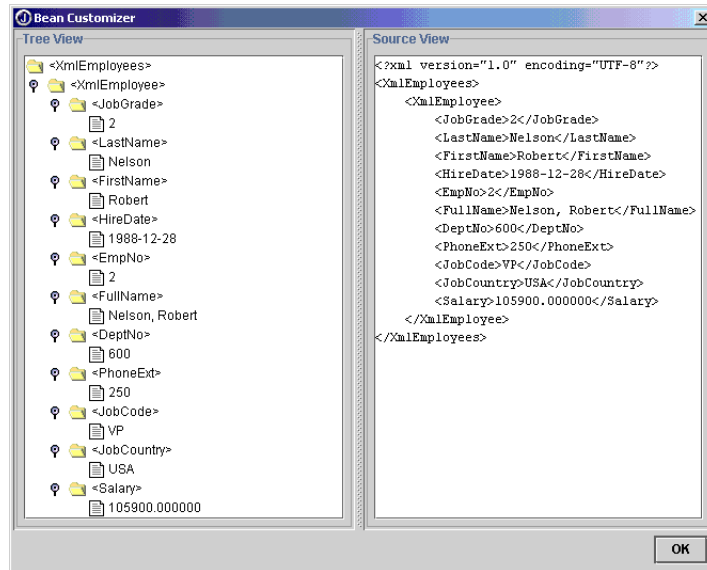
- 1 Edit the XML file so that it contains the values you want transferred to the database table.
- 2 On the Transfer page of the `XMLDBMSTable` customizer, fill in the Input XML File field with the name of the XML file that contains the information you are transferring to the database.
- 3 Specify the map file you created as the value of the MAP File field.  
The remaining two fields are disabled for this type of transfer.
- 4 Click the Transfer button.

To view the results of the Transfer, use Tools | Database Pilot to open the table and view the contents.

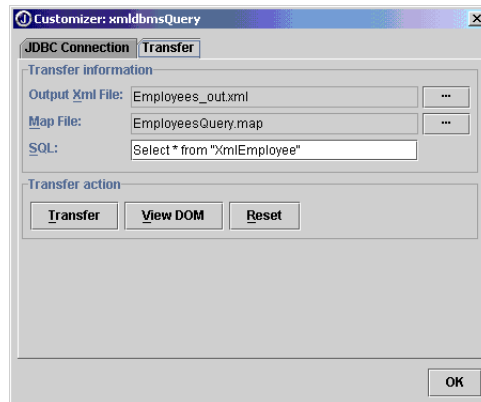
To transfer data from the database to the XML file, follow these steps:

- 1 Click the DB to XML radio button.
- 2 Fill in the Output XML File field with the name of the XML file that will receive the transferred data from the database.
- 3 Specify the map file you created as the value of the MAP File field.
- 4 Specify the name of the table you are transferring from as the value of Table Name.
- 5 Specify the value(s) of the primary key to identify the record(s) to be transferred. For example, if the primary key is EMP\_NO and you want to transfer the data from the row where the EMP\_NO equals 5, specify the Key Value as 5. To determine the key, look in your map file. You'll see it defined as the "CandidateKey" under the <RootTable> node for the given table.
- 6 Choose Transfer.

To view the results of the transfer, choose View DOM to see the structure of the XML file after the transfer.



The Transfer page differs for an XMLDBMSQuery:



The XMLDBMSQuery allows you to specify an SQL query to transfer data from the database to the XML document.

To transfer data from database to the XML file:

- 1 Specify the name of the Output XML File.
- 2 Specify the name of the Map File.
- 3 Enter your SQL statement in the SQL field.
- 4 Choose Transfer.

View the results of the transfer by choosing View DOM.

## Using the Inspector

You can also set the properties of the model-based beans in the designer's Inspector. To open the Inspector,

- 1 Choose the Design tab in the content pane. The Inspector displays to the right of the designer.
- 2 Click the field to the right of a property and enter the appropriate information.

For a tutorial that shows you how to use the `XMLDBMSTable` and `XMLDBMSQuery` components, see Chapter 8, "Tutorial: Transferring data with the model-based XML database components."



# Tutorial: Validating and transforming XML documents

## Overview

---

This tutorial uses features in JBuilder Professional and Enterprise.

This step-by-step tutorial explains how to use JBuilder's XML features for creating an XML document from a DTD, as well as validating and transforming XML documents. Samples are provided in the JBuilder `samples/tutorials/XML/presentation/` directory. A DTD and stylesheets (XSLs) are provided as samples.

This tutorial assumes you have a working knowledge of JBuilder and XML. If you are new to JBuilder, see "The JBuilder environment" chapter in *Building Applications*.

This tutorial contains specific examples that show you how to do the following:

- Create an XML document from an existing DTD.
- Edit the generated XML document with the actual data - employee ID, name, office location, and so on.
- Validate the XML document against the DTD.
- Use JBuilder's IDE to locate an error in the XML document.
- Associate stylesheets with the document.
- Transform the XML document by applying several stylesheets.
- Set transform trace options.
- View the XML document using the XML viewer and JBuilder's default stylesheet tree view.

For more information on JBuilder's XML features, see chapters 2 and 3.

## Step 1: Creating an XML document from a DTD

---

JBuilder provides a wizard for quickly creating an XML document from an existing DTD. Start by creating an XML document from the `Employees.dtd` sample.

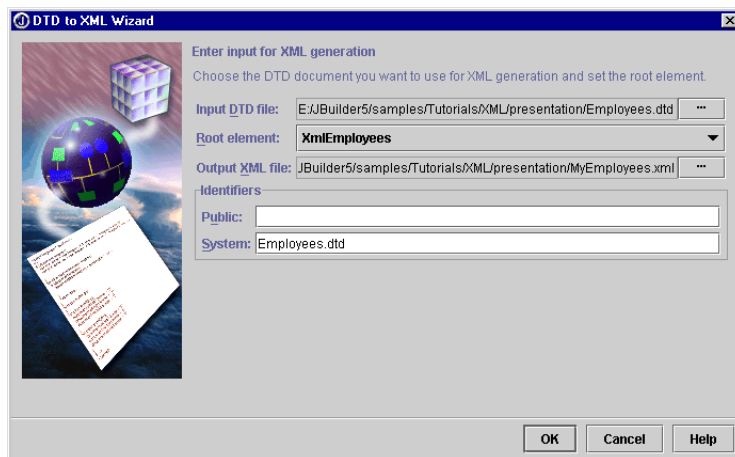
- 1 Open `presentation.jpx` in JBuilder's samples directory: `samples/Tutorials/XML/presentation/`.

**Note** If you have installed JBuilder as root but are running as a regular user, you need to copy the entire samples tree to a directory in which you have full read/write permissions in order to run them.

- 2 Select `Employees.dtd` in the project pane, right-click, and choose **Generate XML** to open the DTD To XML wizard. Selecting the DTD file automatically fills out the Input DTD File field in the wizard. You can also open the wizard from the object gallery (**File | New | XML**).
- 3 Click the drop-down arrow next to the Root Element field to display the list of elements and choose `XmlEmployees`. Be careful **not** to choose `XmlEmployee`.
- 4 Press the ellipsis button next to the Output XML field and rename the default XML file name to `MyEmployees.xml` in the File Name field. Choose **OK** to close the dialog box.
- 5 Enter the name of the DTD file in the System field: `Employees.dtd`. This generates the DOCTYPE declaration, which tells the XML document that a DTD is being used:

```
<!DOCTYPE XmlEmployees SYSTEM "Employees.dtd">
```

The DTD To XML wizard should look like this:



- 6 Click **OK** to close the wizard.
- 7 Save the project.



JBuilder generates an XML document called `MyEmployees.xml` from the DTD. The XML document is open in the editor and is added to the project. JBuilder uses `pcdata` as placeholders between the XML tags, for example, `<EmpNo>pcdata</EmpNo>`. Also, note that `Employees.dtd`, which you entered in the SYSTEM identifier field in the DTD To XML wizard, has been entered in the DOCTYPE declaration.

Next, you need to edit the XML document and replace the `pcdata` placeholders with the actual data.

## Step 2: Editing the generated XML document with the data

---

Now, enter the actual data for each element.

- 1 Create a second employee record by copying the `<XmlEmployee>` `</XmlEmployee>` tags and everything within them. Paste the copy below that record. Select each `pcdata` placeholder and enter the data as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE XmlEmployees SYSTEM "Employees.dtd">
<XmlEmployees>
  <XmlEmployee>
    <EmpNo>2</EmpNo>
    <FirstName>Robert</FirstName>
    <LastName>Nelson</LastName>
    <PhoneExt>250</PhoneExt>
    <HireDate>1988-12-28</HireDate>
    <DeptNo>600</DeptNo>
    <JobCode>VP</JobCode>
    <JobGrade>2</JobGrade>
    <JobCountry>USA</JobCountry>
    <Salary>105900.000000</Salary>
    <FullName>Nelson, Robert</FullName>
  </XmlEmployee>
  <XmlEmployee>
    <EmpNo>4</EmpNo>
    <FirstName>Bruce</FirstName>
    <LastName>Young</LastName>
    <PhoneExt>233</PhoneExt>
    <HireDate>1988-12-28</HireDate>
    <DeptNo>621</DeptNo>
    <JobCode>CEO</JobCode>
    <JobGrade>2</JobGrade>
    <JobCountry>Eng</JobCountry>
    <Salary>97500.000000</Salary>
    <FullName>Young, Bruce</FullName>
  </XmlEmployee>
</XmlEmployees>
```

- 2 Save the project.

## Step 3: Validating the XML document

---

In XML, there are two types of validation: well-formedness and grammatical validity. For a document to be well formed, it must follow the XML rules for the physical document structure and syntax. For example, all XML documents must have a root element. A well-formed document is not checked against an external DTD.

In contrast, a valid XML document is a well-formed document that also conforms to the stricter rules specified in the Document Type Definition (DTD). The DTD describes a document's structure and specifies which element types are allowed, and it defines the properties for each element.

JBuilder performs both types of validation. If a document is not well-formed, errors are displayed in an **Errors** folder in the structure pane. If a document isn't grammatically valid, errors are displayed in the message pane.

This document is well-formed because the **Errors** folder does not appear in the structure pane. Introduce an error by removing the root element of the document. All XML documents must have a root element.

Introduce an error in this well-formed document to see how JBuilder displays errors.

- 1 Select the root element, `<XmlEmployees>`, in the editor and cut it from the document. A well-formed document must have a root element, so this should display as an error. Note that an **Errors** folder displays in the structure pane. Open the **Errors** folder and select the error to highlight it in the code. Double-click the error to change the focus to the line of code in the editor. The line of code indicated by the error message may not be the origin of the error. In this example, the error, `illegal character at end of document`, occurs because the start tag for the root element is missing.
- 2 Re-enter the root element or paste it in the editor. Notice that the **Errors** folder disappears. The document is now well-formed again.

Next, check if your document is grammatically valid compared to the DTD.

- 1 Right-click `MyEmployees.xml` in the project pane and choose **Validate**. A message displays in a **Success** dialog box indicating that the document is valid.
- 2 Introduce a validation error by selecting the **DOCTYPE** declaration and cutting it from the document:

```
<!DOCTYPE XmlEmployees SYSTEM "Employees.dtd">
```

- 3 Validate the document again. The XML Validation Trace page displays in the message pane with an ERROR node.

- 4 Expand the ERROR node in the message pane to display the error:

```
MYEMPLOYEES.XML is invalid
ERROR
    There is no DTD or Schema present in this document
```

- 5 Re-enter or paste the DOCTYPE declaration into the document.

- 6 Introduce another error by changing <FirstName> to <Firstname>.

- 7 Right-click the XML file and choose Validate. Notice the error messages:

```
MYEMPLOYEES.XML is invalid
ERROR
    Element type "Firstname" must be declared.
FATAL_ERROR
    The element type "Firstname" must be terminated by the matching
    end-tag "</Firstname>".
```

There are two errors here: the element `Firstname` is not declared in the DTD and it doesn't have a closing tag.

- 8 Change <Firstname> back to <FirstName>.

- 9 Right-click the XML file and choose Validate. Now your document is valid again.

## Step 4: Associating stylesheets with the document

---

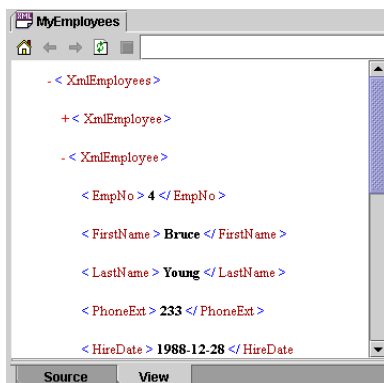
The process of converting an XML document to any other kind of document is called XML transformation. JBuilder's default stylesheet is written in Extensible Style Language Transformations (XSLT) and displays documents in a tree view on the View tab of the content pane. You can also add your own custom stylesheet.

Look at `MyEmployees.xml` with JBuilder's default stylesheet applied and without a stylesheet applied.

- 1 Choose the View tab with `MyEmployees.xml` open in the editor to view the document with the default stylesheet, which is a collapsible tree view. Click the plus (+) and minus (–) symbols to expand and collapse the tree.

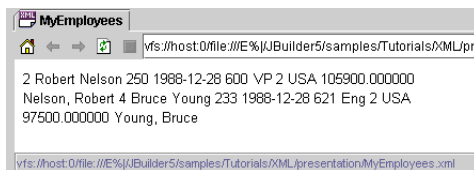
**Note** If the View tab is not available in the content pane, enable JBuilder's XML viewer in the IDE Options dialog box. Choose Tools | IDE Options and set the Enable Browser View option on the XML page of the IDE

Options dialog box. Note that the Apply Default Stylesheet option is set by default. Click OK to close the dialog box.



Now, look at the XML document without a stylesheet:

- 2 Disable the Apply Default Style Sheet option on the XML page of the IDE Options dialog box (Tools | IDE Options).
- 3 Click OK to close the dialog box. Note that the document now displays without any style, in one continuous line.



**Note** You might need to switch to Source and then back to View.

- 4 Enable the stylesheet option again so it is available for later use.

To apply custom stylesheets in JBuilder, you need to associate the stylesheet with the document. Alternately, you could include an XSLT processing instruction in your document and the stylesheets.

Next, associate stylesheets with your document as follows:

- 1 Choose the Transform View tab. Notice that a message indicates that a stylesheet is not associated with the document.



- 2 Click the Add Stylesheets button on the transform view toolbar to open the Configure Node Style Sheets dialog box.
- 3 Choose the Add button and open the `xsls` directory where the stylesheets are located. Select `EmployeesListView.xsl`. Click Add again to add the second stylesheet, `EmployeesTableView.xsl`. The XSL stylesheets are now associated with the document. Click OK to close the dialog box.

**Note** You can also add stylesheets in the Properties dialog box. Right-click the XML document in the project pane and choose Properties.

## Step 5: Transforming the document using stylesheets

Now that the stylesheets are linked to the XML document, you can transform the document using several different stylesheets. The associated stylesheets are now available from the stylesheet drop-down list on the transform view toolbar.

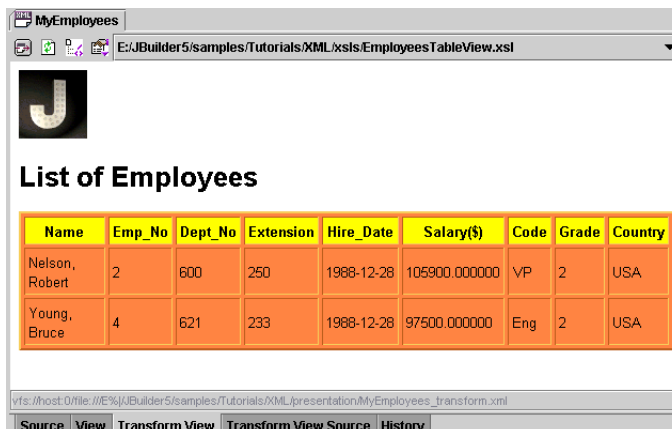
Notice that `MyEmployees.xml` is displayed in the default tree view. By default, transform view uses the default stylesheet to display the document if a stylesheet is not available. Turn this view off, then apply a stylesheet.



- 1 Click the enabled Default Stylesheet button to turn off the tree view.
- 2 Choose `EmployeesListView.xml` from the stylesheet drop-down list. Transform view now displays the transformed document as a list according to the applied stylesheet. The Transform View Source tab displays the source code for the transformed document. Your XML document should look something like this:



- 3 Apply the second stylesheet by choosing `EmployeesTableView.xml` from the drop-down list and look at the resulting transformation to a table in the transform view.



## Step 6: Setting transform trace options

---

You can set Transform Trace options so that when a transformation occurs, you can follow the flow as the stylesheet is applied. These options include Generation, Templates, Elements, and Selections. The traces appear in the message pane. Clicking a trace highlights the corresponding source code. Double-clicking a trace changes the focus to the source code in the editor so you can begin editing.

To set the Trace options,



- 1 Click the Set Trace Options button on the transform view toolbar or choose Tools | IDE Options and click the XML tab.
- 2 Select all the trace options and choose OK.

Now, transform `MyEmployees.xml` by applying `EmployeesListView.xsl` and notice what happens in the message pane.

- 1 Choose `EmployeesListView.xsl` from the stylesheet drop-down list. Note that when the transformation occurs the message pane opens and four nodes appear: generation, templates, elements, selections.
  - generation: outputs information after each result tree generation event, such as start document, start element, characters, and so on.
  - templates: outputs an event when a template is invoked.
  - elements: outputs events that occur as each node is executed in the stylesheet.
  - selections: outputs information after each selection event.
- 2 Expand each node to view the flow of the document's transformation.

# Tutorial: Creating a SAX Handler for parsing XML documents

## Overview

---

This tutorial uses features in JBuilder Enterprise.

This step-by-step tutorial explains how to use JBuilder's SAX Handler wizard to create a SAX parser for parsing your XML documents. Samples are provided in the JBuilder `samples/Tutorials/XML/saxparser/` directory. This tutorial uses a sample XML document that contains employee data, such as employee number, first name, last name, and so on.

SAX, the Simple API for XML, is a standard interface for event-based XML parsing. There are two types of XML APIs: tree-based APIs and event-based APIs. SAX, an event-based API, reports parsing events directly to the application through callbacks. The application implements handlers to deal with the different events, similar to event handling in a graphical user interface.

JBuilder makes it easy to use SAX to manipulate your XML programmatically. The SAX Handler wizard creates a SAX parser implementation template that includes just the methods you want to implement to parse your XML.

This tutorial contains specific examples that show you how to do the following:

- Create a SAX parser with the SAX Handler wizard.
- Edit the SAX parser code to customize the parsing.
- Run the program and view the parsing results.
- Add attributes to the XML document, add code to handle the attributes, and parse the document again.

For the complete SAX parser source code for the tutorial, see “Source code for MySaxParser.java” on page 5-8.

This tutorial assumes you have a working knowledge of JBuilder and XML. If you are new to JBuilder, see “The JBuilder environment” in *Building Applications with JBuilder*.

For more information on JBuilder’s XML features, see chapters 2 and 3.

For information about SAX (Simple API for XML), visit <http://www.megginson.com/SAX/index.html>.

For more information about Xerces, see the Xerces documentation and samples available in the `extras` directory of the JBuilder full install or visit the Apache website at <http://xml.apache.org/>.

## Step 1: Using the SAX Handler wizard

---

JBuilder’s SAX Handler wizard helps you create a SAX parser for custom parsing of your XML documents using the Xerces parsing engine.

To create the SAX parser using the SAX Handler wizard,

**Important**

- 1 Open the project file, `SAXParser.jpx`, located in `samples/Tutorials/XML/saxparser/` in the JBuilder directory.

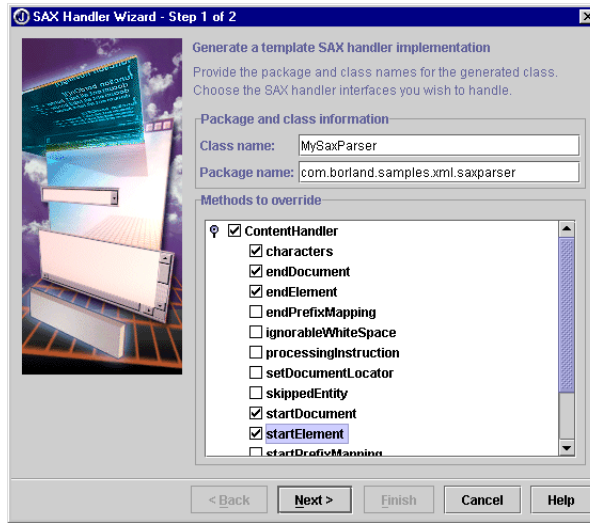
If you have installed JBuilder as root but are running as a regular user, you need to copy the entire samples tree to a directory in which you have full read/write permissions in order to run them.

- 2 Open `Employees.xml` and review the data in the XML document. Notice that there are three employees and that each employee record contains such data as employee number, first name, last name, and full name.



- 3 Choose **File | New** or click the **New** button on the main toolbar to open the object gallery.
- 4 Choose the **XML** tab and double-click the **SAX Handler** icon to open the wizard.
- 5 Make the following changes to the class and package name:
  - **Class Name:** `MySaxParser`
  - **Package Name:** `com.borland.samples.xml.saxparser`
- 6 Check **ContentHandler** as a **interface to override** and expand the **ContentHandler** node. Check these five options to create methods for them: **characters**, **endDocument**, **endElement**, **startDocument**, and **startElement**. Step 1 should look like this:





7 Choose Next to go to Step 2 which lists parser options. For this tutorial you won't select any of these parser options.

8 Choose Finish to close the wizard.

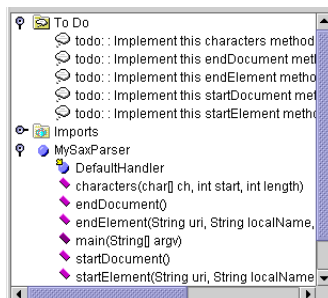
The wizard generates a parser file called `MySaxParser.java`, adds it to the project, and opens it in the editor. Take a moment to look at this file and notice that the wizard generated empty methods that you need to complete.

**Tip** To browse any of the imported classes in this file, open the `Imports` folder in the structure pane and double-click a package to open the `Browse Import Symbol` dialog box and browse to the class you want to look at.

## Step 2: Editing the SAX parser

The wizard generates empty methods that you need to implement. Notice that the structure of `MySaxParser.java` is visible in the structure pane to the left of the editor and that the `To Do` folder contains five methods that need to be implemented:

`characters()`, `endDocument()`, `endElement()`, `startDocument()`, and `startElement()`.



Look at the `main()` method's `try` block generated by the wizard:

```
try {
    XMLReader parser = XMLReaderFactory.createXMLReader(
        "org.apache.xerces.parsers.SAXParser");
    MySaxParser MySaxParserInstance = new MySaxParser();
    parser.setContentHandler(MySaxParserInstance);
    parser.parse(uri);
}
```

This block instantiates a parser, then tells it to parse the XML file specified in the Application Parameters field on the Run page of Project Properties (Project | Project Properties).

Start by adding print statements to the `startDocument()` and `endDocument()` methods that print beginning and ending parsing messages to the screen.

**Tip** Double-click a method in the structure pane or in the To Do folder to move the cursor to that method in the editor.

**1** Add a print statement to the `startDocument()` method:

```
System.out.println("PARSING begins...");
```

**2** Add a print statement to the `endDocument()` method:

```
System.out.println("...PARSING ends");
```

**3** Remove the `throw` statements in each method as these won't be needed.

**4** Create a variable for indenting the parsed output and declare it just before the `characters()` method:

```
private int idx = 0; //indent
public void characters(char[] ch, int start, int length) throws SAXException {
```

**5** Create a constant `INDENT` with a value of 4 just before the `main()` method.

```
private static int INDENT = 4;
```

```
public static void main(String[] argv) {
```

**6** Create a `getIndent()` method at the end of the `MySaxParser` class after the `startElement()` method. This method provides indentation for the parsing output to make it easier to read.

```
private String getIndent() {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < idx; i++)
        sb.append(" ");
    return sb.toString();
}
```

## 7 Add the following code indicated in bold to each of the methods to add indenting to the output:

```

public void characters(char[] ch, int start, int length) throws SAXException {
    //instantiates s, indents output, prints character values in element
    String s = new String(ch, start, length);
    if (ch[0] == '\n')
        return;
    System.out.println(getIndent()+ " Value: " + s);
}

public void endDocument() throws SAXException {
    idx -= INDENT;
    System.out.println(getIndent() + "end document");
    System.out.println("...PARSING ends");
}

public void endElement(String uri, String localName,
    String qName) throws SAXException {
    System.out.println(getIndent() + "end element");
    idx -= INDENT;
}

public void startDocument() throws SAXException {
    idx += INDENT;
    System.out.println("PARSING begins...");
    System.out.println(getIndent() + "start document: ");
}

public void startElement(String uri, String localName, String qName,
    Attributes attributes) throws SAXException {
    idx += INDENT;
    System.out.println('\n' + getIndent() + "start element: " + localName);
}

```

**Tip** You can use Browse Symbol in the editor to browse classes, interfaces, events, methods, properties, packages, and identifiers to learn more about them. Position the cursor in one of these names, right-click, and choose Browse Symbol. In order for a class to be found automatically, it must be on the import path. Results are displayed in the content pane of the AppBrowser. You can also browse classes in the editor from the Search menu (Search | Browse Classes).

## 8 Save the project.

# Step 3: Running the program

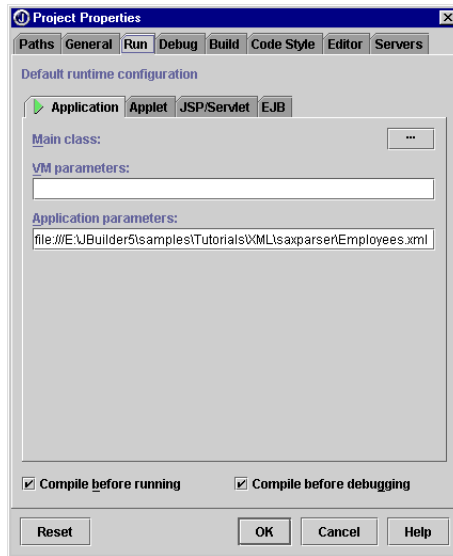
---

Before running the program, you need to specify the XML document as a runtime parameter so the parser application knows what file to parse.

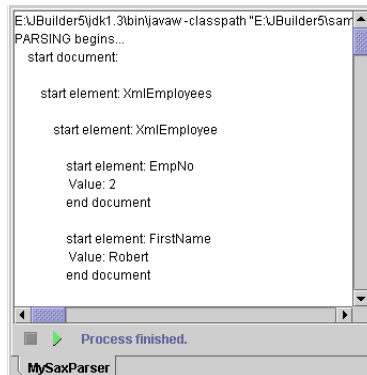
- 1 Choose Project | Project Properties to open the Project Properties dialog box.
- 2 Choose the Application tab on the Run page.

- 3 Input the path to the XML document in the Application Parameter field. For example,

file:///E:/JBuilder5\samples\Tutorials\XML\saxparser\Employees.xml



- 4 Choose OK to close the Run page.
- 5 Right-click `MySaxParser.java` in the project pane and choose Run. The message pane opens and displays the parsing output:



## Step 4: Adding attributes

Next, add attributes to the XML document. Then you need to add code to the parser so it can handle the attributes.

- 1 Open `Employees.xml` and add an attribute to the first `EmpNo` element:

```
<EmpNo att1="a" att2="b">2</EmpNo>
```

**2 Add attributes to the first FirstName element:**

```
<FirstName z="z1" d="d1" k="k1">Robert</FirstName>
```

**3 Add the attList variable just above the main() method:**

```
public class MySaxParser extends DefaultHandler {

    private static int INDENT = 4;
    private static String attList = "" ;
    public static void main(String[] argv) {
```

**4 Add the following code to the startElement() method to handle the attribute:**

```
public void startElement(String uri, String localName, String qName, Attributes
    attributes) throws SAXException {
    idx += INDENT;
    System.out.println('\n'+getIndent() + "start element: " + localName);
    if (attributes.getLength() > 0) {
        idx += INDENT;
        for (int i = 0; i < attributes.getLength(); i++){
            attList = attList + attributes.getLocalName(i) + " = " +
                attributes.getValue(i);
            if (i < (attributes.getLength() - 1))
                attList = attList + ",";
        }
        idx -= INDENT;
    }
}
```

**5 Add the following code to the endElement() method:**

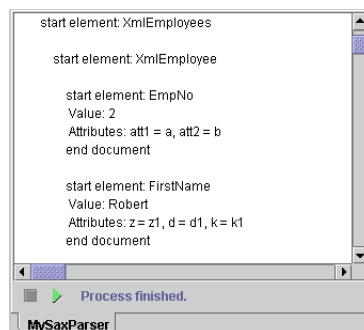
```
public void endElement(String uri, String localName,
    String qName) throws SAXException {
    if (!attList.equals(""))
        System.out.println(getIndent() + " Attributes: " + attList);

    attList = "";

    System.out.println(getIndent() + "end element");
    idx -= INDENT;
}
```

**6 Save the project and run the program again.**

Notice the parsing output now includes the attributes.



## Source code for MySaxParser.java

---

```
package com.borland.samples.xml.saxparser;

import java.io.IOException;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;

public class MySaxParser extends DefaultHandler {

    private static int INDENT = 4;
    private static String attList = "";
    public static void main(String[] argv) {
        if (argv.length != 1) {
            System.out.println("Usage: java MySaxParser [URI]");
            System.exit(0);
        }
        String uri = argv[0];
        try {
            XMLReader parser = XMLReaderFactory.createXMLReader("
                org.apache.xerces.parsers.SAXParser");
            MySaxParser MySaxParserInstance = new MySaxParser();
            parser.setContentHandler(MySaxParserInstance);
            parser.parse(uri);
        }
        catch(IOException ioe) {
            ioe.printStackTrace();
        }
        catch(SAXException saxe) {
            saxe.printStackTrace();
        }
    }

    private int idx = 0;

    public void characters(char[] ch, int start, int length) throws SAXException {
        /**@todo: Implement this characters method*/
        String s = new String(ch, start, length);
        if (ch[0] == '\n')
            return;
        System.out.println(getIndent() + " Value: " + s);
    }

    public void endDocument() throws SAXException {
        /**@todo: Implement this endDocument method*/
        idx -= INDENT;
        System.out.println(getIndent() + "end document");
        System.out.println("...PARSING ends");
    }
}
```

```

public void endElement(String uri, String localName,
                      String qName) throws SAXException {
    /**@todo: Implement this endElement method*/
    if (!attList.equals(""))
        System.out.println(getIndent() + " Attributes: " + attList);
    attList = "";
    System.out.println(getIndent() + "end document");
    idx -= INDENT;
}

public void startDocument() throws SAXException {
    /**@todo: Implement this startDocument method*/
    idx += INDENT;
    System.out.println("PARSING begins...");
    System.out.println(getIndent() + "start document: ");
}

public void startElement(String uri, String localName, String qName,
                        Attributes attributes) throws SAXException {
    /**@todo: Implement this startElement method*/
    idx += INDENT;
    System.out.println('\n' + getIndent() + "start element: " + localName);
    if (attributes.getLength() > 0) {
        idx += INDENT;
        for (int i = 0; i < attributes.getLength(); i++) {
            attList = attList + attributes.getLocalName(i) + " = " +
                attributes.getValue(i);
            if (i < (attributes.getLength() - 1))
                attList = attList + ", ";
        }
        idx -= INDENT;
    }
}

private String getIndent() {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < idx; i++)
        sb.append(" ");
    return sb.toString();
}
}

```





# Tutorial: DTD databinding with BorlandXML

## Overview

---

This tutorial uses features in JBuilder Enterprise.

This step-by-step tutorial explains how to use JBuilder's XML databinding features using DTDs and BorlandXML. Samples are provided in the JBuilder samples: `samples/Tutorials/XML/databinding/fromDTD/`. This tutorial uses an employee database as a sample with such fields as employee number, first name, last name, and so on. An XML document and DTD are provided as samples, as well as a test application to manipulate the data.

This tutorial assumes you have a working knowledge of JBuilder and XML. If you are new to JBuilder, see "The JBuilder environment" in *Building Applications with JBuilder*.

Databinding is a means of accessing data and manipulating it, then sending the revised data back to the database or displaying it with an XML document. The XML document can be used as the transfer mechanism between the database and the application. This transfer is done by binding a Java object to an XML document. The databinding is implemented by generating Java classes to represent the constraints contained in a grammar, such as in a DTD or an XML schema. You can then use these classes to create XML documents that comply to the grammar, read XML documents that comply to the grammar, and validate XML documents against the grammar as changes are made to them.

This tutorial contains specific examples that show you how to do the following:

- Generate Java classes from a DTD using BorlandXML.
- Unmarshal the data from XML objects and convert it to Java objects.

- Edit the data by adding an employee and modifying an existing employee's name.
- Marshall the Java objects back to the XML document.

For more information on JBuilder's XML features, see chapters 2 and 3.

## Step 1: Generating Java classes from a DTD

---

The first step in working with your data is generating Java classes from your existing DTD with the Databinding wizard. When BorlandXML is selected as the databinding type, the Databinding wizard examines the DTD and creates a Java class for each element in the DTD.

To generate Java classes from a DTD using the Databinding wizard,

- 1 Open the project file, `BorlandXML.jpx`, located in `samples/Tutorials/XML/databinding/fromDTD/` in the JBuilder directory.

**Important**

If you have installed JBuilder as root but are running as a regular user, you need to copy the entire samples tree to a directory in which you have full read/write permissions in order to run them.

- 2 Open `Employees.xml` and review the data in the XML document. Notice that there are three employees: Robert Nelson, Bruce Young, and Kim Lambert. Each employee record contains such data as employee number, first name, last name, and full name. This is the data you will be manipulating.

**Note**

You can also view the XML document in the XML viewer. Enable the browser view on the XML page of the IDE Options dialog box (Tools | IDE Options). Then, choose the View tab in the content pane to view the document in the default tree view.

- 3 Open `Employees.dtd` and notice the elements in the XML document: `XmlEmployee`, `EmpNo`, `FirstName`, and so on. The Databinding wizard will generate a Java class for each of these elements.
- 4 Right-click `Employees.dtd` and choose Generate Java to open the Databinding wizard. Notice that BorlandXML is selected as the Databinding Type. BorlandXML generates Java classes from DTDs.

**Note**

The Databinding wizard is also available on the XML page of the object gallery (File | New).

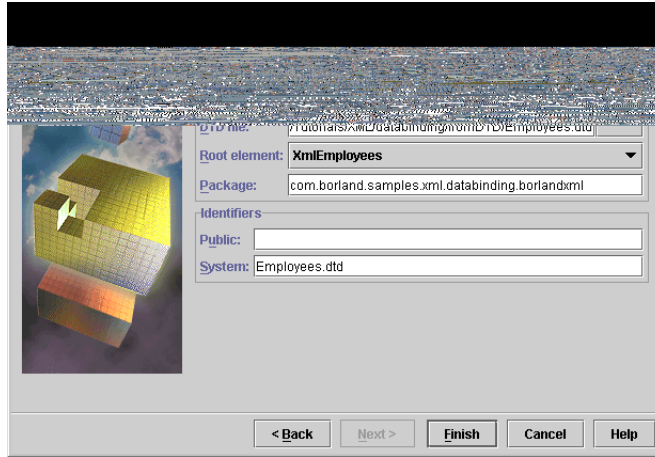
- 5 Click Next to continue to Step 2.

- 6 Fill in the following fields in Step 2 of the wizard:

- **DTD File:** `<jbuilder>/samples/Tutorials/XML/databinding/fromDTD/Employees.dtd`. This field is filled out automatically, because you selected the DTD file in the project pane before opening the wizard.

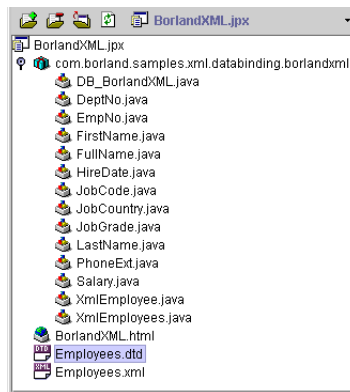
- **Root Element:** Select `XmlEmployees` from the drop-down list. Be sure to select `XmlEmployees`, not the singular element, `XmlEmployee`.
- **Package Name:** Change the package name to `com.borland.samples.xml.databinding.borlandxml`
- **System Identifier:** `Employees.dtd`.

The Databinding wizard looks like this:



7 Click Finish.

- 8 Expand the automatic source package node, `com.borland.samples.xml.databinding.borlandxml`, in the project pane to see the `.java` files generated by the wizard. Notice that each element in the DTD has its own class. The package node also includes the test application, `DB_BorlandXML.java`, which has been supplied as part of the sample. You'll be using the test application to manipulate the data.



9 Save the project.

Before continuing, take a moment to examine some of the generated classes.

- 1 Open `EmpNo.java` and examine the code. Notice that there's a constructor for creating an `EmpNo` object from the `EmpNo` element, as well as methods for unmarshalling the `EmpNo` element to the `EmpNo` object and getting the tag name for the element.
- 2 Open `XmlEmployee.java`. The `XmlEmployee` element in the XML document contains all of the records for the individual, such as `EmpNo`, `FirstName`, and `LastName`. In this class, there's a constructor for creating an `XmlEmployee` object from the `XmlEmployee` element, declarations that define the elements, and methods that set and get the elements contained by `XmlEmployee`. In addition, `unmarshal()` (read) and `marshal()` (write) methods read the XML object into the Java object and then write the Java object back to the XML object after manipulating the object in the Java application.
- 3 Open `XmlEmployees.java`. `XmlEmployees` is the root element of the XML document which contains all the other XML elements. The `XmlEmployees` class has methods to get and set the `XmlEmployee` element, as well as methods that add and remove employees, set and get PUBLIC and SYSTEM IDs, and marshal and unmarshal the data.

**Tip** You can use Browse Symbol in the editor to browse classes, interfaces, events, methods, properties, packages, and identifiers. Position the cursor in one of these names, right-click, and choose Browse Symbol. In order for a class to be found automatically, it must be on the import path. Results are displayed in the content pane of the AppBrowser. You can also browse classes in the editor from the Search menu (Search | Browse Classes).

## Step 2: Unmarshalling the data

---

Now that you have created your Java objects for the XML objects, take a look at the test application, `DB_BorlandXML.java`. This application passes the data between the XML document and the Java objects - unmarshalling (read) and marshalling (write) the data.

- 1 Open `DB_BorlandXML.java` by double-clicking it in the project pane. Notice that there is a class variable, `db_BorlandXML`, in the `main()` method of the application calling different methods, three of which have been commented out.

```
public class DB_BorlandXML {  
    public DB_BorlandXML() {  
    }  
    public static void main(String[] args) {  
        db_BorlandXML = new DB_BorlandXML();  
        db_BorlandXML.readEmployees();  
    }  
}
```

```
// db_BorlandXML.addEmployee();
// db_BorlandXML.modifyEmployee();
// db_BorlandXML.readEmployees();
}
....
}
```

Next, run the application without modifying any of the code, and read the employees from the XML document, converting them to Java objects. Then manipulate the data by changing the code and adding and modifying employees. The first step is to read the employees from the XML document.

- 2 Run the application by right-clicking `DB_BorlandXML.java` and choosing Run. The application runs, reads the employee information, and prints the number of employees and the first and last employee full names to the message pane:

```
== unmarshalling "Employees.xml" ==
Total Number of Employees read = 3
First Employee's Full Name is Nelson, Robert
Last Employee's Full Name is Lambert, Kim
```

## Step 3: Adding an employee

---

Next, add an employee and marshal the data back to the XML document. To do this you need to remove the comments (`//`) before the line that calls the `addEmployee()` method: `db_BorlandXML.addEmployee()`; You'll also add another `readEmployees()` method call to read the new data back to the message pane. Then when you run the program, Charlie Chaplin will be added as a new employee using the `addEmployee()` method.

- 1 Remove the comments from `db_BorlandXML.addEmployee`.
- 2 Add a `readEmployees()` call just below the line you just uncommented. Your code should look like this:

```
public static void main(String[] args) {

    db_BorlandXML = new DB_BorlandXML();
    db_BorlandXML.readEmployees();

    db_BorlandXML.addEmployee();
    db_BorlandXML.readEmployees();
// db_BorlandXML.modifyEmployee();
// db_BorlandXML.readEmployees();
}
```

- 3 Run the program again. Note the printout in the message pane and that Charlie has been added as a fourth employee.

```
== unmarshalling "Employees.xml" ==  
Total Number of Employees read = 3  
First Employee's Full Name is Nelson, Robert  
Last Employee's Full Name is Lambert, Kim  
== unmarshalling "Employees.xml" ==  
Total Number of Employees read = 4  
First Employee's Full Name is Nelson, Robert  
Last Employee's Full Name is Chaplin, Charlie
```

- 4 Switch to `Employees.xml` and notice that Charlie Chaplin has been added as the fourth employee.

## Step 4: Modifying an employee

---

Now, modify Charlie Chaplin's name. To do this you need to add comments to the `addEmployee()` and `readEmployees()` lines once again and uncomment the `modifyEmployee()` and `readEmployees()` lines.

- 1 Comment out these two lines in `DB_BorlandXML.java`:

```
// db_BorlandXML.addEmployee();  
// db_BorlandXML.readEmployees();
```

- 2 Remove the comments from these two lines:

```
db_BorlandXML.modifyEmployee();  
db_BorlandXML.readEmployees();
```

Your code should look like this:

```
public static void main(String[] args) {  
  
    db_BorlandXML = new DB_BorlandXML();  
    db_BorlandXML.readEmployees();  
  
    // db_BorlandXML.addEmployee();  
    // db_BorlandXML.readEmployees();  
    db_BorlandXML.modifyEmployee();  
    db_BorlandXML.readEmployees();  
}
```

Now that you've uncommented the `modifyEmployee()` line, when you run the program again, Charlie Chaplin's name will be replaced with the information in the `modifyEmployee()` method.

- 1 Right-click `DB_BorlandXML.java` in the project pane and choose Run to run the application. Note the printout in the message pane and that Charlie has been changed to Andy Scott.

```
== unmarshalling "Employees.xml" ==  
Total Number of Employees read = 4  
First Employee's Full Name is Nelson, Robert
```

```

Last Employee's Full Name is Chaplin, Charlie
== unmarshalling "Employees.xml" ==
Total Number of Employees read = 4
First Employee's Full Name is Nelson, Robert
Last Employee's Full Name is Scott, Andy

```

- 2 Return to `Employees.xml` and note that the data for Andy Scott has been marshalled from the Java objects to the XML objects and written back to the XML document. So now Andy Scott replaces Charlie Chaplin.

## Step 5: Running the completed application

---

Now that you understand what this application is doing, remove the new data from the XML document and remove the comments in the program, add a print statement to read the new employee, and run the program.

- 1 Remove Andy Scott's employee data from the XML document, being careful not to remove any other data or XML tags.

```

<XmlEmployee>
  <EmpNo>9000</EmpNo>
  <FirstName>Andy</FirstName>
  <LastName>Scott</LastName>
  <PhoneExt>1993</PhoneExt>
  <HireDate>2/2/2001</HireDate>
  <DeptNo>600</DeptNo>
  <JobCode>VP</JobCode>
  <JobGrade>3</JobGrade>
  <JobCountry>USA</JobCountry>
  <Salary>145000.00</Salary>
  <FullName>Scott, Andy</FullName>
</XmlEmployee>

```

Now the XML document only contains the original three employee records.

- 2 Return to `DB_BorlandXML.java` and remove the comments from all of the class variable calls to methods. Your code should look like this:

```

public static void main(String[] args) {

    db_BorlandXML = new DB_BorlandXML();
    db_BorlandXML.readEmployees();
    db_BorlandXML.addEmployee();
    db_BorlandXML.readEmployees();
    db_BorlandXML.modifyEmployee();
    db_BorlandXML.readEmployees();
}

```

- 3 Save the project.

- 4 Run the program to output the data and see the following printout in the message pane:

```
== unmarshalling "Employees.xml" ==  
Total Number of Employees read = 3  
First Employee's Full Name is Nelson, Robert  
Last Employee's Full Name is Lambert, Kim  
== unmarshalling "Employees.xml" ==  
Total Number of Employees read = 4  
First Employee's Full Name is Nelson, Robert  
Last Employee's Full Name is Chaplin, Charlie  
== unmarshalling "Employees.xml" ==  
Total Number of Employees read = 4  
First Employee's Full Name is Nelson, Robert  
Last Employee's Full Name is Scott, Andy
```

- 5 Switch to `Employees.xml` to verify that the data has been marshalled back to the XML document. Notice that Andy Scott's information was added and then modified. When the employees are read a second time, the new employee is counted.

Congratulations. You've completed the tutorial. You've read, added, and modified employee data in an XML document using a Java application and Java classes generated by the Databinding wizard from a DTD.



# Tutorial: Schema databinding with Castor

## Overview

---

This tutorial uses features in JBuilder Enterprise.

This step-by-step tutorial explains how to use JBuilder's XML databinding features using schema and Castor. Samples are provided in the JBuilder samples directory: `samples/Tutorials/XML/databinding/fromSchema`. This tutorial uses an employee database as a sample with such fields as employee number, first name, last name, etc. An XML document and schema file are provided as samples, as well as a test application to manipulate the data.

This tutorial assumes you have a working knowledge of JBuilder and XML. If you are new to JBuilder, see "The JBuilder environment" in *Building Applications with JBuilder*.

Databinding is a means of accessing data and manipulating it, then sending the revised data back to the database or displaying it with an XML document. The XML document can be used as the transfer mechanism between the database and the application. This transfer is done by binding a Java object to an XML document. The databinding is implemented by generating Java classes to represent the constraints contained in a grammar, such as in a DTD or an XML schema. You can then use these classes to create XML documents that comply to the grammar, read XML documents that comply to the grammar, and validate XML documents against the grammar as changes are made to them.

This tutorial contains specific examples that show you how to do the following:

- Generate Java classes from a schema using Castor and the Databinding wizard.
- Unmarshal the data from XML objects and convert it to Java objects.
- Edit the data by adding an employee and modifying an existing employee.
- Marshall the Java objects back to the XML document.

For more information on JBuilder's XML features, see chapters 2 and 3.

For more information on Castor, see the Castor documentation in the `extras` directory of the JBuilder full install or the Castor website at [www.castor.org](http://www.castor.org).

## Step 1: Generating Java classes from a schema

---

The first step in working with your data is generating Java classes from your existing schema. When Castor is selected as the databinding type, the Databinding wizard examines the selected schema file and creates Java classes based on that schema.

To generate Java classes from a schema using the Databinding wizard,

- 1 Open the project file, `castor.jpx`, located in `samples/Tutorials/XML/databinding/fromSchema` in the JBuilder directory.

**Important**

If you have installed JBuilder as root but are running as a regular user, you need to copy the entire samples tree to a directory in which you have full read/write permissions in order to run them.

- 2 Open `Employees.xml` and review the data in the XML document. Notice that there are three employees: Robert Nelson, Bruce Young, and Kim Lambert. Each employee record contains such data as employee number, first name, last name, and full name. This is the data you will be manipulating.

**Note**

You can also view the XML document in the XML viewer. Enable the browser view on the XML page of the IDE Options dialog box (Tools | IDE Options). Then, choose the View tab in the content pane to view the document in the default tree view.

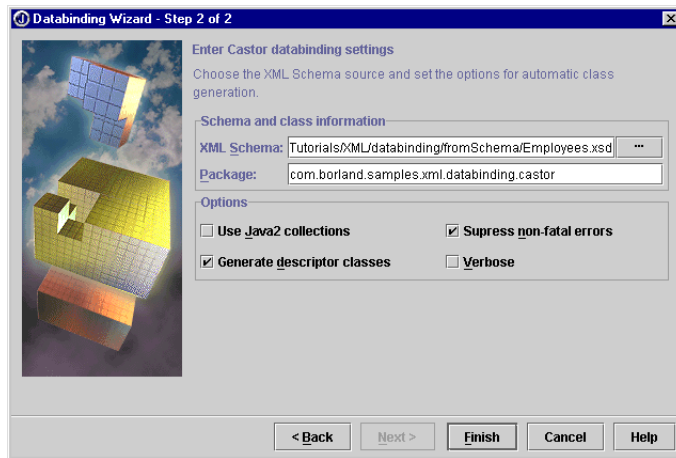
- 3 Open `Employees.xsd` and review the file. The Databinding wizard will generate Java classes according to the schema (XSD) file.
- 4 Right-click `Employees.xsd` in the project pane and choose Generate Java to open the Databinding wizard. By doing this, the XSD field of the wizard is filled out automatically with the file name when you open the wizard.
- 5 Accept Castor as the Databinding Type and click Next to go to Step 2. Castor uses schema (XSD) to create Java classes. Schemas, more robust and flexible than DTDs, have several advantages over DTDs. Schemas

are XML documents, unlike DTDs which contain non-XML syntax. Schemas also support namespaces, which are required to avoid naming conflicts, and offer more extensive data type and inheritance support.

**6** Fill out the fields as follows:

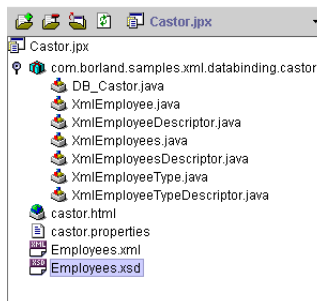
- **XML Schema:** Accept the default path— `/<jbuilder>/samples/Tutorials/XML/databinding/fromSchema/Employees.xsd`. This field is already filled in for you.
- **Package:** Change the package name to `com.borland.samples.xml.databinding.castor`
- Accept the default options.

Step 2 of the wizard should look like this:



**7** Click Finish.

- 8** Expand the `com.borland.samples.xml.databinding.castor` automatic source package node in the project pane to see the `.java` files generated by the wizard. The package node also includes the test application, `DB_Castor.java`, which has been supplied as part of the sample. You'll be using the test application to manipulate the data.



**9** Save the project.

Before continuing to the next step, examine some of the Java classes generated by the wizard. Castor executes databinding by mapping an instance of an XML schema into an appropriate object model. This object model includes a set of classes and types which represent the data. Descriptors are used to obtain information about a class and its fields. In most cases, the marshalling framework uses a set of `ClassDescriptors` and `FieldDescriptors` to describe how an `Object` should be marshalled and unmarshalled from the XML document.

First, notice that there are several types of Java files:

- 1 `XmlEmployee.java`: unmarshals and marshals the data between the XML document and the Java objects.
- 2 `XmlEmployeeDescriptor.java`: gets namespaces, identity, Java class, and so on.
- 3 `XmlEmployees.java`: gets and sets the `XmlEmployee` element and unmarshals and marshals the data between the XML document and the Java objects.
- 4 `XmlEmployeesDescriptor.java`: gets namespaces, identity, Java class, and so on.
- 5 `XmlEmployeeType.java`: defines the datatype of each object.
- 6 `XmlEmployeeTypeDescriptor.java`: initializes the element descriptors and has various methods to get the XML file name, Java class, and namespace.

**Tip** You can use Browse Symbol in the editor to browse classes, interfaces, events, methods, properties, packages, and identifiers. Position the cursor in one of these names, right-click, and choose Browse Symbol. In order for a class to be found automatically, it must be on the import path. Results are displayed in the content pane of the AppBrowser. You can also browse classes in the editor from the Search menu (Search | Browse Classes).

**See also** Castor API in the `extras` directory of the JBuilder full install or the Castor website at <http://www.castor.org/javadoc/overview-summary.html>

## Step 2: Unmarshalling the data

---

Now that you have created your Java objects for the XML objects, take a look at the test application, `DB_Castor.java`. This application will pass the data between the XML document and the Java objects using the marshalling framework - `unmarshal` (read) and `marshal` (write) the data.

Open `DB_Castor.java` by double-clicking it in the project pane. Notice that there are several methods that manipulate the data. First, the application unmarshals or reads the data from the XML document and prints employee count and first and last employee names to the screen. Next, an employee is added, then modified. And lastly, the data is marshalled back to the XML document.

```
try {
    String fileName = "Employees.xml";
    System.out.println("== unmarshalling \"" + fileName + "\" ==");
    //UnMarshal XML file
    XmlEmployees xmlEmployees = XmlEmployees.unmarshal(new FileReader(fileName));
    System.out.println("Total Number of XmlEmployees read = " +
        xmlEmployees.getXmlEmployeeCount());
    System.out.println("First XmlEmployee's Full Name is "+
        xmlEmployees.getXmlEmployee(0).getFullName());
    System.out.println("Last XmlEmployee's Full Name is "+xmlEmployees.getXmlEmployee
        (xmlEmployees.getXmlEmployeeCount()-1).getFullName());

    // Add an XmlEmployee
    xmlEmployees.addXmlEmployee(getXmlEmployee("8000","400",
        "Charlie","Castor","3/3/2001","VP","USA","2","1993","155000.00"));
    //Modify the last XmlEmployee
    xmlEmployees.setXmlEmployee(getXmlEmployee("8000","600","Peter","Castor",
        "3/3/2001","VP","USA","3","2096","125000.00"),
        xmlEmployees.getXmlEmployeeCount()-1);
    // Marshal out the data to the same XML file
    xmlEmployees.marshall(new java.io.PrintWriter(fileName));
}
```

**Tip** To browse any of the imported classes in this file, open the **Imports** folder in the structure pane and double-click a package to open the **Browse Import Symbol** dialog box and browse to the class you want to look at.

Although you could run the application and add and modify an employee consecutively, let's break this down into steps so you can see what is happening.

## Step 3: Adding an employee

---

First, modify the application so it only adds an employee. To do this, you need to comment out the method that modifies an employee, `setXmlEmployee()` method, in `DB_Castor.java`. You'll modify the employee in the next step.

- 1 Comment out the `setXmlEmployee()` method and the print statement below it. With these lines commented out, the application will unmarshal data, print to the screen, add a new employee, and marshal the new data back to the XML document but will not modify the new employee.

```
// Modify the last XmlEmployee
//xmlEmployees.setXmlEmployee(getXmlEmployee("8000","600","Peter","Castor",
    "3/3/2001","VP//","USA","3","2096","125000.00"),
    xmlEmployees.getXmlEmployeeCount()-1);
```

- 2 Also add a print line after the `addXmlEmployee()` method that prints the new employee's name to the screen after it's added.

```
//Add an XmlEmployee
xmlEmployees.addXmlEmployee(getXmlEmployee("8000", "400", "Charlie", "Castor",
    "3/3/2001", "VP", "USA", "2", "1993", "155000.00"));
System.out.println("New XmlEmployee's Full Name is " + xmlEmployees.getXmlEmployee(
    xmlEmployees.getXmlEmployeeCount()-1).getFullName());
```

- 3 Run the application by right-clicking `DB_Castor.java` and choosing Run. The application runs, reads the employee information, and prints the following to the message pane:

```
== unmarshalling "Employees.xml" ==
Total Number of XmlEmployees read = 3
First XmlEmployee's Full Name is Nelson, Robert
Last XmlEmployee's Full Name is Lambert, Kim
New XmlEmployee's Full Name is Castor, Charlie
```

- 4 Switch to `Employees.xml` to verify that the new data has been marshalled to the XML document and notice that Charlie Castor has been added as an employee.

**Note** By default, Castor's marshaller writes XML documents without indentation, because indentation inflates the size of the generated XML documents. To turn indentation on, modify the Castor properties file with the following content: `org.exolab.castor.indent=true`. There are also other properties in this file that you may want to modify. The `castor.properties` file is created automatically by the Databinding wizard in the source directory of the project.

## Step 4: Modifying the new employee data

---

Next, modify Charlie Castor's employee record with the `setXmlEmployee()` method and run the program again to update the employee record in the XML document.

- 1 Return to `DB_Castor.java` and comment out the `addEmployee()` method and the print statement below it:

```
// Add an XmlEmployee
//xmlEmployees.addXmlEmployee(getXmlEmployee("8000", "400", "Charlie", "Castor",
//    "3/3/2001", "VP", "USA", "2", "1993", "155000.00"));
//System.out.println("New XmlEmployee's Full Name is " +
//    xmlEmployees.getXmlEmployee(xmlEmployees.getXmlEmployeeCount()-
//        1).getFullName());
```

- 2 Remove the comments from the `setXmlEmployee()` method and add the print statement:

```
// Modify the last XmlEmployee
xmlEmployees.setXmlEmployee(getXmlEmployee("8000", "600",
    "Peter", "Castor", "3/3/2001", "VP", "USA", "3", "2096", "125000.00"),
    xmlEmployees.getXmlEmployeeCount()-1);
System.out.println("New XmlEmployee's Modified Full Name is " + xmlEmployees.
    getXmlEmployee(xmlEmployees.getXmlEmployeeCount()-1).getFullName());
```

- 3 Run the application by right-clicking `DB_Castor.java` and choosing Run. The application runs, reads the employee information, and prints the number of employees read and the first and last employee full names in addition to the modified employee to the message pane:

```
== unmarshalling "Employees.xml" ==
Total Number of XmlEmployees read = 4
First XmlEmployee's Full Name is Nelson, Robert
Last XmlEmployee's Full Name is Castor, Charlie
New XmlEmployee's Modified Full Name is Castor, Peter
```

## Step 5: Running the completed application

---

Now that you understand what this application is doing, remove the comments, add a print statement to read the new employee, remove the new data from the XML document and run the application in its entirety.

- 1 Remove the comments from the `addXmlEmployee()` method and the print statement below it. Your code should look like this:

```
// Add an XmlEmployee
xmlEmployees.addXmlEmployee(getXmlEmployee("8000", "400", "Charlie", "Castor",
    "3/3/2001", "VP", "USA", "2", "1993", "155000.00"));
System.out.println("New XmlEmployee's Full Name is " +
    xmlEmployees.getXmlEmployee(xmlEmployees.getXmlEmployeeCount()
    -1).getFullName());

// Modify the last XmlEmployee
xmlEmployees.setXmlEmployee(getXmlEmployee("8000", "600", "Peter", "Castor",
    "3/3/2001", "VP", "USA", "3", "2096", "125000.00"),
    xmlEmployees.getXmlEmployeeCount()-1);
System.out.println("New XmlEmployee's Modified Full Name is " +
    xmlEmployees.getXmlEmployee(xmlEmployees.getXmlEmployeeCount()
    -1).getFullName());

// Marshal out the data to the same XML file
xmlEmployees.marshal(new java.io.FileWriter(fileName));
```

- 2 Add another print statement after the `setXmlEmployee()` method's print statement and before the `marshal()` method to read the data again after the new data has been added and modified. Your code should look like this:

```
// Modify the last XmlEmployee
xmlEmployees.setXmlEmployee(getXmlEmployee("8000", "600", "Peter", "Castor",
    "3/3/2001", "VP", "USA", "3", "2096", "125000.00"),
    xmlEmployees.getXmlEmployeeCount()-1);
System.out.println("New XmlEmployee's Modified Full Name is
    "+xmlEmployees.getXmlEmployee(xmlEmployees.getXmlEmployeeCount()
    -1).getFullName());
//read employees again
System.out.println("Total Number of XmlEmployees read =
    "+xmlEmployees.getXmlEmployeeCount());
// Marshall out the data to the same XML file
xmlEmployees.marshal(new java.io.FileWriter(fileName));
```

- 3 Remove Peter Castor's employee data from the XML document, being careful not to remove any other data or XML tags.

- 4 Save the project.
- 5 Run the program again to output the data and see the following printout in the message pane:

```
== unmarshalling "Employees.xml" ==  
Total Number of XmlEmployees read = 3  
First XmlEmployee's Full Name is Nelson, Robert  
Last XmlEmployee's Full Name is Lambert, Kim  
New XmlEmployee's Full Name is Castor, Charlie  
New XmlEmployee's Modified Full Name is Castor, Peter  
Total Number of XmlEmployees read = 4
```

- 6 Switch to `Employees.xml` to verify that the data has been marshalled back to the XML document. Notice that Charlie Castor was added and then modified. When the employees are read a second time, the new employee is counted.

Congratulations. You've completed the tutorial. You've read, added, and modified employee data in an XML document using a Java application and Java classes generated by the Databinding wizard from a schema file.



## Tutorial: Transferring data with the model-based XML database components

This is a feature of JBuilder Enterprise.

This tutorial explains how to use JBuilder's model-based XML database components to transfer data from an XML document to a database, and retrieve that data back again from the database to an XML document. It also explains how to use the XML-DBMS wizard to create the required map file used in the transferring of data and how to create a SQL script file you can use to create the database.

This tutorial assumes you have a working knowledge of JBuilder and XML. If you are new to JBuilder, see "The JBuilder environment" in *Building Applications with JBuilder*.

Model-based components use a map document that determines how the data transfers between an XML structure and the database metadata. Because the user specifies a map between an element in the XML document to a particular table or column in a database, deeply nested XML documents can be transferred to and from a set of database tables. The model-based components are implemented using XML-DBMS, an Open Source XML middleware that is bundled with JBuilder.

Working through this tutorial, you'll learn how to do the following:

- Create a map file that maps the elements of a DTD file to the columns in a database table.
- Create a SQL script file that creates the database table metadata.
- Use the `XDBMSTable` component to retrieve data from an XML document to the database table.

- Use the same `XDBMSTable` component to transfer data from the database table to an XML document.
- Use the `XDBMSQuery` component to transfer data from the database table to an XML document.
- Use `XDBMSTable`'s and `XDBMSQuery`'s customizers to set properties and view the results of those property settings on the transfer of the data.

For more information on JBuilder's XML features, see chapters 2 and 3.

## Getting started

---

This tutorial creates an `XmlEmployee` database table that contains basic employee information such as a unique employee number, the employee's name, salary, hire data, job grade, and so on. Before you create the table, you must have a DTD that defines the metadata of the database table you are going to transfer data into and retrieve data from. You can either create one manually or, if you have an XML document with the correct structure, you can use it to create the DTD by using the XML to DTD wizard.

Within JBuilder, open the `\jbuilder5\samples\Tutorials\XML\database\XMLDBMSBeans\XMLDBMSBeans.jpx` project, which contains the DTD file you need, `Employees.dtd`. It looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT XmlEmployee (EmpNo, FirstName, LastName, PhoneExt, HireDate, DeptNo,
    JobCode, JobGrade, JobCountry, Salary, FullName)>
<!ELEMENT DeptNo (#PCDATA)>
<!ELEMENT EmpNo (#PCDATA)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT FullName (#PCDATA)>
<!ELEMENT HireDate (#PCDATA)>
<!ELEMENT JobCode (#PCDATA)>
<!ELEMENT JobCountry (#PCDATA)>
<!ELEMENT JobGrade (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!ELEMENT PhoneExt (#PCDATA)>
<!ELEMENT Salary (#PCDATA)>
<!ELEMENT XmlEmployees (XmlEmployee+)>
```

The sample project also has an `Employees.xml` file. If the project didn't have this file, you could create it using the DTD to XML wizard and specifying the `Employees.dtd` file as the input DTD. You would then modify the resulting XML structure and add the data. Later you'll use `Employees.xml` to populate the database table and modify its data. `Employees.xml`, as it appears in the sample project, contains data on three employees. It looks like this:

```

<?xml version="1.0"?>
<!DOCTYPE XmlEmployees SYSTEM "Employees.dtd">
<XmlEmployees>
  <XmlEmployee>
    <EmpNo>2</EmpNo>
    <FirstName>Robert</FirstName>
    <LastName>Nelson</LastName>
    <PhoneExt>250</PhoneExt>
    <HireDate>1988-12-28</HireDate>
    <DeptNo>600</DeptNo>
    <JobCode>VP</JobCode>
    <JobGrade>2</JobGrade>
    <JobCountry>USA</JobCountry>
    <Salary>105900.000000</Salary>
    <FullName>Nelson, Robert</FullName>
  </XmlEmployee>
  <XmlEmployee>
    <EmpNo>4</EmpNo>
    <FirstName>Bruce</FirstName>
    <LastName>Young</LastName>
    <PhoneExt>233</PhoneExt>
    <HireDate>1988-12-28</HireDate>
    <DeptNo>621</DeptNo>
    <JobCode>Eng</JobCode>
    <JobGrade>2</JobGrade>
    <JobCountry>USA</JobCountry>
    <Salary>97500.000000</Salary>
    <FullName>Young, Bruce</FullName>
  </XmlEmployee>
  <XmlEmployee>
    <EmpNo>5</EmpNo>
    <FirstName>Kim</FirstName>
    <LastName>Lambert</LastName>
    <PhoneExt>22</PhoneExt>
    <HireDate>1989-02-06</HireDate>
    <DeptNo>130</DeptNo>
    <JobCode>Eng</JobCode>
    <JobGrade>2</JobGrade>
    <JobCountry>USA</JobCountry>
    <Salary>102750.000000</Salary>
    <FullName>Lambert, Kim</FullName>
  </XmlEmployee>
</XmlEmployees>

```

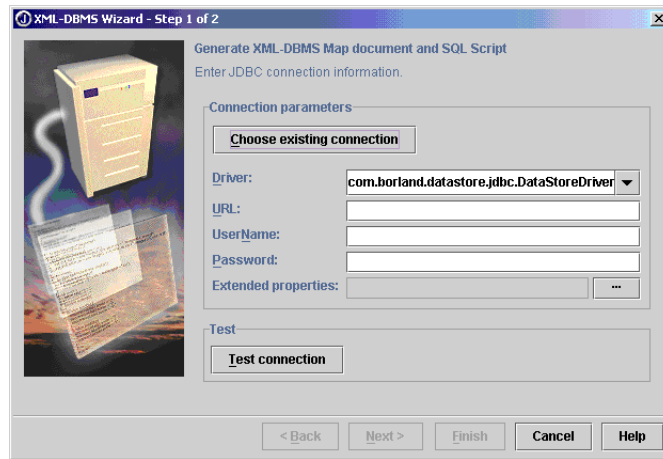
## Creating the map and SQL script files

---

You already have the structure of the database table as defined by the DTD, and you have an XML document that contains data you want to store in the database table. But you don't actually have a database table yet, so you must create it. You also must create a map file that describes how the data is transferred from the XML elements to the correct columns

in the new database. JBuilder's XML-DBMS wizard can create the SQL script file that you can execute to create the table at the same time that it creates the map file needed to transfer the data.

To open the XML-DBMS wizard, choose File | New, click the XML tab, and double-click the XML-DBMS icon.



## Entering JDBC connection information

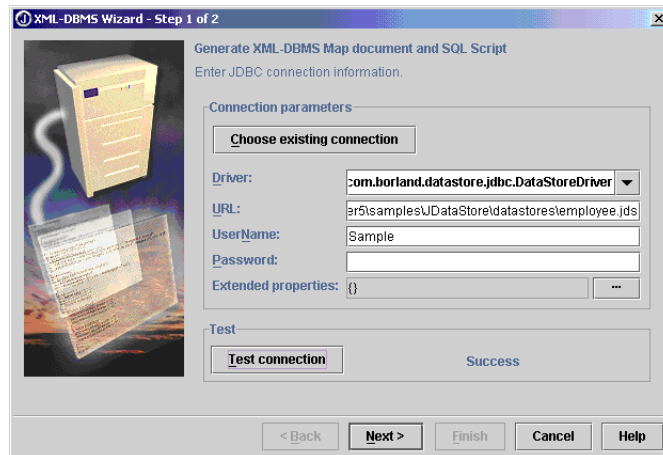
This tutorial uses the `JDataStore employee.jds` database found in the `\jbuilder5\samples\JDataStore\datastores` directory. You may already have an existing JDBC connection to this datastore on your system if you've worked with JDataStore samples. If so, click the Choose Existing Connection button and select it. When you do so, the Connection Parameters are filled in for you. If you don't, you must enter in the information yourself. Follow these steps:

- 1 Select `com.borland.datastore.jdbc.DataStoreDriver` as your Driver from the drop-down list. You must have JDataStore installed on your system. If you need information about working with JDataStore, see *JDataStore Developer's Guide*, "JDataStore fundamentals."
- 2 Specify the URL for the proper datastore you are using, `employee.jds`. When you selected DataStoreDriver as your driver, a pattern appears that guides you in entering the correct URL. Assuming you installed JBuilder on drive C: of your system, the URL to the `employee.jds` datastore in the `samples` directory is this:  

```
jdbc:borland:dslocal:C:\jbuilder5\samples\JDataStore\datastores\employee.jds
```
- 3 Enter Sample as the User Name.
- 4 Enter any value for the Password or just leave the field blank. (`employee.jds` doesn't require a password.)
- 5 Skip the Extended Properties field.

## Testing the connection

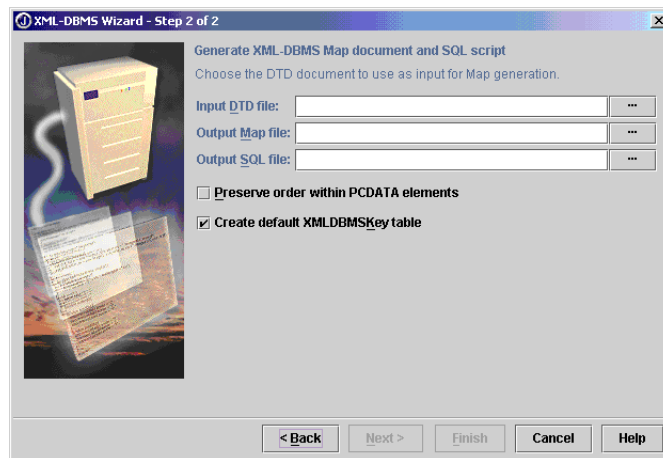
To see if you specified your JDBC connection properly, click the Test Connection button. A Success or Failed message appears on the panel next to the button:



Once your connection is successful, you are ready to go the next page of the wizard. Choose Next.

## Specifying the file names

On the second page of the XML-DBMS wizard, specify the DTD file you are using to create the map file and specify the names you want the generated map file and SQL script file to have.



Follow these steps:

- 1 Use the Input DTD File ... button to navigate to and select the `Employees.dtd` file in the `XMLDBMSBeans.jpx` project.
- 2 Accept the Map File Name JBuilder suggests, `Employees.map`.
- 3 Accept the Output SQL File JBuilder suggests, `Employees.sql`.
- 4 Check the Create Default XMLDBMSKey Table check box.
- 5 Choose Finish.

The XML-DBMS generates the `Employees.map` file and the `Employees.sql` file. They appear in your project. Double-click `Employees.map` to see how the XML elements will be mapped to columns in the database table you will create. If you wanted to change the name of the columns in the database table you are going to create, you could edit the map file and change column names. If you did that, you must also make the same changes to the column names in the SQL script file. For this tutorial, don't make any changes. But often you would want to edit the map and SQL script files the wizard creates to meet your needs.

## Creating the database table(s)

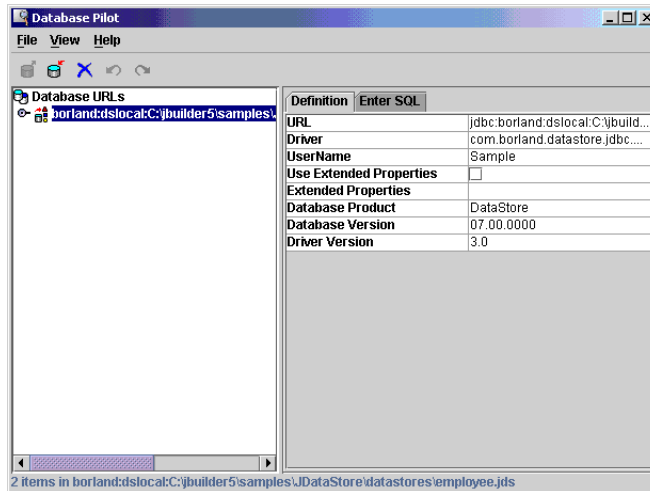
---

Double-click `Employees.sql` in the project pane to see the SQL statements the XML-DBMS wizard generated. When you do, you'll notice that it contains three CREATE TABLE statements, not just one. The first creates an `XmlEmployee` table that contains the metadata the DTD specified with the addition of an "XmlEmployeesFK" column (FK stands for foreign key). The second CREATE TABLE creates an `XmlEmployees` table that contains just one column, "XmlEmployeesPK" (PK stands for primary key). The third creates a `XMLDBMSKey` table. XML-DBMS uses all these tables to represent the structure of the input DTD file.

A convenient way to execute these SQL statements is to use JBuilder's Database Pilot:

- 1 Select all the text in the `Employees.sql` file and choose Edit | Copy to copy it to the clipboard.

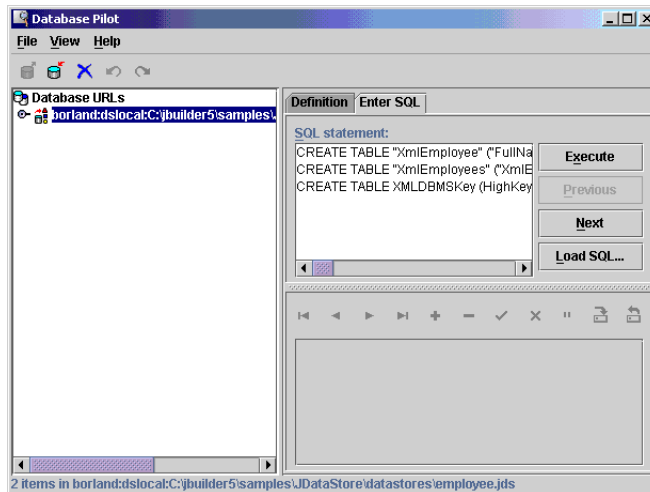
## 2 Choose Tools | Database Pilot to open the Database Pilot.



## 3 Double-click the Database URL you specified in XML-DBMS wizard.

## 4 Click the Enter SQL tab.

## 5 In the SQL Statement box, paste the copied SQL statements:



## 6 Click Execute.

Database Pilot creates the three tables in the `employee.jds` datastore.

## 7 Close Database Pilot.

## Working with the sample test application

---

Usually when you use JBuilder's XML database components, you'll be developing an application that presents a GUI for your users to interact with. You won't be doing that for this tutorial. Instead you'll use the sample application, `XMLDBMS_Test.java`, which is simply a Java class that contains the model-based XML database components and sets the properties of those components. This tutorial shows you how to work with the components' customizers to set properties and view the results of a transfer of data. Once you are certain a transfer works correctly, you can proceed with confidence as you build a GUI application around it.

In the open `XMLDBMSBeans` project, double click the `com.borland.samples.DBMS` package to view its contents. You'll find one file, `XMLDBMS_Test.java`. Double-click it to open it in the editor.

Examine the source code and you'll see that it contains the two components, `XMLDBMSTable` and `XMLDBMSQuery`. If you were creating your own test application, you would simply add these components to your application by following these steps:

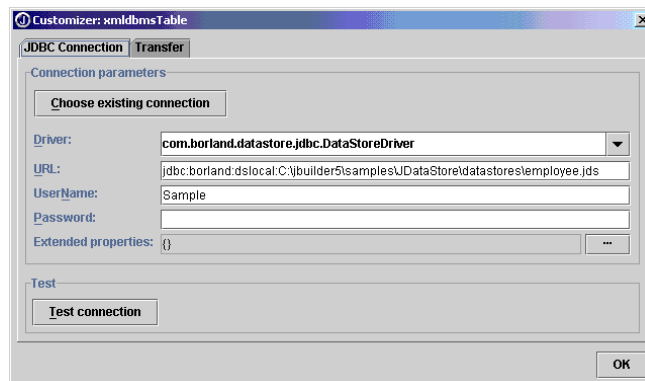
- 1 Display your application class in the editor.
- 2 Click the Design tab.
- 3 Click the XML tab of the component palette.
- 4 Select the `XMLDBMSTable` component and drop it on the UI Designer.
- 5 Select the `XMLDBMSQuery` component and drop it on the UI Designer.

## Using XMLDBMSTable's customizer

---

To begin working with the `XMLDBMSTable` component,

- 1 Click the Design tab while the sample application is open in the editor. You'll see an Other Folder in the structure pane that contains the two model-based components.
- 2 Right-click `xmldbmsTable` in the structure pane and choose the Customizer menu command. The customizer for `xmldbmsTable` appears:





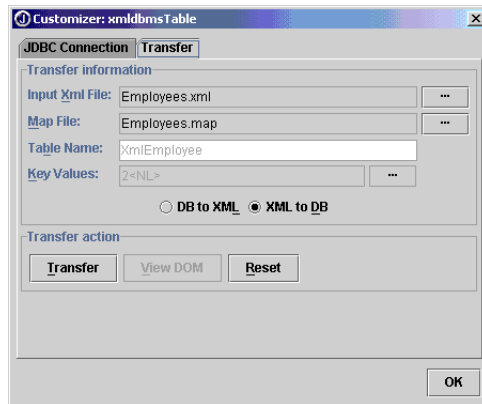
## Selecting a JDBC connection

The sample application already sets all the properties in its source code. If you were creating your own application, you would need to fill in all the fields of the customizer yourself. Imagine that the fields are blank and follow along to fill them in. As you did for the first step of the XML-DBMS wizard, you must select your JDBC connection. Because you have an existing connection already, simply click the Choose Existing Connection button and select the same connection you established to the `employee.jds` datastore.

As soon as you select your connection, the customizer uses the connection data to fill in the rest of the fields. To make sure your connection is specified correctly, click the Test Connection button.

## Transferring data from an XML document to the database table

Click the Transfer tab to view the next page of the customizer:

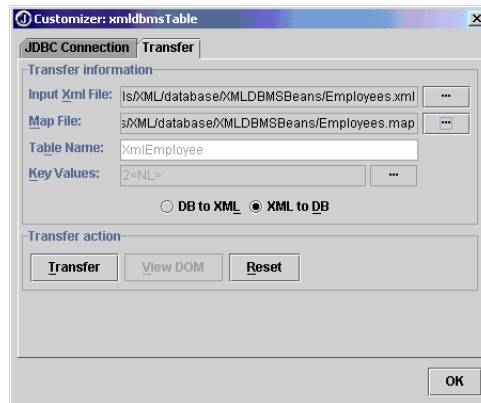


Fill in the required transfer information:

- 1 Click the XML To DB option in the center of the page if it isn't already selected. You are preparing to transfer the data in the `Employees.xml` document to the `XmlEmployee` table.
- 2 Use the Input XML File ... button to navigate to and select the `Employees.xml` file in the project. You should always specify the fully-qualified name, and if you use the ... button, the file name will always include the full path information.
- 3 Use the Map File ... button to navigate to and select the `Employees.map` file in the project.

These are the only fields required for the transfer. The other fields are disabled. You will see data values in them, because the sample application, `XMLDBMSBeans_Test.java`, sets property values for these fields in its source code. The fields aren't used for transferring data from the XML

document to the `XmlEmployees.xml` database table. The customizer should look like this:



To transfer the data in `XmlEmployees.xml` document to the `XmlEmployees.xml` database table you created, choose **Transfer**.

The transfer of data occurs.

## Transferring data from a database table to an XML document

Before you can transfer data from an XML document to the database table, you must modify the `Employees.map` file. You want the “`XmlEmployee`” element to behave as the root. Therefore, `Employees.map` must tell XML-DBMS to ignore the present root, the plural “`XmlEmployees`” element, and instead use the singular “`XmlEmployee`” element. Follow these steps:

- 1 Double-click `Employees.map` in the project pane to open it in the editor.
- 2 Modify the file so that it looks like the following. The text shown in bold is code you add:

```
<?xml version='1.0' ?>
<!DOCTYPE XMLToDBMS SYSTEM "xmldbms.dtd" >

<XMLToDBMS Version="1.0">
  <Options>
  </Options>
  <Maps>
    <IgnoreRoot>
      <ElementType Name="XmlEmployees"/>
      <PseudoRoot>
        <ElementType Name="XmlEmployee"/>
        <CandidateKey Generate="No">
          <Column Name="EmpNo"/>
        </CandidateKey>
      </PseudoRoot>
    </IgnoreRoot>
```

```

<ClassMap>
  <ElementType Name="XmlEmployee"/>
  <ToClassTable>
    <Table Name="XMLEmployee"/>
  </ToClassTable>
  <PropertyMap>
    <ElementType Name="FullName"/>
    <ToColumn>
      <Column Name="FullName"/>
    </ToColumn>
  </PropertyMap>
  <PropertyMap>
    <ElementType Name="JobGrade"/>
    <ToColumn>
      <Column Name="JobGrade"/>
    </ToColumn>
  </PropertyMap>
  <PropertyMap>
    <ElementType Name="Salary"/>
    <ToColumn>
      <Column Name="Salary"/>
    </ToColumn>
  </PropertyMap>
  <PropertyMap>
    <ElementType Name="JobCode"/>
    <ToColumn>
      <Column Name="JobCode"/>
    </ToColumn>
  </PropertyMap>
  <PropertyMap>
    <ElementType Name="PhoneExt"/>
    <ToColumn>
      <Column Name="PhoneExt"/>
    </ToColumn>
  </PropertyMap>
  <PropertyMap>
    <ElementType Name="JobCountry"/>
    <ToColumn>
      <Column Name="JobCountry"/>
    </ToColumn>
  </PropertyMap>
  <PropertyMap>
    <ElementType Name="LastName"/>
    <ToColumn>
      <Column Name="LastName"/>
    </ToColumn>
  </PropertyMap>
  <PropertyMap>
    <ElementType Name="FirstName"/>
    <ToColumn>
      <Column Name="FirstName"/>
    </ToColumn>
  </PropertyMap>
</PropertyMap>

```

```
<ElementType Name="HireDate"/>
<ToColumn>
  <Column Name="HireDate"/>
</ToColumn>
</PropertyMap>
<PropertyMap>
  <ElementType Name="EmpNo"/>
  <ToColumn>
    <Column Name="EmpNo"/>
  </ToColumn>
</PropertyMap>
<PropertyMap>
  <ElementType Name="DeptNo"/>
  <ToColumn>
    <Column Name="DeptNo"/>
  </ToColumn>
</PropertyMap>
</ClassMap>
</Maps>
</XMLToDBMS>
```

To get the above results, you must also remove this block of code:

```
<ClassMap>
  <ElementType Name="XmlEmployees"/>
  <ToRootTable>
    <Table Name="XmlEmployees"/>
    <CandidateKey Generate="Yes">
      <Column Name="XmlEmployeesPK"/>
    </CandidateKey>
  </ToRootTable>
  <RelatedClass KeyInParentTable="Candidate">
    <ElementType Name="XmlEmployee"/>
    <CandidateKey Generate="Yes">
      <Column Name="XmlEmployeesPK"/>
    </CandidateKey>
    <ForeignKey>
      <Column Name="XmlEmployeesFK"/>
    </ForeignKey>
  </RelatedClass>
</ClassMap>
```

### 3 Save the project.

So far you have no proof that any transfer of data occurred except that you received no errors (although you could view the data using Database Pilot). But if data now actually exists in the `XmlEmployee` table, you should be able to transfer data from it to an XML document. You can do this while using the Transfer page of the `XMLDBMSTable`'s customizer.

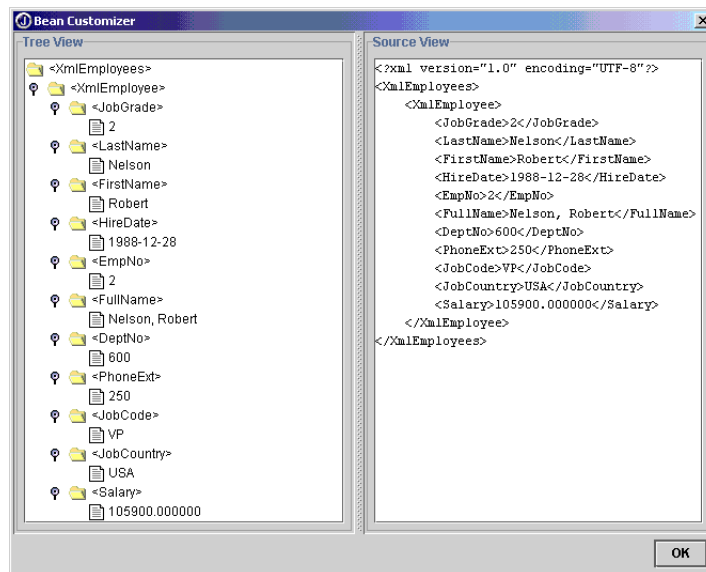
To transfer data from the `XmlEmployee` table to an XML document named `Employees_out.xml`,

- 1 Select the DB To XML option. The customizer fields change slightly.
- 2 Change the name of the Output XML File from `Employees.xml` to `Employees_out.xml`.
- 3 Keep the Map File name as `Employees.map` (file you modified), including the path name.
- 4 Specify the Table Name as `XmlEmployee`.
- 5 Click the Key Values ... button to display the Key Editor and specify a key value of 2:
  - 1 Click the Add button in the Key Editor.
  - 2 Select the added item, change its value to 2, and press *Enter*.
  - 3 Click OK to close the Key Editor.

The value you entered is the value of the “EmpNo” column of the employee you want to transfer from the database table to the `Employees_out.xml` document. The “EmpNo” column is the primary key for `XmlEmployee`. If you want to see the records of multiple employees, use the Key Values field and its property editor to specify multiple employee numbers.

To transfer data from the `XmlEmployee` table to the `Employees_out.xml` document, choose View DOM.

The transfer occurs. You can see the results of your transfer request:



Choose OK.

## Using XMLDBMSQuery's customizer

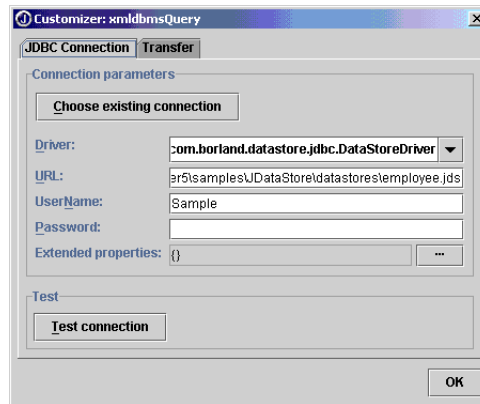
---

You can also use a SQL statement to retrieve data from a database table using the `XMLDBMSQuery` component.

To begin working with the `XMLDBMSQuery` component,

- 1 Click the Design tab while the sample application is open in the editor.
- 2 Right-click `xmlDbmsQuery` in the structure pane and choose the Customizer menu command.

The customizer for `xmlDbmsQuery` appears:

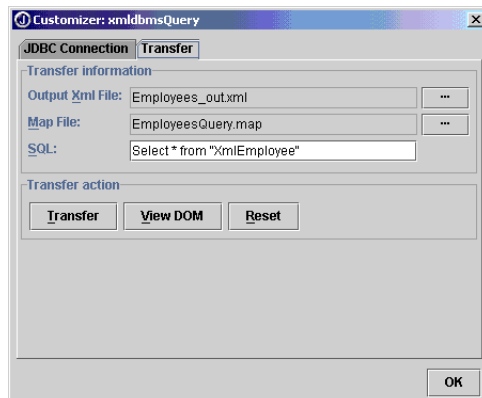


### Selecting a JDBC connection

As you did with the `XMLDBMSTable` component, click the Choose Existing Connection button and specify the connection to `employee.jds` you established earlier.

### Transferring data with a SQL statement

Click Transfer to go to the next page:



Fill in the required transfer information:

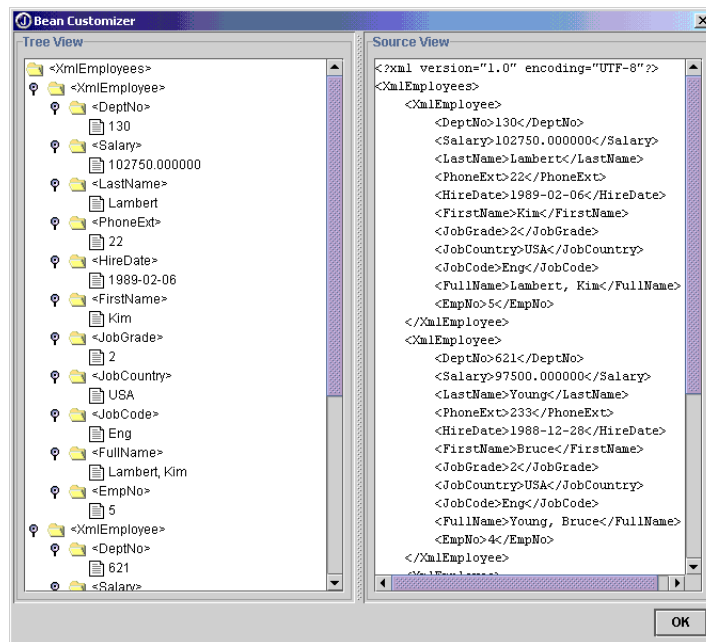
- 1 Use the Output XML File ... button to navigate to and select the `Employees_out.xml` file you created earlier. You should always specify the fully-qualified name, and if you use the ... button, the file name will always include the full path information.
- 2 Use the Map File ... button to navigate to and select the `EmployeesQuery.map` file in the project.
- 3 Enter the following SQL statement as the value of SQL field:

```
Select * from "XmlEmployee"
```

Remember to place the double-quotation marks around the table name. This statement will retrieve all the rows in the `XmlEmployee` table to the `Employees_out.xml` document. Of course, you can use any valid SQL statement you want to query the `XmlEmployee` table. For example,

```
Select * from "XmlEmployee" where "JobCode" = 'VP'
```

- 4 Choose View DOM to see the results. Shown here are the results of the first query:



## Map files for the XMLDBMSQuery component

You probably noticed that the `XMLDBMSQuery` component used a different map file, `EmployeesQuery.map`, than the one used by the `XMLDMSTable` component. This is what the `EmployeesQuery.map` file looks like with the differences between the `Employees.map` file shown in bold:

## Working with the sample test application

```
<?xml version='1.0' ?>
<!DOCTYPE XMLToDBMS SYSTEM "xmldbms.dtd" >

<XMLToDBMS Version="1.0">
  <Options>
  </Options>
  <Maps>
    <IgnoreRoot>
      <ElementType Name="XmlEmployees"/>
      <PseudoRoot>
        <ElementType Name="XmlEmployee"/>
        <CandidateKey Generate="No">
          <Column Name="EmpNo"/>
        </CandidateKey>
      </PseudoRoot>
    </IgnoreRoot>

    <ClassMap>
      <ElementType Name="XmlEmployee"/>
      <ToClassTable>
        <Table Name="Result Set"/>
      </ToClassTable>
      <PropertyMap>
        <ElementType Name="FullName"/>
        <ToColumn>
          <Column Name="FullName"/>
        </ToColumn>
      </PropertyMap>
      <PropertyMap>
        <ElementType Name="JobGrade"/>
        <ToColumn>
          <Column Name="JobGrade"/>
        </ToColumn>
      </PropertyMap>
      <PropertyMap>
        <ElementType Name="Salary"/>
        <ToColumn>
          <Column Name="Salary"/>
        </ToColumn>
      </PropertyMap>
      <PropertyMap>
        <ElementType Name="JobCode"/>
        <ToColumn>
          <Column Name="JobCode"/>
        </ToColumn>
      </PropertyMap>
      <PropertyMap>
        <ElementType Name="PhoneExt"/>
        <ToColumn>
          <Column Name="PhoneExt"/>
        </ToColumn>
      </PropertyMap>
    </ClassMap>
  </Maps>
</XMLToDBMS>
```



```

<PropertyMap>
  <ElementType Name="JobCountry" />
  <ToColumn>
    <Column Name="JobCountry" />
  </ToColumn>
</PropertyMap>
<PropertyMap>
  <ElementType Name="LastName" />
  <ToColumn>
    <Column Name="LastName" />
  </ToColumn>
</PropertyMap>
<PropertyMap>
  <ElementType Name="FirstName" />
  <ToColumn>
    <Column Name="FirstName" />
  </ToColumn>
</PropertyMap>
<PropertyMap>
  <ElementType Name="HireDate" />
  <ToColumn>
    <Column Name="HireDate" />
  </ToColumn>
</PropertyMap>
<PropertyMap>
  <ElementType Name="EmpNo" />
  <ToColumn>
    <Column Name="EmpNo" />
  </ToColumn>
</PropertyMap>
<PropertyMap>
  <ElementType Name="DeptNo" />
  <ToColumn>
    <Column Name="DeptNo" />
  </ToColumn>
</PropertyMap>
</ClassMap>
</Maps>
</XMLToDBMS>

```

When using the `XMLDBMSQuery` component to query the `XmlEmployee` database table, you want the “`XmlEmployee`” element to act as the root. Therefore, the map file must tell XML-DBMS to ignore the present root, the plural “`XmlEmployees`” element, and instead use the singular “`XmlEmployee`” element. If the `EmployeesQuery.map` file didn’t exist as it does in the sample project, you would need to make the changes to the map file yourself. You would add the block of code that begins with `<IgnoreRoot>` and ends with

</IgnoreRoot>. You would also change the output table name to “Result Set”. Finally, you would remove this block of code:

```
<ClassMap>
  <ElementType Name="XmlEmployees"/>
  <ToRootTable>
    <Table Name="XmlEmployees"/>
    <CandidateKey Generate="Yes">
      <Column Name="XmlEmployeesPK"/>
    </CandidateKey>
  </ToRootTable>
  <RelatedClass KeyInParentTable="Candidate">
    <ElementType Name="XmlEmployee"/>
    <CandidateKey Generate="Yes">
      <Column Name="XmlEmployeesPK"/>
    </CandidateKey>
    <ForeignKey>
      <Column Name="XmlEmployeesFK"/>
    </ForeignKey>
  </RelatedClass>
</ClassMap>
```

## Tutorial: Transferring data with the template-based XML database components

This is a feature of JBuilder Enterprise.

This tutorial explains how to use JBuilder's template-based XML database components to retrieve data from a database to an XML file.

This tutorial assumes you have a working knowledge of JBuilder and XML. If you are new to JBuilder, see "The JBuilder environment" in *Building Applications with JBuilder*.

Using the template-based components, you formulate a query and the component generates an appropriate XML document. The query you provide serves as the template that is replaced in the XML document as the result of applying the template. Because there is no predefined relationship between the XML document and the set of database metadata you are querying, the template-based solution is quite flexible. The format of the resulting XML document is flat and relatively simple. You can choose to present the resulting XML document as you like using either the default or custom stylesheets.

This tutorial shows you how to do the following:

- Retrieve data from a database table into an XML document with the `XTable` component.
- Retrieve data from a database table into an XML document using a SQL statement with the `XQuery` component.
- Use `XTable`'s and `XQuery`'s customizers to set properties and view the results of those property settings on the transfer of the data.

For more information on JBuilder's XML features, see chapters 2 and 3.

## Getting started

---

This tutorial uses the same `XmlEmployee` database table you created in Chapter 8, “Tutorial: Transferring data with the model-based XML database components.” If you haven’t worked through that tutorial yet, you should do so now. At the very least, you should use the XML-DBMS wizard to create the map and SQL script files that tutorial describes, and then execute the SQL statements to create the `XmlEmployee` table. Chapter 8, “Tutorial: Transferring data with the model-based XML database components” tells you how.

Within JBuilder, open the sample `\jbuilder5\samples\Tutorials\XML\database\XBeans.jpx` project.

## Working with the sample test application

---

Usually when you use JBuilder’s XML database components, you’ll be developing an application that presents a GUI for your users to interface with. You won’t be doing that for this tutorial. Instead you’ll use the sample application, `XBeans_Test.java`, which is simply a Java class that contains the template-based XML database components and sets the properties of those components. This tutorial shows you how to work with the components’ customizers to set properties and the view the results of a transfer of data. Once you are certain a transfer works correctly, you can proceed with confidence as you build a GUI application around it.

In the XBeans project, double-click the `com.borland.samples.xml.XBean` package to find the `XBeans_Test.java` sample application. Double-click this sample application to display it in the code editor.

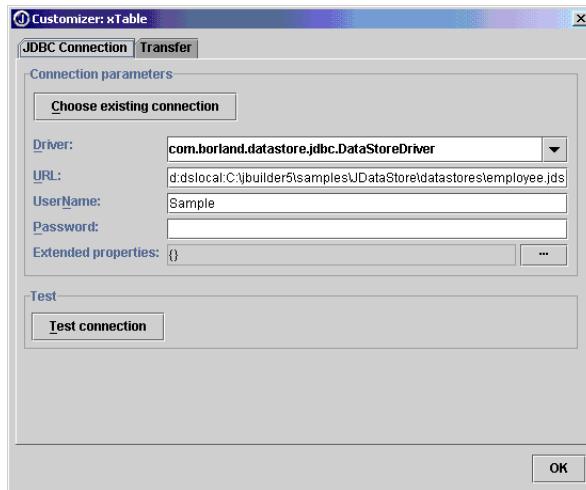
Examine the source code and you’ll see that it contains the two components, `xTable` and `xQuery`. If you were creating your own test application, you would simply add these components to your application by following these steps:

- 1 Display your application class in the editor.
- 2 Click the Design tab.
- 3 Click the XML tab of the component palette.
- 4 Select the `xTable` component and drop it on the UI Designer.
- 5 Select the `xQuery` component and drop it on the UI Designer.

## Using XTable's customizer

To begin working with the XTable component,

- 1 Click the Design tab while the sample application is open in the editor. You'll see an Other Folder in the structure pane that contains the two model-based components.
- 2 Right-click xTable in the structure pane and choose the Customizer menu command. The customizer for xTable appears:



### Entering JDBC connection information

This tutorial uses the JDataStore `employee.jds` database found in the `\jbuilder5\samples\JDataStore\datastores` directory. You may already have an existing JDBC connection to this datastore on your system if you've worked with JDataStore samples. If so, click the Choose Existing Connection button and select it. When you do, the Connection Parameters are filled in for you. If you don't, you must enter in the information yourself. Follow these steps:

- 1 Select `com.borland.datastore.jdbc.DataStoreDriver` as your Driver from the drop-down list. You must have JDataStore installed on your system. If you need information about working with JDataStore, see *JDataStore Developer's Guide: "JDataStore fundamentals."*
- 2 Specify the URL for the proper datastore you are using, `employee.jds`. When you selected DataStoreDriver as your driver, a pattern appears that guides you in entering the correct URL. Assuming you installed JBuilder on drive C: of your system, the URL to the `employee.jds` datastore in the `samples` directory is this:

```
jdbc:borland:dslocal:C:\jbuilder5\samples\JDataStore\datastores\employee.jds
```

- 3 Enter Sample as the User Name.
- 4 Enter any value for the Password. (employee.jds doesn't require one.)
- 5 Skip the Extended Properties field.

To see if you specified your JDBC connection properly, click the Test Connection button. A Success or Failed message appears on the panel next to the button:

## Transferring data from the database table to an XML document

Click the Transfer tab to view the next page of the customizer:

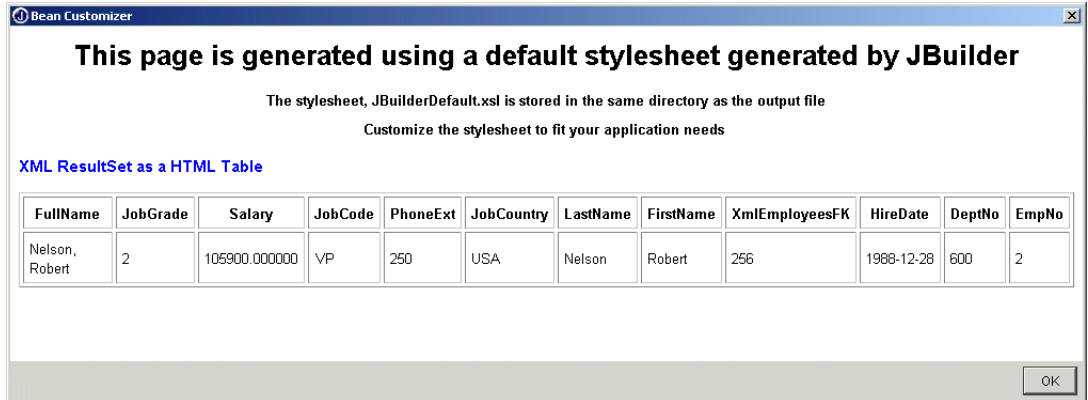
The screenshot shows the 'Customizer: xTable' dialog box with the 'Transfer' tab selected. The 'Transfer information' section includes fields for 'Query File' (blank), 'Output File' (C:/builder5/samples/Tutorials/XML/database/XBeans/XBeans\_out.html), and 'XSL File' (blank). Below these are three sections: 'Column format' with radio buttons for 'As Elements' (selected) and 'As Attributes'; 'Output format' with radio buttons for 'XML' and 'HTML' (selected); and 'Element names' with text boxes for 'Document:' (XmlEmployees) and 'Row:' (XmlEmployee). There is an 'Ignore Nulls' checkbox which is unchecked. The 'Table Name' field contains '"XmlEmployee"'. The 'Keys' field contains '"EmpNo"<NL>'. The 'DefaultParams' field contains '({EmpNo="2"}'. At the bottom, there are 'Transfer', 'View HTML', and 'Reset' buttons, and an 'OK' button at the very bottom right.

Fill in the required transfer information:

- 1 Leave the Query File field blank, as you won't be using a query file for this tutorial. For information about query files, see "XML query document" in *Building Applications with JBuilder*.
- 2 Specify the name of the document you want the data transferred to. You should always provide the fully-qualified name.
- 3 Skip the XSL File field. This tutorial uses the component's default stylesheet.
- 4 Select As Elements as the Column Format option.
- 5 Select HTML as the Output Format option.
- 6 Specify the Document as XmlEmployees and the Row as XmlEmployee for the Element Names options.

- 7 Leave Ignore Nulls unchecked.
- 8 Specify the Table Name as "XmlEmployee".
- 9 Click the Keys ... button to display the Keys Editor and specify a key value of "EmpNo":
  - 1 Click the Add button in the Keys Editor.
  - 2 Select the added item, change its value to "EmpNo", and press Enter. Be sure to add the quotation marks around the value.
  - 3 Click OK to close the Keys Editor.
- 10 Click the DefaultParams ... button to display the Default Params and Param value of "EmpNo" and a value of '2':
  - 1 Click the Add button in the Default Params Editor.
  - 2 Select the added item, change its Param Name value to "EmpNo" and the Param Value value to '2' and press Enter. Be sure to add the double and single quotation marks.
  - 3 Click OK to close the Default Params Editor.

Now you're ready to transfer the data. If you selected the HTML Output Format, choose Transfer, then click View HTML (or simply View HTML to see what the transfer will look like). Your results, which use the default HTML stylesheet, will look like this:



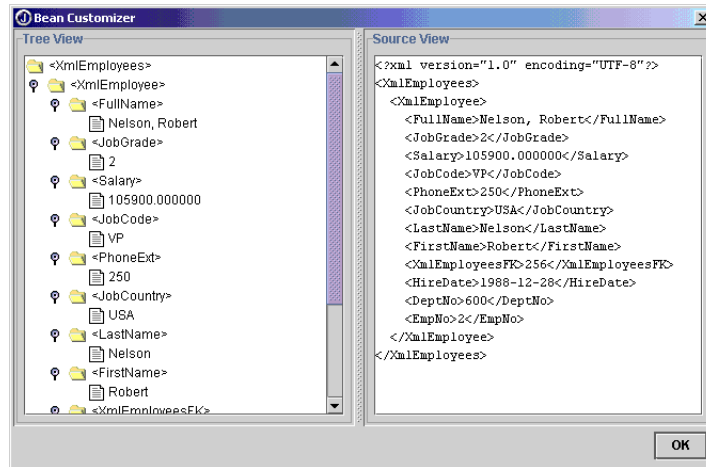
The screenshot shows a window titled "Bean Customizer". Inside, there is a message: "This page is generated using a default stylesheet generated by JBuilder". Below this, it says: "The stylesheet, JBuilderDefault.xml is stored in the same directory as the output file" and "Customize the stylesheet to fit your application needs". A link "XML ResultSet as a HTML Table" is present. Below the link is a table with 12 columns: FullName, JobGrade, Salary, JobCode, PhoneExt, JobCountry, LastName, FirstName, XmlEmployeesFK, HireDate, DeptNo, and EmpNo. The first row of data shows: Nelson, Robert, 2, 105900.000000, VP, 250, USA, Nelson, Robert, 256, 1988-12-28, 600, 2. At the bottom right of the window is an "OK" button.

FullName	JobGrade	Salary	JobCode	PhoneExt	JobCountry	LastName	FirstName	XmlEmployeesFK	HireDate	DeptNo	EmpNo
Nelson, Robert	2	105900.000000	VP	250	USA	Nelson	Robert	256	1988-12-28	600	2

Choose OK.

Now check the XML Output Format option instead. When you do, the View HTML button becomes the View DOM button. Also note that the

Output File you specified now has an .xml file extension. Click View DOM to see default XML tree structure:



Choose OK.

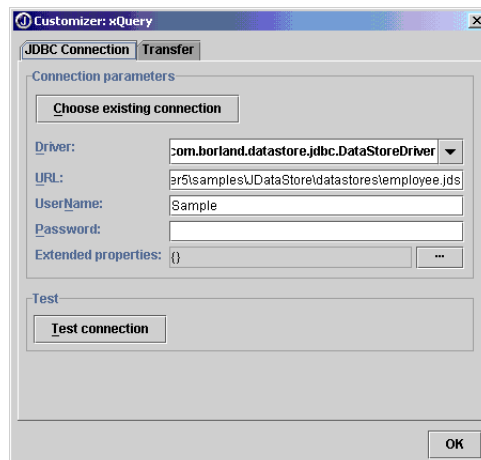
## Using XQuery's customizer

You can also use a SQL statement to retrieve data from a database table using the XQuery component.

To begin working the sample application's XQuery component,

- 1 Click the Design tab while the sample application is open in the editor.
- 2 Right-click xQuery in the structure pane and choose the Customizer menu command.

The customizer for xQuery appears:





## Selecting a JDBC connection

As you did with the `XTable` component, click the Choose Existing Connection button and specify the connection to `employee.jds` you established earlier.

## Transferring data with a SQL statement

Click Transfer to go to the next page:

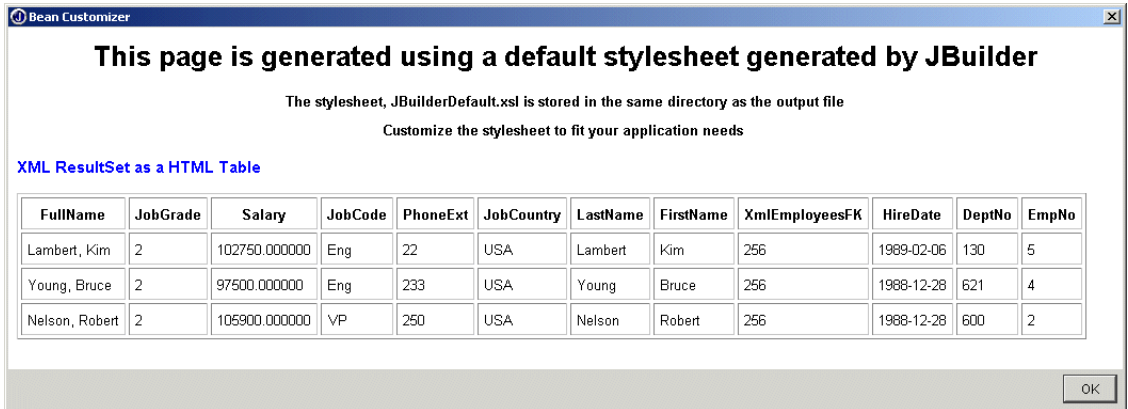
The screenshot shows the 'Customizer: xQuery' dialog box with the 'Transfer' tab selected. The 'Transfer information' section includes fields for 'Query File' (empty), 'Output File' (xQueryOut.html), and 'XSL File' (empty). The 'Column format' section has 'As Elements' selected. The 'Output format' section has 'HTML' selected. The 'Element names' section has 'Document' set to 'XmlEmployees' and 'Row' set to 'XmlEmployee'. The 'Ignore Nulls' checkbox is unchecked. The 'SQL' field contains the statement 'select \* from "XmlEmployee"'. The 'DefaultParams' field is empty. At the bottom, there are 'Transfer', 'View HTML', and 'Reset' buttons, and an 'OK' button at the very bottom right.

Fill in the required transfer information:

- 1 Leave the Query File field blank, as you won't be using a query file for this tutorial. For information about query files, see "XML query document" in *Building Applications with JBuilder*.
- 2 Specify the name of the document you want the data transferred to. You should always provide the fully-qualified name.
- 3 Skip the XSL File field. This tutorial uses the component's default stylesheet.
- 4 Select As Elements as the Column Format option.
- 5 Select HTML as the Output Format option.
- 6 Specify the Document as XmlEmployees and the Row as XmlEmployee for the Element Names options.
- 7 Leave Ignore Nulls unchecked.
- 8 Enter the following SQL statement you want to use to query the database in the SQL field:
 

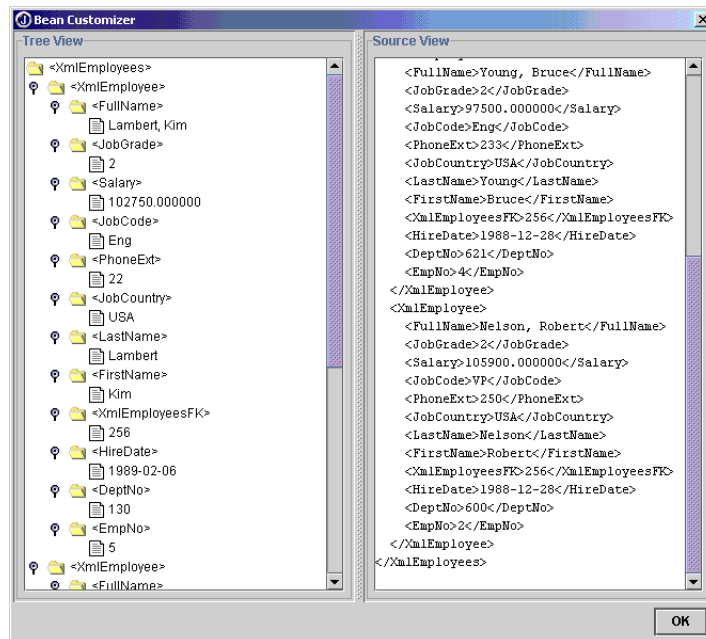
```
select * from "XmlEmployee"
```
- 9 Skip the DefaultParams field.

Choose View HTML to see the results using the default HTML stylesheet:



Choose OK.

Now check the XML Output Format option instead. When you do, the View HTML button becomes the View DOM button. Also note that the Output File you specified now has an .xml file extension. Click View DOM to see default XML tree structure:



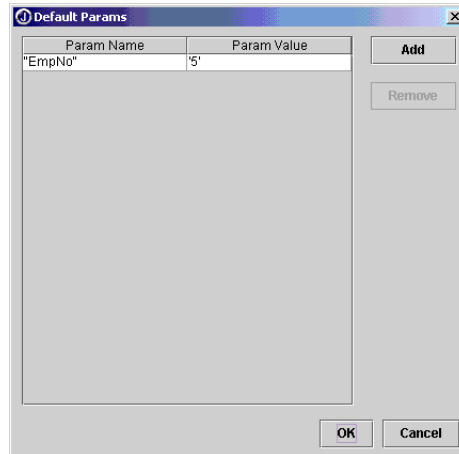
Choose OK.

Next try a parameterized query as your SQL statement:

- 1 Enter this statement in the SQL field:

```
select * from "XmlEmployee" where "EmpNo" = : "EmpNo"
```

- 2 Use the Default Params Editor and add “EmpNo” as the Param Name and ‘5’ as the Param Value:



- 3 Choose OK to close the Default Params editor.

Now choose either View DOM if your output format is XML or View HTML if your output format is HTML. Your results will display the one employee with the employee number of 5.



# Index

## B

---

Borland  
    contacting 1-2  
Borland Online 1-2  
BorlandXML databinding 2-20

## C

---

Cascading Style Sheets (CSS) 2-5  
Castor databinding 2-20  
Castor XML databinding framework 2-22  
Cocoon XML publishing framework 2-9  
    contacting Borland 1-2  
        newsgroups 1-3  
        World Wide Web 1-2  
customizers  
    XML database component 3-2

## D

---

databinding 2-20  
Databinding wizard 2-20, 2-22  
default stylesheet 2-16  
developer support 1-2  
documentation conventions 1-4  
    platform conventions 1-5  
DTD to XML wizard 2-2, 2-4  
DTDs  
    creating from XML documents 2-2, 2-4

## E

---

enabling XML viewer 2-16

## F

---

fonts  
    JBuilder documentation conventions 1-4

## G

---

generating Java classes  
    BorlandXML 2-20  
    Castor 2-22

## J

---

Java classes  
    generating from DTD 2-20  
    generating from schema 2-20, 2-22  
    generating with Databinding wizard 2-20

JDBC connections  
    establishing 3-3  
JDBC drivers  
    specifying 3-3

## L

---

libraries  
    XML 2-17

## M

---

map document 3-10  
marshalling and unmarshalling  
    conversion between Java and XML 2-20  
model-based XML components 3-1, 3-9  
    customizers 3-15  
    setting properties 3-15  
    setting properties with Inspector 3-19  
    specifying transfer information 3-17

## N

---

newsgroups 1-3  
    Borland 1-3

## O

---

object-relational mapping 3-9  
online resources 1-2

## Q

---

query document  
    XML 3-8

## S

---

SAX (Simple API for XML) 2-18  
SAX Handler wizard 2-18  
SAX handlers  
    creating 2-18  
schema files (.xsd) 2-20  
Simple API for XML (SAX) 2-18  
stylesheets  
    applying to XML documents 2-12

## T

---

technical support 1-2  
template-based XML beans  
    setting properties 3-2

- template-based XML components 3-1, 3-2
  - setting properties with Inspector 3-8
  - setting properties with query document 3-8
- tracing
  - enabling 2-12
- Transform Trace options
  - setting 2-16
- transform trace options 2-12
- Transform view toolbar 2-12
- transforming XML documents 2-12
- tutorials
  - Creating a SAX Handler for parsing XML documents 5-1
  - DTD databinding with BorlandXML 6-1
  - Schema databinding with Castor 7-1
  - Transferring data with the model-based XML database components 8-1
  - Transferring data with the template-based XML database components 9-1
  - Validating and transforming XML documents 4-1

## U

---

- Usenet newsgroups 1-3

## V

---

- validating against DTDs 2-7
- validating XML documents 2-7

## W

---

- World Wide Web Consortium 2-1

## X

---

- Xalan stylesheet processor 2-12
- Xalan stylesheeting processor
  - and Xerces 2-12
- XBeans library 3-1
- Xerces parser 2-7
  - and Xalan 2-12
- XML
  - presentation 2-9
  - transformation 2-9
- XML creation 2-2
- XML database components 3-1
- XML database support 2-23
- XML databinding 2-20
- XML defined 2-1
- XML documents
  - applying stylesheets to 2-12
  - creating from DTDs 2-2
  - manipulating programmatically 2-17

- transforming 2-12
- validating 2-7
- viewing 2-5
- XML features in JBuilder 2-1
- XML grammar
  - validating
    - XML error messages 2-7
- XML libraries 2-17
- XML map document 3-10
- XML model-based components 2-23
- XML options
  - setting 2-16
- XML publishing 2-9
- XML query document 3-8
- XML query documents 3-4
- XML template-based components 2-23
- XML to DTD wizard 2-2
- XML traces
  - enabling 2-12
- XML validation 2-2
- XML viewer
  - enabling 2-5, 2-16
- XML well-formedness 2-7
- XML-DBMS 2-23, 3-9, 3-10
  - location 3-9
  - mapping-language 3-10
  - support in JBuilder 3-11
- XML-DBMS wizard 3-11
- XMLDBMSQuery component 2-23, 3-9
  - entering transfer information 3-17
- XMLDBMSQuery customizer 3-15
  - establishing a JDBC connection 3-16
- XMLDBMSTable component 2-23, 3-9
- XMLDBMSTable customizer 3-15
  - entering transfer information 3-17
  - establishing a JDBC connection 3-16
- XQuery component 2-23
- XQuery customizer
  - entering transfer information 3-4
  - establishing JDBC connection 3-3
  - parameterized query 3-6
  - transferring to HTML 3-7
  - transferring to XML 3-7
- XSLT (Extensible Stylesheet Language Transformations) 2-12
- XSLT default stylesheet 2-5
- XTable component 2-23
- XTable customizer
  - entering transfer information 3-4
  - establishing JDBC connection 3-3
  - transferring to HTML 3-7
  - transferring to XML 3-7
  - using parameters 3-6