SYBASE®

Message Bridge for Java™ User's Guide

EAServer

Version 5.2

DOCUMENT ID: DC36716-01-0520-01

LAST REVISED: January 2005

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

Audience

The audience for this document is composed of IT professionals and applications developers building new Web applications in a Java 2 Enterprise Edition (J2EE) environment. Sybase assumes that these professionals have training in Java and XML.

How to use this book

Use this document to understand the use of Sybase Message Bridge for Java™.

Table 1 describes the contents of this book.

Table 1: Chapter descriptions

Chapter	Contents
Chapter 1, “Introduction to Sybase Message Bridge for Java™”	Description of what Message Bridge is
Chapter 2, “Understanding the DataBean Framework”	Description of the framework for DataBeans
Chapter 3, “Using the Message Bridge GUI”	Description of procedures in the Message Bridge GUI
Chapter 4, “Using Views”	Description of views, what they are and how to use them
Chapter 5, “Locating the Java documentation”	Where to find Javadoc
Chapter 6, “Advanced Topics”	Description of using generated code, directory structure, and locating DTDs
Chapter 7, “Using Message Bridge Samples”	Description of code generated from Message Bridge
Chapter 8, “Supported Elements and Declarations for Schemas”	Description of the supported DTD declarations, and supported XML Schema elements

Conventions

This section describes the conventions used in this manual, including terminology and format. The following table shows some of the style conventions used in the documentation for this product.

Table 2: Style conventions

Information Type	Example
• Programs • Utilities • Procedures • Commands	create connection
• File names • Directory names • Properties	*sybase/Message Bridge/bin*
• Code examples • Screen text	`Message Bridge`
• User input • Command line input	`start.bat`
Variables (replace these with the appropriate values for your site)	*host_name*
Variables in code that you type (replace these with the appropriate values for your site)	<designated Jaguar server>

Other sources of information

Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Select Products from the navigation bar on the left.
3. Select a product name from the product list and click Go.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. Click a Certification Report title to display the report.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.

4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the "Technical Support Contact" role to your MySybase profile.

5 Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Accessibility features

EAServer has been tested for compliance with U.S. government Section 508 Accessibility requirements. The online help for this product is also provided in HTML, JavaHelp, and Eclipse help formats, which you can navigate using a screen reader.

EAServer Manager supports working without a mouse. For more information, see "Keyboard navigation" on page 16.

The WST plug-in for Eclipse supports accessibility features for those that cannot use a mouse, are visually impaired or have other special needs. For information about these features refer to Eclipse help:

1 Start Eclipse

2 Select Help | Help Contents

3 Enter Accessibility in the Search dialog box

4 Select Accessible user interfaces or Accessibility features for Eclipse

---

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For additional information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

Introduction to Sybase Message Bridge for Java™

Why use Message Bridge

Message Bridge simplifies and speeds development of applications for today's competitive businesses who want to:

- Generate and accept electronic orders
- Tightly manage the manufacturing process, delivering on time without leaving extra product on the shelves
- Streamline internal processes to improve employee productivity

Message Bridge allows you to build these kinds of collaborative applications enabling developers who need to access, generate, manipulate, and exchange documents and messages quickly and effectively.

The problem

The applications making a difference today are using the Web to enable communication and cooperation between businesses. This new focus forces Information Technology (IT) to go well beyond traditional department or company boundaries. Developing these new applications can be extremely slow and difficult. Data definitions taken from off-the-shelf applications can be cryptic and arcane. Custom application development and integration projects too often use definitions decipherable only to a company's IT "insiders". An effective platform for application interoperability is a goal rarely achieved.

Sometimes useful abstractions can be layered on top of a legacy application. But these abstractions are difficult to design and hard to keep up to date when support for new documents and messages needs to be added every day to a company's application portfolio.

Java and XML capabilities

Java and Extensible Markup Language (XML) are breaking down some of the traditional barriers to building collaborative applications. The Java platform's portable code capability allows Java applications to be deployed to a wide variety of platforms and to any application tier. The Java language itself is a well-designed object-oriented programming language that allows Java developers to easily build applications by creating hierarchies of objects.

XML is a syntax for defining markup languages that in many ways is analogous to Java. Using XML, documents or messages become hierarchical containers within which the data content is "marked up" in a manner that is totally independent of platform or language. Java's object-orientation lends itself well to representing the hierarchy of XML data as a hierarchy of objects. It is important to note that XML can be processed by any language or application—there is no specific tie to Java. As such, XML allows applications to exchange data with minimal dependencies between the producer of a document or message and its recipient.

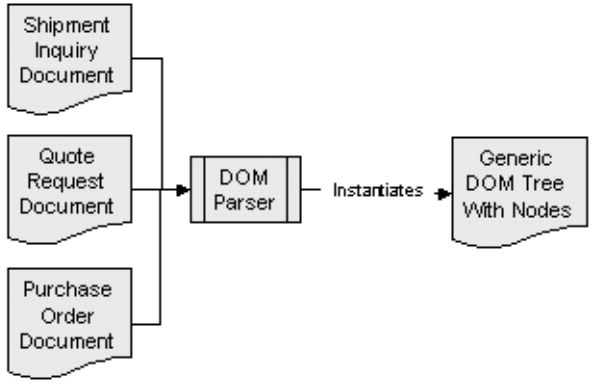
XML parser strengths and limitations

The Java community has made robust XML parsers available to developers for free, and Sun Microsystems has even defined a standard set of Java APIs for XML Parsing (JAXP). JAXP provides a straightforward API for developers to load DOM or Simple API for XML (SAX) XML parsers, and each parser provides methods that allow a developer to access the content of any XML document.

Each of these parser APIs offer its own strengths. Document Object Model (DOM) parsers allow developers to load the data of an entire XML document into memory, and provide powerful features to allow developers to modify the document while it is in memory. Using DOM, developers can both deserialize XML (read a document into in-memory objects) and serialize XML (for example, write a document out to disk). In contrast, because SAX XML parsers are read-only you do not use them to build a new XML document. The SAX event-based parser is faster and consumes far less memory than the DOM parser; consequently, it allows developers to parse the data out of an XML document more effectively.

While these parsers are powerful tools, both are limited in their ability to handle the data in XML documents productively. Although the SAX parser is efficient, when a parse is complete, the only data remaining for an application's use is that which the developer wrote custom code to store. This is acceptable if the developer wants access to only one or two pieces of data in the document and does not mind writing SAX callbacks to capture and store the relevant data in some other custom objects they wrote. But because the document data is not stored in memory, it can require a lot of code on the developer's part to perform data processing on the document (for example, calculating the total dollar value of a series of order line items). The developer can easily use the DOM parser to store the XML document in memory and provide Java APIs to navigate through the document. Unfortunately, the interface to every single XML document is identical in DOM, as shown in Figure 1-1.

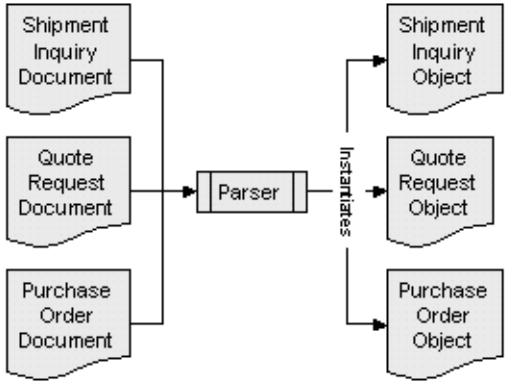
Figure 1-1: DOM's generic representation of unique documents



The DOM parse tree makes every document look the same from a programming API perspective: A request for quote, a purchase order, and a shipping inquiry appear identical to the developer. A purchase order has a reference number, customer information, payment terms, and a list of items being ordered—not generalized document objects, like nodes, node lists, node maps, and so on. This generic abstraction limits developer productivity: Java objects are most valuable when they have some resemblance to the physical entity being modeled.

An ideal solution would be to represent the specific document hierarchy in a corresponding object hierarchy, without requiring months of developer time to achieve the kind of useful data abstraction that permits the rest of the application to be developed quickly. This solution is shown in Figure 1-2.

Figure 1-2: Document to object hierarchy mapping



The solution

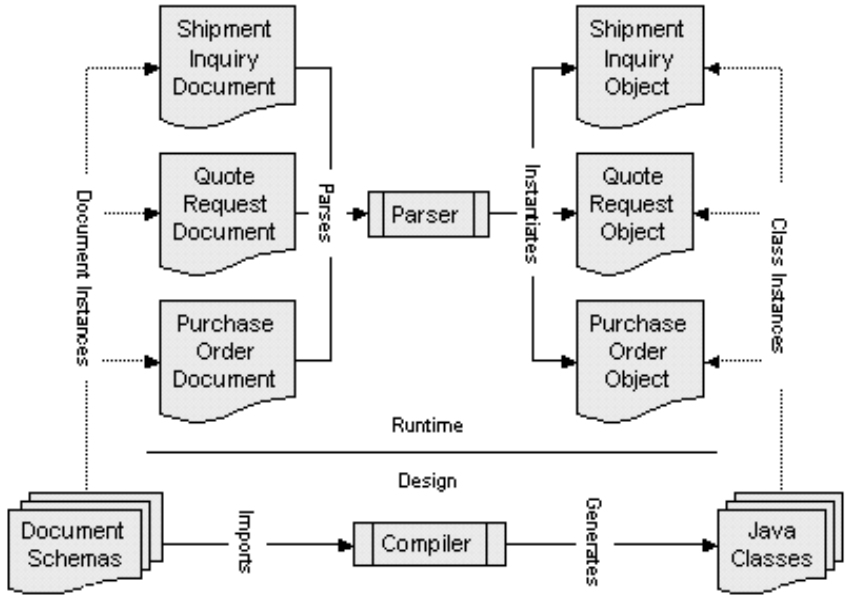
So, how might a solution be approached? The World Wide Web Consortium's (W3C) XML specification does outline rules that XML documents must follow to be considered "well-formed". But two well-formed purchase orders can look very different from each other. Even if they share the same concepts, such as reference number and customer information, the flexibility of XML would make two companies hard pressed to exchange these documents reliably.

Fortunately, the creators of XML recognized the need to allow document structures to be more finely constrained so that multiple parties could interpret them consistently. Generically speaking, the set of rules that outline a document's components, structure, and content is called a schema. Schemas can be written in all sorts of languages, but the important schemas that constrain XML are Document Type Definitions (DTDs) and XML Schema. Documents that conform to a schema are said to be "valid" instead of just "well-formed". Therefore, developers who want to use Java objects to represent the data in XML documents should model their Java classes from whatever schema constrains the XML document.

What is Message Bridge?

Message Bridge compiles schemas into Java classes during design and binds specific XML documents to specific Java objects at runtime, as shown in Figure 1-3.

Figure 1-3: Message Bridge design and runtime



Message Bridge helps developers build applications that make use of structured messages, such as XML documents or messages exchanged between enterprise systems or business partners. Message Bridge improves developer productivity by modeling the schema of a document or message as Java classes. When used in an application, these classes provide an intuitive way to access and manipulate message content in memory, and to read and write messages to and from the network.

Message Bridge provides a schema compiler that binds a document or message schema into Java classes. Each class provides access to the data content of the corresponding schema component through accessor (get) and mutator (set) methods similar to those used in standard JavaBeans. Because these classes model the data content of a document or message instance, we refer to them as *DataBeans*. In short, a DataBean is a Java binding of a particular schema.

Message Bridge features

Features are:

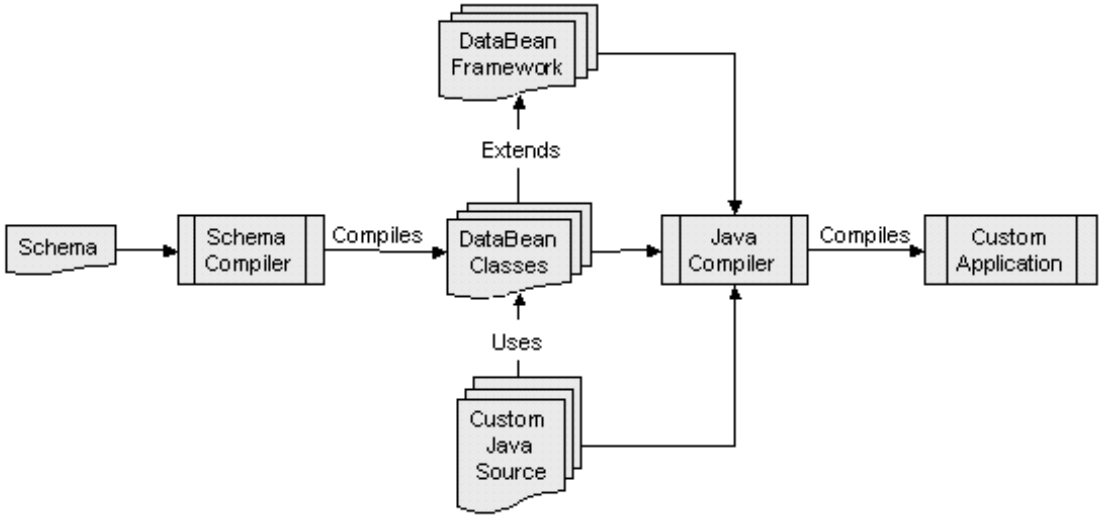
- Message Bridge represents document or message content in memory so it can be used for data-centric applications, yet the representation is not generic—it represents data at the same conceptual level as the document's schema.
- Message Bridge provides a graphical design environment that developers use to import document or message schemas and automatically generate corresponding Java classes. As a result developers can quickly get access to document-specific data abstractions without having to spend time and effort modeling these abstractions themselves.
- Developers using code generated by Message Bridge do not need to write low-level parser code. As a result, they have the freedom to extend the generated classes to fine-tune the data abstractions but do not need to create the abstractions themselves. And Message Bridge provides additional developer artifacts (XML DTDs, XML Schemas, and HTML documentation for DataBeans) to aid development of applications using DataBeans.

How does Message Bridge work?

Message Bridge generates code (DataBeans) that developers use in their application to make generation and consumption of document and message data easier.

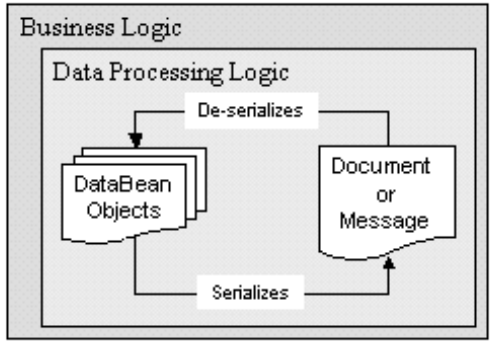
Developers use Message Bridge to import XML DTDs and XML Schemas. Message Bridge converts these schemas into a neutral representation that developers can modify, enhance, and group into projects with other related schemas, as shown in Figure 1-4.

Figure 1-4: Building DataBeans



Using Message Bridge, developers can generate DataBeans for the individual schema definitions they select. These DataBean classes abstract the data contained in documents or messages in an intuitive manner. Each articular DataBean leverages shared runtime classes. The DataBean framework serializes and deserializes content from the network, validates content, and provides a read/write in-memory representation of message data.

Figure 1-5: Using DataBeans



Message Bridge can also generate artifacts to assist developers in using DataBeans in their applications. These artifacts facilitate development in various ways. For example, the XML DTD and Schema provide the developer with content model descriptions of each DataBean. By using these content models during design, developers are able to bring their own XML-based tools to bear, easily modeling runtime systems based on XML data authoring, manipulation, and transmission. The HTML documentation provides the Java developer with a detailed view of each particular DataBean's content model, facilitating the incorporation of particular DataBeans and the DataBean framework into their own custom applications.

CHAPTER 2

Understanding the DataBean Framework

At the heart of Sybase Message Bridge for Java™ is the *DataBean*, which models the content of a specific document or message schema. It can be used to access and manipulate the content of document instances or messages that conform to the schema.

Binding framework

Individual DataBeans rely on an underlying binding framework that allows the components of a document to be mapped to in-memory objects reflecting the document's structure. These objects can then be consumed, manipulated, and possibly serialized back into a document. The runtime classes making up the binding framework provide the implementation of all get() and set() methods exposed for access to class instances, attributes, and data. They also handle deserialization (parsing a document into a set of related classes), serialization (generating a document from the data stored in the related classes) and validation.

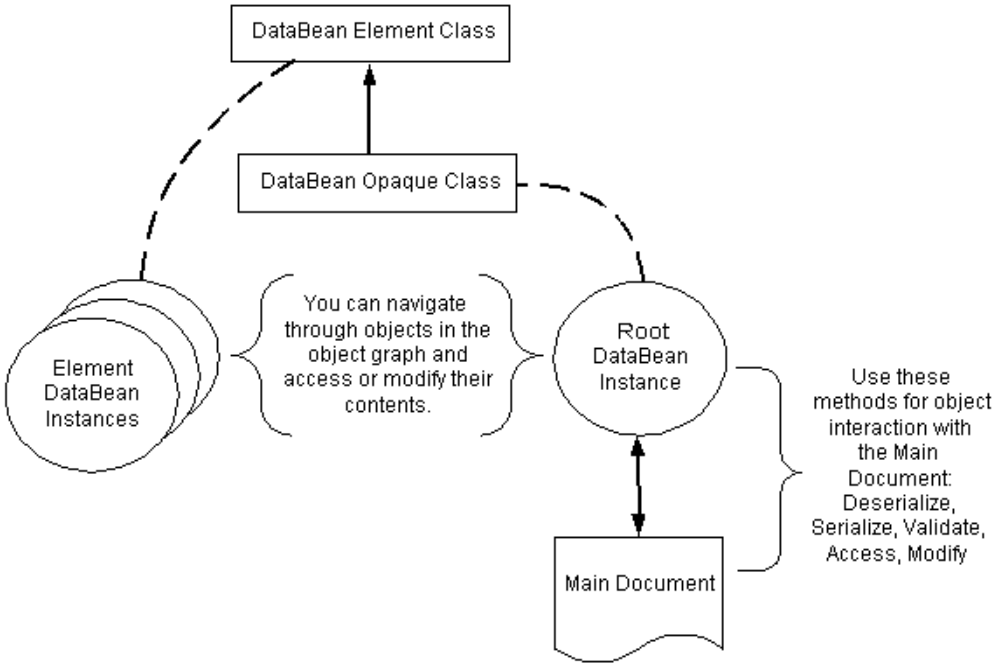
DataBean object graph

It is important to understand the DataBean object graph in order to access these features correctly. When a schema is imported into Message Bridge, a root element is identified directly by Message Bridge itself, or is selected by the user. When code is generated, Message Bridge creates two types of DataBeans::

- A Root DataBean that represents the root element of a complete document described by a schema
- Zero or more additional Element DataBeans that represent the document's member elements, attributes, and data

Both Root and Element DataBeans expose various types of get() and set() methods that are used to navigate through objects in the object graph and to access or modify the content of those objects. All DataBeans extend class DataBeanElement.

Figure 2-1: DataBean object graph



Root DataBeans expose additional methods for interacting with the *main document* they represent. This main document is an instance of a document that conforms to a schema and has a given root. All Root DataBeans extend class DataBeanOpaque, which itself extends class DataBeanElement.

DataBean method summary

All DataBeans expose the following methods:

- get() methods to access member class instances, attributes, and data
- set() methods to modify member class instances, attributes, and data

Root DataBeans also expose the following additional methods:

- Constructors to create a new DataBean object graph

- get()and set() methods to identify and retrieve the main document
- get() and set() methods to access and modify the root name and the location of the source schema for the main document
- Methods to serialize and deserialize a document
- A method to validate a document

Names and identifiers

The set of strings allowed in schemas is much larger than the set of valid Java class identifiers. Therefore, Message Bridge will change the names of schema components when generating DataBeans, according to the following approach. Message Bridge:

- Splits the XML name into a word list by removing any leading and trailing punctuation characters and then searching for word breaks
- Converts the first character of each word to uppercase
- Converts the rest of the characters of each word to lowercase

An example

A good way to illustrate DataBean is to work through an example. In this example, Message Bridge generates bindings for a *Recipe* schema defined in the following DTD:

Recipe DTD schema

The following DTD, called Recipe, is mapped to Java classes described in Table 2-1 on page 13.

XML document instances conforming to this DTD can have two potential roots: *recipe* and *comments*. In this example, *recipe* is the root.

```
<!ELEMENT Recipe (#PCDATA | IndexCard | IngredientList
StepList)* >

<!ELEMENT IndexCard (Name, Description, Source) >

<!ELEMENT Name (#PCDATA)>

<!ELEMENT Description (#PCDATA)>

<!ELEMENT Source (#PCDATA)>

<!ELEMENT IngredientList (Ingredient+)>

<!ATTLIST IngredientList serves CDATA #REQUIRED>

<!ELEMENT Ingredient (Food,Quantity) >

<!ELEMENT Food (#PCDATA)>

<!ELEMENT Quantity (#PCDATA)>

<!ELEMENT StepList (Step+) >

<!ELEMENT Step (#PCDATA)>

<!ELEMENT Comments (#PCDATA)>
```

Schema to class mappings

A summary of how schemas are mapped to classes follows:

- For each element in the schema there is a corresponding Java class.
- If the element has data, setData() and getData() methods are provided to access and manipulate the element data.
- An element containing attributes is mapped to a Java class containing an instance variable for the entire set of attributes.
- Elements having attributes contain the methods setAttribute<attribute name>() and getAttribute<attribute name>() in their class for accessing and manipulating the attributes.
- For elements with children, get() and set() methods are provided to access and modify child elements.

- The get<element name>() methods are used to access child element class instances. They return opaque java.lang.Object. The get() methods always return a valid class instance, and through the hasValidElement() method, you can find out if the class instance holds a valid element.
- The set<element name>() methods are available to access the existing child element class instances or create new child element class instances.
- The root element class has more methods than member element classes because validation, serialization, deserialization and other methods are provided in the root class only.

The following table shows some highlights of the DataBeans generated for the Recipe DTD:

Table 2-1: DTD to Java class mapping

DTD element	Generated Java class

An element containing only text.
<!ELEMENT Name (#PCDATA)>
	A single Java class representing the element, with public accessor and mutator methods for the text.
	`Class: Name`
	`Methods:`
	`public String getData()`
	`public void setData( String data )`

An element containing child elements.
<!ELEMENT Ingredient (Food, Quantity)
	A Java class representing the parent element, with public accessor and mutator methods for the child element.
	`Class: Ingredient`
	`Methods:`
	`public Food getFood()`
	`public Food setFood()`
	`public Quantity getQuantity()`
	`public Quantity setQuantity()`

An element containing a repeating child and attributes.
<!ELEMENT IngredientList (Ingredient+)>
<!ATTLIST IngredientList serves CDATA #REQUIRED>
	A Java class representing the parent element, with public methods for accessing and modifying the list of children.
	`Class: IngredientList`
	`Methods:`
	`public Ingredient setIngredient( int index )`
	`public Ingredient getIngredient( int index )`
	`java.lang.Object (getint index)`
	`java.util.List get Ingredient()`
	`void set Ingredient (Ingredient instance, int index)`
	`java.lang.String getAttributeServes(java.lang.String attrValue)`

DTD element	Generated Java class

An element containing child elements and text. Recipe is the root.
 	
<!ELEMENT Recipe (#PCDATA | IndexCard | IngredientList | StepList)*>
	A Java class representing the parent with accessor and mutator methods for each child, for the text data, and for the object. The get object method is for any element that has more than one child and enables you to retrieve the children in the order they were originally specified.
	
	`Class: Recipe`
	
	`Methods:`
	`public java.util.List getIndexCard()`
	`public IndexCard getIndexCard(int index)`
	`public void setIndexCard(IndexCard instance, int index)`
	`public IndexCard setIndexCard(int index)`
	`public java.util.List getIngredientList()`
	`public IngredientList getIngredientList(int index)`
	`public void setIngredientList(IngredientList instance, int index)`
	`public IngredientList setIngredientList(int index)`
	`public java.util.List getStepList()`
	`public StepList getStepList(int index)`
	`public void setStepList(StepList instance, int index)`
	`public StepList setStepList(int index)`
	`public Data getData( int index )`
	`public Data setData( String data, int index )`
	`public Object get( int index )`
	`public void setMainDocument ( com.sybase.DataBean.serializable. DataBeanSerializable mainDocument )`
	`public void validate()`
	`public void serializeXML ( java.io.OutputStream ostrm )`
	`public void deserializeXML ( java.io.InputStream istrm, boolean validate )`

Using the generated DataBeans

This section contains examples of code written using generated DataBeans from the previous *Recipe* DTD.

Constructing instances of classes from an XML document

The following example illustrates constructing instances of *Recipe* classes from an XML document. In this example, `false` means there is no validation against a DTD document.

```
FileInputStream fiStream = new FileInputStream
   xmlFileName);

Recipe iRecipe = new Recipe(fiStream, false);
```

or

```
FileInputStream fiStream = new FileInputStream
   xmlFileName);

Recipe iRecipe = new Recipe();

iRecipe.deserializeXML(fiStream, false);
```

Getting the IngredientList from the Recipe

```
IngredientList ingList = iRecipe.getIngredientList();
```

Using getAttribute to get the number of servings

```
String serves = ingList.getAttributeServes();
```

Getting the ingredient elements from the IngredientList

To get the first ingredient:

```
   Ingredient ing = ingList.getIngredient(0);

   if( !ing.hasValidElement())

   {

         // We are at the end of the list of Ingredients
         ing=null;
```

```
        break;
    }
```

To target all ingredients:

```
java.util.List ings=ingList.getIngredients();
```

Constructing a new instance of Recipe from scratch (a new Recipe XML document)

```
Recipe oRecipe = new Recipe();
```

Setting the food and quantity of an Ingredient in a Recipe

```
oRecipe.setIngredientList(0).setIngredient(0).
   setFood().setData("Onions");

oRecipe.setIngredientList(0).setIngredient(0).
   setQuantity().setData("2")
```

Creating an XML document from instances of Recipe class

```
FileOutputStream foStream = new FileOutputStream
   (xmlFileName);

oRecipe.serializeXML(foStream);
```

Using DataBeanOpaque to get DTD name prior to choosing binding classes

```
FileInputStream fiStream =
   new FileInputStream(xmlFileName);

DataBeanOpaque recipeVersionUnknown =
   new DataBeanOpaque(fiStream, false);

String dtdName = recipeVersionUnknown.
   getDocTypeSystemID();

if (dtdName.equals("recipe.dtd"))

{

     // Do something with version 1 recipes.
```

```
    com.cooksmart.v1.Recipe = new
    com.cooksmart.v1.Recipe(recipeVersionUnknown,
    true);
  .
  .
}
else
{
  if (dtdName.equals("recipe_v2.dtd"))
  {
      // Do something with version 2 recipes.
      com.cooksmart.v2.Recipe = new
   com.cooksmart.v2.Recipe(recipeVersionUnknown,    true);
 .
 .
   }
 }
```

CHAPTER 3

Using the Message Bridge GUI

This chapter describes how to use the Message Bridge GUI to create and administer schema groups and message definitions, and to generate code. It covers the following topics:

- Schema groups
- Message definitions
- Code generation

Schema groups

You can organize your projects by schema group. A schema group might contain different message definitions for the same schema, or message definitions for multiple schemas.

At start-up, Message Bridge checks to see if any schema groups have been created and saved in a previous session. If it finds none, Message Bridge opens the Default Group, which you can rename.

This section covers the following topics:

- Adding a new schema group
- Changing the name of a schema group

Adding a new schema group

❖ **To add a new schema group**

1. Go to File | New | Schema Group. The Schema Group Name dialog box opens.
2. Enter a name for the new schema group. Click OK. The new schema group appears in the Definitions panel.

You can now add new message definitions to your schema group. See Message definitions for details.

Changing the name of a schema group

❖ **To change the name of an existing schema group**

1 In the Definitions panel, select the Schema Group name you want to change.

2 Right-click the name. The Rename Object dialog box opens.

3 Enter the new name and click OK. The new name appears in the Definitions panel.

Message definitions

When you add a new message definition, you are associating a root element with a schema you have imported into the Message Bridge GUI. The root element is defined in the imported schema. Sometimes a message definition incorporates all the root elements from an imported schema. Other times, your message definition may use only one of many root elements of the imported schema, making it in effect, a subset of the schema.

Your message definition determines the API of the code Message Bridge generates. The generated code binds the schema to a Java DataBean that you can use in an application. See Chapter 2, “Understanding the DataBean Framework”, for more information on DataBeans.

This section covers the following topics:

- Adding a message definition for a DTD
- Adding a message definition for an XML Schema

Adding a message definition for a DTD

You can add a message definition by importing a Document Type Definition (DTD). When you add a message definition, Message Bridge binds the root element to the DTD. To add a message definition, you must know the root elements of the DTD you want to import.

❖ **To add a message definition for a DTD**

1 In the Definitions panel, select a Schema Group. Select Default Group if no others exist. See Schema groups for information on new schema groups.

2 Go to File | New | Message Definition. The Add Message Schema dialog box opens.

3 In the Metadata Source drop-down list, use the default selection, DTD Importer. In the Files of Type drop-down list, use the default selection, Document Type Definition (DTD).

4 Enter the name of the DTD you want to import. You can change directories by clicking the Down arrow in the Look In drop-down list. Navigate to the directory where the DTD file resides and select the file.

5 Click OK. The Select Root Element dialog box opens.

Note If your DTD has namespaces, see Defining namespaces in message definitions that use DTDs. Elements with colons signify namespaces.

---

6 Select a root element from the DTD that corresponds to the document for this message definition. Click OK.

Note To find the root element of the DTD you are importing, review the DTD file you imported in a text editor.

The DTD you imported appears in the Definitions panel, and a graphical representation of the DTD appears in the Message Bridge GUI. The representation shows the element you selected as root and its children, as defined in the DTD.

You can now continue defining your message by:

- Defining namespaces in message definitions that use DTDs
- Modifying an element name
- Creating views

Adding a message definition for an XML Schema

You can add a message definition by importing an XML Schema. When you add a message definition, Message Bridge binds the root element to the XML Schema. To add a message definition, you must know the root elements of the XML Schema you want to import.

❖ **To add a message definition for XML Schema**

1 In the Definitions panel, select a Schema Group. Select Default Group if no others exist. See Schema groups for information on new schema groups.

2 Go to File | New | Message Definition. The Add Message Schema dialog box opens.

3 In the Metadata Source drop-down list, select XML Schema Importer. In the Files of Type drop-down list, XML Schema (*.xsd*) becomes the default.

4 Enter the name of the XML Schema you want to import. You can change directories by clicking the Down arrow in the Look In drop-down list. Navigate to the directory where the XML Schema resides and select it.

5 Click OK. The Select Root Element dialog box opens.

6 Select a root element from the XML Schema that corresponds to the document for this message definition. Click OK.

Note To find the root element of the XML Schema you are importing, review the XML Schema file you imported in a text editor.

The XML Schema you imported appears in the Definitions panel, and a graphical representation of the XML Schema appears in the Message Bridge GUI. The representation shows the element you selected as the root and its children, as defined in the XML Schema.

You can now continue defining your message by:

- Defining namespaces in message definitions that use DTDs
- Modifying an element name
- Creating views

Defining namespaces in message definitions that use DTDs

If you are using a DTD with namespaces, you must register the namespace Uniform Resource Identifier (URI) in the Message Bridge GUI. In order to generate Java bound to your message definition, the URIs must be registered. Since XML Schemas support a namespace declaration, their URIs are automatically registered in the GUI.

You have to determine which elements and attributes have namespaces. Look at the DTD or go to the Select Root Element dialog box. You also have to know the URI of each namespace. When you import your DTD, Message Bridge will assume that every colon represents a namespace.

❖ **To register namespaces in DTDs**

1 Elements with namespaces appear in the Select Root Element dialog box with colons. The Select Root Element dialog box opens when you import your DTD. See Adding a message definition for a DTD to import a DTD. Note the elements that have colons; you are required to register their URIs.

2 Find an element or attribute with a namespace in the graphical representation of the DTD and select it.

3 In the Properties panel, below the Definitions panel, the namespace prefix appears in the Value field. Enter the URI in the Value field of Namespace URI, for example, `sybase=http://sybase.com/sybase`.

The URI is registered for all instances of the namespace when you register it for one.

Note Because namespaces are not supported in DTDs, you have to get the URI from the author of the DTD. If you have an XML document that conforms to the DTD, the URI is in the document.

---

4 Repeat steps 2 and 3 for all namespaces.

Modifying an element name

Because the application you are writing may have local naming conventions or styles to conform with, Message Bridge allows you to change the name of an element that occurs in the code it generates. You change the name in the graphical representation of your schema. The name of the element does not change in the schema, only in the GUI. Your modified name occurs in the DataBean.

❖ **To modify an element name**

1 Select the element whose name you want to modify.

2 Right-click the element name in the graphical representation of the schema.

Note If you right-click an attribute and select Rename Element, you rename the element above the attribute. Renaming attributes is not supported.

3 Select Rename Element. The Rename Object dialog box opens.

4 Enter the new name of the element and click OK. The new name appears in the graphical representation of the schema.

Creating views

You can create a view on one of your message definitions and create a subset of the schema. A graphical representation of the view is displayed in the GUI.

❖ **To create a view**

1 Open the message definition from which you want to create a view.

2 Right-click an element that is part of the view you want to create. Message Bridge automatically includes all the parents and children of the element you select.

3 Select New View. The View Name dialog box opens.

4 Enter a name for your view. Click OK. A graphical representation of your view appears.

Code generation

Message Bridge generates DataBeans for your message definition. See Chapter 2, “Understanding the DataBean Framework” for more information on DataBeans.

❖ **To generate code from your message definitions**

1. In the Definitions panel, select the message definition for which you want to generate code.
2. Go to Tools | Generate.
3. If you have not saved your message definition yet, you are prompted to do so. Save your message definition and click Yes. The Generate dialog box opens.
4. If you want to create Javadoc, click the Create Documentation box.

   This tells Message Bridge to create HTML comments (javadoc) for all the DataBean classes it generates for this message definition.
5. Enter a package name in the Package Name field.

   This identifies the package name used in all the DataBean classes that Message Bridge generates for this message definition.
6. Using the browse button, set the directory to where you want Message Bridge to generate code.

   This is where Message Bridge places the DataBean classes and other developer artifacts.
7. Accept the default *.jar* file name.

   This identifies the *.jar* file into which all of the DataBean classes will be placed. If you are using XSD files, *xsd* is automatically appended to the *.jar* file name.
8. Click OK to generate the DataBean classes for your message definition.

CHAPTER 4

Using Views

This chapter describes views and how developers can use them during development. It covers the following topics:

- The purpose of views
- An example of using a view

The purpose of views

Because XML documents can contain multiple messages and they can have very deep hierarchies, Message Bridge allows developers to create a *view* into an XML document. The view represents a subset of the XML document and that subset's schema information. Once a view is created, a developer can use it at runtime to manipulate a subset of a larger document without needing to understand or even to know about the content model of the parent document. A view can also be serialized and passed around as a smaller XML document for future merger with the parent document. Once the XML document and its content model have been divided into views, the work in the original document can be distributed.

Note These smaller XML documents have the view root as their root.

An example of using a view

You have an XML document and want to receive part of the information for a single large message in a database and part of it in a Java Server Page (JSP). You supply the developers for each group—the database group and the Web group—with their own view of the information that they need to retrieve. This allows each group to concentrate on the information that is pertinent to them, and to use tools to map directly to their view of the schema without dealing with a potentially large message. Later, each resulting view message can be merged back into the original message that will be sent to a target system.

To merge views back into the original message you use the generated class com.sybase.DataBean.util.ViewSet. It takes XML documents that represent views and collects them back into a single XML document. Each of these documents can be added to a ViewSet and then serialized as a single XML document.

In the following example, at design time a developer creates a view on the ingredient element of *recipe.dtd*. This allows the developer to concentrate on a recipe's ingredient element without needing to understand the entire content model of *recipe.dtd*.

Without using a view, to find an ingredient you use the following:

```
Ingredient ing = recipe.getIngredientList().getIngredient();
```

Using a view to find an ingredient, you use the following:

```
ingredient ing = ingView.getIngredient(i);
```

This particular implementation of a view identifies all ingredients of a recipe that contain eggs.

```
public class IngSample

{

    public static void main(String[] args) throws Exception

    {

    System.out.println( "\n...Processing [" + args[0] + "]..."
   );

          // Create the request view document, and set the input
          XML as the main document.

          Recipe recipe = new Recipe(new FileInputStream(
         args[0]),  true);
```

```
IngView ingView = new IngView(new FileInputStream( args[0]),
true);

          IngView ingView = new IngView();

         ingView.setMainDocument( recipe );

          int ingCount = ingView.getSize();

          // Print list of all ingredient names that contain a
           Food item of "egg".

         System.out.println( "Recipe [" +
         recipe.getHeader().getRecipeName().getData() + "] has
         the following ingredients with eggs:" );

          for(int i=0; i<ingCount; i++ )

          {

          String content = ingView.getIngredient(i).getFood().
          getData();

          if( content.startsWith("egg") || content.
          endsWith("eggs") )

            {

          System.out.println( content );

              }

          }

     }

}
```

CHAPTER 5

Locating the Java documentation

To see DataBean-related methods that are used by the generated code, see the Java documentation at:
<Message Bridge Installation directory>/docs/javadoc.

CHAPTER 6

Advanced Topics

This chapter provides information about the following advanced topics:

- DataBeanOpaque
- Deserialization
- Serialization
- Validation
- Directory structure of generated code
- Runtime location of schemas for validation

DataBeanOpaque

The binding framework is implemented in the class called DataBeanOpaque, which represents an XML document. All generated root classes extend class DataBeanOpaque. In addition to providing all methods for serializing and deserializing XML, DataBeanOpaque also provides methods that expose access to the root name and doc type of a particular document instance.

Individual DataBeans use the functionality of the binding framework by extending the class DataBeanOpaque.

Deserialization

The state of DataBeans can be set automatically with the data in an XML instance document. This process is called *deserialization*. After deserializing, you use accessors and mutators to manipulate the XML document data from Java code.

There are different ways to deserialize XML documents into DataBean instances. The most common way is through the following constructor of a DataBean instance:

```
RootName(InputStream istrm, boolean validate)
```

Another way to deserialize XML is by calling the following method on an existing DataBean instance:

```
void deserializeXML(java.io.InputStream istrm, boolean
validate)
```

A third way to deserialize XML into a DataBean instance allows the user to do a *late binding* without incurring the overhead of parsing an XML document twice. This approach is helpful when you require some information from the document instance before determining which particular DataBean instance to bind.

To use this approach, use one of the previous methods to deserialize XML into a DataBeanOpaque object. This object provides methods to retrieve the root name, *SYSTEM URI*, and *PUBLIC URI* from the XML document instance. Then, using this information, you can bind a specific DataBean instance one of two ways:

- By passing the DataBeanOpaque object to its constructor
- By creating a DataBean instance with the default constructor and passing the DataBeanOpaque object to its `setMainDocument()` method.

The XML document is parsed only once with this approach.

Serialization

Serialization allows you to get an XML document directly from the DataBean and use it for your purposes. The data structure from the root class is exported into an XML document through the following method:

```
void serializeXML(java.io.OutputStream ostrm)
```

Validation

Validation is available through the root class, at the XML-document level. There are two ways to perform a validation. In the first, you use the following constructor:

```
RootName(InputStream istrm, boolean validate)
```

If the input XML document does not use a DTD attached and `validate = true`, an exception is thrown.

Only one exception is thrown in an exception case:

```
DataBeanException
```

You can use `get Message()` to get the error information. The error message is usually composed of the class name and function name where the error occurred, as well as a detailed error message.

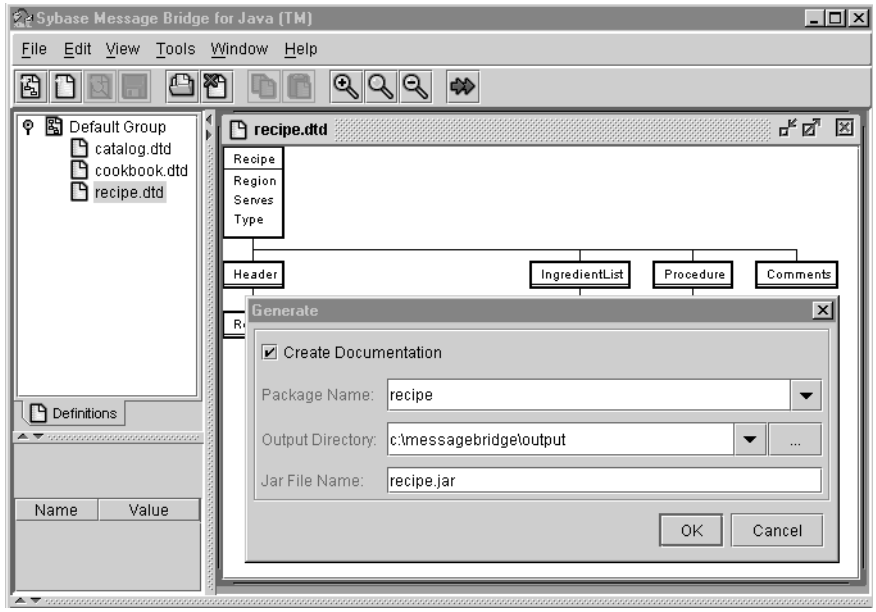
The second way to perform validation is through a method on the DataBean:

```
void  iRecipe.validate()
```

Directory structure of generated code

As shown in Figure 6-1, Message Bridge creates a directory tree in the Output Directory specified in the Generate dialog box.

Figure 6-1: Generate dialog box



In the case of an individual schema, the code generation process creates three subdirectories:

- *docs*

- *lib*
- *schemas*

The *docs* directory is created only if you selected the Create Documentation option in the Generate dialog box. Within the *docs* directory, another subdirectory corresponding to the schema name is created. This contains the output of the JavaDoc command, including HTML and CSS files, as well as deeper subdirectories. The exact contents will depend on the schema for which documentation is being generated and the top-level package name that was supplied in the Generate dialog box.

The *lib* directory contains a *.jar* file whose name is the same as the schema name. This *.jar* file contains *.class* files which comprise the schema-specific portion of the Message Bridge runtime.

The *schemas* directory contains both a DTD and an XML Schema representation of the original schema, when importing a DTD.

Note that it is possible to repeat the process of selecting an individual schema and generating code, while reusing the previously-specified output directory. The result will be multiple *.jar* files in the *lib* directory, and multiple schema and DTD files in the *schemas* directory. Also, the *docs* directory will get a subdirectory for each such schema.

Runtime location of schemas for validation

When creating instances of the generated classes from an XML input document, you can direct the binding framework to do validation. This requires that the XML document contains a declaration that identifies which schema to use for validation. For example, a `DOCTYPE` for DTDs or a `xsi:noNamespaceSchemaLocation` for XSDs. The runtime DataBean framework uses the following rules to locate the DTD.

❖ **If the SYSTEM keyword is specified:**

1 If the SYSTEM keyword specifies an absolute name, the framework tries to use the absolute path specified by the SYSTEM keyword.

2 If the SYSTEM keyword specifies a relative name, the XML parser's default location is used (usually the "current directory").

3 The framework parses the SYSTEM keyword to find the DTD name (removing any path information, if found), and tries to locate the DTD in one of the directories specified by the System property *com.sybase.DataBean.EntityResolver.dtds*, which can contain a semicolon-separated list of directories that will be successively searched for the DTD.

❖ **If the PUBLIC keyword is specified:**

1 If the PUBLIC keyword specifies an absolute name, the framework tries to use the absolute path specified by the PUBLIC keyword.

2 If the PUBLIC keyword specifies a relative name, the XML parser's default location is used (usually the "current directory").

3 The framework parses the PUBLIC keyword to find the DTD name (removing any path information, if found), and tries to locate the DTD in one of the directories specified by the System property *com.sybase.DataBean.EntityResolver.dtds*, which contains a semicolon-separated list of directories that will be successively searched for the DTD.

Most of the time, the name of the DTD will be specified in the DOCTYPE using the SYSTEM keyword. Again, most of the time, this name will be a "relative" name. For example, the following XML fragment indicates that the DTD name is *recipe.dtd*.

```
<!DOCTYPE recipe SYSTEM "recipe.dtd">

<recipe serves="10" region="n-america" type="snacks">

.

.

</recipe>
```

To ensure that the runtime framework can locate the DTD for validation, Sybase recommends that you put your DTDs in a directory (or directories) on the file system, and identify those directories through the *System.property*. This can be done in one of the following ways:

- Use the –D parameter of the java command when starting your application. For example:

  ```
  java -D com.sybase.DataBean.EntityResolver.dtds=
     /software/myapp/dtds
  ```

- Add the system property programmatically. For example, in your application's start-up code, do something like the following:

```
System.setProperty("com.sybase.DataBean.
   EntityResolver.dtds",
   "/software/myapp/dtds");
```

CHAPTER 7

Using Message Bridge Samples

The samples provided with Sybase Message Bridge for Java™ are intended to demonstrate the use of code generated from the GUI. For an explanation of data binding that happens at design time, see Chapter 2, “Understanding the DataBean Framework”.

This chapter covers the following topics about samples:

- Sample descriptions and locations
- Sample directory content
- Running a sample
- Understanding the output

Sample descriptions and locations

Table 7-1 lists the samples and their descriptions. *%MB%* represents the installation directory of Sybase Message Bridge for Java™.

Table 7-1: Sample descriptions and locations

Name	Description	Location
Any Content	Demonstrates the generated code from a DTD with elements that have type ANY as the content	*%MB%\samples\dtd\any_content*
Attributes (DTD)	Demonstrates the generated code from a DTD with an element that has attributes. See also the Attributes (XML Schema) example.	*%MB%\samples\dtd\attributes*
Attributes (XML Schema)	Demonstrates the generated code from an XSD with an element that has attributes. See also the Attributes (DTD) example.	*%MB%\samples\xsd\attributes*

Name	Description	Location
Bookstore	Demonstrates the following DTD-related features: • use of a PUBLIC identifier in a DTD to select generated code for execution • a parameter entity • INCLUDE and IGNORE directives to add or remove declarations • specification of directory paths in which to look for DTD files	*%MB%\samples\dtd\bookstore*
Build XML	This example uses the generated code associated with a DTD to construct a new XML document instance from scratch.	*%MB%\samples\misc\build_xml*
Datatypes	This example shows how the generated code represents (and provides access to) primitive data types in XML Schema.	*%MB%\samples\xsd\datatypes*
Mixed Content	Demonstrates the generated code from a DTD which specifies a sequence of elements and character data.	*%MB%\samples\dtd\mixed_content*
Recursive	Demonstrates the generated code from a DTD in which an element is nested recursively.	*%MB%\samples\dtd\recursive*
SOAP Envelope	Demonstrates parsing and navigation of data carried in a SOAP message.	*%MB%\samples\xsd\soap*
Stock Portfolio	Demonstrates the following XSD-related features: • XSD document-level attributes • Namespaces • INCLUDE and IMPORT directives • Data type definition via restriction	*%MB%\samples\xsd\stock_portfolio*
View	Demonstrates the generated code from a view.	*%MB%\samples\dtd\view*

Sample directory content

The following table outlines the contents of a sample directory. It is specific to the "Elements with Character" sample directory, and its layout it is very similar to all the other sample directory layouts.

Table 7-2: Sample directory contents

Directory name	Contents description
docs	The *docs* directory contains Javadoc of the generated source code.
lib	The *lib* directory the client class file and also contains a jar file that contains the generated classes.
schemas	The *schemas* directory contains the generated DTD and *XSD* files that were produced along with the generated code. This directory only exists for samples based on DTD files, not *XSD* files.
mixed_content.dtd	This is the DTD that was used when generating classes.
mixed_content.xml	This is the XML that was loaded by *client.java* through the generated code.
client.java	This Java file is the source for the client. Each client binds the XML document to the generated Java classes and then gets the value, or values that were set during the binding. The original XML, the result of the binding, expected result of the binding, and a generated XML document are all displayed on the screen.
run.bat, run.sh	*run* files run the client against the generated code.

Running a sample

Each sample directory contains a *run.bat* file if you are running Message Bridge on Windows platforms, and a *run.sh* file if you are using the Sun Solaris platform.

❖ **To run a sample:**

1 Navigate to the directory of the sample you want to run.

For Windows:

```
cd %MB%\samples\<Sample Name>
```

For Sun Solaris:

```
cd $MB/samples/<Sample Name>
```

Note *%MB%* and *$MB* represent the location of the directory in which you installed Message Bridge.

2 Type the appropriate command, based on which operating system you are running.

For Windows, type:

```
run.bat
```

For Sun Solaris, type:

```
run.sh
```

Understanding the output

After you have run a sample, you will see output on the screen. The following example is output after running the "Element with Character" sample. (Output from other samples will be similar to this XML document.)

```
1. <?xml version="1.0" encoding="UTF-8"?>

2. <!-- element containing only character data -->

3. <!DOCTYPE checkbook SYSTEM "element_character.dtd">

4. <checkbook>Deposit $2,000,000 into my
checkbook!</checkbook>

5. payment amount = <Deposit $2,000,000 into my
checkbook!>

6. payment expected amount = <Deposit $2,000,000 into
my checkbook!>

7. ...Building XML...

8. <?xml version="1.0" encoding="UTF-8"?>

9. <!DOCTYPE checkbook SYSTEM
"element_character.dtd"><!-- element containing only
character data --><checkbook>Deposit $2,000,000 into my
checkbook!</checkbook>
```

The first piece of output to appear on the screen is the original XML document, lines 1 through 4. The second piece of output is the result of the binding, line 5. The third piece of output is the expected output of the binding, line 6, which should be exactly the same as line 5. The rest of the output, lines 7 through 9, is the generated XML after the binding, which should be very similar to the original XML document.

Most of the samples' output is similar to the previous description, with a few differences. To validate the output you see on the screen, compare it with the data in the *output.txt* file in the sample's directory.

CHAPTER 8

Supported Elements and Declarations for Schemas

The following tables outline how Message Bridge supports both Document Type Definitions (DTD) and XML Schema (XSD). Each of the declarations and elements listed in these tables are supported, either fully or partially. Where there is not full support, the extent of the support is described in the Comments column.

Message Bridge is capable of importing DTDs and XSDs with these declarations and elements, and it generates the Java code necessary for implementing each of the features in Message Bridge DataBeans.

The following table shows the elements for which Message Bridge provides either full or partial support.

Note Elements that are part of the W3C XML Schema structure that are not listed in the following table are not supported in this release.

Table 8-1: Supported elements for XML Schemas (XSDs)

Elements	Can include	Comments
all		
	annotation	
	element	
any		Its attributes are currently not used for code generation.
attribute		
	simpleType	
attributeGroup		
	attribute	
	attributeGroup	
choice		xsd:any currently only shows by itself under xsd:choice.

Elements	Can include	Comments
	any	Does not support multiple any or any mixed with elements.
	choice	
	element	
	group	
	sequence	
complexContent		
	extension	
	restriction	xsd:restriction for xsd:complexContent should have all its base type members, plus the restriction.
complexType		Its attribute final, block, and abstract are currently not used for code generation.
	all	
	attribute	
	attributeGroup	
	choice	
	complexContent	
	group	
	sequence	
	simpleContent	
element		final, block, and abstract attributes are currently not used for code generation.
	complexType	
	key	
	keyref	
	simpleType	
	unique	
enumeration		
extension		
	attribute	
	attributeGroup	
field		
group		
	all	
	choice	
	sequence	

Elements	Can include	Comments
key		
	field	
	selector	
keyref		
	field	
	selector	
restriction		
	enumeration	Used for XML document validation.
	fractionDigits	
	length	
	maxExclusive	
	maxInclusive	
	maxLength	
	minExclusive	
	minInclusive	
	minLength	
	pattern	
	simpleType	
	totalDigits	
	whiteSpace	
schema		Currently, blockDefault, finalDefault, version, xml:lang attributes and esd:include, xsd:import, and xsd:redefine content are not supported.
	attribute	
	attributeGroup	
	complexType	
	element	
	group	
	notation	
	simpleType	
selector		
	annotation	
sequence		xsd:any only shows by itself under xsd:choice.

Elements	Can include	Comments
	any	
	choice	
	element	
	group	
	sequence	
simpleContent		
	extension	
	restriction	
simpleType		Currently, xsd:list and xsd:union content are not supported.
	restriction	
unique		
	field	
	selector	

The following table shows the declarations for which Message Bridge provides either full or partial support.

Table 8-2: Supported declarations for Document Type Definitions (DTDs)

Declaration	Can include	Comments
DOCTYPE		
ELEMENT		
	, (comma)	Sequence of elements, AND operator
	| (bar)	OR operator
	() (parenthesis)	Content grouping
	? (question mark)	Optional, 0 or 1 occurances
	+ (plus)	at least one, 1 or more occurrences
	* (asterisk)	any number, 0 or more occurrences
	EMPTY	
	ANY	
	PCDATA	

Declaration	Can include	Comments
ATTLIST		
	CDATA	
	NMTOKEN	
	NMTOKENS	Validation is using attached DTD.
	ID	Validation is using attached DTD.
	IDREF	Validation is using attached DTD.
	IDREFS	Validation is using attached DTD.
	ENTITY	Validation is using attached DTD.
	ENTITIES	Validation is using attached DTD.
	Enumerated value list	Validation is using attached DTD.
	NOTATION	Validation is using attached DTD.
	DEFAULT or default value assigned	
	IMPLIED	
	REQUIRED	
	FIXED	
NOTATION		
ENTITY		
	parameter entity	
	internal entity	
	SYSTEM ID (External parsed entity)	
	PUBLIC ID (External parsed entity)	Handled the same as SYSTEM ID.
	SYSTEM ID (External unparsed entity)	Passed on to DTD generation.
	PUBLIC ID (External unparsed entity)	Passed on to DTD generation.

Index

M

N

R

S

U

V

X