IBM® Workplace Forms™

Version 2.6.1

Using the XML Data Model

S325-2608-00

Note
Before using this information and the product it supports, read the information in "Notices," on page 51.

First Edition (September 2006)

This edition applies to version 2.6.1 of IBM Workplace Forms and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Introduction

The XML Data Model makes it easier to integrate XFDL forms with other applications by:

- Separating the form data into a separate block of XML. This makes the data easy to locate and parse within the form, while also allowing the data to conform to any valid XML structure, such as that dictated by a schema.
- Enabling schema validation of data. This ensures that all data collected adheres to a defined schema, thereby reducing input errors.

Additionally, you can use the XML Data Model to enable Smartfill functionality in the Viewer, which can automatically complete portions of the form for the user.

This document explains what the XML Data Model does, and provides practical instructions for using the data model.

Who Should Read this Document

This document is written for system integrators who want to use the XML Data Model. This document assumes that the reader has a working knowledge of XML and XFDL, as well as some programming experience and some familiarity with XML Namespace.

About the XML Data Model

Before discussing the XML Data Model, it's useful to review the structure of XFDL as well as some of the goals of XML.

XFDL: Combining Presentation and Data

XFDL was engineered to combine a form's presentation and data. For example, the following *field* item embeds the user's name, Tom Jones, in the *value* option:

```
<field sid="nameField">
   <value>Tom Jones</value>
</field>
```

By embedding the data in the form description, XFDL offers a number of strengths, including non-repudiation and the ability to save the form to a single file.

However, embedded data can be inconvenient when integrating with other applications. For example, typical XFDL integrations are built by developing a module that reformats the data as it is passed back and forth. Unfortunately, this may require significant custom programming for every integration.

XML: A Common Language for Interoperability

One of the goals of XML is to enable interoperation. Simply put, this means making it easier for applications to work together.

For example, consider an application that processes purchase orders. A typical PO system would receive a purchase order, then spend significant effort parsing the

PO and extracting the data before it began processing. In fact, extracting the data from the PO is often a complicated process that requires a good deal of custom programming.

However, with XML the PO application can be designed to accept a set of XML data that is defined by a schema. Furthermore, if the PO is XML-based, it can be designed to encapsulate this data in a single block of XML that conforms to the schema. This makes the submission process easier, since the PO system can quickly retrieve the block of XML from the form and begin processing almost immediately. In fact, extracting the data from the form can be as simple as a single line command, which greatly reduces the scope of work necessary for integration.

The XML Data Model: Grouping Data for Interoperability

Although XFDL is an XML syntax, it has not provided the ability to easily group data into blocks that would support interoperability. The XML Data Model addresses this problem by allowing form developers to create separate data sets within an XFDL form and to share data between those data sets and regular form elements.

Essentially, the XML Data Model is a block of XML that is placed at the beginning of a form, within the global page's global item, as shown:

```
<globalpage sid="global">
   <global sid="global">
      <xmlmodel>
         ... XML Data Model ...
      </xmlmodel>
   </global>
</globalpage>
```

This block of XML allows for *arbitrary* data, meaning that it can contain any data and can be formatted in any manner. Furthermore, individual elements in the data model can be *bound* to one or more elements in the form description. This *binding* causes the elements to share data. If one element is changed, the other elements are updated to mirror that change.

This allows you to create a separate block of data within the form, format it any way you like, and bind it to form elements so that data entered by the user is automatically copied to the data model. For example, you could include the block of data that is required by an application (such as a PO system), format the data so that it complies with a specific schema, and then bind that data model to the form description.

The result is a block of XML data that can be structured to meet any needs, extracted easily by other applications, and transmitted without the rest of the form.

Validating the XML Data Model Against Schema

Once you have created an XML Data Model, you can associate that model with one or more schema. This allows you to validate the data in the model against any XML schema. Furthermore, you can embed the schema in the form itself, or link to an external schema file.

Smartfill and the XML Data Model

The Viewer's Smartfill feature helps user to complete forms by automatically completing certain sections of the form, such as address information. This feature is enabled through the XML Data Model, which defines the Smartfill information and links it to the body of the form.

While working with a Smartfill enabled form, Smartfill data is also stored on the user's computer as *data fragments*. These fragments contain the information that the user has entered in the past, and are automatically loaded into forms that make use of the Smartfill feature.

When to Use the XML Data Model

You can use the XML Data Model in any of the following scenarios:

XML Applications — The XML Data Model is most useful when integrating eforms with applications that already use XML, especially if those applications already offer XML interfaces. In these cases, you can design forms that will submit the XML data directly to the application, and will not need to program a custom module that extracts the data from the form. Furthermore, you can format the data to match any schema, and validate the data against the schema before submission.

Non-XML Applications — Even if an application does not use XML, you can still benefit from using the XML Data Model. The data model simplifies copying information from one page to another, making wizard-style forms easier to create and manage. Furthermore, although custom programming is still required for back-end processing, the data model makes it far easier to extract data from the form.

Automatic Form Completion — If a form requires users to repeatedly enter the same information, such as, their name and contact information, you can set up the form to use Smartfill. Smartfill can automate portions of form completion by capturing frequently used information and giving the user the option to automatically load that information while they are completing a form.

Overview of the XML Data Model

The XML Data Model serves several purposes. Its core function is to provide a way to achieve interoperability with other applications. In addition, it provides schema validation capabilities, and allows you to enable the Viewer's Smartfill feature, which can automatically complete portions of the form for the user.

The Core of the XML Data Model

The XML Data Model contains three core parts that work together to create a complete model:

- **Data Instances** — Data instances are arbitrary blocks of XML. A data model may contain any number of data instances, and each instance is normally created to serve a particular purpose. For example, if your form provides data to both an accounting application and a shipping application, you may want to create two data instances - one for each application.
- **Bindings** — Each data instance has associated bindings. Bindings tie one element in the data instance to one or more elements in the form description. For example, if a form had a *firstName* field on both the first and second pages, you might bind the *firstName* element in your data instance to both fields. Once this is done, all three elements will share data, meaning that if one element is changed the other two elements are updated to mirror that change.
- **Submission Rules** — Each data instance may have an associated set of *submission rules*. These rules control how a data instance is transmitted when it is submitted for processing. This is an optional feature, and is only necessary when you want to submit the data instance by itself, without the rest of the form. There are many cases in which you may want to submit the entire form, and then retrieve the data instance from the form during processing. This is particularly true when you are using signatures on your forms.

Each of the three parts is contained by its own tags within the <xmlmodel> tag, which is itself contained by the global page's global item, as shown:

```
<globalpage sid="global">
   <global sid="global">
      <xmlmodel>
         <instances>
            ... all data instances ...
         </instances>
         <bindings>
            ... all bindings ...
         </bindings>
         <submissions>
            ... all submission rules ...
         </submissions>
      </xmlmodel>
   </global>
</globalpage>
```

Schema Validation in the XML Data Model

As we have seen, the core of the data model consists of data instances, their bindings and submission rules. In addition, the data model can reference one or more schemas, which allow you to validate data instances against them. A schema can be embedded in the form by including a <schemas> tag in the data model, as shown:

```
<globalpage sid="global">
   <global sid="global">
      <xmlmodel>
         <schemas>
            ...schemas...
         </schemas>
      </xmlmodel>
   </global>
</globalpage>
```

Data Fragment Definitions in the XML Data Model

A data fragment definition specifies what data from a specific data instance can be stored in a data fragment on a user's local computer. Data fragments are intended to simplify the completing of forms, by capturing information that is frequently entered on forms and storing it on the user's computer for future use. For complete information about this feature, see ″Enabling Smartfill″.

```
<globalpage sid="global">
   <global sid="global">
      <xmlmodel>
         <instances>
            ...all data instances....
         </instances
         <bindings
            ...all bindings....
         </bindings>
         <submissions>
            ...all submissions....
         </submissions>
         <datafragments>
            ...all datafragments...
         </datafragments>
      </xmlmodel>
   </global>
</globalpage>
```

Creating An XML Data Model

When creating an XML Data Model, it's a good idea to create your data instances one at a time, and to set up the bindings and submission rules for that instance before moving on to the next data instance. This helps to avoid confusion.

To create an XML Data Model, you must:

- Declare the XML Data Model in the form.
- Create a data instance.
- Bind the elements of the data instance.
- Set up submission rules for the instance (optional).
- Create a submission button for the instance (optional).

Declaring the XML Data Model

The data model is always declared as an option in the global item of a form's global page, and begins with the <xmlmodel> tag, as shown:

```
<globalpage sid="global">
   <global sid="global">
      <xmlmodel>
      </xmlmodel>
   </global>
</globalpage>
```

The <xmlmodel> tag normally includes a definition of the XForms namespace. This definition is necessary because most data instances begin with a tag in the XForms namespace, as you will see later in this document.

To define a namespace, you use the *xmlns* attribute. This attribute assigns the unique URI for the namespace to a prefix, as shown:

```
xmlns:prefix="namespace URI"
```

By convention, the XForms prefix is *xforms*, and the XForms namespace is defined by the following URI:

```
http://www.w3.org/2003/xforms
```

Substituting these values, you get the following *xmlns* attribute:

```
xmlns:xforms="http://www.w3.org/2003/xforms"
```

This attribute is added to the <xmlmodel> tag, as shown:

```
<globalpage sid="global">
   <global sid="global">
      <xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms">
      </xmlmodel>
   </global>
</globalpage>
```

Once you have declared the data model in your form, you can add data instances, bindings, and submission rules.

Creating a Data Instance

Each data instance is inserted within an <instances> tag in the XML model, as shown:

```
<xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms">
   <instances>
      ... all data instances ...
   </instances>
</xmlmodel>
```

Each instance is created within an arbitrary tag. This tag is simply a placeholder for the data instance, but may also provide meaning in other contexts. For example, in this case we'll use an <xforms:instance> tag. This tag indicates that the data instance conforms to the XForms definition of a data instance. The following example shows a data model with two data instances:

```
<xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms">
   <instances>
      <xforms:instance>
         ... data instance 1 ...
      </xforms:instance>
      <xforms:instance>
         ... data instance 2 ...
      </xforms:instance>
   </instances>
</xmlmodel>
```

Each <xforms:instance> tag contains a data instance. Each data instance must be well-formed XML, meaning that it must have a single root element. You should give the root element a meaningful name that reflects the content of the instance. For example, you might use a <customerData> element to begin an instance that contains customer data, as shown:

```
<xforms:instance>
   <customerData>
      ... customer data ...
   </customerData>
<xforms:instance>
```

Your data instance can contain any valid XML. For example, for customer data you might include the customer's first name, last name, and address. In this case, your data instance might look like this:

```
<xforms:instance>
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <address></address>
   </customerData>
</xforms:instance>
```

Instance data is not processed by XFDL parsers, such as Workplace Forms™ Viewer and Designer, and can follow any format necessary. This gives you the freedom to create data models that match defined schemas or other formats. However, this also means that computes do not work when placed within a data instance (for more information, see "Using Computes with the XML Data Model").

Naming a Data Instance

If you have more than one data instance in your form, you must name each instance. You can do this by adding an *id* attribute to the <xforms:instance> tag of your data instance. The *id* attribute follows this format:

```
id="name"
```

For example, if you wanted give the name *customer* to the customer data instance, you would use the following tag to begin the data instance:

```
<xforms:instance id="customer">
```

Defining Namespaces for an Instance

You can also use the <xforms:instance> tag to define:

- The default namespace for the data instance.
- Any other namespace prefixes you want to use in the instance.

For example, if you were creating an XBRL instance, you might set the default namespace of the data instance to match the XBRL namespace.

Defining the Default Namespace

To define the default namespace, you must add an *xmlns* attribute to the <xforms:instance> tag. The *xmlns* attribute is assigned the URI that defines the namespace, as shown:

```
xmlns="namespace URI"
```

For example, if you wanted to place an instance in your company's Human Resources namespace, the <xforms:instance> tag of our customer data might look like this:

```
<xforms:instance xmlns="http://www.mycompany.com/namespaces/HR">
```

When you set the default namespace for an element, both the opening element and all children of that element are placed in that namespace.

Defining and Using Other Namespaces

You may also want to create a namespace that you use selectively. For example, you might have a data instance that should be in your company's general namespace, except for two elements that should be in the Human Resources namespace. In this case, you would assign the Human Resources namespace to a prefix, and then use that prefix to tag specific data elements.

When defining other namespaces to use, it's best to declare them on the <xmlmodel> tag. This makes them available to the entire data model. You use the *xmlns* attribute to assign the unique URI for the namespace to a namespace prefix, as shown:

```
xmlns:prefix="namespace URI"
```

For example, the following tag creates an *hr* prefix for a Human Resources namespace:

```
<xmlmodel xmlns:hr="http://www.mycompany.com/namespaces/HR">
```

You can now add the prefix to any tag within the data instance to indicate that the tag belongs to the Human Resources namespace, as shown:

```
prefix:tag
```

For example, if you wanted the first name and last name in our customer data to belong to the Human Resources namespace, you would write:

```
<purchaseOrderData>
   <hr:firstName></hr:firstName>
   <hr:lastName></hr:lastName>
   <address></address>
</purchaseOrderData>
```

In this case, both the first and last name are in the Human Resources namespace, but the street is not since it has no prefix. Also, notice that each closing tag must also include the prefix.

Binding the Elements of a Data Instance

Setting the First Element to Bind

The first bound element must be part of a data instance, and is identified by enclosing a reference to it in a <ref> tag, as shown:

```
<bind>
   <ref>reference</ref>
</bind>
```

The reference to the element is written as a standard array reference that starts at the <xforms:instance> tag. In this case, array references use brackets to enclose each level of depth. For example:

```
[Level1][Level2][Level3]
```

Furthermore, the brackets can contain either a zero-based index to the element, or the name of the element's tag if it is unique within the scope of its parent.

Consider the following data instance for a customer:

```
<xforms:instance>
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <address>      </address>
   </customerData>
</xforms:instance>
```

To refer to the *customerData* element, you could use either of these references:

```
[customerData]
[0]
```

To refer to elements at a greater depth, such as the *address* element, you simply add the next level of depth, as shown in the following references:

```
[customerData][address]
[0][2]
```

Once you have determined the correct reference, include it in the <ref> tag as shown:

```
<bind>
   <ref>[customerData][address]</ref>
</bind>
```

Referencing an Attribute in the Data Instance

In some cases, you may need to reference an attribute in the data instance rather than an element. To do this, use the following notation in your reference:

```
[element]@attribute
```

For example, consider the following data instance:

```
<xforms:instance>
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <address street="" city="" country=""></address>
   </customerData>
</xforms:instance>
```

In this case, the *address* element stores the street, city, and country in attributes. To refer to the *street* attribute, you would use the following reference:

```
<bind>
   <ref>[customerData][address]@street</ref>
</bind>
```

Using Namespaces in Element References

References assume that you are working in the XFDL namespace. To refer to an element that is in another namespace you must include the appropriate namespace prefix in your reference, as shown:

```
[prefix:element]@prefix:attribute
```

An element might be in a non-XFDL namespace for two reasons. First, the default namespace of the data instance may not be XFDL. Second, the element may have a namespace prefix that places it in a non-XFDL namespace.

For example, in the following data instance a namespace prefix is used to place some of the elements in the Human Resources namespace:

```
<xforms:instance>
   <customerData>
      <hr:firstName></hr:firstName>
      <hr:lastName></hr:lastName>
      <address></address>
   </customerData>
</xforms:instance>
```

In this case, to refer to the *firstName* element, you would use the following reference:

```
[customerData][hr:firstName]
```

Alternately, you can choose to add a namespace and prefix to the form that is in the same namespace as the instances's default namespace. You can then use that prefix to reference those elements within the model as if they explicitly had that prefix. For example, you could add the following to your root node

```
xmlns:example="http://www.hr.com
```

Then, you could reference this namespace from your instance data:

```
<xforms:instance>
   <customerData xmlns="http://www.hr.com>
      <firstName></firstName>
      <lastName></lastName>
      <address></address>
   </customerData>
</xforms:instance>
```

You could then still reference the nodes in your instance using the prefix, even though the prefix doesn't appear with the nodes:

```
global.global.xmlmodel[instances][0][hr:customerData][hr:firstName]
```

Setting Which Data Instance Contains the First Element

By default, a bind assumes that the first bound element is part of the first data instance in your form. However, if you have more than one data instance, you must declare which data instance contains the bound element. To do this, enclose the name of the data instance in an <instanceid> tag, as shown:

```
<bind>
   <instanceid>name</instanceid>
</bind>
```

The name of the data instance is defined by the *id* attribute in the <xforms:instance> tag. For example, you might begin a customer data instance with the following tag:

```
<xforms:instance id="customer">
```

In this case, you would refer to the customer data instance as shown:

```
<bind>
   <instanceid>customer</instanceid>
</bind>
```

Setting the Second Element to Bind

The second bound element can be either part of the data model or an option in the form description. You define this element by enclosing a reference to it in a <boundoption> tag, as shown:

```
<bind>
   <boundoption>reference</boundoption>
</bind>
```

The reference is a standard XFDL reference, but is relative to the *boundoption* element. This means that you must ensure your reference provides enough information, such as the correct page, item, or option tag.

For example, consider the following form:

```
<page sid="Page1">
   <field sid="firstNameField">
      <value>Tom</value>
   </field>
</page>
```

To bind the *value* option of the field, you would use an absolute reference as shown:

```
<bind>
   <boundoption>Page1.firstNameField.value</boundoption>
</bind>
```

Next, consider the following data instance:

```
<globalpage sid="global>
   <global sid="global">
      <xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms">
         <xforms:instance>
            <customerData>
               <firstName></firstName>
               <lastName></lastName>
               <address></address>
            </customerData>
         </xforms:instance>
```

To bind the *firstName* element of the instance, you would use an option level reference, as shown:

```
<bind>
   <boundoption>xmlmodel[0][customerData][firstName]</boundoption>
</bind>
```

Note: Lists, popups, and radio buttons require special binding methods. For more information, see "Binding Lists, Popups, and Radio Buttons".

Using Namespaces in Option References

In some cases, you may need to reference elements in the form description that are not in the XFDL namespace. For example, you may use certain custom options to store data in your form, and these options may be in the *custom* namespace.

When referring to form elements that are not in the XFDL namespace, you must include the appropriate namespace prefix for each element in the reference. In a reference including the page, item, and option, you would also include the namespace prefix on option and array elements, as shown:

```
page.item.prefix:option[prefix:element]...
```

Notice that the page and item elements do not require a namespace prefix. This is because page and item reference refer to the *sid* of the element, not the local name.

For example, consider the following form:

```
<page sid="Page1">
   <field sid="firstNameField">
      <value>Tom</value>
      <custom:userNumber>22</custom:userNumber>
   </field>
</page>
```

Notice that the *userNumber* tag is at the option level, but is also in the *custom* namespace. To reference this option, you must include the namespace as shown:

```
Page1.firstNameField.custom:userNumber
```

Example of a Complete Bind

The following example shows a complete bind. In this case, the *firstName* element in the *customerData* data instance is bound to the *firstNameField* on the first page of the form:

```
<bind>
   <instanceid>customer</instanceid>
   <ref>[customerData][firstName]</ref>
   <boundoption>Page1.firstNameField.value</boundoption>
</bind>
```

Binding Computed Elements

In many cases, elements in your form will have computed values. For instance, a purchase order might have a "Total" field. The value of this field might be computed by adding the values in other fields.

In cases such as this, binding the computed value creates a one-way relationship. Data is copied from the computed value to the data instance, but not the other way around. This prevents computed values from being overwritten by inaccurate values in the data instance.

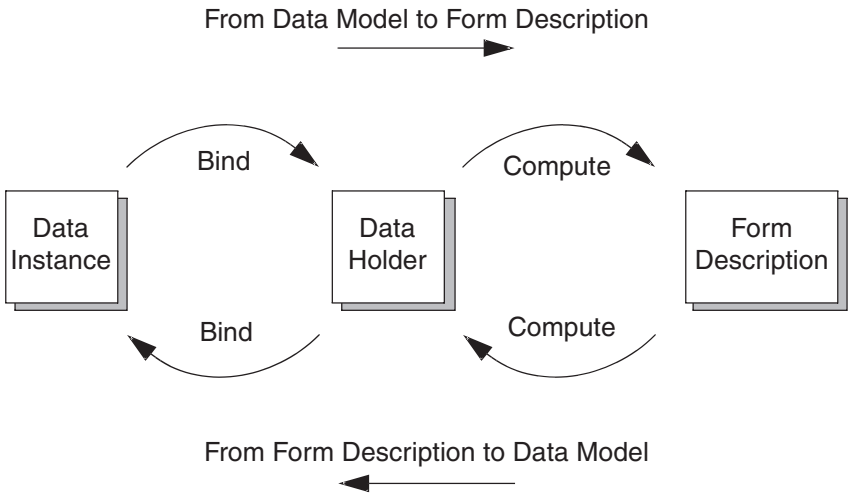
Binding Lists, Popups, and Radio Buttons

Lists, popups, and radio buttons are challenging to bind because they do not contain the value that you normally want.

For example, lists contain the name of the cell that was selected, but in most cases you want to know the value of the cell, not its name. If you bind to a list normally, you might get a value of *cell1* or *cell2* instead of *green* or *blue*.

Similarly, radio buttons contain an *on* or *off* value, but in most cases what you really want to know is which radio button is selected. Binding to radio buttons directly not only gives you an *on* or *off* value, as opposed to *greenRadio* or *blueRadio*, but also requires you to create extra elements in your data instance.

To solve this problem, you must add three custom elements to the form: a data holder and two toggle elements. The data holder is bound to the data instance, so that data is automatically copied between the data holder and the data instance. The two toggle elements contains computes that transfer the data between the data holder and the form description. This creates a three step process, in which data is copied from the data instance, to the data holder, and then to the form description, and vice versa:



This approach solves the problem in all cases. However, lists and popups require different computes than radio buttons.

Binding Lists and Popups

To bind to a list or popup, you must create three elements in your form:

- A data holder.
- A toggle element that copies data from the list or popup to the data holder.
- A toggle element that copies data from the data holder to the list or popup.

Creating a Data Holder

The data holder is created in the <bind> element, and can have any name you like. However, it cannot be in the *XFDL* namespace. You can use any namespace you want for the data holder - in this case, we'll use the *custom* namespace. For example, if you had a popup that listed all of the states, your data holder would hold the state the user chose. In this case, you might use the following tag:

```
<custom:state>
```

You must also bind the data holder to your data instance. For example, consider the following data instance:

```
<xforms:instance id="customer">
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <address>
         <street></street>
         <city></city>
         <state></state>
      </address>
   </customerData>
</xforms:instance>
```

In this case, you would bind the <custom:state> data holder to the <state> element in the data instance, as shown:

```
<bind>
   <instanceid>customer</instanceid>
   <ref>[customerData][address][state]</ref>
   <boundoption>..[custom:state]</boundoption>
   <custom:state></custom:state>
</bind>
```

Copying Data from the List or Popup to the Data Holder

To copy data from the list or popup to the data holder, you must create another element that contains a compute. This element can be in any namespace and have any name you like. In this case, we'll use the *custom* namespace and call the element *toggle1*, as shown:

```
<bind>
   <custom:toggle1></custom:toggle1>
</bind>
```

The *toggle1* element's compute is based on the *toggle* function, and follows this algorithm:

1. Begin processing when the value of the popup changes (detected with the toggle function).
2. Copy the value of the selected cell to the data holder (using the *set* function).

The compute looks like this:

```
toggle(reference to list's value option) == '1'
   ? set('reference to data holder', dereference to cell's value)
   : ''
```

For example, if you had a popup named *statePopup* on the first page of the form and your placeholder was named <custom:state>, your compute would look like this:

```
toggle(Page1.statePopup.value) == '1'
   ? set('..[custom:state]', Page1.statePopup.value->value)
   : ''
```

Placing this compute in a bind, it looks like this:

```
<bind>
   <custom:toggle1
      xfdl:compute="toggle(Page1.statePopup.value) == '1' &#xA;
      ? set('..[custom:state]', &#xA;
        Page1.statePopup.value->value) &#xA;
      : ''"></custom:toggle1>
</bind>
```

Copying Data from the Data Holder to the List or Popup

To copy data from the data holder to the list or popup, you must create another element that contains a compute. This element can be in any namespace and have any name you like. In this case, we'll use the *custom* namespace and call the element *toggle2*, as shown:

```
<bind>
   <custom:toggle2></custom:toggle2>
</bind>
```

The *toggle2* element's compute is based on the *toggle* function, and follows this algorithm:

1. Begin processing when the value of the data holder changes (detected with the *toggle* function).
2. Get the name of the cell that has a value equal to the data holder (using the *getGroupedItem* function).
3. Set the value of the list to the name of the cell.

The compute looks like this:

```
toggle(reference to data holder) == '1'
   ? set('reference to popup's value',
     getGroupedItem(reference to popup's group, 'value',
     reference to data holder,
     'reference to page containing popup', 'page', 'form'))
   : ''
```

For example, if you had a popup named *statePopup* on the first page of the form and your placeholder was named <custom:state>, your compute would look like this:

```
toggle(..[custom:state]) == '1'
   ? set('Page1.statePopup.value',
     getGroupedItem(Page1.statePopup.group, 'value',
     ..[custom:state], 'Page1', 'page', 'form'))
   : ''
```

Placing this compute in a bind, it looks like this:

```
<bind>
   <custom:toggle2
      xfdl:compute="toggle(..[custom:state]) == '1' &#xA;
      ? set('Page1.statePopup.value', &#xA;
        getGroupedItem(Page1.statePopup.group, 'value', &#xA;
        ..[custom:state], 'Page1', 'page', 'form'))   &#xA;
      : ''"></custom:toggle2>
</bind>
```

Example of a Complete Bind for Lists and Popups

Adding all of the elements together, including the data holder and toggle elements, a complete bind looks like this:

```
<bind>
   <instanceid>customer</instanceid>
   <ref>[customerData][address][state]</ref>
   <boundoption>..[custom:state]</boundoption>
   <custom:state></custom:state>
   <custom:toggle1
      xfdl:compute="toggle(Page1.statePopup.value) == '1' &#xA;
      ? set('..[custom:state]', &#xA;
        Page1.statePopup.value->value) &#xA;
      : ''"></custom:toggle1>
```

```
   <custom:toggle2
      xfdl:compute="toggle(..[custom:state]) == '1' &#xA;
      ? (..[custom:state) == '' &#xA;
         ? set('Page1.statePopup.value', &#xA;
            getGroupedItem(Page1.statePopup.group, 'value', &#xA;
        ..[custom:state], 'Page1', 'page', 'form'))   &#xA;
      : ''"></custom:toggle2>
</bind>
```

Remember that this example assumes you have popup item named *statePopup* and a corresponding element in your data instance named *state*.

Binding Radio Buttons

To bind to a group of radio buttons, you must create three elements in your form:

- A data holder.
- A toggle element that copies data from the radio buttons to the data holder.
- A toggle element that copies data from the data holder to the radio buttons.

Creating a Data Holder

The data holder is created in the <bind> element, and can have any name you like. However, it cannot be in the *XFDL* namespace. You can use any namespace you want for the data holder - in this case, we'll use the *custom* namespace. For example, if you had a group of radio buttons that selected the user's citizenship, your data holder would hold the name of the citizenship button the user selected. In this case, you might use the following tag:

```
<custom:citizenship>
```

You must also bind the data holder to your data instance. For example, consider the following data instance:

```
<xforms:instance id="customer">
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <citizenship></citizenship>
      <address>
         <street></street>
         <city></city>
         <state></state>
      </address>
   </customerData>
</xforms:instance>
```

In this case, you would bind the <custom:citizenship> data holder to the <citizenship> element in the data instance, as shown:

```
<bind>
   <instanceid>customer</instanceid>
   <ref>[customerData][citizenship]</ref>
   <boundoption>..[custom:citizenship]</boundoption>
   <custom:citizenship></custom:citizenship>
</bind>
```

Copying Radio Name from Radio Buttons to the Data Holder

To copy data from the radio buttons to the data holder, you must create another element that contains a compute. This element can be in any namespace and have any name you like. In this case, we'll use the *custom* namespace and call the element *toggle1*, as shown:

```
<bind>
   <custom:toggle1>      </custom:toggle1>
</bind>
```

The *toggle1* element's compute is based on the *toggle* function, and follows this algorithm:

1. Begin processing when any of the radio buttons change value (detected with *toggle* function).
2. Get the name of the radio button that is *on* (using the *getGroupedItem* function).
3. Set the name of the radio button to the data holder (using the *set* function).

Assuming that you had three radio buttons, the compute would look like this:

```
toggle(reference to first radio button's value, 'off', 'on') == '1' or
   toggle(reference to second radio button's value, 'off', 'on') == '1' or
   toggle(reference to third radio button's value, 'off', 'on') == '1'
   ? set('reference to data holder',
     getGroupedItem('reference to group', 'value', 'on',
     'reference to page containing group', 'page', 'page'))
   : ''
```

Notice that each radio button has a separate *toggle* function. This causes the compute to run when any of the radio buttons are changed. When writing this compute, you must add a *toggle* for every radio button in your group. You add each *toggle* with the *or* operator.

For example, if you had three radio buttons named *citizenRadio*, *landedImmigrantRadio*, and *otherRadio* in a group called *citizenship*, and your data holder was named <custom:citizenship>, your compute would look like this:

```
toggle(Page1.citizenRadio.value) == '1' or
   toggle(Page1.landedImmigrantRadio.value, 'off', 'on') == '1' or
   toggle(Page1.otherRadio.value, 'off', 'on') == '1'
   ? set('..[custom:citizenship]',
     getGroupedItem('Page1.citizenship', 'value', 'on',
     'Page1', 'page', 'page'))
   : ''
```

Placing this compute in a bind, it looks like this:

```
<bind>
   <custom:toggle1
      xfdl:compute="toggle(Page1.citizenRadio.value) == '1' &#xA;
      or toggle(Page1.landedImmigrantRadio.value, 'off', &#xA;
      'on') == '1' &#xA;
      or toggle(Page1.otherRadio.value, 'off', 'on') == '1' &#xA;
      ? set('..[custom:citizenship]', &#xA;
        getGroupedItem('Page1.citizenship', 'value', 'on', &#xA;
        'Page1', 'page', 'page')) &#xA;
      : ''"></custom:toggle1>
</bind>
```

Copying Data from the Data Holder to the Radio Buttons

To copy data from the data holder to the radio buttons, you must create another element that contains a compute. This element can be in any namespace and have any name you like. In this case, we'll use the *custom* namespace and call the element *toggle2*, as shown:

```
<bind>
   <custom:toggle2></custom:toggle2>
</bind>
```

The *toggle2* element's compute is based on the *toggle* function, and follows this algorithm:

1. Begin processing when the value of the data holder changes (detected with the *toggle* function).
2. Set all of the radio buttons in the group to off (using the *set* function).
3. Set the radio button named in the data holder to on (using the *set* function).

The actual compute looks like this:

```
toggle(reference to data holder) == '1'
   ? set('reference to value of first radio button', 'off') +.
     set('reference to value of second radio button', 'off') +.
     set('reference to value of third radio button', 'off') +.
     set('page reference.' +. reference to data holder +.
     '.value', 'on')
   : '' "></custom:toggle2>
```

Notice that each radio button has a corresponding *set* function that turns it off before the correct radio button is turned on. This ensures that two radio buttons are never turned on at the same time. When writing this compute, you must add a *set* for every radio button in your group, adding each *set* with the concatenation operator (+.).

For example, if you had three radio buttons named *citizenRadio*, *landedImmigrantRadio*, and *otherRadio* in a group, and your data holder was named <custom:citizenship>, your compute would look like this:

```
toggle(..[custom:citizenship]) == '1'
   ? set('Page1.citizenRadio.value', 'off') +.
     set('Page1.landedImmigrantRadio.value', 'off') +.
     set('Page1.otherRadio.value', 'off') +.
     set('Page1.' +. ..[custom:citizenship] +.
     '.value', 'on')
   : '' "></custom:toggle2>
```

Placing this compute in a bind, it looks like this:

```
<bind>
   <custom:toggle2
      xfdl:compute="toggle(..[custom:citizenship]) &#xA;
      == '1' &#xA;
      ? set('Page1.citizenRadio.value', 'off') +. &#xA;
        set('Page1.landedImmigrantRadio.value', 'off') +. &#xA;
        set('Page1.otherRadio.value', 'off') +. &#xA;
        set('Page1.' +. ..[custom:citizenship] +. &#xA;
        '.value', 'on') &#xA;
      : '' "></custom:toggle2>
```

Example of a Complete Bind for Radio Buttons

Adding all of the elements together, including the data holder and toggle elements, a complete bind would look like this:

```
<bind>
   <instanceid>customer</instanceid>
   <ref>[customerData][citizenship]</ref>
   <boundoption>..[custom:citizenship]</boundoption>
   <custom:citizenship></custom:citizenship>
   <custom:toggle1
      xfdl:compute="toggle(Page1.citizenRadio.value) == '1' &#xA;
      or toggle(Page1.landedImmigrantRadio.value) == '1' &#xA;
      or toggle(Page1.otherRadio.value) == '1' &#xA;
      ? set('..[custom:citizenship']], &#xA;
        getGroupedItem('Page1.citizenship', 'value', 'on', &#xA;
```

```
        'Page1', 'page', 'page')) &#xA;
      : '' "></custom:toggle1>
   <custom:toggle2
      xfdl:compute="toggle(..[custom:citizenship]) == '1' &#xA;
      ? set('Page1.citizenRadio.value', 'off') +. &#xA;
        set('Page1.landedImmigrantRadio.value', 'off') +. &#xA;
        set('Page1.otherRadio.value', 'off') +. &#xA;
        set('Page1.' +. ..[custom:citizenship] +. &#xA;
        '.value', 'on') &#xA;
      : '' "></custom:toggle2>
</bind>
```

Remember that this example assumes you have three radio buttons named *citizenRadio*, *landedImmigrantRadio*, and *otherRadio* in a group called *citizenship*, and a corresponding data element named <custom:citizenship>.

Reformatting Data With a Bind

In some cases, the data in the form description may not match the format required in the data instance. For example, in a purchase order the total amount might be formatted (using the *format* option) with a dollar sign, commas, and a two digit decimal place, as shown:

```
$1,123.59
```

However, in your data instance you may want to remove the dollar sign and the commas, so that you have a number with no formatting:

```
1123.59
```

You can control the formatting of data by using an optional *autoformat* tag. This tag is set to either *on* or *off*, as follows:

- **on** — The bind automatically strips all formatting from the form data before moving it to the data instance. Note that this applies only to formatting introduced by the *format* option of the form item, and does not change any formatting the user may have applied manually.
- **off** — The bind respects all formatting, and copies all data ″as is″ to the data instance.

For example, the following bind uses the *autoformat* tag to ensure that the bind respects all formatting:

```
<bind>
   <instanceid>customer</instanceid>
   <ref>[customerData][firstName]</ref>
   <boundoption>Page1.firstNameField.value</boundoption>
   <autoformat>off</autoformat>
</bind>
```

By default, all binds strip formatting from the data.

Manually Changing Formatting

In some cases, you may find that you need different formats than those produced with the *autoformat* tag. You can perform more complicated formatting by creating computes that reformat the value and store it in a data holder. This is very similar to binding lists, and requires the following elements:

- **Data Holder** — This element holds the formatted value, and is bound to the data instance.
- **Compute 1** — This element contains a compute that copies the value from the form description to the data holder, and reformats the value appropriately.

- **Compute 2** — This element contains a compute that copies the value from the data holder to the form description, and reformats the value appropriately.

For a more detailed explanation of how these elements work together, see ″Binding Lists, Popups, and Radio Buttons″.

Creating Submission Rules for an Instance

When submitting a form that contains an XML Data Model, you can submit either the entire form or just a particular data instance. This makes it possible to send your data instance directly to processing applications, rather than having to parse the complete form and extract the data instance.

If you want to submit a data instance, you must create a set of submission rules. These rules help determine what data is submitted, how the data is submitted, and where the data goes. In addition to submission rules, you must also create a submission button that is linked to the rules (for more information, see ″Creating a Submission Button″).

Each set of submission rules is inserted within the <submissions> tag in the XML model, as shown:

```
<xmlmodel>
   <submissions>
      ... all submission rules ...
   </submissions>
</xmlmodel>
```

Within the <submissions> tag, each set of submission rules is defined by a separate <submission> tag, as shown:

```
<submissions>
   <submission>
      ... submission 1 ...
   </submission>
   <submission>
      ... submission 2 ...
   </submission>
</submissions>
```

Each submission is further defined by adding attributes to the <submission> tag and by including an optional <ref> element. This is explained in more detail in the following sections.

Naming the Submission Rules

Each submission tag must include an *id* attribute. This tag names the submission rules, and follows this format:

```
id="name"
```

For example, if you wanted to call the submission rules *submitCustomerData*, you would use the following tag:

```
<submission id="submitCustomerData">
```

Setting the Target URL for a Submission

Use the *action* attribute to define the target URL for the submission. The *action* attribute is written in the following format:

```
action="URL"
```

You can only list one URL in the *action* attribute. For example, if you wanted to submit your data instance to a cgi script on your server, you might use the following submission tag:

```
<submission id="submitCustomerData"
   action="http://www.myserver.com/cgi">
```

If you do not provide an *action* attribute, the submission is sent to the first URL listed in the *url* option of the linked submission button.

Setting the Content Type of the Submission

Each submission tag may also include an optional *mediatype* attribute. This attribute is a MIME type that sets the content type of the HTTP submission. For example, if you wanted to set a MIME type of *application/vnd.xfdl* you would use the following tag:

```
<submission id="submitCustomerData"
   mediatype="application/vnd.xfdl">
```

If you do not provide a media type, it defaults to *application/xml*.

Setting Which Data is Submitted

By default, the first data instance in a form is submitted. If you want to submit a different data instance, you must define that instance using the <instanceid> tag, as shown:

```
<instanceid>instance id</instanceid>
```

Each data instance is identified by the *id* attribute in the <xforms:instance> tag.

You can also choose to submit the entire data instance or only a portion of the instance. When submitting only a portion of the data instance, you must identify the *root element* of the submission. The root element determines which portion of the instance is submitted, since only the root element and its children are sent.

For example, consider the following data instance:

```
<xforms:instance>
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <address>
         <street></street>
         <city></city>
         <country></country>
      </address>
   </customerData>
</xforms:instance>
```

By default, the first tag within the instance is the root element. In this case, <customerData> is the first tag and therefore the root element. This means that <customerData> and all of its children would be submitted by default.

However, if you only wanted to submit the address information, you could set the address tag to be the root element. In that case, only the <address> tag and its children would be submitted.

You set the root element by enclosing a <ref> tag in the <submission> tag, as shown:

```
<submission id="Page1.submitPOData">
   <ref>reference to root element</ref>
</submission>
```

The reference to the root element is written in the same array notation that is used by the <ref> element in bindings. For example, to refer to the <address> tag, you would write:

```
<ref>[customerData][address]</ref>
```

Note that this reference also obeys the same namespace rules as the <ref> element used for bindings. For more information, see ″Using Namespaces in Element References″.

Filtering Inherited Namespaces

By default, when you submit a data instance, the instance includes all of the namespaces that it inherits. For example, consider the following data instance:

```
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom>
   ...
      <xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms">
         <instances>
            <xforms:instance xmlns="http://www.mycompany.com/HR">
               <customerData>
                  <firstName></firstName>
                  <lastName></lastName>
                  <address></address>
               </customerData>
            <xforms:instance>
         </instances>
         ...
      </xmlmodel>
   ...
</XFDL>
```

When you submit the *customerData* instance, the root element of the submission is modified so that it declares all of the namespaces that it inherits. Assume that the root element is *customerData*. In this case, that tag declares a default namespace but also inherits the XFDL, custom, and XForms namespaces from the XFDL tag. As a result, the submission declares those namespaces on the *customerData* element, as shown:

```
<customerData xmlns="http://www.mycompany.com/HR"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom
   xmlns:xforms="http://www.w3.org/2003/xforms">
```

In some cases, you may want to restrict the inherited namespaces that are included with a data instance. For example, you may want to submit non-namespace-aware XML for DTD validation.

To restrict the inherited namespaces that are included, use the *includenamespaceprefixes* attribute. This attribute lists the prefixes for those namespaces that you want to include in the submission, and follows this syntax:

```
includenamespaceprefixes="prefix1 prefix2 prefix3"
```

Each prefix is separated by whitespace, such as a space. For example, to include only the XFDL and the custom namespaces in a submission, you would use the following submission tag:

```
<submission id="submitCustomerData"
   includenamespaceprefixes="XFDL custom">
```

If you want to submit only those namespaces that are used in your data instance, you can do this automatically by using an empty string, as shown:

```
<submission id="submitCustomerData" includenamespaceprefixes="">
```

This automatically removes all namespaces that are not used in your data instance.

Exceptions to Filtering

Filtering never excludes namespaces that are used in the data instance. This includes both the default namespace and any namespaces that are used within the instance.

For example, consider the following data instance:

```
<xforms:instance xmlns="http:www.mycompany.com/HR">
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <custom:address></custom:address>
   </customerData>
</xforms:instance>
```

This instance is in the Human Resources namespace by default, and uses the *custom* namespace for the address element. In this case, your submission would always include definitions for both the Human Resources namespace (as default) and the custom namespace, even if you did not include it in your filter.

For example, your submission filter might be set to include only the XDFL namespace, as shown:

```
<submission id="submitCustomerData"
   includenamespaceprefix="XFDL">
```

In this case, the root element of your submission would look like this:

```
<customerData xmlns="http:www.mycompany.com/HR"
   xmlns:XFDL="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom>
```

Including a Default Empty Namespace

You may have a data instance that defaults to the empty namespace, which is defined as "". For example, the following instance tag would create an instance in the default namespace:

```
<xforms:instance xmlns="">
```

Submissions that default to the empty namespace do not normally declare that namespace. If you want your submission to declare the empty namespace in the root element, you must include the appropriate prefix in the *inlcudenamespaceprefixes* attribute. The empty namespace is identified by the prefix: *#default*.

For example, to ensure your submission declares the empty namespace as default, you would use the follow filter:

```
<submission id="submitCustomerData"
   includenamespaceprefix="XFDL #default">
```

In this case, the root element of your submission would look like this:

```
<customerData xmlns=""
   xmlns:XFDL="http://www.ibm.com/xmlns/prod/XFDL/7.0">
```

Creating a Submission Button

Submission buttons are only necessary if you want to submit a data instance without the rest of the form. The button triggers the submission, and the data submitted is determined by combining the filters for the button with the submission rules in the data model.

To create a submission button, create a button of type *submit* or *done* and set the *transmitformat* option to:

```
application/xml;id="submission rule id"
```

By including the name of the appropriate submission rules, you link the button to that set of rules. This name is defined by the *id* attribute of the appropriate *submission* tag.

For example, if your submission rules had an *id* of ″submitCustomerData″, you would use the following button:

```
<button sid="submitCustomerDataButton">
   <type>done</type>
   <transmitformat>application/xml;
      id="submitCustomerData"</transmitformat>
</button>
```

For more information about using the submission button to filter the submission, see ″Filtering Submissions″.

Adding Schema Validation

When adding schema validation, you can choose to either embed one or more schema files in the form itself, or refer to external schema files that are saved on the user's computer. Embedding schemas in the form will increase the overall size of the form and may affect performance, especially when low bandwidth is available. However, referring to external schema files requires you to distribute those files to client computers. The architecture of your overall application will probably dictate which solution you should use.

Normally, each instance in the data model is validated against all available schemas. However, if a schema is defined for a particular namespace, only those instances that belong to that namespace are validated against it. This allows you to apply specific schemas to specific data instances.

Finally, you must restrict all schemas to a single, self-contained file. The Viewer does not support the use of the *import* or *include* tags.

To add schema validation to a form, you must:

1. Embed the schemas you want to include in the form.
2. Register the embedded schemas.
3. Register any external schemas.
4. Add the *xmlmodelValidate* function to the form.

Embedding a Schema in a Form

You can embed any number of schemas in a form. Each schema is inserted within the <schemas> tag in the XML model, as shown:

```
<xmlmodel>
   <schemas>
      ... all schemas ...
   </schemas>
</xmlmodel>
```

Within the <schemas> tag, each schema is defined in a separate <schema> tag, as shown:

```
<xmlmodel>
   <schemas>
      <schema>
         ... schema 1 ...
      </schema>
      <schema>
         ... schema 2 ...
      <schema>
   </schemas>
</xmlmodel>
```

Each schema is placed between the opening and closing schema tags, and must conform to the rules for XML schemas.

Naming a Schema

You must name each schema that you embed in the form with a globally unique *id*. You can do this by adding an *id* attribute to the <schema> tag. The *id* attribute follows this format:

```
id="name"
```

For example, if you wanted to give the name *customerData* to your schema, you would use the following <schema> tags to enclose the schema description:

```
<schema id="customerData">
   ... schema definition ...
</schema>
```

Registering Embedded Schemas

Embedded schemas are not automatically used when the data model is validated. Instead, only those schemas listed in the *schema* attribute on the <xmlmodel> tag are used. The *schema* attribute is written as shown:

```
schema="list of schemas"
```

You must list each of the embedded schemas by their *id* attribute, and precede each id with the # symbol. Furthermore, the list must be space delimited. For example, to register the *customerData* and *pricingData* schemas, you would use the following *schema* attribute:

```
schema="#customerData #pricingData"
```

The *schema* attribute is added to the <xmlmodel> tag. For example, a complete data model, with the *customerData* and *pricingData* schemas, would look like this:

```
<xmlmodel schema="#customerData #pricingData">
   <schemas>
      <schema id="customerData">
         ... schema definition ...
      </schema>
      <schema id="pricingData">
         ... schema definition ...
      </schema>
   </schema>
</xmlmodel>
```

Registering External Schemas

External schemas must be placed in the Viewer's *schema* folder or they will not be available to the form. You can add sub-folders to the Viewer's *schema* folder, but cannot place any schemas outside of that folder. As with embedded schemas, only those external schemas listed in the *schema* attribute on the <xmlmodel> tag are used during validation. Any other schemas in the Viewer's *schema* folder are ignored.

The *schema* attribute is written as shown:

```
schema="list of schemas"
```

You must list each of the external schemas by path and filename, relative to the Viewer's *schema* folder. This list is space delimited, which means that your schema filenames cannot contain spaces. Furthermore, you must add a prefix of *xsf:* to each path. For example, if both the *customerData.xsd* and *pricingData.xsd* schemas we in the Viewer's *schema* folder, you would use the following *schema* attribute to register those schemas:

```
schema="xsf:customerData.xsd xsf:pricingData.xsd"
```

Note that the *schema* attribute can refer to both embedded and external schema, as shown:

```
schema="#customerData xsf:pricingData.xsd"
```

The *schema* attribute is added to the <xmlmodel> tag. For example, a complete data model, with external *customerData* and *pricingData* schemas, would look like this:

```
<xmlmodel schema="xsf:customerData.xsd xsf:pricingData.xsd">
</xmlmodel>
```

Since the schemas are external, there is no need to include the <schemas> tag in the data model.

Adding the xmlmodelValidate Function

The data model is validated against all registered schemas when you call the *xmlmodelValidate* function in the form. In most cases, you will want to tie this function to the submission of the form, so that the data model is validated just before the form is submitted.

Optionally, you may prefer to validate the data model during processing on your back-end systems. In this case, you can use any available XML schema tools to validate the data model.

Validating Data on Submission

To validate data on submission, you must use the *toggle* function to trigger the *xmlmodelValidate* call when the user clicks the submit button. The most common way to do this is to create a compute that evaluates the results of the validation and either allows or aborts the submission.

The following algorithm describes this compute:

1. Use the toggle function to detect when the user clicks the submit button.
2. When the user clicks the button, trigger the *xmlmodelValidate* function.
3. Use the *set* function to place the results of *xmlmodelValidate* into a custom option.
4. Use the *strlen* function to determine whether the results indicate an error.
   - If the data validates, the result string will be empty and therefore have a length of zero.
   - If the data does not validate, the result string will have a length greater than zero.
5. If the data does not validate:
   - Use the *messagebox* function to inform the user.
   - Use the *set* function to deactivate the submit button. This will abort the submission.

The actual compute looks like this:

```
compute="toggle(activated, 'off', 'on') == '1'
   ? set('custom:results', xmlmodelValidate())
   + (strlen(reference to custom option storing results) > '0'
   ? viewer.messageBox('message to user')
   + set('activated', 'off')
   : '') : '' "
```

And if you placed this compute into a submit button, you would have the following:

```
<button sid="submitForm">
   <value>Submit</value>
   <type>done</type>
   <custom:results></custom:results>
   <custom:opt xfdl:compute="toggle(activated, 'off', 'on') &#xA;
      ==    '1' ? &#xA;
      set('custom:results', xmlmodelValidate()) &#xA;
      + (strlen(custom:results) > '0' &#xA;
      ? viewer.messageBox(custom:results) &#xA;
      + set('activated', 'off') &#xA;
      : '') : '' "></custom:opt>
</button>
```

Notice that this button contains two custom options: the *custom:opt* option contains the compute that validates the form and performs the necessary logic, while the *custom:results* option is set to contain the results of the *xmlmodelValidate* call. Also, notice that in this case the *messageBox* function displays the results of the *xmlmodelValidate* call. However, you could use this function to display any message you wanted.

Validating Data During Processing

To validate data during processing, you can use any of the publicly available tools for processing XML schema. IBM® does not supply a specific tool for this need.

Enabling Smartfill

Smartfill enables users to store frequently used form information on their local computers and to re-use this information when completing other forms that require it. Smartfill is intended to make it easier to fill out forms that require commonly used information, such as the user's name and address.

Smartfill works by creating *data fragments*. As the name suggests, these are small groupings (or *fragments*) of data from the form. Each data fragment represents a particular set of data. For example, you might create one data fragment for the user's home address and a second data fragment for the user's work address.

The first time a user submits or saves a Smartfill enabled form, the system will save the data fragments defined in the form to the user's local computer. The next time the user completes a form requiring those data fragments, the Viewer will offer to automatically load the information into the form.

Once a data fragment has been saved, the Viewer will offer to load that data into each form that requires it. However, once loaded, the user can modify the data at any time. Any changes the user makes will be saved over the old data fragment when the user saves or submit the form.

Because data fragments are stored as XML files within the user's profile on the local computer, they can be accessed by other applications. As such, they should only capture commonly used information, such as names and addresses. Never use them to store confidential information, such as credit card numbers or passwords.

The Smartfill feature allows you to include any number of data fragments in your form, and you should give careful consideration to how you want to define and arrange your data fragments.

Managing Smartfill Data

The Smartfill feature offers enormous flexibility with respect to which portions of a form can be automatically completed for the user. However, achieving this flexibility requires proper planning of both the data instances and the data fragments in your form.

Before creating a Smartfill enabled form, you should spend some time planning how to design the form, and should consider the following points:

1. Data fragments are best added to forms that are filled out frequently by the same user, or to forms that are part of a series that include the same basic information.
2. Data fragments should store commonly used information, such as names and addresses. Remember, never include confidential information in data fragments.
3. The information included in data fragments must overlap the information included in the data instances in your form. Because of this, you should design your data instances and data fragments at the same time, to ensure you are grouping the data correctly for both purposes. For more information about the relationship between data instances and data fragments, refer to "Defining the Contents of a Data Fragment".

4. Forms that will be completed in several stages by multiple users, or which complete different data fragments depending upon the user filling the form, must match form sections and data fragments with users. This will allow the form to determine which data fragments are valid for each user. For more information about creating forms with multiple data fragments intended for multiple users, see ″Specifying Whether a Data Fragment is Active″.

Creating a Data Fragment

You can embed any number of data fragments in a form. Each data fragment is inserted within the <datafragments> tag in the XML model, as shown:

```
<xmlmodel>
   <datafragments>
      ... all data fragments...
   </datafragments>
</xmlmodel>
```

Within the <datafragments> tag, each data fragment is defined in a separate <datafragment> tag, as shown:

```
<xmlmodel>
   <datafragments>
      <datafragment>
         ... data fragment 1 ...
      </datafragment>
      <datafragment>
         ... data fragment 2 ...
      <datafragment>
   </datafragments>
</xmlmodel>
```

Identifying a Data Fragment

You must uniquely identify each data fragment in the form. To do this, you must add both a <package> and a <name> element to the data fragment, as shown:

```
<datafragment>
   <package>package name</package>
   <name>fragment name</name>
</datafragment>
```

The package name is a unique name that identifies those data fragments used by your company. As a general rule, we recommend that you use your company's URL as your package name. For example:

```
<package>ibm.com</package>
```

The fragment name should uniquely identify the data fragment itself. For example, if you are storing address information in the data fragment, you may want to call it *addressInfo*, as shown:

```
<name>addressInfo</name>
```

Versioning a Data Fragment

Over time, you may modify or update the data fragments in your forms. For example, you may begin with an address fragment that has two fields for the street address, and then six months later realize that you only need one.

By making a change like this, you are effectively creating a new version of your address data fragment. The first version has two fields for the street, while the second version only has one. Unfortunately, this can cause problems when you begin using the updated form.

When a data fragment is loaded from disk, that portion of the form's data model is completely overwritten. This means that a portion of the existing data model is replaced with the information loaded from disk, regardless of whether the tags in that information match the tags in the data model.

This means that a form could potentially load the wrong information. For example, suppose your user was filling out the old version of your form. Once the form is completed, a data fragment with two fields for the street is saved to disk. A day later, the user opens a new version of your form, containing only one field for the street. However, since the name and package match, the data fragment with two fields is loaded into the form. Since there is only one field in the form to support the data, some data may be lost.

To avoid this, it's a good idea to embed version numbers when naming your data fragments. For example, rather than naming the fragment *addressData*, you could name it *addressDataV1*. Then, if you update that data fragment, you can create *addressDataV2*. This will prevent older versions of the data fragment from overwriting information in newer forms.

Providing a Description for a Data Fragment

You must also provide a description for your data fragment. This description is shown to the user whenever they are prompted to load or save the data, so you should tailor it for your users.

To add a description of the data fragment, enclose it in a <description> tag as shown:

```
<datafragment>
   <description>data fragment description</description>
</datafragment>
```

For example, you might want to explain to your users that the data fragment stores address information. In this case, your description might look like this:

```
<datafragment>
   <description>Address information, including street, city,
      ZIP, and country.</description>
</datafragment>
```

Defining the Contents of a Data Fragment

Each data fragment is defined as a sub-set of a single data instance. To define the data fragment, you link it to a single element in a data instance, and that element and all of its children are then included in the data fragment.

For example, suppose you had defined the following data instance in your data model:

```
<xforms:instance id="customer">
   <customerData>
      <firstName></firstName>
      <lastName></lastName>
      <address>
         <street></street>
         <city></city>
         <state></state>
      </address>
   </customerData>
</xforms:instance>
```

In this case, if you linked the data fragment to the <customerData> element, then the data fragment would include all of the data in the instance. However, if you linked the data fragment to the <address> element, then the data fragment would include only the street, city, and state information.

To create this link, you must add an <instanceid> element and a <ref> element to your data fragment, as shown:

```
<datafragment>
   <instanceid>id of instance</instanceid>
   <ref>reference to element in instance</ref>
</datafragment>
```

The <instanceid> tag contains the *id* of the data instance to which you want to link your data fragment. For example, to link to the *customer* data instance, you would use the following tag:

```
<instanceid>customer</instanceid>
```

The <ref> tag contains a reference to the element in the data instance that will be the root of your data fragment. For instance, if you wanted to link to the <address> element in the above example, you would use the following tag:

```
<ref>[customerData][address]</ref>
```

Specifying Whether a Data Fragment is Active

If your form will be completed in several stages by multiple users, such as in a workflow, or if it should complete different data fragments depending upon the user's identity, you may want it to load separate data fragments for each user. This means you must design the form so that it only loads data fragments when you want them to be loaded. You can use the *active* option to set whether a data fragment is active, as shown:

```
<datafragment>
   <instance id>
   <reference>
   <active>active setting</active>
</datafragment>
```

Valid settings for *active* are *on* and *off*. For example:

```
<active>off</active>
```

If a data fragment's *active* option is set to *on*, the data fragment is loaded when the form is first opened. If the fragment's *active* option is set to *off*, it is not loaded. Turning a data fragment's *active* to *on* after the form is open does not trigger a new data fragment to load until the next time the form is opened.

The default setting for *active* is *on*.

Note: You will need to use a compute to activate data fragments that have been specified as off when the form is opened. For more information regarding this compute, see "".

Example of a Complete Data Fragment

The following example shows a complete data fragment. This fragment links to the *address* element in a data instance named *customer*:

```
<datafragments>
   <datafragment>
      <package>ibm.com</package>
      <name>customerData</name>
```

```
      <description>Your address.</description>
      <instanceid>customer</instanceid>
      <ref>[customerData][address]</ref>
      <active>off</active>
   </datafragment>
</datafragments>
```

Working with Data Fragments

Once you have added a data fragment to your form, the Viewer will recognize and use that data fragment as part of the Smartfill feature. In general, this means that the Viewer will load and save Smartfill data as appropriate. However, there are some scenarios that may prevent the Smartfill feature from loading data. These scenarios include:

- Data fragments with storage ids
- Data fragments with *active* options set to *off*

About Storage IDs

Whenever the Viewer saves the data fragments in a form (during saving, signing, or submission), it also adds a <storageid> element to all of the *active* data fragment definitions. This element is used to indicate that the Smartfill data has been loaded, and to link the data fragment to the information stored on disk.

The <storageid> element belongs to the *df* namespace, and is added alongside the <package>, <name>, and other elements, as shown:

```
<datafragments>
   <datafragment>
      <package>ibm.com</package>
      <name>customerData</name>
      <description>Your address.</description>
      <instanceid>customer</instanceid>
      <ref>[customerData][address]</ref>
      <active>on<active>
      <df:storageid>unique identifier</df:storageid>
   </datafragment>
</datafragments>
```

Once the <storageid> element is in a data fragment, the Viewer will no longer attempt to load that Smartfill data. Furthermore, the Viewer will only save Smartfill data if: (a) the user changes the data in data fragment, and (b) the original data fragment is already saved on the local computer (as identified by the storageid). In general, these rules prevent the Viewer from loading or saving Smartfill data when it should not.

Note: If the <storageid> element is added to a form before it is distributed, it can prevent the form from functioning properly. For this reason, you should be extremely careful when designing and testing your forms. If you save a form while testing it in the Viewer, you may inadvertently add the <storageid> element to the form.

Activating Data Fragments

If you have specified that certain data fragments should be off when the form is originally opened, you need to be able to activate those data fragments at a later point. In other words, you need to use a compute to change a data fragment's *active* option to *on* after certain conditions are met. You can trigger this compute in several ways from within the form or on the server side. The simplest way is to

trigger the compute when the form is submitted. For example, to set the *active* option of the second datafragment to *on* when the Submit button is pushed, you would:

- Create a custom option inside the Submit button.
- Create an If/Then/Else statement which specifies that when the submit button has been activated, it should set the second data fragment's *active* option to on.

For example:

```
<custom:turnDFon xfdl:compute="toggle(activated, 'off', 'on')== '1' &#xA;
   ? set('global.global.xmlmodel[datafragments][1][active]', &#xA;
   'on') : '' "></custom:turnDFon>
```

Note: The form will only try to load a data fragment when it is first opened. If you activate a data fragment after that point, it will not be loaded until the next time the form is opened.

Sample XML Data Models

The following pages show a complete sample of the core XML data model, as well as samples of a data model with a schema, a data model with data fragment definitions, and a sample data fragment file.

Core XML Data Model

The following example shows a core XML Data Model with a single data instance. This data model includes submission rules that are linked to the *submitCustomerData* button in the form and that submit the complete data instance to the back-end for processing.

```
<xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms">
   <instances>
      <xforms:instance id="customer">
         <customerData>
            <firstName></firstName>
            <lastName></lastName>
            <address>
               <street></street>
               <city></city>
               <country></country>
            </address>
         </customerData>
      </xforms:instance>
   </instances>
   <bindings>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][firstName]</ref>
         <boundoption>Page1.firstNameField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][lastName]</ref>
         <boundoption>Page1.lastNameField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][address][street]</ref>
         <boundoption>Page1.streetField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][address][city]</ref>
         <boundoption>Page1.cityField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][address][country]</ref>
         <boundoption>Page1.countryField.value</boundoption>
      </bind>
   </bindings>
   <submissions>
      <submission id="submitCustomerData"
         action="http://www.myserver.com/cgi"
         mediatype="application/xml">
      </submission>
   </submissions>
</xmlmodel>
```

Data Model with Schema Validation

The following data model contains all the core parts of a data model, as well as a schema.

```
<xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms"
   schema="#customer xsf:invoice.xsd">
   <instances>
      <xforms:instance id="customer">
         <customerData>
            <firstName></firstName>
            <lastName></lastName>
            <address>
               <street></street>
               <city></city>
               <country></country>
            </address>
         </customerData>
      </xforms:instance>
   </instances>
   <bindings>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][firstName]</ref>
         <boundoption>Page1.firstNameField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][lastName]</ref>
         <boundoption>Page1.lastNameField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][address][street]</ref>
         <boundoption>Page1.streetField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][address][city]</ref>
         <boundoption>Page1.cityField.value</boundoption>
      </bind>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][address][country]</ref>
         <boundoption>Page1.countryField.value</boundoption>
      </bind>
   </bindings>
   <submissions>
      <submission id="submitCustomerData"
         action="http://www.myserver.com/cgi"
         mediatype="application/xml">
      </submission>
   </submissions>
   <schemas>
      <schema>
         ...schema details...
      </schema>
   </schemas>
</xmlmodel>
```

Data Model with Smartfill

The following data model contains all the core parts of a data model, as well as a data fragment that enables the Viewer's Smartfill feature.

```
<xmlmodel xmlns:xforms="http://www.w3.org/2003/xforms">
   <instances>
      <xforms:instance id="agent">
         <agentContactInfo>
            <name>
               <firstName>Leslie</firstName>
               <lastName>Smith</lastName>
            </name>
            <phone>555-455-7676</phone>
         </agentContactInfo>
      </xforms:instance>
   </instances>
   <bindings>
      <bind>
         <instanceid>agent</instanceid>
         <ref>[agentContactInfo][firstName]</ref>
         <boundoption>Page1.firstNameField.value</boundoption>
      </bind>
      <bind>
         <instanceid>agent</instanceid>
         <ref>[agentContactInfo][lastName]</ref>
         <boundoption>Page1.lastNameField.value</boundoption>
      </bind>
      <bind>
         <instanceid>agent</instanceid>
         <ref>[agentContactInfo][phone]</ref>
         <boundoption>Page1.phoneField.value</boundoption>
      </bind>
   </bindings>
   <submissions>
      <submission id="submitAllData"
         action="http://www.myserver.com/cgi"
         mediatype="application/xml">
      </submission>
   </submissions>
   <datafragments>
      <datafragment>
         <package>ibm.com</package>
         <name>customerData</name>
         <description>Your address.</description>
         <instanceid>customer</instanceid>
         <ref>[customerData][address]</ref>
         <active>on<active>
      </datafragment>
   </datafragments>
</xmlmodel>
```

Filtering Submissions

When submitting forms that use the XML Data Model, you can use the transmit filters (*transmititems*, *transmitoptions*, and so on) to apply indirect filtering to the data model. While the transmit filters do not allow you to specify elements of the data model, the filters are still applied to the data model through the bindings that exist. For example, omitting the *firstName* field from your transmission would also omit the *firstName* data element if that element was bound to the field.

You can submit data from the data model in two different ways:

- **Submit the Complete Form** — In this case, you create a submission button that submits the form, and filters out the parts of the form you do not need. The filters you create apply both to the data elements in the form and the bindings in the form. For example, if you omit the *firstName* field, then the *firstName* data element and the bind that links the two elements are also omitted.
- **Submit a Data Instance Only** — In this case, you create a submission button that submits a single data instance from the data model. The filters you create apply to the data elements based on their bindings.

If you submit only a data instance, you can also filter the submission by setting the root element for the submission. For more information about this, see ″Setting Which Data is Submitted″.

Applying Transmit Filters to the XML Data Model

Transmit filters are applied to the data model based on the bindings in the form. For example, if you omit the *firstName* field, and that field is bound to a *firstName* data element, then that element is also omitted.

This indirect filtering is governed by a number of rules that may conflict with each other depending on the complexity of your data instance. As a general rule, a data element is included whenever one or more rules support its inclusion. For example, if three rules would omit the element, but one rule would include it, then that element is included.

Furthermore, if any data element is included, then any bindings that include that data element are also included.

Filtering Rules

This section explains the rules that apply when determining which data elements are filtered.

Basic Rules for Filtering Data Elements

In the simplest cases, filtering follows these rules:

- If a data element is bound to a form element that is included, then that data element is also included.
- If a data element is bound to a form element that is omitted, then that data element is also omitted.

Filtering Data Elements with Multiple Binds

If a data element is bound to multiple form elements, the following rules apply:

- If a data element is bound to multiple form elements, and any of those form elements are included, then the data element is also included.
- If a data element is bound to one or more form elements, and all of those form elements are omitted, then the data element is also omitted.

Filtering Data Elements that are Bound to Other Data Elements

If a data element is bound to another data element, the following rules apply:

- If a data element is bound to another data element, then the first data element follows the behavior of the second data element.
- If a data element is bound to multiple data elements, and any of those data elements are included, then the first data element is also included.

Filtering Data Elements with No Binds

If a data element is not bound to anything, the following rules apply:

- If a data element is not bound to the form, then that element is included.
- If a data element is not bound to the form, and contains any children that are either not bound or are included, then that element is included.
- If a data element is not bound to the form, and all of its children are omitted, then that data item is omitted.

Filtering Data Elements with Attributes

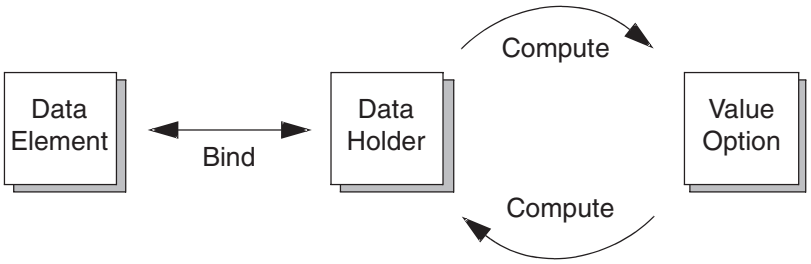
If a data element contains attributes, the following rules apply:

- If any of an element's attributes are bound to another element that is included, then the first element is included with all attributes that are not explicitly omitted.
- If all of an element's attributes are bound to elements that are omitted, then the first element and all of its attributes are also omitted.

Filtering Lists, Popups, and Radio Buttons

Filtering of the XML data model is normally governed by bindings. For instance, if the value of a field is bound to an element in the data instance, and that value is omitted by a filter, then the element in the data instance is also omitted because of the link created by the bind.

However, the values of lists, popups, and radio buttons are not directly linked to a data element through a bind. Instead, the bind links the element to a data holder, which is then linked to the value option by two computes, as shown:



Because of this indirect relationship, filtering does work properly with these item types. The item's value is not directly bound to anything, so the corresponding element in the data model is not filtered properly.

To correct this, you must add an <associated> tag to your bind, as shown:

```
<bind>
   <associated>reference</associated>
</bind>
```

The tag contains a reference to an option or array element in the form. This creates a direct link between the data element in the XML model and the option in the form, and this link is used when determining how the data element is filtered.

For example, consider the following bind (note that the computes have been removed to improve readability):

```
<bind>
   <instanceid>customer</instanceid>
   <ref>[customerData][citizenship]</ref>
   <boundoption>..[custom:citizenship]</boundoption>
   <custom:citizenship></custom:citizenship>
   <custom:toggle1 compute></custom:toggle1>
   <custom:toggle2 compute></custom:toggle2>
   <associated>Page1.citizenPopup.value</associated>
</bind>
```

This bind is for a citizenship popup called *citizenPopup*. The data element, <citizenship>, is bound to the data holder, <custom:citizenship>. The bind also includes two custom elements that provide computes for copying data from the <custom:citizenship> element to the popup's value. Finally, the <associated> element creates a direct link from the <citizenship> data element to the value of the *citizenPopup*.

This link ensures that filtering applied to the *citizenPopup* is also applied to its associated data element.

Using Computes with the XML Data Model

In general, computes work normally with the XML Data Model. However, before you use computes to change your data model, you should understand:

- The limitations to using computes with the data model.
- How computed changes to the data model affect bindings.

Limitations To Using Computes

The following limitations apply when using computes with the data model:

- Computes do not automatically update the data model in memory.
- Computes do not work within data instances.

Updating the Data Model in Memory

There are two ways that the data model may change. The first type of change occurs when data is copied between a data instance and the form description through a binding. This process occurs automatically as the user interacts with the form, or when the data instances are populated from a database.

The second type of change occurs when the data model itself is changed. For example, a compute might change a *boundoption* from *Field1.value* to *Field2.value*. Changes like this affect the overall structure of the data model, and are not updated automatically. Instead, the data model is updated when you use a special function called *xmlmodelUpdate*. This function is available both in the XFDL language and in the Workplace Forms API.

If you want to use the *xmlmodelUpdate* function in a form, you would call it with the following syntax:

```
xmlmodelUpdate()
```

For example, you might create a compute that sets a reference in the bindings section of the data model, as follows:

```
set('global.global.xmlmodel[bindings][0][boundOption]',
   'Page1.firstNameField.value')
```

This sets the first bind in the data model to use the *firstNameField*. However, once you've set this bind you may also want the data model to update immediately. To do this, you would add the *xmlmodelUpdate* function to your compute, as shown:

```
set('global.global.xmlmodel[bindings][0][boundOption]',
   'Page1.firstNameField.value') + xmlmodelUpdate()
```

This would set the bind and then immediately update the XML model to reflect that change.

If you want the data model to update while processing the form on a server, you would be more likely to use the API function. This works in a similar manner, but would be called by your processing application at the appropriate time. For more information about this function, refer to the *IBM Workplace Forms Server - API User's Manual*.

Computes in Data Instances

Computes do not work within data instances. For example, the following data instance attempts to use a compute to populate the *firstName* element:

```
<xforms:instance>
   <customerData>
      <firstName compute="Page1.firstNameField.value"></firstName>
   </customerData>
</xforms:instance>
```

In cases like this, the compute is simply ignored. However, you can use the *create*, *destroy*, and *set* functions in other parts of the form to change data instances. This allows you to add or remove elements from the data instance or set values as the user works with the form.

How Computed Changes Affect Bindings

Computed changes can affect bindings in two ways:

- Computes can create or destroy bound elements.
- Computes can create or destroy bindings.

Different rules apply in each case, and you should familiarize yourself with these rules before using computes to change bindings.

Creating and Destroying Bound Elements

You can use computes to create or destroy bound elements within a form. This does not affect the binding itself, as the binding remains in the form. However, if one or more of the bound elements is missing from the form, there are two possible results, depending on whether *xmlmodelUpdate* is called:

- If a bound element is missing and *xmlmodelUpdate* is called, then the bound element is created. The only exception to this is if the element's page or item level ancestors do not exist. For example, if the *value* option of *Field1* was bound, but *Field1* itself did not exist, then it is impossible to create the *value* option.
- If a bound element is missing and *xmlmodeUpdate* is not called, then the bind is deactivated. The bind still exists in the form, but cannot copy information to or from the missing element. The bind is reactivated as soon as the missing element is restored.

For example, consider the following data model:

```
<xmlmodel>
   <instances>
      <xforms:instance>
         <customerData>
            <firstName></firstName>
            <lastName></lastName>
         </customerData>
      </xforms:instance>
   </instances>
   <bindings>
      <bind>
         <instanceid>customer</instanceid>
         <ref>[customerData][firstName]</ref>
         <boundoption>Page1.firstNameField.value</boundoption>
      </bind>
   </bindings>
</xmlmodel>
```

In this model, the *firstName* element is bound to the *value* option of the *firstNameField*. If you destroyed the *value* option and did not call *xmlmodelUpdate*, the bind would remain but would be inactive. If you later created the *value* option again, the bind would become active once again.

If you want to permanently destroy a bound element, you should also destroy the binding. This ensures that the element is not created again when *xmlmodelUpdate* is called.

Creating and Destroying Bindings

You can use computes to create or destroy bindings. However, this sort of change is not registered until you call the *xmlmodelUpdate* function.

For example, consider the following data model:

```
<xmlmodel>
   <instances>
      <xforms:instance>
         <customerData>
            <firstName></firstName>
            <lastName></lastName>
         </customerData>
      </xforms:instance>
   </instances>
   <bindings>
      <bind>
         <instanceid>customerData</instanceid>
         <ref>[customerData][firstName]</ref>
         <boundoption>Page1.firstNameField.value</boundoption>
      </bind>
   </bindings>
</xmlmodel>
```

In this case, you could destroy the *bind* element to remove the binding between the *firstName* element and the *firstNameField*. While the bind element is destroyed immediately, you must call the *xmlmodelUpdate* function to update the model in memory.

Signing an XML Data Model

By default, signatures apply to the complete form, including the XML data model. Once signed, the XML data model is locked along with the rest of the form. This means that the Viewer does not allow the user to make changes, and any changes made through other means will break the signature.

You can also filter signatures by including options such as *signitems*, *signoptions*, and so on, in your signature button. These filters are applied to the XML data model in the same way as transmit filters. For detailed rules for filtering the XML Data Model, see ″Filtering Submissions″. For further information regarding signing filters and their order of precedence, refer to the *Creating Signature Buttons in XFDL* document.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Office 4360
One Rogers Street
Cambridge, MA 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM
Workplace
Workplace Forms

Other company, product, or service names may be trademarks or service marks of others.

Index

X

Program Number:

Printed in USA

S325-2608-00