XEP-0336: Data Forms - Dynamic Forms

Peter Waher
mailto:peterwaher@hotmail.com
xmpp:peter.waher@jabber.org
http://www.linkedin.com/in/peterwaher

2015-11-09
Version 0.2

Status	Type	Short Name
Deferred	Standards Track	dynamic-forms

This specification provides extensions to the data forms model defined in previous XEPs that permit enhanced end-user interaction and a better user experience. These extensions permit forms to react on user input by permitting the addition, updating or removal of fields in the form and server-side validation of fields. The extension also defines new states making it possible to display disabled controls, controls with undefined values or error messages, while still being backwards compatible with the existing data form model with available extensions.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

Data forms are used in many XEPs and provide a mechanism whereby a form can be hosted on one end and displayed on another. XMPP data forms are defined and enhanced in many different XEPs, as is shown in the following list:

- Data Forms (XEP-0004) [1] defines the basics of data forms: How forms are defined, sent to the recipient, how the recipient submits forms (or cancels them) and how results can be returned. It defines the concepts of field, field type and field value.

- Data Forms Validation (XEP-0122) [2] enhances the data forms architecture permitting rules for client-side validation of fields in the form.

- Publishing Stream Initiation Requests (XEP-0137) [3] defines a new data type that can be used to publish file upload controls.

- Data Forms Layout (XEP-0141) [4] enhances the data forms architecture permitting the form to have pages or tabs with sections containing grouped controls.

- Data Forms Media Element (XEP-0221) [5] defines a means to include content such as images, video or audio in forms.

- Data Forms - Color Field Types (XEP-0331) [6] permits the publication of color fields, where the end user can be presented with a color picker dialog instead of having to enter a color value manually or select one from a limited list of colors.

This specification enhances the data form model further by providing the following features:

- Form post-back when fields flagged for post-back are edited. This permits the form to adapt itself based on user input, i.e. creating, changing or removing fields in the form, creating dynamic forms. It also provides a mechanism to publish server-side validation of fields during user input.

[1] XEP-0004: Data Forms <https://xmpp.org/extensions/xep-0004.html>.
[2] XEP-0122: Data Forms Validation <https://xmpp.org/extensions/xep-0122.html>.
[3] XEP-0137: Publishing Stream Initiation Requests <https://xmpp.org/extensions/xep-0137.html>.
[4] XEP-0141: Data Forms Layout <https://xmpp.org/extensions/xep-0141.html>.
[5] XEP-0221: Data Forms Media Element <https://xmpp.org/extensions/xep-0221.html>.
[6] XEP-0331: Data Forms - Color Field Types <https://xmpp.org/extensions/xep-0331.html>.

- Allows fields to be disabled. They will be shown as normal controls, but in a disabled state. Combining post-back form fields it is possible to enable and disable (or add or remove) fields depending on user input.

- Allows fields to have not well-defined values. This can be used for instance, when a value is unknown or when multiple objects are edited at the same time in the same form and the objects have different values for the corresponding field.

- Permits fields to have errors flagged on them. This can be used to give users intuitive feedback on errors in forms. Together with post-back fields, this provides a mechanism whereby server-side validation of fields can be made while the user is still editing the form.

- Permits spontaneous updates of the form, pushed from the form server to the client showing the form. This might be necessary in cases where the process of aquiring current values may take some time, but the client is required to show a form right away.

Note: This extension is only dependent upon the Data Forms XEP. It works in parallel with any of the above mentioned data form extensions, but do not require that any of them are supported. The examples provided in this document may still reference extensions made in other documents, but these are considered to be examples only, used to illustrate a specific point or example.

1.1 Comparison to Ad-hoc commands

Ad-Hoc Commands (XEP-0050) [7] defines a mechanism to enhance data forms creating dialog wizards with actions that guide the user through pages, where each page can depend on the input from previous pages. The following list consists of a comparison of the differences between Ad-hoc commands and Dynamic Forms, and why this extension is not based on the concepts defined in XEP-0050:

- The main point of this extension is to make the current form dynamic. Pages in a wizard, as defined in ad-hoc commands, are still static requiring page navigation for the dialog to be updated.

- There might be a desire to make individual pages inside a wizard dynamic by themselves. In this case, this extension might be used to extend Ad-hoc commands with

[7] XEP-0050: Ad-Hoc Commands <https://xmpp.org/extensions/xep-0050.html>.

dynamic pages inside wizards.

- Wizards order their pages linearly, in a one-dimensional array of pages. Navigation is only performed using prev and next actions, and completed with the execute or complete actions. This might work sufficiently well where page output only is based on input from the previous page. But as soon as the input depends on more pages, navigation becomes unnecessarily cumbersome. For example a dialog with country, region, city, area, street, building, apartment. To change the country parameter when you are on the apartment page, requires you to back through many pages, while in the dynamic form you just change the parameter directly, since all are visible in the same form.
  The problem gets worse if this is a normal behavior for a form. Say a dialog with a post carrier field, followed by country, city, office, etc. When the user comes to office and notices that the selected carrier has no office close to you, you change carrier, but don't want to change country and city, if possible.

- The pages in a wizard do not allow for spontaneous or immediate feedback to the user. Such immediate feedback is of great importance in user-friendly forms, where complex input is required. Consider the following simple example:
  Imagine a dialog used for configuring a TCP port scanner to search for specific devices in a network. It might take five parameters: IP range, port range, number of threads, connection timeout (ms) and number of tries before failing. However, the operator wants to know how much time the operation will take (which is #IP-addresses * #port numbers * Connection Timeout * Nr Tries / #Threads). To see this in a separate page and then have to go back to the different pages, forwarding to the time estimate calculation page, etc., is a very cumbersome process. Presenting all this information on a single page using Dynamic Forms is creates a much better and richer user experience. Changing the value of any of the five parameters will directly update a text parameter presenting the total expected time of the operation.

- Wizards as defined in Ad-hoc commands are hard-wired to commands as such. Navigation actions are defined outside of the scope of the form itself. This extensions defines Dynamic Forms as an extension of the Data Form concept. All navigation, post-back fields, etc., are defined within the actual form itself. This makes it re-usable everywhere data forms are used, including pages inside an Ad-Hoc command wizard.

- Ad-hoc command wizards include the feature to return notes including information, warnings or error information to the user. This information however, is associated with the current page in the wizard itself. Dynamic Forms contains a means to attach error information directly to individual fields, making feedback more precise. Together with the server post-back feature immediate server-side validation during user input is

possible.

- Dynamic Forms contains a feature that ad-hoc command wizards do not: Read-only fields. These differs from other text fields in that they are rendered as normal controls, except input is read-only. Together with the server post-back feature this allows the server to enable and disable parameters depending on user input on the same page. An example can be a checkbox that enabled another field if checked, as a security measure.

- Dynamic Forms defines the concept of an undefined field value. These fields are presented with the available value (if any), but greyed out, signaling to the user that the value is not well defined for some reason. Perhaps it is only one of many values held my multiple objects in the case when editing multiple objects at once. Undefined values are not sent back to the server when the form is posted back. When the user edits such a field, the undefined flag is cleared and the field is presented as a normal control. The main purpose of this flag is to allow for editing multiple objects at once or for editing control forms where current states are not known at the moment if creating the form.

- Dynamic Forms supports asynchronous updates of the form, where the server can push changes to the form not resulting from user input. This is a powerful feature that allows the server to update a form being edited by the user to reflect changes on the server. Consider the following examples:
  Consider a control form containing control fields on a remote device. At the time of displaying the form, the current states of some fields might be unknown. So the fields are marked perhaps with some default values, but with the undefined flag set for the corresponding fields. At the time of creating the form, a parallel request is made to the device by the form server, requesting information about current states of the device. When these are received by the server, it issues an update of the form to the client with the newly received and current field values. The request for values might take some time, so using this mechanism provides a form to the user quickly, clearly indicating what is missing, and then complements the form when data is available.
  Another example might be a dialog showing contents on the form server in a multi-user environment where updates to the contents is made. An example can be a file system. If changes to the contents is made by another user, the server has the possibility to update any current forms to reflect changes made. This decreases the possibility of inconsistencies in the system, and at the same time increases the user-friendliness of the end-user experience of the application.

2 Glossary

The following table lists terms and corresponding descriptions or definitions for use throughout this document.

Control The visual control used to display a field.

Data Form A form of parameters (a.k.a. fields), as defined in Data Forms.

Field A field representing a parameter or control, in a data form, as defined in Data Forms.

Form Client For simplicity, we will call the XMPP client that fills in the data form, the form client.

Form Server For simplicity, we will call the XMPP client that hosts a data form, the form server.

Parameter Used sometimes synonymously with field.

3 Use Cases

The following subsections list use cases for the different enhancements defined by this extension. Elements are defined using the namespace **urn:xmpp:xdata:dynamic** and namespace prefix **xdd**.

3.1 Publishing post-back fields

Post-back fields are fields that require the client to post the form to the server after being edited. This permits the server to perform server-side validation on the field, provide immediate user feed-back and change the form according to user input.
Post-back fields are declared as normal fields in a form, except they also have the **xdd:postBack** flag present in the declaration, as is shown in the following example:

Listing 1: Publishing post-back fields

```
<x xmlns="jabber:x:data" type="form"
       xmlns:xdd="urn:xmpp:xdata:dynamic"
       xmlns:xdv="http://jabber.org/protocol/xdata-validate">
        <title>Current location</title>
        <instructions>Select your current location to continue.</
            instructions>
        <field var='xdd_session' type='hidden'>
            <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
        </field>
        <field var="Country_ISO_3166_1" type="list-single" label="
            Country:">
            <desc>Select your country of residence.</desc>
            <xdv:validate xmlns:xdv="http://jabber.org/protocol/xdata-
                validate" datatype="xs:string">
```

```
            <xdv:basic/>
        </xdv:validate>
        <value/>
        <xdd:postBack/>
        <option label="Chile">
            <value>CL</value>
        </option>
        <option label="Sweden">
            <value>SE</value>
        </option>
        <option label="United States">
            <value>US</value>
        </option>
        ...
    </field>
</x>
```

Note: A system maintaining multiple dynamic forms open at the same time needs to maintain control of available open forms. This can be done using a hidden session variable, as is shown in the example above. How this is done however, is implementation specific.

3.2 Performing a server post-back

A server post-back is performed by sending an IQ set stanza with a **submit** child element containing the current state of the form. The type of the form must be **submit**. The client should also provide the current user language in a **xml:lang** attribute, if available.

Listing 2: Performing a server post-back

```
<iq type='set'
    from='formclient@example.org/client'
    to='formserver@example.org'
    id='1'>
    <submit xmlns='urn:xmpp:xdata:dynamic' xml:lang='en'>
        <x xmlns="jabber:x:data" type="submit">
            <field var='xdd_session'>
                <value>009c7956-001c-43fb-8edb-76bcf74272c9</value
                    >
            </field>
            <field var="Country_ISO_3166_1">
                <value>CL</value>
            </field>
        </x>
    </submit>
</iq>
```

It is important to note that this server post-back is part of editing the form, and must not be treated by the server as a final submission of the data form.

As a response to a successful form post-back, the server returns the new data form, as is shown in the following example:

Listing 3: Post-back response

```
<iq type='result'
        from='formserver@example.org'
        to='formclient@example.org/client'
        id='1'>
        <x xmlns="jabber:x:data" type="form"
           xmlns:xdd="urn:xmpp:xdata:dynamic"
           xmlns:xdv="http://jabber.org/protocol/xdata-validate">
            <title>Current location</title>
            <instructions>Select your current location to continue.</
                instructions>
            <field var='xdd_session' type='hidden'>
                <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
            </field>
            <field var="Country_ISO_3166_1" type="list-single" label="
                Country:">
                <desc>Select your country of residence.</desc>
                <xdv:validate xmlns:xdv="http://jabber.org/protocol/
                    xdata-validate" datatype="xs:string">
                    <xdv:basic/>
                </xdv:validate>
                <value>CL</value>
                <xdd:postBack/>
                <option label="Chile">
                    <value>CL</value>
                </option>
                <option label="Sweden">
                    <value>SE</value>
                </option>
                <option label="United_States">
                    <value>US</value>
                </option>
                ...
            </field>
            <field var="Region_ISO_3166_2" type="list-single" label="
                Region:">
                <desc>Select your region of residence.</desc>
                <xdv:validate xmlns:xdv="http://jabber.org/protocol/
                    xdata-validate" datatype="xs:string">
                    <xdv:basic/>
                </xdv:validate>
                <value/>
                <xdd:postBack/>
                <option label="Antofagasta">
                    <value>AN</value>
```

```
                </option>
                <option label="Arica y Parinacota">
                    <value>AP</value>
                </option>
                <option label="Atacama">
                    <value>AT</value>
                </option>
                ...
            </field>
        </x>
    </iq>
```

In this example, the server adds a new parameter to the form, containing options that depend on the value of the first parameter.
Note: Server post-back should be made after the user is done editing a post-back field, but before actually navigating to the next field. It should not be done while the user is editing the field. For check boxes or list boxes, it is easy for the application to decide when the field has been edited, since the application can react to the controls click event. But for text edit boxes, you cannot perform a server post-back only just because the text property of the control has changed. You need to wait until the user leaves the control or something similar. However, as new fields can be added to the form, the client should wait for the response before deciding which control is the next control to go to.

3.3 Publishing read-only fields

Read-only fields should be laid out on the form just as a similar but editable field would, except editing should be disabled. Together with post-back fields, this option allows the form to enable/disable fields depending on user input. Read-only fields are declared as normal fields in a form, except they also have the **xdd:readOnly** flag present in the declaration, as is shown in the following example:

Listing 4: Publishing read-only fields

```
<x xmlns="jabber:x:data" type="form"
       xmlns:xdd="urn:xmpp:xdata:dynamic"
       xmlns:xdv="http://jabber.org/protocol/xdata-validate">
        <title>Object properties</title>
        <field var='xdd_session' type='hidden'>
            <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
        </field>
        <field var="ID" type="text-single" label="ID:">
            <desc>ID of object.</desc>
            <xdv:validate xmlns:xdv="http://jabber.org/protocol/xdata-
                validate" datatype="xs:string">
                <xdv:basic/>
            </xdv:validate>
```

```
            <value>Object 1</value>
            <xdd:readOnly/>
        </field>
        <field var="RenameID" type="boolean" label="Rename object">
            <desc>To avoid accidental renaming of the objcet, this box
                must be checked to rename the object.</desc>
            <value>0</value>
            <xdd:postBack/>
        </field>
        ...
    </x>
```

Note: Using this flag is different from the field type **fixed** defined in Data Forms. Fields of type **fixed** can also be used to display read-only content. However, this is done as static text, and not as a disabled control, specific for the type of content corresponding to the data in question. Using the **readOnly** flag instead, gives greater flexibility when it comes to presentation, as well as permitting the form server to enable and disable controls during the lifetime of the form.
Note 2: Make sure to check the implementation note Merging Client-Side Values for information on how to merge updates received from the server with current input made by the client.

3.4 Publishing fields with undefined values

There are many cases where you want to flag a control as having an **undefined value** or an **uncertain** value, instead of simply a missing value. This permits the form client to display the control in a specific way (for instance greyed), and omit the field when submitting the form back to the server, to avoid errors.
This technique can be used for instance, when editing the properties of multiple objects at the same time. Attributes that are equal among the objects can be reported as normal fields, while attributes that have different values among the objects, can be reported as having a value that is not the same everywhere. The client can then show the corresponding control greyed and omitting it in submissions of the form. If the user edits the control however, the form client can remove the flag and render the control normally. Submitting the now well-defined field value will set the corresponding attribute in all objects to the same new value. Omitting undefined values makes sure that the corresponding attributes are left as-is.
Fields with undefined or uncertain values are declared as normal fields in a form, except they also have the **xdd:notSame** flag present in the declaration, as is shown in the following example:

Listing 5: Publishing fields with undefined values

```
<x xmlns="jabber:x:data" type="form"
       xmlns:xdd="urn:xmpp:xdata:dynamic"
       xmlns:xdv="http://jabber.org/protocol/xdata-validate">
        <title>Communication properties</title>
```

```
        <field var='xdd_session' type='hidden'>
            <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
        </field>
        <field var="Address" type="text-single" label="Bus_Address:">
            <desc>Enter the bus address of the device.</desc>
            <xdv:validate xmlns:xdv="http://jabber.org/protocol/xdata-
                validate" datatype="xs:int">
                 <xdv:range min="1" max="250"/>
            </xdv:validate>
            <value>1</value>
            <xdd:notSame/>
        </field>
        <field var="BaudRate" type="list-single" label="Baud_rate:">
            <desc>Baud rate to use when communicating with the device.
                </desc>
            <value>2400</value>
            <option label='300_baud'><value>300</value></option>
            <option label='2400_baud'><value>2400</value></option>
        </field>
        ...
    </x>
```

In the above example communication properties of a set of objects are edited. The form contains two fields, one address field, which will contain unique values for each individual object and therefore be flagged with the **xdd:notSame** flag, and a second baud rate field. This second field will probably contain the same value, if devices are connected to the same bus. Therefore, the field value is not flagged with the **xdd:notSame** flag.
Note: If the **xdd:notSame** flag is available in a field, the field must not be flagged as being **required**. They must also be omitted in any form submissions unless they have been edited first. Furthermore, any field submitted in a post-back as described in this document, must not be returned in the post-back response having the **xdd:notSame** flag.
Notice also that there's a difference between a missing value, i.e. a field without a **value** element defined, and a field with a **value** defined, but with the **xdd:notSame** flag present. In the latter example, the value might represent the value of one of the objects in a set of objects being edited.

3.5 Publishing fields containing errors

A field can be flagged with an error message using the **xdd:error** element. Together with the post-back feature this allows for more advanced server-side validation of field values, as is shown in the following example.

Listing 6: Publishing fields with undefined values

```
<x xmlns="jabber:x:data" type="form"
       xmlns:xdd="urn:xmpp:xdata:dynamic"
```

```
        xmlns:xdv="http://jabber.org/protocol/xdata-validate">
         <title>Expression</title>
         <field var='xdd_session' type='hidden'>
             <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
         </field>
         <field var="Expression" type="text-single" label="Expression:"
             >
             <desc>Enter the expression you want to plot.</desc>
             <xdv:validate xmlns:xdv="http://jabber.org/protocol/xdata-
                 validate" datatype="xs:string">
                 <xdv:basic/>
             </xdv:validate>
             <value>sin(x</value>
             <xdd:postBack/>
             <xdd:error>Unexpected end of expression. ) expected.</
                 xdd:error>
         </field>
         ...
    </x>
```

In the above example, the user can enter a script expression into a text box using a server-specific scripting language. There's no way for the client to validate the script expression. However, by flagging the field with **xdd:postBack** the client posts the form back to the server when the user is finished writing the expression, and the server can validate it. In the example, the user has made an error, and the server returns the field, with an **xdd:error** flag, containing an error message that can be displayed to the user in an appropriate way.
Note: If a field is flagged with an error, and the user starts editing it, the client should remove the error flag of the field. The field can be flagged again with a new error flag during the next post-back.

3.6 Cancelling a dynamic form

The Data Forms XEP provides a mechanism to cancel forms, by submitting the form using **type='cancel'**. The Data Forms XEP stipulates that fields should not be included when using the form type cancel. However dynamic forms, i.e. forms containing post-back fields, will need the fields, since any dynamic form session will be identified using hidden fields in the form. So, to cancel a dynamic form, the **xdd:cancel** element with the entire submitted form should be sent to the form server using an IQ set stanza, as is shown in the following example:

Listing 7: Cancelling a dynamic form

```
<iq type='set'
        from='formclient@example.org/client'
        to='formserver@example.org'
        id='4'>
        <cancel xmlns='urn:xmpp:xdata:dynamic'>
```

```
            <x xmlns="jabber:x:data" type="submit">
                <field var='xdd_session'>
                    <value>009c7956-001c-43fb-8edb-76bcf74272c9</value
                        >
                </field>
                ...
            </x>
        </cancel>
    </iq>
```

After receiving the cancel request the form server returns an empty response if the form was found (and therefore cancelled), or an IQ error stanza with an **item-not-found** error if the form was not found. The following example shows a response where the form was found and cancelled:

Listing 8: Dynamic form cancelled

```
<iq type='result'
        from='formserver@example.org'
        to='formclient@example.org/client'
        id='4'/>
```

Note: If cancelling a dynamic form using the approach described in this document, there's no need to also submit a cancel form as defined in the Data Forms XEP. The form server automatically makes sure the form is cancelled in all instances on the form server.
Note 2: If the dynamic form is invoked from a specific operation that includes its own cancel procedure, like Ad-hoc command sessions, the dynamic form is automatically and implicitly cancelled if the corresponding operation is cancelled. There is no need to explicitly cancel the dynamic form as explained in this section in such cases.

3.7 Dynamic form not found during post-back

It might happen that the form server does not find the dynamic form posted by the form client during a post back. Reasons for this can be that the form does not include a post-back field, or that a form session timeout has occurred and the form server has discarded the dynamic form to avoid memory leaks. Regardless of the reason, the form server responds using an IQ error stanza with the **item-not-found** error, when the client posts a form that is not found back.

Listing 9: Dynamic form not found during post-back

```
<iq type='error'
        from='formserver@example.org'
        to='formclient@example.org/client'
        id='2'>
        <error type='cancel'>
```

```
            <item-not-found xmlns='urn:ietf:params:xml:ns:xmpp-stanzas
                '/>
        </error>
    </iq>
```

Note: Form post-back must only be performed on forms containing post-back fields. The form server is not required to maintain dynamic form sessions for forms that lack post-back fields.

3.8 Other error during post-back

If another error occurs during post-back, the form server can inform the client about this by using the relevant error element and provide further information in a **text** element to describe the error, as is shown in the following example:

Listing 10: Other error during post-back

```
<iq type='error'
        from='formserver@example.org'
        to='formclient@example.org/client'
        id='3'>
        <error type='cancel'>
            <internal-server-error xmlns='urn:ietf:params:xml:ns:xmpp-
                streams'/>
            <text>An internal error occurred: Stack limit has been
                reached.</text>
        </error>
    </iq>
```

3.9 Asynchronous updates of forms (server push)

The server may send asynchronous updates to open forms on the client. This can be done if the server detects changes that it wants to inform the client about. Changes are made by sending a **message** stanza including a **updated** element which in turn contains the new updated form.
Examples can include a control form showing control parameters. While the server is trying to retrieve the current values it presents the control form with undefined values, and later when values are received by the server, it sends an update to the client with actual values.
Another example can include a dialog containing information on items on the server in a multi-user environment (for instance a file system). Changes made by users can be displayed in open dialogs to other users as they change.
The client may have more than one form open at any given time. It might also be so that the form has been closed prior to receiving or handling the update message, and is therefore no longer visible. To be able to identify to which form the update corresponds, the **updated** element is required to include a **sessionVariable** attribute in which it identifies a **unique**

identifying field in the form. When the client receives the update, it goes through all forms it has open. If a form has a field variable with a corresponding name, and the field variable has a value equal to the value in the updated form, the form should be updated by the contents of the message. If no form is found, the update is simply ignored. If multiple forms are found, all should be updated.

Listing 11: Asynchronous update of form (server push)

```
<x xmlns="jabber:x:data" type="form"
       xmlns:xdd="urn:xmpp:xdata:dynamic"
       xmlns:xdv="http://jabber.org/protocol/xdata-validate">
        <title>Control parameters</title>
        <field var='xdd_session' type='hidden'>
            <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
        </field>
        <field var="AnalogOutput" type="text-single" label="Analog_
            Output:">
            <desc>Enter a new value for the analog output.</desc>
            <xdv:validate xmlns:xdv="http://jabber.org/protocol/xdata-
                validate" datatype="xs:int">
                <xdv:range min="0" max="65535"/>
            </xdv:validate>
            <value>0</value>
            <xdd:notSame/>
        </field>
    </x>

    ...
    <message from='server@example.org'
             to='client@example.org/client'>
        <updated xmlns='urn:xmpp:xdata:dynamic' sessionVariable='xdd_
            session' xml:lang='en'>
            <x xmlns="jabber:x:data" type="form"
               xmlns:xdd="urn:xmpp:xdata:dynamic"
               xmlns:xdv="http://jabber.org/protocol/xdata-validate">
                <title>Control parameters</title>
                <field var='xdd_session' type='hidden'>
                    <value>009c7956-001c-43fb-8edb-76bcf74272c9</value
                        >
                </field>
                <field var="AnalogOutput" type="text-single" label="
                    Analog_Output:">
                    <desc>Enter a new value for the analog output.</
                        desc>
                    <xdv:validate xmlns:xdv="http://jabber.org/
                        protocol/xdata-validate" datatype="xs:int">
                        <xdv:range min="0" max="65535"/>
                    </xdv:validate>
                    <value>49152</value>
```

```
                </field>
            </x>
        </updated>
    </message>
```

Note: Make sure to check the implementation note Merging Client-Side Values for information on how to merge updates received from the server with current input made by the client.
Note 2: The client should also provide the current user language in a **xml:lang** attribute in the **updated** element, if available, as is shown in the example above.

4 Determining Support

If an entity supports the protocol specified herein, regardless if the entity represents a form server or a form client; it MUST advertise that fact by returning a feature of "urn:xmpp:xdata:dynamic" in response to Service Discovery (XEP-0030) [8] information requests.

Listing 12: Service discovery information request

```
<iq type='get'
        from='formclient@example.org/client'
        to='formserver@example.org'
        id='disco1'>
        <query xmlns='http://jabber.org/protocol/disco#info'/>
    </iq>
```

Listing 13: Service discovery information response

```
<iq type='result'
        from='formserver@example.org'
        to='formclient@example.org/client'
        id='disco1'>
        <query xmlns='http://jabber.org/protocol/disco#info'>
            ...
            <feature var='urn:xmpp:xdata:dynamic'/>
            ...
        </query>
    </iq>
```

In order for an application to determine whether an entity supports this protocol, where possible it SHOULD use the dynamic, presence-based profile of service discovery defined in Entity Capabilities (XEP-0115) [9]. However, if an application has not received entity capabilities information from an entity, it SHOULD use explicit service discovery instead.

[8] XEP-0030: Service Discovery <https://xmpp.org/extensions/xep-0030.html>.
[9] XEP-0115: Entity Capabilities <https://xmpp.org/extensions/xep-0115.html>.

5 Implementation Notes

5.1 Dynamic Form Sessions

For a form server to maintain the status of the dynamic form, it will probably need to publish state or session information using hidden fields in the dynamic form. It's important that form clients be aware that such hidden fields are available and must always return them in all submissions of the form to the server.
A form client that supports dynamic forms and opens a dynamic form, i.e. containing parameters requiring post-back, must call the specific **cancel** method described in this document to cancel the form if not submitted. This to let the form server to release any dynamic form session resources it maintains.
Note: When submitting a dynamic form the normal way, any dynamic form session resources are also automatically released.

5.2 Session Timeouts

The form server must be aware that some form clients do not support dynamic forms. This in turn implies that form clients may not call the correct cancel method to cancel a dynamic form. To protect the form server from memory leaks, it must include a session timeout, and release any dynamic form session resources if no activity has been made during the corresponding time period. If the client would perform a post-back after the timeout period, an **item-not-found** error message must be returned, to show the corresponding dynamic form session no longer exists, and therefore could not be found.
For normal operations, a dynamic form session timeout of 15 minutes is sufficient.

5.3 Merging Client-Side Values

When receiving asynchronous form updates from the server, or when performing a server post-back of a form, it is important to know how to merge responses from the server with the current form being displayed to the user. As the operation is asynchronous, and since user input is quick, the user might have input things not known to the server and therefore not available in form updates. Also, fields not marked for post-back might not have been reported at all to the server, and therefore, the client is the only one that knows what the user has entered into these fields.
So, when receiving form updates, either asynchronously, or as part of a server post-back response, the client needs to merge the updated form, with the current form. The following rules must be applied. Here, the **updated form** represents the form in the recent message from the server, the **current form** represents the form currently being edited by the user and the **resulting form** represents the result of the merger of the updated form and the current form.

- New fields in the updated form not available in the **current form**, are added as-is to the **resulting form**.

- Fields not available in the **updated form** but available in the **current form**, must be removed from the **resulting form** regardless if user input is available. Any such user input is lost.

- The order of fields in the resulting form must be the same as the order of fields in the **updated form**.

- Fields available in both the **updated form** and the **current form** are handled depending on if the user has entered values in the corresponding field in the **current form** or not:

  - If the user has **not** edited the value in the corresponding field, the value from the **updated form** is used.
  - If the user has edited the value the corresponding field, the value from the **current form** is used.
  - If the user has edited the value of the corresponding field in the **current form**, but the value is the same as the value available in the **updated form**, the flag stating that user input has occurred in the field can be cleared.

- If a field in the **updated form** is flagged as **undefined**, but the **current form** has an edited value, the form in the **resulting form** must **not** be marked as **undefined**.

- All other properties for fields must be taken from the **updated form** and copied to the **resulting form**.

How the above merger is made is implementation specific. One simple implementation can simply be taking the updated form, adding any client-side values to it (i.e. values edited in the current form) perhaps removing any undefined value flags, and then use the result as a model for the resulting form.

6 Security Considerations

There are no security concerns related to this specification above, beyond those described in the relevant section of **XMPP Core**.

7 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA) [10].

8 XMPP Registrar Considerations

The protocol schema needs to be added to the list of XMPP protocol schemas.

9 XML Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='urn:xmpp:xdata:dynamic'
    xmlns='urn:xmpp:xdata:dynamic'
    xmlns:xd="jabber:x:data"
    elementFormDefault='qualified'>

    <xs:import namespace='jabber:x:data'/>

    <xs:element name='postBack'>
        <xs:annotation>
            <xs:documentation>Flags a field as requiring server post-
                back after having been edited.</xs:documentation>
        </xs:annotation>
        <xs:complexType/>
    </xs:element>

    <xs:element name='readOnly'>
        <xs:annotation>
            <xs:documentation>Flags a field as being read-only.</
                xs:documentation>
        </xs:annotation>
        <xs:complexType/>
    </xs:element>

    <xs:element name='notSame'>
        <xs:annotation>
            <xs:documentation>Flags a field as having an undefined or
                uncertain value.</xs:documentation>
```

[10] The Internet Assigned Numbers Authority (IANA) is the central coordinator for the assignment of unique parameter values for Internet protocols, such as port numbers and URI schemes. For further information, see <http://www.iana.org/>.

```
        </xs:annotation>
        <xs:complexType/>
    </xs:element>

    <xs:element name='error'>
        <xs:annotation>
            <xs:documentation>Flags a field as having an error.</
                xs:documentation>
        </xs:annotation>
        <xs:simpleType>
            <xs:restriction base='xs:string'/>
        </xs:simpleType>
    </xs:element>

    <xs:element name='submit'>
        <xs:complexType>
            <xs:sequence>
                <xs:element ref='xd:x' minOccurs='1' maxOccurs='1'/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name='updated'>
        <xs:complexType>
            <xs:sequence>
                <xs:element ref='xd:x' minOccurs='1' maxOccurs='1'/>
            </xs:sequence>
            <xs:attribute name='sessionVariable' type='xs:string' use=
                'required'/>
        </xs:complexType>
    </xs:element>

    <xs:element name='cancel'>
        <xs:complexType>
            <xs:sequence>
                <xs:element ref='xd:x' minOccurs='1' maxOccurs='1'/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

10 Acknowledgements

Thanks to Karin Forsell and Lance Stout for all valuable feedback.