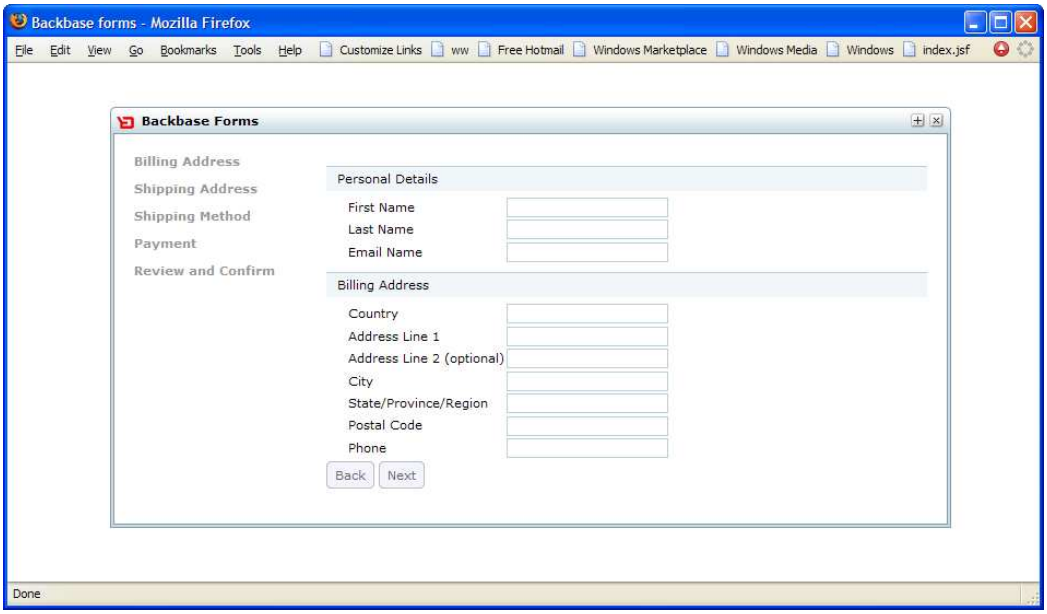
The Forms Starter Kit

1. Introduction



The Forms starter kit is a special starter kit, which almost purely concentrates on form based functionality. It aims to provide a framework for client-side form validation. Since it is a dynamic BXML-based application, it does however also touch on other essential areas of BXML such as include files, behaviors and custom client controls. All of these will be briefly explained before we move on and start to really dissect this forms-based functionality.

In a nutshell the application itself is merely a set of forms that enables the user to enter his or her personal details, shipping address, billing address and payment details, finally providing a summary of this. It is therefore not a freestanding application but should instead be seen as a module, to allow for the completion of an ordering process. The series of forms is clearly divided up into steps. Before any of these steps can be completed, all of the required fields must have been filled in and some kind of field level validation will have been preformed on those fields, which require data to be entered into them in a fixed format – an email address is a good example of this. This entire field-based validation and ensuring that required fields are filled is done on the client. The whole interface is in fact not only a single page interface, it also consists of only

two main files, one that determines the structure of the forms-based interface and the other that contains all of the behaviors.

2. Including Files

A key technique used for keeping functionally distinct modules separate from each other is the use of include files. Include files are well-formed xml files, which can contain both BXML and normal HTML. They can be small and simple, merely containing a few behavioral instructions or a small module such as a shopping cart. They can also be very large and themselves contain multiple nested include files.

In the Forms Starter Kit two files are included right at the start of the main index.html file.

```
<s:include b:url="forms.xml"/>
<s:include b:url="../../controls/basic/button.xml"/>
```

The forms.xml file loads in all of the functionality used for the form logic. It also contains two custom controls which are used to help bring structure and grouping to the forms based interface. These are the b:step and the b:group elements, which will be explained later in more detail. The second include file is for button.xml, which is used to include a standard b:button control from the control library.

3. Behaviors

Behaviors are of fundamental importance to BXML. Almost the entirety of forms.xml consists of behaviors. It is here that all of the form-based logic for the checking of required fields and validating special fields is contained. This section will aim to explain basic behaviors and how they work. The next section will explain a little XPath and then finally we will put it all together and try to explain the salient points of this form-handling framework.

One of the key strengths of BXML is the ease and simplicity with which events can be handled. The main tool set used for this is behaviors. A behavior is a generic construct in which, you the developer, can define which instructions or in BXML terminology, tasks, should be executed when a given event occurs. This behavior can then be used by any given element, which will then inherit all of the event handlers defined within that behavior. This makes it easy to reuse functionality and also to separate the content of the page from its behavior.

Since all of the behaviors in this Forms Starter Kit are rather complicated, it is better to examine a more basic behavior, which isn't found in this starter kit. The show-hide behavior shown below allows the element, which uses this behavior, to be shown when it is selected and hidden when it becomes deselected.

```
<s:behavior b:name="show-hide">
  <s:event b:on="select">
    <s:task b:action="show" />
  </s:event>
  <s:event b:on="deselect">
    <s:task b:action="hide" />
  </s:event>
</s:behavior>
```

This show-hide behavior can now be used by an infoview or some other box, which we want to show and hide as and when it becomes selected.

A behavior can be made more sophisticated by applying control flow tags like s:if or s:choose to it. The follow code fragment, which is a part of the form-field-generic behavior from the forms.xml include file, demonstrates this:

```
<s:behavior b:name="form-field-generic">
  <s:event b:on="construct">
    <s:choose>
      <s:when b:test="@type='radio'">
        <s:setatt class="input-radio"/>
      </s:when>
      <s:when b:test="@type='button'">
        <s:setatt class="input-button"/>
      </s:when>
      <s:when b:test="@type">
        <s:setatt class="input-standard"/>
      </s:when>
      <s:otherwise>
        <s:setatt class="form-element"/>
      </s:otherwise>
    </s:choose>
  </s:event>

  ... rest of event handlers ...

</s:behavior>
```

As you can see the construct event of all of the elements, which use this behavior are given a handler. This construct event is triggered when either the index.html file is loaded or when an include file gets loaded in. When the construct event is handled the interpreter finds a s:choose tag. This s:choose tag works similarly to the JavaScript switch operator. A number of tests are performed sequentially by the s:when tags, until one of the tests returns true. If none of the tests return true, then any instructions in the s:otherwise tag gets executed. In this case the tests performs an XPath instruction, which retrieves the value of the type attribute and then compares this to a string value. If it matches one of these string values, like ‘radio’ or ‘button’, then it executes an s:setatt instruction. This s:setatt instruction causes any attributes found within this tag to be set onto the actual element that uses this behavior. In this case the class attribute gets set to the value given in the s:setatt tag. So as you can see, this construct event handler ensures that all elements that use this behavior get given an appropriate class when they are constructed, based on their type attribute.

4. XPath

An essential aspect of BXML is the ability to target elements on the screen and to be able to retrieve information about these elements or their attributes. The XPath language is used for all of these types of operations. Almost every task that is executed has a target upon which this task should be executed. This target is not always visible in the statement, since the default target for most tasks is the element, which is using the behavior itself. It is very important to have at least a basic understanding of XPath if you want to be able to build BXML applications. The more thoroughly you understand XPath, the more you will be able to leverage its power across BXML and the more powerful and flexible applications you will be able to build. Such a thorough understanding of XPath is especially important for this starter kit, since the form validation uses especially heavy XPath.

XPath statements which are used to select on screen elements are commonly found in the b:target attribute of a s:task statement. For example if you examine the form-field behavior in the forms.xml include file, you will see the following instruction in the mouseenter event handler:

```
<s:task b:action="show" b:target="id('form-field-info')" />
```


This instruction selects the element indicated by the b:target attribute, which is the element with the b:id attribute of 'form-field-info'. Since this is a show action, this element will then get shown.

If you now look at the rest of this event handler you can see a lot more XPath. Lets examine it piece by piece and try to really understand how it all works.

```
<s:event b:on="mouseenter">
  <s:if b:test="@b:valid and @b:valid != 'true'">
    <s:task b:action="set" b:target="id('form-field-info')" b:attribute="innerHTML"
            b:value="{@b:info}" />
    <s:task b:action="show" b:target="id('form-field-info')"/>
    <s:task b:action="position" b:type="place" b:target="id('form-field-info')"
            b:destination="." b:position="after-pointer"/>
  </s:if>
</s:event>
```

Firstly there is an s:if tag which is executed right at the start of the event handler to determine whether any action should be taken at all. The b:test attribute contains the following XPath statement:

```
@b:valid and @b:valid != 'true'
```

When analyzing an XPath there are two important things to be aware of. Firstly there is the question of data type. Different attributes expect the XPath resolver to return different types of values to them. A b:target for example expects an element or a collection of elements to be returned to it – this is called a nodeset in XPath parlance. A b:value attribute on the other hand expects a string value to be returned to it. And in this case the b:test expects a boolean value to be returned to it. If the 'wrong' type of value gets given as the result of an XPath then the BXML engine will try to convert it to the expected data type. So in this case the b:test attribute requires a boolean value. The XPath expression consists of two parts joined by an and operator. The first half of the statement simply tests whether there is a b:valid attribute at all, irregardless of its value. The second part tests whether it contains a value that is not equal to 'true'. So the tasks contained in this event handler will not be executed unless there is a b:valid attribute, which has a value other than 'true'.

The event handler then contains 3 tasks. The first task is a set task. A set task is generally used to set an attribute value. It has 3 important attributes that control this process. There is another b:target attribute, which is used to indicate which element should have its attribute set. In this case it uses an id based XPath to target the form-field-info div element again. This is a div element that is normally hidden and is used to display error messages when a form field has not had its value entered correctly.
The b:attribute attribute is used to indicate which attribute on the form-field-info element should be set. In this case it is the innerHTML attribute of the form-field-info element. This will cause the text contents of the element to be set. Finally there is the b:value attribute. This is used to indicate the value that should be set. This can be a static value, which will then always be the same for every use of this behavior. It is also possible to use an XPath statement, which is done here. This is done by surrounding the contents of the b:value attribute with {curly braces}. The XPath in this case is simply:

```
@b:info
```

This causes the contents of the b:info attribute of the current element to be copied to the contents of the form-field-info element, so that it can show the value of the b:info attribute as the error message.

The second task was the show task, which we already explained. Finally there is the third task, which is a position task. Positioning tasks are used to move elements around the screen and absolutely position or resize them in some way. The key attribute is the b:type attribute which is used to determine which type of positioning should be used. In this case it is a place operation. This is used to position the target element relatively to a second destination element. In this case the target element, i.e. the one that gets moved is indicated by the b:target element and the destination element, which is the anchor point for the positioning is provided by the b:destination attribute. In this case the target element is the form-field-info element, and the destination is the element which receives the mouseenter event as indicated by the XPath statement '.'.

This example should hopefully have made it quite clear what an essential role XPath plays in most behaviors.

5. Custom Client Controls

An important technique used in building BXML application is the creation and reuse of custom client controls. These are also sometimes referred to as widgets. Basically one of these custom controls is a definition for a new BXML tag, which can then be used throughout your application.

A relative simple example of such a custom client control is provided in the forms.xml file. Towards the bottom of this file you can find a s:htmlstructure tag. This tag is used to define the b:group tag. This b:group tag is a new tag, which can be used to group form components and then easily switch them on and off, both visually and as far as validation is concerned. Lets have a look at the code and see how this is done.

```
<s:htmlstructure b:name="b:group">
  <div style="display: none">
    <s:innercontent/>
  </div>
</s:htmlstructure>
```

Basically as you can see the definition for this control consists of a special tag called s:htmlstructure. This is the tag, which is used to define new tags. With the b:name attribute you give the name of the new tag, which is b:group in this case. All custom defined tags must be in the b namespace. Within the s:htmlstructure tag you can insert any HTML tags which you want to be rendered when the new tag is used. Note that only HTML tags may be used within the s:htmlstructure tag. Finally inside this HTML the s:innercontent tag is inserted at the point where you want the child elements of the new tags to be placed. Obviously the child elements of the new tag may be any BXML tag and not just HTML. When we examine the HTML that makes up this new b:group tag, it doesn't seem to do much more than place an invisible div around the contents, thereby making the entire contents of the tag invisible too. This doesn't sound very useful at all, but there is more. A b-group behavior is defined and an s:default tag is specified:

```
<s:behavior b:name="b-group">
  <s:event b:on="construct">
    <s:task b:action="trigger" b:event="deselect"/>
  </s:event>
  <s:event b:on="select">
    <s:task b:action="show"/>
    <s:task b:action="set" b:target="* | label/*" b:attribute="b:disabled"
            b:value="false" />
  </s:event>
  <s:event b:on="deselect">
    <s:task b:action="hide"/>
    <s:task b:action="set" b:target="* | label/*" b:attribute="b:disabled"
```

```
            b:value="true" />
  </s:event>
</s:behavior>

<s:default b:tag="b:group" b:attribute="b:behavior" b:value="b-group"/>
```

It is important to note the bottom line in the code fragment above. This is the s:default tag, which is the glue that binds this behavior to all instances of b:group. The s:default tag is relatively easy and simple to understand. It simply binds a given value to a given attribute on a given element. In this case it is the b-group behavior which gets bound to all b:group elements. By examining this b-group behavior the working of the b:group element should become clear.

Firstly there is an event handler for the construct event. This construct event is triggered when either the form.html file is loaded or when an include file gets loaded in which contains an element that uses this behavior. The event handler contains a single trigger task. A trigger task is a special task that is used to trigger a new event. In this case it is a deselect event. Since no b:target attribute has been specified, it uses the default value for b:target which is the element on which the event occurs itself. The deselect handler, which now gets executed, first hides the b:group element, then issues a set task. It is this set which is of vital importance for the functionality of the b:group tag. The b:target of the deselect is:

```
* | label/*
```

This is a fairly involved XPath that may need a little explaining. The asterix (*) is a wild card, which means that it will select any child element of the current context-node, which is the b:group element. After the asterix there is a pipe (|). This pipe acts as a union operator, which means that this XPath is a compound statement, consisting of two separate XPath statements. The result of these two statements will be combined, when they have both been executed. The second statement select all children of b:group which are label entities and then uses the wild card again to select any child of these label entities. Once this group of targets have been selected, then the b:disabled attribute of these targets is set to ‘true’. This is important because during validation of the form any element with a b:disabled attribute with a value of ‘true’, gets ignored. So this is what enables the contents of b:group element to be hidden and effectively switched off as far as far form validation is concerned.

The select event handler does the opposite. It shows the b:group element and its contents and it sets all of its target elements’ b:disabled attributes to ‘false’.

Now lets look at how this b:group tag is used in form.html. It is used as part of the second step of the shipping form. Since a client’s billing address and shipping address are frequently the same, but can be different, the user is given the option of specifying whether they are the same or not. This option is presented using a set of radio buttons. When the radio button with the label: ‘Different Address’ is selected then suddenly the group becomes selected and is made visible. When the other radio button is selected, then the group disappears. The code for this fragment is displayed below.

```
<div class="label">Same as the billing address
  <div class="form-field">
    <input type="radio" name="shipping-address" value="same-as-billing" />
  </div>
</div>

<div class="label">Different address
  <div class="form-field">
    <input type="radio" name="shipping-address" value="different-from-billing" />
```



```
  </div>
</div>

<b:group b:name="shipping"
         b:followstate="ancestor::b:step[1]//input[@value='different-from-billing']">
  <div class="label">Country
    <div class="form-field">
      <input type="text" name="shipping_country" b:required="true" />
    </div>
  </div>
  <div class="label">Address Line 1
    <div class="form-field">
      <input type="text" name="shipping_address" b:required="true" />
    </div>
  </div>

  ....

</b:group>
```

You might now, quiet fairly, be wondering how clicking on one of these two radio buttons cause the b:group element to become selected or deselected. Indeed if you look carefully, neither of these radio buttons seem to have any special behaviors attached to them. The clue is in the b:followstate attribute on the b:group tag. An element that has a b:followstate attribute on it, follows the state of its target. So if its target becomes selected it then too becomes selected. And if its target becomes deselected it in turn becomes deselected. The value of the b:followstate attribute is an XPath. It isn't the most straightforward XPath so lets examine it:

```
ancestor::b:step[1]//input[@value='different-from-billing']
```

The first step of this XPath uses the ancestor axis. What this does is look up the BXML node hierarchy, instead of down it. So from the b:group context node it selects the first b:step element above it. From there it uses a double forward slash (//) to indicate that it should go back down the hierarchy and select any descendant of this b:step element, which is an input element and which has a value attribute of 'different-from-billing'. This element is the second radio button. So now, effectively the b:group element follows the state of the radio button with the label 'Different Address'. When this radio button becomes selected, the b:group element becomes selected and vice versa.

6. Putting it all Together

So by now some of the basic and most important concepts of BXML should be clear and it is time to put it altogether and explain the heart of this form processing application. There are three levels of validation in this process: field-level validation, step-level validation and finally form submission.

Field-Level Validation

The primary level of validation is field-level validation, this specifically checks to see whether correct content has been entered into an input field. A field can be designated to be a postcode field for example, which means that it has to fit a postcode pattern. Or a field could be designated to be a month, which would mean it has to have a value between 1 and 12. So how does this validation take place? The key to this process is the form-field-generic behavior and the use of certain user-defined attributes on the input and other form fields. Lets take a look at this process in more detail.

The key event on the form-field-generic behavior is the validate event. This is a user-defined event, which means that it has to be triggered by another task. It

gets triggered at several points in the whole process, but for the actual field-level validation the key trigger point is at the change event.

```
<s:event b:on="change">
  <s:task b:action="trigger" b:event="validate"
          b:target="ancestor::form[1]//*[@name = current()/@name]"/>
</s:event>
```

The change event is triggered, when a form field loses its focus and its value has changed, since it received the focus. The event handler for the change event, now triggers the validate event on all the children of the parent form which have the same name attribute as the current field. This might seem like a bit of a round-about way to trigger a validate event on the current field, but it is possible that more than one field have the same name attribute, such as a group of check boxes or a group of radio buttons.

Once the validate event is triggered, its handler is executed. This is the largest and most complex event handler in the whole of the forms application. It is suggested that you take a good look at this. Because it is so large, it won't be fully explained here, but lets take a look at some important parts of it now. It essentially consists of two sets of s:choose tags. The first one is responsible for determining whether the field is valid or invalid. It consists of more than 10 different s:when options. All of these steps test for different conditions. A few of these cases will now be illustrated. The following condition tests whether a required field has not been filled in:

```
<s:when b:test="@b:required='true' and regexp(@value, '^[ ]*$')
                and not(@type='radio' or @type='checkbox')">
  <s:setatt b:valid="required" b:info="This is a required field"/>
</s:when>
```

As you can see if the b:required attribute has been set to 'true' and the field is empty or only consists of whitespace, then a s:setatt command is executed, which sets the b:valid attribute to 'required' and sets the special b:info attribute which is used by the mouseenter event handler to display a user message as was explained in the section about XPath.

The next condition tests whether the field validates to a Dutch zip code, which have the format 1234AA.

```
<s:when b:test="@b:validation='zipcode'
                and not(regexp(@value, '^[1-9][0-9]{3} ?[a-zA-Z]{2}$'))">
  <s:setatt b:valid="false" b:info="A zip code should be in the form 8888AA"/>
</s:when>
```

This tests to see whether the target has a b:validation attribute which is equal to 'zipcode' and if so if the value of the input then matches the appropriate regular expression pattern which is used to express a Dutch zip code.

In this way many different conditions get tested. If you examine all of these conditions you will see that the following parameters can be used to assist and enable this type of form validation.

- The **b:required** attribute can have 'true' or 'false' values to indicate if the field is required.
- The **b:validation** attribute can have the values 'month', 'year', 'email' and 'zipcode', which indicates that the field value should be matched against the appropriate regular expression to validate this.
- The **b:disabled** attribute can be set to 'true' or 'false'. If set to 'true', the field is not checked for validity.

- The **b:minoccurs** attribute can be set to a numeric value on checkboxes or multi select boxes to set a minimum number of checked items within the group.
- The **b:maxoccurs** attribute can be set to a numeric value on checkboxes or multi select boxes to set a maximum number for checked items within the group.
- The **b:valid** attribute can have the values 'true', 'false' and 'required'. This attribute is not set while writing the code, but is set by the *validate* event handler.
- The **b:info** attribute can be filled with a message by the *validate* event handler, that is shown on *mouseenter* of an invalid field.

If all of these tests fail the execution ends up in the s:otherwise statement, which sets the b:valid attribute to 'true'. The second s:choose in the validate event handler simply checks whether this b:valid attribute is set to 'true' or not. If so it triggers a valid event, otherwise an invalid event.

It is interesting to see how these valid and invalid events are handled. They are handled by a second behavior, form-field, which is related to the original form-field-generic behavior. By using a special property of behaviors it is possible for one behavior to extend another behavior. This is the case for the form-field behavior, which extends the form-field-generic behavior. If we look at the definition of the form-field-generic behavior, you can see a b:behavior attribute on the s:behavior tag itself. It is this b:behavior attribute which indicates that the form-field behavior extends the form-field-generic behavior.

```
<s:behavior b:name="form-field" b:behavior="form-field-generic">
  <s:event b:on="valid">
    <s:setstyle b:border="" />
  </s:event>
  <s:event b:on="invalid">
    <s:setstyle b:border="1px solid red" />
  </s:event>

  ....

</s:behavior>
```

So what does this mean? When one behavior extends another behavior, this extending behavior then inherits all of the base-level behavior's event handlers. It is a little bit like sub-classing of a super class in object-oriented programming. For full details of this process see the manual. It is this second, extending behavior, which is used by all of the input, select and textarea elements in the application. The relation between these elements and the form-field behavior is laid using several s:default tags.

Step-Level Validation

The next part of this picture is formed by the b:step element. This is another custom tag, which is defined within the forms.xml file. It has a very similar structure to that of b:group; the custom tag which was previously examined. It consists of a s:htmlstructure tag, which defines another hidden div element. Its linked behavior is somewhat more complicated and is called b-step. These b:step elements are used to both visually and logically group parts of the form together. If you examine this b-step behavior, then you will see that it has a number of interesting event handlers. The construct event triggers the following handler:

```
<s:event b:on="construct">
  <s:if b:test="@b:state='selected' or position() = 1">
    <s:task b:action="trigger" b:event="select" />
  </s:if>
```



```
</s:event>
```

This ensures that if either a b:step element has a b:state attribute with the value 'selected' or it is the first b:step element within its parent container, then it will be selected at start up. The select handler makes sure that the selected step is shown and the other steps are hidden.

```
<s:event b:on="select">
  <s:task b:action="show"/>
  <s:task b:action="hide"
          b:target="preceding-sibling::b:step | following-sibling::b:step"/>
  <s:if b:test="position() = last()">
    <s:task b:action="copy"
            b:source="id('step-overview')/*[not(@b:exclude-from-summary)]"
            b:destination="." b:mode="aslastchild"/>
  </s:if>
</s:event>
```

Each step has a 'next' button at the bottom of it, which triggers the next event on the step.

```
<s:event b:on="next">
  <s:task b:action="trigger" b:target="* | */* | */*/*" b:event="validate" />
  <s:if b:test="not(.//*[@b:valid][@b:valid != 'true'])">
    <s:choose>
      <s:when b:test="following-sibling::b:step">
        <s:task b:action="set" b:attribute="b:step" b:target="ancestor::b:panelset[1]"
                b:value="{@b:step}"/>
        <s:task b:action="trigger" b:target="* | */* | */*/* | */*/*/*"
                b:event="step-validated" />
        <s:task b:action="select" b:target="following-sibling::b:step[1]" />
      </s:when>
      <s:otherwise>
        <s:task b:action="submit" b:target="ancestor::form[1]"/>
      </s:otherwise>
    </s:choose>
  </s:if>
</s:event>
```

This is a rather complicated event handler that does a number of things. Firstly it triggers a validate event on all of the children of the current step. Then if all of children are valid, it either shows the next step, or if this is the final step it submits the form. It also takes care copying the salient contents of each step to a little summary on the left hand side by triggering the step-validated event on the form-field behavior.

Form Submission

The final piece of the puzzle is the actual form submission. Submission of forms in BXML is somewhat different from normal HTML. It doesn't necessary lead to a whole page reload, as it would in a normal web page. In most cases the information is submitted to the server and then part of the page gets replaced by the response file. If you look at the form tag in form.html you will see several BXML attributes:

```
<form action="response.xml" b:target="." method="post" b:behavior="form">
```

As you can see the action attribute points to response.xml that is a BXML file which is the response file. The target that will be replaced by this response file is the form element itself. This is indicated by the b:target attribute. The form has a behavior called form. This behavior takes care of the validation and the submission.

```
<s:event b:on="submit">
  <s:choose>
    <s:when b:test=".//b:step">
```

```
      <s:task b:action="send" />
    </s:when>
    <s:otherwise>
      <s:task b:action="trigger" b:target=".//*" b:event="validate" />
      <s:choose>
        <s:when b:test="not(.//*[@b:valid][@b:valid != 'true'])">
          <s:task b:action="alert" b:value="valid"/>
          <s:task b:action="send" />
        </s:when>
        <s:otherwise>
          <s:task b:action="alert" b:value="invalid"/>
        </s:otherwise>
      </s:choose>
    </s:otherwise>
  </s:choose>
</s:event>
```

The submit event handler, does a few things. Firstly it checks whether there are any b:step elements. If this is the case, then it assumes that these steps will have taken care of validating all of the fields. All it then does is execute the send task, which takes care of the final submission of the form. If there are no b:step elements then it triggers a validate event and only if the entire form is valid does it issue a send task to submit the form.

This in short provides a basic model for client side validation of forms. You may use this and extend this in any of your own BXML applications. This document is in no way meant to be complete or to cover every aspect of form handling or the rest of BXML. You are advised to look through the code of Form Starter Kit yourself carefully and examine the rest of the BXML documentation where necessary.