WebSphere software

WebSphere Process Server 7.0

Customizing HTML-Dojo Forms for Human Tasks in Business Space

Michael Friess, Thomas Rossmann, Andreas Schön
IBM Development Lab Böblingen, Germany

August 2010

Disclaimer

This document is subject to change without notification and will not comprehensively cover issues encountered in any customer situation.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS.

For updates or newer releases please contact the authors.

About the authors

This document was written by the WebSphere Business Process Management (BPM) team in Böblingen. The authors are:

Michael Friess

Senior Software Engineer

IBM Software Group, Application and Integration Middleware Software

WebSphere BPM Architecture, User Interface Clients

Thomas Rossmann

Software Developer

IBM Software Group, Application and Integration Middleware Software

WebSphere BPM Development, Human Task Editor and Client Generation

Andreas Schön

Software Developer

IBM Software Group, Application and Integration Middleware Software

WebSphere BPM Development, BPM Clients

We would like to thank the WebSphere Business Process Management team for their contributions to this document.

Trademarks

Table of Contents

1. Abstract

WebSphere® Process Server includes Business Space powered by WebSphere, which provides an integrated user experience for business users. The Human Task Management widgets allow business users and managers to interact with business processes and human tasks.

The Task Information widget is used to initiate services, create, work on, view or check human tasks. The widget uses a task form specification that is associated with a human task for rendering the form and its content.

The user interface generators in WebSphere Integration Developer can be used to generate task forms based on either HTML with Dojo Toolkit or Lotus™ Forms. User interface developers often need to customize the generated HTML-Dojo forms for specific customer scenarios.

This paper describes various aspects of the generated HTML-Dojo form, such as its structure, how input and output messages are displayed, the embedded Cascading Style Sheets (CSS) information, and how input fields are handled. It then goes on to describe some typical ways in which you can customize the generated form, such as using alternative style sheets, integrating JavaScript™ code and custom widgets, and providing event handlers.

The paper concludes with a chapter about performance recommendations.

2. Introduction

WebSphere® Process Server includes Business Space powered by WebSphere, which provides an integrated user experience for business users across the IBM® WebSphere Business Process Management portfolio.

Business Space is a browser-based, graphical user interface based on widgets, which allows business users to interact with content and functions from different sources. They can create, share, and use business spaces that are composed of widgets and organized in pages.

WebSphere Process Server includes the Human Task Management widgets. These widgets allow business users and managers to interact with business processes and human tasks. They can be easily configured and combined with other widgets to create powerful business spaces.

Users initiate services and processes, create, work on, view, or check human tasks and initiated services using the Task Information widget. It renders the task form and provides information about the task, such as the priority, owner, status, and related tasks.

Human tasks are used to model human interaction in business processes. There are three kinds of human tasks:

- To-do tasks represent work for people that is initiated by a service invocation.
- Invocation tasks allow a user to initiate a service or process.
- Collaboration tasks allow the collaboration between people in a structured and controlled way.

A human task is modeled as a service component that implements a task done by a person. The interface of a human task is described using Web Service Definition Language (WSDL) with a single operation. The operation can have an input message, an output message, or both. These messages represent the content that the user can interact with.

WebSphere Integration Developer provides user interface (UI) generators to generate task forms for a specific client type. Task forms for use in Business Space can be generated based on HTML with the Dojo Toolkit, or Lotus Forms technology. The Task Information Widget renders the generated form so that users can work with the human task.

This paper covers the concepts of HTML-Dojo forms and how the generated form can be customized.

3. HTML-Dojo forms

An HTML-Dojo form is implemented as an HTML form with Dojo widgets for the input fields. "Dojo Toolkit is an open source modular JavaScript library (or more specifically JavaScript toolkit) designed to ease the rapid development of cross-platform, JavaScript/Ajax-based applications and web sites" [1]. See the Dojo Toolkit web site for more information [2].

In WebSphere Integration Developer, the UI generator for HTML-Dojo pages for use in Business Space generates a task form for a human task as an HTML file. The URL for the HTML file representing the task form is specified in the user interface settings for client type "Dojo" of the human task.

3.1 Structure of the generated HTML file

```
<html>
  <style>
        CSS defintion
  </style>
<div class="HTM_Taskname_id_version">
  <form class="view">
    <div class="inputMessage">
      fields for the input message
    </div>
    <div class="outputMessage">
      fields for the output message
    </div>
  </form>
</div>
</html>
```

The main parts of the file are:

- Cascading Style Sheet (CSS) definitions in the `<style>` element
- A `<div>` element with a unique class identifier for the generated file
- A single, nested `<form>` element that encapsulates the actual form

- Two `<div>` elements within the `<form>` element that contain the fields for the input and output messages of the human task. The messages are identified by the `inputMessage` and `outputMessage` class names.

3.2 Display mode

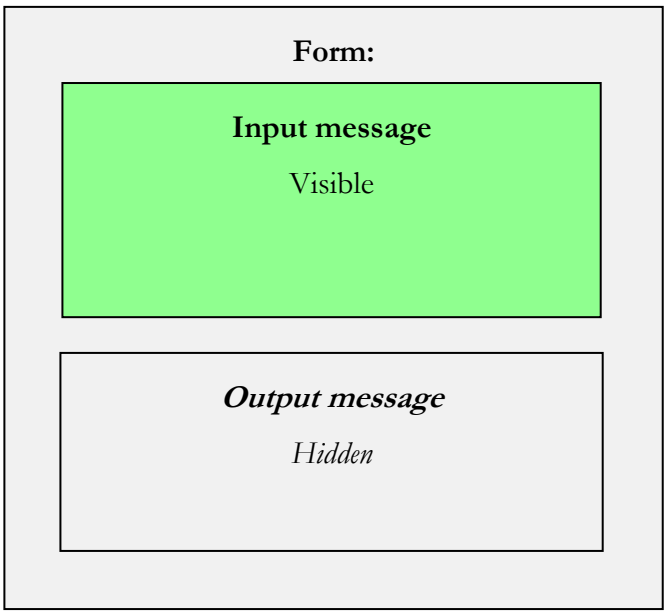
The `class` attribute of the `<form>` element defines the mode in which the form is displayed. The default is the view mode. The form element for the task form is realized by the `dijit.form.Form` widget.

The Task Information widget displays a task form in one of the following modes depending on the context in which the form is used:

- init mode

  This mode is used when a user fills out the form to initiate a service, process, or task (invocation and collaboration tasks).

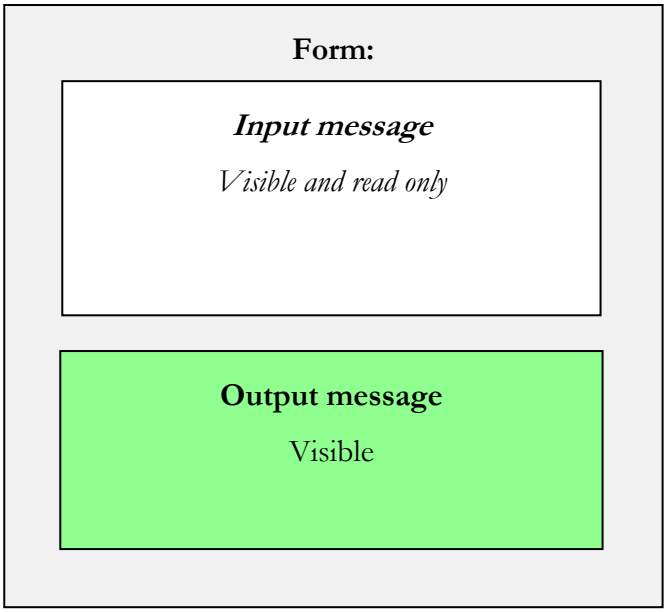
  Only the input message part of the form is displayed so that the data can be entered to start the task:



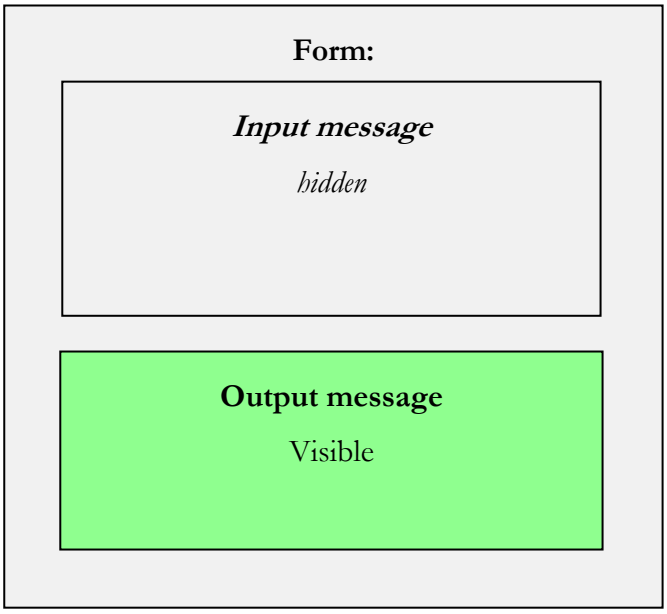
- edit mode

  This mode is used when a user works on, or edits a task (to-do or collaboration tasks).

  Both the input message and output message parts of the form are shown. The input message part is displayed in read-only mode. The output message part can be edited so that the user can enter the data to complete the task:



If the interfaces of the input message and the output message have the same business object type and not a simple type, in the generated HTML form the input message part is hidden, the output message part is displayed, and the fields are filled with the corresponding values from the input message.

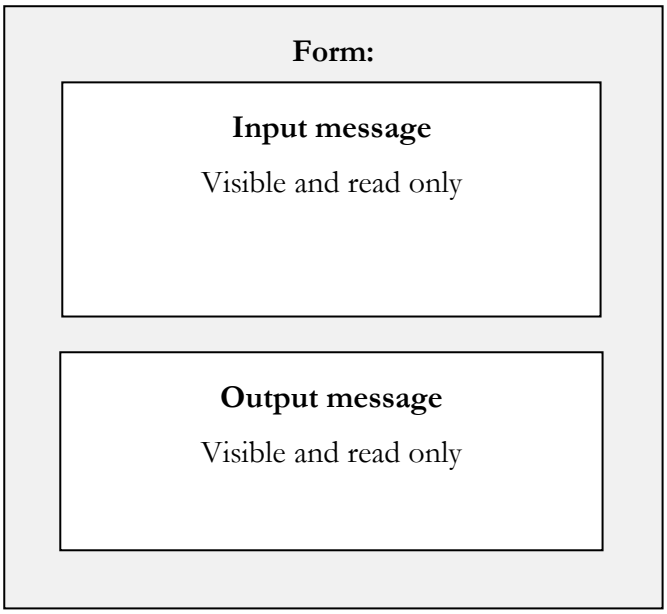


When the input and output messages do not have the same business object type, you can customize the generated task form so that the output message is filled with the values from the input message. See section 4.3 for more information on how to customize the form in this way.

- view mode

This mode is used when a user views the current task information (invocation, to-do, and collaboration tasks).

Both the input message and the output message parts of the form are shown. All the fields are read only:



3.3 Cascading style sheet information

The `class` attribute of the `<div>` element of the form specifies a unique CSS class for the generated task form.

```
<div class="HTM_CheckOrder_1556404392_7_0_0_v20091121_0200">
       <form dojoType="dijit.form.Form" class="view" action="">
       ...
       </form>
</div>
```

The name of the class is composed as follows:

- A static part 'HTM'
- The name of the human task, for example, 'CheckOrder'
- A unique identifier for the task, for example, '1556404392'
- The version of WebSphere Integration Developer used for generating the form, for example, '7_0_0'
- The build number of WebSphere Integration Developer used for generating the form, for example, ' v20091121_0200'

When the form is displayed in the Task Information widget, the layout is determined by the style definitions for the elements of the HTML-Dojo form. The following code sample shows part of the generated style definitions.

```
<style type="text/css">
    /**
```

```
     * General
     */
    .HTM_CheckOrder_1556404392_7_0_0_v20091121_0200 .fieldInput {
        width: 150px;
        text-align: left;
        display: inline;
    }

    .HTM_CheckOrder_1556404392_7_0_0_v20091121_0200 .fieldLabel {
        width: 150px;
        text-align: left;
         float:left;
    }
…
</style>
```

The CSS selectors of the style definitions correspond to the classes defined on the HTML elements of the form. The unique identifier is used as part of the selector in the style definitions so that they only apply to the HTML file where they are defined.

The following table describes the CSS styles defined in the header of the HTML file.

CSS style / scope	Description
`.fieldInput`	The style for an input field
`.fieldLabel`	The style for the label of an input field
`.arrayTemplate`	The style for the template element of an array
`.arrayLabelFont`	The style for the label for array elements
`span.addButton`	The style for the Add button, or the Add link of an array
`span.removeButton`	The style for the Remove button, or the Remove link of an array
`div.array_header_left`	The style used for the left part of an array header or an array element
`div.array_header_right`	The style used for the right part of an array header or an array element
`div.fieldContainer`	The style for a container of an input field and its corresponding label
`div.businessObject`	The style for a business object that is not an array
`div.arrayBO`	The style for the complete array area

CSS style / scope	Description
`div.arrayHeader`	The style for the array header area
`.arrayHeaderContainer`	The style for the container of an array header
`img.collapseStyle`	The style for the image which controls the collapsing and expanding of an array
`div.caseFolder`	The style for a case folder
`.init div.inputMessage`	Determines the visibility of the input message when a task is created
`.init div.outputMessage`	Determines the visibility of the output message when a task is created
`.edit div.inputMessage`	Determines the visibility of the input message when a task is worked on
`.edit div.outputMessage`	Determines the visibility of the output message when a task is worked on
`.view div.inputMessage`	Determines the visibility of the input message when a task is only viewed
`.view div.outputMessage`	Determines the visibility of the output message when a task is only viewed
`span.requiredMarker`	The style of the asterisk that marks a required field
`.fieldInput .requiredBackgroundColor`	Defines the background color of the required fields. To disable the background color, set 'background: none'

3.4 Input fields

The most important components of a form are the input fields. The Task Information widget allows several human tasks to be open simultaneously, which means that multiple instances of the same task form can be open. Therefore the generated input fields have an ID attribute that contains the HTMUniqueWidgetID special marker. This marker is replaced at runtime with a unique value for each task instance[1]:

```
<input
id="http___Test_ShipPublicTransportationTicketProcessInterfaceoperation1Request
_input5_HTMUniqueWidgetID"
dojoType="..."
… />
```

[1] Custom widgets also need to use this marker in their IDs to ensure unique IDs across the whole page.

Note that this marker is replaced only in the HTML file of the form, and not in the related artifacts, for example, JavaScript code. This behavior can be leveraged, for example, to reference an input field in a custom event handler.

In addition, the task ID of the opened task is also provided by the `{HTMTaskId}` special marker, which is replaced at runtime if the task ID has already been established.[2]

The input message and output message are based on the interface of the corresponding Web interface. The message type can be either a simple type or a business object type. A business object type can comprise simple types, other business object types, array types, and case folders.

3.4.1 Simple types

A form contains an input field for each simple type element that is part of the message type. The fields are `<div>` elements with a CSS class of `fieldContainer`, a `<label>` element for the input field, and a `<div>` element for the `<input>` element. The following example shows the structure of an input field:

```
<div class="fieldContainer">
        <label class="fieldLabel"
                  for="http___OrderSolution__HTMUniqueWidgetID">
                ArticleDescription
        </label>
        <div class="fieldInput">
                <input type="text"
                    name="/order/Order[]/ArticleDescription"
                        value=""
                        sdoPosition="0"
                        dojoType="dijit.form.ValidationTextBox"
                        regExp=".*"
                        id="http___OrderSolution__HTMUniqueWidgetID"
                        sdoMessageType="input"
                        sdoPrepopulation=""
                        required="false"
                        class="fieldInput"/>
        </div>
</div>
```

The values for the `for` attribute of the label and the `id` attribute of the input field must be the same so that the label can be correlated with the input field[3]. The CSS classes `fieldContainer`, `fieldLabel`, and `fieldInput` are used as the CSS style selectors for the style definitions. An input field has the following attributes:

[2] For example, if a task is about to be created and started, it does not yet have a task ID.

[3] The values are truncated in the code sample for readability.

- `type`
  Specifies the HTML type of the input element.
- `value`
  Specifies the value of the input element.
- `name`
  Contains an XPath-like expression. It is used to write and read the XML document of the message.
- `sdoPosition`
  Specifies the position of the corresponding XML element as part of an XSD construct, such as a sequence. This attribute is required to create a valid XML document and must not be modified.
- `dojoType`
  Specifies the Dojo widget used for the entry field.
- `regExp`
  Specifies a regular expression for the validation of the entry field as defined by the Dojo dijit.form.ValidationTextBox widget.
- `sdoMessageType`
  Specifies whether the field is part of the input message or the output message
- `sdoPrepopulation`
  Holds an XPath-like expression that references the input message that is used to prepopulate the input field in the output message.
- `required`
  Specifies whether the entry field must be provided to submit the task form. This Boolean attribute is generated for all the elements that are required by the XML schema of the business object in order to create a valid XML document. This attribute cannot be removed.

3.4.2 Business objects

A business object can contain simple types, single business objects, and arrays. If the type of the input or output message is a business object, a `<div>` element with a `businessObject` class is created. This element contains an `<h3>` element that serves as a header for the business object and the fields for the message as shown in the following code sample:

```
<div class="businessObject" sdoMinOccurence="1" name="/order" sdoMaxOccurence="1"
sdoPosition="0">
<h3 class="input">order</h3>
…
</div>
```

The attributes for input fields for business objects are the same as those for simple types. In addition, the following attributes are also generated for the division element:

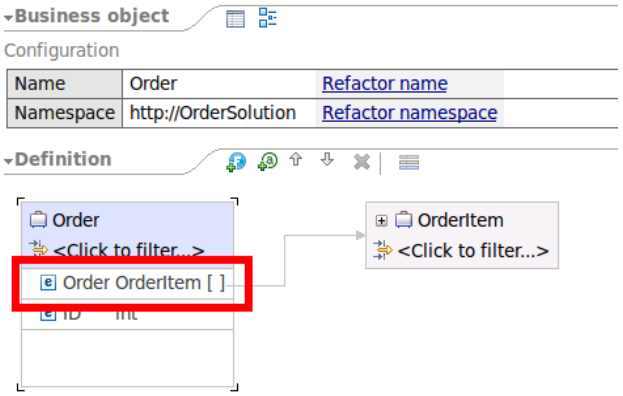
- `sdoMinOccurence`
  Defines the minimum number of occurrences of the business object.

- `sdoMaxOccurence`
  Defines the maximum number of occurrences of the business object.

This representation is generated only for a required business object, that is, the `sdoMinOccurence` and `sdoMaxOccurence` attributes both have the value of 1. This is always true if the business object is at the top level of the data structure. In all other cases, the business object is represented as an array.

3.4.3 Arrays

A schema definition can allow multiple occurrences of an element of the same simple type or business object. These are represented as arrays. For example, the `Order` property is an array of `OrderItem` objects:



The array representation provides a mechanism to add and remove elements and includes a template with the representation of the nested elements.

An array is represented by a `<div>` element with the `arrayBO` CSS `class` attribute:

```
<div name="/order/Order[]"
     class="arrayBO"
     dojoType="bpc.forms.FormArrayContainer"
     sdoMessageType="input"
     sdoMinOccurence="0"
     sdoMaxOccurence="-1"
     sdoPosition="0">
```

An array has the following attributes:

- `name`
  Indicates an array by the square brackets of the last element, for example, `Order[]`).

- `dojoType`
  Set to the `bpc.forms.FormArrayContainer` class. This class is part of the Task Information widget and provides the rendering for arrays.

- `sdoMessageType`
  Indicates whether the array is part of the input message or output message of the human task.

- `sdoMinOccurence`
  Defines the minimum number of occurrences of the element.

- `sdoMaxOccurence`
  Defines the maximum number of occurrences of the element. The value of `-1` indicates that there is no upper limit, this is equivalent to "unbounded" in an XML schema.

- `sdoPosition`
  Defines the position in the parent data structure.

The `<div>` element representing the array contains two `<div>` elements; one for the header of the array and one for the content template of the array.

- Array header
  The following code sample shows an example of an array header.

```
<div class="arrayHeaderContainer" dojoAttachPoint="arrayHeaderContainer">
     <img class="collapseStyle" dojoAttachPoint="collapseImage"></img>
     <div class="arrayHeader">
           <span class="arrayLabelFont"
                                  dojoAttachPoint="arrayHeaderLabel">
               Order
          </span>
          <div class="array_header_right">
               <span class="addButton">Add</span>
          </div>
     </div>
</div>
```

  The `<div>` element with the `arrayHeaderContainer` style class contains an image for collapsing and expanding the array, and the actual header containing the name of the business object, for example, "Order". The header also includes the add action to add a new element at the end of the array. The `dojoAttachPoint` attributes are required by the ArrayContainer widget.

- Content template
  The following code sample shows an example of a content template for an array:

```
<div dojoAttachPoint="ArrayTemplate" class="arrayTemplate">
     <div class="arrayHeaderContainer">
           <div class="array_header_left">
```

```
                    <span class="arrayLabelFont">Order</span>
                </div>
                <div class="array_header_right">
                    <span class="removeButton">Remove</span>
                </div>
            </div>
            <div class="fieldContainer">
                <label class="fieldLabel" for="http___input0_HTMUniqueWidgetID">
                    ArticleDescription
                </label>
                <div class="fieldInput">
                    <input type="text" name="/order/Order[]/ArticleDescription"
                        value="" sdoPosition="0"
                        dojoType="dijit.form.ValidationTextBox" regExp=".*"
                        id="http____input0_HTMUniqueWidgetID"
                     sdoMessageType="input"  sdoPrepopulation=""
                        required="false" class="fieldInput"/>
                </div>
            </div>
            <!-- more array template elements -->
        </div>
```

The element with the arrayTemplate style class holds the template information that the Task Information widget uses to create new elements. It is not visible itself, but is used as template for inserting new array elements.

The content template has a header part as a element with the arrayHeaderContainer style class. The header consists of a left part with the name of the business object as the header label, and a right part with an action to remove the array element. Then all the data fields follow. These fields can be simple types, business objects, or other arrays.

The value of the name attribute for a nested data element includes the XPath-like expression of the array it is contained in, for example:
order/order[ ]/ArticleDescription

The Task Information widget inserts a new array element by copying the "arrayTemplate" element, inserting the array index into the name attribute, and making this new element visible.

Note: an optional business object is displayed like an array of this business object type with a sdoMinOccurence of 0 and a sdoMaxOccurence of 1.

3.4.4 Case folder

The case handling support in WebSphere Process Server includes a collaboration scope and predefined data types for case folders. To use these predefined data types and the

interfaces for the collaboration scope, you must enable the resources for the collaboration scope in the dependency editor in WebSphere Integration Developer.

The Task Information widget provides specific handling for case folders. This is realized through the `com.ibm.bpc.widget.editor.form.CaseFolderContainer` Dojo widget. The generated HTML for the predefined case folder type sets the dojoType attribute accordingly. The following code sample shows an example of an input field for a case folder:

```
<div name="/input1/attachment[]"
     class="arrayBO_input"
     userId="{HTMUserId}"
     dojoType="com.ibm.bpc.widget.editor.form.CaseFolderContainer"
     sdoMessageType="input"
     sdoPrepopulation=""
     sdoMinOccurence="0"
     sdoMaxOccurence="-1">
</div>
```

3.5 Mapping form values to business objects

The Task Information widget maps the form values to XML documents of the input and output message. It reads the form values from the XML document of the input and output message. It also constructs the XML document for the input (init mode) or output (edit mode) message from the form content.

The `name` attribute is used like an XPath expression to read the data from the XML document. The `name` and `sdoPosition` attributes are used to construct the respective elements in the XML document. For example, an address business object is defined with the city, zip code, and street attributes. The generated form contains three `<input>` elements with the following `name` and `sdoPosition` attributes:

```
<input name="/address/city" sdoPosition="2"/>
<input name="/address/zipcode" sdoPosition="3"/>
<input name="/address/street" sdoPosition="1"/>
```

A user now enters "1234 Main Street", "New York", "11000" for the various input fields. The Task Information widget uses the `name` and `sdoPosition` attributes and the values of the input fields to generate the XML fragment:

```
<address>
        <street>1234 Main Street</street>
        <city>New York</city>
        <zipcode>11000</zipcode>
</address>
```

4. Common customization tasks

While the customization needs for specific scenarios can vary, there are some typical ways in which you can customize the generated forms.

4.1 Changing the labels and the order of generated fields

Every input field in the input and the output message is wrapped in a `<div>` element with the `fieldContainer` class, which contains the label for the field and the input control.

```
<div class="fieldContainer">
        <label class="fieldLabel" for="..._input0_...">
                firstName
        </label>
        <div class="fieldInput">
                <input id="..._input0_..." class="fieldInput" sdoPosition="0" … />
        </div>
</div>
```

You can directly change the label, but you must not change, or remove the attributes of the input field. For example, the `sdoPosition` attribute is needed to construct the message with this element in the correct order in the containing sequence.

You can also rearrange the input fields in the form by rearranging the field containers. Field containers are embedded in the `<div>` element of the corresponding input or output message as described in section 3.1 Structure of the generated HTML file. The field containers are used to construct the input and output message.

```
<form dojoType="dijit.form.Form" class="view" ...>
        <div class="inputMessage">
                <!-- Only rearrange within this div -->
        </div>
        <div class="outputMessage>
                <!-- Only rearrange within this div -->
        </div>
</form>
```

Make sure you do not change the overall structure of the document and only rearrange field containers within the blocks for the input message or output message.

In addition, fields can be part of a business object hierarchy or an array, which means that the field containers are nested in `<div>` elements with the `businessObject` or `arrayTemplate` class, for example:

```
<div class="businessObject" name="/input1/customer" … >
        <!-- Only rearrange within this div -->
</div>
```

The field container must remain within the same `<div>` element of the business object or the array.

4.2 Hiding generated fields

The task form provides input controls for the various fields of the input message and the output message. Depending on the mode of operation these controls are shown, hidden, or disabled.

To hide an input field, add a style definition that hides the field container. For example:

```
<div class="fieldContainer" style="display: none">
        <label class="fieldLabel"
                  for="http___OrderSolution__HTMUniqueWidgetID">
                ArticleDescription
        </label>
        <div class="fieldInput">
                <input type="text"  … />
        </div>
</div>
```

Alternatively, you can use a CSS style class:

```
<div class="fieldContainer inputArticleDescriptionField">
        …
</div>
```

The style class allows you to hide the field depending on the display mode. For example, to hide the field only when starting the task, you would create a CSS selector that refers to the init mode:

```
.HTM_CheckOrder_1556404392_7_0_0_v20091121_0200 .init .inputArticleDescriptionField {
        display: none
}
```

4.3 Prepopulating the output message

To initially set a field in the output message to the value of a field in the input message, you can instruct the Task Information widget to prepopulate the field.

When the input and output messages have the same type, all the fields in the output message in the generated HTML-Dojo form are set for prepopulation. The CSS styles are defined so that the input message is hidden in the edit mode.

In the following example, selectors are used to define the visual attributes of input and output messages. The selector comprises a reference to the CSS class for the mode of operation (`.edit` in the example), and a reference to the input or output message.

```
/**
* CSS Styles for editing a task. Since the input message equals the
* output message, only the output message is shown.
*/
.HTM_ArrayTask_1985205702_7_0_0_v20091130_1326 .edit div.inputMessage {
        visibility: collapse;
```

```
        display:none;
}
.HTM_ArrayTask_1985205702_7_0_0_v20091130_1326 .edit div.outputMessage {
}
```

If the input and output message types do not match, there might still be properties that are used in both the input and the output messages. To enable prepopulation for this case, set the `sdoPrepopulation` attribute of the input field in the output message to the XPath expression used for the `name` attribute of the corresponding input field in the input message. For example:

```
<form dojoType="dijit.form.Form" class="edit" ...>
        <div class="inputMessage">
                <input  name="/input1/a/int" sdoMessageType="input"  … />
        </div>
        <div class="outputMessage>
                <!-- Field in the output message that refers to a field in the input
                     message through the sdoPrepopulation attribute.
                -->
                <input name="/output1/b/int" sdoMessageType="output"
                        sdoPrepopulation="/input1/a/int" … />
        </div>
</form>
```

Make sure that the types of the elements in the input and output messages are the same.

4.4 Using other style sheets

You can include a different style sheet by adding either a `link` element in the `<head>` section, or an `@import` instruction in the `<style>` element of the generated form:

```
<head>
        …
        <!-- link tag in head section -->
        <link rel="stylesheet" href="style.css" TYPE="text/css" ></link>
        ...

        <style type="text/css">
        <!-- alternatively you may add @import instructions into the style section -->
        @import "style.css";
        ...
        </style>
</head>
```

You typically locate the style sheets together with the HTML file for the task form and use a relative URL to refer to the style sheet.

4.5 Integrating custom logic

Customizing the task form often involves implementing custom logic, for example, as simple JavaScript functions, reusing other Dojo widgets, or providing custom Dojo widgets.

4.5.1 Integrating custom JavaScript functions

You must put custom JavaScript functions in a Dojo module. JavaScript code that is placed in a script element within the form is ignored.

A Dojo module is a JavaScript file with the same name as the module name and located in a directory that matches the package name. For example, the definition of the following Dojo module should be stored in a `form.js` file in the `company` directory:

```
dojo.provide("company.form");

// The Task Information widget does not set the char-set attribute.
// Therefore the script is processed in the encoding of the Business Space which is utf-8
//
// Calculate the number of the next year.
//
company.form.setDojoComponentToNextYear = function(id) {
        var comp = dijit.byId(id);
        var year = (new Date()).getFullYear() + 1;
        if (comp != null) {
                        comp.attr("value",year);
        }
}
```

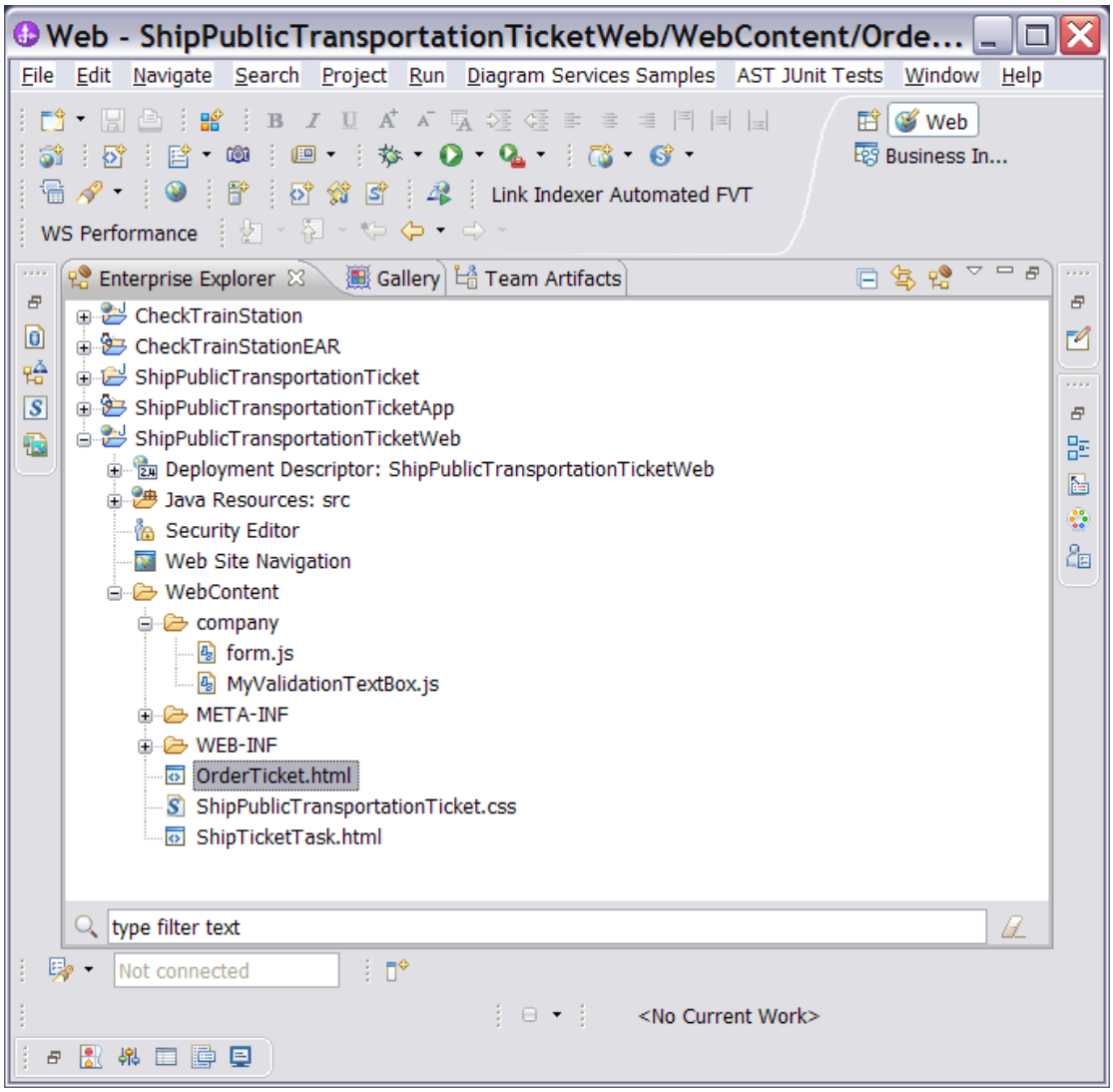
A Dojo module always starts with a `dojo.provide` statement that declares the name of the module. For more information on the Dojo Toolkit, see the Dojo documentation ([2]).

To include a Dojo module in an HTML-Dojo form, add a script element that defines the location of the package, and the name of the module. Be aware that the script element is parsed and only `dojo.registerModulePath` and `dojo.require` statements are recognized.

```
<script type="text/javascript">
            dojo.registerModulePath("company","company");
            dojo.require("company.form");
 </script>
```

The Task Information widget supports only relative paths. It therefore converts the path in the `dojo.registerModulePath(package,path)` statement so that it is relative to the location of the task form. Note that `dojo.registerModulePath` in the Dojo Toolkit converts a relative path so that it is relative to the location of the dojo.js file.

The script tag in the example ensures that the company.form module is loaded by the `OrderTicket.html` form. Therefore the following file structure is required:



4.5.2 Reusing other widgets from the Dojo Toolkit

The generated task form implements input fields using either widgets from the Dojo Toolkit or widgets provided with the Task Information widget. The `dojoType` attribute identifies the Dojo widget that is used for the input field as shown in the following example:

```
<form dojoType="dijit.form.Form" class="view" action="">
        <div class=”inputMessage”>
        …
        <!-- example of an input element in the form -->
        <input type="text" name="/input1/year" dojoType="dijit.form.NumberTextBox"…/>
        </div>
        <div class=”outputMessage”>
        …
        </div>
```

```
</form>
```

You can replace the implementation with other widgets from the `dijit.form` package of the Dojo Toolkit by adding a `dojo.require` statement for the Dojo widget module, for example:

```
<script type="text/javascript">
       dojo.require("dijit.form.NumberSpinner");
</script>
```

Set the `dojoType` attribute of the input field to the Dojo widget name:

```
<input type="text" name="/input1/year"  dojoType="dijit.form.NumberSpinner".../>
```

You can add additional attributes as defined by the Dojo widget.

4.5.3 Using custom Dojo widgets

In general, you derive your custom Dojo widget from the widget in the generated form that you intend to replace. This ensures that the interaction between the input widget and the Task Information Widget works as expected. For example, if the widget implements an `isValid` method, this method is used to validate the input from the widget and enable the Submit button only if all the fields are valid.

You need to provide a module that contains your custom Dojo widget. For example, you create a custom widget, `company.MyValidationTextBox,` which extends the `dijit.form.ValidationTextBox` widget that it replaces. Your custom widget must be in the `MyValidationTextBox.js` file of the `company` directory. The following code sample shows how you might implement this example:

```
<script type="text/javascript">
       dojo.registerModulePath("company","company");
       dojo.require("company.MyValidationTextBox");

</script>


dojo.provide("company.MyValidationTextBox");
dojo.require("dijit.form.ValidationTextBox");
dojo.declare("company.MyValidationTextBox",dijit.form.ValidationTextBox,
{
       startup: function() {
            this.inherited(arguments);
             var year = (new Date()).getFullYear() + 1;
             this.attr("value",year);
       }
});
```

4.6 Providing event handlers

Event handlers are an essential aspect of developing user interfaces with JavaScript. They are functions that are invoked when a particular event occurs.

4.6.1 Defining a startup handler

The form element for the task form is implemented using the `dijit.form.Form` widget. When the form is opened, the widget triggers a `startup` event after it and all its children are instantiated. You can use this event to run initialization logic in a similar way to the `onload` event for the window in an HTML/JavaScript page.

Use a script element of type `dojo/connect` to declare a Dojo event handler:

```
<form dojoType="dijit.form.Form" class="view" action="">
        <script type="dojo/connect" event="startup">
        dijit.byId('http___Test_ShipPublicTransportationTicket
        ProcessInterfaceoperation1Request_input5_HTMUniqueWidgetID').attr('value',
        (new Date()).getFullYear() + 1);
        </script>
…
</form>
```

Because the event handler is specified declaratively, the ID provided in the `dijit.byId` call is unique and it references the input field in the same task form instance.[4]

Another way to locate widgets is to provide a unique CSS class for the element containing the widget, and use the `dojo.query` method within a given scope.

4.6.2 Using an event handler for input fields

You define an event handler for an input field declaratively. However, you cannot define an event handler declaratively on an HTML input element[5]:

```
<input type="text" name="/input1/endPoint" dojoType="dijit.form.ValidationTextBox"
.../>
```

To define a Dojo event handler, replace the `<input>` element by a `<div>` element as shown in the following example:

```
<div type="text" name="/input1/endPoint" dojoType="dijit.form.ValidationTextBox"...>
        <!-- The following event handler will be called when field loses the focus -->
        <script type="dojo/connect" event="onBlur">
                <!--  Only the unique widget id is provided that allows to
                      refer to all other widgets in this instance of the form.
                      This way the amount of logic in the form is kept to a minimum.
                -->
                company.form.updateProductCode("HTMUniqueWidgetID");
```

[4] See 3.4 for more information about the HTMUniqueWidgetID marker.

[5] A Dojo event handler can only be defined in the markup of elements that support an `innerHTML` property, which is not the case for the `input` element.

```
        </script>
 </div>
```

Dojo replaces the element during the parsing of the form.

As explained in section 4.5.1 on using JavaScript code, the called method must be part of the Dojo module that is registered in a <script> element. If other fields are updated, it is important to provide the unique ID of the current task (See 3.4).

The following code sample shows an example of a method that is part of a Dojo module:

```
company.form.updateProductCode = function(htmUniqueWidgetId) {
    var compartmentWidget = …;
    var yearWidget =...;
    var startingPointWidget =...;
    var endPointWidget = dijit.byId("http___Test_ShipPublicTransportationTicket
                        ProcessInterfaceoperation1Request_input7_" + htmUniqueWidgetId);
    var productCodeWidget =...;
    // Set the computed value on the productCodeWidget.
    productCodeWidget.attr("value",
        yearWidget.attr("value") +
        "#" +
        compartmentWidget.attr("value") +
        "#" +
        startingPointWidget.attr("value") +
        "#" +
        endPointWidget.attr("value"));
}
```

4.7 Initializing field values

Often form fields must be set to an initial value. The initial value can be static or calculated dynamically.

4.7.1 Static initialization

You can set the value attribute on the input element to its initial value.

```
<input type="text" name="/input1/date" value="2010-10-31"
dojoType="dijit.form.DateTextBox" … />
```

Make sure that the format of the literal that is set to the value attribute complies with the programming model of the component that you are using. For example, for an xsd:date field, a dijit.form.DateTextBox widget is used that expects the value according to ISO 8601 / RFC 3339.[3] [7]

4.7.2 Dynamic initialization

The initial value of a form field might need to be determined when the form is opened, for example, to set a date field to the current date.

You can use a startup handler to implement the dynamic initialization as shown in 4.6.1 Defining a startup handler.

When the calculation is more complex, you might want to implement the initialization in a separate function, and call the function in the event handler as shown in the following example:

```
<form dojoType="dijit.form.Form" class="view" action="">
        <script type="dojo/connect" event="startup">
        company.form.setDojoComponentToNextYear('http___Test_ShipPublicTransportationTicketProcessInterfaceoperation1Request_input5_HTMUniqueWidgetID');
        </script>
…
</form>
```

In this example, the Dojo event handler `(type="dojo/connect")` calls the `setDojoComponentToNextYear` function that is defined in the `company.form` module.

```
company.form.setDojoComponentToNextYear = function(id) {
        var comp = dijit.byId(id);
        var year = (new Date()).getFullYear() + 1;
        if (comp != null) {
                comp.attr("value",year);
        }
}
```

The function looks up the Dojo component, and sets the value to the next year. Note that the dijit ID is passed as an argument, and the literal is used only in the HTML file of the task form to make sure that the `HTMUniqueWidgetID` is properly replaced.

4.8 Calling a REST service

You can call a REST service by using the dojo.xhrGet functionality (See the Dojo documentation for more details [2]). If the host of the REST service is different to the host of the Business Space, the call needs to be directed to the Business Space proxy using the following URL:

`/mum/proxy/http/<host>:<port>/...`

where `/mum` is the context root of the Business Space, and `/proxy` is the context path setting in the proxy-config.xml file.

Be aware that you might want to restrict access to the proxy in a production environment using:

`<profile>\BusinessSpace\<node>\<server>\mm.runtime.prof\config\proxy-config.xml`

For more information about using the proxy especially in more complex environments, see [6].

4.9 Localizing HTML-Dojo forms

The Task Information Widget retrieves the HTML-Dojo form using an AJAX call for the URL that was set in WebSphere Integration Developer as the user interface setting

for the `Dojo` client type. Therefore the form can also be served using a servlet or a JavaServerPage so that you can check the locale, and serve the correct localized version.

5. Performance Recommendations

Most performance problems with HTML-Dojo forms are due to the size of the HTML-Dojo forms. These problems can be avoided by designing your processes and tasks with that in mind.

The infrastructure is also an important factor. For example, there are huge differences in the performance between browsers that have been designed for Rich Internet Applications (RIA) and older versions which have been released before the advent of Rich Internet Applications. (See [8], [9],[10])

5.1 Design Guidelines for Processes and Tasks

The size of the HTML-Dojo forms depends mainly on the complexity of the interface of a human task. You can greatly reduce the complexity of a human task by splitting the single human task into a page flow of multiple human tasks. In addition to that a page flow may also help users to better understand the process. (See [9])

5.2 Infrastructure considerations

You should consider to use the most recent version of your favorite browser that is supported by Business Space. For example, Internet Explorer 8 which has been released in 2009 is much better than Internet Explorer 6 whose first version has been released in 2001. Today Rich Internet Applications like Business Space that heavily use JavaScript are common place while at the start of the decade such application did not yet exist. (For more details see [8])

5.3 Important fixes

The internal processing of the HTML-Dojo forms has been improved with all service packs. As of this writing service pack 7.0.0.3 is the latest service pack and includes important performance improvements.

For example, APAR IZ87329 greatly improves the performance for processing the input and output messages in the Task Information widget. Furthermore, it provides the ability to disable displaying the task input message which reduces the form's size resulting in better response times. (For more details see [9])

6. References

1. "Dojo Toolkit". Wikipedia. 2010-06-10. http://en.wikipedia.org/wiki/Dojo_Toolkit.
2. Dojo toolkit. The Dojo Foundation. http://www.dojotoolkit.org/
3. "ISO 8601: 2004". International Organization for Standardization. 2008-03-18. http://www.iso.org/iso/catalogue_detail?csnumber=40874
4. "Customizing HTML-Dojo forms for Business Space". Klaus Schaefers, Marius Kreis. 2009-04-09. http://www.ibm.com/developerworks/websphere/techjournal/0810_kreis/0810_kreis.html
5. Business Space Information Center. International Business Machines Corporation. http://publib.boulder.ibm.com/infocenter/dmndhelp/v7r0mx/index.jsp?topic=/com.ibm.bspace.ic.main.doc/welcome/bspace_welcome.html
6. "Configuring the HTTP proxy for AJAX applications: Mashup Center 2.0". International Business Machines Corporation. 2009-12-31. http://www-10.lotus.com/ldd/mashupswiki.nsf/dx/Configuring_the_HTTP_proxy_for_AJAX_applications__Mashup_Center_2.0
7. http://www.dojotoolkit.org/reference-guide/dijit/form/DateTextBox.html#standard-date-format
8. Scalability and Performance of Business Space and Human Task Management Widgets in WebSphere Process Server v7. http://www-304.ibm.com/support/docview.wss?uid=swg27020684
9. Slow rendering of complex HTML+Dojo human task forms in Task Information widget (Business Space) http://www-01.ibm.com/support/docview.wss?uid=swg21468931
10. JavaScript benchmark results for several browser http://ie.microsoft.com/testdrive/benchmarks/sunspider/default.html